Логическо програмиране

Антон Зиновиев

18 януари 2017 г.

Съдържание

Глава 1

Математическа логика и програмиране

1.1. Безсмисленото начало

> Каква е ползата от Вашите красиви изследвания за π? Защо да се занимаваме с такива неща при положение, че ирационалните числа не съществуват?
>
> Леополд Кронекер

В математиката винаги явно или неявно са се използвали множества, но в продължение на дълго време множествата били образувани по сравнително прост начин. Имало е множества от реални числа, множества от функции между реални числа, множества от точки и т.н. и винаги е имало ясно и разбираемо правило, определящо кои точно са елементите на дадено множество. Математиците не се увличали по измамни занимания с далечни от ума обекти като множества, които съдържат всички подмножества на множеството, което съдържа всички подмножества на множеството, което съдържа всички подмножества на множеството, което съдържа всички естествени числа.

През последната четвърт на 19-ти век обаче се появила съблазън, която разклатила здравомислието на математиците. Основна заслуга за това имал Георг Кантор, който неблагоразумно дръзнал да разработи основите на това, което днес се нарича *теория на множествата*. За

съжаление, оказало се, че заниманията с теория на множествата неизбежно изискват от математѝка да описва свойствата на обекти, които поради самия им характер е невъзможно човек да си ги представи ясно. Тогава малцина съзрели колко опасни могат да бъдат тези прелъстителни занимания, а с особена прозорливост сред тях се отличил Леополд Кронекер. Той не спирал доблестно да предупреждава всички, че Кантор е псевдоучен, шарлатанин, ренегат и развратител на младежта, но — уви — малцина се вслушали в гласа на разума. И наказанието, не се забавило...

Едно след друго в теория на множествата се появили противоречия. Така например Кантор доказал, че за всяко множество X множеството 2^X има повече елементи от X. Но какво правим когато X е множеството, което съдържа всички множества? Как така 2^X ще съдържа повече елементи от множеството, което съдържа всички множества? За да избегне това противоречие, Кантор решил, че множеството, което съдържа всички множества, е твърде голямо и затова не можело да бъде множество. Друг парадокс е свързан с т.н. ординални числа. Оказало се, че от всяко ординално число има по-голямо, а в същото време множеството от всички ординални числа трябва да бъде най-голямото ординално число. И в този случай Кантор решил, че множеството от всички ординални числа е твърде голямо и затова не можело да бъде множество. Друго известно противоречие е парадоксът на Ръсел.[*] За да бъде преодоляно то, през 1908 Цермело измислил аксиоми, според които множествата се строят на безкрайна трансфинитна редица от стъпки като всяко множество може да съдържа само елементи, които вече са били образувани на някоя предшестваща стъпка. По този начин математиката започнала да прилича на съвременна компютърна програма, която е пълна с грешки и която след всяка новооткрита грешка трябва да се оправя.[**]

Кронекер починал, но призивът за конструктивизъм в математиката бил подет от други математици, напр. Анри Поанкаре. Особено активен бил Брауер, който измислил нов вид конструктивна математика, която нарекъл *интуиционизъм*.[***] Брауер установил, че ако искаме

[*] Обичайните множества не съдържат себе си като елементи. Да наречем такива множества *нормални* и нека R е множеството, което съдържа всички нормални множества. Самото R нормално множество ли е?

[**] Все още не е открито противоречие в аксиомите на Цермело.

[***] Името „интуиционизъм“ идва от това, че според Брауер математиката трябва да включва само конструкции, които можем да си ги представим ясно. Благодарение на това математикът може да установява правилността на една конструкция посредством просто позоваване на интуицията си.

да разработваме математиката конструктивно, е удобно да използваме логика, която е различна от обичайната. Така например в интуиционизма не е вярно, че всяко съждение е вярно или невярно, но въпреки това няма съждение, което едновременно не е вярно и не е невярно.

С огорчение Давид Хилберт гледал как дори собствените му ученици започвали да се увличат от идеите на интуиционизма. А когато към интуиционистите се присъединил и Херман Вейл, Хилберт решил да се намеси. Той заклеймил интуиционистите като метежници, организирали идеологическа чистка, които без да разполагат с доказателство за виновност унищожавали всичко, за което решат, че буди подозрения. А конкретно за Брауер той писал: „Аз съм безкрайно удивен от факта, че дори и сред математиците се оказва, че силата на внушението на един единствен човек, без значение колко темпераментен или остроумен, е в състояние да предизвика толкова невероятни и чудати въздействия.“

През 1922 г. Хилберт обявил своята знаменита програма за „спасяване“ на математиката. Тя се състои в следното:

(1). Дефиниране на формално точен математически език, на който може да се формулират математическите съждения. Така формулираните математически съждения се наричат *формули*.

(2). Формулиране на точно определени правила, посредством които от вече доказани формули можем да получаваме нови доказани формули.

(3). Доказателство, че така формулираните правила са достатъчни, за да се докажат всички математически верни съждения.

(4). Доказателство, че с така формулираните правила не може да се стига до противоречие. В това доказателство Хилберт иска да се използват само методи, които биха били убедителни дори и за Брауер.

(5). Доказателство, че когато използваме в доказателствата сложни и неконструктивни обекти, ще можем да правим по-кратки доказателства, но няма да може да се докажат неща, които не могат да се докажат и без използването на неконструктивни обекти.

(6). Намиране на метод, посредством който може да се решава дали едно математическо съждение е вярно.

Отначало нещата потръгнали добре. Още през 1879 г. Готлоб Фреге бил дефинирал логически език, който е еквивалентен по изразителна сила на съвременната предикатна логика от първи ред и на който може да се формулира цялата математика. Освен това Фреге формулирал и точни правила, посредством които от вече доказани формули може да

се получават нови доказани правила. С това първата и втората точка от плана на Хилберт можело да се считат за изпълнени.

Някои значителни успехи били постигнати и по отношение на останалите точки от плана. Например през 1929 г. Курт Гьодел доказал, че правилата на Фреге са достатъчни, за да се докаже с тях всяка логическа истина. Много на брой и извънредно важни резултати били получени в продължение само на няколко години. Това било окуражаващо и вдъхновяващо и изглеждало, че аха-аха, още съвсем малко и програмата на Хилберт ще бъде изпълнена.

Но през 1931 г. се случило немислимото — Гьодел доказал своите знаменити теореми за непълнота. От тях следвало, че:

- ако една математическа теория включва в себе си аритметиката,[*] то е невъзможно с краен брой формални правила и аксиоми да се докажат всички верни неща в нея;
- ако една математическа теория включва в себе си аритметиката, то тогава нейната непротиворечивост не може да се докаже, използвайки само методи, които се включват в тази теория.

Това означава, че по колкото и хитър начин да формулираме аксиомите, винаги ще има верни математически съждения, които не следват от тях. И също така означава, че ако решим да се ограничим само с онези съждения, които следват от аксиомите, няма да може да докажем, че от избраните аксиоми не може да се получи противоречие. Малко по-късно (през 1936 г.) Чърч и Тюринг показали, че и последната точка от плана на Хилберт е неосъществима — няма алгоритмичен метод, посредством който можем да познаваме дали едно математическо съждение е вярно или не. Оригиналният план на Хилберт се оказал неосъществим...[**]

В резултат от усилията да се изпълни програмата на Хилберт се появил един нов раздел в математиката — *математическата логика*. Този нов раздел обаче не успял да осъществи това, заради което бил създаден. Той не успял да подсигури основите на математиката, а обикновените математици продължили да си правят математика без

[*] По точно, ако в нея може да се дефинира какво значи събиране и умножение на естествени числа.

[**] Всъщност през 1936 г. Герхард Генцен успял да докаже по задоволителен начин непротиворечивостта на аритметиката. Няколко десетилетия по-късно Гаиши Такеучи доказал непротиворечивостта на различни теории, които са достатъчно богати, за да обхванат по-голямата част от съвременната математика. Засега обаче не знаем дори как да подходим към проблема за непротиворечивостта на теорията на множествата на Цермело, а тя все още се счита за основа на съвременната математика.

много-много да се интересуват какво се случва в математическата логика. Изобщо математическата логика била абстрактна математическа дисциплина, която се оказала толкова безполезна за каквото и да е, че според Мартин Давис, когато той бил студент (1944 – 1950 г.), дори тополозите* се присмивали на логиците, че витаели някъде далеч в космическото пространство.

Но това съвсем скоро щяло да се промени...

1.2. Денят не си личи по заранта

> Самата конструкция е изкуството,
> а приложението ѝ в света — зъл паразит.
>
> Лойтцен Егбертъс Ян Брауер

Въпреки че приложенията на математическата логика в останалите дялове на математиката все още са като цяло незначителни,** приложенията ѝ в областта на информатиката и компютрите са толкова много и нерядко неочаквани, че човек неволно си задава въпроса защо това е така. Наистина, и други дялове на математиката имат компютърни приложения, напр. линейната алгебра, геометрията, теория на числата, числените методи, теория на вероятностите, теория на графите и комбинаториката. На нито един от тях обаче приложенията не са толкова разнообразни и всеобхватни, колкото приложенията на математическата логика.

Традиционно математическата логика се разделя на четири поддяла. Всеки един от тях има важни „компютърни“ приложения. Тези поддялове са:

*Топологията е полезна математическа дисциплина, в която заедно с полезните неща има и има страшно много теореми, които вероятно никога няма да получат каквото и да е практическо приложение.

**Разбира се те са незначителни само на фона на големите очаквания. Ето някои немаловажни математически приложения на математическата логика. Теория на моделите има някои приложения в алгебрата, например доказателството на основната теорема на алгебрата (това, че всяко поле има алгебрически затворено разширение) се опростява, ако се използва логическата теорема за компактност. В математическия анализ обосноваването на инфинитезималното смятане на Лайбниц с безкрайно малки величини може да се счита за едно от най-важните постижения на математиката на 20-ти век. Много от резултатите в топологията пък са тясно свързани с дълбоки резултати в теория на множествата, а основните приложения на т.н. *дескриптивна теория на множествата* са във функционалния анализ, ергодичната теория и операторната алгебра.

– обща логика и теория на моделите;
– теория на множествата;
– теория на изчислимостта;
– теория на доказателствата и конструктивна математика.

Обща логика и теория на моделите

Машините, които хората ползвали от праисторическо време, променяли свойствата на различни материални обекти. Например пещта на грънчаря преобразувала неопечените глинени съдове в годна за използване керамика, каруцата на търговеца променяла географските координати на натоварените материални ценности, така че стойността им на новите координати да бъде по-голяма и т.н. Устройствата, които работели с нематериални обекти, т.е. данни, били малко на брой и сравнително прости.

През 1642 г. Блез Паскал изобретил първата механична сметачна машина. Сметачните машини били първите машини, които можели да преобразуват данни, но както при всички останали машини от онова време, действията, които те извършвали, се контролирали непосредствено от човек.

През 1805 г. Жозеф Мари Жакард изобретил тъкачен стан, който можел да тъче платове с повтарящи се орнаменти. Машината можела да чете данните, описващи нужните орнаменти, от перфокарти. Това била първата машина, чиито операции не се управлявали непосредствено от човек, а зависели от програма, записана в паметта на машината.

След като тези две изобретения били комбинирани, се получили т.н. *програмируеми сметачни машини* — сметачни машини, които се управляват не непосредствено от човек, а от програма. Първият проект за програмируема сметачна машина била „Аналитичната машина“ на Чарлз Бебидж (1837 г.). Програми за тази машина са писани от Ада Лавлейс, поради което тя днес се счита за пръв програмист.

За съжаление работата на Бебидж останала малко известна и не е използвана от по-късните конструктори. Първата действително конструирана и работеща програмируема сметачна машина е „Колумбийският диференциален табулатор“, създаден почти един век по-късно (1930 г.) от фирмата Ай Би Ем. Вестниците, винаги радващи се на сензации, нарекли тази машина „суперкомпютърно устройство с умствените способности на 100 математика, дори при решаване на много сложни алгебрични задачи“.

Възможностите на програмируемите сметачни машини бързо се увеличавали и машината „Цет 3“, създадена от Конрад Цузе през 1941 г. била първото устройство, което като изключим по-малката му памет и скорост на работа, има на теория същите изчислителни възможности както и съвременните компютри.

За да получим *компютър*, оставало да се реализира само още един ключов елемент — за програмируемите сметачни машини програмите и обработваните от тях данни били две напълно различни неща. Те можели да четат програмите си, можели да четат и входните си данни, но можели да записват само данни. Това ограничение означава, че за тях не може да се пишат компилатори, защото генерираният от компилатора изпълним код представлява хем данни, хем програма. Първият компютър, при който не се прави разлика между данни и програми, е машината „Ес Ес И Си“ на Ай Би Ем (1948 г.), вторият — машината „ЕДСАК“ на Кеймбриджкия университет (1949 г.) и третият — машината „МЕСМ“ на Киевския институт по електротехнология (1950 г.).

Това, че в данните, подавани на един компютър, се съдържат и инструкциите за компютъра, означава, че имаме нужда от формален език, на който да се формулират тези инструкции. Математическата логика е разделът от математиката, който ни дава техники за дефиниране на синтаксиса и семантиката на различни формални езици и изследване на свойствата на тези езици. Формалните езици, които могат да бъдат полезни на практика, далеч не се ограничават само с различни видове езици за програмиране. Всеки формален език, който допуска ефективна обработка, обикновено се оказва или полезен, или много полезен.

* * *

Много и най-разнообразни езици са дефинирани в математическата логика, но ако трябва само един от тях да бъде наречен „Езикът на математическата логика“, то честта я има *предикатната логика от първи ред*. Този език притежава голяма изразителна сила — почти цялата съвременна математика може да бъде формулирана на него. Освен това той е и много удобен за използване — нерядко в математически статии твърденията се формулират, използвайки този език, а не на естествен език (български, английски и т.н.). А приложението на този език като език за заявки към бази данни представлява един от най-впечатляващите примери за компютърно приложение на математическата логика.

По принцип, една от пречките за ефективна обработка на предикатни формули е използването в тях на т.н. свързани променливи. Тази пречка била преодоляна още през 1948 г. от Алфред Тарски, който в желанието си да алгебризира предикатната логика, измислил *цилинд-*

ричните алгебри. Езикът на цилиндричните алгебри има същата изразителна сила като езика на предикатната логика с равенство,[*] но за разлика от него не притежава свързани променливи.

Оставало само да се види как идеите от цилиндричните алгебри можели да се реализират ефективно. Това направил Едгар Код през 1969 г., предлагайки *релационния модел за управление бази данни*. Код доказал, че релационният модел притежава същата изразителна сила, както и предикатната логика от първи ред. Освен това този модел допуска и изключително ефективна реализация — ако разполагахме с възможност за неограничена паралелизация, тогава заявките към една релационна база данни биха се изпълнявали за константно време без значение какъв е нейният размер. Съвсем скоро след публикацията на Код се появил и езикът ес кю ел, който реализира релационния модел и е най-популярен език за управление на бази данни вече повече от 40 години.[**]

* * *

Един друг важен за математическата логика език е езикът на λ-*смятането* измислено от Аланзо Чърч (1930 – 1936 г.). Също както предикатната логика е дала математическата основа на релационните бази данни, така λ-смятането е дало математическата основа на функционалното програмиране. И също както в предикатното смятане има свързани променливи, поради които то е неудобно за непосредствена реализация, така и в λ-смятането има свързани променливи. Само че докато при базите данни хората (поне засега) се мъчат и вместо езици със свързани променливи използват по-неудобни езици като ес кю ел, то при функционалното програмиране почти няма функционални езици без свързани променливи. Решението, което гарантира хем удобство, хем ефективност, е следното: ще предоставим на програмистите удобни езици със свързани променливи, но на ниско ниво ще реализираме тези езици посредством бързи виртуални машини без свързани променливи, а преводът ще се прави от компилатора напълно автоматично.

Първият и изключително остроумен начин за елиминация на свър-

[*] При цилиндричните алгебри няма функционални символи, но от гледна точка на изразителната сила това не е съществено ограничение.

[**] В последно време скоростта на компютрите при последователни пресмятания лека полека започва да достига своя максимум. Все още обаче има възможност за увеличаване на паралелността. Тъй като релационните бази данни могат да използват в пълнота паралелността на компютрите, това означава, че през следващите години можем да очакваме, че бързината и възможностите на специализираните системи за управление на бази данни ще се увеличат в значително по-голяма степен, отколкото бързината на изпълнение на обичайните компютърни програми.

заните променливи от λ-смятането всъщност е измислен преди да се появи самото λ-смятане. Това е *комбинаторната логика* на Мойсей Шейнфинкел и Хаскел Къри (1924 – 1930 г.). На нейна база много покъсно е измислена т.н. SK-*машина*, която се използва за компилация напр. на езиците SASL и миранда. Освен оригиналния метод на Шейнфинкел и Къри, вече са разработени и много други методи за отстраняване на свързаните променливи. На практика всяка ефективна виртуална машина, използвана за реализация на съвременен функционален език (напр. категорната абстрактна машина, виртуалните машини с комбинатори и суперкомбинатори), е получена в резултат на теоретични логически изследвания.

* * *

Въпреки че като цяло уменията на компютрите да измислят математически доказателства са несравнимо по-слаби от уменията на един професионален математик, има една област, в която компютрите нерядко се справят значително по-добре от хората — доказателства за коректност на сложни паралелни програми.

Работата на една паралелна програма протича недетерминистично и за да можем да кажем, че тя е коректна, трябва да сме сигурни, че при всички възможни начини на работата програмата работи вярно. Броят на начините за протичане на изчисленията е огромен и е непосилно човек да провери ръчно всеки един от тях. Затова ако една програма е паралелна по сложен начин, програмистите няма да бъдат в състояние да се уверят, че тя е вярна. Не случайно в съвременната програмистка практика въпреки нарастващата паралелизация на съвременните компютри, съществено паралелни програми почти не се използват.

Въпреки че отказът от паралелизация е донякъде допустим подход при програмиране, той е напълно непрактичен когато се разработват интегрални схеми. Изчислителните процеси в една интегрална схема задължително трябва да бъдат силно паралелни, защото в противен случай интегралната схема би работила твърде бавно.

Решението на проблема се състои в използването на т.н. *проверка на модели* (на англ. model checking). Първо задачата, която искаме да решим, се формулира на някакъв логически език, а след това проверяваме, че интегралната схема или програмата удовлетворява логическата формулировка. Една от най-използване за целта логики е *линейната темпорална логика*. Тази логика е удобна, защото верността при нея може да се проверява посредством сравнително бързи алгоритми, използващи крайни автомати.[*] В днешно време няма разработчик на

[*]Под „сравнително бързи“ тук се има предвид с експоненциална сложност. При

интегрални схеми, който не използва проверка на модели. Що се касае до използването на този метод за проверка на софтуер, то тук приложенията са засега по-ограничени и свързани най-вече с проверка на драйвери на физически устройства.

Трябва да отбележим, че проверката на модели е нещо напълно различно от т.н. доказателствено програмиране, за което ще спомена малко по-нататък. Тъй като далеч не всяка задача, която искаме да решим, може да се формулира на езика на линейната темпорална логика, методът проверка на модели е приложим само за един сравнително тесен клас задачи. Ако обаче той е приложим, то той със сигурност ще ни даде еднозначен отговор дали програмата е вярна или не. За разлика от проверката на модели, доказателственото програмиране е приложимо при всякакъв вид задачи, но не винаги дава отговор.

* * *

Различни видове *логики за знания* могат да се използват, за да се доказва коректност и безопасност на комуникационни протоколи, на възстановяване на база данни след аварии и др. Да разгледаме един хипотетичен пример. Хакерите Чингиз и Атила решили да атакуват уеб-сайта на Демагог. За да бъде успешна атаката им, те трябва да я започнат едновременно. Как могат да се договорят за това? Чингиз праща съобщение до Атила, в което съобщава часа на атаката и му пише, че няма да атакува, ако не получи потвърждение. Атила отговаря положително на запитването на Чингиз и също му пише, че ще участва в атаката, ако знае, че в нея ще участва и Чингиз. Ще осъществят ли атаката двамата? Не, защото Атила не знае дали Чингиз ще атакува и затова Атила няма да атакува. От друга страна Чингиз знае, че Атила не знае дали Чингиз ще атакува и затова знае, че Атила няма да атакува и значи и той няма да атакува. За да бъде преодолян този проблем, Чингиз пише до Атила, че знае, че Атила знае, че Чингиз иска да атакува. Но и след това потвърждение Чингиз все още не може да атакува, защото не знае дали съобщението му е било получено от Атила. Затова Атила пише до Чингиз, че знае, че Чингиз знае, че Атила знае, че Чингиз иска да атакува. Но тъй като в този момент той не знае дали Чингиз знае, че Атила знае, че Чингиз знае, че Атила знае, че Чингиз иска да атакува, то атаката отново е невъзможна.

Очевидно по този начин Чингиз и Атила не могат да започнат координирана атака. Има ли друг, по-смислен протокол, посредством който двамата могат да постигнат желаната договорка? Оказва се, че не. С

този вид задачи сме се примирили да считаме експоненциалните алгоритми за бързи.

използването на логики за знания може да се докаже, че колкото и да са хитри двамата, е невъзможно да се уверят, че всеки от тях знае каквото трябва да знае.

Съждителната динамична логика с номинали е един от най-мощните логически езици, за които е известен алгоритъм за проверка на верността. С тази логика може да се решават всички задачи, които могат да се решават посредством линейната темпорална логика, но за разлика от нея, тя може да се използва и като логика за знания с цел да се проверява коректността на комуникационни протоколи. Тя е открита от Соломон Паси и проф. Тинко Тинчев, а Георги Гаргов е установил, че за тази логика съществува алгоритъм, който проверява дали една логическа формула е вярна. Това е едно от онези български математически постижения, които са цитирани най-често в международните научни публикации.

Използвайки съждителната динамична логика с номинали, Борислав Ризов е разработил система, с помощта на която филолози могат да откриват думи, притежаващи определени семантични свойства. Например те могат да зададат на системата следния въпрос: „намери ми дума, която в българския език е синоним на думата 'обитавам', но ако преведем тази дума и думата 'обитавам' на английски, ще получим думи, които не са синоними“.[*]

* * *

През последните години бурно развитие имат различни логики, описващи взаимодействията между различни геометрични обекти, в които базовото геометрично понятие не е „точка в пространството“ а „област (регион) в пространството“. Такива логики могат да се използват при разработката на „умни“ устройства, които могат да се придвижват в пространството без да бъдат управлявани от човек. Тези логики се използват също и при разработката на „изкуствения интелект“ на някои видове компютърни игри. Няколко български математици са работили върху „безточковите“ геометрии.[**] Например Владислав Ненчев е изобретил безточкови геометрични логики, в които могат да се формулират сложни съждения от типа „докато обект A се допира до B или е в контакт с C, A ще се намира изцяло в региона D“. Освен това Ненчев е намерил и ефективен алгоритъм, посредством който е възможно да се проверява верността на съждения, изказани в тези логики.

[*] Достъп до системата може да се получи на адрес `http://dcl.bas.bg/bulnet/`.

[**] Сред тях са Николай Белухов, проф. Димитър Вакарелов, проф. Георги Димов, Татяна Иванова, Владислав Ненчев, проф. Тинко Тинчев.

Теория на множествата

Компютърните програми се пишат на езици с точен синтаксис и общо взето точна семантика. Независимо по колко сложен начин са свързани по между си компонентите на една програма, този начин е описан по най-прецизен начин в текста на всяка една компютърна програма. А този програмен текст не се появява от нищото, ами преди изобщо да сме в състояние да започнем да го пишем, е нужно да отделим не малко време за мислене. И колкото по-сложна е една програма и по колкото по-сложен начин са свързани по между си компонентите ѝ, толкова по-сложна и решаващо важна е тази първа стъпка от създаването на програмата — обмислянето на нейната архитектура, на потоците от данни в нея, на взаимодействието на програмата с външния свят, на взаимодействието на един програмен компонент с друг, на структурите данни и алгоритмите, които ще се използват, на методите за откриване на грешки, на възможностите за бъдещо развитие на програмата. Ако по време на това предварително обмисляне допуснем грешка, последствията са сериозни и нерядко непоправими. Програмата трябва или да се пренапише, или да се остави с дефекти, описани в документацията ѝ.

По време на предварителното обмисляне на една програма се налага да работим с много по-абстрактни понятия, отколкото когато дойде време да я пишем. Начинът, по който разсъждаваме, е същият, който използваме тогава, когато измисляме нова математическа теория. А всеки работещ математик е познал от собствен опит, че математическата интуиция може да заблуждава. След като измисли нещо ново, математикът задължително го проверява — дава прецизни дефиниции на понятията и формулира разсъжденията, които дотогава са съществували само в неясен вид в ума му. Ако всичко излезе както трябва — хубаво, но нерядко се оказва, че след точна формулировка нещата излизат не точно такива, каквито си ги е мислил математикът.

Същото се случва и при предварителното обмисляне на една програма. И колкото по-рано открием допуснатите грешки, толкова по-добре. Само че докато математикът е трениран да използва точен математически език и точни методи за разсъждение, то проектантът на една програма за съжаление най-често е самоук и не знае как провери предварителните си идеи. На него не му остава нищо друго, освен да пристъпи към написването на програмата и попътно да си доизяснява нещата и да ги поправя, ако все още не е твърде късно.

В университетските математически курсове на студентите се преподават доказателства. На пръв поглед това може да се стори ненужно — защо например е нужно в курса по математически анализ да ни се

преподава доказателството на правилото на Гийом Франсоа дьо Лопитал? Самото правило се помни лесно и се използва лесно, а хиляди математици преди нас са чели доказателството му и са се уверили, че то е вярно. Защо трябва и ние да учим това доказателство, нима се съмняваме във верността му? Е, някога математиката се е преподавала без доказателства, а просто като съвкупност от проверени практически правила, но за щастие от тогава сме се поучили, че така не бива. В областта на програмирането обаче, още не е дошло времето да си извлечем аналогична поука.

В древността е имало велики архитекти, проектирали сгради, на които и днес се възхищаваме, но не е имало наука архитектура. За да бъдеш добър архитект се е искало огромно количество талант, изкуство, усет, опитност, вярна интуиция. Чак когато натрупаният опит бил оформен във вид на систематизирано знание, станало възможно и по-обикновени хора да станат архитекти или строителни инженери и да проектират сгради, които няма да паднат преди да са построени. Аналогична е ситуацията и със софтуерната архитектура и софтуерното инженерство. За съжаление обаче опитите за систематизиране на знанията в тези области са сравнително малко, така че добрите софтуерни архитекти и софтуерни инженери са такива не защото са завършили специалност софтуерно инженерство в някой университет.

Съвременната ситуация с обучението по информатика не е добра — в много университети студентите изобщо не се обучават как да проектират правилни компютърни програми. Езиците за спецификация, ако изобщо се преподават, се преподават сравнително повърхностно, а най-лошото е, че студентите така и не разбират кога и как трябва да се използват тези езици на практика. Немалко специалисти например препоръчват в книгите си използването на формални спецификации като вид споразумение между клиентите и програмистите, а няма съмнение, че те точно това преподават и на студентите си. Въпреки че в някои случаи това наистина е полезно, най-често формалната спецификация от една страна не отговаря на истинските желания на клиента, а от друга — забранява на програмистите да реализират варианти, които са хем по-лесни за програмиране, хем по-полезни за клиента.

В други случаи се създава погрешното впечатление, че е най-добре най-напред да се специфицира цялата програма и едва след това да се пристъпи към писане на програмния код. В действителност ако програмата е достатъчно сложна, това е наистина абсурден начин за програмиране. То е все едно учените и инженерите, участващи в проект за изпращане на хора до Луната, директно да се опитват да създадат чертежите на необходимата ракета и космическия кораб, както и плана

на полета, без преди това да са осъществили някоя по-проста част от целия проект. Полетът до Луната би бил невъзможен, ако преди това разработчиците не бяха усвоили излизането със скафандър в открития космос, маневрирането в околоземното и окололунното пространство, скачването на два апарата в околоземното и окололунното пространство и т.н. По същия начин трябва да се постъпва и при програмиране. Колкото и внимателно да сме подготвили спецификацията на програмата, неминуемо реалното писане на код ще ни поднесе изненади. Затова след като някоя част от програмата ни се изясни и сме готови със спецификацията ѝ, най-добре е да пристъпим към програмирането ѝ, макар останалата част от програмата все още да остава неясна.

За съжаление доброто желание не винаги е достатъчно... Много е лесно ако използването на език за спецификации бъде наложено силово от ръководството на една фирма, а екипът няма нужния опит, вместо този език да стане мощно средство за подпомагане на програмистите, той да се превърне в инструмент за мъчение.* Наистина има какво да се сбърка — кои свойства на програмата е важно да бъдат включени в предварителната спецификация и за кои ще бъде вредно това да се прави, до каква степен да проявяваме гъвкавост и да модифицираме предварителната спецификация, ако след като започне писането на програмата установим, че нещо е било по-добре да се направи по друг начин и т.н. И всички тези проблеми са при положение, че екипът вече е усвоил използването на формален език за спецификации, което само по себе си е предизвикателство.

* * *

Вече споменахме, че използването на формална спецификация може да помогне на проектантите да открият фундаментални грешки и недостатъци в конструкцията на програмата много преди да започне нейното писане. Тази полза е налице дори тогава, когато проектантът е един човек и прави спецификацията само за себе си. И наистина, формалното описание на нужните свойства отнема много по-малко време, отколкото самото написване на програмата, а откритите благодарение на него грешки се поправят много по-бързо. И когато наистина дойде време да пишем програмния текст, структурата на програмата ни е ясна и затова програмираме по-бързо и допускаме по-малко грешки.

От използването на формални спецификации произтича и една друга, по-незабележима, но съвсем не маловажна полза. Каквато и дейност да извършва човек, той неволно винаги се опитва да я свърши по въз-

*Един познат програмист ми каза точно това — езиците за спецификация са инструмент за измъчване на програмистите и заблуждаване на клиентите.

можно най-лесния начин. Затова когато програмистите пишат програмния текст, те разсъждават локално — как да свършат задачата по най-простия и елегантен начин без да мислят дали създадените по този начин библиотеки от класове или функции са прости за използване и с ясно и разбираемо действие. Когато обаче се пише спецификация, човек пак в желанието да си спести труда, не мисли колко трудна е реализацията, а само за това колко просто са формулирани желаните свойства на програмните компоненти. В резултат на това взаимодействието на програмните компоненти става възможно най-опростено и лесно за разбиране, а програмните грешки най-често са локални и лесни за оправяне. Опростеното взаимодействие на програмните компоненти помага и при последващата поддръжка на програмния код. Когато модифицират вече написаната програма, програмистите, дори да не са участвали в първоначалното ѝ написване, се чувстват много по-уверени, че действията им няма да доведат до непредвидени грешки.

Фирми, които са опитвали да специфицират вече написани програми, са установили, че това е безсмислена задача, защото изведнъж става нужно да опишат точно всичката онази невидима сложност във вече написаната програмата. [2, стр. 16] Сложност, която със сигурност е затруднила поправянето на програмните грешки и със сигурност ще затрудни последващата поддръжка.

* * *

Къде във всичко това присъства теория на множествата? Е, оказва се, че всички езици за формална спецификация по един или друг начин се свеждат до използване на езика на теория на множествата. Тази незаменимост на езика на теория на множествата не е изненадваща — щом като този език е от една страна необходим в съвременната математика, а от друга страна напълно достатъчен, за да изкажем на него цялата съвременна математика, без нито едно изключение, защо да не очакваме същото да важи и за програмирането?*

По-старите езици за спецификация (напр. зед) умишлено подчертават връзката си с теория на множествата. Използването на тези езици естествено е невъзможно от програмисти, които не са получили нужната математическа подготовка. В по-ново време обаче стана ясно, че вместо непосредствено за множества, можем да използваме терминология, по-разбираема за програмистите. Например вместо да казваме, че x е елемент на множеството A, можем да казваме, че x е обект от

*Дълго време изглеждаше, че конструктивната математика задължително изисква използването на не-множествен език. След създаването на конструктивната теория на множествата стана ясно, че това не е така.

класа A; вместо да казваме, че предикатът $p(x, y)$ е истина за x и y, можем да казваме, че обектът x има поле с име p, чиято стойност е y; вместо да използваме квантори, може да *скулемизираме* и т.н. Всичко това обаче в никакъв случай не е заместител на теория на множествата, а просто начин да се даде на програмистите без математическа подготовка език, равносилен на езика на теория на множествата. При използването на такъв език е важно да се знае, че „класовете“, за които се говори в спецификацията, изобщо не е нужно да съответстват на реални класове в програмата, „полетата“, които притежават обектите, не е нужно да съществуват и в истинската програма и т.н.

Теория на изчислимостта

Когато на света се появили компютрите, оказало се, че важна част от теорията на това, какво и как може да се смята с тях, вече съществувала. Днес този дял от математическата логика се нарича „*теория на изчислимостта*“.*

В доказателството на своите теореми за непълнота Гьодел дефинирал понятието „*примитивнорекурсивна функция*“. Малко по-късно, през 1934 г., Гьодел дефинирал и понятието „*общорекурсивна функция*“, което се оказало еквивалентно на съвременното понятие „изчислима функция“.** По онова време обаче Гьодел все още не допускал, че понятието, което той дефинирал, обхващало всички функции на естествени числа, за които може да се счита, че са изчислими.

Тезата, че интуитивното понятие „изчислима функция“ отговаря на даден, точно дефиниран клас от математически функции, е изказана за пръв път през 1936 г. от трима математици — Аланзо Чърч, Алън Тюринг и Емил Пост. Всеки от тях дал различна дефиниция на това какво значи изчислима функция, но тези дефиниции се оказали еквивалентни както по между си, така и на по-ранната дефиниция на Гьодел. Според Чърч изчислимите функции са точно функциите, които могат да се дефинират с т.н. λ-термове. Това твърдение днес е известно като „*тезис на Чърч*“, а създаденото от Чърч *λ-смятане* служи за математическа основа на почти всички съвременни функционални езици. Тюринг пък дефинирал това, което днес е известно като „машина на Тюринг“, а твърдението, че изчилимите функции са точно функциите, които може да се пресмятат с машина на Тюринг, е известно като

*Дълго време този дял се наричал „теория на рекурсията“, а изчислимите функции — рекурсивни функции.

**Според Гьодел идеята за тези функции му е била подсказана от Ербран.

„*тезис на Тюринг*“.* Дефиницията на Пост за изчислима функция е удивително сходна с дефиницията на Тюринг, макар да е получена напълно независимо. Простотата на тези две дефиниции ги прави много удобни за математически изследвания.

По-късно се появили и други, различни от тях дефиниции на „изчислима функция“, но и те се оказали еквивалентни по между си. Сред тях по-важни са каноничните системи на Пост (1943), нормалните алгоритми на Марков (1948) и различните видове регистрови машини (1954 – 1967). Каноничните системи на Пост и нормалните алгоритми на Марков стоят в основата на символните пресмятания в съвременната компютърна алгебра, както и на някои езици за програмиране, напр. рефал. Регистровите машини пък и особено машините на Мински съчетават теоретичните удобства, които имат машините на Тюринг, с обичайния стил на програмиране, който се използва при императивните езици за програмиране.

В някои случаи теорията на различните изчислителни модели дава идеи за нови стилове за програмиране (напр. идеята на функционалното програмиране е дошла от λ-смятането). В други случаи тя дава идеи за ефективно използване на виртуални машини. Методи от математическата логика се използват също и при разработката на компилатори и особено на оптимизиращи компилатори. Все по-голямо значение добиват системите за статичен анализ на програмен код. Такива системи например автоматично могат да откриват дали някоя променлива се използва без да е инициализирана.

* * *

В теория на изчислимостта се разработват различни методи за дефиниране на семантиката на различните езици за програмиране. Наличието на точна семантика на един език е много важно при реализацията му, защото в противен случай комбинираното използване на различни свойства на езика, всяко от които само по себе си изглежда ясно, се оказва, че води до програми, за които не е ясно какво точно трябва да правят, и затова компилаторът работи неправилно.

Различните изчислителни модели и семантиките на езиците за програмиране могат да подскажат на създателите на нови езици за програмиране как да дефинират езиците така, че те да бъдат ясни и с проста семантика (в противен случай става много трудно да се пишат верни програми на тези езици). Например в областта на обектноориентираното програмиране използването на сходна терминология от различните

*Тъй като дефинициите на Чърч и Тюринг са еквивалентни, от математическа гледна точка тезисът на Чърч и тезисът на Тюринг казват едно и също нещо.

езици за програмиране създава илюзията, че всички те реализират на практика едно и също нещо. В действителност на семантиката на различните обектноориентирани езици показва, че те могат да се разделят на две групи. В едната група спадат езици, чиято математическа теория се базира на безтипови обекти. В тази група попадат например езиците смолтоук и обджектив си.* При множественото наследяване при тези езици най-много един от родителските класове може да бъде истински, докато останалите могат да предлагат само интерфейс, но не и конкретна реализация. Втората група обектноориентирани езици са тези, чиято математическа теория се базира на някоя теория на типовете. Към тази група езици спада айфел. При множественото наследяване при тези езици често се допуска всеки един от родителските класове да предлага конкретна реализация. А има и езици като си++, при чието създаване явно не се е обръщало достатъчно внимание на това каква точно трябва да бъде тяхната семантика. Може би затова множественото наследяване при си++ е трудно за използване и всъщност никой не го използва.**

В днешно време има много езици, които изобщо не притежават оператор goto, но това, че операторът goto наистина не е нужен, става ясно от една теорема на Корадо Бьом и Джузепе Якопини от 1966 г.

Важни резултати в областта на семантиките на езиците за програмиране са получени в България от проф. Иван Сосков. По-горе се спомена, че съществуват различни дефиниции на понятието изчислима функция и всички те се оказват еквивалентни. Обаче това е така само когато говорим за функции върху естествени числа. Затова един естествен въпрос е какво се случва ако вместо функции върху естествени числа използваме функции, дефинирани за произволен абстрактен тип данни. Оказва се, че в този случай различните математически модели за изчислимост не са еквивалентни. Така например Сосков е установил, че: 1. за всяка императивна програма можем да намерим еквивалентна на нея функционална програма, но с функционални програми понякога може да се правят и неща, които не могат да се направят с императивни [28] и 2. за всяка функционална програма можем да намерим еквивалентна на нея логическа програма, но с логически програми

*Към тази група езици спада и джава, въпреки че езикът създава илюзията за статична типизация.

**Понякога хора, повлияни от си++, твърдят, че множественото наследяване е едва ли не безполезно и опасно, когато повече от един от родителските класове предлага реализация. Всъщност езикът айфел нагледно показва точно обратното — ако езикът е базиран на хубава математическа теория, такова наследяване не само е напълно безопасно и полезно, но също така и много приятно за използване.

понякога може да се правят и неща, които не могат да се направят с функционални. [18]

* * *

Един от важните проблеми в съвременната информатика е свързан с удобното и безопасно използване на паралелни пресмятания. Този проблем има голямо практическо значение, защото компютрите стават все по-паралелни. За решаването му се разчита на създаването на прост математически модел на паралелните пресмятания, но за съжаление такъв все още не е измислен. Вредата от това е не само теоретична — отсъствието на хубава математическа теория води до това, че паралелното програмиране с право се счита за опасно и на практика почти винаги води до трудни за откриване и оправяне програмни грешки. Два съществуващи модела на паралелните пресмятания се дават напр. от *π-смятането* и *джойн-смятането*. Макар те да са по-сложни, отколкото ни се иска, езиците за програмиране, в които средствата за паралелизъм са базирани на някое от тези две смятания, са по-удобни за писане на правилни паралелни програми, отколкото езиците, при които паралелизмът не е базиран на никаква теория.[*]

Скоро след появата на компютрите станало ясно, че има разлика между теоретично изчислима функция и функция, която е изчислима на практика. Това мотивирало бързото развитие на изключително важната теория на сложността. За съжаление в тази област има много задачи, които се оказват изключително трудни и все още остават нерешени. Най-знаменитата и все още не решена задача е въпросът дали „P=NP“.[**] Тук P обозначава функциите, които грубо казано може да се пресмятат за полиномиално време на последователен компютър, а NP — функциите, които може да се пресмятат за полиномиално време на неограничено паралелен компютър. Този проблем е важен по две причини. От една страна класът P съдържа функциите, за които можем да считаме, че се смятат за разумно кратко време на компютър,[***] а от

[*] Изводът, който можем да си направим, е следният: програмирането с лоша математика е по-добро от програмирането с никаква математика.

[**] Обявена е награда от 1 000 000 долара за този, който пръв намери вярно решение на този проблем.

[***] Става въпрос за компютър със стандартна архитектура. Квантовите компютри могат да решават ефективно някои задачи, които изискват неограничена паралелизация. В частност повечето от използваните в момента алгоритми за асиметрична криптография ще престанат да бъдат сигурни, ако разполагаме с квантови компютри с няколко хиляди кубита памет. Предполага се обаче, че не всяка NP задача може да се решава ефективно от квантов компютър, и има алгоритми за асиметрична криптография, които са използваеми с нормален компютър и за които се счита, че биха останали сигурни дори и да разполагаме с квантови компютри с голяма

друга — има много на брой изключително важни за практиката задачи, които принадлежат към класа NP. Ако имаме щастието да се окаже, че P=NP, то това означава, че ще разполагаме с ефективен алгоритъм за решаване на тези важни задачи. От друга страна, ако имаме нещастието да се окаже, че P=NP, то това може да направи неизползваеми всички използвани на практика криптографски алгоритми.

Освен с директно измерване на времето за пресмятане и използваната памет, един алтернативен подход за измерване на сложността на една изчислима функция е това колко сложна е нейната дефиниция (на пръв поглед не се вижда защо има връзка между сложността на дефиницията на една функция и времето за нейното пресмятане, но такава има). Функциите, които притежават проста дефиниция, се изучават в теорията на субрекурсивните функции. Субрекурсивните функции описват по-добре отколкото изчислимите функции това, какво на практика може да се пресмята с компютър. Затова е важно да бъде изследвана не само субрекурсивната изчислимост между естествени числа, но също и субрекурсивната изчислимост между други полезни математически обекти. Субрекурсивната изчислимост на реални числа все още е сравнително малко изследвана област, като повечето от съществуващите резултати са получени в България от Иван Георгиев и проф. Димитър Скордев.

Когато в математиката бъде разработена теорията на определен вид, в някакъв смисъл подобни обекти, е естествено тази подобност да бъде формализирана, след което систематично да бъдат потърсени други обекти, притежаващи така формулираните свойства. Например в алгебрата след като се открият общите свойства на различните числови полета, получаваме аксиомите на абстрактното понятие поле и установяваме, че има полезни нечислови полета. В областта на теория на изчислимостта същото нещо е направено за пръв път от проф. Димитър Скордев [27, 17]. Той е дефинирал аксиоматично понятието „*комбинаторно пространство*“ и е показал че някои от основните резултати в теория на изчислимостта могат да бъдат изведени като следствие от аксиомите на комбинаторните пространства. Самите комбинаторни пространства пък се оказва, че имат много интересни модели, които ни дават неочаквани нови видове изчислимости (много от тях все още очакват някой да им разработи теорията в пълнота). След комбинаторните пространства Любомир Иванов [10] е дефинирал понятието „*оперативно пространство*“, а важни резултати в тази област са получени и от Йордан Зашев. Всички тези резултати са дали началото на това,

памет.

което днес се нарича аксиоматична теория на изчислимостта.*

Теория на доказателствата и конструктивна математика

Вече бе споменато, че ако искаме да разработваме математиката конструктивно, ще се наложи да разсъждаваме, използвайки алтернативна логика, която е различна от класическата. Тази логика се нарича *интуиционистка логика*, а законите ѝ били формулирани от Аренд Хейтинг през 1930 г.**

В класическата математика когато твърдим, че нещо съществува, е все едно дали разполагаме с метод, посредством който можем да намерим съществуващото нещо. В конструктивната математика това не е така. Ако един конструктивист докаже, че съществува обект с определено свойство, то значи той разполага с метод, посредством който може да бъде намерен този обект. Ако пък бъде доказано съждение от вида „за всяко x е вярно $x \leqq 0$ или $x \geqq 0$“, от тук автоматично ще следва, че има метод, с помощта на който за всяко x можем да познаваме дали е вярно $x \leqq 0$ или $x \geqq 0$.

Според конструктивистите доказателството на едно съждение A представлява точно определена конструкция, която реализира A. Например доказателството на твърдение от вида „ако A, то B“ ни дава конкретен метод, с помощта на който ако разполагаме с конкретно доказателство на A, ще получим конкретно доказателство на B. С други думи можем да си мислим, че доказателството на „ако A, то B“ е функция, на която ако ѝ дадем като аргумент доказателство на A, ще ни върне като стойност доказателство на B. Доказателството пък на твърдение от вида „A и B“ представлява просто комбинацията от конкретно доказателство на A и конкретно доказателство на B.***

Казаното до тук може би подсказва, че би трябвало да има някаква връзка между интуиционистката логика и програмирането. Дали няма да се окаже, че всяка интуиционистка логическа формула дефинира

*Това засега може би е единственият дял в математиката, който е създаден в България. В България той по-често се нарича „алгебрична теория на рекурсията“.

**Това, че е възможно формулираме конструктивните разсъждения, използвайки сравнително прости формални правила и аксиоми, е било изненада за създателя на интуиционизма Брауер. Отначало той считал, че не някакви формални правила и аксиоми, а единствено ясната математическа интуиция може да ни даде правилна основа на математиката. Затова той се е противопоставял на опитите за формализация на интуиционизма, а изследванията на Хейтинг наричал „безрезултатни упражнения“.

***Този начин за интерпретация на логическите връзки е известен като интерпретация на Брауер-Хейтинг-Колмогоров.

някаква изчислителна задача, а доказателството на тази интуиционистка формула представлява програма, която решава тази изчислителна задача? Оказва се, че това е точно така и това е забелязано от Хаскел Къри и Уилям Алвин Хауърд. Може да считаме, че всяка интуиционистка формула представлява някакъв тип данни, а всяко доказателство на формулата е програмен обект (напр. функция), притежаващ съответния тип. Вярно е и обратното — може да си мислим, че типовете данни са интуиционистки формули от определен вид, а всяка компютърна програма представлява някакво интуиционистко доказателство. Наборът обекти, дефинирани в една компютърна библиотека, може да бъде считан за набор от аксиоми на интуиционистка математическа теория, а всяка програма, която е написана, използвайки единствено обектите от предоставената библиотека, представлява интуиционистка теорема, доказана използвайки само дадените аксиоми.

Това удивително съответствие между програмиране и математика е известно като „*изоморфизъм на Къри – Хауърд*“. То може да бъде изказано не само неформално (както направихме тук), но също и като точно математическо твърдение. Изоморфизмът на Къри – Хауърд показва, че програмирането и правенето на математика представляват не просто две подобни дейности, а напълно идентични дейности.

Ползата от този изоморфизъм е не само философска, защото той прояснява какво всъщност представляват типовете данни в езиците за програмиране. Създателите на почти всички типизирани функционални езици за програмиране умишлено разчитат на този изоморфизъм, за да си гарантират, че системата от типове ще бъде хем проста за използване, хем пълна. Напълно заслужено на името на Хаскел Къри е наречен един от най-популярните в момента езици за функционално програмиране — хаскел.

Този изоморфизъм има и други практически приложения. Например той подсказва как да разширим типовете данни в един език за програмиране по такъв начин, че спецификацията на това какво прави една функция да бъде част от нейния тип. И също както при използването на обикновени типове компилаторите могат статично да проверяват дали няма нарушения на типовете, така и при използването на тези разширени типове се оказва, че това е възможно. Това позволява компилаторът не само да компилира програмата, но и да удостоверява, че тя е вярна и не съдържа нито една програмна грешка. Въпреки че засега всички съществуващи системи от този тип са трудни за използване, вече има оптимизиращ компилатор на си, за който знаем, че няма програмни грешки, защото е програмиран, използвайки система с интуиционистки типове.

В България изследвания свързани с изоморфизма на Къри – Хауърд са правени от Трифон Трифонов.

* * *

От напълно различен вид е връзката между математика и програмиране при логическото програмиране. При логическото програмиране логическите формули се интерпретират не като типове данни (както е при функционалното програмиране), а като компютърни програми. Процесът на търсене на доказателство на логическата формула пък представлява изпълнението тази програма.

Освен като теоретична основа на логическото програмиране, компютърното търсене на доказателства е полезно и само по себе си. Наистина, съществуващите в момента алгоритми за автоматично доказателство на теореми не могат да се сравняват по ефективност с уменията на един професионален математик. Всеки, който се е занимавал професионално с програмиране, обаче знае, че през повечето време на програмистите им се налага да пишат напълно тривиален и безинтересен от математическа гледна точка програмен код. Не може ли „отговорността“ за този тривиален програмен код да бъде поверена на компютрите, а за хората да останат само по-интересните и интелектуално предизвикателни задачи?

На този въпрос може да бъде отговорено двояко. От една страна — да. Съществуват системи, които се справят много добре с генерирането на програмен код. Но на практика — не. Причината е тази, че да обясним на компютъра какво точно искаме да направи дадена програма може да бъде не по-малко досадно, отколкото просто да седнем и да си напишем сами програмата. Въпреки това изследванията за намиране на удобна за използване система за автоматично генериране на код продължават.*

Съвсем по-друг начин стоят нещата при доказателственото програмиране, т.е. тогава, когато се пишат програми, за които искаме да сме сигурни, че не съдържат грешки. До голяма степен именно благодарение на успехите в областта на автоматичното доказателство на теореми доказателственото програмиране буквално през последните няколко години от теоретично академично упражнение се превърна в нещо, което е напълно използваемо на практика. Фирмите, които се занимават с писането на програми без грешки, са установили, че дори и при много големи програми обикновено е достатъчно в екипа да има само един математик логик, който да установява коректността на онези места в

*Когато такава система бъде измислена, навсякъде ще се търсят логици, а програмистите ще останат без хляб.

програмата, чиято коректност не е могла да се докаже автоматично от компютър [2, стр. 376]. Интересно е, че системите за доказателствено програмиране могат да формулират задачите, които не са успели сами да докажат, по такъв начин, че не е нужно математикът логик да умее да програмира и всъщност не е нужно дори да знае за какво служи програмата, от която е възникнала поставената задача.

1.3. Логическо програмиране

Създаването на пролог

През 1958 Франция била разкъсвана от политически и икономически проблеми. Остров Корсика бил завлядян от алжирски военни и имало реална опасност от държавен преврат. В обществото нямало единство, политическата власт била слаба. Тогава в политиката се намесил генерал Шарл дьо Гол, герой от Втората световна война. Генерал Дьо Гол решил, че за да се решат проблемите, френското общество трябвало да бъде по-обединено, а единственият начин това да стане, е да го поведе под патриотични лозунги. Затова той заявил: „Нашата страна пред лицето на другите страни трябва да се стреми към велики цели и да не се прекланя пред никого, защото в противен случай може да се окаже в смъртна опасност“. Франция променила коренно политиката си. Въпреки ненавистта си към комунизма, Дьо Гол изкарал Франия от НАТО и дори започнал политическа заигравка със СССР. Използвайки противопоставянето между тогавашните суперсили САЩ и СССР, Дьо Гол успял да увеличи значително международната значимост на Франция и правото ѝ да води независима политика.

През 1967 пред стохиляден митинг в Монреал Шарл дьо Гол се провикнал: „Да живее свободен Квебек!“. Отношенията между Франция и Квебек достигнали ниво подобно на това, което имат независимите държави. Франция отзовала своя „Генерален консул в град Квебек“ и вместо него назначила „Генерален консул пред правителството на Квебек“. Започнали взаимни визити на френски и квебекски политици. Френските политици редовно нарушавали дипломатическите правила като посещавали Квебек без изобщо да се появят преди това в канадската столица Отава. Разширило се научното сътрудничество между Франция и Квебек. Френски учени се премествали да работят и преподават в Квебек, а квебекски — във Франция.

През 1967 г. в Монреал, Квебек, се преместил и младият френски учен Ален Колмеро. Тук той създал Кю-системите — един от първите формализми за обработка на естествени езици. Колмеро бил научен

ръководител на квебекчанина Жан Трудел, който работил в областта на автоматичното доказателство на теореми. През 1970 г. Колмеро поканил в Монреал и току-що дипломиралите се французи Робер Пазеро и Филип Русел. Там те се запознали с компютърната обработка на естествени езици.

През 1970 г. Дьо Гол починал и година по-късно групата около Колмеро заедно с квебекчанина Пазеро се върнала във Франция. Тук те създали система, която можела да води диалог на естествен език, като сама правела логически изводи. Езиковата част била разработена от Колмеро и Пазеро, а за автоматичното правене на логически изводи отговаряли Трудел и Русел. Още през същата година системата била достатъчно развита, за да води следния диалог:

Потребител:

Котките убиват мишки.

Том е котка, която не обича мишки, които ядат сирене.

Джери е мишка, която яде сирене.

Макс не е мишка.

Какво прави Том?

Компютър:

Том не обича мишки, които ядат сирене.

Том убива мишки.

Потребител:

Кой е котка?

Компютър:

Том.

Потребител:

Какво яде Джери?

Компютър:

Сирене.

Потребител:

Кой не обича мишките, които ядат сирене?

Компютър:

Том.

Потребител:

Какво яде Том?

Компютър:

> Това, което ядат котките, които не харесват мишки, които ядат сирене.

През това време Русел и Трудел продължили да експериментират с различни методи за автоматично правене на логически изводи. През 1971 Трудел се запознал с току-що измислената от Робърт Ковалски и Доналд Кунър SL-резолюция. Ковалски и Кунър работели в Единбург, но Трудел убедил останалите от групата да поканят Ковалски за кратко посещение в Марсилия, за да им разкаже за метода.

Марсилската група около Колмеро се състояла от хора практици. Те не пропускали случая да се запознаят с нови теоретични резултати, които биха могли да им свършат работа, но собствената им научна дейност не била свързана със създаване на нова математика. От друга страна, Ковалски бил теоретик. Когато прилагаме една теория на практика, винаги се налага да решаваме проблеми, които са от практическо естество и не представляват теоретичен интерес. На математиците теоретици, какъвто бил и Ковалски, не им е безинтересно как се прилагат на практика откритите от тях неща, но много често на тях не им харесва сами да решават практическите проблеми свързани с приложението на теорията. Когато обаче се срещнат добри практици с добри теоретици, винаги всеки научава нещо ново и полезно за себе си. Марсилската група се запознала по-добре със свойствата на SL-резолюцията, а това се оказало решаващо важно за измислянето на пролог. Ковалски пък научил неща за практическото използване на резолюцията, които не знаел по-рано, създал теоретичните основи на логическото програмиране и в крайна сметка точно с това се и прочул. За SL-резолюцията, от която тръгнало всичко, днес малцина си спомнят, а още по-малко са тези, които знаят какво всъщност представлява тя.

През 1972 г. Ковалски бил поканен да посети Марсилия за втори път, този път за по-дълго. По това време групата вече имала по-ясна интуиция как да си представя автоматичното доказване с SL-резолюция като изчислителна процедура. Те знаели как да аксиоматизират прости неща като събиране на числа, конкатенация на списъци, обръщане на списък и т.н. по такъв начин, че SL-резолюцията да намира нужния резултат ефективно.

След заминаването на Ковалски, Колмеро измислил как да елиминира Кю-системите. Всъщност, използвайки открития от Колмеро начин, и днес на пролог можем да много лесно синтактичен разбор за произволен безконтекстен език. По този начин за пръв път започнала да изглежда възможна идеята, цялата система, която разработвали в

Марсилия, да се реализира, използвайки логика.

Взето било едно много важно решение: оказало се, че за да се „изпрограмират“ нужните неща, било достатъчно да се използват само хорнови дизюнкти. С цел по-добра ефективност и по-ясна процедурна семантика, те взели още едно важно решение: позволили само резолвенти с първия литерал на хорновите дизюнкти. По този начин от чисто практически съображения те открили това, което днес се нарича SLD-резолюция. Те предполагали, че SLD-резолюцията е непълна, но считали, че това е приемлива цена заради по-добрата ефективност. Така в края на 1972 г. се появила първата реализация на езика пролог. На следващата година (1973) Ковалски доказал, че SLD-резолюцията всъщност не е непълна [11], а през 1974 заедно с Мартен ван Емден дефинирал и семантика на логическите програми с неподвижна точка [21].*

Името „пролог“ било измислено от съпругата на Филип Русел и е съкращение от „PROgrammation en LOGique“ (програмиране посредством логика).

Приложенията на пролог

Езикът пролог впечатлява с това по какъв елегантен начин съчетава в себе си мощ и простота. Кой друг език не съдържа нито една запазена дума и при това остава удобен за използване, дори ако го лишим от всички вградени в него предикати, функции и т.н.? На мнозина от първите ентусиасти на пролог им се е струвало, че е само въпрос на време, кога пролог ще стана най-популярният език за програмиране. Някои считали, че други езици за програмиране изобщо не било нужно да се учат.

Разбира се, вече знаем, че тези представи се оказали много далеч от истината. Езикът пролог изобщо не успял да стане толкова популярен, колкото очаквали. Въпреки това има някои видове приложения, при които пролог се е доказал като един от най-удобните съществуващи езици за програмиране.

* * *

Първата група приложения на пролог са свързани с основния тип данни на езика — термовете. Ако ни се налага да извършваме сложни манипулации на дървесни структури данни, то си струва да обмислим, дали да не програмираме на пролог. Компилаторите, и особено оптимизиращите компилатори, са програми, на които им се налага ма-

*Ван Емден е автор на термина „SLD-резолюция“.

нипулират такива дървесни структури. Първата реализация на езика ерланг била написана на пролог. Също и при езика СПАРК важна част от реализацията, която се грижи за коректността на компилираната програма, е написана на пролог.

* * *

Втората група от приложения на пролог е свързана с това, че всяка програма на пролог представлява съвкупност от правила, а в някои случаи е най-удобно да опишем действията на приложението, което искаме да реализираме, именно посредством съвкупност от правила. Ако решим да програмираме една такава програма на традиционен език, бихме получили огромно количество вложени един в друг условни оператори. Логиката на такава програма е трудна за проследаване, а грешките — невъзможни за оправяне.

На пролог е удобно да се реализират т.н. бизнес правила. Така например корпорацията Ериксон с такава система е вземала решения, свързани с продажбите на продукти, а банки и застрахователни компании са оценявали заявленията за предоставяне на кредити и очакваните печалби.

На пролог е написана една от най-популярните система за автоматизация на документи (DealBuilder). Тя може автоматично да генерира текстове на договори и други юридически документи, използвайки сложен набор от правила.

На пролог е написана система, която помага на потребителите да настроят мобилните си телефони. Тя се използва на уеб сайтовете на много мобилни оператори. Ясно е, че не е лесно да се съчетаят в прост алгоритъм специфичните изисквания на всеки един телефон със специфичните изисквания на отделните услуги, предоставяни от оператора.

Около една трета от резервациите на самолетни билети по света се извършва посредством система, написана на пролог.

В Windows NT 3.1 системата, която решава кои драйвери е най-подходящо да се заредят от ядрото, е написана на пролог.

* * *

Третата група от приложения са тези, при които програмата трябва да реагира по сложен начин на огромно количество взаимнозависими фактори.

По данни на Централното разузнавателно управление на САЩ, на пролог е написан софтуерът на руската космическа совалка „Буран“, която за пръв път извършила кацане на летище в напълно автоматичен режим (безпилотно и без радиоуправление).

На космонавтите на Международната космическа станция често им се налага да извършват сложни дейности. При тяхното извършване те

трябва да си подсказват с инструкции от таблет, което отвлича вниманието им и губи време. Американската космическа агенция НАСА обаче е реализирала на пролог система за гласово управление на браузър. Космонавтът казва на глас каква информация иска от компютъра, а компютърът също му отговаря със синтезиран глас.

Понякога на пролог е удобно да се пишат системи за извличане на ценна информация от огромно количество неструктурирани данни (data mining). Например Американската агенция за електронно разузнаване използва пролог, за да анализира социалните мрежи и да открива хора, за които има вероятност да се превърнат в терористи.

Много от съществуващите експертни системи са написани на пролог. Някои примери за експертни системи, написани на пролог са:

- експертна система, която анализира условията, при които се отглеждат свинете в свинефермите, и предлага промени за начина на хранене на свинете, температурата, влажността и вентилацията на въздуха, използваните породи и т.н. В някои случаи тази експертна система е увеличавала печалбата до пет пъти;
- на пролог са написани няколко експертни системи, които анализират околната среда. Те се използват за прогнозиране на времето, за прогнозиране на предстоящи глобални изменения в климата, за прогнозиране на замърсеността на въздуха в градовете и др.;
- експертна система, която предлага методи за снабдяване на населението с питейна вода в случай на стихийни бедствия, производствени аварии или война;
- самолетостроителната фирма Боинг използва пролог за програма, която снабдява работниците с необходими инструкции. Тези инструкции се получават от различни източници — обикновени файлове, бази данни, обектноориентирани бази данни, други експертни системи. Достъпът до тези източници обикновено е отдалечен (по мрежата), но всичко става невидимо за потребителя. Един от разработчиците казва: „Работата по написването на машина за заявки на пролог бе минимална в сравнение с лисп. А да я напишем на някой обикновен език би било немислимо, защото това на практика би означавало да измислим отново пролог“;
- „изкуственият интелект“ на някои компютърни игри използва пролог.

Логическо програмиране с ограничения

. . .
. . .
. . . *constraint (logic) programming*
. . .
. . .

Други разновидности логическо програмиране

Най-строгата дефиниция на това какво означава „логическо програмиране“ е следната:

Дефиниция. *Логическото програмиране* е метод за описание на компютърни алгоритми, при който:

- компютърната програма представлява логическа задача, формулирана на някакъв формален логически език;
- изпълнението на програмата от компютъра представлява търсене на доказателство на поставената логическа задача.

Например при езика пролог програмата представлява задача от следния вид: ако Γ е множество от хорнови клаузи, а φ е конюнкция от атомарни формули, да се докаже, че от Γ следва изпълнимостта на φ. Методът, който компютърът използва, за да реши тази задача, е SLD-резолюцията.

Ако вместо обикновени хорнови клаузи, използваме друг логически език, получаваме други езици за логическо програмиране.

* * *

Езикът мъркюри е на пръв поглед тривиално разширение на пролог — той се получава просто като добавим към езика типове данни и т.н. режими на предикатите. От логическа гледна точка добавянето на типове не дава на езика по-голяма изразителна сила. Когато обаче става въпрос за програмиране, тази проста добавка има следните две важни следствия. Първо, езикът мъркюри позволява да се пишат изключително бързи програми. Второ, за разлика от пролог, при мъркюри конюнкцията е напълно комутативна, а ако програмата не се зацикля, тогава и дизюнкцията става комутативна и съответствието между програмата и желаната логическа семантика е много по-пълно. Въпреки че при мъркюри няма нелогически предикати като предиката за отсичане на пролог, на мъркюри може да се пишат далеч по-бързи програми, отколкото на пролог.

* * *

Езикът ламбда-пролог добавя към пролог следните неща:

– Типове данни.

– Явни квантори. Доказателството на $\exists \mathtt{x}\, \mathtt{p}(\mathtt{x})$ се прави по същия начин, както и на пролог (където кванторът $\exists$ се подразбира). Доказателството на $\forall \mathtt{x}\, \mathtt{p}(\mathtt{x})$ се прави посредством *скулемизация* — компютърът добавя нов символ за константа `c`, доказва $\mathtt{p}(\mathtt{c})$ и след това „забравя“ за символа `c`.

– Импликация в целите. Доказателството на $\varphi \Rightarrow \psi$ се прави по следния начин — компютърът добавя към базата знания φ, доказва ψ и след това премахва φ. При програмиране тази възможност е удобно да се използва, за да се намали броят на аргументите на един предикат, както и за модулно програмиране.

– Термове със свързани променливи. Тъй като не е известен логически език или език за програмиране, при който формулите или програмите да не могат да се представят по естествен начин посредством термове със свърани променливи, то ламбда-пролог е вероятно най-удобният програмен език за обработка на формални езици.

Добавките, които ламбда-пролог прави към пролог не са „случайни приумици“, а се базират на интересна логическа теория.

* * *

Има няколко известни езика, които макар и да не отговарят на строгата дефиниция за „логическо програмиране“, дадена по-горе, все пак са силно повлияни от пролог и логическото програмиране.

* * *

Езикът ерланг се използва за писане на разпределени, паралелни програми, които са устойчиви на аварии и работят без да се спират. Много от най-натоварените уеб-сайтове по света използват ерланг за част от услугите, които предоставят. Първата версия на ерланг е реализирана на пролог, а синтаксисът на ерланг е силно повлиян от синтаксиса на пролог.

* * *

Тъй като езикът ес кю ел не използва свързани променливи, а алгебрични операции, той е лесен за реализация, но неудобен за използване. Най-перспективният език със свързани променливи за заявки към бази данни е дейталог. Една заявка на дейталог на пръв поглед прилича на програма на пролог. Нека например в базата данни имаме таблица $\mathtt{оценка}(x, y)$, която казва, че студентът x има оценка y по логическо

програмиране. И нека още имаме таблица `група`(x, y), която казва, че студентът x е от група y. Питаме се в кои административни групи има студенти, получили слаба оценка по логическо програмиране. На дейталог това може да бъде направено така:

```
проблем(Група) :- оценка(Студент,2), група(Студент,Група).
?- проблем(Група).
```

Същата заявка на ес кю ел включва следните операции:

- нека t_1 е таблицата, която се получава от `проблем` като отделим само редовете, в които оценката е слаба;
- нека t_2 е таблицата, която се получава от t_1 като изтрием стълба с оценките;
- нека t_3 е таблицата, която се получава като слеем таблица `група` с таблица t_2, приравнявайки стълбовете със студентите;
- нека t_4 е таблицата, която се получава от t_3 като изтрием стълба със студентите;
- изкарай като отговор таблица t_4.

* * *

Счита се, че не съществува алгоритъм, който винаги успява да решава ефективно задачи, за които е доказано, че са NP-пълни. Едно сравнително скорошно неочаквано откритие е това, че има алгоритми, които макар и не винаги, но в много случаи успяват да решават NP-пълни задачи. Тъй като има много важни за практиката задачи, които са NP-пълни, това откритие има не само теоретична, но и голяма приложна стойност. Реализирани са системи за answer set programming, които в повечето случаи използват езици, които много приличат на пролог.

*
* * *
*

За подобряването на тази глава помогнаха д-р Стефан Вътев, Стефан Герджиков, Соломон Паси, проф. Димитър Скордев, Александра Соскова, проф. Тинко Тинчев и други. Нито един от тях не може по никакъв начин да се счита отговорен за допуснатите грешки. Също така, ни най-малко не може да се счита, че те споделят изцяло или дори част от изказаните в този текст нематематически тези.

Глава 2

Синтаксис

> Фразите на един формален език всъщност не са редици от символи, а абстрактни обекти, означавани с редици от символи, също както естествените числа са абстрактни обекти, означавани с редици от цифри. Следователно да дефинираме семантиките посредством функции върху редици от символи би било също толкова заобиколно, колкото ако дефинирахме аритметичните функции върху редици от цифри.
>
> Джон Рейнолдс [16]

2.1. Формални езици

Синтаксис и семантика

Обичайният език, на който говорят хората е двусмислен и неточен (дори тогава, когато се използва за официални юридически документи). Затова, когато математиците формулират някоя дефиниция или твърдение, те спазват определени от математическата традиция правила. Благодарение на тези неписани правила, езикът на математиката има ясен за всеки математик смисъл.

Има случаи, когато спазването неписани правила е недостатъчно. В тези случаи използваме т.н. формални езици. Съвкупността от правила и принципи, чрез които може да се формулираме правилни форму-

```
function нод(m, n)
    if n = 0
       return m;
    else
       return нод(n, m mod n);
```

Фиг. 1. Примерен псевдокод за алгоритъма на Евклид

```
function НОД(M, N: Natural) return Natural is
begin
   if N = 0 then
      return M;
   else
      return НОД(N, M mod N);
   end if;
end НОД;
```

Фиг. 2. Програма на ада за алгоритъма на Евклид

лировки от даден език, се нарича *синтаксис* на този език. Смисълът пък на така построените правилни формулировки е *семантиката* този език. Езиците, чиито синтаксис и семантика са формулирани точно и недвусмислено, се наричат *формални езици*.

Псевдокодът, който се използва например в книгите за структури данни, е пример за език с точен смисъл, който обаче не е формален език. Макар и да е разбираем за читателя, псевдокодът, няма формално дефинирани синтаксис и семантика. Езиците за програмиране пък са примери за формални езици. Сравнете програмите от фиг. 1 и фиг. 2, които представят рекурсивния вариант на алгоритъма на Евклид съответно на псевдокод (точен математически език) и на ада (формален език). Няма да се случи нищо нередно, ако в края на първия ред на програмата от фиг. 1 добавим двоеточие, защото смисълът на така променения програмен текст остава напълно ясен. Обаче в програмата от фиг. 2 не можем да правим подобни промени, защото езикът ада си има точно определен синтаксис, който не позволява да слагаме двоеточия където поискаме.

Синтаксисът на компютърните формални езици често е сложен. При много езици за програмиране синтаксисът се дефинира с помощта на стотици правила, определящи неща като приоритетите на операциите, къде може и къде не може да слагаме интервали и празни редове и т.н.

Когато искаме да обсъждаме семантиката, т.е. смисълът на нещо, написано на някой формален език, например компютърна програма, тогава би било добре да не се налага да си отвличаме вниманието с ненужни синтактични подробности. Например не искаме да има значение дали сме написали някой аритметичен израз с интервали `x + y` или без интервали `x+y`. Затова се налага да говорим за два вида синтаксис: *конкретен синтаксис* и *абстрактен синтаксис*. Конкретният синтаксис е синтаксисът, който използваме, когато пишем изрази от съответния език. Например една компютърна програма е написана съгласно правилата на конкретния синтаксис на използвания език за програмиране. Абстрактният синтаксис пък използваме тогава, когато искаме да разсъждаваме за смисъла на изразите от даден език, например как точно ще работи дадена програма. Абстрактният синтаксис е много опростен и лесен за математическо използване, без обаче да ограничава възможностите на съответния формален език.

Езици за хора и езици за компютри

Формалните езици се разделят на две големи групи — езици, които са предназначени да бъдат обработвани от компютър, и езици, при които компютърната обработка не е сред основните им цели.

Освен езиците за програмиране, други примери за формални езици от първата група са например езиците за заявки към бази данни, езиците XML и HTML, използвани за уеб-страниците в Интернет и много други. Понякога да формулираме точния синтаксис и семантика на езиците от тази група по начин, разбираем за хората, е непосилна задача. Например до момента никой не е успял да формулира по разбираем начин какъв точно е синтаксисът на езика за програмиране пърл, а се счита, че са правилни онези програми, които се приемат от транслатора.*

Обаче без значение дали конкретният синтаксис на даден език за програмиране има разбираема дефиниция, този синтаксис винаги е фиксиран и точно определен. Например не може да направим компилатор на даден език за програмиране, без да сме определили кои редици от

*Синтаксисът на повечето езици за програмиране обикновено не е толкова сложен и може да се опише по разбираем за хората начин. Но дори един език за програмиране да има прост синтаксис, неговата семантика не рядко се оставя без точна математическа дефиниция. В някои случаи тази неопределеност е съществена. На програмистите им се налага да пишат примерни програми, за да видят как точно ще се изпълни даден код. Разбира се, няма гаранции, че следващите версии на компилатора ще изпълнят този код по същия начин.

символи ще считаме, че са правилни програми от съответния език. Що се касае до абстрактния синтаксис на компютърните формални езици, то той много често изобщо не се дефинира явно. Това е така, защото от абстрактния синтаксис имаме нужда само тогава, когато решим да разсъждаваме математически за семантиката на формалния език.

Втората група формални езици са тези, при които компютърната обработка не е сред основните им цели. Такива са например някои езици за спецификация на програми, както и разнообразните езици, използвани в математическата логика. Синтаксисът и семантиката на езиците от тази група винаги имат точна и разбираема за хората дефиниция. Едно интересно свойство на езиците от тази група е това, че в повечето случаи не е нужно да се грижим синтаксисът им да бъде лесен за използване от хората. Причината за това е следната: тъй като текстовете на тези езици няма да се четат от компютър, а от хора, то значи дори и да не пишем „правилно“ — т.е. точно според синтаксиса на формалния език — читателят, ще може да поправи сам допуснатите „грешки“. От всичко това следва следното: при човешките формални езици винаги дефинираме точно техния абстрактен синтаксис, докато конкретния синтаксис обикновено не се дефинира точно.

2.1.1. Пример. Да разгледаме езика, чийто абстрактен синтаксис се задава от следната безконтекстна граматика (S е нетерминал, който е началният символ на граматиката, а `a`, `b`, `1`, `2`, `(`, `)`, `+` и `*` са терминални символи):

$$S \longrightarrow \texttt{(}S\texttt{+}S\texttt{)} \mid \texttt{(}S\texttt{*}S\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{1} \mid \texttt{2}$$

Един примерен, правилно построен израз от този език е `(a+(2*b))`. Да забележим, че граматиката на този език ни задължава да слагаме скоби около всяка аритметична операция. Благодарение на това не се налага да се усложняваме с приоритети на операциите, неща като лява и дясна асоциативност на операциите и др. Няма никакъв проблем да се уговорим когато използваме този език да изпускаме подразбиращите се скоби. По този начин за удобство вместо `(a+(2*b))` може да пишем просто `a+2*b`. Важно е да знаем обаче, че работим с формален език, имащ точно определен синтаксис, в който аритметичните операции задължително се ограждат със скоби. Това означава, че дори когато сме написали `a+2*b`, ние всъщност имаме предвид `(a+(2*b))`. Например ако в някое математическо разсъждение се налага да преброим броя на символите в израза `a+2*b`, този брой ще бъде 9, а не 5.

В горния пример може да считаме, че дадената граматика дефинира точно абстрактният синтаксис на формалния език. Изрази като

`a+2*b` пък може да считаме че отговарят на конкретния синтаксис на езика, който няма нужда да дефинираме точно, защото езикът не е предназначен за компютри, а за хора.

Има много формални езици, за които не се е предвиждало да се обработват с компютър, но в последствие това се е оказало полезно. В такива случаи компютърната реализация на формалния език използва по-удобен за използване синтаксис в сравнение със синтаксиса от първоначалната математическа дефиниция. С други думи, преди да се появи компютърната реализация, конкретният синтаксис на езика е оставал неформално дефиниран, но след появата на компютърна реализация този синтаксис е станал точно определен и описан в софтуерната документация.

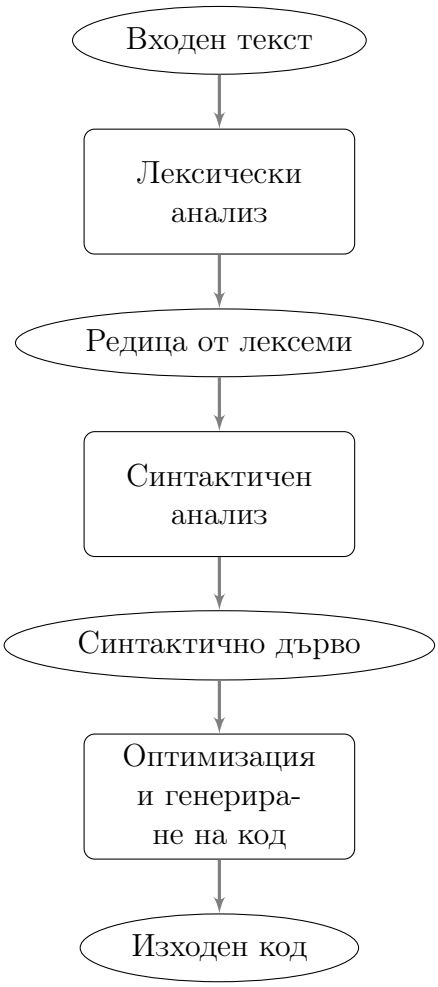
Забележка: Може да направим следните изводи:

- Ако даден формален език е предназначен за компютърна обработка, той със сигурност има точно определен конкретен синтаксис.
- Ако даден формален език не е предназначен за компютърна обработка, той може и да няма точно определен конкретен синтаксис. Например може по напълно неформален начин „да се уговорим“ да изпускаме някои от скобите или да добавяме интервали за по-добра четливост.
- Ако искаме да правим точни математически разсъждения за изразите от даден формален език, то имаме нужда от точно определен абстрактен синтаксис.
- Оказва се, че когато пишем компилатор на даден език също има нужда да дефинираме точно абстрактния синтаксис на езика. Това всъщност не е изненадващо, тъй като точната интерпретация на програмата е свързана с математически преобразувания, на които се основава коректността на компилатора.

За да се научим да дефинираме по хубав начин абстрактния синтаксис на даден формален език, напр. език за програмиране, нека видим как всъщност работят транслаторите на езиците за програмиране.

Синтактични дървета

Транслаторът е *програма*, която преобразува програма от един формален компютърен език (т.н. *входен език*) на друг формален компютърен език (т.н. *изходен език*). Когато входният език е от високо ниво (напр. ада, си, джава), а изходният — от ниско (машинен език или байт-код), транслаторът се нарича *компилатор*. Когато пък входният



Фиг. 3. Обичаен строеж на транслаторите

език е от ниско ниво, а изходният — машинен език, тогава транслаторът се нарича *асемблер*. Има и транслатори, които превеждат от един език от високо ниво на друг език от високо ниво. Такива транслатори се наричат *конвертори*.

Обикновено транслацията се извършва на няколко стъпки, както това е показано във фигура 3.

Първата стъпка от процеса на транслацията се извършва от т.н. лексически анализатор. На входа му постъпва програмният текст под формата на редица от символи (напр. илюстрирания във фигура 2). След като бъде обработена от лексическия анализатор, програмата се превръща в редица от лексеми. Примерна редица от лексеми, отговаряща

function | нод | (| m | , | n | : | natural |) | return | natural | is | begin | if
n | = | 0 | then | return | m | ; | else | return | нод | (| n | , | m | mod | n
) | ; | end | if | ; | end | нод | ;

Фиг. 4. Редица от лексеми за програмата на ада от фиг. 2

на програмата от фиг. 2, е показана във фигура 4.* Както може да се забележи, всяка лексема представлява или дума (напр. идентификатора `Natural` и служебната дума `return`), или литерал (число, низ, и др.), или разделител, или оператор, или коментар. Като основна техника за реализиране на лексическия анализатор се използват крайни автомати. Благодарение на това алгоритмите, използвани от лексическия анализатор са много бързи.**

След като от лексическия анализ получим редица от лексеми, преминаваме към втората стъпка от транслацията — синтактичния анализ. Целта на синтактичния анализатор е да се получи т.н. синтактично дърво. Като основна техника за реализиране на синтактичния анализатор се използват безконтекстни граматики от специален вид. ***

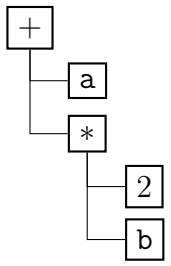
Смисълът на синтактичното дърво е да изрази ясно структурата на израза във вид, удобен за последваща комютърна обработка. Да разгледаме например аритметичния израз „$\mathtt{a}+2*\mathtt{b}$“. Никак не е очевидно как може да се напише компютърна програма, която пресмята стойност-

*В езика ада не се прави разлика между малки и главни букви. Например „Natural“, „natural“ и „NATURAL“ са един и същи идентификатор. Това е причината, поради която в лексемите от фигура 4 са използвани само малки букви.

**По принцип цялата работа на лексическия анализатор може да се свърши и от синтактичния, така че ако не се интересувахме от бързината на компилатора, то изобщо не би имало нужда от лексически анализатор.

***Алгоритмите, които могат да се използват за синтактичен разбор на произволна безконтекстна граматика са неефективни. Например алгоритъмът на Кок–Янгър–Касами за разбор на безконтекстна граматика, приведена в нормална форма на Чомски, който обикновено се учи в курсовете по дискретна математика, дискретни структури и езици, автомати и изчислимост във ФМИ, е с кубична сложност. Безконтекстните граматики на повечето езици за програмиране обаче имат специални свойства, които позволяват разборът да се извършва за линейно време (макар че дори и в този случай синтактичният анализ е значително по-бавен от лексическия).

Както е известно, крайните автомати разпознават по-тесен клас езици от безконтекстните граматики. Поради това не е възможно да елиминираме нуждата от безконтекстни граматики и да разчитаме изцяло на бързите алгоритми от лексическия анализатор. Форт е един извънредно рядък пример за език от високо ниво, който обаче има регулярна граматика и поради това може да се анализира без да се използват безконтекстни граматики.



Фиг. 5. Синтактично дърво на `a` + 2 ∗ `b`

та на такъв израз. Как например тази програма ще определи с какво трябва да съберем `a` — с 2 или с 2∗`b`? Само от израза не можем да отговорим на този въпрос, защото от него не личи какъв е приоритетът на различните операции. Нека разгледаме обаче съответното синтактично дърво от фигура 5. Това дърво не оставя никакви съмнения, че `a` се събира с 2 ∗ `b`, а рекурсивният му вид прави алгоритмичното пресмятане на аритметичния израз сравнително проста задача.

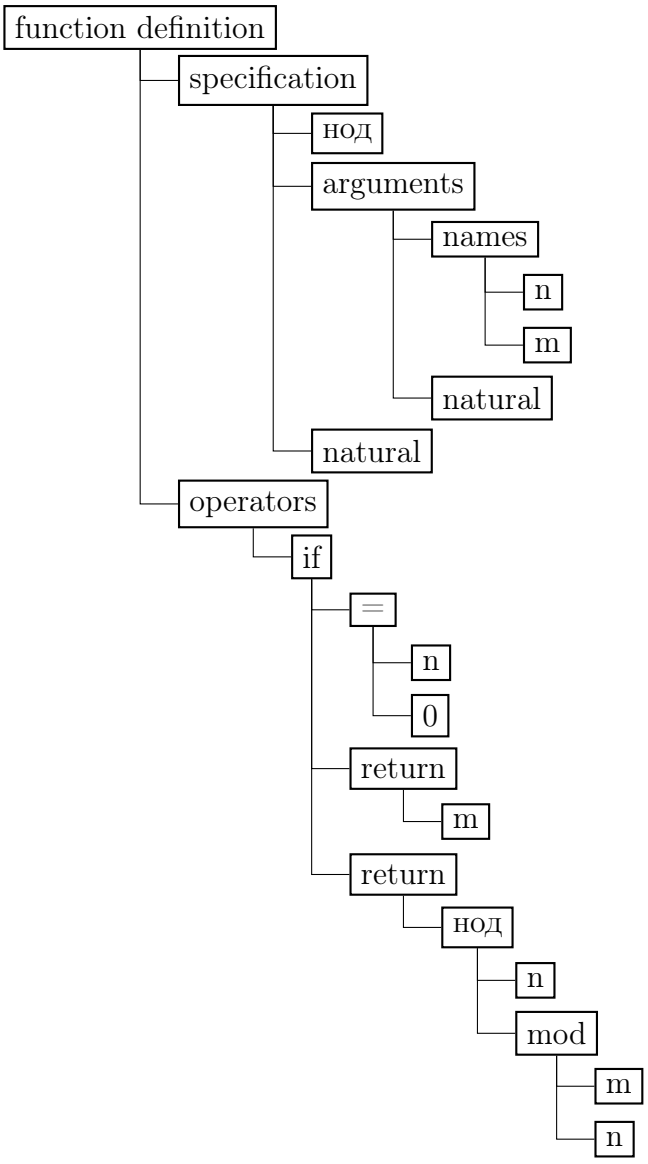
Във фигура 6 е показано примерно синтактично дърво за програмата от фигура 2.

***Задача* 1:** Напишете синтактични дървета за следните изрази на езика си: `x+(y+z)` и `(x+y)+z`. Обърнете внимание, че в си операцията `+` е лявоасоциативна, така че изразите `(x+y)+z` и `x+y+z` имат едно и също синтактично дърво. Същото синтактично дърво имат и изразите `((x+y))+z` и `(x)+y+z`.

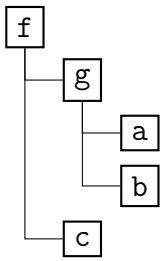
***Задача* 2:** Напишете синтактични дървета за следните изрази на езика си: `x+(++y)`, `(x++)+y` и `x+++y`.

Видове формални записи

В предния подраздел видяхме, че компилаторите преобразуват първоначалния текст на програмата в синтактични дървета. Последващата обработка на синтактичните дървета е много по-лесна и естествена, отколкото ако компилаторът трябваше да работи директно върху първоначалния програмен текст. Синтактичните дървета не само имат по-ясен смисъл, който не зависи например от приоритетите на операциите, но освен това ни спестяват и други ненужни подробности. Например изрази като `a+2*b` и `a+  ((2) *b)`, които очевидно казват едно и също нещо, ще получат едно и също синтактично дърво — дървото от фигура 5.



Фиг. 6. Примерно синтактично дърво за програмата от фиг. 2



Фиг. 7. Синтактично дърво за израза $\texttt{f}(\texttt{g}(\texttt{a}, \texttt{b}), \texttt{c})$

Същото обаче важи и за математическите разсъждения — много полесно е да разсъждаваме върху синтактичното дърво на програмата, отколкото върху първоначалната редица от символи. Това ни дава и ключа за правилното дефиниране на абстрактния синтаксис на даден формален език. Абстрактният синтаксис трябва така да бъде дефиниран, че връзката между изразите, отговарящи на този синтаксис, и съответните синтактични дървета да бъде възможно най-естествена и непосредствена.

Ще опишем някои от по-често използваните начини за дефиниране на абстрактния синтаксис.

2.1.2. Пример (инфиксен запис). Да си припомним езика от пример 2.1.1, чийто абстрактен синтаксис се задаваше от следната безконтекстна граматика (S е нетерминал, който е началният символ на граматиката, а `a`, `b`, `1`, `2`, `(`, `)`, `+` и `*` са терминални символи):

$$S \longrightarrow \texttt{(}S\texttt{+}S\texttt{)} \mid \texttt{(}S\texttt{*}S\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{1} \mid \texttt{2}$$

Граматиката на този език ни задължава да слагаме скоби около всяка аритметична операция. Благодарение на това не се налага да се усложняваме с приоритети на операциите, неща като лява и дясна асоциативност на операциите и др. Един примерен израз от този език е `(a+(2*b))`.

Въпреки че формалният инфиксен запис ни задължава да слагаме повече скоби, отколкото обикновено правим, той има това предимство, че е лесен за четене от хора. Недостатък пък е това, че не може да го използваме за произволни операции. Например не е ясно как можем да използваме инфиксен запис за израза, чието синтактично дърво е илюстрирано във фигура 7.

2.1.3. Пример (полски запис). Както вече бе отбелязано, често при формалните езици е важно те да имат прост синтаксис, който да позволява лесно да доказваме свойствата на тези езици, дори когато точното

спазване на този синтаксис е неудобно. През 1924 полският математик Ян Лукашевич измислил т.н. *полски запис*, при който изобщо не се налага да пишем скоби, не се налага да говорим за приоритет и асоциативност на операциите, а свойствата на формалните езици, използващи полски запис, се доказват по-лесно, отколкото при използване на запис със скоби. При полския запис първо записваме операцията, а веднага след нея, без скоби или каквито и да е други разделители, аргументите ѝ. Например изразът $x+y$ в полски запис изглежда така: $+xy$. Изразът `(a+(2*b))` в полски запис изглежда така: `+a*2b`. Сравнете това с израза `((a+2)*b)`, чийто полски запис изглежда така: `*+a2b`.

Въпреки че полският запис опростява формалните разсъждения, той има един съществен недостатък — обикновено хората не записват изразите по този начин. Затова в днешно време този запис се използва рядко в математическата логика. Вместо това се използват формални езици, при които се поставят всички скоби както в пример 2.1.2 и след това се уговаряме, че ще изпускаме скобите, когато няма опасност от недоразумения. При записа със скоби става малко по-трудно да докажем, че за всеки израз има едно единствено синтактично дърво.

2.1.4. Пример (обратен полски запис). Ако записваме операции не преди, а след операндите, получаваме т.н. *обратен полски запис*. Изразът `(a+(2*b))` в обратен полски запис става `a2b*+`, а изразът `((a+2)*b)` става `a2+b*`. Обратният полски запис много рядко се използва за формални езици в математическата логика, но за сметка на това има много приложения в информатиката. Обратният полски запис е открит първо от Артър Бъркс, Дан Уорън и Джеси Райт [1] през 1954 г., след което е бил преоткриван и от други математици и информатици.

2.1.5. Пример (функционален запис). Обикновено в математиката прилагането на някоя функция към аргументи се записва по следния начин: $f(a_1, a_2, \ldots, a_n)$. С други думи, най-напред записваме името на функцията, след това лява скоба, след това аргументите на функцията, разделени със запетая, и накрая дясна скоба. Този запис може да наречем *функционален запис*. Изразът `(a+(2*b))` при функционален запис става `+(a,*(2,b))`, а изразът `((a+2)*b)` става `*(+(a,2),b)`.

Забележка: Ако от един израз, записан във функционален запис, отстраним всички скоби и запетаи, получаваме израз в полски запис. Например от `+(a,*(2,b))` получаваме `+a*2b`.

Ако ни е дадено някое синтактично дърво, не е трудно да напишем израз във функционален запис, който има това синтактично дър-

во. Например на синтактичното дърво от фигура 7 отговаря изразът $\mathtt{f}(\mathtt{g}(\mathtt{a},\mathtt{b}),\mathtt{c})$, на синтактичното дърво от фигура 5 отговаря изразът $+(\mathtt{a},*(2,\mathtt{b}))$, а на голямото синтактично дърво от фигура 6 отговаря следният израз (който очевидно няма как да поберем само на един ред):

```
function_definition(specification(нод,
                                  arguments(names(n,m),natural),
                                  natural),
                    operators(if(=(n,0),
                                 return(n),
                                 return(нод(n,mod(m,n)))
```

2.2. Съждителна логика

Исторически бележки

Обществото на Древна Гърция е интересно явление. В него учените се считали за едни от най-видните членове на обществото. Всички се вълнували от откритията им. И най-обикновените хора развълнувано обсъждали споровете им по пазарите. „Жълтите“ журналисти разпространявали клюки какво се случило с този или онзи учен. Държавата ги увековечавала със скулптури.*

Именно в Древна Гърция — място, където научните изследвания се правели не за печалба, а от любов към знанието — се появили и първите логически изследвания. Безспорен основател на логиката като научна дисциплина е Аристотел. Той систематизирал изследванията на предшествениците си и създал т.н. *силогистика* — сравнително прост от съвременна гледна точка логически език. Аристотел изследвал какъв е точният смисъл на съжденията, които можем да изкажем, използвайки езика на силогистиката, както и начините за доказателство на такива съждения. По времето на Аристотел обаче все още никой не бил изследвал точния смисъл на обичайните съждителни логически операции като „и“, „или“, „ако . . . , то“.

Оказало се, че импликацията е една от най-проблемните логически операции. Когато в обикновен разговор направим изявлението „ако вали дъжд, то времето е облачно“, с това ние казваме, че има някак-

*След като Платон измислил утопична теория за това каква трябва да бъде идеалната държава, оказало се, че не било чак толкова трудно да намери малко царство, където да реализира на практика представите си за идеалната държава. Експериментът завършил зле както за Платон, така и за злополучния местен цар — и двамата били изгонени позорно от разбунтуваните граждани.

ва причинно-следствена връзка между валенето на дъжд и облачното време. Никога в обикновен разговор няма да чуем твърдение от вида „ако Витоша е планина, то София е град“ или „ако репичката е синя, то Пенчо обича Марийка“, защото не се вижда да има някаква връзка между това Витоша да бъде планина, а София — град, нито между цвета на репичката и любовта на Пенчо. Оказва се обаче, че ако искаме да разсъждаваме правилно, се налага да влагаме в импликацията по-различен смисъл, отколкото при ежедневното общуване. Ако съждението B е вярно, то и импликацията „ако A, то B“ е вярна без значение какво е съждението A. Тази импликация е вярна също и тогава, когато е невярно съждението A, без значение какво е съждението B.

Това че при обичаен разговор ние използваме един вид импликация, а при точни разсъждения — друг, не било забелязано веднага и отначало водело до грешки. Например Аристотел неправилно считал, че от едно съждение A никога не може да следва неговото отрицание $\neg A$. Главна заслуга за разкриване на смисъла на импликацията има т.н. Мегарска школа. Именно в тази школа Диодор Крон е дал първата широко възприета в античността дефиниция на смисъла на импликацията. А неговият ученик Филон Мегарски е дал дефиниция, която по същество съвпада с дефиницията, която използваме в съвременната съждителна логика.

Въз основа на всички тези изследвания, около сто години след Филон, в една друга школа — тази на стоиците — Хризип създал съждителната логика.* Хризип е написал над 700 научни съчинения, като почти половината от тях са посветени на логиката. Заслугите му били общопризнати в античността. Така например древният биограф Диоген Лаертски казва, че „ако боговете използваха логика, то те щяха да използват не някоя друга, а логиката на Хризип“. А през втори век християнският писател Климент Александрийски го нарича „майсторът на логиците, също както Омир е майсторът на поетите“. За съжаление през средновековието постиженията на Хризип не се разбирали и били забравени. От всичките му логически съчинения до наши дни не е оцеляло нищо, така че за тях се налага да съдим главно въз основа на цитати у по-късни автори.

*Всъщност Хризип създал нещо повече от съждителната логика — логиката на Хризип е безкванторният фрагмент на предикатната логика без функционални символи.

Изречения, изказвания, съждения, твърдения

Когато говорим използваме *изречения*. За някои изречения е безсмислено да казваме, че са верни или неверни, например „О, мило мое логическо програмиране!“. Изреченията, за които това не е безсмислено, се наричат *изказвания*. Ето пример за изказвания на четири различни езици (български, английски, полски и сръбски):

> Импликацията е невярна, ако предпоставката ѝ е вярна, а заключението ѝ — невярно; във всички други случаи импликацията е вярна.
>
> The implication is false if its antecedent is true and its consequent is false; in all other cases the implication is true.
>
> Implikacja jest więc fałszywa, jeśli jej poprzednik jest prawdziwy, a następnik fałszywy; we wszystkich innych przypadkach implikacja jest prawdziwa.
>
> Импликација је лажна ако њен антецедент је истина, а њен консеквент је лажан; у свим осталим случајевима импликација је истина.

Въпреки че горните четири изказвания са изказани на различни езици, може да забележим (ако знаем съответните езици), че мисълта, която се съдържа в тях, е една и съща. Мисълта, която се съдържа в едно изказване, се нарича *съждение*; от тук идва и името на логиката — съждителна логика. Когато пък за едно съждение заявим, че е вярно, това наше заявление се нарича *твърдение*.

На разликите между изречение, изказване, съждение и твърдение се набляга във философската логика.[*] В математически контекст обаче тези разлики не са важни. Много често когато математици използват думата „твърдение“, те всъщност имат предвид това, което философите наричат „съждение“, освен в случаите, когато те са формулирали официално твърдението като нещо, за което са заявили, че е вярно.

Булеви функции

Ако пренебрегнем философските въпроси, свързани със *съждителната логика*, то може да считаме, че тя представлява формален език за изразите, които използваме, когато дефинираме булеви функции. Да припомним, че *булева функция* означава, функция, чиито аргументи са елементи на множеството $\{0, 1\}$ и стойността ѝ също е елемент на $\{0, 1\}$.

[*] Съответните английски термини са sentence, statement, proposition, assertion.

	отрицание
	„не“
p	$\neg p$
	$\sim p$
0	1
1	0

Таблица 1. Едноместна съждителна операция

		конюнкция	дизюнкция	импликация	еквиваленция
		„и“	„или“	„ако . . . , то . . .“	„тогава и само тогава, когато“
p	r	$p \,\&\, r$	$p \vee r$	$p \Rightarrow r$	$p \Leftrightarrow r$
		$p \wedge r$		$p \rightarrow r$	$p \leftrightarrow r$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Таблица 2. Двуместни съждителни операции

В математическата логика, и в частност в съждителната логика, се използват малко по-различни означения за съждителните операции в сравнение с означенията, използвани в дискретната математика. Например отрицанието най-често се отбелязва посредством символите $\neg$ или $\sim$, а не с хоризонтална черта над израза (вж. таблица 1). Използваните означения за двуместните съждителни операции пък са дадени в таблица 2. Забележете, че конюнкцията се отбелязва посредством & или $\wedge$, а не като умножение.

В математическата логика съждителните операции имат следните приоритети:

– с най-висок приоритет е отрицанието $\neg$;

– със следващ по сила приоритет са конюнкцията & и дизюнкцията $\vee$;

– с най-слаб приоритет са импликацията $\Rightarrow$ и еквиваленцията $\Leftrightarrow$.

Например $p \Rightarrow \neg q_1 \vee q_2$ е същото като $p \Rightarrow ((\neg q_1) \vee q_2)$. Да забележим, че операциите & и $\vee$ са с един и същ приоритет, така че винаги трябва да използваме скоби, за да уточним коя от тях се извършва първо.*

*Забележете, че това не е така в дискретната математика, където конюнкцията

Аналогично е положението и при операциите $\Rightarrow$ и $\Leftrightarrow$.

Ето някои примерни булеви функции, дефинирани с помощта на означенията, използвани в логиката:

$$\begin{aligned} f_1(p,r) &= p \,\&\, \neg r \\ f_2(p,r) &= (p \,\&\, \neg r) \vee (\neg p \,\&\, r) \\ f_3(p,r,q) &= \neg(p \vee r \vee q) \end{aligned}$$

Да забележим, че съответствието между булеви функции и изрази не е взаимно еднозначно. Така например изразът $p \vee r$ може да се използва за дефинирането на следните различни по между си булеви функции:

$$\begin{aligned} f(p,r) &= p \vee r \\ g(r,p) &= p \vee r \\ h(p,r,q) &= p \vee r \end{aligned}$$

Както се вижда от дефиницията на функцията h, възможността да се използват фиктивни аргументи означава, че един израз може да се използва за дефиниране на безброй много булеви функции. Така например следващият списък съдържа само някои от безброй многото функции, които можем да дефираме с израза $p \vee r$:

$$\begin{aligned} h_1(p,r,q_1) &= p \vee r \\ h_2(p,r,q_1,q_2) &= p \vee r \\ h_3(p,r,q_1,q_2,q_3) &= p \vee r \\ h_4(p,r,q_1,q_2,q_3,q_4) &= p \vee r \\ h_5(p,r,q_1,q_2,q_3,q_4,q_5) &= p \vee r \\ &\dots \end{aligned}$$

Синтаксис на съждителната логика

По традиция, правилно построените изрази от някоя логика, вкл. от съждителната логика, се наричат *формули*. Това означава, че в математическата логика и логическото програмиране думата „формула“ има по-различен смисъл, отколкото в другите дялове на математиката.

обикновено се обозначава като умножение и е с по-голям приоритет от дизюнкцията. Например в дискретната математика $p \vee q_1 q_2$ е същото като $p \vee (q_1 q_2)$, докато в логиката изразът $p \vee q_1 \,\&\, q_2$ е двусмислен.

Съждителната логика представлява формален език за изразите, които използваме, когато дефинираме булеви функции. Следователно формулите от съждителната логика представляват точно дефинирани низове от символи.* Добре е да обърнем по-специално внимание на това свойство на формулите. Да попитаме от колко символа е съставен неформалният израз $(p \,\&\, \neg r) \vee (\neg p \,\&\, r)$ е безсмислено, защото това е неформален израз, а не низ от символи. Формулите обаче, както и изобщо правилно построените изрази на кой да е формален език, са редици от символи и затова за тях не е безсмислено да се запитаме от колко символа са образувани.

Символите, които ще използваме като аналози на променливите p, r, q и т.н., ще наречем съждителни променливи. Преди да дадем дефиницията на формула от съждителната логика, трябва да определим кои точно символи ще използваме като съждителни променливи. За целта ще дефинираме понятието „сигнатура за съждителната логика“:

2.2.1. Дефиниция. а) *Сигнатурата за съждителната логика* е безкрайно множество от символи, което не съдържа никой от символите (,), $\neg$, $\&$, $\vee$ и $\Rightarrow$.

б) Нека $\mathfrak{sig}$ е сигнатура за съждителната логика. Елементите на $\mathfrak{sig}$ се наричат *съждителни променливи* при сигнатура $\mathfrak{sig}$.

Вече сме готови да дадем дефиницията на съждителна формула. Ако използваме инфиксен запис (вж. пример 2.1.2), то тази дефиниция може да изглежда например по следния начин:

2.2.2. Дефиниция. Нека $\mathfrak{sig}$ е сигнатура за съждителната логика. Съждителните формули при сигнатура $\mathfrak{sig}$ се дефинират индуктивно посредством следните три правила:

а) Ако $\mathtt{p}$ е съждителна променлива от сигнатурата $\mathfrak{sig}$, то $\mathtt{p}$ е съждителна формула при сигнатура $\mathfrak{sig}$;

б) ако φ е съждителна формула при сигнатура $\mathfrak{sig}$, то низът $\neg\varphi$ е съждителна формула при сигнатура $\mathfrak{sig}$;

*Математиците обикновено говорят за думи, съставени от букви, вместо за низове от символи. Гьоте е казал, че математиците са като французите — ти им кажеш едно нещо, а те си го превеждат на собствения език и в крайна сметка се получава нещо съвсем различно. Например когато един дискретен математик говори за „букви“, той всъщност си мисли за произволни символи (напр. скоби или символи за логически операции). Когато същият математик говори за „думи“, той всъщност си мисли за низове и затова не се плаши от филологически абсурди като „празната дума“.

в) ако φ и ψ са съждителни формули при сигнатура $\mathfrak{sig}$, то низовете $(\varphi\&\psi)$, $(\varphi\vee\psi)$ и $(\varphi\Rightarrow\psi)$ са съждителни формули при сигнатура $\mathfrak{sig}$.

Да отбележим, че в така дадената дефиниция не се казва, че ако φ и ψ са формули, то низът $(\varphi \Leftrightarrow \psi)$ е съждителна формула. Направихме това умишлено, за да илюстрираме нещо, което логиците често правят — в дефиницията на формула те споменават само част от логическите операции, а за останалите приемат, че са съкращение. Например може да се уговорим, че когато пишем израз от вида $(\varphi \Leftrightarrow \psi)$, ние всъщност имаме предвид $((\varphi \Rightarrow \psi)\,\&\,(\psi \Rightarrow \varphi))$. По този начин, без да намаляваме изразителната сила на съждителната логика, ние можем да опростим формулировките на някои дефиниции, както и много от доказателствата на твърденията. Всъщност можеше още повече да опростим тази дефиниция като споменем в нея само две операции, например само операциите отрицание и конюнкция, защото всяка булева функция може да се изрази, използвайки единствено отрицание и конюнкция.*

***Задача* 3:** Нека $\mathtt{p}$ е съждителна променлива при използваната сигнатура. Колко символа съдържа формулата $(\mathtt{p} \Leftrightarrow \mathtt{p})$?

Когато записваме формули, може да се уговорим да пропускаме ненужните скоби, използвайки обичайния приоритет на съждителните операции.

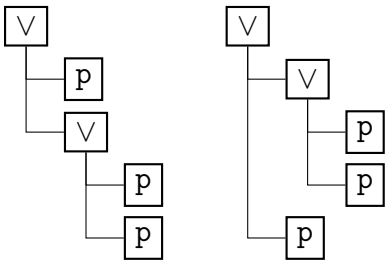
***Задача* 4:** Нека $\mathtt{p}$, $\mathtt{q}$ и $\mathtt{r}$ са съждителни променливи. Намерете точния вид на следните формули, както и синтактичните им дървета: $\mathtt{p}\Rightarrow\mathtt{r}\,\&\,\mathtt{q}$, $\mathtt{p}\vee\mathtt{r}\Rightarrow\mathtt{q}$, $\neg(\neg\mathtt{p}\Rightarrow\neg(\mathtt{q}\,\&\,(\mathtt{r}\vee\mathtt{p}))\,\&\,\neg\mathtt{r})$.

Дефиницията на съждителна формула е дадена по такъв начин, че да направи вярно следното твърдение:

2.2.3. Твърдение за еднозначен синтактичен разбор. *Всяка съждителна формула притежава единствено синтактично дърво.*

От еднозначността на синтактичния разбор следва например, че никоя съждителна формула не може да бъде едновременно от вида $(\varphi_1\,\&\,\varphi_2)$ и $(\psi_1\vee\psi_2)$, нито пък едновременно от вида $(\varphi_1\Rightarrow\varphi_2)$ и $\neg\psi$. Също така от тук следва например, че една съждителна формула може да бъде едновременно от вида $(\varphi_1\vee\varphi_2)$ и $(\psi_1\vee\psi_2)$ само когато $\varphi_1=\psi_1$ и $\varphi_2=\psi_2$.

*Малко по-нестандартно решение би било да използваме само стрелката на Пирс или само чертата на Шефер. В този случай можеше да минем с една единствена съждителна операция.



Фиг. 8. Две синтактични дървета за израза $\mathtt{p} \vee \mathtt{p} \vee \mathtt{p}$

Не е проблем в дефиниция 2.2.2 вместо инфиксен запис да се използва и някой друг запис. Например

- за да получим съждителни формули в полски запис* (вж. пример 2.1.3), трябва да кажем че ако φ е формула, то $\neg\varphi$ е формула и ако φ и ψ са формули, то $\&\varphi\psi$, $\vee\varphi\psi$ и $\Rightarrow\varphi\psi$ са формули;
- за да получим съждителни формули в обратен полски запис (вж. пример 2.1.4), трябва да кажем че ако φ е формула, то $\varphi\neg$ е формула и ако φ и ψ са формули, то $\varphi\psi\&$, $\varphi\psi\vee$ и $\varphi\psi\Rightarrow$ са формули;
- за да получим съждителни формули във функционален запис (вж. пример 2.1.5), трябва да кажем че ако φ е формула, то $\neg(\varphi)$ е формула и ако φ и ψ са формули, то $\&(\varphi, \psi)$, $\vee(\varphi, \psi)$ и $\Rightarrow(\varphi, \psi)$ са формули.

Обаче без значение кой точно запис решим да използваме, той задължително трябва да бъде такъв, че да осигурява еднозначен прочит на съждителните формули. Не е трудно с малки промени в дефиниция 2.2.2 да направим така, че твърдението за еднозначен синтактичен разбор да не е вярно. Нека например дефиницията казва, че ако φ и ψ са съждителни формули, то $\varphi\vee\psi$ също е съждителна формула. В такъв случай ако $\mathtt{p}$ е съждителна променлива (при използваната сигнатура), то $\mathtt{p} \vee \mathtt{p} \vee \mathtt{p}$ ще се окаже съждителна формула. Последният израз обаче няма еднозначен прочит, защото притежава две различни синтактични дървета, илюстрирани във фигура 8.

Много често логиците формулират твърдението за еднозначен синтактичен разбор, но го оставят без доказателство. За това има две причини.

Първата причина е следната. Когато изследваме някоя логика, и в частност съждителната логика, ние се интересуваме единствено от онези свойства, които по никакъв начин не зависят от това кой точно

*Ян Лукашевич е използвал други означения за съждителните операции в своята версия на полския запис.

запис ще използваме (инфиксен, полски, обратен полски, функционален, някакъв друг). В частност всички формулировки на дефиниции и твърдения, както и разсъжденията, използвани в доказателствата, са такива, че те не се променят съществено, ако решим да сменим използвания запис на съждителните формули. Твърдението за еднозначност на синтактичния разбор обаче не може да се докаже без да се интересуваме от това какъв точно е използваният запис. Това означава, че когато доказва това твърдение, един логик трябва да се интересува от неща, за които обикновено не му се налага да се интересува.* И тъй като логикът не иска да се занимава с неща, с които обикновено не се занимава, той предпочита да си спести доказателството.

Втората причина е следната. Когато един логик остави еднозначността на синтактичния разбор недоказана, той все едно ни казва следното: „Дори да се окаже, че съждителните формули нямат еднозначен синтактичен разбор, защото съм сбъркал дефиницията на съждителна формула, нека считаме че вместо дадената тук дефиниция използваме някоя друга, при която има еднозначност на синтактичния разбор.“ Това не е безсмислено, защото, както вече отбелязахме, в математическата логика дефинициите, твърденията и доказателствата се правят по такъв начин, че те да не зависят от това какъв точно запис на формулите използваме.**

Някои логици дефинират формулите не като редици от символи, а като дървета. По този начин напълно отпада нуждата да доказваме, че всяка формула притежава единствено синтактично дърво, защото всяка формула съвпада със своето синтактично дърво.

Семантика на съждителната логика

След като дефинирахме синтаксиса на съждителната логика, т.е. какво значи един израз да бъде правилно построена съждителна формула, трябва да дефинираме и семантиката на тази логика, т.е. какъв е смисълът на една съждителна формула. Смисълът на съждителната формула φ ще означаваме с $[\![\varphi]\!]$. Без ясно дефинирана семантика, съждителните формули биха били просто низове от символи и нищо повече.

*Например трябва да се интересуваме от поднизовете на една формула, от това кои от тези поднизове завършват с разделител, кои от префиксите на тези поднизове съдържат повече леви скоби, отколкото десни, и други подобни неща.

**Да отбележим, че дадената тук дефиниция 2.2.2 не е сбъркана и не се налага да я сменяме.

Нека $\mathtt{p}$ и $\mathtt{r}$ са съждителни променливи (при използваната сигнатура). В такъв случай изразът $(\mathtt{p} \vee \mathtt{r})$ ще бъде съждителна формула. Да се запитаме какъв е смисълът на този израз.

Не можем просто да кажем, че $[\![(\mathtt{p} \vee \mathtt{r})]\!]$ е функцията дизюнкция, защото трябва да уточним какви са аргументите на тази дизюнкция. Вече видяхме, че има безброй много булеви функции, които можем да дефинираме с израза $(\mathtt{p} \vee \mathtt{r})$ и всяка от тях е в някакъв смисъл дизюнкция. Би могло по някакъв, може би изкуствен начин да изберем една от всички тези безброй много булеви функции и да считаме, че $[\![(\mathtt{p} \vee \mathtt{r})]\!]$ е така избраната булева функция. Много по-просто обаче ще бъде да считаме, че смисълът $[\![\varphi]\!]$ на коя да е съждителна формула φ е функция, която зависи от стойностите на всички съждителни променливи, включително променливите, които не се срещат във φ.

Тъй като $[\![\varphi]\!]$ трябва зависи от стойностите на всички съждителни променливи, ще дефинираме понятието съждителна интерпретация:

2.2.4. Дефиниция. Нека $\mathfrak{sig}$ е сигнатура за съждителната логика. Казваме, че $\mathbf{I}$ е съждителна интерпретация при сигнатура $\mathfrak{sig}$, ако $\mathbf{I}$ е функция, чиято дефиниционна област е множеството на всички съждителни променливи при сигнатура $\mathfrak{sig}$, и $\mathbf{I}(\mathtt{p}) \in \{0, 1\}$ за всяка съждителна променлива $\mathtt{p}$.

Вече сме готови да дефинираме семантиката на съждителната логика:

2.2.5. Дефиниция. Нека $\mathbf{I}$ е коя да е съждителна интерпретация (при използваната сигнатура). *Стойността на съждителната формула* φ при интерпретация $\mathbf{I}$ ще означаваме с $[\![\varphi]\!]^I$ и ще дефинираме по следния начин:

– ако $\mathtt{p}$ е съждителна променлива, то нека

$$[\![\mathtt{p}]\!]^{\mathbf{I}} = \mathbf{I}(\mathtt{p})$$

– за всяка съждителна формула φ, нека

$$[\![\neg\varphi]\!]^{\mathbf{I}} = \neg([\![\varphi]\!]^{\mathbf{I}}) \tag{1}$$

– за всеки две съждителни формули φ и ψ, нека

$$[\![(\varphi \,\&\, \psi)]\!]^{\mathbf{I}} = [\![\varphi]\!]^{\mathbf{I}} \,\&\, [\![\psi]\!]^{\mathbf{I}} \tag{2}$$

$$[\![(\varphi \vee \psi)]\!]^{\mathbf{I}} = [\![\varphi]\!] \vee [\![\psi]\!]^{\mathbf{I}} \tag{3}$$

$$[\![(\varphi \Rightarrow \psi)]\!]^{\mathbf{I}} = [\![\varphi]\!]^{\mathbf{I}} \Rightarrow [\![\psi]\!]^{\mathbf{I}} \tag{4}$$

Обърнете внимание, че смисълът на знаците $\neg$, $\&$, $\vee$ и $\Rightarrow$ отляво и отдясно на равенствата (1)–(4) е много различен. Отляво на равенствата тези знаци са част от изрази, които са съждителни формули. Следователно тук знаците $\neg$, $\&$, $\vee$ и $\Rightarrow$ са просто символи — символи, които могат да бъдат въвеждани от клавиатурата, отпечатвани на компютърния екран и т.н. Отдясно на равенствата обаче тези знаци имат напълно различен смисъл. Тук те не са символи, а математически означения за булевите функции от таблици 1 и 2 на стр. 49.

2.2.6. Пример. Нека $\mathtt{p}$ и $\mathtt{r}$ са съждителни променливи (при използваната сигнатура), а съждителната интерпретация $\mathbf{I}$ е такава, че $\mathbf{I}(\mathtt{p}) = 1$ и $\mathbf{I}(\mathtt{r}) = 0$. Тогава:

$$[\![\mathtt{p}]\!]^{\mathbf{I}} = 1$$
$$[\![\neg\mathtt{p}]\!]^{\mathbf{I}} = 0$$
$$[\![\mathtt{r} \vee \mathtt{p}]\!]^{\mathbf{I}} = 1$$
$$[\![\neg\mathtt{r} \vee \mathtt{p}]\!]^{\mathbf{I}} = 1$$
$$[\![\neg(\mathtt{r} \vee \mathtt{p})]\!]^{\mathbf{I}} = 0$$

2.2.7. Дефиниция. а) Когато $[\![\varphi]\!]^{\mathbf{I}} = 1$, казваме, че съждителната формула е *вярна* при интерпретацията $\mathbf{I}$. Ако пък $[\![\varphi]\!]^{\mathbf{I}} = 0$, то казваме, че формулата е *невярна* при интерпретацията $\mathbf{I}$.

б) Някои съждителни формули са верни при всички интерпретации. Такива съждителни формули се наричат *съждителни тавтологии*.

Пример. Нека $\mathtt{p}$ и $\mathtt{r}$ са съждителни променливи при използваната съждителна сигнатура. Ето някои примерни съждителни тавтологии:

$$\mathtt{p} \,\&\, \mathtt{r} \Leftrightarrow \mathtt{r} \,\&\, \mathtt{p}$$
$$\mathtt{p} \vee \mathtt{p} \Leftrightarrow \mathtt{p}$$
$$\neg(\mathtt{p} \vee \mathtt{r}) \Leftrightarrow \neg\mathtt{p} \,\&\, \neg\mathtt{r}$$

Изводимост

Освен разнообразни логически езици, логиката изучава и принципите за правене на правилни доказателства. По отношение на съждителната логика това означава, че трябва да изследваме в кои случаи ако вече сме доказали, че формулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са верни при някаква интерпретация, от тук ще можем да получим като следствие някоя нова формула ψ.

Изказване от вида

> ако формулите $\varphi_1, \varphi_2, \ldots, \varphi_n$ са верни при някоя съждителна интерпретация, то формулата ψ също е вярна при тази интерпретация

ще наричаме *правило за извод* за съждителната логика.

Не е трудно да формулираме примерно правило за извод. Например ако вече сме доказали, че формулите φ и ψ са верни при някоя интерпратация, ясно е, че от тук ще следва, че формулата $\varphi \,\&\, \psi$ също е вярна. Това правило извод можем да запишем по следния начин:

$$\frac{\varphi \quad \psi}{\varphi \,\&\, \psi}$$

Изобщо използваме запис от вида

$$\frac{\varphi_1 \quad \varphi_2 \quad \ldots \quad \varphi_n}{\psi}$$

за да означим правилото, което казва, че ако вече сме доказали формулите над чертата, то ще можем да получим като следствие и формулата под чертата. Формулите над чертата се наричат *предпоставки* на правилото, а формулата под чертата — негово *заключение*.*

Древногръцкият логик Хризип формулирал следните пет правила за извод, които той наричал „примитивни“ правила.

– Първото примитивно правило на Хризип е:

$$\frac{\varphi \Rightarrow \psi \quad \varphi}{\psi}$$

В съвременната логика това правило се нарича *modus ponens*.

– Второто примитивно правило на Хризип е:

$$\frac{\varphi \Rightarrow \psi \quad \neg\psi}{\neg\varphi}$$

В съвременната логика това правило се нарича *modus tollens*.

– Третото примитивно правило на Хризип е:

$$\frac{\neg(\varphi \,\&\, \psi) \quad \varphi}{\neg\psi}$$

*Вместо „предпоставка“ се използва и терминът „антецедент“, а вместо „заключение“ — „консеквент“ или „сукцедент“.

– Четвъртото примитивно правило на Хризип е:

$$\frac{\neg(\varphi \Leftrightarrow \psi) \quad \neg\varphi}{\psi}$$

– Петото примитивно правило на Хризип е:

$$\frac{\varphi \vee \psi \quad \neg\varphi}{\psi}$$

***Забележка*:** Оригиналното четвърто правило на Хризип било формулирано с помощта на т.н. „изключващо или“. В петото правило също се говорело за изключващото, а не за обикновеното „или“. Обикновената дизюнкция, която използваме в съвременната математика, е била измислена след Хризип.

***Задача* 5:** По кое от правилата на Хризип се извършва всяко едно от следните разсъждения:

1. Ако вали дъжд, то времето е облачно. Но вали дъжд, значи времето е облачно.
2. Ако вали дъжд, то времето е облачно. Но времето не е облачно, значи не вали дъжд.
3. Не може едновременно да вали сняг и да има гръмотевици. Но вали сняг, значи няма гръмотевици.
4. В мача на шахматните претенденти (2006 г.) Камски печели тогава и само тогава, когато Топалов не печели. Но Камски не печели, значи Топалов печели.
5. Ще тичам в парка или ще ходя на планина. Но няма да тичам в парка, значи ще ходя на планина.

Освен петте „примитивни“ правила, Хризип е формулирал и четири начина, наречени от него „теми“, посредством които от петте правила можем да генерираме нови правила за извод. За да установи верността на някое разсъждение, Хризип постъпвал по следния начин: започвал да прилага четирите теми към разсъждението като по този начин го разбивал на все по-примитивни методи за разсъждение. Ако в крайна сметка всичко може да се сведе до петте примитивни правила за извод, значи разсъждението е вярно. Точният вид на четирите теми на Хризип не е известен със сигурност, но може да се реконструира. [6]

В съвременната логика не е обичайно от някакви примитивни правила за извод да получаваме по-сложни правила за извод, както е правел

Хризип. Вместо това се използва точно определен набор правила за извод, които са такива, че да бъдат достатъчни сами по себе си, без да се налага генерираме нови правила за извод. Изследвани са различни видове системи за извод — от Хилбертов тип, Генценова секвенциална изводимост, естествена изводимост, резолютивна изводимост и др.

Ще опишем системите за извод от Хилбертов тип, които имат сравнително най-проста форма. Да наречем правило за извод, което има нула предпоставки, *аксиома*. Такова правило просто казва, че заключението му е винаги вярно. При системите за извод от Хилбертов тип се стремим да имаме само едно правило за извод, което не е аксиома — правилото modus ponens:

$$\frac{\varphi \Rightarrow \psi \quad \varphi}{\psi}$$

Следва примерен списък от аксиоми, които заедно с modus ponens са достатъчни, за да получим всички съждителни тавтологии.

– Аксиоми за импликацията:

$$(\varphi \Rightarrow (\psi \Rightarrow \chi)) \Rightarrow ((\varphi \Rightarrow \psi) \Rightarrow (\varphi \Rightarrow \chi))$$
$$\varphi \Rightarrow (\psi \Rightarrow \varphi)$$
$$((\varphi \Rightarrow \psi) \Rightarrow \varphi) \Rightarrow \varphi$$

– Аксиоми за конюнкцията:

$$\varphi \,\&\, \psi \Rightarrow \varphi$$
$$\varphi \,\&\, \psi \Rightarrow \psi$$
$$\varphi \Rightarrow (\psi \Rightarrow \varphi \,\&\, \psi)$$

– Аксиоми за дизюнкцията:

$$(\varphi \Rightarrow \chi) \Rightarrow ((\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi))$$
$$\varphi \Rightarrow \varphi \vee \psi$$
$$\psi \Rightarrow \varphi \vee \psi$$

– Аксиоми за отрицанието:

$$\neg\varphi \Rightarrow (\varphi \Rightarrow \psi)$$
$$(\varphi \Rightarrow \neg\varphi) \Rightarrow \neg\varphi$$

2.3. Логически запис на прости изказвания на естествен език

Както отбелязахме в предния раздел, съждителната логика представлява формален език за изразите, които използваме при дефиниране на булеви функции. И въпреки че някои прости разсъждения могат да се извършат в рамките на съждителната логика, като цяло изразителната сила на тази логика е доста слаба. Не е лесно да се открие дори и една математическа теорема, която може да се докаже използвайки единствено средствата от съждителната логика. Да видим защо това е така.

В дефиниция 2.2.2 видяхме, че съждителните формули се формират от по-прости съждителни формули, използвайки съждителните операции $\neg$, $\&$, $\vee$ и $\Rightarrow$. А най-простите съждителни формули — тези, които не съдържат нито една съждителна операция — са съждителните променливи. Всяка съждителна формула може да бъде разпадната на съставните си части, като най-малките формули, от които е образувана тя, са съждителните променливи, които се съдържат в тази формула.

По-подобен начин може да анализираме не само формулите, но и изобщо всяко едно съждение. В изказването на всяко съждение може да открием по-прости съждения, които са свързани с логически операции като отрицание, конюнкция, дизюнкция, импликация и др. Само че докато най-простите съждителни формули — съждителните променливи — имат доста тривиална структура (по-точно нямат никаква вътрешна структура), то дори да разпаднем изказването на едно съждение на съставните му части и стигнем до по-прости съждения, които са изказани без да се използва нито една логическа операция, изказванията на тези по-прости съждения ще имат нетривиална вътрешна структура.

Да илюстрираме това с няколко примера на прости съждения, които са изказани без да се използва нито една логическа операция:

1. *Солон царува.*
2. *Сократ е смъртен.*
3. 79 *е четно число.*
4. $9 < 2$
5. *Орфей обича Евридика.*
6. *Елена се страхуваше от Менелай.*
7. *Аристотел преподава на Александър Македонски логика.*

От тези примери се вижда, че изказването на всяко от тези съждения притежава вътрешна структура. В съждителната логика тази

вътрешна структура се тривиализира — на всяко от горните съждения в съждителната логика би съответствала някоя съждителна променлива. Това е и основната причина за неадекватността на съждителната логика като език за записване на математически (и други) съждения.

Ако искаме някой логически език, да може да се използва за записване на твърдения и доказателства, то на този език ще трябва да можем да изразим вътрешната структура на съждения като горните седем. Нека я анализираме.

Първо, да забележим, че всяко едно от тези седем съждения е изказано посредством просто изречение. В традиционните теории за синтаксиса се счита, че простите изречения могат да съдържат части като сказуемо, подлог, допълнение, обстоятелство и др. От своя страна, всяка от тези части на изречението може да има свои собствени подчинени думи — определения. Например в изречението „Солон царува“ думата „царува“ е сказуемо, а „Солон“ — подлог. А в изречението „Аристотел преподава на Александър Македонски логика“ думата „преподава“ е сказуемото, „Аристотел“ — подлог, „логика“ — пряко допълнение и „Александър Македонски“ — непряко допълнение.

В по-модерните теории за синтаксиса се приема, че простото изречение може да съдържа следните части — предикат, аргументи и адюнкти. Сказуемото заедно със сказуемното определение образува предиката. Адюнкти са онези думи, за които има начин те да се премахнат от изречението и да се отделят в ново изречение, което не използва същото сказуемо, без при това да се загуби изказаната информация. Останалите думи (всяка от които може да има собствени определения) са аргументите на предиката.

Да разгледаме следното примерно изречение:

Вчера Петкан разучаваше в общежитието езика пролог.

Тук сказуемото „разучаваше“ е предикатът. Думите „вчера“ и „в общежитието“ са адюнкти, защото могат да се отделят от основното изречение по следния начин:

Петкан разучаваше езика пролог.
Това той правеше вчера в общежитието.

Останалите части на изречението — „Петкан“ и „езика пролог“ — са аргументите на предиката „разучаваше“. Тези думи не могат да се отделят по естествен начин от основното изречение. Например:

Вчера разучаваше в общежитието езика пролог.
Това правеше Петкан.

Или:

Вчера Петкан разучаваше в общежитието.
Това правеше за езика пролог.

Броят на аргументите, които един предикат може да има, граматиците наричат *валентност* на предиката, а математиците — *арност* или *местност* на предиката. В естествените езици този брой зависи единствено от вида на използвания глагол. Например:

– Някои безлични глаголи нямат аргументи. Пример: „съмва се“.
– Непреходните глаголи имат един аргумент — подлогът. Пример: „царува“. Кой царува? Солон.
– При преходните глаголи аргументи са подлогът и допълненията. Пример: „обича“. Кой обича? Орфей. Кого обича? Евридика.
– При някои глаголи аргументите се свързват с разнообразни, но зависещи от конкретния глагол предлози. Пример: „страхува се“. От кого се страхува? От Менелай.*

В естествените езици от какъв тип е даден аргумент се определя по сложен начин въз основа на:

– позицията на аргумента в изречението;
– падежът, в който е аргументът;
– залогът на сказуемото;
– използвания предлог;
– много други.

Например в изречението

Иван ухапа кучето.

думата „Иван“ е хапещият подлог, а „кучето“ — ухапаното допълнение, като това се разбира от позицията на тези думи спрямо сказуемото „ухапа“. Ако искаме да кажем, че кучето хапе, а Иван е ухапан, най-просто е да разменим думите „Иван“ и „кучето“, но това не е единственият начин. Например в разговорния български език можем да използваме енклитичното местоимение „го“ по следния начин:

Иван го ухапа кучето.

*Едно от най-трудните неща, когато се учи чужд език, е да се научи с какви предлози се свързват аргументите на различните глаголи. Въпреки че глаголите в различните езици използват различни предлози, кои точно са тези предлози не можем да разберем, поглеждайки в речника. Нужно е да четем разнообразни текстове и пак няма как да сме сигурни, че сме срещали всички видове аргументи, които може да има даден глагол.

В книжовния български език пък можем да сменим залога на глагола от деятелен на страдателен:

Иван бе ухапан от кучето.

Вижда се, че правилата за определяне на вида на даден аргумент могат да бъдат доста сложни. Освен това тези правила са много различни в различните езици. За да си спестим тези големи усложнения, както и за да имаме логически език, който не е повлиян от това какъв е майчиният ни език, в логиката се използва следният запис за съждения, съставени от предикат и аргументи:

$$\textit{предикат}(\textit{аргумент}_1, \textit{аргумент}_2, \ldots, \textit{аргумент}_n)$$

При този запис от какъв тип е даден аргумент зависи единствено от това на коя позиция е сложен той. Няма нужда от падежи, предлози, глаголни залози и др.

Използвайки този запис, ето как можем да изкажем горните седем съждения:

1. царува(Солон)
2. смъртен(Сократ)
3. четно_число(79)
4. $<(9, 2)$
5. обича(Орфей, Евридика)
6. страхуваше_се(Елена, Менелай)
7. преподава(Аристотел, логика, Александър_Македонски)

Когато такива формулировки се използват в рамките на точно дефиниран логически език, те се наричат *атомарни формули*. Една особеност на атомарните формули, която ги отличава от естествените езици, е това, че при тях не се допускат липсващи аргументи. Това не е проблем, защото можем да „запълним“ липсващия аргумент, използвайки т.н. квантор за съществуване $\exists$. Например изказването

Аристотел преподава логика.

може да се преведе на логически език по следния начин:

$\exists$x преподава(Аристотел, логика, x)

2.3.1. ***Забележка*:** В езика за програмиране пролог няма експлицитни квантори. От дясната страна на клаузите обаче се подразбира квантор за съществуване $\exists$ за всички променливи, които не се срещат никъде другаде. Затова тук не е проблем да имитираме липсващи аргументи. Ако пък искаме да покажем по-явно, че не се интересуваме от някой

аргумент на предикат, пролог ни позволява да използваме символа _ по следния начин:*

```
преподава(аристотел, логика, _)
```

От лявата страна на клаузите обаче, а също и при фактите, пролог подразбира квантор за всеобщност $\forall$. Затова тук е невъзможно да се използват липсващи аргументи. Не е възможно да обясним на пролог какво означава някой да преподава нещо, без при това да обясним също какво означава някой да преподава нещо някому. Например ако използваме клауза от вида

```
преподава(X, логика, _) :- .........
```

това, което ще обясним на пролог с тази клауза, не е при какви условия е вярно, че X преподава логика, а при какви условия е вярно, че X преподава логика всекиму и на всичко.

В много случаи съществителните и прилагателните имена в изречението се държат не като аргументи на предикат, а като отделни предикати. Да разгледаме изказването

Иван намери мимоза.

Тук очевидно думата „Иван“ обозначава конкретно лице, така че това съществително е аргумент на предиката „намери“. Думата „мимоза“ обаче не обозначава конкретна мимоза, която Иван е намерил, а някаква неопределена мимоза. Следователно преводът

намери(Иван, мимоза)

е неправилен. Вместо това трябва да използваме по-сложна формулировка:

$\exists$x (намери(Иван, x) & мимоза(x))

Аналогично се тълкува изказването

Иван намери цъфнала мимоза.

При него всяка от думите „намери“, „цъфнала“ и „мимоза“ е отделен предикат:

$\exists$x (намери(Иван, x) & мимоза(x) & цъфнало(x))

От друга страна, при изказването

Иван намери срамежлива мимоза.

*Думата „аристотел“ е записана с малка буква, защото на пролог думите, започващи с главна буква, са променливи.

думата „срамежлива“ не е отделен предикат, защото словосъчетанието „срамежлива мимоза“ обозначава специален вид мимози, а не мимоза, която се срамува:

$\exists \mathtt{x}$ (намери(Иван, $\mathtt{x}$) & срамежлива_мимоза($\mathtt{x}$))

Ако членуваме словосъчетанието „срамежлива мимоза“ така:

Иван намери срамежливата мимоза.

тогава то ще обозначава някоя конкретна срамежлива мимоза. Затова в този случай това словосъчетание не е отделен предикат, а аргумент на предиката „намери“:

намери(Иван, срамежливата_мимоза)

Членуваните съществителни или прилагателни в множествено число се превеждат с квантор за всеобщност $\forall$. Например изказването

Шерлок Холмс залови престъпниците.

се превежда така:

$\forall \mathtt{x}$ (престъпник($\mathtt{x}$) $\Rightarrow$ залови(Шерлок_Холмс, $\mathtt{x}$))

т.е. все едно казваме „за всяко $\mathtt{x}$, ако $\mathtt{x}$ е престъпник, то $\mathtt{x}$ бе заловен от Шерлок Холмс.“

Да обобщим казаното със следните правила:

– Нечленувани съществителни и прилагателни в единствено или множествено число* се превеждат с квантор за съществуване $\exists$. Пример: „Учен намери *срамежлива мимоза*“ — някой учен намерил някаква срамежлива мимоза.

– Съществителни собствени, както и членувани прилагателни и съществителни в единствено число са аргументи на предикат. Пример: „Иван намери срамежливата мимоза“.

– Членувани съществителни и прилагателни в множествено число се превеждат с квантор за всеобщност $\forall$. Пример: „Срамежливите мимози са мимози“, т.е. всички представители на вида „срамежлива мимоза“ спадат към рода „мимоза“ (на семейство Fabaceae, подсемейство Mimosaceae).

Във всички разгледани примери аргументите на предикатитите се оказваха или конкретни обекти (Сократ, Евридика, Шерлок_Холмс), или променливи ($\mathtt{x}, \mathtt{y}, \mathtt{z}$). Има случаи, когато това не е достатъчно и се

*Въпреки множественото число, обикновено логическият смисъл на изказването „*Иван намери цъфнали мимози*“ е същият, както ако изречението бе формулирано с единствено число.

налага да използваме по-сложни изрази, наречени *термове*.* Да разгледаме следното изказване:

Фредерик Кайо откри столицата на Нубия.

Тук словосъчетанието „столицата на Нубия“ е аргументът на предиката „откри“. Това словосъчетание логиците превеждат с помощта на следния терм:

столицата_на(Нубия)

Следователно тук използваме израза „столицата на“ сякаш е функция, която има за аргумент „Нубия“, и връща като стойност столицата на Нубия. Затова горното изказване може да се преведе по следния начин:

открИ(Фредерик_Кайо, столицата_на(Нубия))

Ето пример с по-сложен терм. Изказването

Баща на баща е дядо.

може да се преведе така:

$\forall$x дядо(бащата_на(бащата_на(x)), x)

2.4. Термове

Уводни бележки

В предния раздел видяхме, че за да може на един логически език да формулираме разнообразни съждения, в този език задължително трябва да има атомарни формули, например

царува(Солон)

Освен това желателно е в този език да има и сложни термове, например

бащата_на(бащата_на(x))

Преди да видим каква трябва да бъде точната математическа дефиниция на понятията *терм* и *атомарна формула*, нека обърнем внимание, че математическите обекти винаги представляват идеализации, които притежават само онези свойства, които са нужни на математиците. Например точките в геометрията имат точно определено местоположение, но нямат нито дължина, нито ширина, нито височина, нито ориентация. Разбира се, в реалния свят няма обекти без размер. Въпреки това, има много ситуации, при които се интересуваме единствено

*По-простите изрази като „Сократ“ и променливата „x“ също се считат за вид термове.

от местоположението на дадени обекти, но не и от техните размери или ориентация. В тези случаи е удобно да считаме, че тези обекти са точки.

Нека забележим, че е погрешно да считаме, че размерите на обектите, отъждествявани с точки, са пренебрежимо малки. Например в определени ситуации един астроном може да счита, че планетите и звездите са точки в пространството. А в други ситуации не можем да приемем за точкови дори частици като протоните и неутроните.[*] Това, че считаме дадени обекти за точки, няма никакво отношение към размерите на тези обекти, а означава само едно — че не се интересуваме от техните размери и ориентация, а само от местоположението им.

Всъщност оказва се, че геометрията може да съществува и без в нея да се използват точки.[**] При това се оказва, че в безточковите геометрии можем да правим съвсем същите неща, които и в обикновените геометрии. При безточковите геометрии обаче се работи по-сложно, защото когато искаме да обозначаваме позиции в пространството не можем за целта да използваме просто точки, а се налага да правим това по по-заобиколен начин.[***]

Нека сега видим кои са ценните математически свойства на термовете и атомарните формули, т.е. кои са свойствата, които е удобно да бъдат взети под внимание в тяхната дефиниция.

Преди всичко да забележим, че в атомарните формули и термовете от предния раздел използвахме низове като „срамежлива_мимоза“, „преподава“, „Шерлок_Холмс“, „столицата_на“. В нито един случай

[*]Например при експерименти с дълбочинно нееластично разсейване се проявяват трите кварка, от които са образувани тези частици.

[**]Теорията на безточковите геометрии е разработвана и от много български математици — Николай Белухов, Димитър Вакарелов, Георги Димов, Татяна Иванова, Владислав Ненчев, Тинко Тинчев.

[***]Квантовата механика ни кара да се запитаме има ли всъщност точки в пространството и не е ли всъщност геометрията на реалния свят безточкова. Засега обаче никой не е разработил безточкова геометрия, която да съответства на света квантовата механика. Всички разработени до момента безточкови геометрии са по един или друг начин еквивалентни на обикновената евклидова геометрия. Регионите, с които се работи в тези геометрии, имат точно определени граници, докато обектите в квантовата механика не притежават точно определени размери, координати и скорост. Тази неопределеност не се дължи на ограничените ни познания, а е фундаментална характеристика на света. Съгласно една теорема на Джон Стюард Бел, ако допълним квантовата механика с допълнителни параметри, които да направят нещата в нея точно определени, то получената теория или няма да дава същите предсказания за поведението на света, както съществуващата квантова механика, или ще трябва да допуска предаване на информация със скорост по-бърза от скоростта на светлината. Това е така, дори ако предположим, че тези допълнителни параметри са „скрити“ и недостъпни за наблюдение.

вътрешната структура на тези низове не бе от значение. Например не е от значение, че „срамежлива_мимоза“ се състои от двете думи „срамежлива“ и „мимоза“. Също така без никакво значение е и това, че низът „Нубия“ се състои от пет символа, а не от седем. Затова в математическите дефиниции на *терм* и *атомарна формула* се приема, че всеки един от тези низове се състои от един единствен символ. В частност ще считаме, че низовете „срамежлива_мимоза“ и „Нубия“ са съставени от един символ. Разбира се, че в действителност думата „Нубия“ не се състои от един символ, а от пет. Това обаче за нашите цели ще бъде без значение. Също и планетите не са точки, но понякога астрономът може да си мисли, че са.

Нещо интересно е това, че тази математическа абстракция се оказва полезна не само на теория, но и на практика. Тя съответства на начина, по който се правят транслаторите. Да си припомним, че лексическият анализатор превръща израза `return` в *лексема*, след което синтактичният анализатор работи с тази лексема все едно, че тя представлява един единствен символ. Следователно преобразуването на програмата, извършено от лексическия анализатор, е коректно и не води до неправилна работа на компилатора, само защото когато се интересуваме от смисъла на една програма, е безопасно да си мислим, че думата `return` е съставена от един единствен символ — истинският брой на символите в тази дума няма абсолютно никакво отношение към това какво трябва да прави програмата.

Да разгледаме терма $\mathtt{f}(\mathtt{g}(\mathtt{a},\mathtt{b}),\mathtt{c})$ и синтактичното му дърво от фигура 7. В този терм надписите `f` и `g`, които наподобяват извиквания на функции с аргументи, отговарят на възли в синтактичното дърво, които не са листа. Математиците наричат такива символи *функционални символи*. Символите пък `a`, `b` и `c`, които в синтактичното дърво са листа, се наричат *символи за константи*, *индивидни константи* или просто *константи*. При термовете, които срещахме в предния раздел, „столицата_на“ и „бащата_на“ са функционални символи, а „Сократ“, „Евридика“, „Александър_Македонски“ и др. са символи за константи. Думи пък като „царува“, „обича“, „преподава“, „намери“ и др. се наричат *предикатни символи*. Предикатните символи са означения за предикати, функционалните символи — означения за функции, а символите за константи — означения за конкретни обекти.

Забележка: Много е важно да не забравяме, че символите за константи, функционалните символи и предикатните символи са символи, които сами по себе си нямат фиксиран смисъл. Също както π не е число, а буква от гръцката азбука, с която означаваме определено число,

така и символът „Солон“ не е цар на Атѝна, а означение, което можем да използваме, за да обозначаваме този цар.

За краткост, често вместо термина „символ за константа“ се използва по-краткото „константа“. Това е възможно, тъй като в контекста на логическото програмиране и математическата логика истински константи почти няма и значи винаги когато кажем, че нещо е „константа“, е ясно, че всъщност искаме да кажем, че то е „символ за константа“. В това отношение функционалните и предикатните символи са онеправдани, защото в логическото програмиране се използват функции и предикати и затова не можем вместо „функционален символ“ да казваме накратко „функция“, нито пък вместо „предикатен символ“ може да казваме „предикат“.*

В много програмни езици имената на функции, методи, процедури и т.н. могат да се претоварват. Това означава, че на две напълно различни функции могат да се дадат едни и същи имена, стига начинът, по който функциите се извикват, да ни позволява да различаваме коя точно функция се има предвид. Например на пролог едноименни предикати с различен брой аргументи се считат за напълно различни предикати. На си++ не само броят на аргументите на една функция или метод е от значение, но също и типовете на аргументите. А при ада дори типът на стойността е от значение.

Да забележим обаче, че претоварването, макар понякога да е удобно когато даваме имена, всъщност не добавя никакви съществено нови възможности към езика за програмиране. Ако в даден език за програмиране претоварването е забранено, просто можем да даваме различни имена на различните функции, методи и т.н. И при термовете ситуацията е аналогична. Няма проблем да позволяваме един функционален символ да се използва с различен брой аргументи, но това няма да добави никакви съществено нови възможности. Това е така, защото вместо да използваме едновременно термове $\mathtt{f}(\mathtt{a},\mathtt{b})$ и $\mathtt{f}(\mathtt{a},\mathtt{b},\mathtt{c})$, винаги можем вместо това да използваме напр. $\mathtt{f}_2(\mathtt{a},\mathtt{b})$ и $\mathtt{f}_3(\mathtt{a},\mathtt{b},\mathtt{c})$, т.е. при различен брой аргументи да използваме различни функционални символи (в случая $\mathtt{f}_2$ и $\mathtt{f}_3$). И тъй като това ще ни облекчи когато правим математически разсъждения, ние ще постъпим точно така — ще приемем, че всеки функционален и всеки предикатен символ има фиксирана *арност*, която казва с колко аргумента той се използва. Впрочем и при програмирането претоварването не винаги е желателно и затова някои

*На изпита по логическо програмиране се счита за груба грешка, ако за нещо, което е функционален символ, се каже че е функция.

езици за програмиране умишлено го забраняват. Например в подмножеството на ада, което се използва за писане на програми без грешки,* претоварването е забранено.

Някои математици считат, че символите за константи всъщност са специален вид функционални символи, а именно — функционални символи с арност 0, т.е. с нула аргументи. При други математици пък функционалните символи задължително имат арност поне 1 и символите за константи не се считат за функционални.

За константите (или функционалните символи с арност 0), както и за предикатните символи с арност 0 казваме, че са *нулместни* или *нуларни*. За функционалните и предикатни символи с арност 1 казваме, че са *едноместни* или *унарни*. За функционалните и предикатни символи с арност 2 казваме, че са *двуместни* или *бинарни*. За функционалните и предикатни символи с арност 3 казваме, че са *триместни* или *тернарни*.

2.4.1. Уговорка. За да не се налага винаги изрично да се уточнява кой символ е константа, кой функционален, кой предикатен и кой променлива, в математиката се използват следните уговорки:

- за променливи се използват буквите $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}$;
- буквите $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}$ са символи за константи;
- за функционални (не нулместни) символи се използват буквите $\mathtt{f}, \mathtt{g}, \mathtt{h}$, а при нужда още и $\mathtt{j}, \mathtt{k}, \mathtt{l}$.
- за предикатни символи се използват буквите $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$.

Ако трябват повече символи, се използват индекси, примове и т.н., напр. $\mathtt{x}$, $\mathtt{x}'$, $\mathtt{x}_3$ и $\mathtt{x}''_2$ са променливи, а $\mathtt{p}$, $\mathtt{p}_1$ и $\mathtt{p}''_2$ са предикатни символи.

Благодарение на тази уговорка, когато видим израз като $\mathtt{f}(\mathtt{g}(\mathtt{c}), \mathtt{x}, \mathtt{b})$, е ясно, че това е терм, а не атомарна формула (защо?). В този терм $\mathtt{f}$ е триместен функционален символ, $\mathtt{g}$ е едноместен функционален символ,

*Става въпрос за SPARK. Да — има технологии, позволяващи писането на програми (почти) без грешки. В някои случаи писането на програми без грешки просто е необходимост. Например и най-малката грешка в компютърна програма, управляваща пътнически самолет, може да се окаже фатална. На много места има метро, в което влаковете не се управляват от човек, а автоматично от компютър — тук също не искаме управляващата програма да има грешки. Или да си представим програма, управляваща медицински скенер, където една излишна нула може да облъчи пациента с фатална доза радиация. Една грешка в програма за електронен подпис може да вкара невинен човек в затвора. Компютъризацията в обществото непрекъснато расте, а с това се увеличава и броят на приложенията, при които е нужно програмите да бъдат без грешки.

`c` и `b` са символи за константи и `x` е променлива. От друга страна изразът $\mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{x}), \mathtt{c}))$ е атомарна формула, а не терм. В нея `p` е едноместен предикатен символ, `c` е символ за константа, `f` е двуместен функционален символ, а `g` е едноместен функционален символ.

***Задача* 6:** Открийте функционалните символи, константите и променливите в терма $\mathtt{f}(\mathtt{f}(\mathtt{c}, \mathtt{a}), \mathtt{f}(\mathtt{g}(\mathtt{g}(\mathtt{x})), \mathtt{h}(\mathtt{y})))$. За всеки от функционалните символи определете неговата арност.

***Задача* 7:** Кои от следните изрази не са термове и защо: $\mathtt{f}(\mathtt{c})$, $\mathtt{c}(\mathtt{f})$, `c`, `f`, $\mathtt{x}(\mathtt{c})$, $\mathtt{f}(\mathtt{f}(\mathtt{x}))$, $\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{c})))$, $\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{c}, \mathtt{c})))$, $\mathtt{g}(\mathtt{g}(\mathtt{x}, \mathtt{x}), \mathtt{g}(\mathtt{x}, \mathtt{y}))$.

***Задача* 8:** Кои от следните изрази са атомарни формули и кои от тях не съдържат променливи: `c`, $\mathtt{f}(\mathtt{c})$, $\mathtt{p}(\mathtt{c})$, $\mathtt{p}(\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{c}))))$, $\mathtt{p}(\mathtt{f}(\mathtt{x}))$, $\mathtt{f}(\mathtt{p}(\mathtt{x}))$, $\mathtt{p}(\mathtt{p}(\mathtt{x}))$, $\mathtt{f}(\mathtt{f}(\mathtt{x}))$?

Сигнатури

Въпреки че за практически цели уговорка 2.4.1 върши работа, често имаме нужда от по-гъвкав критерий за това кой символ какъв е. При дефиницията на съждителна формула 2.2.2 използвахме съждителна сигнатура (вж. деф. 2.2.1), която ни казваше кои точно символи са съждителните променливи. Така ще направим и сега. Ще дефинираме понятието *сигнатура за предикатната логика*, която ще ни каже кои точно са променливите, символите за константи, функционалните символи, предикатните символи, както и тяхната арност.

Всъщност значението на сигнатурите за предикатната логика е много по-голямо, отколкото за съждителната. Изразителната сила на съждителната логика винаги е една и съща, без значение кои точно символи ще бъдат съждителните променливи. При предикатната логика обаче може да имаме най-различни, нееквивалентни по между си сигнатури. При някои сигнатури например може да няма функционални символи, а при други да има. При някои има само един предикатен символ, който е едноместен, при други само един, който е двуместен, при трети имаме безброй много триместни предикатни символи и т.н. Изразителната сила на предикатната логика е различна при всяка една от тези сигнатури. Ще използваме фрази като „терм τ при сигнатура $\mathfrak{sig}$“ и „атомарна формула φ при сигнатура $\mathfrak{sig}$“, за да кажем, че термът τ и атомарната формула φ са построени, използвайки символите, предоставени от сигнатурата $\mathfrak{sig}$.*

*Терминът сигнатура (на англ. signature) се използва в математическата логика,

Има различни, еквивалентни по между си, начини да дефинираме сигнатура. При тук използваната дефиниция сигнатурата ще бъде функция. Аргументът на тази функция е символ, за който искаме да разберем дали е променлива, константа и т.н., а стойността ѝ ще ни казва какъв точно е символът, даден като аргумент. Например ако $\mathfrak{sig}(\xi) = \texttt{пром}$, то значи символът ξ е променлива от сигнатурата $\mathfrak{sig}$.

2.4.2. Дефиниция. а) За символите лява и дясна скоба, запетая, &, $\vee$, $\neg$, $\Rightarrow$ и $\Leftrightarrow$ ще казваме, че са *запазени*.

б) Ще казваме, че функцията $\mathfrak{sig}$ е *сигнатура* (за предикатната логика), ако дефиниционната ѝ област включва само символи, които не са запазени и има безброй много символи $\texttt{x}$, за които $\mathfrak{sig}(\texttt{x}) = \texttt{пром}$.

в) Символът $\texttt{x}$ е *променлива* от сигнатурата $\mathfrak{sig}$, ако $\mathfrak{sig}(\texttt{x}) = \texttt{пром}$.

г) Символът $\texttt{c}$ е *символ за константа* (или нулместен функционален символ) от сигнатурата $\mathfrak{sig}$, ако $\mathfrak{sig}(\texttt{c}) = \texttt{конст}$.

д) Нека $n \geqq 1$. Символът $\texttt{f}$ е n-местен *функционален символ* от сигнатурата $\mathfrak{sig}$, ако $\mathfrak{sig}(\texttt{f}) = \langle \texttt{функсим}, n \rangle$.

е) Нека $n \geqq 0$. Символът $\texttt{p}$ е n-местен *предикатен символ* от сигнатурата $\mathfrak{sig}$, ако $\mathfrak{sig}(\texttt{p}) = \langle \texttt{предсим}, n \rangle$. Понякога вместо „предикатен символ“ се използва терминът *релационен символ*.

За да не усложняваме излишно формулировките на дефинициите и твърденията, нека фиксираме една конкретна сигнатура $\mathfrak{sig}$. Ако в някоя дефиниция или твърдение говорим просто за „функционални символи“, „променливи“, „термове“ и т.н., ние всъщност ще имаме предвид функционални символи, променливи и термове при така фиксираната сигнатура $\mathfrak{sig}$.

Да забележим, че когато символът $\texttt{f}$ е функционален символ, не е достатъчно за сигнатурата да ни каже само това, защото трябва от някъде да разберем и каква е арността на $\texttt{f}$. Затова когато $\texttt{f}$ е функционален символ, ще поискаме $\mathfrak{sig}(\texttt{f})$ да бъде наредена двойка, чийто пръв елемент ще бъде думата $\texttt{функсим}$, а вторият елемент ще ни дава арността на $\texttt{f}$. Също и когато символът $\texttt{p}$ е предикатен символ, сигнатурата трябва да ни каже каква е арността на $\texttt{p}$ и затова в този случай $\mathfrak{sig}(\texttt{p})$ е наредена двойка, чийто пръв елемент е думата $\texttt{предсим}$, а вторият елемент ни дава арността на $\texttt{p}$.

алгебрата, теория на езиците за програмиране и информатиката. Понякога в математическата логика и алгебрата вместо за сигнатура се говори за „език“ (language). В универсалната алгебра сигнатурите понякога се наричат „типове“ (types). В теория на моделите пък се използва терминът „речник“ (vocabulary).

Ограничението сигнатурата да не е дефинирана за запазените символи се дължи на това, че ще използваме тези символи като означения за логически операции при конструирането на формули и няма да бъде удобно същите символи да се използват и като променливи, символи за константи, функционални или предикатни символи.

Да забележим, че условието в дефиниция 2.4.2, според което има безброй много символи $\mathtt{x}$, за които $\mathfrak{sig}(\mathtt{x}) = \mathtt{пром}$, ни гарантира, че винаги съществуват безброй много променливи. В доказателството на някои твърдения се налага да се използват изречения от следния вид: „Нека $\mathtt{x}$ е произволна променлива, която не се среща във формулата φ“. Такива доказателства не би биха били верни, ако се окаже, че формулата φ съдържа всички възможни променливи. За да си спестим този проблем, е удобно да поискаме да съществуват безброй много променливи. Така, тъй като в една формула могат да се съдържат само краен брой променливи, със сигурност ще има променливи, които не се съдържат във φ.

Дефиниция на терм

Вече сме готови да дадем и дефиницията на терм. Дефиницията ще бъде индуктивна. Това означава, че ако за даден израз успеем да докажем, че е терм, използвайки в доказателството единствено трите правила от дефиниция 2.4.3, то тогава този израз е терм. Ако пък за даден израз е невъзможно да се докаже, че е терм, използвайки единствено тези три правила, то тогава той не е терм.

2.4.3. Дефиниция. Нека $\mathfrak{sig}$ е сигнатура. Понятието *терм* при сигнатура $\mathfrak{sig}$ се дефинира индуктивно посредством следните правила:

а) Ако $\mathtt{x}$ е променлива от $\mathfrak{sig}$, то $\mathtt{x}$ е терм при сигнатура $\mathfrak{sig}$.

б) Ако $\mathtt{c}$ е символ за константа от $\mathfrak{sig}$, то $\mathtt{c}$ е терм при сигнатура $\mathfrak{sig}$.

в) Ако $\mathtt{f}$ е n-местен функционален символ от $\mathfrak{sig}$ и $\tau_1, \tau_2, \ldots, \tau_n$ са термове при сигнатура $\mathfrak{sig}$, то низът $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ е терм при сигнатура $\mathfrak{sig}$.

2.4.4. *Забележка*: Вижда се, че в така дадената дефиниция използваме функционален запис за термовете (вж. пример 2.1.5). Нямаше да има никакъв проблем в нея да използваме кой да е друг запис, който гарантира еднозначност на синтактичния разбор. Например ако искахме да ползваме термове в полски запис, в 2.4.3 в) вместо за $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ можеше да кажем, че низът $\mathtt{f}\tau_1\tau_2\ldots\tau_n$ е терм при сигнатура $\mathfrak{sig}$.

***Задача* 9:** Нека сигнатурата $\mathfrak{sig}$ е такава, че $\mathtt{x}$ е променлива, $\mathtt{c}$ е символ за константа, а $\mathtt{f}$ — двуместен функционален символ. Използвайки правилата от дефиниция 2.4.3, докажете, че $\mathtt{f}(\mathtt{c}, \mathtt{f}(\mathtt{c}, \mathtt{x}))$ е терм при сигнатура $\mathfrak{sig}$.

***Задача* 10:** Съществуват ли термове:

- които съдържат скоби, но не съдържат запетаи?
- които съдържат запетаи, но не съдържат скоби?
- в които броят на левите скоби е по-голям от броя на десните скоби?

***Задача* 11:** Напишете терм, който съдържа 2 скоби и 6 запетаи. Напишете терм, който съдържа 6 скоби и 2 запетаи.

****Задача* 12:** Да допуснем, че дефиницията позволяваше специалните символи да се използват като константи и функционални символи. Нека лявата скоба е символ за константа, дясната — едноместен функционален символ, а запетаята — двуместен функционален символ. Открийте функционалните символи и символите за константи в „термовете“, оградени с правоъгълници:

$\boxed{(}$ $\boxed{)(()}$ $\boxed{)()(())}$ $\boxed{,((,()}$ $\boxed{,()(()),,((,())}$

Един любопитен факт с дълго доказателство е това, че подобни изрази винаги притежават еднозначен прочит — никога не възниква ситуация, при която някоя скоба е възможно да се изтълкува хем като истинска скоба, хем като функционален символ. Това отчасти се използва в езика пролог, в който символът запетая се използва хем като разделител на аргументите на функционалните символи, хем като двуместен функционален символ.

В раздела за формални езици (вж. раздел 2.1) споменахме, че формалните езици се делят на два вида — такива, за които е предназначено да се обработват от компютър, и такива, които са предназначени за хора. Също така споменахме, че когато формалният език е предназначен за хора, не е нужно изразите от езика да се дефинират по начин, който е удобен за използване от хора. Този парадокс обяснихме по следния начин: щом като изразите от формалния език ще се четат от хора, значи няма да възникнат проблеми, ако ги пишем не съвсем според дефиницията, но когато ги четем, се сещаме кой точно формален израз имаме предвид. При езиците, обработвани от компютър, не може да постъпваме по този начин, защото компютърът не може сам да поправя допуснатите от нас волни или неволни синтактични грешки.

В дадената току-що дефиниция 2.4.3 термовете не са дефинирани по начин, удобен за използване от хора. Това е така, защото за по-лесно ще дефинираме езика на предикатната логика като език за хора, а не като език за компютри. Ако искахме да дефинираме език, който ще се обработва от компютри, тогава например:

- символите за константи и функционалните символи всъщност не биха били символи, а редици от символи (да си припомним „срамежливата_мимоза“ от раздел 2.3);
- за по-добра четливост бихме позволявали използването на интервали в термове от вида $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$;
- в някои случаи бихме допускали инфиксен запис, напр. $\mathtt{a}+\mathtt{b}$ вместо $+(\mathtt{a}, \mathtt{b})$.

Но въпреки че нашата дефиниция не споменава изрично всички тези „удобства“, няма никакви причини да не се възползваме от тях. Спокойно може да пишем термове като $\mathtt{a} + \mathtt{b}$, стига да се сещаме, че това всъщност е термът $+(\mathtt{a}, \mathtt{b})$.

***Задача* 13:** Колко скоби се съдържат в терма $\mathtt{c} + \mathtt{x} * \mathtt{a}$?

Индукция за термове

Винаги, когато даваме индуктивна дефиниция на дадено понятие, е в сила съответен принцип за индукция. Сравнете внимателно трите правила от дефиницията на терм с трите условия в следния принцип за индукция:

2.4.5. Принцип за индукция по построението на терм. *Нека* $\mathfrak{p}$ *е такова свойство, че:*

а) ако $\mathtt{x}$ *е променлива, то* $\mathtt{x}$ *притежава свойството* $\mathfrak{p}$*;*

б) ако $\mathtt{c}$ *е символ за константа, то* $\mathtt{c}$ *притежава свойството* $\mathfrak{p}$*;*

в) ако $\mathtt{f}$ *е* n*-местен функционален символ и* $\tau_1, \tau_2, \ldots, \tau_n$ *са термове, които притежават свойството* $\mathfrak{p}$*, то също и* $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ *притежава свойството* $\mathfrak{p}$*.*

В такъв случай всички термове притежават свойството $\mathfrak{p}$*.*

Доказателство. Нека τ е произволен терм.

Тъй като τ е терм, то това може да бъде доказано, използвайки единствено трите правила от дефиниция 2.4.3. В едно такова доказателство ще се изказват твърдения, че определени изрази са термове. Нека $\sigma_1, \sigma_2, \ldots, \sigma_m$ са всички изрази, за които в това доказателство се

казва, че са термове, подредени според срещането им в доказателството. Ще докажем, че термовете $\sigma_1, \sigma_2, \ldots, \sigma_m$ притежават свойството $\mathfrak{p}$. Тъй като в това доказателство със сигурност ще е изказано и твърдението, че τ е терм (т.е. $\tau = \sigma_i$ за някое i), то по този начин ще получим исканото.

Да изберем $i \in \{1, 2, \ldots, m\}$. Ще докажем, че σ_i притежава свойството $\mathfrak{p}$ с пълна математическа индукция по i. Индукционното предположение ще казва, че термовете $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ притежават свойството $\mathfrak{p}$.

Тъй като в разглежданото доказателство се позволява да се използват единствено трите правила от дефиниция 2.4.3, то значи фактът, че σ_i е терм, е доказан с някое от тези три правила.

Ако този факт е доказан с правило 2.4.3 а), то значи σ_i е променлива. В такъв случай от правило 2.4.5 а) получаваме, че σ_i притежава свойството $\mathfrak{p}$.

Ако този факт е доказан с правило 2.4.3 б), то значи σ_i е символ за константа. В такъв случай от правило 2.4.5 б) получаваме, че σ_i притежава свойството $\mathfrak{p}$.

Остава възможността това да е било доказано с правило 2.4.3 в). В този случай $\sigma_i = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$, където $\mathtt{f}$ е n-местен функционален символ, а $\tau_1, \tau_2, \ldots, \tau_n$ са изрази, за които по-рано в разглежданото доказателство вече е било доказано, че са термове (с други думи всеки от термовете $\tau_1, \tau_2, \ldots, \tau_n$ е равен на някой от термовете $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$). Съгласно индукционното предположение $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ притежават свойството $\mathfrak{p}$, а значи и $\tau_1, \tau_2, \ldots, \tau_n$ притежават свойството $\mathfrak{p}$ и от правило 2.4.5 в) получаваме, че σ_i притежава свойството $\mathfrak{p}$. ∎

Забележка: Много е важно този принцип за индукция да не се учи наизуст, а човек сам да може да съобразява какво гласи той, помнейки единствено каква е дефиницията на терм. Доказателството му трябва да се разбере, но също не е нужно да се учи.

2.4.6. Пример. Ще използваме индуктивния принцип за термове, за да докажем, че всички термове съдържат равен брой леви и десни скоби.

Доказателство. а) Ако $\mathtt{x}$ е променлива, то $\mathtt{x}$ съдържа равен брой леви и десни скоби (нула), защото променливите не са запазени символи, а скобите са и значи $\mathtt{x}$ не е скоба.

б) Ако $\mathtt{c}$ е символ за константа, то $\mathtt{c}$ съдържа равен брой леви и десни скоби (нула), защото символите за константи не са запазени символи, а скобите са и значи $\mathtt{c}$ не е скоба.

в) Ако $\mathtt{f}$ е n-местен функционален символ, то нека допуснем, че $\tau_1, \tau_2, \ldots, \tau_n$ са термове с равен брой леви и десни скоби.* Нека k_i е броят на левите скоби в τ_i (същото число е равно и на броя на десните скоби). Тъй като $\mathtt{f}$ не е запазен символ, то $\mathtt{f}$ не е скоба. Следователно броят на левите скоби в терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ е $1 + k_1 + k_2 + \cdots + k_n$ — една отворена скоба непосредствено след символа $\mathtt{f}$ плюс скобите в $\tau_1, \tau_2, \ldots, \tau_n$. Броят на десните скоби е същият — затворената скоба в края на терма плюс скобите в $\tau_1, \tau_2, \ldots, \tau_n$. ■

***Задача* 14:** Нека сигнатурата $\mathfrak{sig}$ е такава, че в нея няма нито един символ за константа. Използвайки индуктивния принцип за термове, докажете, че всеки терм при сигнатура $\mathfrak{sig}$ съдържа поне една променлива.

***Задача* 15:** Нека сигнатурата $\mathfrak{sig}$ е такава, че всички функционални символи от $\mathfrak{sig}$ имат арност 2 (или 0). Използвайки индуктивния принцип за термове, докажете, че във всички термове при сигнатура $\mathfrak{sig}$ броят на запетаите е равен на броя на десните скоби.

***Задача* 16:** Да разгледаме следната индуктивна дефиниция на понятието „буртант“:

1. 5 е буртант;
2. 8 е буртант;
3. Ако n и m са буртанти, то nm^2 е буртант.

Какъв е индуктивният принцип за буртанти? Използвайки го, докажете, че ако n е буртант, то $n + 1$ е естествено число, което се дели на 3 без остатък.

2.5. Предикатна логика

Атомарни формули

В предния раздел дефинирахме какво значи „сигнатура за предикатната логика“ (дефиниция 2.4.2). Тази сигнатура ни казва кои символи са променливи, кои са символи за константи и кои са функционални. Без да знаем това, не бихме могли да дефинираме какво значи терм (дефиниция 2.4.3). Освен това сигнатурата ни казва и кои символи са предикатни символи, а тях все още не сме ги използвали. Ще ги използваме сега в дефиницията на атомарна формула:

*Това допускане са нарича *индукционно предположение*.

2.5.1. Дефиниция. Нека е дадена сигнатура $\mathfrak{sig}$. Ако $\mathtt{p}$ е n-местен предикатен символ от $\mathfrak{sig}$ и $\tau_1, \tau_2, \ldots, \tau_n$ са термове при сигнатура $\mathfrak{sig}$, то низът $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е *атомарна формула* при сигнатура $\mathfrak{sig}$.

2.5.2. Означение. За удобство атомарните формули, образувани от нулместен предикатен символ, обикновено се записват без скоби, т.е. $\mathtt{p}$, а не $\mathtt{p}()$. Това е аналогично термовете, образувани от нулместни функционални символи, т.е. символи за константи. Например ако $\mathtt{c}$ е символ за константа, то термът, образуван от $\mathtt{c}$ е просто $\mathtt{c}$, а не $\mathtt{c}()$. Причината, поради която дефиниция 2.5.1 не е съобразена с този начин за запис, е това, че така си спестяваме необходимостта да разглеждаме случаи за арността на предикатните символи в доказателствата, в които се говори за атомарни формули. Впрочем тук не сме напълно последователни, защото ако в дефиницията за терм не бяхме отделили символите за константи като отделен случай, а считахме, че те са просто вид функционални символи (по-точно нулместни функционални символи), то бихме си опростили доказателствата още повече.

Просто сравнение на тази дефиниция с дефиницията на терм (вж. точка 2.4.3 в) от дефиницията) показва, че една атомарна формула $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ прилича по всичко на терм с единствената разлика, че $\mathtt{p}$ не е функционален, а предикатен символ. И също както при термовете, когато например $+$ е двуместен функционален символ се уговорихме за удобство да използваме инфиксен запис като пишем например $\mathtt{a} + \mathtt{b}$ вместо $+(\mathtt{a}, \mathtt{b})$, така и при атомарните формули при някои предикатни символи за удобство ще използваме инфиксен запис. Например ако „$<$“ е двуместен предикатен символ, ще пишем например $\mathtt{a} < \mathtt{b}$ вместо $<(\mathtt{a}, \mathtt{b})$ и също ако „$=$“ е двуместен предикатен символ, ще пишем $\mathtt{a} = \mathtt{b}$ вместо $=(\mathtt{a}, \mathtt{b})$. Разбира се, това правим само за наше удобство. Правилните атомарни формули, отговарящи на дефиниция 2.5.1 са $<(\mathtt{a}, \mathtt{b})$ и $=(\mathtt{a}, \mathtt{b})$.

Тъй като атомарните формули приличат на термове, може да възникне въпросът защо изобщо имаме нужда от това понятие. Не може ли да считаме символите $=$ и $<$ за функционални, при което $\mathtt{a} = \mathtt{b}$ и $\mathtt{a} < \mathtt{b}$ ще станат термове и няма да имаме нужда от атомарни формули?

Отговорът на този въпрос е следният: ако не се интересуваме от „смисъла“ на термовете, а само от външния им вид, тогава наистина можем да минем и без атомарни формули. Когато обаче вземем предвид смисъла на термовете и атомарните формули, се вижда защо може да използваме термове като аргументи на функционален символ и тогава получаваме по-голям терм. Също може да използваме термове като

аргументи на предикатен символ и в резултат получаваме атомарна формула. Обаче ако използваме атомарни формули като аргументи на функционален или предикатен символ, резултатът е безсмислица.

2.5.3. Пример. Ето примерен терм, който се получава като дадем на функционалния символ „баща_на“ като аргумент по-малкия терм „Зевс“:

баща_на(Зевс)

Ето примерна атомарна формула, която се получава като дадем на двуместния предикатен символ „чичо“ термовете „баща_на(Зевс)“ и „Прометей“:

чичо(баща_на(Зевс), Прометей)

Ето примерна безсмислица, която се получава като дадем атомарна формула като аргумент на функционален символ:

баща_на(чичо(Кронос, Прометей))

Интуитивно атомарната формула „чичо(Кронос, Прометей)“ казва, че Кронос е чичо на Прометей. Т.е. това е някакво съждение. Кой е бащата на това съждение? Разбира се, това е безсмислен въпрос. Съжденията могат да бъдат например верни и могат също да бъдат неверни, но нямат бащи. Ето още една безсмислица, която се получава като дадем атомарна формула като аргумент на предикатен символ:

чичо(Хиперион, чичо(Кронос, Прометей))

Чий чичо е Хиперион? На съждението „Кронос е чичо на Прометей“.

2.5.4. Пример. Да разгледаме следните три израза, в които 0 е символ за константа, „+“ е двуместен функционален символ и = е двуместен предикатен символ:

$$0 + (0 + 0)$$
$$0 = (0 + 0)$$
$$0 + (0 = 0)$$

Първият от тези изрази е терм. Например ако интерпретираме символа 0 като числото нула, а „+“ като събиране на числа, тогава стойността на този терм е нула. Вторият от тези изрази е атомарна формула. Ако отново интерпретираме символа 0 като числото нула, „+“ като събиране на числа и „=“ като равенство, тогава тази атомарна формула

е вярна. Третият от горните изрази обаче е безсмислица. Не може да съберем $0 = 0$ с числото нула.*

Предикатни формули

В раздел 2.3 вече видяхме, че много съждения не могат да се изразят, използвайки само средствата, с които разполага съждителната логика. Например съждения като „Сократ е смъртен“ и „Орфей обича Евридика“ е най-естествено да се запишат като атомарни формули. Обаче не може да се ограничим само с атомарните формули, защото много съждения се формулират, използвайки различни логически операции, а в атомарните формули няма логически операции. Да разгледаме например следното съждение:

Ако Афродита е най-красивата, то Елена ще обича Парис.

Използвайки средствата на съждителната логика, можем да представим това съждение посредством формулата $\mathtt{p} \Rightarrow \mathtt{q}$, където $\mathtt{p}$ е съждителна променлива, с която означаваме съждението „Афродита е най-красивата“, а $\mathtt{q}$ съждителна променлива, с която означаваме съждението „Елена обича Парис“. Съждителните променливи $\mathtt{p}$ и $\mathtt{q}$ обаче по никакъв начин не показват какви съждения сме означили с тях. От друга страна, използвайки атомарни формули, можем да преведем не цялото съждение, а само поотделно двете му части, например така:

най-красива(Афродита)
обича(Елена, Парис)

Възниква въпросът — не може ли да дефинираме такава логика, в която хем имаме атомарни формули, хем можем да използваме и съждителни операции, както в съждителната логика? Отговорът на този въпрос е положителен.

Да си припомним дефиницията на съждителна формула (дефиниция 2.2.2). Тя бе индуктивна като най-простите съждителни формули са съждителните променливи, а от произволни вече построени съждителни формули можеше да образуваме нова съждителна формула, използвайки съждителните операции $\neg$, $\&$, $\vee$ и $\Rightarrow$. Например ако вече знаем, че φ и ψ са съждителни формули, то тогава може да заключим, че низът $(\varphi \,\&\, \psi)$ също е съждителна формула.

*В някои езици за програмиране, напр. си, този израз не е безсмислен (по-точно изразът `0+(0==0)`). Това е така само защото в тези езици равенството не е логическа, а аритметична операция, чиято стойност е числото едно или нула, а не истина или лъжа.

Ако в момента дадем аналогична дефиниция, само че вместо съждителни променливи използваме атомарни формули, ще получим това, което в логиката се нарича „безкванторна предикатна формула“. Използвайки безкванторна предикатна формула, горното съждение може да бъде преведено ето така:

най-красива(Афродита) $\Rightarrow$ обича(Елена, Парис)

Много неща могат да се направят, използвайки безкванторни предикатни формули. Сега обаче ще дадем направо най-общата дефиниция за произволна предикатна формула. Разликата между дефиницията на произволна предикатна формула и дефиницията на безкванторна предикатна формула се състои в това, че при произволните предикатни формули освен съждителните операции $\neg$, $\&$, $\vee$ и $\Rightarrow$ може да се използват и кванторите $\forall$ и $\exists$.

2.5.5. Дефиниция. *Предикатните формулите* или просто *формулите* се дефинират индуктивно:

а) ако φ е атомарна формула, то φ е формула;

б) ако φ е формула, то низът $\neg\varphi$ е формула;

в) ако φ и ψ са формули, то низовете $(\varphi \,\&\, \psi)$, $(\varphi \vee \psi)$ и $(\varphi \Rightarrow \psi)$ са формули;

г) ако $\mathtt{x}$ е променлива и φ е формула, то низовете $\forall\mathtt{x}\,\varphi$ и $\exists\mathtt{x}\,\varphi$ са формули.

Разбира се, навсякъде в тази дефиниция трябва да се имат пред вид формули при определена сигнатура. При различните сигнатури имаме различни атомарни формули, а значи при различните сигнатури ще имаме и различни формули. Множеството от всички формули при някоя конкретна сигнатура наричаме *език на предикатната логика* или просто *език*.

2.5.6. Когато се дава точната дефиниция на понятието „формула“, не е нужно в нея да се споменават изрично всички логически операции. Например в дефиниция 2.5.5 не сме казали нищо за операцията еквиваленция. Това не създава проблеми, защото можем да считаме, че всяка формула от вида $\varphi \Leftrightarrow \psi$ представлява съкратен запис на формулата $((\varphi \Rightarrow \psi) \,\&\, (\psi \Rightarrow \varphi))$.

2.5.7. Пример. Нека $=$ и $\in$ са двуместни предикатни символи, а $\mathtt{x}$, $\mathtt{y}$ и $\mathtt{z}$ са променливи. Тогава изразът*

$$\forall\mathtt{x}\,\forall\mathtt{y}\,(\forall\mathtt{z}\,(\mathtt{z} \in \mathtt{x} \Leftrightarrow \mathtt{z} \in \mathtt{y}) \Rightarrow \mathtt{x} = \mathtt{y})$$

*В теория на множествата този израз се нарича аксиома за екстенсионалност или аксиома за обемност.

е съкратен запис на формулата

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\forall \mathtt{z}\, ((\in(\mathtt{z}, \mathtt{x}) \Rightarrow \in(\mathtt{z}, \mathtt{y})) \,\&\, (\in(\mathtt{z}, \mathtt{y}) \Rightarrow \in(\mathtt{z}, \mathtt{x}))) \Rightarrow =(\mathtt{x}, \mathtt{y}))$$

2.5.8. За улеснение, когато записваме формули ще си позволяваме някои „неточности“. Например вече споменахме, че атомарните формули от вида $=(+(0,0),0)$ ще бъдат записвани по-четливо като $0+0=0$. Освен това ще си позволяваме да изпускаме някои от скобите. Например може да считаме, че всяка формула от вида $\varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4$ е съкратен запис на формулата $(((\varphi_1 \vee \varphi_2) \vee \varphi_3) \vee \varphi_4)$. Ще считаме, че импликацията ($\Rightarrow$) и еквиваленцията ($\Leftrightarrow$) са с най-малък приоритет, конюнкцията ($\&$) и дизюнкцията ($\vee$) са със среден (и равен по между си) приоритет и унарните операции ($\neg$, $\forall$ и $\exists$) са с най-голям приоритет. Например всяка формула от вида $\varphi_1 \,\&\, \varphi_2 \Rightarrow \varphi_3 \vee \varphi_4$ е съкратен запис на формулата $((\varphi_1 \,\&\, \varphi_2) \Rightarrow (\varphi_3 \vee \varphi_4))$.

Дефиницията на формула е дадена по такъв начин, че да направи вярно следното твърдение:

2.5.9. Твърдение за еднозначен синтактичен разбор. *Всяка формула притежава единствено синтактично дърво.*

От еднозначността на синтактичния разбор следва например, че никоя формула не може да бъде едновременно от вида $(\varphi_1 \& \varphi_2)$ и $(\psi_1 \vee \psi_2)$, нито пък едновременно от вида $(\varphi_1 \Rightarrow \varphi_2)$ и $\neg\psi$. Също така от тук следва например, че една съждителна формула може да бъде едновременно от вида $(\varphi_1 \vee \varphi_2)$ и $(\psi_1 \vee \psi_2)$ само когато $\varphi_1 = \psi_1$ и $\varphi_2 = \psi_2$.

2.5.10. Пример. Синтактичното дърво на формулата

$$\forall \mathtt{x}\, (\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$

е илюстрирано във фигура 9.

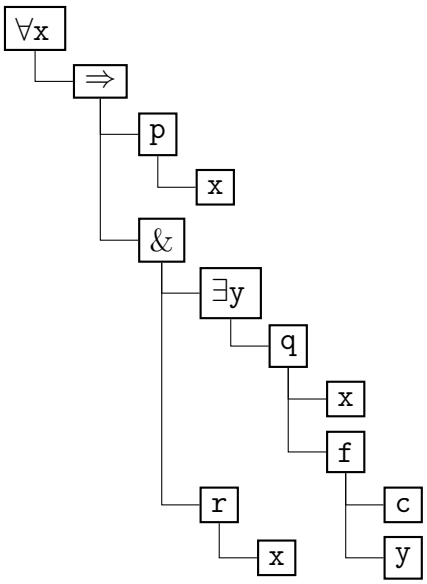
***Задача* 17:** Кое е синтактичното дърво на следната формула:

$$\exists \mathtt{x}\, (\neg \forall \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}))$$

Грешка ще бъде да считаме, че всяка формула има фиксиран смисъл. Да разгледаме например следната формула:

$$\mathtt{x} + 0 = \mathtt{x} \tag{5}$$

В тази формула $=$ е двуместен предикатен символ, $+$ е двуместен функционален символ, 0 е символ за константа и $\mathtt{x}$ е променлива. Какво



Фиг. 9. Синтактично дърво за формулата .
$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$

ни казва тази формула? Например какви стойности може да приема променливата $\mathtt{x}$? Естествени числа, множества от думи или квадратни матрици 3×3? Какъв е смисълът на символа $+$? Събиране на естествени числа, обединение на множества или събиране на матрици?

По-нататък ще дефинираме понятието структура. Именно структурата ще дава конкретен смисъл на символите, които използваме във формулите. При различните структури тези символи ще придобиват различен смисъл и следователно е безсмислено да питаме за една формула дали е вярна, или не, без да сме казали коя е структурата, спрямо която интерпретираме символите във формулата. Тъй като смисълът на формулите зависи от структурата, спрямо която ги интерпретираме, то значи при някои структури една формула може да бъде вярна, а при други — не.

2.5.11. Пример. Да разгледаме няколко различни структури, в които да интерпретираме формулата (5).

а) В първата структура, която ще разгледаме, нека възможните стойности на променливата $\mathtt{x}$ са естествените числа. Нека „$=$“ е равенство, символът за константа 0 е числото нула, а „$+$“ е събиране на естествени числа. В тази структура формула (5) ни казва, че всяко естествено число, събрано с нула ни дава същото естествено число. Следователно в тази структура формулата е вярна.

б) Сега да разгледаме структура, която е същата като предходната, само че символът 0 се интерпретира като числото едно. В тази структура формулата ни казва, че всяко естествено число, събрано с единица, дава същото естествено число. Следователно в тази структура формулата не е вярна.

в) Да разгледаме структура, в която стойностите на променливите са езици (т.е. множества от думи), символът 0 се интерпретира като празният език $\varnothing$, а символът + като обединение на езици. В тази структура формулата ни казва, че ако обединим някой език с празния език, ще получим същия език. Следователно в тази структура формулата е вярна.

г) Да разгледаме структура, в която стойностите на променливите са реални числа или $+\infty$, символът 0 се интерпретира като $+\infty$, а + се интерпретира като операцията минимум. В тази структура формулата казва, че ако x е реално число или $+\infty$, то по-малкият елемент на множеството $\{x, +\infty\}$ е равен на x.

д) Да разгледаме структура, в която стойностите на променливите са многозначни функции от $\mathbb{N}$ в $\mathbb{N}$, символът 0 се интерпретира като празната функция, която никога не дава резултат, а + е обединението на многозначни функции. В тази структура формула (5) ни казва, че обединението на коя да е многозначна функция с функцията, която никога не дава резултат, е същата функция. От гледна точна на теория на изчислимостта това означава, че ако пуснем някой процес `x` паралелно с процес, който се зацикля и нищо не прави, това е все едно да пуснем единствено процеса `x` (очевидно това предполага математическата идеализация, че разполагаме с безкрайни ресурси и затова не се интересуваме, че зациклящият се процес може да забави пресмятанията).

2.6. Абстрактна алгебра и програмиране

Многообразия и квазимногообразия

В абстрактната алгебра се дефинират различни видове алгебрични структури, като също се формулират и аксиоми, които се удовлетворяват във всяка структура от съответния вид. Например аксиомите за поле са краен брой предикатни формули, които притежават следните две свойства:

– аксиомите за поле са верни във всяко поле;

– ако аксиомите за поле са верни в някоя структура, то тази структура е поле.

Ето една примерна формула, която обикновено се включва сред аксиомите за поле:

$$\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$$

Да забележим, че макар тази формула да е вярна във всяко поле, смисълът на функционалния символ + е различен в различните полета. Например в полето на реалните числа е вярно $2 + 2 = 4$, а в полето $\mathbb{Z}_3$ е вярно $2 + 2 = 1$.

2.6.1. Дефиниция. Нека Θ е клас от структури за език с равенство, в който няма други предикатни символи освен равенството. Да забележим, че в този език всяка атомарна формула представлява равенство между два терма. Казваме, че Θ е *многообразие*,* ако съществува такова множество Γ от атомарни формули от езика на Θ, че за всяка структура за езика на Θ е вярно, че:

– ако в структурата са верни формулите от Γ, то тази структура е елемент на Θ;

– ако структурата е елемент на Θ, то в нея са верни формулите от Γ.

Формулите от Γ се наричат *аксиоми* на Θ.

Например групите и пръстените образуват многообразия, защото както при групите, така и при пръстените всяка от аксиомите представлява равенство между някакви изрази (т.е. термове). Едно малко по-общо понятие от многообразията са квазимногобразията. Няма популярни класове от структури, които са квазимногообразия, но не са многообразия. Въпреки това квазимногообразията имат важни компютърни приложения.

2.6.2. Дефиниция. *Квазимногообразията* се дефинират аналогично на многообразията, само че при аксиомите позволяваме не само атомарни формули, но и формули от вида

$$\tau_1 = \sigma_1 \,\&\, \ldots \,\&\, \tau_n = \sigma_n \Rightarrow \tau = \sigma$$

*На английски *variety*. В алгебричната геометрия има алгебрични многообразия (algebraic varieties), които са различни от многообразията, за които говорим тук. Освен това в диференциалната геометрия на английски съществува терминът *manifold*, който няма нищо общо с variety, но на български също се превежда като „многообразие“.

Полетата са важен клас алгебрични структури, които не само че не са многобразие, но не са дори и квазимногообразие. Това е така, защото една от аксиомите за поле казва, че всеки ненулев елемент има обратен елемент. С формула това може да бъде записано по следния начин:

$$\neg(\mathtt{x} = 0) \Rightarrow \mathtt{x}.\mathtt{x}^{-1} = 1$$

Използването на $\mathtt{x}^{-1}$ тук не е проблем, защото може да считаме, че $^{-1}$ е едноместен функционален символ. Например може да считаме, че $\mathtt{x}^{-1}$ е по-четлив запис на терма $f(\mathtt{x})$. Проблем е обаче наличието на отрицание от лявата страна на импликацията, защото това не се позволява при аксиомите на квазимногообразие. Може да се докаже, че този проблем е съществен — не е възможно така да измислим аксиомите за поле, че всяка от тях да има вида, позволен за аксиомите на квазимногообразие (вж. твърдение **??**).

Ще се спрем малко по-подробно на две многобразия, които имат разнообразни приложения в теоретичната информатика — моноидите и полупръстените.

Моноиди

Структура с една асоциативна операция (означена по-долу с „.“), която има неутрален елемент (означен по-долу с 1), се нарича *моноид*. С други думи, ако в някоя структура за език с равенство са изпълнени следните аксиоми, то тя е моноид:

$$\mathtt{x}.(\mathtt{y}.\mathtt{z}) = (\mathtt{x}.\mathtt{y}).\mathtt{z}$$
$$\mathtt{x}.1 = \mathtt{x}$$
$$1.\mathtt{x} = \mathtt{x}$$

Пример. Някои примери за моноиди са:

а) Положителните цели числа с операцията умножение. Неутрален елемент е числото 1.

б) Положителните цели числа, като „.“ е операцията събиране, а символът 1 е числото нула. В тази структура аксиомата $\mathtt{x}.1 = \mathtt{x}$ казва, че всяко естествено число, събрано с нула, дава същото число.

в) Думите от дадена азбука Σ с операцията конкатенация. Неутрален елемент е празната дума ε. Този моноид се нарича *свободен моноид* над Σ.

г) Всички езици от думи над дадена азбука с операцията конкатенация на езици. Неутрален елемент е $\{\varepsilon\}$.

д) Всички функции, изобразяващи елементите на дадено множество (например естествените числа) в същото множество, с операцията композиция на функции. Неутрален елемент е функцията идентитет.*

Асоциативни операции (а значи и моноиди) се появяват често в задачите, възникващи при програмиране. Благодарение на това много алгоритми могат да се разпаралелят по лесен и безопасен начин. Например ако е необходимо да пресметнем произведението

$$\prod_{i=1}^{8000} k_i$$

на компютър с осемядрен процесор, ние можем да извършим това по следния начин: на първото ядро пресмятаме $\prod i = 1^{1000} k_i$, на второто — $\prod_{i=1001}^{2000} k_i$, на третото — $\prod_{i=2001}^{3000} k_i$ и т.н.** Защо не можем да използваме този метод когато операцията не е асоциативна?

Полупръстени

Полупръстените образуват по-сложно многообразие. Полупръстенът е структура с две операции „+“ и „.“, които са асоциативни и имат неутрални елементи съответно 0 и 1. Операцията „+“ е комутативна. Освен това е в сила дистрибутивен закон за „+“ и „.“, а 0 действа като нулев елемент по отношение на операцията „.“. Ето как всичко това

*Да си припомним, че ако $f, g \colon M \to M$ са две функции, то тяхната композиция $f \circ g$ е функцията, за която

$$(f \circ g)(\mu) = f(g(\mu))$$

за всяко $\mu \in M$. Функцията идентитет id е функцията, за която

$$\mathrm{id}(\mu) = \mu$$

за всяко $\mu \in M$.

**Този метод няма да подобри крайното бързодействие, ако пресмятането на произведението може да се извърши бързо и на едно ядро. Ако има вероятност пресмятането на различните частични произведения да отнема различно по продължителност време, трябва да се обмисли използвато по-сложен метод за разпаралеляване, защото иначе може се случи седем от ядрата да си свършат бързо работата, а цялата трудна работа да се падне само на осмото ядро.

може да бъде изразено с формули:

$$\begin{aligned}
\mathtt{x} + (\mathtt{y} + \mathtt{z}) &= (\mathtt{x} + \mathtt{y}) + \mathtt{z} \\
\mathtt{x} + \mathtt{y} &= \mathtt{y} + \mathtt{x} \\
0 + \mathtt{x} &= \mathtt{x} \\
\mathtt{x}.(\mathtt{y}.\mathtt{z}) &= (\mathtt{x}.\mathtt{y}).\mathtt{z} \\
1.\mathtt{x} &= \mathtt{x} \\
\mathtt{x}.1 &= \mathtt{x} \\
\mathtt{x}.(\mathtt{y} + \mathtt{z}) &= (\mathtt{x}.\mathtt{y}) + (\mathtt{x}.\mathtt{z}) \\
(\mathtt{y} + \mathtt{z}).\mathtt{x} &= (\mathtt{y}.\mathtt{x}) + (\mathtt{z}.\mathtt{x}) \\
0.\mathtt{x} &= 0 \\
\mathtt{x}.0 &= 0
\end{aligned}$$

Пример. Някои примери за полупръстени са:

а) Естествените числа с операции събиране и умножение.

б) Езиците над дадена азбука като операцията „+“ е обединението на езици, операцията „.“ е конкатенацията на езици, 0 е празният език, а 1 е $\{\varepsilon\}$.

в) Многозначните изчислими функции. Операцията „+“ е обединението на две многозначни функции, операцията „.“ е композицията на многозначни функции, 0 е функцията която никога не връща резултат, а 1 е функцията идентитет. Този полупръстен се използва в теория на изчислимостта и може да се използва за моделиране на алгоритмични пресмятания, при които паралелните процеси не комуникират по между си. [12]*

г) Реалните числа заедно с $+\infty$, където дефинираме $\mathtt{x} + \mathtt{y}$ да бъде по-малкото от $\mathtt{x}$ и $\mathtt{y}$, а $\mathtt{x}.\mathtt{y}$ е сборът на $\mathtt{x}$ и $\mathtt{y}$. Неутрален елемент за „+“ е $+\infty$ и неутрален елемент за „.“ е 0. Този полупръстен се нарича *тропически полупръстен*.** Той има различни прило-

*За съжаление все още не е открит и може би не съществува хубав математически формализъм, който може да се използва за моделиране на комуникиращи паралелни процеси. Такива процеси може да се моделират например посредством π-*смятането* и *джойн-смятането*, но и двете имат много лоши алгебрични свойства.

**Приложната математика обикновено използва математически обекти, които са били дефинирани от математици теоретици, които изобщо не са се интересували, че с измислените от тях понятия в бъдеще ще може да се решават практически задачи. Тропическият полупръстен е рядък пример за интересен математически обект, който е бил откриван и преоткриван много пъти от математици приложници и едва в последствие е заинтересувал и математиците теоретици.

жения, едно от които е свързано с анализ на дискретни системи, т.е. системи, чието поведение не е „гладко“ и затова не може да се опише посредством диференциални уравнения. Примери за дискретни системи са транспортните мрежи, комуникационните и компютърните мрежи, централните процесори на компютрите, производствените цехове в заводите и др.

Многосортни структури

Моноидите и полупръстените представляват примери за т.н. едносортни структури. При едносортните структури всички обекти са от един и същи тип — в случая това са елементите на моноида или полупръстена. Има обаче и *многосортни структури*. Например линейните пространства са пример за двусортна структира, в която има два вида обекти — скалари и вектори. За всяка променлива, срещаща се във формула от многосортен език е определен тип, който показва кои са позволените стойности на съответната променлива. Например една от аксиомите за линейно пространство е двусортната формула

$$(\mathtt{x}.\mathtt{y}).\mathtt{v} = \mathtt{x}.(\mathtt{y}.\mathtt{v})$$

в която стойностите на `x` и `y` са всевъзможните скалари в структурата (напр. реални числа), а стойностите на `v` — всевъзможните вектори в структурата.

Въпреки че теорията на многосортните структури не е по-сложна от тази на едносортните, с цел да избегнем някои технически усложнения в този курс ще използваме само едносортни структури.

Алгебрично програмиране

Има една група езици за формална спецификация и програмиране, при които дефинираме различните типове данни аксиоматично — посредством задаване на свойствата, удовлетворявани от обектите от съответните типове. По практически съображения се налага да наложим някои ограничения върху вида на аксиомите. Тези ограничения са такива, че всъщност алгебричното програмиране се основава на използването на многосортни квазимногообразия. Мауде* е един от популярните езици за алгебрично програмиране. Ето как изглежда на мауде описанието на квазимногообразието на моноидите:

*`http://maude.cs.uiuc.edu`

```
fth MONOID is
   sort Elt .
   op 1 : → Elt .
   op _*_ : Elt Elt → Elt [assoc id: 1] .
endfth
```

Тук дефинираме *сорт* (т.е. тип данни) `Elt` и две операции `1` и `*`. Операцията `1` е просто елемент на моноида, а операцията `*` е двуместна, асоциативна и притежаваща `1` като неутрален (единичен) елемент.

Полупръстените могат да бъдат дефинирани на мауде по следния начин:

```
fth SEMIRING is
   sort Elt .
   op 0 : → Elt .
   op _+_ : Elt Elt → Elt [assoc comm id: 0] .
   op 1 : → Elt .
   op _*_ : Elt Elt → Elt [assoc id: 1] .
   vars x y z : Elt .
   eq x * (y + z) = (x * y) + (x * z) [nonexec] .
   eq (x + y) * z = (x * z) + (y * z) [nonexec] .
   eq 0 * x = 0 [nonexec] .
   eq x * 0 = 0 [nonexec] .
endfth
```

Тук елементът 0 е описан като неутрален за асоциативната и комутативната операция +, елементът 1 като неутрален за асоциативната *, а дистрибутивните закони са описани с явно зададени аксиоми. Атрибутът `nonexec` казва на мауде, че не е нужно да обръща внимание на последните четири аксиоми докато пресмята стойността на операциите + и *.

От математическа гледна точка езиците за алгебрично програмиране се основават на съществуването на т.н. *инициални структури*, които са единствени с точност до изоморфизъм. Съществуването на инициална структура за всяко многообразие е доказано от Гарет Биркхоф [5], а за квазимногообразията — от Анатолий Иванович Малцев [23, 24, стр. 271].

Естествено възниква въпросът каква част от структурите, които бихме искали да използваме при програмирането, могат да се опишат като инициални структури на някое квазимногообразие. Йоханес Бергстра и Джон Тъкер са доказали [3, 4], че ако е възможно операциите на

една структура да се пресмятат алгоритмично,[*] то тогава тази структура може да се опише аксиоматично като инициална структура на многообразие, стига да обогатим структурата с краен брой нови операции. Следователно всички структури, чиито операции могат да се пресмятат с компютър, могат да се представят като инициални структури на някакво многообразие.

Въпреки че езиците за алгебрично програмиране наистина могат да се използват за програмиране, тяхното предназначение е друго — като езици за формална спецификация и описание на семантиката на езици за програмиране. Има различни езици за спецификация, но едно ценно свойство, което отличава езиците за алгебрично програмиране от повечето други езици за спецификация, е това, че при алгебричните езици самата спецификация ни дава изпълним код. В някои случаи скоростта на изпълнение на такава програма се оказва изненадващо бърза. Например ако запишем на мауде денотационната семантика на езика скийм, то ще получим интерпретатор на скийм, чиято скорост достига до 75% от скоростта на обикновен интепретатор на скийм. И въпреки че ако оставим на мауде да изпълни формалната спецификация на виртуалната машина на джава, ще получим доста бавен интерпретатор, да правим това изобщо не е безсмислено. Макар и бавен, този интерпретатор със сигурност отговаря на формалната си спецификация (т.е. не съдържа програмни грешки) и затова може да се използва за безопасно изпълнение на програми, които са потенциално злонамерени (вируси и др.).[**]

2.7. Свободни и свързани променливи

Уводни бележки

Променливите в математиката и програмирането са два вида — свободни и свързани. Двата вида променливи са много различни един от

[*] Такива структури се наричат *изчислими* или *рекурсивни*.

[**] Всъщност когато става въпрос за софтуерна безопасност рядко можем да имаме пълна сигурност за каквото и да е. Например злонамереният код на джава може се възползва от програмни грешки в интерпретатора на мауде, за да излезе от наложените му ограничения. Ако пък целта на злонамерения код е само шпионска, тогава защитата става още по-сложна. Дори и без да излиза от наложените му ограничения, такъв код може да осъществи нерегламентирано предаване на данни, като влияе по подходящ начин на натоварването на централния процесор — натоварване, което може да бъде следено от разстояние или посредством електромагнитните излъчвания на процесора, или по шума, издаван от вентилатора, или дори по консумацията на електроенергия от електрическата мрежа.

друг. Да разгледаме израза

$$\sum_{i=1}^{n} i^2$$

За да пресметнем този израз е нужно да знаем каква е стойността на променливата n, но не и стойността на променливата i. Въпреки че в този израз се срещат две променливи — n и i – очевидно тези променливи са много различни. Ако искаме с горния израз да дефинираме функция, тази функция ще зависи само от n, но не и от i:

$$f(n) = \sum_{i=1}^{n} i^2$$

Променливите, подобни на n, се наричат *свободни променливи*, а подобните на i се наричат *свързани променливи*.

Ето още няколко примера. В следващия израз променливата x е свързана, а y е свободна:

$$\int_0^1 f(x, y)dx$$

В следния програмен блок променливата `i` е свързана, а променливите `a` и `x` са свободни:

```
{
   int i;
   for(i=1;i<1000;i++){
      x[i] = (x[i-1]*x[i-1]) % a;
   }
}
```

И така, да обобщим — стойността (смисъла) на един израз зависи от стойността на свободните променливи, срещащи се в него, но не зависи от стойността на свързаните променливи в израза.

Едно друго различие между свободните и свързаните променливи е следното. Ако в един израз заменим всяко срещане на свободната променлива x с някакъв израз, ще се получи пак смислен израз. Да заменяме свързана променлива с израз не е позволено.

Например ако в израза

$$\sum_{i=1}^{n} i^2$$

заменим n с k^2+5 получаваме израза

$$\sum_{i=1}^{k^2+5} i^2$$

Ако в израза

$$\int_0^1 f(x,y)dx$$

заменим y с z^2+5 получаваме израза

$$\int_0^1 f(x,z^2+5)dx$$

Ако в горния програмен блок заменим променливата `a` с израза `a*a+5` получаваме новия програмен блок:*

```
{
   int i;
   for(i=1;i<1000;i++){
      x[i] = (x[i-1]*x[i-1]) % (a*a+5);
   }
}
```

Ако по подобен начин заменяме свързаните променливи с изрази, обикновено се получават безсмислици. Например ако в израза

$$\sum_{i=1}^{n} i^2$$

заменим i с k^2+5 получаваме

$$\sum_{(k^2+5)=1}^{n} (k^2+5)^2$$

Ако в израза

$$\int_0^1 f(x,y)dx$$

*При подобни замени трябва да се спазват типовете. Например ако променливата `a` се срещаше отляво на оператор за призвояване, тогава бихме могли да заменяме тази променлива само с израз, притежаващ определен адрес в паметта. В този случай замяната на `a` с `a*a+5` не би била коректна, но замяната с `x[4]` ще бъде коректна.

заменим x с z^2+5 получаваме

$$\int_0^1 f(z^2+5,y)d(z^2+5)$$

Ако в горния програмен блок заменим `i` с `a*a+5` получаваме

```
{
   int a*a+5;
   for(a*a+5=1;a*a+5<1000;(a*a+5)++){
      x[i] = (x[i-1]*x[i-1]) % (a*a+5);
   }
}
```

Да забележим, че във всеки от горните случаи свързаната променлива има нещо като „свързващ оператор“, от който съвсем ясно личи, че се получава безсмислица. Това са $\sum_{(k^2+5)=1}^{n}$, $\int_0^1 \ldots d(z^2+5)$ и декларацията `int a*a+5;`.

При преименуване на свързана променлива смисълът на израза се запазва. Например

$$\sum_{i=1}^{n} i^2 = \sum_{j=1}^{n} j^2 \neq \sum_{i=1}^{m} i^2$$

$$\int_0^1 f(x,y)dx = \int_0^1 f(z,y)dz \neq \int_0^1 f(x,z)dx$$

Също и в горния програмен блок смисълът се запазва, ако преименуваме `i` на `j`, но не и ако преименуваме `a` на `b`.

Една друга разлика между свободните и свързаните променливи е това, че свободните променливи имат глобална видимост, а свързаните — локална. Да разгледаме следния програмен фрагмент:

```
{
   int i, j;
   i = a * a;
   {
      int i;
      i = b + 5;
      j = 3;
   }
```

свободни променливи	*свързани променливи*
изразът зависи от стойността им	стойността им не е релевантна
може да се заместват с изрази	не може да се заместват
не може да се преименуват	може да се преименуват
глобална видимост	локална видимост

Таблица 3. Свойства на свободните и свързаните променливи

```
   {
      int i, a;
      i = b + j;
      a = i * i;
      b = a * i;
   }
}
```

В този фрагмент са декларирани три променливи с име `i`. Въпреки че използват едно и също име, тези три променливи са напълно различни една от друга. Освен това всяка си има своя област на видимост. Променливите `i`, декларирани във вътрешните блокове, са видими само в рамките на тези блокове. Променливата `i`, декларирана във външния блок, също е видима само в рамките на този блок, но не и извън него. Освен това вътрешните променливи `i` скриват външната променлива `i`, в следствие на което тя не се вижда във вътрешните блокове. Променливата `j` обаче не се скрива от вътрешна декларация и затова се вижда и във вътрешните блокове.

В този програмен фрагмент се използват и две свободни променливи — `a` и `b`. Всяко срещане на променлива `b` в този програмен фрагмент се отнася за една и съща променлива. Във втория вътрешен блок обаче е декларирана свързана променлива `a`. Тази свързана променлива `a` скрива свободната променлива `a`.

При свързаните променливи може да има различни свързани променливи с едно и също име, но при свободните това е невъзможно — едноименните свободни променливи винаги са една и съща променлива.

Казаното дотук за свободните и свързаните променливи е резюмирано в таблица 3.

Всички променливи, които се срещат в един терм са свободни, а не свързани. За да се убедим в това, да разгледаме едно по едно свойствата от таблица 3:

– От разгледаните разнообразни примери за алгебрични структури

се вижда, че в общия случай стойността на един терм зависи от стойността на променливите, които се срещат в него. Например стойността на терма $\mathtt{x}+\mathtt{x}$ зависи от стойността на променливата $\mathtt{x}$.

– Ако заменим една променлива в терм с друг терм, пак получаваме терм. Например ако заменим променливата $\mathtt{x}$ в терма $\mathtt{f}(\mathtt{x})$ с терма $\mathtt{g}(\mathtt{x},\mathtt{y})$, ще получим терма $\mathtt{f}(\mathtt{g}(\mathtt{x},\mathtt{y}))$.

– Ако преименуваме променливите в един терм, получаваме различен терм. Термовете $\mathtt{f}(\mathtt{x})$ и $\mathtt{f}(\mathtt{y})$ имат различен смисъл.

– Всички променливи $\mathtt{x}$, които се срещат в един терм, означават една и съща променлива $\mathtt{x}$. Следователно променливите в термовете имат глобална видимост.

Всички променливи, които се срещат в една атомарна формула, също са свободни, а не свързани. В това можем да се убедим по същия начин:

– Не е безсмислено да се питаме каква е верността на една атомарна формула при някакви конкретни стойности на променливите, които се срещат в нея. Например верността на атомарната формула $\mathtt{x} = 0$ зависи от стойността на променливата $\mathtt{x}$.

– Ако заменим една променлива в атомарна формула с друг терм, пак получаваме атомарна формула. Например ако заменим променливата $\mathtt{x}$ в атомарната формула $\mathtt{p}(\mathtt{x})$ с терма $\mathtt{g}(\mathtt{x},\mathtt{y})$, ще получим атомарната формула $\mathtt{p}(\mathtt{g}(\mathtt{x},\mathtt{y}))$.

– Ако преименуваме променливите в една атомарна формула, получаваме различна атомарна формула. Атомарните формули $\mathtt{p}(\mathtt{x})$ и $\mathtt{p}(\mathtt{y})$ имат различен смисъл.

– Всички променливи $\mathtt{x}$, които се срещат в една атомарна формула, означават една и съща променлива $\mathtt{x}$. Следователно променливите в атомарните формули имат глобална видимост.

В една произволна формула обаче може да има и свързани променливи. Това се дължи на кванторите. Да разгледаме например формулата $\forall\mathtt{x}\,\mathtt{p}(\mathtt{x})$. В нея променливата $\mathtt{x}$ е свързана, защото:

– Верността на формулата $\forall\mathtt{x}\,\mathtt{p}(\mathtt{x})$ не зависи от никаква конкретна стойност на променливата $\mathtt{x}$.

– Ако заменим в тази формула променливата $\mathtt{x}$ с терма $\mathtt{f}(\mathtt{x})$ получаваме безсмислица: $\forall\mathtt{f}(\mathtt{x})\,\mathtt{p}(\mathtt{f}(\mathtt{x}))$.

– Ако преименуваме в тази формула променливата $\mathtt{x}$ с $\mathtt{y}$, получаваме формула с напълно същия смисъл: $\forall\mathtt{y}\,\mathtt{p}(\mathtt{y})$.

– Кванторите създават локална видимост за променливата си. Ако в контекста на формулата $\forall \mathtt{x}\, \mathtt{p}(\mathtt{x})$ се срещат някакви променливи $\mathtt{x}$, то това са напълно различни променливи $\mathtt{x}$. Например първата променлива $\mathtt{x}$ от формулата $\mathtt{q}(\mathtt{x}) \,\&\, \forall \mathtt{x}\, \mathtt{p}(\mathtt{x})$ е напълно различна променлива $\mathtt{x}$ (която впрочем е свободна), от втората и третата променлвива $\mathtt{x}$, която е свързана.

Подформули

За да направим по-ясно кои са свързаните и кои са свободните променливи във формула, ще въведем едно помощно понятие:

2.7.1. Дефиниция. Когато формулата ψ е част от формулата φ (т.е. ψ е подниз на φ), казваме, че ψ е *подформула* на φ.

2.7.2. За да разберем правилно смисъла на току-що дадената дефиниция, е най-добре да си мислим не за самите формули, които са някакви редици от символи, а за техните синтактични дървета. Ако си мислим за синтактичното дърво на една формула, някои от поддърветата ѝ ще бъдат нейни подформули, а други (по-малките) — термове. Подформулите на една формула се получават от всички поддървета, които са формули, а не термове. Областта на действие на всеки квантор е поддървото, което се намира под съответния квантор. Разгледайте фигура 10, в която е илюстрирано синтактичното дърво на формулата

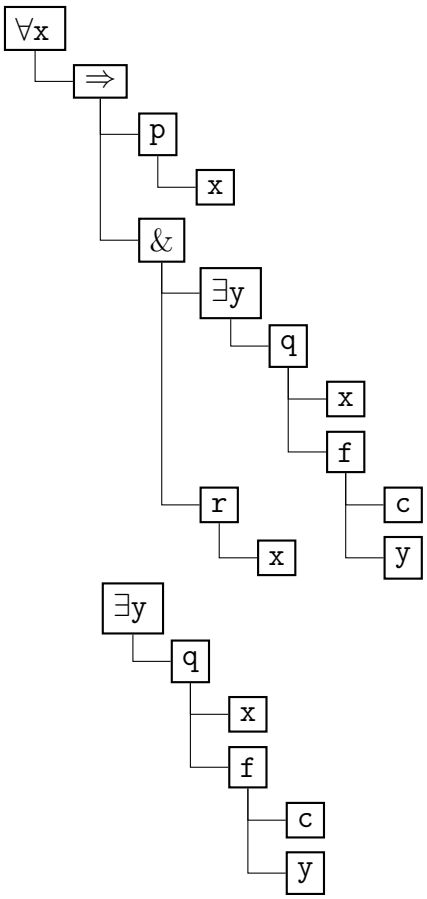
$$\forall \mathtt{x}\, (\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$

както и синтактичното дърво на областта на действие на квантора $\exists \mathtt{y}$ в тази формула.

Ако решим да си мислим не за синтактичните дървета, а за самите формули като редици от символи, тогава дефиниция 2.7.1 ще бъде вярна само тогава, когато изписваме формулата изцяло, без да пропускаме нито едни скоби. Да разгледаме отново формулата

$$\forall \mathtt{x}\, (\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$

Според дефиниция 2.7.1, ако една подформула започва с квантор, тогава тази подформула е областта на действие на този квантор. Една подформула на горната формула е формулата $\exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x})$. Тази формула започва с квантора $\exists \mathtt{y}$ и значи излиза, че тя трябва да е областта на действие на този квантор. Достатъчно е обаче да погледнем дървото от фигура 10, за да видим, че това е невярно!



Фиг. 10. Синтактично дърво на формула и подформула

Тази грешка се обяснява по следния начин — не е вярно, че формулата $\exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x})$ започва с квантор. Ако я напишем изцяло, без да пропускаме скоби, тогава ще видим че тази формула започва не с квантор, а със скоба, защото дефиниция 2.5.5 в) изисква около всяка конюнкция да се слагат скоби:

$$(\exists \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))$$

Следователно тази подформула започва не с квантор, а със скоба.

Тъй като е много неудобно всеки път да преценяваме къде се слагат всички нужни скоби и дали дадена подформула започва с квантор, или със скоба, която не е написана, най-добре е когато откриваме подформулите на една формула, да си мислим за синтактичното дърво на формулата, а не за самата формула.

***Задача* 18:** Намерете всички подформули, на формулата от фигура 10.

Свързани и свободни променливи във формула

Ако приложим всичко казано до момента за свободните и свързаните променливи в една формула, ще получим следната дефиниция:

2.7.3. Дефиниция. а) Когато формулата φ съдържа подформула от вида $\forall \mathtt{x}\, \psi$ или $\exists \mathtt{x}\, \psi$ (където ψ е формула), казваме, че тази подформула е *областта на действие* на квантора $\forall \mathtt{x}$ или $\exists \mathtt{x}$, с който започва подформулата.

б) Едно срещане на променливата $\mathtt{x}$ в някоя формула е *свързано*, ако попада в областта на действие на някой квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$.

в) Едно срещане на променливата $\mathtt{x}$ в някоя формула е *свободно*, ако не попада в областта на действие на никой квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$.

г) Променливата $\mathtt{x}$ е *свободна променлива* на формулата φ, ако $\mathtt{x}$ се среща в φ на място, което не попада в областта на действие на никой квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$. С други думи $\mathtt{x}$ е свободна променлива, ако $\mathtt{x}$ има свободно срещане във формулата.

Много е важно да се разберат следващите три примера.

2.7.4. Пример. В безкванторните формули няма квантори, следователно не е възможно една променлива да попадне в областта на действие на квантор. Следователно всички променливи, които се срещат в безкванторните формули са техни свободни променливи. Например променливите $\mathtt{x}$, $\mathtt{y}$ и $\mathtt{z}$ са свободните променливи на формулата

$$(\mathtt{p}(\mathtt{x}, \mathtt{c}) \vee \neg \mathtt{q}(\mathtt{y})) \Rightarrow \mathtt{p}(\mathtt{z}, \mathtt{x})$$

Атомарните формули са вид безкванторни формули, следователно всички променливи, които се срещат в атомарните формули са техни свободни променливи.

2.7.5. Пример. Променливите $\mathtt{x}$ е единствената свободна променлива на формулата

$$\forall \mathtt{x}\, \forall \mathtt{y}\, (\mathtt{p}(\mathtt{x}, \mathtt{c}) \vee \neg \mathtt{q}(\mathtt{y})) \Rightarrow \forall \mathtt{z}\, \mathtt{p}(\mathtt{z}, \mathtt{x})$$

В тази формула променливата $\mathtt{x}$ от подформулата $\mathtt{p}(\mathtt{x}, \mathtt{c})$ е свързана, защото попада в областта на действие на квантора $\forall \mathtt{x}$. Същевременно променливата $\mathtt{x}$ от подформулата $\mathtt{p}(\mathtt{z}, \mathtt{x})$ е свободна, защото не попада в областта на действие на квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$.

Също както в езиците за програмиране декларирането на локална променлива в даден програмен блок скрива едноименните променливи, декларирани в по-външен блок, така и кванторите скриват едноименните свободни променливи, както и свързаните променливи с по-външен квантор.

2.7.6. Пример. Във формулата

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\mathtt{p}(\mathtt{x}) \vee \mathtt{p}(\mathtt{x})) \vee \mathtt{p}(\mathtt{x})$$

в първата атомарна формула $\mathtt{p}(\mathtt{x})$ променливата $\mathtt{x}$ се управлява от втория квантор, във втората атомарна формула $\mathtt{p}(\mathtt{x})$ променливата $\mathtt{x}$ се управлява от първия квантор, а в третата — променливата $\mathtt{x}$ е свободна.

***Задача* 19:** За всяка променлива, срещаща се във формулата

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{y}))$$

определете дали е свободна или свързана. Кои са кванторите, управляващи свързаните променливи? Коя е областта на действие на всеки един от кванторите? Кое е множеството от свободните променливи на тази формула?

Преименуване на свързаните променливи

В уводните бележки на този раздел бе споменато, че свързаните променливи в един израз може да се преименуват без това да промени смисъла на израза. Ако формулата φ може да се сведе до формулата ψ посредством преименуване на свързаните променливи, казваме, че тези две формули са *конгруентни* (точна дефиниция на това понятие е дадена малко по-долу). Ако две формули са конгруентни, то техният смисъл е един и същ.

2.7.7. Пример. Формулите $\forall \mathtt{x}\,\mathtt{p}(\mathtt{x}, \mathtt{y})$ и $\forall \mathtt{z}\,\mathtt{p}(\mathtt{z}, \mathtt{y})$ са конгруентни. Втората формула може да се получи от първата като преименуваме $\mathtt{x}$ на $\mathtt{z}$. Тези формули обаче не са конгруентн с формулата $\forall \mathtt{x}\,\mathtt{p}(\mathtt{x}, \mathtt{t})$, защото променливата $\mathtt{y}$ е свободна и не може да се преименува.

Ако във формулата $\forall \mathtt{x}\,\mathtt{p}(\mathtt{x}, \mathtt{y})$ преименуваме свързаната променлива $\mathtt{x}$ на $\mathtt{y}$, получаваме формулата $\forall \mathtt{y}\,\mathtt{p}(\mathtt{y}, \mathtt{y})$. Тези две формули обаче не са конгруентни, защото новопоявилият се квантор $\forall \mathtt{y}$ обхваща и свободната променлива $\mathtt{y}$ и я превръща от свободна в свързана.

Можем да дефинираме конгруентните формули по следния начин:

2.7.8. Дефиниция. а) Нека формулата φ съдържа квантор $\forall \mathtt{x}$ или $\exists \mathtt{x}$ и променливата $\mathtt{y}$ не се среща никъде в областта на действие на този квантор. Ако заменим в областта на действие на квантора всяко срещане на променливата $\mathtt{x}$ с $\mathtt{y}$, казваме, че новата формула е получена посредством *еднократно преименуване на свързана променлива* във φ.

б) Ако във формулата φ извършваме последователно няколко пъти (може и нула) еднократни преименувания на свързани променливи, то казваме, че получената накрая формула φ' е получена от φ посредством *преименуване на свързаните променливи*.

в) Формулата φ' е *конгруентна* с формулата φ, ако φ' може да се получи от φ посредством преименуване на свързаните променливи. Когато φ' е конгруентна с φ, записваме това така:

$$\varphi' \equiv \varphi$$

2.7.9. ***Забележка*:** Нормално е, когато преименуваме променливата на даден квантор, да преименуваме само променливите, които се „управляват“ от този квантор. Например ако във формулата

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\mathtt{p}(\mathtt{x}) \vee \mathtt{p}(\mathtt{x})) \vee \mathtt{p}(\mathtt{x}) \tag{6}$$

решим да преименуваме първия квантор от $\forall \mathtt{x}$ на $\forall \mathtt{y}$, тогава трябва да получим

$$\forall \mathtt{y}\,(\forall \mathtt{x}\,\mathtt{p}(\mathtt{x}) \vee \mathtt{p}(\mathtt{y})) \vee \mathtt{p}(\mathtt{x}) \tag{7}$$

Въпреки това, ако извършим преименуването така, както е определено в дефиниция 2.7.8 а), ще получим

$$\forall \mathtt{y}\,(\forall \mathtt{y}\,\mathtt{p}(\mathtt{y}) \vee \mathtt{p}(\mathtt{y})) \vee \mathtt{p}(\mathtt{x}) \tag{8}$$

Дефиницията е дадена по този начин, за да опростим нейната формулировка, но всъщност това по никакъв начин не е променило кои формули ще се окажат конгруентни и кои не. Наистина, нищо не ни пречи от формула (6) да получим най-напред формула (8), а след това с още едно преименуване от формула (8) да получим формула (7).

***Задача* 20:** Конгруентни ли са формулите:

1. $\forall \mathtt{x}\,\forall \mathtt{y}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x},\mathtt{y})$ и $\forall \mathtt{x}\,\forall \mathtt{z}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x},\mathtt{y})$
2. $\forall \mathtt{x}\,\forall \mathtt{y}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x},\mathtt{y})$ и $\forall \mathtt{y}\,\forall \mathtt{x}\,\forall \mathtt{x}\,\mathtt{p}(\mathtt{y},\mathtt{x})$
3. $\forall \mathtt{x}\,\forall \mathtt{y}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x},\mathtt{y})$ и $\forall \mathtt{y}\,\forall \mathtt{y}\,\forall \mathtt{x}\,\mathtt{p}(\mathtt{y},\mathtt{x})$

2.7.10. Твърдение. *а)* $\varphi \equiv \varphi$

б) ако $\varphi \equiv \psi$ *и* $\psi \equiv \chi$*, то* $\varphi \equiv \chi$

в) ако $\varphi \equiv \psi$*, то* $\psi \equiv \varphi$

Доказателство. **(а)** φ се получава от φ посредством нула на брой еднократни преименувания на свързани променливи.

(б) Ако от φ с няколко преименувания получим ψ и после с още няколко — χ, то значи от φ с няколко преименувания можем да получим χ.

(в) Да забележим, че ако можем да извършим еднократно преименуване на променливата $\mathtt{x}$ на $\mathtt{y}$, то са изпълнени условията, за да можем да извършим и обратното преименуване — от $\mathtt{y}$ на $\mathtt{x}$. Следователно ако ψ е получена от φ с помощта на няколко еднократни преименувания на свързани променливи, то нищо не ни пречи да получим φ от ψ като прилагаме преименуванията в обратен ред. ∎

Вече споменахме, че в теорията на предикатната логика ще искаме да отъждествяваме конгруентните формули, т.е. да работим „с точност до конгруентност“. Това означава, че трябва да бъдат верни твърдения, подобни на следващото.

2.7.11. Твърдение. *Ако формулите* φ *и* φ' *са конгруентни, то те имат едни и същи свободни променливи.*

Доказателство. Достатъчно е да докажем, че ако една формула е получена посредством еднократно преименуване на свързана променлива, то тя има същите свободни променливи, както и първоначалната формула. Нека формулата φ има подформула от вида $\forall\mathtt{x}\,\psi$, променливата $\mathtt{y}$ не се среща никъде в тази подформула и ψ' е получена като заменим в ψ всички срещания на $\mathtt{x}$ с $\mathtt{y}$. Нека формулата φ' се получава от φ като заменим подформулата $\forall\mathtt{x}\,\psi$ с $\forall\mathtt{y}\,\psi'$. В такъв случай формулата φ изглежда по следния начин (областта на действие на квантора $\forall\mathtt{x}$ е подчертана):

$$\varphi = \ldots\ldots\ldots \underline{\forall\mathtt{x}\,\psi} \ldots\ldots\ldots$$

а формулата φ' изглежда така (областта на действие на квантора $\forall\mathtt{y}$ е подчертана):

$$\varphi' = \ldots\ldots\ldots \underline{\forall\mathtt{y}\,\psi'} \ldots\ldots\ldots$$

Ако променливата $\mathtt{z}$ е различна както от $\mathtt{x}$, така и от $\mathtt{y}$, и има свободно срещане в φ, то $\mathtt{z}$ ще се среща свободно и в φ'. И обратно, ако

$\mathtt{z}$ се среща свободно в φ', то $\mathtt{z}$ ще се среща свободно и в φ'. Това е така, защото разликите между φ и φ' са свързани единствено с променливите $\mathtt{x}$ и $\mathtt{y}$ и по никакъв начин не влияят на променливата $\mathtt{z}$ и кванторите $\forall\mathtt{z}$ и $\exists\mathtt{z}$.

Всички променливи $\mathtt{x}$ в подформулата $\forall\mathtt{x}\,\psi$ са свързани, а променливата $\mathtt{y}$ не се среща в тази подформула. Освен това променливата $\mathtt{x}$ не се среща в подформулата $\forall\mathtt{y}\,\psi'$, а всички променливи $\mathtt{y}$ в тази подформула са свързани. Това означава, че всички свободни срещания на $\mathtt{x}$ или $\mathtt{y}$ в φ или φ' са извън тези две подформули, на мястото, означено по-горе с точки. Тъй като при преименуването не правим промени на местата, означени по-горе с точки, то ако някоя от променливите $\mathtt{x}$ или $\mathtt{y}$ се среща свободно в φ, тя ще се среща свободно и в φ' и също ако се среща свободно в φ', то тя ще се среща свободно и в φ.

Разгледахме случая на преименуване в подформула от вида $\forall\mathtt{x}\,\psi$. Случаят на подформула $\exists\mathtt{x}\,\psi$ е аналогичен. ∎

За да не се усложняваме ненужно, дефиниция 2.5.5 за формула е така дадена, че да можем да пишем „объркващи“ формули като например

$$\forall\mathtt{x}\,\forall\mathtt{y}\,(\forall\mathtt{x}\,\mathtt{p}(\mathtt{x},\mathtt{y})\vee\mathtt{q}(\mathtt{x},\mathtt{y}))$$

В тази формула има два квантора с променливата $\mathtt{x}$. Променливата $\mathtt{x}$ в подформулата $\mathtt{p}(\mathtt{x},\mathtt{y})$ ще бъде управлявана от втория квантор $\forall\mathtt{x}$, а променливата $\mathtt{x}$ в подформулата $\mathtt{q}(\mathtt{x},\mathtt{y})$ — от първия квантор $\forall\mathtt{x}$. Но въпреки, че използването на такива формули се позволява от дефинициите, обикновено е добре да пишем „нормални“ и по-лесни за осмисляне формули, в които няма квантори с една и съща променлива. Това винаги е възможно, защото ако има квантори с повтарящи се променливи, с подходящо преименуване на свързаните променливи може да получим конгруентна формула, в която няма квантори с една и съща променлива.

Друг вид „объркваща“ формула е например следната:

$$\forall\mathtt{x}\,\mathtt{p}(\mathtt{x})\Rightarrow\mathtt{q}(\mathtt{x})$$

Тази формула също е объркваща, защото в нея променливата $\mathtt{x}$ се среща хем като свободна, хем като свързана. В подформулата $\mathtt{p}(\mathtt{x})$ тя е свързана, а в $\mathtt{q}(\mathtt{x})$ — свободна. Но и тук няма проблем да се освободим от тази странност като преименуваме свързаната променлива $\mathtt{x}$ на някоя друга.

Ако пък Θ е някакво (крайно) множество от променливи, които по някакви причини „не харесване“, то отново — няма да има проблеми

така да преименуваме, че да намерим конгруентна формула, в която няма да има квантори с променливи от Θ. Следващата лема доказва всички тези неща накуп.

2.7.12. Лема за преименуване на свързаните променливи. *За всяка формула φ и крайно множество Θ от променливи съществува формула, конгруентна на φ, в която не се срещат квантори с променливи от Θ, всички квантори са с различни променливи и никоя от кванторните променливи не е свободна променлива.*

Доказателство. Нека Θ' е обединението на Θ с множеството от всички променливи (свързани и несвързани), които се срещат в φ. Множеството Θ' е крайно.

Ще построим редица от конгруентни формули $\varphi = \varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n$, в която последната формула ще отговаря на изискванията на лемата — всички кванторни променливи ще са различни и никоя кванторна променлива няма да е елемент на Θ'.

Ако разполагаме с формулата φ_i, можем да получим формулата φ_{i+1} по следния начин. Нека $\forall \mathtt{z}\, \psi$ или $\exists \mathtt{z}\, \psi$ е произволна подформула на φ_i, чиято променлива $\mathtt{z}$ е елемент на Θ' и в ψ не се срещат квантори с променливи от Θ'. Нека $\mathtt{z}'$ е произволна променлива, която не се среща нито в φ_i, нито е елемент на Θ'. Да преименуваме навсякъде в подформулата $\forall \mathtt{z}\, \psi$ или $\exists \mathtt{z}\, \psi$ променливата $\mathtt{z}$ на $\mathtt{z}'$. Нека така получената формула бъде φ_{i+1}.

На всяка стъпка броят на кванторите с променливи от Θ' намалява и значи процесът на строене на тази редица от конгруентни формули спира. Това ще се случи тогава, когато стигнем до формула φ_n, в която няма повече квантори с променливи от Θ'. Тъй като при всяко преименуване сме избирали напълно нова променлива, която не се среща нито в φ_i, нито е елемент на Θ, то φ_n ще отговаря на условията на лемата. ∎

2.7.13. Пример. Да видим с конкретен пример как работи току-що доказаната лема. Нека

$$\varphi = \forall \mathtt{y}\, \forall \mathtt{x}\, (\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{p}(\mathtt{x}, \mathtt{y})) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

и $\Theta = \{\mathtt{y}, \mathtt{z}\}$. В тази формула има много „дефекти“ — квантори с една и съща променлива ($\mathtt{x}$), променлива, която е хем свързана, хем свободна ($\mathtt{y}$), както и квантори с променливи от Θ. Нека поправим тези дефекти по начина, описан в доказателството на лемата. Нека най-напред изберем втория квантор $\forall \mathtt{x}$. Ще заменим променливата му $\mathtt{x}$ с $\mathtt{z}_{13}$ (избирама коя да е променлива, която не се среща нито във формулата,

нито е елемент на Θ). След замяната получаваме

$$\forall \mathtt{y}\, \forall \mathtt{x}\, (\forall \mathtt{z}_{13}\, \mathtt{p}(\mathtt{z}_{13}, \mathtt{y}) \vee \mathtt{p}(\mathtt{x}, \mathtt{y})) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

Сега да изберем другия квантор $\forall \mathtt{x}$. Ще заменим $\mathtt{x}$ например с $\mathtt{y}_{42}$. Получаваме

$$\forall \mathtt{y}\, \forall \mathtt{y}_{42}\, (\forall \mathtt{z}_{13}\, \mathtt{p}(\mathtt{z}_{13}, \mathtt{y}) \vee \mathtt{p}(\mathtt{y}_{42}, \mathtt{y})) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

Накрая да изберем $\forall \mathtt{y}$ и да заменим $\mathtt{y}$ с $\mathtt{z}'''$. Получаваме

$$\forall \mathtt{z}'''\, \forall \mathtt{y}_{42}\, (\forall \mathtt{z}_{13}\, \mathtt{p}(\mathtt{z}_{13}, \mathtt{z}''') \vee \mathtt{p}(\mathtt{y}_{42}, \mathtt{z}''')) \vee \mathtt{p}(\mathtt{y}, \mathtt{z})$$

Така получената формула нито съдържа квантори с една и съща променлива, нито има променлива, която е хем свързана, хем свободна, нито има квантори с променливи от Θ. Същевременно тази формула е конгруентна с φ, защото се получава с преименувания на свързаните променливи.

***Задача* 21:** Да се докаже, че ако $\varphi \equiv \psi$, то φ и ψ съдържат равен брой символи.

***Задача* 22:** Нека φ е формула, която не съдържа променливата $\mathtt{y}$ и ψ се получава от φ като заменим във φ всяко срещане на променливата $\mathtt{x}$ с $\mathtt{y}$. Да се докаже, че всяка свободна променлива на φ, която е различна от $\mathtt{x}$, е свободна променлива и на ψ.

***Задача* 23:** Нека φ съдържа подформула ψ и ψ' е конгруентна на ψ. Нека φ' се получава от φ като заменим подформулата ψ с ψ'. Да се докаже, че φ и φ' са конгруентни.

***Задача* 24:** Да се докаже, че

1. $\neg\varphi \equiv \neg\varphi' \longleftrightarrow \varphi \equiv \varphi'$
2. $\varphi \,\&\, \psi \equiv \varphi' \,\&\, \psi' \longleftrightarrow \varphi \equiv \varphi'$ и $\psi \equiv \psi'$
3. $\varphi \vee \psi \equiv \varphi' \vee \psi' \longleftrightarrow \varphi \equiv \varphi'$ и $\psi \equiv \psi'$
4. $\varphi \Rightarrow \psi \equiv \varphi' \Rightarrow \psi' \longleftrightarrow \varphi \equiv \varphi'$ и $\psi \equiv \psi'$

Сравнете тази задача със задача 32.

***Задача* 25:** Да се докаже, че ако $\forall \mathtt{x}\, \varphi \equiv \forall \mathtt{x}\, \psi$, то $\varphi \equiv \psi$.

2.8. Субституции

Субституция в изрази без свързани променливи

В предния раздел казахме, че свързаните променливи могат да се преименуват като това не променя смисъла на терма или формулата в която извършваме преименуването, а свободните променливи не могат да се преименуват без това да смени смисъла. Също така видяхме, че свободните променливи може се заменят с изрази, които не са променливи, докато замяната на една свързана променлива с израз, който не е променлива, със сигурност води до безсмислица.

В предния раздел дадохме точна дефиниция за това как могат да се преименуват свързаните променливи в една формула. Остава да видим как можем да заменяме свободните променливи с други термове. Най-напред да дефинираме понятието, което ще бъде играе главната роля в този раздел:

2.8.1. Дефиниция. *Субституцията* е функция, която на всяка променлива съпоставя терм.

Забележка: Тъй като субституциите се използват в различни алгоритми, дефиницията им обикновено включва допълнителното изискване, че само за краен брой променливи $s(\mathtt{x}) \neq \mathtt{x}$, а за всички останали $s(\mathtt{x}) = \mathtt{x}$. По този начин, за да представим в компютъра една субституция, е достатъчно да помним само двойките $\langle \mathtt{x}, s(\mathtt{x})\rangle$, при които $\mathtt{x} \neq s(\mathtt{x})$, а те са само краен брой. С цел да не усложняваме ненужно разсъжденията, в нашата дефиниция сме изпуснали това допълнително изискване. Въпреки това, ще отбележим, че всички субституции, които ще използваме на практика, всъщност удовлетворяват това допълнително изискване.

Тъй като в термовете и безкванторните формули няма свързани променливи, прилагането на субституция към такива изрази става по възможно най-простия и естествен начин — ако s е субституция, просто трябва да заменим в израза всяка променлива $\mathtt{x}$ с $s(\mathtt{x})$:

2.8.2. Дефиниция. Ако s е субституция, а τ — терм или предикатна формула без квантори, с $\tau[s]$ ще означим израза, който се получава като заместим в τ всяка променлива $\mathtt{x}$ с $s(\mathtt{x})$. Изразът $\tau[s]$ се нарича *резултат от прилагането на субституцията s към τ*.

2.8.3. Твърдение. *Нека s е субституция.*

а) Ако τ е терм, то $\tau[s]$ е терм.

б) Ако φ е безкванторна формула, то $\varphi[s]$ е безкванторна формула.

Доказателство. Следва от лема 2.8.4, която ще ни свърши работа и по-нататък. Твърдението обаче може да се докаже и без помощна лема с помощта на индукция. ∎

2.8.4. Лема. *а) Ако τ е терм и изразът τ' се получава като по напълно произволен начин заменим в τ някои променливи с термове, то τ' е терм.*

б) Ако φ е формула и изразът φ' се получава като по напълно произволен начин заменим в φ с термове някои от променливите, непосредствено пред които не стоят символите $\forall$ или $\exists$, то φ' е формула.

Доказателство. **(а)** С индукция по броя на символите в терма τ. Ако $\tau = \mathtt{x}$ е променлива, то τ' няма какво друго да бъде, освен $\mathtt{x}$ или терм с който променливата $\mathtt{x}$ е заменена. Ако $\tau = \mathtt{c}$ е константа, то $\tau' = \mathtt{c}$ и е терм. Ако $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$, то τ' има вида $\mathtt{f}(\tau_1', \tau_2', \ldots, \tau_n')$, където $\tau_1', \tau_2', \ldots, \tau_n'$ се получават от $\tau_1, \tau_2, \ldots, \tau_n$ по начина, описан в условието на лемата (заменяме произволни променливи с произволни термове). Съгласно индукционното предположение $\tau_1', \tau_2', \ldots, \tau_n'$ са термове, следователно по дефиниция 2.4.3 в) τ' също е терм.

(б) Аналогично. С индукция по броя на символите във формулата φ. Ако φ е атомарна формула и има вида $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$, то φ' има вида $\mathtt{p}(\tau_1', \tau_2', \ldots, \tau_n')$, където $\tau_1', \tau_2', \ldots, \tau_n'$ се получават от $\tau_1, \tau_2, \ldots, \tau_n$ по начина, описан в условието на лемата (заменяме произволни променливи с произволни термове). Съгласно вече доказаното $\tau_1', \tau_2', \ldots, \tau_n'$ са термове, следователно по дефиниция 2.5.1 φ' също е атомарна формула.

Ако $\varphi = \neg\psi$, то φ' има вида $\neg\psi'$. Съгласно индукционното предположение ψ' е формула, следователно по дефиниция 2.5.5 б) φ' също е формула.

Ако $\varphi = \psi_1 \,\&\, \psi_2$, то φ' има вида $\psi_1' \,\&\, \psi_2'$. Съгласно индукционното предположение ψ_1' и ψ_1' са формули, следователно по дефиниция 2.5.5 в) φ' също е формула.

Случаите когато φ има вида $\psi_1 \vee \psi_2$ или $\psi_1 \Rightarrow \psi_2$ се разглеждат аналогично.

Ако $\varphi = \forall\mathtt{x}\,\psi$, то съгласно условието на лемата не можем да заменяме променливата $\mathtt{x}$, пред която стои символът $\forall$, и значи φ' има вида $\forall\mathtt{x}\,\psi'$. Съгласно индукционното предположение ψ' е формула, следователно по дефиниция 2.5.5 г) φ' също е формула.

Случаят когато $\varphi = \exists\mathtt{x}\,\psi$ се разглежда аналогично. ∎

2.8.5. Означение. С $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n$ ще означаваме субституцията s, за която

$$s(\mathtt{z}) = \begin{cases} \tau_1, & \text{ако } \mathtt{z} = \mathtt{x}_1 \\ \tau_2, & \text{ако } \mathtt{z} = \mathtt{x}_2 \\ \ldots & \\ \tau_n, & \text{ако } \mathtt{z} = \mathtt{x}_n \\ \mathtt{z}, & \text{ако } \mathtt{z} \notin \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\} \end{cases}$$

2.8.6. Пример. Нека $s = (\mathtt{x}, \mathtt{y} \coloneqq \mathtt{f}(\mathtt{y}), \mathtt{x})$ и $\tau = \mathtt{g}(\mathtt{x}, \mathtt{y}, \mathtt{f}(\mathtt{x}))$. Тогава $\tau[s] = \mathtt{g}(\mathtt{f}(\mathtt{y}), \mathtt{x}, \mathtt{f}(\mathtt{f}(\mathtt{y})))$.

***Задача* 26:** Намерете

$$\mathtt{f}(\mathtt{x}, \mathtt{y})[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{y}, \mathtt{y}]$$
$$\mathtt{f}(\mathtt{x}, \mathtt{y})[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{g}(\mathtt{x}), \mathtt{x}]$$
$$\mathtt{f}(\mathtt{g}(\mathtt{x}), \mathtt{y})[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{g}(\mathtt{x}), \mathtt{x}]$$
$$\mathtt{f}(\mathtt{x}, \mathtt{g}(\mathtt{y}))[\mathtt{x}, \mathtt{y} \coloneqq \mathtt{g}(\mathtt{x}), \mathtt{x}]$$

Прилагане на субституция в общия случай

> Повечето формулировки на правилото за субституция, които са били публикувани — дори и от най-способните логици — преди 1940, са явно сбъркани.
>
> Хаскел Къри и Робер Фей [8]

Остава да видим как можем да прилагаме субституция към произволна предикатна формула. При термовете и безкванторните формули беше лесно. Например за произволна безкванторна формула φ ако s е субституция, то формулата $\varphi[s]$ се получава като заменим в φ всяка променлива $\mathtt{x}$ с $s(\mathtt{x})$. Можехме да използваме такава проста дефиниция, защото в термовете, атомарните формули и клаузите няма свързани променливи.

Да видим защо при наличието на свързани променливи прилагането на субституции трябва да се прави по-внимателно. Това е така, защото при прилагането на субституция към израз със свързани променливи може да се допуснат два вида грешки.

2.8.7. Първия вид грешка вече я споменахме — при прилагане на субституция не трябва да заменяме свързаните променливи, защото се

получават безсмислици. Например ако в израза

$$\sum_{i=1}^{100} i^2$$

заменим i с x^2, ще получим безсмислицата

$$\sum_{x^2=1}^{100} (x^2)^2$$

и ако в предикатната формула

$$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x})$$

заменим $\mathtt{x}$ с терма $\mathtt{f}(\mathtt{y})$, ще получим безсмислицата

$$\forall(\mathtt{f}(\mathtt{y}))\, \mathtt{p}(\mathtt{f}(\mathtt{y}))$$

Извод: Когато прилагаме субституция, трябва да игнорираме променливите, които са свързани от някой квантор. Например ако към формулата

$$\mathtt{p}(\mathtt{x}) \vee \forall \mathtt{x}\, \mathtt{q}(\mathtt{x})$$

искаме да приложим субституция, заменяща променливата $\mathtt{x}$ с терма $\mathtt{f}(\mathtt{y})$, ще получим формулата

$$\mathtt{p}(\mathtt{f}(\mathtt{y})) \vee \forall \mathtt{x}\, \mathtt{q}(\mathtt{x})$$

2.8.8. Вторият вид грешка е по-труден за откриване. Да предположим, че сме дефинирали числовата функция f така:

$$f(y) = \int_0^1 (x^2 + y^3) dx$$

Това, че в дефиницията на тази функция сме използвали променливата x не означава, че от тук нататък до края на света нямаме право да използваме променливи x за други цели. Да допуснем, че в даден момент сме дефинирали някаква променлива x. На колко ще бъде равно $f(2x)$? Ако просто навсякъде заменим y с $2x$ и кажем, че

$$f(2x) = \int_0^1 (x^2 + (2x)^3) dx$$

това ще бъде грешка! Да забележим, че изразът от дясната страна на равенството по никакъв начин не зависи от стойността на x, защото променливата x е свързана. Верният отговор е следният:

$$f(2x) = \int_0^1 (z^2 + (2x)^3)dz$$

Разбира се, вместо z можем да използваме коя да е друга променлива, която не се използва за други цели.

Ето друг пример. Нека

$$a_n = \sum_{i=1}^{n} i^3$$

Ако след време дефинираме променлива i, на колко ще бъде равно a_{2i}? Верният отговор е

$$a_{2i} = \sum_{j=1}^{2i} j^3$$

Тук също, вместо j можем да използваме коя да е друга променлива, която не се използва за други цели.

Извод: Когато прилагаме субституция, трябва да преименуваме кванторните променливи, ако има опасност някой квантор да „свърже“ променлива, която ще се постави от субституцията и трябва да остане свободна. Например ако към формулата

$$\forall \mathtt{y}\, \mathtt{q}(\mathtt{x}, \mathtt{y})$$

искаме да приложим субституция, заменяща променливата `x` с терма `f(y)`, ще трябва най-напред да преименуваме свързаната променлива `y` например на `z`

$$\forall \mathtt{z}\, \mathtt{q}(\mathtt{x}, \mathtt{z})$$

и едва след това ще може да заменим `x` с `f(y)`:

$$\forall \mathtt{z}\, \mathtt{q}(\mathtt{f}(\mathtt{y}), \mathtt{z})$$

Обмислете добре тези примери, преди да продължите четенето на този раздел! Опитайте се да решите следващата задача и проверете дали отговорът ви е бил верен.

***Задача* 27:** Още не сме дали точната дефиниция на $\varphi[s]$ при произволна субституция s и формула φ. Въпреки това, имайки предвид съображенията, изказани дотук, преценете какви трябва да бъдат

1. $(\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{q}(\mathtt{x}))[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{y})]$
2. $(\forall \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \vee \mathtt{q}(\mathtt{y}))[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{y})]$

И така, анализът на грешките от първи и втори тип (вж. 2.8.8 и 2.8.8) показва, че когато субституцията замества едни променливи с термове, съдържащи други променливи, тези променливи — както заменените, така и новопоявилите се — не трябва да имат нищо общо с кванторните променливи. Ако една заменена променлива е била в областта на действие на квантор, получаваме грешка от първи тип, а когато новопоявила се променлива попадне в областта на действие на квантор, получаваме грешка от втори тип.

За да избегнем грешките от първи тип, в дефиницията трябва да кажем, че заменяме само онези срещания на дадена променлива, които са свободни — в никакъв случай не трябва да променяме свързаните променливи. За да избегнем грешките от втори тип пък, ще трябва да видим кои ще са новите променливи, които ще се появят след като приложим субституцията, и да се погрижим никой от кванторите да не „свърже“ тези нови променливи.

Променливите, които заменя субституцията, са свободните променливи. Ако $\mathtt{z}$ е свободна променлива, тогава субституцията s ще я замени с терма $s(\mathtt{z})$. Следователно новите променливи, които ще се появят след като приложим субституцията, са всички променливи, които се срещат в термовете $s(\mathtt{z})$, където $\mathtt{z}$ е свободна променлива във формулата. Да наречем такива променливи „опасни“.

2.8.9. Дефиниция. Ще наричаме *опасни променливи* за формулата φ при субституция s всички променливи, които се срещат в термовете $s(\mathtt{z})$, където $\mathtt{z}$ е свободна променлива на φ.

Да забележим, че опасните променливи винаги са краен брой. В предния раздел видяхме, че свързаните променливи могат да се преименуват без това да промени смисъла на формулата. В частност доказахме лема 2.7.12, според която ако си изберем Θ да бъде множеството от опасните променливи, тогава можем да намерим конгруентна формула, която не съдържа нито един квантор, чиято променлива е от това множество. Именно това ни дава възможност да дефинираме прилагането на субституция към произволна формула. Ако във формулата няма квантори с опасни променливи, то няма проблеми да приложим субституцията към всички свободни променливи. В противен случай най-напред намираме конгруентна формула, която не съдържа квантори с опасни променливи и след това прилагаме субституцията към новата формула. Следващата дефиниция формализира всичко това.

2.8.10. Дефиниция. Нека s и φ са произволни субституция и формула, а Θ е крайното множество от всички опасни за φ променливи при субституция s. Ако φ не съдържа квантори с променливи от Θ, нека $\varphi' = \varphi$, а в противен случай нека φ' е конгруентна с φ формула, която не съдържа квантори с променливи от Θ (такава съществува благодарение на лема 2.7.12).

В такъв случай, с $\varphi[s]$ ще означим израза, който се получава като заместим във φ' всяка срещане на свободна променлива $\mathtt{z}$ с терма $s(\mathtt{z})$. Изразът $\varphi[s]$ се нарича *резултат от прилагането на субституцията s към φ*.

2.8.11. Да забележим, че според тази дефиниция заменяме само свободните променливи. Ако някое срещане на променливата $\mathtt{z}$ не е свободно, тогава на това място $\mathtt{z}$ не се заменя с терма $s(\mathtt{z})$. В дефиниция 2.8.2 нямаше нужда да правим специална уговорка, че прилагаме субституцията само към свободни променливи, защото в термовете и безкванторните формули всички променливи са свободни. Това впрочем показва, че току-що дадената дефиниция не противоречи на дефиниция 2.8.2, а само я допълва — да заменим всяка променлива $\mathtt{z}$ в терм или безкванторна формула с $s(\mathtt{z})$ е същото каквото да заменим всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$.

2.8.12. Пример. Нека s е субституция, за която

$$\begin{aligned}
s(\mathtt{x}_1) &= \mathtt{f}(\mathtt{x}_1 + \mathtt{x}_2, \mathtt{z}) \\
s(\mathtt{x}_2) &= \mathtt{y} + \mathtt{x}_3 \\
s(\mathtt{x}_3) &= \mathtt{y} \\
s(\mathtt{y}) &= \mathtt{f}(\mathtt{y}, \mathtt{y}) \\
s(\mathtt{z}) &= (\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1 \\
s(\mathtt{t}) &= \mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t})
\end{aligned}$$

Трябва да приложим тази субституция към формулата

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{y}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{t}))$$

Тъй като тази формула съдържа квантори, трябва да проверим дали някой от кванторите не е с опасна променлива. Свободни променливи са: $\mathtt{y}, \mathtt{x}_1, \mathtt{t}$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{y}), s(\mathtt{x}_1), s(\mathtt{t})$, т.е. променливите $\mathtt{y}, \mathtt{x}_1, \mathtt{x}_2, \mathtt{z}, \mathtt{t}$. Кванторът $\exists \mathtt{y}$ е с опасна променлива и трябва да го преименуваме. Например ако преименуваме неговата променлива $\mathtt{y}$ на $\mathtt{y}_{178}$ ще получим

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{y}) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{t}))$$

Към тази формула вече може да прилагаме субституцията s (заменяме само свободните променливи). Получаваме

$$\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{f}(\mathtt{y}, \mathtt{y})) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_3) = \mathtt{f}(\mathtt{x}_1 + \mathtt{x}_2, \mathtt{z}) + (\mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t}))))$$

***Задача* 28:** Нека s е същата субституция като в пример 2.8.12. Намерете резултатите от прилагането на субституцията s към следните формули:

$$\begin{gathered}
\mathtt{p}(\mathtt{x}_1, \mathtt{z}) \vee \mathtt{x}_1 + \mathtt{x}_2 = \mathtt{f}(\mathtt{t}, \mathtt{y}) \\
\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{t}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{t})) \\
\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{z}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_1 + \mathtt{z})) \\
\forall \mathtt{x}_3(\mathtt{p}(\mathtt{x}_3, \mathtt{z}) \Rightarrow \exists \mathtt{y}\,(\mathtt{f}(\mathtt{y}, \mathtt{x}_3) = \mathtt{x}_2 + \mathtt{z}))
\end{gathered}$$

Свойства на субституциите

> Това едва ли може да се нарече интуитивна дефиниция за субституция. Тя обаче има точните свойства.
>
> Джон Рейнолдс [16]

В дефиниция 2.8.10 нарекохме $\varphi[s]$ просто „израз“, а не формула. Всъщност $\varphi[s]$ е формула, но това се нуждае от доказателство.

2.8.13. Твърдение. *За всяка субституция s и формула φ изразът $\varphi[s]$ е формула.*

Доказателство. Ако във формулата φ няма квантори с опасни променливи при субституция s, тогава $\varphi[s]$ се получава просто като заменяме всяко срещане на свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$. Затова искането следва от лема 2.8.4.

Ако във формулата има квантори с опасни променливи, то съгласно дефиниция 2.8.10 трябва най-напред да преименуваме свързаните променливи по подходящ начин, получавайки конгруентна формула φ', след което действаме по вече описания начин. Затова в този случай $\varphi[s]$ също е формула. ∎

Вече споменахме, че ще искаме да отъждествяваме конгруентните формули. Това би създало проблем, ако се окаже, че като приложим някоя субституция към две конгруентни формули, е възможно да получим две съществено различни формули. Следващото твърдение показва, че това за щастие не се случва.

2.8.14. Твърдение. *Ако $\varphi_1 \equiv \varphi_2$, то $\varphi_1[s] \equiv \varphi_2[s]$. С други думи, ако формулите φ_1 и φ_2 са конгруентни, то за произволна субституция s, формулите $\varphi_1[s]$ и $\varphi_2[s]$ са конгруентни.*

Доказателството на това твърдение е сложно. Нека най-напред докажем възможно най-простия негов частен случай.

2.8.15. Лема. *Ако формулата φ' се получава от φ с еднократно преименуване на свързана променлива и нито във φ, нито във φ' има квантори с променливи, които са опасни при субституцията s, то формулата $\varphi'[s]$ се получава от $\varphi[s]$ с еднократно преименуване на свързана променлива.*

Доказателство. Съгласно дефиницията за еднократно преименуване на свързана променлива, формулите φ и φ' имат вида

$$\varphi = \ldots\ldots \forall \mathtt{x}\, \psi \ldots\ldots$$
$$\varphi' = \ldots\ldots \forall \mathtt{x}'\, \psi' \ldots\ldots$$

където на мястото на многоточията формулите φ и φ' съвпадат, а формулата ψ' се получава от ψ като заменим всяко срещане на променливата $\mathtt{x}$ с $\mathtt{x}'$.

Тъй като в тези формули няма квантори с опасни променливи, то субституцията s се прилага просто като заменим всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$. Това означава, че след като приложим тази субституция, няма да се появят различия на мястото на многоточията. Следователно формулите $\varphi[s]$ и $\varphi'[s]$ имат вида

$$\varphi[s] = \ldots\ldots (\forall \mathtt{x}\, \psi)[s] \ldots\ldots$$
$$\varphi'[s] = \ldots\ldots (\forall \mathtt{x}'\, \psi')[s] \ldots\ldots$$

където на мястото на многоточията формулите $\varphi[s]$ и $\varphi[s]'$ съвпадат. За да докажем лемата, е достатъчно да установим, че формулата $(\forall \mathtt{x}'\, \psi')[s]$ може да се получи от $(\forall \mathtt{x}\, \psi)[s]$ като заменим всяко срещане на $\mathtt{x}$ на $\mathtt{x}'$. За да видим, че това е така, нека забележим, че при прилагането на субституцията s към $\forall \mathtt{x}\, \psi$ и $\forall \mathtt{x}'\, \psi'$ не се случват никакви промени, свързани с променливите $\mathtt{x}$ и $\mathtt{x}'$. Наистина:

- във формулата $\forall \mathtt{x}\, \varphi$ не заместваме променливата $\mathtt{x}$, защото заместваме само свободни променливи, а тук $\mathtt{x}$ е свързана;
- във формулата $\forall \mathtt{x}\, \varphi$ не заместваме променливата $\mathtt{x}'$, защото тук тази променлива не се среща (съгласно дефиницията за еднократно преименуване);

– във формулата $\forall \mathtt{x}\, \varphi'$ не заместваме променливата $\mathtt{x}$, защото тук тази променлива не се среща (защото е вече е заменена с $\mathtt{x}'$);
– във формулата $\forall \mathtt{x}\, \varphi'$ не заместваме променливата $\mathtt{x}'$, защото заместваме само свободни променливи, а тук $\mathtt{x}'$ е свързана;
– термовете, които ще се появят, след като заменим всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$, няма да съдържат нито $\mathtt{x}$, нито $\mathtt{x}'$, защото в противен случай $\mathtt{x}$ или $\mathtt{x}'$ би била опасна променлива, а знаем, че няма квантори с опасни променливи.

■

Сега да усилим тази лема, така че в нея да се говори за конгруентност, а не за еднократно преименуване.

2.8.16. Лема. *Ако $\varphi_1 \equiv \varphi_2$ и нито φ_1, нито φ_2 съдържат квантори с опасни променливи при субституция s, то $\varphi_1[s] \equiv \varphi_2[s]$*

Доказателство. Съгласно дефиницията на конгруентни формули, съществува такава редица от формули

$$\varphi_1 = \psi'_1, \psi'_2, \ldots, \psi'_n = \varphi_2$$

че всеки член се получава от предходния посредством еднократно преименуване на свързана променлива. Нека Θ е множеството от всички променливи, които са опасни за φ_1 при субституция s. Съгласно твърдение 2.7.11, формулите $\varphi_1 = \psi'_1, \psi'_2, \ldots, \psi'_n = \varphi_2$ имат едни и същи свободни променливи, следователно Θ е множеството от опасните при субституция s променливи не само за φ_1, но също и за всяка от тези формули.

Нека $\mathtt{x}$ е някоя променлива от Θ, която се среща в квантор на някоя формула от тази редица и нека $\mathtt{y}$ е променлива, която не е елемент на Θ и не се среща никъде в тази редица. Може да забележим, че ако навсякъде в тази редица заменим променливата $\mathtt{x}$ с $\mathtt{y}$, отново ще получим редица от формули

$$\varphi_1 = \psi''_1, \psi''_2, \ldots, \psi''_n = \varphi_2$$

в която всеки член се получава от предходния посредством еднократно преименуване на свързана променлива. В новата редица обаче ще се срещат по-малко квантори с променливи от Θ. Ако повторим това преобразование нужния брой пъти, ще получим редица, в която всеки член се получава от предходния с еднократно преименуване на свързана променлива и никоя формула не съдържа квантори с опасни променливи. Затова към така получената редица може да приложим лема 2.8.15 и да получим исканото. ■

Вече сме да готови да докажем исканото твърдение. След като сме се потрудили с лемите, доказателството на самото твърдение не е дълго.

Доказателство на твърдение 2.8.14. Съгласно дефиницията за прилагане на субституция към произволна формула, има такива формули φ'_1 и φ'_2, че $\varphi_1 \equiv \varphi'_1$, $\varphi_2 \equiv \varphi'_2$, $\varphi_1[s] = \varphi'_1[s]$ и $\varphi_2[s] = \varphi'_2[s]$ и φ'_1 и φ'_2 не съдържат квантори с променливи, които са опасни при субституцията s. Тъй като φ_1 и φ_2, са конгруентни, то φ'_1 и φ'_2 също са конгруентни и затова получаваме исканото от лема 2.8.16. ∎

2.8.17. Твърдение. *Нека субституциите s_1 и s_2 съвпадат за всички свободни променливи на формулата φ. Тогава $\varphi[s_1] \equiv \varphi[s_2]$.*

Доказателство. Нека φ' е конгруентна с φ формула, която не съдържа квантори с променливи, които са опасни при субституция s_1. Формулата φ' има същите свободни променливи като φ, а за всяка такава свободна променлива субституциите s_1 и s_2 съвпадат. Това означава, че формулата φ' не съдържа квантори с опасни променливи също и при субституция s_2. От дефиницията за прилагане на субституция (дефиниция 2.8.10) следва, че $\varphi'[s_1] = \varphi'[s_2]$.

От тук получаваме исканото, защото от една страна $\varphi[s_1] \equiv \varphi'[s_1]$, а от друга $\varphi'[s_2] \equiv \varphi[s_2]$. ∎

2.8.18. Твърдение. *За всеки две субституции s_1 и s_2, съществува такава субституция s, че за всяка формула φ*

$$\varphi[s] \equiv (\varphi[s_1])[s_2]$$

Доказателство. Нека s е субституцията, за която $s(\mathtt{z}) = (s_1(\mathtt{z}))[s_2]$ за всяка променлива $\mathtt{z}$. Ще докажем, че за произволна формула $\varphi[s] \equiv (\varphi[s_1])[s_2]$.

Нека Θ е множеството от свободните променливи на φ. Нека Θ_1 е обединението на Θ с множеството от всички променливи, които се срещат в термовете $s_1(\mathtt{z})$, където $\mathtt{z} \in \Theta$. С други думи, Θ_1 е множеството от опасните променливи в φ при субституция s_1. Нека Θ_2 е обединението на Θ_1 с множеството от всички променливи, които се срещат в термовете $s_2(\mathtt{z})$, където $\mathtt{z} \in \Theta_1$. Множеството Θ_2 е крайно, значи от лемата за преименуване на свързаните променливи 2.7.12 можем да намерим формула ψ, която е конгруентна с φ, и която не съдържа квантори с променливи от Θ_2.

Тъй като Θ_1 е множеството от опасните променливи за φ при субституция s_1 и $\Theta_1 \subseteq \Theta_2$, то ψ не съдържа квантори с променливи, които

са опасни при субституция s_1. Следователно $\psi[s_1]$ се получава като заменим в ψ всяка свободна променлива $\mathtt{x}$ с терма $s_1(\mathtt{x})$. Това означава, че свободните променливи на $\psi[s_1]$ са елементи на Θ_1, а формулата $\psi[s_1]$ не съдържа квантори с променливи, които са опасни при субституция s_2, следователно $(\psi[s_1])[s_2]$ се получава от $\psi[s_1]$ като заменим всяка свободна променлива $\mathtt{y}$ с терма $s_2(\mathtt{y})$. Тъй като всички свободни променливи в $\psi[s_1]$ се съдържат в термовете $s_1(\mathtt{x})$, които са заменили свободните променливи $\mathtt{x}$ в ψ, то това означава, че $(\psi[s_1])[s_2]$ може да се получи от ψ като заменим всяка свободна променлива $\mathtt{x}$ в ψ с терма $(s_1(\mathtt{x}))[s_1]$.

Тъй като ψ не съдържа квантори с променливи от Θ_2, то ψ не съдържа квантори с променливи, които са опасни при субституция s. Следователно формулата $\psi[s]$ може да се получи от ψ като заменим всяка свободна променлива $\mathtt{x}$ в ψ с терма $s(\mathtt{x})$, т.е. с терма $(s_1(\mathtt{x}))[s_1]$.

Доказахме, че $\psi[s] = (\psi[s_1])[s_2]$, откъдето следва $\varphi[s] \equiv (\varphi[s_1])[s_2]$.

■

***Задача* 29:** Да се докаже, че $\mathtt{x}$ е свободна променлива на формулата $\varphi[s]$ тогава и само тогава, когато $\mathtt{x}$ се среща в терм от вида $s(\mathtt{z})$, където $\mathtt{z}$ е свободна променлива на φ.

****Задача* 30:** Нека променливата $\mathtt{y}$ не се среща никъде във формулата φ и формулата ψ се получава от φ като заменим всяко срещане (свободно и свързано) на променливата $\mathtt{x}$ с $\mathtt{y}$. Да се докаже, че $\psi \equiv \varphi[\mathtt{x} \coloneqq \mathtt{y}]$.

***Задача* 31:** Да се докаже, че

1. $(\neg\varphi)[s] \equiv \neg(\varphi[s])$
2. $(\varphi \,\&\, \psi)[s] \equiv \varphi[s] \,\&\, \psi[s]$
3. $(\varphi \vee \psi)[s] \equiv \varphi[s] \vee \psi[s]$
4. $(\varphi \Rightarrow \psi)[s] \equiv \varphi[s] \Rightarrow \psi[s]$

****Задача* 32:** Да се докаже, че $\forall\mathtt{x}\,\varphi \equiv \forall\mathtt{y}\,\psi$ тогава и само тогава, когато $\psi \equiv \varphi[\mathtt{x} \coloneqq \mathtt{y}]$. Сравнете тази задача със задача 24.

****Задача* 33:** Да се докаже, че ако $\forall\mathtt{x}\,\varphi \equiv \forall\mathtt{x}'\varphi'$ и $\forall\mathtt{x}\,\psi \equiv \forall\mathtt{x}'\psi'$, то $\forall\mathtt{x}\,(\varphi \,\&\, \psi) \equiv \forall\mathtt{x}'(\varphi' \,\&\, \psi')$.

Глава 3

Семантика

> Семантиките са странен вид приложна математика; тя се интересува от прозорливи дефиниции вместо от трудни теореми.
>
> Джон Рейнолдс (според [20, стр. 3])

3.1. Структури

Вече имахме възможност да работим неформално с някои структури. Например всички разгледани моноиди и полупръстени бяха примери за структури. Видяхме, че един и същи терм може да има различни стойности в различните структури, а една формула може да бъде вярна в една структура и невярна в друга. Все още обаче не сме дали точна математическа дефиниция нито на това що е структура, нито на това как да пресмятаме стойността на терм в структура, нито пък на това кога една формула е вярна в структура.

За да се ориентираме по-добре как точно трябва да дадем тези дефиниции, нека разгледаме формулата

$$\mathtt{x} \leqq \mathtt{y} \Rightarrow \mathtt{x.z} \leqq \mathtt{y.z} \tag{9}$$

и да видим какъв е нейният смисъл в някои конкретни структури.

1. Ако структурата включва реалните числа и символите „$\leqq$“ и „$.$“ са интерпретирани по обичайния начин като „по-малко или равно“ и

умножение, тогава тази формула не е вярна. Например при $\mathtt{x} = 2$, $\mathtt{y} = 3$ и $\mathtt{z} = -1$ получаваме $2 \leqq 3 \Rightarrow -2 \leqq -3$, което е лъжа.

2. Ако пък вместо за реалните числа структурата е за естествените числа, тази формула става вярна, защото стойността на $\mathtt{z}$ не може да е отрицателно число и значи при умножение неравенството не може да си смени посоката.
3. Обаче никъде не е казано, че символите „$\leqq$“ и „$.$“ трябва да се интерпретират по обичайния начин. Ако структурата е както в 2, но символът „$\leqq$“ се интерпретира не като „по-малко или равно“, а като „по-малко“, тогава формулата отново става невярна, защото при $\mathtt{z} = 0$ заключението на импликацията ще бъде $0 < 0$.
4. Но дори да интерпретираме „$\leqq$“ като „строго по-малко“, ако освен това интерпретираме „$.$“ не като умножение, а като събиране, тогава формулата отново става вярна.

Разгледаната формула показва какви свойства трябва да има понятието „структура“. *Първо,* структурата трябва да определя какъв е смисълът на предикатните символи (напр. „$\leqq$“). *Второ,* структурата трябва да определя какъв е смисълът на функционалните символи (напр. „$.$“). *Трето,* въпреки че в разгледаната формула нямаше символи за константи, структурата трябва да определя смисълът и на тези символи (напр. 0 и 1). Да забележим още, че структурата не трябва да дава **конкретни** стойности на променливите (в случая $\mathtt{x}$, $\mathtt{y}$ и $\mathtt{z}$). Вместо това структурата трябва да определя какви са всички **възможни** стойности на променливите.

3.1.1. Означение. а) Множеството от всички възможни стойности на променливите в дадена структура $\mathbf{M}$ се означава с $|\mathbf{M}|$ и се нарича *универсум*[*] на структурата.[**]

б) Смисълът или *интерпретацията* в дадена структура $\mathbf{M}$ на символите за константи, функционалните символи и предикатните символи ще означаваме с горен индекс $^{\mathbf{M}}$. Например

– $\mathtt{c}^{\mathbf{M}}$ е интерпретацията на символа за константа $\mathtt{c}$ в структурата $\mathbf{M}$,

[*] То се нарича още и *носител*, *основа*, *област* или *домейн* на структурата. Едва ли в математиката има друга област, където имената и означенията на основните понятия са толкова малко стандартизирани, колкото в математическата логика...

[**] Някои математици използват за структурите удебелени латински букви — $\mathbf{A}$, $\mathbf{B}$, $\mathbf{M}$, $\mathbf{N}$, други калиграфски букви — $\mathcal{A}$, $\mathcal{B}$, $\mathcal{M}$, $\mathcal{N}$, трети готически — $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{M}$, $\mathfrak{N}$. Универсумът на структурите се означава или с вертикални черти — $|\mathbf{A}|$, $|\mathbf{B}|$, $|\mathbf{M}|$, $|\mathbf{N}|$, или посредством съответните обикновени латински букви — A, B, M, N.

– $\mathtt{f}^{\mathbf{M}}$ е интерпретацията на функционалния символ $\mathtt{f}$ в структурата $\mathbf{M}$ и
– $\mathtt{p}^{\mathbf{M}}$ е интерпретацията на предикатния символ $\mathtt{p}$ в структурата $\mathbf{M}$.

По отношение на това какъв точно може да бъде универсумът на една структура и как трябва да се интерпретират различните символи в нея, да обърнем внимание на следното.

Универсумът е множество. Не се налага да ограничаваме по какъвто и да е начин какво точно съдържа това множесто — елементите му могат да бъдат числа, функции, други множества и т.н. Единственото ограничение, което се налага да приемем, е това универсумът да бъде непразно множество. Причината, поради която налагаме това ограничение, е следната — когато универсумът е празното множество, се появяват някои странности, които неужно ще усложняват формулировките на твърденията и техните доказателства. И тъй като структурите с празен универсум очевидно не могат да бъдат кой знае колко полезни, си струва да си спестим тези усложнения като забраним структурите с празен универсум.

Интерпретацията на символите за константи трябва да бъде елемент на универсума на структурата. Например ако универсумът на структурата $\mathbf{M}$ е множеството на реалните числа (т.е. $|\mathbf{M}| = \mathbb{R}$) и 1 е символ за константа, тогава $1^{\mathbf{M}} \in \mathbb{R}$. Ако пък универсумът е множеството на естествените числа, тогава $1^{\mathbf{M}} \in \mathbb{N}$. И т.н.

Интерпретацията на един n-местен функционален символ трябва да бъде функция с n аргумента. В по-горния пример това се вижда от интерпретацията на двуместния функционален символ „.“. Във всяка една структура този функционален символ се интерпретира като функция с два аргумента, например като функцията умножение на две реални числа. Вижда се, че както аргументите, така и стойността на тази функция са елементи на универсума на структурата.

Интерпретацията на един n-местен предикатен символ също трябва да бъде функция с n аргумента. В по-горния пример това се вижда от интерпретацията на двуместния предикатен символ „$\leqq$“. Пак от по-горния пример се вижда и кое отличава функционалните от предикатните символи — функциите, интерпретиращи функционални символи, връщат стойности от универсума на структурата, докато функциите, интерпретиращи предикатни символи връщат като стойност някое съждение. Например без значение коя е структурата и как точно в нея се интерпретира предикатният символ „$\leqq$“ и без значение каква е стойността на променливите $\mathtt{x}$ и $\mathtt{y}$, стойността на $\mathtt{x} \leqq \mathtt{y}$ не е елемент на

универсума, а конкретно съждение, което може да бъде вярно, може да бъде невярно и не може да не е нито вярно, нито невярно.

3.1.2. Пример. В първата от структурите, в която интерпретирахме формулата

$$\mathtt{x} \leqq \mathtt{y} \Rightarrow \mathtt{x.z} \leqq \mathtt{y.z} \tag{9}$$

универсум беше множеството от реалните числа, функционалният символ „.“ се интерпретираше като функцията умножение на реални числа, а предикатният символ „$\leqq$“ се интерпретираше като двуместна функция, която за произволни аргументи x и y връща като стойност съждението „x е по-малко или равно на y“.

Следващата дефиниция прояснява произхода на имената на различните видове символи. Тя показва, че символите за константи се интерпретират като конкретни елементи на универсума, т.е. като константи в универсума, функционалните символи се интерпретират като функции в универсума и предикатните символи се интерпретират като предикати в универсума.

3.1.3. Дефиниция. Нека X е произволно множество.

а) n-местна *функция* в X означава n-местна функция с аргументи от X и стойност в X.

б) n-местен *предикат* в X означава n-местна функция с аргументи от X, която приема като стойност някое съждение.

В съвременната математическа традиция не е прието съжденията да се използват като пълноправни математически обекти. Това означава, че дадената по-горе дефиниция на „n-местен предикат“ е донякъде неприемлива. Това затруднение може да се избегне по два начина. Първият начин е да дефинираме предикатите като функции, които вместо съждения връщат стойност „И“ или „Л“, „true“ или „false“, 1 или 0. Вторият начин се състои в използването на релации вместо предикати.

3.1.4. Дефиниция. n-местна *релация* в X е подмножество на $X^n = \underbrace{X \times X \times \cdots \times X}_{n \text{ пъти}}$.

3.1.5. Вижда се, че в дефиницията на понятието „релация“ се използват множества. Оказва се, че винаги когато имаме нужда да използваме съждения като математически обекти, има начин това да бъде избегнато посредством подходящо използване на множества. Конкретно за

предикатите и релациите може да забележим, че тези две понятия са взаимнозаменяеми по следния начин.

Нека p е произволен n-местен предикат в X. В такъв случай съответната му n-местна релация е множеството, което съдържа всички n-торки $\langle x_1, x_2, \ldots, x_n\rangle$, за които е вярно съждението $p(x_1, x_2, \ldots, x_n)$.

И обратно, ако ни е дадена n-местна релация R в X, то на нея ѝ съответства предикатът, който за произволни аргументи $x_1, x_2, \ldots, x_n$ връща като стойност съждението $\langle x_1, x_2, \ldots, x_n\rangle \in R$.

3.1.6. Пример (допълнение на пример 3.1.2). Ако решим да интерпретираме предикатните символи не с предикати, а с релации, то в пример 3.1.2 ще трябва да интерпретираме предикатния символ $\leqq$ не като функция, която за аргументи x и y връща като стойност съждението „x е по-малко или равно на y“, а като множеството от всички двойки $\langle x, y\rangle$, за които е вярно това съждение.

Вече сме готови да дадем точната дефиниция на структура.

3.1.7. Дефиниция. Нека $\mathfrak{sig}$ е сигнатура. Наредената двойка $\mathbf{M} = \langle \Omega, \mathfrak{i}\rangle$ е *структура* за $\mathfrak{sig}$, ако Ω е непразно множество, наречено *универсум* на структурата, а $\mathfrak{i}$ е функция, наречена *интерпретация*, която притежава изброените по-долу свойства а), б) и в). Множеството Ω се означава с $|\mathbf{M}|$ и за произволен символ $\mathtt{k}$ вместо $\mathfrak{i}(\mathtt{k})$ ще пишем $\mathtt{k}^{\mathbf{M}}$.

а) Ако $\mathtt{c}$ е символ за константа, то $\mathtt{c}^{\mathbf{M}} = \mathfrak{i}(\mathtt{c})$ е елемент на $|\mathbf{M}|$.

б) Ако $\mathtt{f}$ е n-местен функционален символ, то $\mathtt{f}^{\mathbf{M}} = \mathfrak{i}(\mathtt{f})$ е n-местна функция в $|\mathbf{M}|$.

в) Ако $\mathtt{p}$ е n-местен предикатен символ, то $\mathtt{p}^{\mathbf{M}} = \mathfrak{i}(\mathtt{p})$ е n-местен предикат в $|\mathbf{M}|$ или n-местна релация в $|\mathbf{M}|$.

Интерпретация на символите за константи и функционалните символи в структура:

$$\mathtt{c}^{\mathbf{M}} \in |\mathbf{M}|$$
$$\mathtt{f}^{\mathbf{M}} \colon |\mathbf{M}|^n \to |\mathbf{M}|$$

Интерпретация на предикатните символи:

$$\mathtt{p}^{\mathbf{M}} \colon |\mathbf{M}|^n \to \text{ съждение} \qquad \text{(с предикат)}$$
$$\mathtt{p}^{\mathbf{M}} \subseteq |\mathbf{M}|^n \qquad \text{(с релация)}$$

Съгласно забележка 3.1.5 е все едно дали ще интерпретираме предикатните символи с предикати или с релации. По подразбиране в този

курс ще използваме структури, в които предикатните символи се интерпретират с предикати .

3.1.8. Дефиниция. а) Когато сигнатурата съдържа двуместен предикатен символ $=$, а структурата интерпретира този символ като равенство, то за тази структура казваме, че е *структура с равенство*.

б) Когато сигнатурата е такава, че единствен предикатен символ в нея е символът $=$ и структурата интерпретира този символ като равенство, тогава такава структура се нарича *алгебрична структура* или просто *алгебра*. Причината за тази терминология е това, че повечето от структурите, които се използват в алгебрата, са точно такива.*

3.1.9. Пример. Магмите, полугрупите, моноидите, групите, квазигрупите, полупръстените, пръстените, пръстените на Ли, полетата, алгебрите на Клини, полурешетките, решетките и булевите алгебри са само някои от многото примери за алгебрични структури. Например пръстенът на целите числа е структура, чийто универсум е множеството на целите числа, има два символа за константи 0 и 1, които се интерпретират стандартно като числото нула и числото едно, един едноместен функционален символ „$-$“, който се интерпретира като операцията „смяна на знака“, два двуместни функционални символа „$+$“ и „$.$“, които се интерпретират като операциите събиране и умножение и един предикатен символ $=$, който е двуместен и се интерпретира като равенство.

3.1.10. Пример. Графите са пример за структури, които не са алгебрични. Един *граф* $\mathbf{G}$ може да се мисли като структура, чийто универсум е множеството от върховете на графа, символи за константи и функционални символи няма и има единствен предикатен символ е $\mathtt{p}$, който е двуместен и за всеки два върха v_1 и v_2:

- ако интерпретираме $\mathtt{p}$ с предикат, то $\mathtt{p}^{\mathbf{G}}(v_1, v_2)$ е истина тогава и само тогава, когато има ребро от v_1 до v_2;
- ако интерпретираме $\mathtt{p}$ с релация, то $\mathtt{p}^{\mathbf{G}} \ni \langle v_1, v_2 \rangle$ е истина тогава и само тогава, когато има ребро от v_1 до v_2, т.е.

$$\mathtt{p}^{\mathbf{G}} = \{\, \langle v_1, v_2 \rangle \mid \text{има ребро от } v_1 \text{ до } v_2 \,\}$$

*Понякога структурите в алгебрата са снабдени с някаква наредба. Строго погледнато, такива структури не са алгебрични.

3.1.11. Пример. Нека сигнатурата $\mathfrak{sig}$ е такава, че в нея има единствен символ за константа $\mathtt{c}$, единствен функционален символ $\mathtt{f}$, който е двуместен, и единствен предикатен символ $\mathtt{p}$, който също е двуместен. Нека структурата $\mathbf{M}$ за $\mathfrak{sig}$ е с универсум множеството на реалните числа, $\mathtt{c}^{\mathbf{M}} = 3$, $\mathtt{f}^{\mathbf{M}}(x, y) = x + y$ и $\mathtt{p}^{\mathbf{M}}(x, y)$ е съждението „$x < y$“. В такъв случай формулата

$$\mathtt{p}(\mathtt{c}, \mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{c}, \mathtt{f}(\mathtt{x}, \mathtt{x}))$$

казва, че ако едно реално число x е по-голямо от 3, то $x + x$ също е по-голямо от 3.

***Задача* 34:** Какво казват в структурата от пример 3.1.11 следните формули:

$$\mathtt{p}(\mathtt{x}, \mathtt{y}) \,\&\, \mathtt{p}(\mathtt{y}, \mathtt{z}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{z}) \tag{10}$$

$$\mathtt{p}(\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{p}(\mathtt{f}(\mathtt{x}, \mathtt{z}), \mathtt{f}(\mathtt{y}, \mathtt{z})) \tag{11}$$

$$\mathtt{p}(\mathtt{c}, \mathtt{c}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{x}) \tag{12}$$

***Задача* 35:** В пример 3.1.10 видяхме, че графите може да се мислят като специален вид структури. Напишете клауза, която е вярна в един граф тогава и само тогава, когато

1. графът е неориентиран;
2. графът е пълен

3.2. Оценки

В предния раздел дадохме дефиницията на структура, но все още не сме дали точна дефиниция за това какво значи една формула да бъде вярна в структура. В пример 3.1.11 разчитахме повече на интуицията си и здравия смисъл, отколкото на някаква точна математическа дефиниция.

За да дефинираме какво значи една формула да бъде вярна структура, е нужно да се научим да намираме стойността на термовете в структурата. За да пресметнем стойността на един терм обаче, не е достатъчно да знаем коя е структурата. Да разгледаме например терма $\mathtt{x} + \mathtt{x}$. Структурата ще ни каже какъв е смисълът на функционалният символ $+$, но не и стойността на променливата $\mathtt{x}$. А при различни стойности на променливата $\mathtt{x}$, стойността на този терм най-вероятно ще бъде различна. Затова ще дефинираме понятието оценка на променливите:

3.2.1. Дефиниция. Функцията v е *оценка* в структурата $\mathbf{M}$, ако v съпоставя на всяка променлива елемент на универсума на $\mathbf{M}$.

Нека например структурата $\mathbf{M}$ е с универсум $\mathbb{R}$ и $+^{\mathbf{M}}$ е функцията събиране на реални числа. Нека оценката v в $\mathbf{M}$ е такава, че $v(\mathtt{x}) = 5$ и $v(\mathtt{y}) = 3$. Тогава стойността на терма $\mathtt{x} + \mathtt{y}$ в структурата $\mathbf{M}$ при оценка v трябва да бъде числото 8. Да забележим, че за да кажем каква трябва да бъде стойността на терма $\mathtt{x} + \mathtt{y}$, не бе нужно да знаем на колко е равно $v(\mathtt{z})$ за променливи $\mathtt{z}$, различни от $\mathtt{x}$ и $\mathtt{y}$. Единствените променливи, които имат отношение към стойността на терма $\mathtt{x} + \mathtt{y}$ са променливите, срещащи се в този терм, т.е. $\mathtt{x}$ и $\mathtt{y}$.

***Задача* 36:** Докажете, че за произволна структура $\mathbf{M}$ съществува поне една оценка в $\mathbf{M}$.

Точната дефиниция за стойност на терм е следната:

3.2.2. Дефиниция. Нека $\mathbf{M}$ е структура за сигнатурата $\mathfrak{sig}$ и v е оценка в $\mathbf{M}$. *Стойността* в структурата $\mathbf{M}$ при оценка v на термовете при сигнатура $\mathfrak{sig}$ се дефинира индуктивно:

- ако $\mathtt{x}$ е променлива, то стойността на терма $\mathtt{x}$ е $v(\mathtt{x})$;
- ако $\mathtt{c}$ е символ за константа, то стойността на терма $\mathtt{c}$ е $\mathtt{c}^{\mathbf{M}}$;
- ако $\mathtt{f}$ е n-местен функционален символ и $\tau_1, \tau_2, \ldots, \tau_n$ са термове, то стойността на терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ е равна на $\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$, където μ_i е стойността на τ_i в $\mathbf{M}$ при оценка v.

Да обърнем внимание, че стойността на терм в структура е дефинирана само ако термът и структурата са за една и съща сигнатура. Например няма как да пресметнем стойността на терма $\mathtt{x}+\mathtt{y}$, ако структурата е за сигнатура, в която няма функционален символ $+$. Стойността на атомарна формула също е дефинирана само ако атомарната формула и структурата са за една и съща сигнатура.

3.2.3. Означение. Нека $\mathbf{M}$ е структура и τ е терм. *Смисълът* или *денотацията* на терма τ в структурата $\mathbf{M}$ е функция, означавана с $[\![\tau]\!]^{\mathbf{M}}$, чийто аргумент е оценка v в $\mathbf{M}$, а стойността ѝ е равна на стойността на τ в структурата $\mathbf{M}$ при оценка v. Следователно можем да означаваме стойността на терм τ в структура $\mathbf{M}$ при оценка v с $[\![\tau]\!]^{\mathbf{M}}(v)$.*

*Едно от често използваните в алгебрата означения за стойността на терм τ в структура $\mathbf{M}$ при оценка v е $\tau^{\mathbf{M}}[v]$. За съжаление в математическата логика се използват различни означения за стойността на терм. Означението $[\![\tau]\!]^{\mathbf{M}}(v)$, което използваме в този курс, е заимствано от теорията на езиците за програмиране.

3.2.4. ***Забележка*:** Ако временно означим стойността в структура $\mathbf{M}$ при оценка v на произволен терм τ с $\overline{\tau}$, тогава предната дефиниция може да се преформулира по следния начин:

- $\overline{\mathtt{x}} = v(\mathtt{x})$ за всяка променлива $\mathtt{x}$;
- $\overline{\mathtt{c}} = \mathtt{c}^{\mathbf{M}}$ за всеки символ за константа $\mathtt{c}$;
- $\overline{\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)} = \mathtt{f}^{\mathbf{M}}(\overline{\tau_1}, \overline{\tau_2}, \ldots, \overline{\tau_n})$ за всеки n-местен функционален символ $\mathtt{f}$ и термове $\tau_1, \tau_2, \ldots, \tau_n$.

Това показва, че пресмятането на стойността на терм представлява вкарване на хоризонталните черти навътре в подизразите като при това заменяме всеки символ за константа $\mathtt{c}$ с $\mathtt{c}^{\mathbf{M}}$, всеки функционален символ $\mathtt{f}$ с $\mathtt{f}^{\mathbf{M}}$ и всяка променлива $\mathtt{x}$ с $v(\mathtt{x})$. Например

$$\begin{aligned}\overline{\mathtt{g}(\mathtt{f}(\mathtt{x},\mathtt{c}),\mathtt{y})} &\longleftrightarrow \mathtt{g}^{\mathbf{M}}(\overline{\mathtt{f}(\mathtt{x},\mathtt{c})},\overline{\mathtt{y}})\\ &\longleftrightarrow \mathtt{g}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(\overline{\mathtt{x}},\overline{\mathtt{c}}),v(\mathtt{y}))\\ &\longleftrightarrow \mathtt{g}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(v(\mathtt{x}),\mathtt{c}^{\mathbf{M}}),v(\mathtt{y}))\end{aligned}$$

3.2.5. ***Забележка*:** Използвайки стандартните ни означения, дефиниция 3.2.2 може да бъде преформулирана и по следния начин:

- $[\![\mathtt{x}]\!]^{\mathbf{M}}(v) = v(\mathtt{x})$ за всяка променлива $\mathtt{x}$;
- $[\![\mathtt{c}]\!]^{\mathbf{M}}(v) = \mathtt{c}^{\mathbf{M}}$ за всеки символ за константа $\mathtt{c}$;
- $[\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)]\!]^{\mathbf{M}}(v) = \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_2]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v))$ за всеки n-местен функционален символ $\mathtt{f}$ и термове $\tau_1, \tau_2, \ldots, \tau_n$.

Едно важно свойство на стойността на терм е това, че стойността не зависи от променливите, които не се срещат в терма. Например за да пресметнем стойността на терма $\mathtt{x} + (\mathtt{y} + \mathtt{y})$, не е нужно да знаем каква стойност има променливата $\mathtt{z}$. Да докажем този факт.

3.2.6. Твърдение. *Нека v и v' са оценки в структурата $\mathbf{M}$. Ако $v(\mathtt{x}) = v'(\mathtt{x})$ за всяка променлива $\mathtt{x}$, която се среща в терма τ, то $[\![\tau]\!]^{\mathbf{M}}(v) = [\![\tau]\!]^{\mathbf{M}}(v')$.*

Доказателство. С индукция по терма τ, използвайки изискванията на дефиницията за стойност на терм (вж. забележка 3.2.5).

Ако $\tau = \mathtt{x}$ е променлива, то $[\![\tau]\!]^{\mathbf{M}}(v) = [\![\mathtt{x}]\!]^{\mathbf{M}}(v) = v(\mathtt{x})$. Тъй като по условие v и v' съвпадат за променливи, срещащи се в τ (а в случая за $\mathtt{x} = \tau$), то последното е равно на $v'(\mathtt{x}) = [\![\mathtt{x}]\!]^{\mathbf{M}}(v') = [\![\tau]\!]^{\mathbf{M}}(v')$.

Ако $\tau = \mathtt{c}$ е символ за константа, то както $[\![\tau]\!]^{\mathbf{M}}(v) = [\![\mathtt{c}]\!]^{\mathbf{M}}(v)$, така и $[\![\tau]\!]^{\mathbf{M}}(v') = [\![\mathtt{c}]\!]^{\mathbf{M}}(v')$ е равно на $\mathtt{c}^{\mathbf{M}}$.

Нека $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Тогава

$$[\![\tau]\!]^{\mathbf{M}}(v) = \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_2]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v))$$
$$[\![\tau]\!]^{\mathbf{M}}(v') = \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v'), [\![\tau_2]\!]^{\mathbf{M}}(v'), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v'))$$

Изразите отдясно на равенствата обаче са равни, защото съгласно индукционното предположение $[\![\tau_i]\!]^{\mathbf{M}}(v) = [\![\tau_i]\!]^{\mathbf{M}}(v')$. ∎

***Задача* 37:** Ако термът τ не съдържа нито една променлива, то за произволна структура $\mathbf{M}$ и оценки v и v' в $\mathbf{M}$, стойността на τ в структурата $\mathbf{M}$ при оценка v е равна на стойността на τ в $\mathbf{M}$ при оценка v'.

Тъй като стойността на терм зависи само от стойността на променливите, срещащи се в терма, то ще дефинираме понятието „частична оценка“.

3.2.7. Дефиниция. *Частична оценка* в структурата $\mathbf{M}$ е функция, чиято дефиниционна област съдържа само променливи, която на всяка променлива от дефиниционната си област съпоставя елемент от универсума на $\mathbf{M}$.

Разбира се, всяка оценка в $\mathbf{M}$ е и частична оценка в $\mathbf{M}$, но не всяка частична оценка в $\mathbf{M}$ е оценка в $\mathbf{M}$.

3.2.8. Дефиниция. Нека частичната оценка v в структура $\mathbf{M}$ е дефинирана за всички променливи, срещащи се в терм τ. Тогава стойността на τ в структурата $\mathbf{M}$ при частична оценка v ще дефинираме да бъде равна на стойността на τ при коя да е оценка v', която съвпада с v за променливите, срещащи се в τ. Да забележим, че съгласно твърдение 3.2.6 така дефинираната стойност не зависи от избора на v'. Стойността на τ в структура $\mathbf{M}$ при частична оценка v означаваме също както стойността при обикновена оценка: $[\![\tau]\!]^{\mathbf{M}}(v)$.

3.2.9. Означение. За произволни различни променливи $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на структура $\mathbf{M}$, с

$$\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n$$

ще означаваме частичната оценка v, чиято дефиниционна област е множеството $\{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$ и за която $v(\mathtt{x}_i) = \mu_i$ за всяко $i \in \{1, 2, \ldots, n\}$.

3.2.10. Означение. а) Когато τ е терм, ще използваме записа

$$\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$$

когато искаме да кажем, че променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са различни по между си и всички променливи, които се срещат в τ, са измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

б) Нека $\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е терм и $\mu_1, \mu_2, \ldots, \mu_n$ са произволни елементи на универсума на структурата $\mathbf{M}$. Ще пишем

$$[\![\tau]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$$

вместо

$$[\![\tau]\!]^{\mathbf{M}}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n)$$

***Задача* 38:** Нека структурата $\mathbf{M}$ е с универсум множеството на реалните числа, двуместният функционален символ $\mathtt{f}$ се интерпретира като функцията събиране (т.е. $\mathtt{f}^{\mathbf{M}}(a, b) = a + b$ за произволни $a, b \in \mathbb{R}$) и двуместният функционален символ $\mathtt{g}$ се интерпретира като умножение (т.е. $\mathtt{f}^{\mathbf{M}}(a, b) = ab$ за произволни $a, b \in \mathbb{R}$). Намерете такъв терм $\tau(\mathtt{x}_1, \mathtt{x}_2)$, че

$$[\![\tau]\!]^{\mathbf{M}}(a, b) = a^2 + b$$

за произволни реални числа a и b.

3.3. Вярност на формула в структура

Да си припомним, че стойността на един терм в структура зависи от оценката, при която оценяваме терма. Това е така, защото структурата не дава стойност на променливите, които се срещат в терма. Също така да си припомним, че за да оценим един терм е достатъчно да знаем каква е стойността на променливите, които се срещат в него (твърдение 3.2.6).

Подобно се оказва и положението при формулите с тази разлика, че тук съществена е само стойността на свободните променливи. Да разгледаме например формулата

$$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}) \tag{13}$$

За да може да се дефинира верността на тази формула е необходимо:

- структурата да каже кое е множеството, в което „работи“ кванторът $\forall \mathtt{x}$, т.е. универсумът;
- структурата да каже как се интерпретира предикатният символ $\mathtt{p}$;
- оценката да каже каква е стойността на променливите $\mathtt{y}$, $\mathtt{z}$ и $\mathtt{t}$. Стойността на останалите променливи, включително на $\mathtt{x}$, е без значение.

Нека например универсумът на структурата $\mathbf{M}$ е множеството на естествените числа и $\mathtt{p}^{\mathbf{M}}$ е предикатът

$$\mathtt{p}^{\mathbf{M}}(n, m, k, l) \longleftrightarrow m^{n+3} + m^{n+3} \neq l^{n+3}$$

В такъв случай формула (13) е вярна при оценка v тогава и само тогава, когато е вярно съждението:

За всяко естествено число n е вярно $a^{n+3} + b^{n+3} \neq c^{n+3}$

където $a = v(\mathtt{y})$, $b = v(\mathtt{z})$ и $c = v(\mathtt{t})$.

В общия случай, ако универсумът на структурата $\mathbf{M}$ е $|\mathbf{M}|$, тогава горната формула е вярна в $\mathbf{M}$ при оценка v тогава и само тогава, когато е вярно съждението:

За всеки елемент μ на $|\mathbf{M}|$ е вярно $\mathtt{p}^{\mathbf{M}}(\mu, v(\mathtt{y}), v(\mathtt{z}), v(\mathtt{t}))$.

Сега нека разгледаме по-общата ситуация на формула, в която след квантора стои не атомарна формула, а произволна формула

$$\forall \mathtt{x}\, \varphi$$

Кога трябва да считаме, че тази формула е вярна в структурата $\mathbf{M}$ при оценка v? Като пръв и най-естествен опит може да пробваме със следното съждение:

При всяка възможна стойност на $\mathtt{x}$ от универсума на $\mathbf{M}$ формулата φ е вярна.

Един проблем в тази формулировка е това, че в нея нищо не се казва за останлите променливи във φ и за оценката, която дава стойност на тези променливи. На втори опит получаваме следното:

При всяка възможна стойност на $\mathtt{x}$ от универсума на $\mathbf{M}$ формулата φ е вярна при оценка v.

В тази формулировка обаче възниква друг проблем. Какво означава „вярна при оценка v при еди-каква си стойност на променливата $\mathtt{x}$“? Та нали оценката v дава стойност на всички променливи, включително и на $\mathtt{x}$? На трети опит получаваме следното съждение:

За всеки елемент μ на универсума на $\mathbf{M}$ формулата φ е вярна при модифицираната оценка v', която се дефинира по следния начин:

$$v'(\xi) = \begin{cases} \mu, & \text{ако } \xi = \mathtt{x} \\ v(\xi), & \text{иначе} \end{cases}$$

Именно на този вариант ще се спрем, когато дефинираме стойността на формула в структура при оценка. Преди това обаче нека дефинираме по-формално понятието „модифицирана оценка“.

3.3.1. Дефиниция. Нека v е оценка в структурата $\mathbf{M}$, $\mathtt{x}$ е променлива и μ е елемент на универсума на $\mathbf{M}$. Тогава оценката

$$v'(\xi) = \begin{cases} \mu, & \text{ако } \xi = \mathtt{x} \\ v(\xi), & \text{иначе} \end{cases}$$

се нарича *модифицирана оценка* и ще бъде означавана така:

$$v|\mathtt{x} := \mu$$

С други думи, модифицираната оценка $v|\mathtt{x} := \mu$ е функция, която при аргумент $\mathtt{x}$ връща стойност μ, а за останалите аргументи съвпада с оценката v.

Използвайки модифицирани оценки, можем да дадем дефиницията за верността на формула по следния начин:

3.3.2. Дефиниция. Нека $\mathbf{M}$ е структура. За всяка формула φ от сигнатурата на $\mathbf{M}$ и оценка v в $\mathbf{M}$ ще дефинираме съждение, което ще записваме

$$\mathbf{M} \vDash \varphi[v]$$

и ще четем така: „Формулата φ е вярна в структурата $\mathbf{M}$ при оценка v.“

Съждението $\mathbf{M} \vDash \varphi[v]$ се дефинира рекурсивно по формулата φ както следва:

а) ако $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_2)$ е атомарна формула, то $\mathbf{M} \vDash \varphi[v]$ е съждението*

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_2]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v))$$

б) $\mathbf{M} \vDash \psi_1 \,\&\, \psi_2[v]$ е съждението

$$\mathbf{M} \vDash \psi_1[v] \text{ } \textit{и} \text{ } \mathbf{M} \vDash \psi_2[v]$$

в) $\mathbf{M} \vDash \psi_1 \vee \psi_2[v]$ е съждението

$$\mathbf{M} \vDash \psi_1[v] \text{ } \textit{или} \text{ } \mathbf{M} \vDash \psi_2[v]$$

г) $\mathbf{M} \vDash \psi_1 \Rightarrow \psi_2[v]$ е съждението

*Ако $\mathtt{p}^{\mathbf{M}}$ е релация, а не предикат, тогава $\mathbf{M} \vDash \varphi[v]$ е съждението

$$\mathtt{p}^{\mathbf{M}} \ni \langle [\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_2]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v) \rangle$$

ако $\mathbf{M} \vDash \psi_1[v]$, *то* $\mathbf{M} \vDash \psi_2[v]$

д) $\mathbf{M} \vDash \neg\psi[v]$ е съждението

не е вярно, че $\mathbf{M} \vDash \psi[v]$

е) $\mathbf{M} \vDash \forall \mathtt{x}\, \psi[v]$ е съждението

за всяко $\mu \in |\mathbf{M}|$ *е вярно* $\mathbf{M} \vDash \psi[v|\mathtt{x} \coloneqq \mu]$

ж) $\mathbf{M} \vDash \exists \mathtt{x}\, \psi[v]$ е съждението

съществува такова $\mu \in |\mathbf{M}|$, *че* $\mathbf{M} \vDash \psi[v|\mathtt{x} \coloneqq \mu]$

3.3.3. Дефиниция. Когато съждението $\mathbf{M} \vDash \varphi[v]$ не е вярно, казваме, че формулата φ не е вярна в структурата $\mathbf{M}$ при оценка v.

3.3.4. Пример. Нека сигнатурата $\mathfrak{sig}$ е такава, че в нея има единствен символ за константа $\mathtt{c}$, единствен функционален символ $\mathtt{f}$, който е двуместен, и единствен предикатен символ $\mathtt{p}$, който също е двуместен. Нека структурата $\mathbf{M}$ за $\mathfrak{sig}$ е с универсум множеството на реалните числа и

$$\mathtt{c}^{\mathbf{M}} = 3$$
$$\mathtt{f}^{\mathbf{M}}(x, y) = x + y$$
$$\mathtt{p}^{\mathbf{M}}(x, y) \longleftrightarrow x < y$$

Нека освен това оценката v е такава, че $v(\mathtt{x}) = 35$ и $v(\mathtt{y}) = 13$. Тогава атомарната формула $\mathtt{p}(\mathtt{f}(\mathtt{x}, \mathtt{c}), \mathtt{y})$ не е вярна в структурата $\mathbf{M}$ при оценка v, защото

$$\begin{aligned}\mathbf{M} \vDash \mathtt{p}(\mathtt{f}(\mathtt{x}, \mathtt{c}), \mathtt{y})[v] &\longleftrightarrow \mathtt{p}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(v(\mathtt{x}), \mathtt{c}^{\mathbf{M}}), v(\mathtt{y})) \\ &\longleftrightarrow \mathtt{p}^{\mathbf{M}}(\mathtt{f}^{\mathbf{M}}(35, 3), 13) \\ &\longleftrightarrow \mathtt{p}^{\mathbf{M}}(35 + 3, 13) \\ &\longleftrightarrow 35 + 3 < 13 \longleftrightarrow \text{лъжа}\end{aligned}$$

3.3.5. Дефиниция. Две формули φ и ψ са *еквивалентни*, ако при произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$, $\mathbf{M} \vDash \varphi[v]$ е еквивалентно на $\mathbf{M} \vDash \psi[v]$. Записваме това така:

$$\vDash \varphi \Leftrightarrow \psi$$

3.3.6. Следствие. *а)* $\vDash \varphi \Leftrightarrow \varphi$

б) ако $\vDash \varphi \Leftrightarrow \psi$, *то* $\vDash \psi \Leftrightarrow \varphi$

в) ако $\vDash \varphi \Leftrightarrow \psi$ *и* $\vDash \psi \Leftrightarrow \chi$, *то* $\vDash \varphi \Leftrightarrow \chi$

Доказателство. Следва непосредствено от дефиниция 3.3.5. ∎

Еквивалентните формули не са задължително равни, нито дори конгруентни. Може да си мислим за тях като изрази, които винаги имат една и съща стойност. Също както изразите

$$(x+y)^2 \text{ и } x^2+2xy+y^2$$

са различни, но винаги са равни, така и еквивалентните формули дори да са различни като запис, винаги имат еднаква вярност. Да бъдат две формули еквивалентни означава, че смисълът им е един и същ във всяка структура и при всяка оценка. Ако две формули са еквивалентни, то без значение каква е структурата и каква е оценката, винаги ако едната от тези формули се окаже вярна, то и другата ще бъде вярна, и ако едната от тях е невярна, то и другата ще бъде невярна.

3.3.7. Пример. Формулите

$$\mathtt{p(x)} \vee \mathtt{q(f(y))} \text{ и } \mathtt{q(f(y))} \vee \mathtt{p(x)}$$

са еквивалентни — без значение как структурата интерпретира символите `p` и `f` и как оценката оценява променливите `x` и `y`, винаги ако едната от тези формули се окаже вярна, то и другата ще бъде вярна, и ако едната от тях е невярна, то и другата ще бъде невярна.

Когато в някой аритметичен израз заменим някой подизраз с друг подизраз, който е равен, тогава и целият израз ще бъде равен. Например изразът

$$\oint_C (dx + (a+b)^2 dy)$$

е равен на израза

$$\oint_C (dx + (a^2+2ab+b^2) dy)$$

като за да стигнем до този извод дори не е нужно да знаем какво значи криволинеен интеграл.* Нещо подобно е вярно и за еквивалентните формули и трябва да си го докажем.

3.3.8. Твърдение. *Нека формулата φ има подформула ψ и формулата ψ е еквивалентна на ψ'. Ако формулата φ' се получава от φ като заменим подформулата ψ с ψ', то формулите φ и φ' са еквивалентни.*

*Всъщност и двата интеграла са равни на нула.

Доказателство. С индукция по построението на формулата φ.

Ако $\psi = \varphi$, то като заменим ψ с еквивалентна на нея формула ψ', ще получим $\varphi' = \psi'$ и значи φ и φ' ще са еквивалентни. Остава да разгледаме случая когато $\psi \neq \varphi$.

Ако φ е атомарна, то единствената ѝ подформула е $\psi = \varphi$, а вече разгледахме този случай.

Ако $\varphi = \chi_1 \,\&\, \chi_2$ и $\psi \neq \varphi$, то ψ ще е подформула на χ_1 или χ_2. Нека за определеност ψ е подформула на χ_1 и като заменим ψ на ψ' от χ_1 получаваме χ_1'. Съгласно индукционното предположение χ_1 и χ_1' са еквивалентни. Тъй като $\varphi' = \chi_1' \,\&\, \chi_2$, от тук получаваме, че φ и φ' също са еквивалентни.*

Случаите когато $\varphi = \chi_1 \vee \chi_2$ и $\varphi = \chi_1 \Rightarrow \chi_2$ се разглеждат аналогично.

Ако $\varphi = \neg\chi$ и $\psi \neq \varphi$, то ψ ще е подформула на χ. Като заменим ψ на ψ' от χ получаваме χ'. Съгласно индукционното предположение χ и χ' са еквивалентни. Тъй като $\varphi' = \neg\chi'$, от тук получаваме, че φ и φ' също са еквивалентни.

Ако $\varphi = \forall \mathtt{x}\, \chi$ и $\psi \neq \varphi$, то ψ е подформула на χ. Като заменим ψ с ψ' от χ получаваме χ'. Съгласно индукционното предположение χ и χ' са еквивалентни. Тъй като $\varphi' = \forall \mathtt{x}\, \chi'$, то от тук получаваме, че φ и φ' също са еквивалентни.**

Случаят когато $\varphi = \exists \mathtt{x}\, \chi$ се разглежда аналогично. ∎

*По-подробно това се обосновава така. Щом при произволна структура $\mathbf{M}$ и оценка v, $\mathbf{M} \vDash \chi_1[v]$ е еквивалентно на $\mathbf{M} \vDash \chi_1'[v]$, значи

$$\begin{aligned}
\mathbf{M} \vDash \varphi[v] &\longleftrightarrow \mathbf{M} \vDash (\chi_1 \,\&\, \chi_2)[v] \\
&\longleftrightarrow \mathbf{M} \vDash \chi_1[v] \text{ и } \mathbf{M} \vDash \chi_2[v] \\
&\longleftrightarrow \mathbf{M} \vDash \chi_1'[v] \text{ и } \mathbf{M} \vDash \chi_2[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\chi_1' \,\&\, \chi_2)[v] \\
&\longleftrightarrow \mathbf{M} \vDash \varphi'[v]
\end{aligned}$$

**По-подробно това се обосновава така. Щом при произволна структура $\mathbf{M}$ и оценка v, $\mathbf{M} \vDash \chi[v]$ е еквивалентно на $\mathbf{M} \vDash \chi'[v]$, значи

$$\begin{aligned}
\mathbf{M} \vDash \varphi[v] &\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{x}\, \chi[v] \\
&\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно, че } \mathbf{M} \vDash \chi[v|\mathtt{x} := \mu] \\
&\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно, че } \mathbf{M} \vDash \chi'[v|\mathtt{x} := \mu] \\
&\longleftrightarrow \mathbf{M} \vDash \forall \mathtt{x}\, \chi'[v] \\
&\longleftrightarrow \mathbf{M} \vDash \varphi'[v]
\end{aligned}$$

Също както за термовете определихме как да ги оценяваме при частична оценка (вж. дефиниция 3.2.8), полезно е същото да можем да правим и при формулите. По този начин ще може да използваме напр. записа

$$\mathbf{M} \vDash \varphi[5, 4]$$

когато искаме да кажем, че формулата φ е вярна в $\mathbf{M}$ когато стойността на някои, предварително уточнени променливи, е 5 и 6. Дефиницията може да бъде аналогична:

3.3.9. Дефиниция. Нека частичната оценка v в структура $\mathbf{M}$ е дефинирана за всички свободни променливи във формулата φ. Ще казваме, че φ е вярна в структурата $\mathbf{M}$ при частична оценка v, ако φ е вярна в $\mathbf{M}$ при коя да е оценка v', която съвпада с v за свободните променливи на φ. Ще означаваме това така: $\mathbf{M} \vDash \varphi[v]$.

За да бъде коректна току-що дадената дефиниция, е нужно да покажем, че верността на φ не зависи от конкретния избор на оценката v'. Това следва от следното твърдение:

3.3.10. Твърдение. *Нека v и v' са оценки в структура $\mathbf{M}$. Ако v и v' съвпадат за всички свободни променливи на формулата φ, то $\mathbf{M} \vDash \varphi[v]$ е еквивалентно на $\mathbf{M} \vDash \varphi[v']$.*

Доказателство. С индукция по формулата φ. Когато φ е атомарна и има вида $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_n]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v))$$

Съгласно твърдение 3.2.6 последното е еквивалентно на

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v'), [\![\tau_n]\!]^{\mathbf{M}}(v'), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v')) \longleftrightarrow \mathbf{M} \vDash \varphi[v']$$

Ако $\varphi = \psi_1 \,\&\, \psi_2$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \psi_1[v] \text{ и } \mathbf{M} \vDash \psi_2[v]$$

Съгласно индукционното предположение последното е еквивалентно на

$$\mathbf{M} \vDash \psi_1[v'] \text{ и } \mathbf{M} \vDash \psi_2[v'] \longleftrightarrow \mathbf{M} \vDash \varphi[v']$$

Когато φ има вида $\psi_1 \vee \psi_2$, $\psi_1 \Rightarrow \psi_2$ или $\neg\psi$, може да разсъждаваме аналогично.

Ако $\varphi = \forall \mathtt{x}\, \psi$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[v|\mathtt{x} \coloneqq \mu]$$

и

$$\mathbf{M} \vDash \varphi[v'] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[v'|\mathtt{x} \coloneqq \mu]$$

Съгласно индукционното предположение изразите отдясно на тези две еквивалентности са еквивалентни. (Защо може да прилагаме индукционното предположение?)

Когато $\varphi = \exists \mathtt{x}\, \psi$, се разсъждава аналогично. ∎

3.3.11. Означение. а) Когато φ е формула, ще използваме записа

$$\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$$

когато искаме да кажем, че променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са различни по между си и всички свободни променливи на φ, са измежду $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

б) Нека $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е формула и $\mu_1, \mu_2, \ldots, \mu_n$ са произволни елементи на универсума на структурата $\mathbf{M}$. Ще пишем

$$\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n]$$

вместо

$$\mathbf{M} \vDash \varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n]$$

***Задача* 39:** Нека структурата $\mathbf{M}$ е с универсум множеството на реалните числа, двуместният функционален символ $\mathtt{f}$ се интерпретира като функцията събиране (т.е. $\mathtt{f}^{\mathbf{M}}(a, b) = a + b$ за произволни $a, b \in \mathbb{R}$), двуместният функционален символ $\mathtt{g}$ се интерпретира като умножение (т.е. $\mathtt{f}^{\mathbf{M}}(a, b) = ab$ за произволни $a, b \in \mathbb{R}$) и двуместният предикатен символ $\mathtt{p}$ се интерпретира като равенство (т.е. $\mathtt{p}^{\mathbf{M}}(a, b) \longleftrightarrow a = b$ за произволни $a, b \in \mathbb{R}$). Намерете такива терм $\tau(\mathtt{x}_1, \mathtt{x}_2)$ и атомарна формула $\varphi(\mathtt{x}_1, \mathtt{x}_2)$, че

$$[\![\tau]\!]^{\mathbf{M}}(a, b) = a^2 + b$$
$$\mathbf{M} \vDash \varphi[a, b] \longleftrightarrow a^2 = 2b$$

за произволни реални числа a и b.

Тъй като искаме да отъждествяваме конгруентните формули, трябва да докажем, че стойностите на кои да е две конгруентни формули са еквивалентни.

3.3.12. Твърдение. *Ако две формули са конгруентни, то те са еквивалентни.*

Доказателство. Ако две формули са конгруентни, то едната може да се получи от другата като приложим краен брой еднократни преименувания на свързани променливи. Съгласно следствие 3.3.6 в) е достатъчно да видим, че при прилагането на едно еднократно преименуване на свързана променлива получаваме еквивалентна формула. Тъй като по дефиниция всяко еднократно преименуване представлява замяна на една подформула с друга, то съгласно твърдение 3.3.8 е достатъчно да докажем, че тези подформули са еквивалентни. Това следва от следващата лема. ∎

3.3.13. Лема. *Нека променливата* $\mathtt{z}'$ *е различна от* $\mathtt{z}$ *и не се съдържа във формулата* φ. *Нека* φ' *се получава от* φ *като заменим всяко срещане на* $\mathtt{z}$ *с* $\mathtt{z}'$. *Тогава*

а) $\models \forall \mathtt{z}\, \varphi \Leftrightarrow \forall \mathtt{z}'\varphi'$

б) $\models \exists \mathtt{z}\, \varphi \Leftrightarrow \exists \mathtt{z}'\varphi'$

Доказателство. **(а)** Нека $\mathbf{M}$ е произволна структура (за сигнатурата на формулите) и v е оценка в $\mathbf{M}$. Тогава $\mathbf{M} \models \forall \mathtt{z}\, \varphi[v]$ тогава и само тогава, когато за всяко $\mu \in |\mathbf{M}|$ е вярно $\mathbf{M} \models \varphi[v|\mathtt{z} := \mu]$. Аналогично, $\mathbf{M} \models \forall \mathtt{z}'\varphi'[v]$ тогава и само тогава, когато за всяко $\mu \in |\mathbf{M}|$ е вярно $\mathbf{M} \models \varphi'[v|\mathtt{z}' := \mu]$.

За да завършим доказателството, ще бъде достатъчно да докажем, че $\mathbf{M} \models \varphi[v|\mathtt{z} := \mu]$ е еквивалентно на $\mathbf{M} \models \varphi'[v|\mathtt{z}' := \mu]$. Това следва от точка б) на следващата лема, защото оценката $v|\mathtt{z} := \mu$ дава на променливата $\mathtt{z}$ същата стойност, каквато оценката $v|\mathtt{z}' := \mu$ дава на променливата $\mathtt{z}'$. Освен това за всички останали свободни променливи на φ тези две оценки дават една и съща стойност, защото $\mathtt{z}'$ не се среща в φ и значи ако $\mathtt{x}$ е свободна променлива, която е различна от $\mathtt{z}$, то $\mathtt{x} \neq \mathtt{z}'$.

(б) се доказва аналогично на (а). ∎

3.3.14. Лема. *Нека* $\mathtt{z}$ *и* $\mathtt{z}'$ *са различни променливи. Когато* ξ *е израз (терм или формула), който не съдържа променливата* $\mathtt{z}'$, *то в тази лема с* ξ' *ще означаваме израза, който се получава от* ξ *като заменим всяко срещане на променливата* $\mathtt{z}$ *с* $\mathtt{z}'$.

а) Нека оценките v *и* v' *в структура* $\mathbf{M}$ *са такива, че* $v(\mathtt{z}) = v'(\mathtt{z}')$ *и освен това за всяка променлива* $\mathtt{x}$, *която се среща в терма* τ *и е различна от* $\mathtt{z}$, *имаме* $v(\mathtt{x}) = v'(\mathtt{x})$. *Тогава* $[\![\tau]\!]^{\mathbf{M}}(v)$ *тогава и само тогава, когато* $[\![\tau']\!]^{\mathbf{M}}(v')$.

б) Нека оценките v *и* v' *в структура* $\mathbf{M}$ *са такива, че* $v(\mathtt{z}) = v'(\mathtt{z}')$ *и освен това за всяка свободна променлива* $\mathtt{x}$ *на формулата* φ,

която е различна от $\mathtt{z}$*, имаме* $v(\mathtt{x}) = v'(\mathtt{x})$*. Тогава* $\mathbf{M} \vDash \varphi[v]$ *тогава и само тогава, когато* $\mathbf{M} \vDash \varphi'[v']$*.*

Доказателство. **(а)** С индукция по терма τ. Ако $\tau = \mathtt{y}$ е променлива и $\mathtt{y} \neq \mathtt{z}$, то $\tau' = \tau$ и значи $[\![\tau]\!]^{\mathbf{M}}(v) = v(\mathtt{y}) = v'(\mathtt{y}) = [\![\tau']\!]^{\mathbf{M}}$.

Ако $\tau = \mathtt{z}$, то $\tau' = \mathtt{z}'$ и значи $[\![\tau]\!]^{\mathbf{M}}(v) = v(\mathtt{z}) = v'(\mathtt{z}') = [\![\tau']\!]^{\mathbf{M}}$.

Ако $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$, то

$$[\![\tau]\!]^{\mathbf{M}}(v) = \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_n]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v))$$

Съгласно индукционното предположение последното е равно на

$$\mathtt{f}^{\mathbf{M}}([\![\tau_1']\!]^{\mathbf{M}}(v'), [\![\tau_n']\!]^{\mathbf{M}}(v'), \ldots, [\![\tau_n']\!]^{\mathbf{M}}(v'))$$

което пък е равно на $[\![\tau']\!]^{\mathbf{M}}(v')$, защото

$$\tau' = (\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n))' = \mathtt{f}(\tau_1', \tau_2', \ldots, \tau_n')$$

(б) С индукция по формулата φ. Ако $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е атомарна, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(v), [\![\tau_n]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v))$$

Съгласно доказаното в точка (а) последното е еквивалентно на

$$\mathtt{p}^{\mathbf{M}}([\![\tau_1']\!]^{\mathbf{M}}(v'), [\![\tau_n']\!]^{\mathbf{M}}(v'), \ldots, [\![\tau_n']\!]^{\mathbf{M}}(v'))$$

което пък е еквивалентно на $\mathbf{M} \vDash \varphi'[v']$, защото

$$\varphi' = (\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n))' = \mathtt{p}(\tau_1', \tau_2', \ldots, \tau_n')$$

Ако $\varphi = \psi_1 \,\&\, \psi_2$, то

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \psi_1[v] \text{ и } \mathbf{M} \vDash \psi_2[v]$$

Съгласно индукционното предположение последното е еквивалентно на

$$\mathbf{M} \vDash \psi_1'[v'] \text{ и } \mathbf{M} \vDash \psi_2'[v'] \longleftrightarrow \mathbf{M} \vDash \varphi'[v']$$

Когато φ има вида $\psi_1 \vee \psi_2$, $\psi_1 \Rightarrow \psi_2$ или $\neg\psi$, може да разсъждаваме аналогично.

Ако $\varphi = \forall \mathtt{x}\, \psi$, където $\mathtt{x} \neq \mathtt{z}$, то $\varphi' = \forall \mathtt{x}\, \psi'$ и значи

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[v|\mathtt{x} \coloneqq \mu]$$

и

$$\mathbf{M} \vDash \varphi'[v'] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi'[v'|\mathtt{x} \coloneqq \mu]$$

Съгласно индукционното предположение изразите отдясно на тези две еквивалентности са еквивалентни.

Ако пък $\varphi = \forall \mathtt{z}\, \psi$, то $\varphi' = \forall \mathtt{z}\,' \psi'$ и значи

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[v|\mathtt{z} \coloneqq \mu]$$

и

$$\mathbf{M} \vDash \varphi'[v'] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi'[v'|\mathtt{z}' \coloneqq \mu]$$

и отново съгласно индукционното предположение изразите отдясно на тези две еквивалентности са еквивалентни.

Когато $\varphi = \exists \mathtt{x}\, \psi$, се разсъждава аналогично. ∎

Ако s е произволна субституция, а v произволна оценка в някоя структура $\mathbf{M}$, то може да дефинираме оценка w по следния начин:

$$w(\mathtt{z}) = [\![\mathtt{z}[s]]\!]^{\mathbf{M}}(v)$$

С други думи стойността на коя да е променлива $\mathtt{z}$ при оценка w е равна на стойността на терма $s(\mathtt{z})$ при оценка v. Следващото твърдение показва, че ако дефинираме оценката w по този начин, тогава ще имаме такава връзка не само за променливите, но също и за произволни термове. След това, в твърдение 3.3.18, ще видим че същото е вярно и за формулите.

3.3.15. Твърдение. *Нека s е произволна субституция, а v произволна оценка в някоя структура $\mathbf{M}$. Да дефинираме оценката w по следния начин:*

$$w(\mathtt{z}) = [\![\mathtt{z}[s]]\!]^{\mathbf{M}}(v)$$

Тогава за всеки терм τ

$$[\![\tau]\!]^{\mathbf{M}}(w) = [\![\tau[s]]\!]^{\mathbf{M}}(v)$$

Доказателство. С индукция по терма τ. Ако термът $\tau = \mathtt{x}$ е променлива, то получаваме искането от дефиницията на w:

$$[\![\tau]\!]^{\mathbf{M}}(w) = w(\mathtt{x}) = [\![\mathtt{x}[s]]\!]^{\mathbf{M}}(v) = [\![\tau[s]]\!]^{\mathbf{M}}(v)$$

Ако термът $\tau = \mathtt{c}$ е символ за константа, то получаваме искането съвсем лесно:

$$[\![\tau]\!]^{\mathbf{M}}(w) = \mathtt{c}^{\mathbf{M}} = [\![\mathtt{c}]\!]^{\mathbf{M}}(v) = [\![\mathtt{c}[s]]\!]^{\mathbf{M}}(v) = [\![\tau[s]]\!]^{\mathbf{M}}(v)$$

Ако $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$, то получаваме исканото, използвайки индукционното предположение за $\tau_1, \tau_2, \ldots, \tau_n$:

$$\begin{aligned}
[\![\tau]\!]^{\mathbf{M}}(w) &= [\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)]\!]^{\mathbf{M}}(w) \\
&= \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(w), [\![\tau_2]\!]^{\mathbf{M}}(w), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(w)) && \text{(деф. 3.2.2)} \\
&= \mathtt{f}^{\mathbf{M}}([\![\tau_1[s]]\!]^{\mathbf{M}}(v), [\![\tau_2[s]]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n[s]]\!]^{\mathbf{M}}(v)) && \text{(инд. предп.)} \\
&= [\![\mathtt{f}(\tau_1[s], \tau_2[s], \ldots, \tau_n[s])]\!]^{\mathbf{M}}(v) && \text{(деф. 3.2.2)} \\
&= [\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)[s]]\!]^{\mathbf{M}}(v) && \text{(деф. 2.8.2)} \\
&= [\![\tau[s]]\!]^{\mathbf{M}}(v)
\end{aligned}$$

■

Преди да докажем аналогично твърдение за произволни формули, ще докажем два частни случаи във вид на леми. Най-напред случая, когато формулата е атомарна формула.

3.3.16. Лема. *Нека s е произволна субституция, а v произволна оценка в някоя структура $\mathbf{M}$. Да дефинираме оценката w по следния начин:*

$$w(\mathtt{z}) = [\![\mathtt{z}[s]]\!]^{\mathbf{M}}(v)$$

Тогава за всяка атомарна формула φ

$$\mathbf{M} \vDash \varphi[w] \longleftrightarrow \mathbf{M} \vDash \varphi[s][v]$$

Доказателство. За произволна атомарна формула $\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ получаваме исканото, използвайки вече доказаното в твърдение 3.3.15:

$$\begin{aligned}
&\mathbf{M} \vDash \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)[w] \\
&\quad\longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(w), [\![\tau_2]\!]^{\mathbf{M}}(w), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(w)) && \text{(деф. 3.3.2 а))} \\
&\quad\longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1[s]]\!]^{\mathbf{M}}(v), [\![\tau_2[s]]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n[s]]\!]^{\mathbf{M}}(v)) && \text{(тв. 3.3.15)} \\
&\quad\longleftrightarrow \mathbf{M} \vDash \mathtt{p}(\tau_1[s], \tau_2[s], \ldots, \tau_n[s])[v] && \text{(деф. 3.3.2 а))} \\
&\quad\longleftrightarrow \mathbf{M} \vDash \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)[s][v] && \text{(деф. 2.8.2)} \\
&\quad\longleftrightarrow \mathbf{M} \vDash \tau[s][v]
\end{aligned}$$

■

В следващата лема доказваме друг частен случай. Този път формулите са произволни, но не съдържат квантори с опасни променливи, нито пък два квантора с една и съща променлива.

3.3.17. Лема. *За всяка формула φ, която не съдържа квантори с една и съща променлива, е вярно следното твърдение:*

Ако субституцията s е такава, че φ не съдържа квантори с опасни при субституция s променливи, а оценките v и w в структура $\mathbf{M}$ са такива, че за всяка свободна променлива $\mathtt{z}$ във формулата φ е изпълнено $w(\mathtt{z}) = [\![\mathtt{z}[s]]\!]^{\mathbf{M}}(v)$, то $\mathbf{M} \vDash \varphi[w]$ е еквивалентно на $\mathbf{M} \vDash \varphi[s][v]$.

Доказателство. Ще докажем това твърдение с индукция по броя на символите във формулата φ.

Когато $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$ е атомарна, нека w' е субституцията, за която $w'(\mathtt{z}) = [\![\mathtt{z}[s]]\!]^{\mathbf{M}}(v)$ за всяка променлива $\mathtt{x}$. Тъй като оценките w и w' съвпадат за всички променливи, срещащи се в термовете $\tau_1, \tau_2, \ldots, \tau_n$, от твърдение 3.2.6 получаваме $[\![\tau_i]\!]^{\mathbf{M}}(w) = [\![\tau_i]\!]^{\mathbf{M}}(w')$ и значи $\mathbf{M} \vDash \varphi[w]$ е еквивалентно на $\mathbf{M} \vDash \varphi[w']$. От лема 3.3.16 пък получаваме, че последното е еквивалентно на $\mathbf{M} \vDash (\varphi[s])[v]$.

Когато $\varphi = \psi_1 \,\&\, \psi_2$, то съгласно индукционното предположение* $\mathbf{M} \vDash \psi_1[w]$ е еквивалентно на $\mathbf{M} \vDash (\psi_1[s])[v]$ и $\mathbf{M} \vDash \psi_2[w]$ е еквивалентно на $\mathbf{M} \vDash (\psi_2[s])[v]$. Следователно

$$\begin{aligned}
\mathbf{M} \vDash \varphi[w] &\longleftrightarrow \mathbf{M} \vDash (\psi_1 \,\&\, \psi_2)[w] \\
&\longleftrightarrow \mathbf{M} \vDash \psi_1[w] \text{ и } \mathbf{M} \vDash \psi_2[w] \\
&\longleftrightarrow \mathbf{M} \vDash (\psi_1[s])[v] \text{ и } \mathbf{M} \vDash (\psi_2[s])[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\psi_1[s] \,\&\, \psi_2[s])[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\psi_1 \,\&\, \psi_2[s])[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\varphi[s])[v]
\end{aligned}$$

Когато $\varphi = \psi_1 \vee \psi_2$ или $\varphi = \psi_1 \Rightarrow \psi_2$, се разсъждава аналогично.

Когато $\varphi = \neg\psi$, то съгласно индукционното предположение $\mathbf{M} \vDash \psi[w]$ е еквивалентно на $\mathbf{M} \vDash (\psi[s])[v]$. Следователно

$$\begin{aligned}
\mathbf{M} \vDash \varphi[w] &\longleftrightarrow \mathbf{M} \vDash (\neg\psi)[w] \\
&\longleftrightarrow \text{не е вярно, че } \mathbf{M} \vDash \psi[w] \\
&\longleftrightarrow \text{не е вярно, че } \mathbf{M} \vDash (\psi[s])[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\neg\psi[s])[v] \\
&\longleftrightarrow \mathbf{M} \vDash ((\neg\psi)[s])[v] \\
&\longleftrightarrow \mathbf{M} \vDash (\varphi[s])[v]
\end{aligned}$$

Когато $\varphi = \forall\mathtt{x}\,\psi$, нека s' е субституцията

$$s'(\mathtt{z}) = \begin{cases} \mathtt{x}, & \text{ако } \mathtt{z} = \mathtt{x} \\ s(\mathtt{z}), & \text{иначе} \end{cases}$$

*Защо можем да прилагаме индукционното предположение? Защо ψ_1 и ψ_2 не съдържат квантори с променливи, които са опасни при субституция s?

Понеже трябва да докажем, че $\mathbf{M} \models \forall \mathtt{x}\, \psi[w]$ е еквивалентно на $\mathbf{M} \models (\forall \mathtt{x}\, \psi)[s][v]$, да разпишем тези два израза, за да видим какво ще се получи. За първия израз имаме:

$$\mathbf{M} \models \forall \mathtt{x}\, \psi[w] \longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно, че } \mathbf{M} \models \psi[w|\mathtt{x} := \mu] \quad (14)$$

Преди да разпишем втория израз, да съобразим, че опасните променливи в $\forall \mathtt{x}\, \psi$ при субституция s са същите като опасните променливи при субституция s' (тъй като $\mathtt{x}$ не е свободна променлива в тази формула, а двете субституции могат да се различават само за $\mathtt{x}$). И щом по условие във формулата няма квантори с опасни променливи при субституция s, то значи няма квантори и с опасни променливи при субституция s'. Това означава, че както $\forall \mathtt{x}\, \psi[s]$, така и $\forall \mathtt{x}\, \psi[s']$ се получават като просто заменим във формулата всяка свободна променлива $\mathtt{z}$ с $s(\mathtt{z})$ или съответно $s'(\mathtt{z})$. Но за всяка такава свободна променлива s и s' съвпадат, значи

$$(\forall \mathtt{x}\, \psi)[s] = (\forall \mathtt{x}\, \psi)[s'] \quad (15)$$

Да видим сега какви са опасните променливи във ψ при субституция s'. Единствената променлива, която би могла да бъде свободна в ψ без да е свободна в $\forall \mathtt{x}\, \psi$, е променливата $\mathtt{x}$. Следователно единствените променливи, които биха могли да бъдат опасни в ψ без да са били опасни и в $\forall \mathtt{x}\, \psi$, са променливите в терма $s'(\mathtt{x})$, т.е. само променливата $\mathtt{x}$. Но в ψ няма квантори с променливата $\mathtt{x}$, защото по условие в голямата формула няма квантори с една и съща променлива (а вече имаме един квантор $\forall \mathtt{x}$). Следователно в ψ няма квантори с опасни променливи при субституция s' и значи $\psi[s']$ може да се получи като просто заменим всяка свободна променлива $\mathtt{z}$ с $s'(\mathtt{z})$. Затова от (15) получаваме:

$$(\forall \mathtt{x}\, \psi)[s] = \forall \mathtt{x}\, (\psi[s'])$$

Вече сме готови да разпишем $\mathbf{M} \models (\forall \mathtt{x}\, \psi)[s][v]$:

$$\begin{aligned} \mathbf{M} \models (\forall \mathtt{x}\, \psi)[s][v] \\ &\longleftrightarrow \mathbf{M} \models \forall \mathtt{x}\, (\psi[s'])[v] \\ &\longleftrightarrow \text{за всяко } \mu \in |\mathbf{M}| \text{ е вярно, че } \mathbf{M} \models (\psi[s'])[v|\mathtt{x} := \mu] \end{aligned} \quad (16)$$

Като сравним (14) с (16), виждаме, че остава да докажем единствено това, че $\mathbf{M} \models \psi[w|\mathtt{x} := \mu]$ е еквивалентно на $\mathbf{M} \models (\psi[s'])[v|\mathtt{x} := \mu]$. Това можем да получим непосредствено от индукционното предположение за ψ. Да проверим обаче, че имаме право да приложим индукционното

предположение за s', $(w|\mathtt{x} \coloneqq \mu)$ и $(v|\mathtt{x} \coloneqq \mu)$. Вече видяхме, че ψ не съдържа квантори с опасни променливи при субституция s'. Освен това за всяка свободна променлива $\mathtt{y}$ на ψ трябва да докажем, че е вярно

$$(w|\mathtt{x} \coloneqq \mu)(\mathtt{y}) = [\![\mathtt{y}[s']]\!]^{\mathbf{M}}(v|\mathtt{x} \coloneqq \mu)$$

Наистина, ако $\mathtt{y} = \mathtt{x}$, то

$$\begin{aligned}(w|\mathtt{x} \coloneqq \mu)(\mathtt{y}) &= (w|\mathtt{x} \coloneqq \mu)(\mathtt{x}) = \mu \\ &= [\![\mathtt{x}]\!]^{\mathbf{M}}(v|\mathtt{x} \coloneqq \mu) = [\![s'(\mathtt{x})]\!]^{\mathbf{M}}(v|\mathtt{x} \coloneqq \mu) = [\![\mathtt{y}[s']]\!]^{\mathbf{M}}(v|\mathtt{x} \coloneqq \mu)\end{aligned}$$

Ако пък $\mathtt{y}$ е различна от $\mathtt{x}$, то $\mathtt{y}$ ще трябва да бъде свободна променлива и на φ и затова

$$(w|\mathtt{x} \coloneqq \mu)(\mathtt{y}) = w(\mathtt{y}) = [\![\mathtt{y}[s]]\!]^{\mathbf{M}}(v)$$

Тъй като φ не съдържа квантори с променливи, които са опасни при субституция s, то термът $s(\mathtt{y})$ не съдържа променливата $\mathtt{x}$ и затова можем да продължим горната редица от равенства по следния начин:

$$[\![\mathtt{y}[s]]\!]^{\mathbf{M}}(v) = [\![\mathtt{y}[s]]\!]^{\mathbf{M}}(v|\mathtt{x} \coloneqq \mu) = [\![\mathtt{y}[s']]\!]^{\mathbf{M}}(v|\mathtt{x} \coloneqq \mu)$$

Когато $\varphi = \exists \mathtt{x}\, \psi$, се разсъждава аналогично. ∎

Вече сме готови да докажем аналога на твърдение 3.3.15 за формули.

3.3.18. Твърдение. *За всяка субституция s и оценка v в структура $\mathbf{M}$ съществува такава оценка w в $\mathbf{M}$, че за всяка формула φ, $\mathbf{M} \vDash \varphi[w]$ е еквивалентно на $\mathbf{M} \vDash (\varphi[s])[v]$.*

Доказателство. Нека w е оценката

$$w(\mathtt{z}) = [\![\mathtt{z}[s]]\!]^{\mathbf{M}}(v)$$

Нека φ е произволна формула. Нека ψ е конгруентна с φ формула, която не съдържа квантори с опасни при субституция s променливи и в която всички квантори имат различни променливи (такава формула има съгласно лема 2.7.12). Съгласно лема 3.3.17, $\mathbf{M} \vDash \psi[w]$ е еквивалентно на $\mathbf{M} \vDash (\psi[s])[v]$. От тук получаваме исканото, защото формулата φ е конгруентна, а значи и еквивалентна с ψ, а формулата $\varphi[s]$ е конгруентна, а значи и еквивалентна с формулата $\psi[s]$. ∎

3.4. Тъждествена вярност, изпълнимост, следване

Обикновено когато пишем някоя клауза ние не си мислим, че променливите в нея имат някакви конкретни стойности. Например формулата

$$\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$$

ни казва, че двуместната функция, означена с „+“, е комутативна, т.е. $\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$ при произволни стойности на $\mathtt{x}$ и $\mathtt{y}$. Това показва, че понятието „вярност в структура при оценка“, което вече дефинирахме, не ни дава точно това, което искаме. Например горната формула ще бъде вярна в някоя структура при определена оценка тогава и само тогава, когато $\mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$ е вярно при някакви конкретни стойности на променливите $\mathtt{x}$ и $\mathtt{y}$, определени от оценката, а не при произволни стойности.*

Когато ни е дадена само структура, но не и оценка, една формула може да има различни „степени на вярност“. Например:

- формулата може да бъде вярна при всяка оценка;
- формулата може да бъде вярна при поне една оценка;
- формулата може да бъде невярна при произволна оценка.

Когато за една формула кажем просто, че е вярна, без да уточним при коя оценка, удобно е да считаме, че формулата е вярна при всяка оценка. Затова можем да дадем следната дефиниция:

3.4.1. Дефиниция. а) Формулата φ е *(тъждествено) вярна* в структура $\mathbf{M}$, ако е вярна в структурата $\mathbf{M}$ при произволна оценка v.** Записваме това така:

$$\mathbf{M} \vDash \psi$$

б) Формулата φ е *тъждествено вярна* или *(предикатна) тавтология*, ако φ е тъждествено вярна във всяка структура.*** Записваме това така:

$$\vDash \psi$$

*Нека структурата е с универсум положителните реални числа и функционалният символ „+“ е интерпретиран като операцията деление. Тогава горната клауза не е вярна, защото например $2/3 \neq 3/2$. Ако обаче оценката е такава, че $v(\mathtt{x}) = v(\mathtt{y}) = 42$, тогава горната клауза ще бъде вярна при тази оценка, защото $42/42 = 42/42$.

**Други термини, които означават същото като „тъждествено вярна“ са *общовалидна* и *валидна*. На английски universally valid и valid.

***За предикатните тавтологии се използват още термините *общовалидна формула* и *валидна формула*.

Забележка: Без опасност от недоразумения може да изпускаме думата „тъждествено“ от термина „тъждествено вярна формула в структура“. Причината за това е следната: съгласно дефиниция 3.3.2 сме дефинирали какво значи „вярна формула в структура **при оценка**“. Следователно ако просто кажем, че някоя формула е вярна в структура без да уточним коя е оценката, няма как да имаме предвид дефиниция 3.3.2. Най-естествено е да считаме, че в такъв случай формулата е вярна при произволна оценка, т.е. че е тъждествено вярна. Да си припомним аксиомите на моноид и полупръстен (вж. стр. 86 и 88). Ако за тези аксиоми кажем, че са верни във всеки моноид или съотв. полупръстен, ние всъщност ще имаме предвид, че тези аксиоми са тъждествено верни, а не че са верни при някоя конкретна, неясно каква оценка.

Понякога в контекста на предикатната логика думата „тавтология“ се използва само за онези предикатни тавтологии, които могат да се установят от съждителната логика. Тук няма да използваме такъв тип тавтологии и затова може да си позволим да изпускаме думата „предикатна“ от словосъчетанието „предикатна тавтология“.

Една формула може да бъде тъждествено вярна в една структура без да е тъждествено вярна във всяка структура. Например формулата $\mathtt{x.y} = \mathtt{y.x}$ е тъждествено вярна във всяка структура, която е абелева група или комутативен пръстен, но не е тъждествено вярна структурите, които са неабелеви групи или пък некомутативни пръстени. Разбира се, тази формула е и пример на формула, която може да бъде тъждествено вярна в някоя конкретна структура (коя да е абелева група или комутативен пръстен) без да бъде тъждествено вярна по принцип (т.е. без да бъде предикатна тавтология).

Съгласно дефиниция 3.3.5, ако две формули φ и ψ са еквивалентни, записваме това така: $\models \varphi \Leftrightarrow \psi$. Съгласно дефиниция 3.4.1 б) пък, когато запишем $\models \varphi \Leftrightarrow \psi$, това означава, че формулата $\varphi \Leftrightarrow \psi$ е тавтология. Следното твърдение показва, че нямаме опасна колизия в означенията, защото тези две неща са еквивалентни.

3.4.2. Твърдение. *Две формули φ и ψ са еквивалентни тогава и само тогава, когато формулата $\varphi \Leftrightarrow \psi$ е предикатна тавтология.*

Доказателство. Да си припомним, че формулата $\varphi \Leftrightarrow \psi$ е съкращение на формулата $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \varphi)$. Затова от дефиниции 3.3.5 и 3.4.1 а) следва, че φ и ψ са еквивалентни тогава и само тогава, когато при произволна структура $\mathbf{M}$ и оценка v в $\mathbf{M}$, от $\mathbf{M} \models \varphi[v]$ следва $\mathbf{M} \models \psi[v]$ и от $\mathbf{M} \models \psi[v]$ следва $\mathbf{M} \models \varphi[v]$. Последното означава, че $\mathbf{M} \models \varphi[v]$ и $\mathbf{M} \models \psi[v]$ са еквивалентни. ∎

Когато някоя формула е вярна не при всяка оценка, а при някоя оценка, или не във всяка структура, а в някоя структура, използваме термина „изпълнима формула“.

3.4.3. Дефиниция. а) Формулата φ е *изпълнима* в структурата $\mathbf{M}$, ако съществува оценка, при която φ е вярна в $\mathbf{M}$. Записваме това така:

$$\mathbf{M} \models^{\exists} \varphi$$

б) Формулата φ е *изпълнима*, ако съществуват структура и оценка, при които φ е вярна. Записваме това така:

$$\models^{\exists} \varphi$$

Забележка: За съжаление използваната терминология на български е малко объркваща. Вместо „изпълнима“ на английски се използва прилагателното satisfiable. На това прилагателно съответства глаголът satisfy, който на български превеждаме с напълно различна дума — „удовлетворява“. Вместо „удовлетворима формула“ казваме „изпълнима формула“ и вместо (напр. в контекста на логическото програмиране) да казваме, че дадена цел се изпълнява, казваме, че целта се удовлетворява. На английски обаче във всеки един от тези случи се използва една и съща дума.

3.4.4. Дефиниция. а) Формула φ е *тъждествено невярна* в структурата $\mathbf{M}$, ако при всяка оценка v в $\mathbf{M}$, формулата φ е невярна в $\mathbf{M}$ при оценка v.

б) Формулата φ е *неизпълнима* в структурата $\mathbf{M}$, ако не съществува оценка, при която φ е вярна в $\mathbf{M}$.

в) Формула φ е *тъждествено невярна*, ако при всяка структура $\mathbf{M}$ и оценка v в $\mathbf{M}$, формулата φ е невярна в $\mathbf{M}$ при оценка v.

г) Формулата φ е *неизпълнима*, ако не съществуват структура и оценка, при които φ е вярна.

Следващите две задачи показват, че макар понятията от току-що дадената дефиниция да са естествени, те по същество не ни дават нищо ново.

***Задача* 40:** Нека φ е формула и $\mathbf{M}$ е структура. Докажете, че следните три неща са еквивалентни:

- φ не е изпълнима в $\mathbf{M}$;
- φ е неизпълнима в $\mathbf{M}$;

– φ е тъждествено невярна в $\mathbf{M}$.

***Задача* 41:** Нека φ е формула. Докажете, че следните три неща са еквивалентни:

– φ не е изпълнима;

– φ е неизпълнима;

– φ е тъждествено невярна.

***Задача* 42:** Възможно ли е една формула да не е тъждествено вярна в някоя структура, а пък да бъде изпълнима в същата структура?

***Задача* 43:** Възможно ли е една формула да бъде едновременно неизпълнима и тъждествено вярна в някоя структура? Защо при отговора на този въпрос се налага да използваме това, че универсумът на структурите е непразно множество?

3.4.5. Дефиниция. Ако формула φ е тъждествено вярна в структура $\mathbf{M}$ (т.е. $\mathbf{M} \vDash \varphi$), казваме, че $\mathbf{M}$ е модел за φ. Ако всеки елемент на множество Γ от формули е тъждествено верен в структура $\mathbf{M}$, казваме, че $\mathbf{M}$ е модел за Γ и пишем $\mathbf{M} \vDash \Gamma$.

3.4.6. *Забележка*: Понякога в текстове, които не са писани от логици, думата „модел“ се приема за синоним на „структура“. Това е неправилно. Когато кажем, че $\mathbf{M}$ е модел, това означава не само, че $\mathbf{M}$ е структура, но също и че определени (известни може би от контекста) неща са верни в $\mathbf{M}$.

***Задача* 44:** Докажете, че ако някое множество от формули има поне един модел, то то има безброй много модели.

3.4.7. Дефиниция. *Затворена формула* означава формула, която не съдържа свободни променливи.

3.4.8. Съгласно твърдение 3.3.10 верността на една затворена формула в структура $\mathbf{M}$ не зависи по никакъв начин от оценката, защото всеки две оценки съвпадат за свободните променливи на една затворена формула. Следователно има смисъл когато говорим за затворена формула да четем „$\mathbf{M} \vDash \varphi$“ като „$\varphi$ е вярна в $\mathbf{M}$“, изпускайки наречието „тъждествено“.

3.4.9. Твърдение. *Ако φ е затворена формула, а $\mathbf{M}$ — произволна структура, то за всяка оценка v в $\mathbf{M}$*

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \varphi \longleftrightarrow \mathbf{M} \overset{\exists}{\vDash} \varphi$$

Доказателство. Да припомним, че $\mathbf{M} \vDash \varphi$ означава, че формулата φ е тъждествено вярна в $\mathbf{M}$, т.е. φ е вярна при всяка оценка, а $\mathbf{M} \overset{\exists}{\vDash} \varphi$ означава, че формулата φ е изпълнима в $\mathbf{M}$, т.е. φ е вярна при някоя оценка.

Съгласно твърдение 3.3.10 за кои да е две оценки v_1 и v_2 в $\mathbf{M}$ съждението $\mathbf{M} \vDash \varphi[v_1]$ е еквивалентно на $\mathbf{M} \vDash \varphi[v_2]$. Следователно $\mathbf{M} \overset{\exists}{\vDash} \varphi$ е истина тогава и само тогава, когато $\mathbf{M} \vDash \varphi[v]$ е истина при някоя оценка v, тогава и само тогава, когато $\mathbf{M} \vDash \varphi[v]$ е истина за всяка оценка v, тогава и само тогава, когато $\mathbf{M} \vDash \varphi$. ∎

***Задача* 45:** Възможно ли е една затворена формула да бъде изпълнима, но да не бъде тъждествено вярна?

Полезно е едно понятие, което казва, че от някакви формули следва някоя формула. Нека например множеството Γ съдържа аксиомите за полупръстен. Може да считаме, че формулата φ следва от Γ, ако φ е вярна във всички полупръстени. Следната дефиниция формализира интуицията на тази забележка:

3.4.10. Дефиниция. Нека Γ е множество от формули. Казваме, че формулата φ *следва* от Γ, ако φ е тъждествено вярна във всяка структура, в която са тъждествено верни формулите в Γ. В този случай казваме още и накратко „от Γ следва φ“ и използваме следното означение:

$$\Gamma \vDash \varphi$$

3.4.11. *Забележка*: Нека не се оборкваме и обърнем внимание, че използваме знака $\vDash$ за няколко различни цели. Когато Γ е множество от формули, тогава $\Gamma \vDash \varphi$ означава „от формулите в Γ следва φ“. Когато $\mathbf{M}$ е структура, $\mathbf{M} \vDash \varphi$ означава „φ е (тъждествено) вярно в $\mathbf{M}$“. А когато $\mathbf{M}$ е структура и v е оценка, $\mathbf{M} \vDash \varphi[v]$ означава „φ е вярна в $\mathbf{M}$ при оценка v“.

***Задача* 46:** Нека Γ и Δ са множества от формули и φ е формула. Докажете, че:

1. $\varphi \in \Gamma \longrightarrow \Gamma \vDash \varphi$
2. $\Delta \subseteq \Gamma$ и $\Delta \vDash \varphi \longrightarrow \Gamma \vDash \varphi$
3. Ако всеки елемент на Δ следва от Γ и $\Delta \vDash \varphi$, то $\Gamma \vDash \varphi$.

***Забележка*:** Условия а) и в) от задача 46 аксиоматизират това, което в абстрактната алгебрична логика се нарича *релация на следване*.*

*На английски consequence relation.

В контекста на логическото програмиране е полезно и друго понятие за следване, при което за формулата φ не искаме да бъде тъждествено вярна, а само изпълнима. Нека например Γ съдържа клаузите от някоя програма на пролог. Тогава целта `?-p(X,X)` ще се удовлетвори не когато от програмата следва, че атомарната формула `p(X,X)` е вярна при произволна стойност на променливата `X`, а когато тя е вярна при поне една стойност на `X`.

3.4.12. Дефиниция. Нека Γ е множество от формули. Казваме, че формулата φ се *удовлетворява* от Γ, ако φ е изпълнима във всяка структура, в която са тъждествено верни формулите от Γ. Записваме това така:

$$\Gamma \models^{\exists} \varphi$$

3.5. Хомоморфизми

Когато наблюдаваното поведение на една физическа система не се променя, ако подложим системата на някаква трансформация, физиците казват, че е налице *симетрия*. Симетриите са фундаментално понятие в съвременната физика и не малко от т.н. природни закони могат да се изведат като следствие от наличието на една или друга симетрия. Ето някои примери:

– **Транслация.** Всяка проста транслация в пространството е пример за симетрия. Да си изберем някаква четиримерна координатна система и да опишем поведението на света, използвайки координати от така избраната координатна система. Сега да изберем нова координатна система, която се получава от първата с транслация в пространството. Дали ако опишем поведението на света в новата координатна система ще получим нов вид физически зависимости? Не, физическите зависимости не се променят при транслация на координатната система. От тук, впрочем, може да се изведе като следствие т.н. закон за запазване на импулса.[*]

– **Ротация.** Всяка проста ротация също е пример за симетрия. Да си изберем някаква четиримерна координатна система и да опишем поведението на света, използвайки координати от така избраната координатна система. Сега да изберем нова координатна

[*]На земята може да се придвижваме пеш или с автомобил без да се налага да изхвърляме вещество. В космоса това не е възможно и единственото средство за придвижване е ракетният двигател.

система, която има същото начало като първата, но е завъртяна. Дали ако опишем поведението на света в новата координатна система ще получим нов вид физически зависимости? Не, физическите зависимости не се променят при ротация на координатната система. От тук, впрочем, може да се изведе като следствие т.н. закон за запазване на въртящия момент.*

– **Отместване във времето.** Отместването във времето също е пример за симетрия. Да си изберем някаква четиримерна координатна система и да опишем поведението на света, използвайки координати от така избраната координатна система. Сега да изберем нова координатна система, която пространствено е ориентирана по същия начин като първата, но началото ѝ е отместено във времето. Дали ако опишем поведението на света в новата координатна система ще получим нов вид физически зависимости? Не, физическите зависимости не се променят при отместване във времето — каквито са били вчера, такива ще бъдат и утре. От тук, впрочем, може да се изведе като следствие т.н. закон за запазване на енергията.
– **Скоростта на светлината.** Скоростта на светлината винаги се оказва една и съща — без значение с каква скорост се движи фенера и с каква скорост се движи измервателя. Използвайки това свойство заедно с някои други симетрии може да изведем като следствие *Специалната теория на относителността*.
– **Гладки трансформации.** Ако приемем, че свойствата на физическите обекти не се променят, ако извършим гладко преобразуване на координатите, то от тук, използвайки и някои други симетрии, може да изведем като следствие *Общата теория на относителността*.

Това, което физиците наричат симетрия, математиците наричат *автоморфизъм*. Трансформациите, при които определени свойства на системата се запазват, са от изключителна важност не само във физиката, но и в математиката. Ето някои примери:

– В линейната алгебра *линейните трансформации* запазват линейните операции: $h(\vec{x}+\vec{y}) = h(\vec{x}) + h(\vec{y})$, $h(a\vec{x}) = ah(\vec{x})$. Когато едно такова изображение е биективно, то се нарича *изоморфизъм*.

*На земята може да се въртим във всички посоки, защото имаме опорна точка. В космоса телата не могат да ускорят или забавят въртенето си без да изхвърлят част от себе си. Впрочем това не е задължително да става с ракетен двигател.

– В абстрактната алгебра *хомоморфизмите* запазват операциите в алгебричната структура. Например за хомоморфизмите между полета е вярно, че $h(0) = 0$, $h(1) = 1$, $h(x + y) = h(x) + h(y)$ и $h(xy) = h(x)h(y)$. Когато едно такова изображение е биективно, то се нарича *изоморфизъм*.

– В топологията *непрекъснатите изображения* запазват свойството „допиране“ — ако две множества A и B се допират, то $h(A)$ и $h(B)$ също ще се допират. Когато едно такова изображение е биективно, то се нарича *хомеоморфизъм*.

– В геометрията *гладките функции* запазват гладкостта — ако множеството A е гладко многообразие, то $h(A)$ също ще бъде гладко многообразие (локално). Когато едно такова изображение е биективно, то се нарича *дифеоморфизъм*.

Впрочем също както във физиката е възможно да извеждаме свойствата на физическите системи като следствие от наличието на различни симетрии, така и в математиката е възможно да описваме математическите обекти, използвайки най-вече трансформации, запазващи свойствата им. Това е подходът, използван в *Теория на категориите*.*

Вече видяхме, че алгебричните структури се оказват частни случаи на нашето понятие за структура от дефиниция 3.1.7 (вж. напр. моноидите и полупръстените от раздел 2.6 и пример 3.1.9). Не може ли след като разгледаме как алгебриците са дефинирали понятието „хомоморфизъм“ за своите структури, от тук по аналогия да получим и дефиниция за хомоморфизъм за нашите структури? Нека видим как изглежда дефиницията на хомоморфизъм между различни видове алгебрични структури.

Езикът на групите съдържа константа e — единичния елемент — и „умножение“, означавано с точка. Ако $\mathbf{G}$ и $\mathbf{H}$ са групи, то $h\colon \mathbf{G} \to \mathbf{H}$ е хомоморфизъм, ако h е такова изображение от универсума на $\mathbf{G}$ в универсума на $\mathbf{H}$, че

$$h(e^{\mathbf{G}}) = e^{\mathbf{H}}$$

*Категориите са измислени в хомологичната алгебра и в тази област те са незаменими. Освен това те са получили приложения и в много други области. Например приложенията им в компютърните науки са многобройни и разнообразни и при това важни за практиката. Категориите позволяват да даваме дефиниции на абстрактни понятия, които трудно бихме могли да формулираме без използването на теоретикокатегорен език. За съжаление често тези абстракции се оказват самоцелни и ненужни и това понякога води до лоша слава на теория на категориите.

където с $e^{\mathbf{G}}$ и $e^{\mathbf{H}}$ сме означили единичните елементи съответно на $\mathbf{G}$ и $\mathbf{H}$, и освен това за произволни елементи a и b на универсума на $\mathbf{G}$

$$h(a \,.^{\mathbf{G}}\, b) = h(a) \,.^{\mathbf{H}}\, h(b)$$

където с $.^{\mathbf{G}}$ и $.^{\mathbf{H}}$ сме означили умноженията съответно в групите $\mathbf{G}$ и $\mathbf{H}$.

Да разгледаме сега понятието хомоморфизъм между пръстени (хомоморфизмите между полета се дефинират аналогично). Езикът на пръстените съдържа константи 0 и 1, адитивна операция „+“ и мултипликативна операция „.“. Ако $\mathbf{R}$ и $\mathbf{T}$ са пръстени, то $h\colon \mathbf{R} \to \mathbf{T}$ е хомоморфизъм, ако h е такова изображение от универсума на $\mathbf{R}$ в универсума на $\mathbf{T}$, че

$$h(0^{\mathbf{R}}) = 0^{\mathbf{T}}$$
$$h(1^{\mathbf{R}}) = 1^{\mathbf{T}}$$

където с $0^{\mathbf{R}}$, $0^{\mathbf{T}}$, $1^{\mathbf{R}}$ и $1^{\mathbf{T}}$ сме означили нулевия и единичния елемент на $\mathbf{R}$ и $\mathbf{T}$, и освен това за произволни елементи a и b на универсума на $\mathbf{R}$

$$h(a \;+^{\mathbf{R}}\; b) = h(a) \;+^{\mathbf{T}}\; h(b)$$
$$h(a \,.^{\mathbf{R}}\, b) = h(a) \,.^{\mathbf{T}}\, h(b)$$

където с $+^{\mathbf{R}}$, $+^{\mathbf{T}}$, $.^{\mathbf{R}}$ и $.^{\mathbf{T}}$ сме означили адитивната и мултипликативната операция в пръстените $\mathbf{R}$ и $\mathbf{T}$.

Очевидно е, че тези дефиниции си приличат. Ако решим да си направим труда да проверим как изглеждат дефинициите на други видове хомоморфизми между алгебрични структури, ще установим, че и техните дефиниции са подобни на току-що дадените. По-точно, ще забележим, че за всички константи от езика на съответната структура искаме равенства от вида

$$h(e^{\mathbf{G}}) = e^{\mathbf{H}}$$
$$h(0^{\mathbf{R}}) = 0^{\mathbf{T}}$$
$$h(1^{\mathbf{R}}) = 1^{\mathbf{T}}$$

Това означава, че навярно ще бъде разумно в нашата дефиниция за хомоморфизъм $h\colon \mathbf{M} \to \mathbf{K}$ да поискаме за всеки символ за константа c от сигнатурата да се изпълнява следното равенство:

$$h(\mathsf{c}^{\mathbf{M}}) = \mathsf{c}^{\mathbf{K}}$$

Освен това в различните алгебрични дефиниции за хомоморфизъм се забелязва, че за всички алгебрични операции се изискват равенства от вида

$$h(a \mathbin{.^{\mathbf{G}}} b) = h(a) \mathbin{.^{\mathbf{H}}} h(b)$$
$$h(a \mathbin{+^{\mathbf{R}}} b) = h(a) \mathbin{+^{\mathbf{T}}} h(b)$$
$$h(a \mathbin{.^{\mathbf{R}}} b) = h(a) \mathbin{.^{\mathbf{T}}} h(b)$$

Това ни подсказва, че навярно ще бъде разумно в нашата дефиниция за хомоморфизъм $h\colon \mathbf{M} \to \mathbf{K}$ да поискаме за всеки n-местен функционален символ $\mathtt{f}$ от сигнатурата и произволни елементи $\mu_1, \mu_2, \ldots, \mu_n$ от универсума на $\mathbf{M}$ да се изпълнява следното равенство:

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = \mathtt{f}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

Остава да видим какво изискване да поискаме за предикатните символи. Алгебрата тук не може да ни помогне, защото в алгебричните структури единственият предикатен символ е равенството. За съжаление, оказва се, че има различни смислени варианти за това кои свойства на предикатните символи трябва да се запазват. В следващата дефиниция за хомоморфизъм сме поискали от хомоморфизма да запазва възможно най-малко свойства:

3.5.1. Дефиниция. Нека $\mathbf{M}$ и $\mathbf{K}$ са произволни структури.* Казваме, че h е *хомоморфизъм* от $\mathbf{M}$ в $\mathbf{K}$ и запизваме това така:

$$h\colon \mathbf{M} \to \mathbf{K}$$

ако h е изображение от универсума на $\mathbf{M}$ в универсума на $\mathbf{K}$ и освен това:

а) за всеки символ за константа $\mathtt{c}$

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$$

б) за всеки n-местен функционален символ $\mathtt{f}$ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = \mathtt{f}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

*Както обикновено, когато не уточняваме с какви сигнатури работим, се предполага, че всички структури, термове и формули използват една и съща сигнатура.

в) за всеки n-местен предикатен символ $\mathtt{p}$ и елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$\text{ако } \mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n), \text{ то } \mathtt{p}^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

3.5.2. Пример. Хомоморфизмите в алгебрата се оказват частни случаи на току-що даденото понятие. При символите за константи и функционалните символи сме се погрижили нашата дефиниция да съвпада с това, което се казва в алгебричните дефиниции. В нашата дефиниция обаче има допълнително изискване за предикатните символи, което го няма в алгебрата. Да видим какво казва това изискване.

Единственият предикатен символ в алгебричните структури е равенството. За равенството условието от дефиниция 3.5.1 в) казва следното: за произволни елементи μ_1 и μ_2 на универсума на $\mathbf{M}$

$$\text{ако } \mu_1 = \mu_2, \text{ то } h(\mu_1) = h(\mu_2)$$

Това обаче е очевидно — ясно е, че ако $\mu_1 = \mu_2$, то също и $h(\mu_1) = h(\mu_2)$. Следователно едно изображение е хомоморфизъм между моноиди, групи, попупръстени, пръстени, полета и т.н. тогава и само тогава, когато то е хомоморфизъм по смисъла на дефиниция 3.5.1.

В теория на графите хомоморфизмите се дефинират по следния начин: h е хомоморфизъм от граф $\mathbf{G}_1$ в граф $\mathbf{G}_2$, ако h изобразява всеки връх на $\mathbf{G}_1$ в някой връх на $\mathbf{G}_2$ и винаги когато в $\mathbf{G}_1$ има ребро от v_1 до v_2, в $\mathbf{G}_2$ ще има ребро от $h(v_1)$ до $h(v_2)$. Например ако графът $\mathbf{G}_2$ има два върха и нито едно ребро, то тогава съществува хомоморфизъм от $\mathbf{G}_1$ към $\mathbf{G}_2$ тогава и само тогава, когато върховете на $\mathbf{G}_1$ могат да се оцветят в два цвята, така че никое ребро да не свързва едноцветни върхове, т.е. когато графът е *двуделен*.

3.5.3. Пример. Да видим какво казва нашата дефиниция за хомоморфизъм в случая, когато структурите са графи. Да си припомним от пример 3.1.10, че един граф $\mathbf{G}$ може да се мисли като структура за сигнатура, в която няма символи за константи и функционални символи и има един единствено предикатен символ $\mathtt{p}$ и той е двуместен. Универсумът $|\mathbf{G}|$ на структурата се състои от върховете на графа, а $\mathtt{p}^{\mathbf{G}}(v_1, v_2)$ е истина тогава и само тогава, когато в графа има ребро от връх v_1 до връх v_2.

Тъй като при графите няма нито символи за константи, нито функционални символи, то h е хомоморфизъм от граф $\mathbf{G}_1$ в граф $\mathbf{G}_2$, ако h изобразява всеки връх на $\mathbf{G}_1$ в някой връх на $\mathbf{G}_2$ и винаги когато $\mathtt{p}^{\mathbf{G}_1}(v_1, v_2)$ е истина, т.е. когато в $\mathbf{G}_1$ има ребро от v_1 до v_2, то

$\mathtt{p}^{\mathbf{G}_2}(h(v_1), h(v_2))$ също ще бъде истина, т.е. в $\mathbf{G}_2$ ще има ребро от $h(v_1)$ до $h(v_2)$. Но това е точно дефиницията от теория на графите! Следователно хомоморфизмите в теория на графите също се оказват частен случай на нашето понятие.

Казахме, че дефиниция 3.5.1 е дадена така, че от хомоморфизмите да искаме да запазват възможно най-малко свойства. Да помислим какво в повече бихме могли да поискаме от един хомоморфизъм да запазва по отношение на символите за константи и функционалните символи. Вероятно едно от най-силните условия е следното: ако $h\colon \mathbf{M} \to \mathbf{K}$ е хомоморфизъм и τ е някакъв израз, съставен от символи за константи и функционални символи (т.е. терм), то образът на стойността на τ в $\mathbf{M}$ е равен на стойността на τ в $\mathbf{K}$. Оказва се обаче, че няма нужда изрично да поискаме такова свойство в дефиницията на хомоморфизъм, защото то може да се получи като следствие. Всъщност това е причината, поради която алгебриците нямат онези проблеми, които имат логиците, когато дефинират понятието хомоморфизъм — дори в дефиницията да поискаме от h да удовлетворява възможно най-слабите условия, пак ще се окаже, че h удовлетворява всичко, каквото можем да поискаме от един хомоморфизъм. Следващото твърдение показва точно това.

(Припомнете си означение 3.2.10 б).)

3.5.4. Твърдение. *Нека $h\colon \mathbf{M} \to \mathbf{K}$ е хомоморфизъм и $\tau(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е терм. Тогава за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$*

$$h([\![\tau]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = [\![\tau]\!]^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

Доказателство. С индукция по броя на символите в терма τ. Когато $\tau = \mathtt{x}$ е променлива, то $\tau = \mathtt{x}_i$ за някое i и значи

$$h([\![\tau]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = h(\mu_i) = [\![\tau]\!]^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

Когато $\tau = \mathtt{c}$ е символ за константа

$$h([\![\tau]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) = h(\mathtt{c}^{\mathbf{M}})$$

Тъй като h е хомоморфизъм, то $h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$ и значи може да продължим горното равенство по следния начин:

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}} = [\![\tau]\!]^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_n))$$

Когато $\tau = \mathtt{f}(\tau_1, \ldots, \tau_m)$

$$\begin{aligned} &h([\![\tau]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)) \\ &\qquad = h(\mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n), \ldots, [\![\tau_m]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n))) \end{aligned}$$

Тъй като h е хомоморфизъм, последното е равно на

$$\mathtt{f}^{\mathbf{K}}(h([\![\tau_1]\!]^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n)),\ldots,h([\![\tau_m]\!]^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n)))$$

което пък съгласно индукционното предположение е равно на

$$\mathtt{f}^{\mathbf{K}}([\![\tau_1]\!]^{\mathbf{K}}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n)),\ldots,[\![\tau_m]\!]^{\mathbf{K}}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n))) = [\![\tau]\!]^{\mathbf{K}}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n))$$

■

За съжаление не може да формулираме аналогично твърдение за формули. Без доказателство ще посочим, че е вярно следното значително по-слабо твърдение:

Твърдение. *Нека* $h\colon \mathbf{M}\to\mathbf{K}$ *е хомоморфизъм и* $\varphi(\mathtt{x}_1,\mathtt{x}_2,\ldots,\mathtt{x}_n)$ *е безкванторна формула, в която не се срещат други съждителни операции, освен* $\&$ *и* $\vee$*. Тогава за произволни* $\mu_1,\mu_2,\ldots,\mu_n\in|\mathbf{M}|$

$$\textit{ако } \mathbf{M}\vDash\varphi[\mu_1,\mu_2,\ldots,\mu_n],\ \textit{то } \mathbf{K}\vDash\varphi[h(\mu_1),h(\mu_2),\ldots,h(\mu_n)]$$

Дефиницията на силен хомоморфизъм е аналогична на дефиницията на хомоморфизъм, но в точка 3.5.1 в) използваме еквиваленция вместо импликация:

3.5.5. Дефиниция. Нека $\mathbf{M}$ и $\mathbf{K}$ са произволни структури. Казваме, че h е *силен хомоморфизъм* от $\mathbf{M}$ в $\mathbf{K}$ и запизваме това така:

$$h\colon \mathbf{M}\to\mathbf{K}$$

ако h е изображение от универсума на $\mathbf{M}$ в универсума на $\mathbf{K}$ и освен това:

а) за всеки символ за константа $\mathtt{c}$

$$h(\mathtt{c}^{\mathbf{M}}) = \mathtt{c}^{\mathbf{K}}$$

б) за всеки n-местен функционален символ $\mathtt{f}$ и елементи $\mu_1,\mu_2,\ldots,\mu_n$ на универсума на $\mathbf{M}$

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n)) = \mathtt{f}^{\mathbf{K}}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n))$$

в) за всеки n-местен предикатен символ $\mathtt{p}$ и елементи $\mu_1,\mu_2,\ldots,\mu_n$ на универсума на $\mathbf{M}$

$$\mathtt{p}^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n)\longleftrightarrow\mathtt{p}^{\mathbf{K}}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n))$$

3.5.6. Да видим какъв е смисълът на силните хомоморфизми в алгебрата. Тъй като в алгебричните структури единствен предикатен символ е равенството, то изображението $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм тогава и само тогава, когато h е хомоморфизъм и освен това за произволни елементи μ_1 и μ_2 на универсума на $\mathbf{M}$ е вярно, че

$$\mu_1 = \mu_2 \longleftrightarrow h(\mu_1) = h(\mu_2)$$

Посоката отляво надясно е тривиално вярна. Обратната посока обаче казва, че h е инекция. Оказва се, че това е изключително ограничително изискване, което намалява полезността на това понятие в алгебрата. Всъщност не само между алгебрични структури, но изобщо винаги когато h е силен хомоморфизъм между структури с равенство, h трябва да бъде инекция. За щастие ние ще използваме силни хомоморфизми между структури, в които няма предикатен символ, който се интерпретира като равенство, а в този случай може да има по-нетривиални примери за силни хомоморфизми.

Когато използваме силни хомоморфизми, може да докажем за формулите твърдение, аналогично на твърдение 3.5.4. То не е вярно за произволни формули, а само за безквантворни формули, но и това е нещо...

3.5.7. Твърдение. *Нека $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм и формулата $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ е безквантворна. Тогава за произволни $\mu_1, \mu_2, \ldots, \mu_n$ от универсума на $\mathbf{M}$*

$$\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \mathbf{K} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Доказателство. С индукция по броя на символите в φ. Когато $\varphi = \mathtt{p}(\tau_1, \ldots, \tau_m)$ е атомарна

$$\begin{aligned}\mathbf{M} \vDash \mathtt{p}(\tau_1, \ldots, \tau_m)[\mu_1, \mu_2, \ldots, \mu_n] &\longleftrightarrow \\ &\longleftrightarrow \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_m), \ldots, [\![\tau_m]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_m))\end{aligned}$$

Тъй като h е силен хомоморфизъм, последното е еквивалентно на

$$\mathtt{p}^{\mathbf{K}}(h([\![\tau_1]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_m)), \ldots, h([\![\tau_m]\!]^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_m)))$$

Съгласно твърдение 3.5.4, това пък е еквивалентно на

$$\begin{aligned}\mathtt{p}^{\mathbf{K}}([\![\tau_1]\!]^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_m)), \ldots, [\![\tau_m]\!]^{\mathbf{K}}(h(\mu_1), h(\mu_2), \ldots, h(\mu_m))) \\ \longleftrightarrow \mathbf{K} \vDash \mathtt{p}(\tau_1, \ldots, \tau_m)[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]\end{aligned}$$

Когато $\varphi = \psi_1 \,\&\, \psi_2$

$$\begin{aligned}
&\mathbf{M} \vDash \psi_1 \,\&\, \psi_2[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\
&\quad \longleftrightarrow \mathbf{M} \vDash \psi_1[\mu_1, \mu_2, \ldots, \mu_n] \text{ и } \mathbf{M} \vDash \psi_1[\mu_1, \mu_2, \ldots, \mu_n] \\
&\quad \longleftrightarrow \mathbf{K} \vDash \psi_1[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \text{ и } \mathbf{M} \vDash \psi_1[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \\
&\quad \longleftrightarrow \mathbf{K} \vDash \psi_1 \,\&\, \psi_2[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]
\end{aligned}$$

Когато $\varphi = \psi_1 \vee \psi_2$ или $\varphi = \psi_1 \Rightarrow \psi_2$ се разсъждава аналогично.
Когато пък $\varphi = \neg\psi$

$$\begin{aligned}
&\mathbf{M} \vDash \neg\psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\
&\quad \longleftrightarrow \text{не е вярно, че } \mathbf{M} \vDash \psi[\mu_1, \mu_2, \ldots, \mu_n] \\
&\quad \longleftrightarrow \text{не е вярно, че } \mathbf{K} \vDash \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \\
&\quad \longleftrightarrow \mathbf{K} \vDash \neg\psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]
\end{aligned}$$

Тъй като φ е безквантorна, то други възможности за φ няма. ∎

Току-що доказаното твърдение не е вярно за произволни формули. В теория на моделите се дефинират изображения, наречени *елементарни влагания*, които запазват верността на всички предикатни формули.

3.5.8. Да видим защо не може да докажем горното твърдение за произволни формули. Да разгледаме случая когато $\varphi = \forall \mathtt{x}_i\, \psi$. Тогава имаме:

$$\begin{aligned}
&\mathbf{M} \vDash \forall \mathtt{x}_i\, \psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \\
&\qquad \longleftrightarrow \text{за всяко } \nu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n]
\end{aligned}$$

Съгласно индукционното предположение последното е еквивалентно на

$$\text{за всяко } \nu \in |\mathbf{M}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), h(\nu), h(\mu_{i+1}), \ldots, h(\mu_n)] \quad (17)$$

От друга страна,

$$\begin{aligned}
&\mathbf{K} \vDash \forall \mathtt{x}_i\, \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \longleftrightarrow \\
&\quad \longleftrightarrow \text{за всяко } \kappa \in |\mathbf{K}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), \kappa, h(\mu_{i+1}), \ldots, h(\mu_n)]
\end{aligned} \quad (18)$$

Като сравним (17) с (18) виждаме, че в единия случай твърдим, че нещо е вярно за всеки елемент от универсума на $\mathbf{K}$, а в другия случай — че същото нещо е вярно за всеки елемент от универсума на $\mathbf{K}$, който е от вида $h(\nu)$ за някое $\nu \in |\mathbf{M}|$.

Нека отбележим следното полезно за интуицията наблюдение. Когато формулата φ е безкванторна, тогава верността на $\mathbf{M} \models \varphi[\mu_1, \ldots, \mu_n]$ зависи единствено от свойствата на онези елементи на универсума на $\mathbf{M}$, които могат да се изразят явно посредством символите за константи, функционалните символи, както и елементите $\mu_1, \ldots, \mu_n$. Свойствата на останалите елементи на $\mathbf{M}$ не могат по никакъв начин да повлияят на верността на φ. С други думи, може да кажем, че безкванторните формули имат локален поглед към света. Кванторите пък позволяват на формулите да имат глобален поглед към целия свят. Например $\forall \mathtt{x}\, \psi$ означава, че нещо е вярно за всички елементи на универсума на $\mathbf{M}$ — дори онези, за които не разполагаме с никакъв начин да ги изразим явно.

Следващото твърдение показва, че е възможно да изкажем твърдение подобно на 3.5.7, в което вместо за обикновена вярност се говори за тъждествена вярност или изпълнимост, но вместо импликация ще имаме следване само в едната посока.

3.5.9. Твърдение. *Нека $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм. Тогава за всяка безкванторна формула φ:*

а) $\mathbf{K} \models \varphi \longrightarrow \mathbf{M} \models \varphi$;

б) $\mathbf{M} \models^{\exists} \varphi \longrightarrow \mathbf{K} \models^{\exists} \varphi$.

Доказателство. Нека формулата е $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$.

(а) $\mathbf{K} \models \varphi$ е вярно тогава и само тогава, когато за произволни елементи $\kappa_1, \kappa_2, \ldots, \kappa_n \in |\mathbf{K}|$ е вярно $\mathbf{K} \models \varphi[\kappa_1, \kappa_2, \ldots, \kappa_n]$. Но ако това е така, то в частност за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ ще бъде вярно $\mathbf{K} \models \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$. От твърдение 3.5.7 следва, че за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ е вярно $\mathbf{M} \models \varphi[\mu_1, \mu_2, \ldots, \mu_n]$, следователно $\mathbf{M} \models \varphi$.

(б) $\mathbf{M} \models \varphi$ е вярно тогава и само тогава, когато съществуват такива $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$, че $\mathbf{M} \models \varphi[\mu_1, \mu_2, \ldots, \mu_n]$. От твърдение 3.5.7 следва, че за същите тези $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ е бъде вярно също и $\mathbf{K} \models \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$. Това означава, че съществуват такива $\kappa_1, \kappa_2, \ldots, \kappa_n \in |\mathbf{K}|$, че $\mathbf{K} \models \varphi[\kappa_1, \kappa_2, \ldots, \kappa_n]$, следователно $\mathbf{K} \models^{\exists} \varphi$. ∎

3.5.10. Дефиниция. а) Ако $h\colon \mathbf{M} \to \mathbf{K}$ е силен хомоморфизъм, което е биекция между универсумите на $\mathbf{M}$ и $\mathbf{K}$, казваме, че h е *изоморфизъм*.

б) Когато съществува изоморфизъм между структурите $\mathbf{M}$ и $\mathbf{K}$, казваме, че $\mathbf{M}$ е *изоморфна* на $\mathbf{K}$.

в) Когато $h\colon \mathbf{M} \to \mathbf{M}$ е изоморфизъм на една структура със същата структура, казваме, че h е *автоморфизъм*.

г) Изображението идентитет винаги е автоморфизъм. То се нарича *тривиален автоморфизъм*.

Забележка: Въпреки че има различни понятия за морфизми между структури (хомоморфизъм, силен хомоморфизъм, хомоморфно влагане, елементарно влагане и др.), оказва се, че понятието „изоморфизъм“ е абсолютно. Едно изображение е биективен силен хомоморфизъм, тогава и само тогава, когато е биективно хомоморфно влагане, тогава и само тогава, когато е биективно елементарно влагане.

3.5.11. Твърдение. *а) Всяка структура е изоморфна сама себе си.*

б) Ако структурата $\mathbf{M}$ *е изоморфна с* $\mathbf{K}$*, то* $\mathbf{K}$ *е изоморфна с* $\mathbf{M}$*.*

в) Ако структурата $\mathbf{M}$ *е изоморфна с* $\mathbf{N}$ *и* $\mathbf{N}$ *е изоморфна с* $\mathbf{K}$*, то* $\mathbf{M}$ *е изоморфна с* $\mathbf{K}$*.*

Доказателство. Вж. твърдения 3.5.12, 3.5.13 и 3.5.14. ∎

3.5.12. Твърдение. *Нека* $\mathbf{M}$ *е структура, а* h *е изображението, което изобразява всеки елемент на универсума на* $\mathbf{M}$ *в същия елемент. Тогава* h *е изоморфизъм.*

Доказателство. Очевидно h е биекция. Освен това непосредствено може да се провери, че h удовлетворява изискванията на дефиницията за силен хомоморфизъм. ∎

3.5.13. Твърдение. *Ако* $h\colon \mathbf{M} \to \mathbf{K}$ *е изоморфизъм, то обратното изображение* $h^{-1}\colon \mathbf{K} \to \mathbf{M}$ *също е изоморфизъм.*

Доказателство. Да бъде h е биекция означава за произволни μ от универсума на $\mathbf{M}$ и κ от универсума на $\mathbf{K}$ да е вярно $h^{-1}(h(\mu)) = \mu$ и $h(h^{-1}(\kappa)) = \kappa$. Тъй като тези условия са симетрични спрямо μ и κ, то значи функцията h е обратна на h^{-1}, откъдето следва, че h^{-1} също е биекция.

Остава да докажем, че h^{-1} е силен хомоморфизъм. Това е така, защото

– за всеки символ за константа c, $h^{-1}(\mathsf{c}^{\mathbf{K}}) = h^{-1}(h(\mathsf{c}^{\mathbf{M}})) = \mathsf{c}^{\mathbf{M}}$.

– за всеки n-местен функционален символ $\mathtt{f}$

$$\begin{aligned} h^{-1}(\mathtt{f}^{\mathbf{K}}(\kappa_1,\kappa_2,\ldots,\kappa_n)) = \\ &= h^{-1}(\mathtt{f}^{\mathbf{K}}(h(h^{-1}(\kappa_1)),h(h^{-1}(\kappa_2)),\ldots,h(h^{-1}(\kappa_n)))) \\ &= h^{-1}(h(\mathtt{f}^{\mathbf{K}}(h^{-1}(\kappa_1),h^{-1}(\kappa_2),\ldots,h^{-1}(\kappa_n)))) \\ &= \mathtt{f}^{\mathbf{K}}(h^{-1}(\kappa_1),h^{-1}(\kappa_2),\ldots,h^{-1}(\kappa_n)) \end{aligned}$$

– за всеки n-местен предикатен символ $\mathtt{p}$

$$\begin{aligned} \mathtt{p}^{\mathbf{K}}(\kappa_1,\kappa_2,\ldots,\kappa_n) &\longleftrightarrow \\ &\longleftrightarrow \mathtt{p}^{\mathbf{K}}(h(h^{-1}(\kappa_1)),h(h^{-1}(\kappa_2)),\ldots,h(h^{-1}(\kappa_n))) \\ &\longleftrightarrow \mathtt{p}^{\mathbf{K}}(h^{-1}(\kappa_1),h^{-1}(\kappa_2),\ldots,h^{-1}(\kappa_n)) \end{aligned}$$

■

3.5.14. Твърдение. *Нека $h_2\colon \mathbf{M}\to\mathbf{N}$ и $h_1\colon \mathbf{N}\to\mathbf{K}$ са силни хомоморфизми и h е композицията на h_1 и h_2, т.е. $h(\mu)=h_1(h_2(\mu))$ за всеки елемент μ на универсума на $\mathbf{M}$. Тогава $h\colon \mathbf{M}\to\mathbf{K}$ е силен хомоморфизъм.*

Доказателство. Очевидно h изобразява елементите на $|\mathbf{M}|$ в елементи на $|\mathbf{K}|$. Освен това:

– За всеки символ за константа $h(\mathtt{c}^{\mathbf{M}})=h_1(h_2(\mathtt{c}^{\mathbf{M}}))=h_1(\mathtt{c}^{\mathbf{N}})=\mathtt{c}^{\mathbf{K}}$.

– За всеки n-местен функционален символ и елементи $\mu_1,\mu_2,\ldots,\mu_n$ на универсума на $\mathbf{M}$

$$h(\mathtt{f}^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n))=h_1(h_2(\mathtt{f}^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n)))$$

Тъй като h_2 е силен хомоморфизъм, то последното е равно на

$$h_1(\mathtt{f}^{\mathbf{N}}(h_2(\mu_1),h_2(\mu_2),\ldots,h_2(\mu_n)))$$

Тъй като h_1 също е силен хомоморфизъм, това е равно на

$$\mathtt{f}^{\mathbf{K}}(h_1(h_2(\mu_1)),h_1(h_2(\mu_2)),\ldots,h_1(h_2(\mu_n)))=\mathtt{f}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n))$$

– Нека $\mathtt{p}$ е n-местен предикатен символ и $\mu_1,\mu_2,\ldots,\mu_n$ са елементи на универсума на $\mathbf{M}$. Тъй като h_2 е силен хомоморфизъм, то

$$\mathtt{p}^{\mathbf{M}}(\mu_1,\mu_2,\ldots,\mu_n)\longleftrightarrow\mathtt{p}^{\mathbf{N}}(h_2(\mu_1),h_2(\mu_2),\ldots,h_2(\mu_n))$$

Тъй като h_1 също е силен хомоморфизъм, това е еквивалентно на

$$\begin{aligned} \mathtt{p}^{\mathbf{K}}(h_1(h_2(\mu_1)),&h_1(h_2(\mu_2)),\ldots,h_1(h_2(\mu_n)))\longleftrightarrow \\ &\longleftrightarrow \mathtt{f}^{\mathbf{K}}(h(\mu_1),h(\mu_2),\ldots,h(\mu_n)) \end{aligned}$$

■

3.5.15. Твърдение. *Нека* $h\colon \mathbf{M} \to \mathbf{K}$ *е изоморфизъм. Тогава за произволни формула* $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ *и* $\mu_1, \mu_2, \ldots, \mu_n$ *от универсума на* $\mathbf{M}$

$$\mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow \mathbf{K} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Доказателство. С индукция по дължината на формулата φ. Случаите, когато φ е атомарна формула или е получена с някоя съждителна операция се разглеждат идентично както в доказателството на твърдение 3.5.7. Ще разгледаме само случаите с квантори.

Когато $\varphi = \forall \mathtt{x}_i\, \psi$

$$\mathbf{M} \vDash \forall \mathtt{x}_i\, \psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow$$
$$\longleftrightarrow \text{за всяко } \nu \in |\mathbf{M}| \text{ е вярно } \mathbf{M} \vDash \psi[\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n]$$

Съгласно индукционното предположение последното е еквивалентно на

$$\text{за всяко } \nu \in |\mathbf{M}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), h(\nu), h(\mu_{i+1}), \ldots, h(\mu_n)]$$

Тъй като h е сюрекция, това е еквивалентно на

$$\text{за всяко } \kappa \in |\mathbf{K}|\ \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), \kappa, h(\mu_{i+1}), \ldots, h(\mu_n)]$$

т.е. на

$$\mathbf{K} \vDash \forall \mathtt{x}_i\, \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

Когато $\varphi = \exists \mathtt{x}_i\, \psi$

$$\mathbf{M} \vDash \exists \mathtt{x}_i\, \psi[\mu_1, \mu_2, \ldots, \mu_n] \longleftrightarrow$$
$$\longleftrightarrow \text{съществува такова } \nu \in |\mathbf{M}|, \text{ че } \mathbf{M} \vDash \psi[\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n]$$

Съгласно индукционното предположение последното е еквивалентно на

$$\text{съществува такова } \nu \in |\mathbf{M}|, \text{ че } \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), h(\nu), h(\mu_{i+1}), \ldots, h(\mu_n)]$$

Тъй като h е сюрекция, това е еквивалентно на

$$\text{съществува такова } \kappa \in |\mathbf{K}|, \text{ че } \mathbf{K} \vDash \psi[h(\mu_1), \ldots, h(\mu_{i-1}), \kappa, h(\mu_{i+1}), \ldots, h(\mu_n)]$$

т.е. на

$$\mathbf{K} \vDash \forall \mathtt{x}_i\, \psi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)]$$

■

Забележка: От доказателството на твърдение 3.5.15 се вижда, че то е вярно не само за изоморфизми, но и за произволни сюрективни силни хомоморфизми.

Да напомним, че съгласно 3.3.11 б) когато пишем $\varphi(\mathtt{x})$ имаме предвид, че единствено $\mathtt{x}$ може да бъде свободна променлива на φ. С други думи $\mathtt{x}$ е единствената свободна променлива на φ или φ няма свободни променливи. Когато пишем $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, това означава, че всички свободни променливи на φ са измежду различните променливи $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

3.5.16. Дефиниция. Нека $\mathbf{M}$ е структура.

а) Подмножество X на универсума на $\mathbf{M}$ е *определимо* посредством формулата $\varphi(\mathtt{x})$, ако за всяко $\mu \in |\mathbf{M}|$

$$\mu \in X \longleftrightarrow \mathbf{M} \vDash \varphi[\mu]$$

б) Подмножество $X \subseteq |\mathbf{M}|^n$ е *определимо* посредством формулата $\varphi(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, ако за произволни елементи $\mu_1, \mu_2, \ldots, \mu_n$ на универсума на $\mathbf{M}$

$$\langle \mu_1, \mu_2, \ldots, \mu_n \rangle \in X \longleftrightarrow \mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n]$$

3.5.17. Теорема за определимите множества. *Нека* $\mathbf{M}$ *е структура и* $h\colon \mathbf{M} \to \mathbf{M}$ *е произволен автоморфизъм.*

а) Ако подмножество X *на универсума на* $\mathbf{M}$ *е определимо, то за всеки елемент* μ *на универсума на* $\mathbf{M}$

$$\mu \in X \longleftrightarrow h(\mu) \in X$$

б) Ако подмножество $X \subseteq |\mathbf{M}|^n$ *е определимо, то за произволни елементи* $\mu_1, \mu_2, \ldots, \mu_n$ *на универсума на* $\mathbf{M}$

$$\langle \mu_1, \mu_2, \ldots, \mu_n \rangle \in X \longleftrightarrow \langle h(\mu_1), h(\mu_2), \ldots, h(\mu_n) \rangle \in X$$

Доказателство. **(а)** е частен случай на (б).

(б) Нека множеството X е определимо посредством формулата φ. Тъй като всеки автоморфизъм е изоморфизъм, съгласно твърдение 3.5.15

$$\begin{aligned} \langle \mu_1, \mu_2, \ldots, \mu_n \rangle \in X &\longleftrightarrow \mathbf{M} \vDash \varphi[\mu_1, \mu_2, \ldots, \mu_n] \\ &\longleftrightarrow \mathbf{M} \vDash \varphi[h(\mu_1), h(\mu_2), \ldots, h(\mu_n)] \\ &\longleftrightarrow \langle h(\mu_1), h(\mu_2), \ldots, h(\mu_n) \rangle \in X \end{aligned}$$

■

Накрая да отбележим, че ако добавим нов функционален или предикатен символ към сигнатурата и съответно към структурата, то това, което е било автоморфизъм в по-бедната структура, може да престане да бъде автоморфизъм в по-богатата структура.

Нека например структурата е с универсум реалните числа, единствен функционален символ е „+“, който се интерпретира като събиране и единствен предикатен символ е равенството. Тогава изображението

$$h(x) = -x$$

е автоморфизъм, защото:

1. h е биекция;
2. $h(x + y) = h(x) + h(y)$;
3. $x = y \longleftrightarrow h(x) = h(y)$.

Ако обаче обогатим структурата с нов функционален символ „.“, който се интерпретира като умножение, тогава h престава да бъде автоморфизъм. Всъщност, оказва се, че в обогатената структура единственият автоморфизъм е тривиалният автоморфизъм. Такива структури се наричат *твърди структури*.

Нещо аналогично се случва и във физиката. Например ако обърнем посоката на времето, фундаменталните закони на физиката остават в сила... Е, остават в сила, но с едно „незначително“ изключение за т.н. слабо взаимодействие. Можем да преведем на логически език това по следния начин — ако в сигнатурата не присъстват символи, изразяващи слабото взаимодействие, тогава обръщането на посоката на времето се оказва автоморфизъм, но ако обогатим структурата със слабото взаимодействие, тогава това престава да бъде автоморфизъм.

3.6. Интуиционистка логика

Класическа и конструктивна математика

При класическия начин за правене на математика съжденията, които изказваме, имат дескриптивен, т.е. описателен характер. Във всяко математическо твърдение се говори за свойствата, притежавани от математическите обекти. Самите математически обекти са статични и неизменни — по времето на древните гърци те са притежавали същите свойства, които притежават и днес. Математическите обекти сякаш живеят в свой идеален и неизменен свят. Математикът не се меси в този свят, не нарушава спокойствието му, а само като страничен наблюдател

описва свойствата на математическите обекти. Ако изкажем съждението, че всяко реално число, повдигнато на квадрат, е неотрицателно, верността на това съждение няма нищо общо с това какво можем или не можем, какво искаме или не искаме, какво знаем или не знаем.

Конструктивният начин за правене на математика е алтернатива на класическия. При него също можем да изказваме дескриптивни съждения, но освен това обръщаме внимание и на това какво ние самите можем да правим с математическите обекти. Две числа не просто имат сбор, но могат да бъдат събирани, едно уравнение не просто има решение, но може да бъде решавано, производната на една функция не просто съществува, но може да бъде пресмятана и т.н. Например когато разработваме алгебрата конструктивно, ние не се задоволяваме с това да изкажем съждение според което полиномите с комплексни коефициенти имат комплексен корен, а започваме да изследваме въпроса дали има начин, с който да можем да намерим корена. С други думи, когато правим математиката конструктивно, в математическите съждения се говори не само за идеални математически обекти, но и за нас самите — какво можем или не можем да правим с математическите обекти.

Забележка: Има различни философски теории за математиката. Въз основа на тези философии много от конструктивистите не биха се съгласили с обясненията, които ще дадем за това какво представлява и как се прави конструктивната математика. Тук обаче няма да обръщаме никакво внимание на тези противоречащи си една на друга философии, нито на това кое според тях е „правилно“ и кое „неправилно“. В частност няма да се интересуваме от това дали някоя математическа теория — конструктивна или не — противоречи на едни или други философски принципи. Математическа значимост, а значи и значимост от гледна точка на практическите приложения, имат единствено следните три неща:

- Кои съждения могат да бъдат формулирани, използвайки езика на дадена математическа теория?
- Каква част от съжденията теорията може да докаже?
- Какви допускания теорията приема без доказателство?

Ще отбележим все пак едно по-съществено несъответствие между тукашните обяснения и едно от най-важните конструктивни направления в математиката — *интуиционизмът*. Според обясненията които даваме тук, също както и в класическата математика, ако допуснем, че даден обект не съществува и стигнем до противоречие, това означава, че този обект все пак съществува, макар и не винаги да сме състояние

да го намерим. Това означава, че в нашите разсъждения предполагаме съществуването на някакъв „свят“, в който съществуват математическите обекти. Ако допуснем, че в този свят даден обект не съществува и стигнем до противоречие, значи въпросният обект съществува. От друга страна, според математическия интуиционизъм не е правомерно да предполагаме съществуването на математически свят, в който има математически обекти, за които не можем дори въобразаемо да допуснем, че могат да бъдат конструирани. Можем да считаме, че съществува само това, което наистина можем конструираме и можем да считаме за вярно само това, което можем да докажем, че е вярно. Ако допуснем, че даден обект не съществува, ние всъщност допускаме, че не сме в състояние да конструираме въпросния обект. Ако това ни доведе до противоречие, това означава, че никога няма да можем да стигнем до извода, че не можем да конструираме дадения обект. С други думи, ако допуснем, че даден обект не съществува и стигнем до противоречие, това означава само, че винаги ще съществува надеждата, че този обект съществува, но само това — няма никакви гаранции, че обектът наистина съществува.

Интуиционистка логика

Това, че в конструктивната математика можем да изказваме и доказваме съждения, които не могат дори да се формулират на езика на класическата математика, може да ни наведе на мисълта, че конструктивната математика се нуждае от логика с нови логически операции, при която съжденията трябва да се доказват по принципно нов начин. За щастие, оказва се, че това съвсем не е така. Например всички неща, които са доказани в този математически текст до момента, са доказани конструктивно и за да направим това дори не ни се наложи да предупредим предварително читателя. Използваните тук доказателства на различните твърдения са конструктивни, но в същото време те могат да се четат и от хора, които никога не са чували за конструктивната математика, нито пък знаят какво означава едно доказателство да бъде конструктивно.

Как става възможно това? Начинът е следният — ще използваме обичайните логически операции, но ще ги тълкуваме по нов начин. Това между другото означава, че конструктивната математика се прави донякъде на принципа „говорим едно, ама разбираме друго“.

Когато един конструктивист изкаже твърдение от вида

„съществува такова x, че ...“,

той всъщност има предвид

„може да се намери такова x, че ...“.

Когато пък конструктивистът изкаже твърдение от вида

„A или B“,

той всъщност има предвид

„може да се установи дали A, или B“.

За останалите логически операции може да считаме, че те в конструктивната математика се тълкуват по същия начин, както и в класическата.

Дали това, че тълкуваме някои от логическите операции по друг начин, означава, че в конструктивната математика трябва да се използват по-различни доказателства, в сравнение с класическата? Удивително е, че отговорът на този въпрос е отрицателен — логиката, която се използва в конструктивната математика, наречена *интуиционистка логика*, не се нуждае от по-различен вид доказателства в сравнение с класическата логика.* Достатъчно ще бъде да спазваме две прости правила, и това ще ни гарантира, че доказателството е конструктивно.

Първото правило е следното: не трябва да използаме метода „допускане на противното“. Ако допуснем A и стигнем до противоречие, това ще ни гарантира, че A не е вярно, но ако допуснем, че A не е вярно и стигнем до противоречие, ние ще докажем не съждението A, а неговото двойно отрицание. Ще видим, че когато разсъждаваме конструктивно, двойното отрицание на едно съждение не винаги може да се счита еквивалентно на самото съждение.

Второто правило за правене на конструктивни доказателства е следното: имаме право да разглеждаме случаи за това дали едно съждение е вярно, или не, само тогава, когато разполагаме с метод, посредством който можем да проверим дали съждението е вярно, или не.

По-нататък в този раздел ще видим с конкретни примери как спазването на тези две прости правила наистина ще ни позволи да доказваме твърденията по такъв начин, че доказателствата да са верни без значение дали тълкуваме изказванията на съждения както в конструктивната математика, или както в класическата.

*Всъщност ако интуиционистката логика се нуждаеше от по-различен вид доказателства, то не би имало особен смисъл да използваме обичайните класически логически операции, пък да ги тълкуваме по друг начин. Вместо „съществува x“ можеше да казваме „може да се намери x“ и вместо „A или B“ можеше да казваме „може да установим дали A, или B“.

Двойно отрицание

Нека видим защо в интуиционистката логика двойното отрицание не се унищожава. Нека A е следното твърдение:

> „Съществува цифра, която се среща безброй много пъти в десетичното представяне на π“.

Тъй като има само десет различни цифри — 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9 — то няма как всяка от тях да се среща само краен брой пъти в десетичното представяне на числото π. Следователно твърдението A е вярно.

Ако обаче изтълкуваме изказването на това твърдение конструктивно, тогава смисълът му ще стане следният:

> „Може да се намери цифра, която се среща безброй много пъти в десетичното представяне на π“.

Засега за нито една от десетте цифри не знаем със сигурност дали се среща безброй много пъти в десетичното представане на π и затова ако тълкуваме изказването на твърдение A конструктивно, тогава това твърдение не може да се счита за доказано. Най-вероятно всяка цифра среща безброй много пъти, но засега това не е нищо повече от предположение.

Да допуснем, че твърдението A е невярно, означава да допуснем, че не съществува цифра, която се среща безброй много пъти в десетичното представяне на π. От тук не е трудно да се стигне до противоречие, следователно това допускане не е вярно. Щом като допускането, че твърдението A не е вярно, води до противоречие, значи е вярно двойното отрицание на A.

И така, доказахме двойното отрицание на A, макар че засега самото твърдение A остава недоказано.

При интуиционистката логика всяко математическо твърдение носи определена информация. Когато информацията представлява някой конкретен математически обект или метод за преобразуване на математически обекти, казваме, че тази информация е конструктивна. Например информацията за конкретна цифра, която се среща безброй пъти в π е конструктивна. Когато пък информацията не ни дава конкретни математически обекти или методи, а само сведения за свойствата на математическите обекти, казваме, че тази информация е дескриптивна. Например информацията, която казва, че има цифра, която се среща безброй пъти в π, но не ни казва коя точно е тази цифра, е дескриптивна.

Какъв е интуитивният смисъл на двойното отрицание? Оказва се, че можем да използваме двойно отрицание, за да премахнем конструктив-

ната информация от едно съждение. Например твърдението A ни дава конкретна цифра, която се среща безброй пъти в десетичното представяне на π. Тъй като засега не знаем коя е тази цифра, то твърдението A все още няма конструктивно доказателство. Отрицанието на A казва, че няма цифра, която се среща безброй пъти, и значи двойното отрицание на A казва, че не е възможно да няма цифра, която се среща безброй пъти. Е, щом не е възможно да няма цифра, която се среща безброй пъти в десетичното представяне на π, значи такава цифра съществува. Твърдението „не не A“ обаче не ни дава информация коя е тази цифра — видяхме, че ние можем да докажем това твърдение дори и без да знаем коя точно цифра се среща безброй пъти.

3.6.1. Факт. *Съждението A е дескриптивно, т.е. не носи конструктивна информация, тогава и само тогава, когато A е еквивалентно на „не не A“.*

Да отбележим, че няма смисъл да правим тройно отрицание на едно съждение, защото тройното отрицание е еквивалентно на единичното. Това е така, защото единичното отрицание на едно съждение просто казва, че нещо е невъзможно или невярно, и не ни носи никаква конструктивна информация. Единственото нещо, което прави двойното отрицание, пък е това да премахне конструктивната информация от едно съждение. Затова ако приложим двойно отрицание към съждение, което не съдържа конструктивна информация, например към единично отрицание, резултатът ще бъде еквивалентно на него съждение.

Твърдение. *За произволно съждение A*

а) от A следва „не не A“;

б) от „не A“ следва „не не не A“;

в) от „не не не A“ следва „не A“;

г) „не не не A“ е еквивалентно на „не A“.

Доказателство. **(а)** Да допуснем, че A е вярно. Трябва да докажем „не не A“. За да докажем това, да допуснем, че „не A“. От това веднага получаваме противоречие (имаме едновременно A и не A) и значи допускането „не A“ е невъзможно. Значи е вярно „не не A“.

(б) се получава от (а) като сложим „не A“ на мястото на A.

(в) Да допуснем, че „не не не A“. Трябва да докажем, че „не A“. За да докажем това, да допуснем, че A е вярно. От тук и от (а) получаваме

„не не A“, а това ни дава противоречие (имаме „не не A“ и отрицанието на „не не A“). Противоречието се дължи на допускането, че A е вярно и значи A не е вярно.

(г) следва от (б) и (в). ∎

Забележка: От тук нататък, когато искаме да изкажем съждение с двойно отрицание, ще използваме изрази от вида „неможе да не е вярно, че ...“. Например следното твърдение е вярно конструктивно: „не може да не съществува цифра, която се среща безброй много пъти в десетичното представяне на числото π“.

Превод на класическата логика в интуиционистката

Благодарение на двойното отрицание, всичко, което можем да правим в класическата математика, може да бъде направено и в конструктивната. Вече споменахме, че има две логически операции, които в интуиционистката логика се тълкуват по-различно, отколкото в класическата — кванторът за съществуване и дизюнкцията. Конструктивното съществуване ни казва кой точно е съществуващият обект, докато класическото съществуване ни казва, че обектът съществува по принцип, без да ни дава метод как да намерим този обект. Конструктивната дизюнкция „A или B“ ни казва дали A е вярно, или B, докато при класическата дизюнкция може и да не знаем кое точно от тези две съждения е вярно. Също така видяхме, че можем да използваме двойно отрицание, за да премахваме конструктивната информация от едно съждение. Това означава, че можем да използваме двойно отрицание, за да преведем всяко съждение, използващо класическата логика, на езика на интуиционистката логика.

3.6.2. Факт. *Ако вземем съждение, изказано използвайки класическата логика, и пред всеки квантор за съществуване, както и пред всяка дизюнкция, сложим двойно отрицание, ще получим равносилно на него съждение, изказано използвайки интуиционистката логика.*

Този превод на класическата логика в интуиционистката се нарича *отрицателен превод*.

Конструктивната математика включва в себе си класическата математика, но я разширява по съществен и нетривиален начин. Така например Аритметиката на Пеано представлява формална теория, в която могат да се изказват съждения за естествени числа. Конструктивен аналог на Аритметиката на Пеано е Аритметиката на Хейтинг.

Всяко съждение, което може да се формулира на езика на Аритметиката на Пеано, може да се преведе и на езика на Аритметиката на Хейтинг и всяко съждение, което може да се докаже от първата теория, може да се докаже и от втората. На езика на Аритметиката на Хейтинг обаче могат да се формулират и много съждения, които не могат да се изкажат на езика на Аритметиката на Пеано.*

Аналогично е положението и с Теорията на Цермело – Френкел (ZF), която представлява формална теория, в която могат да се изказват съждения за множества и обикновено се използва като основа на съвременната класическа математика. Конструктивен аналог на ZF е Интуиционистката теория на Цермело – Френкел (IZF). Всяко съждение, което може да се формулира на езика на ZF, може да се преведе и на езика на IZF и всяко съждение, което може да се докаже от първата теория, може да се докаже и от втората. На езика на IZF обаче могат да се формулират и много съждения, които не могат да се изкажат на езика на ZF.

Забележка: Исторически конструктивната математика възникнала във връзка със желанието да се даде по-обоснован фундамент на математиката. Считало се, че класическата математика е необоснована и заплашена от вътрешни противоречия, докато съждения в конструктивната математика имат ясен смисъл и затова е много невероятно в нея да се открие противоречие. Реалното сравнение на теоретикодоказателствената сила на различни класически и конструктивни теории показва, че това мнение е неправилно. Например ако някога се открие противоречие в теорията ZF, то и теорията IZF ще се окаже противоречива. Въпросът, който има реално значение за основите на математиката, е не това дали да правим математиката конструктивно, или не, а това дали да позволяваме т.н. *непредикативни дефиниции*. Мнението, че конструктивната математика е „по-обоснованият“ начин за правене на математика, се дължи на това, че теорията ZF е непредикативна, докато почти всички математици-конструктивисти предпочитат да правят математиката предикативно. Вместо IZF те използват теорията CZF (която е предикативен аналог на IZF), теория на типовете или дори език без множества.

*Едно такова съждение е *тезисът на Чърч*. На езика на Аритметиката на Хейтинг тезисът на Чърч може да се формулира така:

$$\forall \mathtt{x}\, \exists \mathtt{y}\, \varphi[\mathtt{x}, \mathtt{y}] \Rightarrow \exists \mathtt{e}\, \forall \mathtt{x}\, (!\{\mathtt{e}\}(\mathtt{x}) \,\&\, \varphi[\mathtt{x}, \{\mathtt{e}\}(\mathtt{x})])$$

Изоморфизъм на Къри – Хауърд

Приложенията на интуиционистката логика далеч не се изчерпват само с това, че тя ни позволява да разработваме математиката конструктивно. Оказва се, че всяко съждение, изказано посредством интуиционистката логика, може да се интерпретира като тип данни, а всяко интуиционистко доказателство ни дава обект, притежаващ съответния тип данни. Вярно е и обратното — типовете данни в езиците за програмиране може да се интерпретират като интуиционистки изказвания на съждения, а обектите, дефинирани в компютърните програми — като интуиционистки доказателства. Тази връзка между логика и програмиране се нарича *изоморфизъм на Къри – Хауърд*.

За да си обясним как е възможно съжденията да бъдат типове данни, а програмните обекти — доказателства, да разгледаме един пример. Нека A е съждението

„Можем да намерим мегапираня“,

а B е съждението

„Можем да намерим хипопозавър“.

Конструктивната информация, която се съдържа в съждението A, е метод за намиране на мегапираня, а конструктивната информация, която се съдържа в съждението B, е метод за намиране на хипопозавър. Да докажем съждението A означава да посочим метод за намиране на мегапираня, а да докажем B означава да посочим метод за намиране на хипопозавър. На съждението A съответства типът данни „метод за откриване на мегапираня“, а на съждението B — „метод за откриване на хипопозавър“. Всеки обект, чийто тип е „метод за откриване на мегапираня“, може да се счита за доказателство на съждението A и всеки обект, чийто тип е „метод за откриване на хипопозавър“, може да се счита за доказателство на съждението B.

Нека видим как изглежда това съответствие между съждения и типове и между доказателства и програмни обекти при различните логически операции.

Конюнкция

Съждението „A и B“ казва, че A и B са верни, т.е. разполагаме с метод за откриване на мегапираня и метод за откриване на хипопозавър. Това означава, че информацията, която се съдръжа в съждението „A и B“ е комбинация от информацията, която се съдържа в A, и информацията, която се съдържа в B.

Следователно на съждението „A и B“ съответства следният тип данни: „наредена двойка от метод за откриване на мегапираня и метод за откриване на хипопозавър“. На хаскел този тип се обозначава така:

`(`A`,`B`)`

В математиката доказваме конюнкцията по следния начин:

– Доказваме A.

– Доказваме B.

– От тук заключаваме, че е вярно „A и B“.

Ако x е обект от тип A, а y е обект от тип B, тогава наредената двойка от x и y на хаскел се обозначава така:

`(`x`,`y`)`

Следователно, ако си мислим x като доказателство на A и y — като доказателство на B, то `(`x`,`y`)` ще бъде доказателство на „A и B“

Обратно, ако вече сме доказали „A и B“, от тук директно можем да получим като следствия A и B. Също и на хаскел ако вече разполагаме с обект z от тип `(`A`,`B`)`, тогава

`fst` z

ще ни даде първия елемент на z, а

`snd` z

ще ни даде втория. Следователно ако z е доказателство на „A и B“, то „`fst` z“ ще бъде доказателство на A, а „`snd` z“ ще бъде доказателство на B.

Дизюнкция

Когато тълкуваме дизюнкцията конструктивно, съждението „A или B“ има следния смисъл:

> „Може да се установи дали можем да намерим мегапираня,
> или можем да намерим хипопозавър.“

Следователно съждението „A или B“ ни дава метод за откриване на мегапираня или метод за откриване на хипопозавър.

На съждението „A и B“ съответства тип данни, който в известен смисъл представлява обединение на типовете A и B. Всеки обект от тип „A или B“ ни дава обект от тип A или обект от тип B. На хаскел този тип се обозначава така:

`Either` A B

В математиката доказваме дизюнкцията по следните два начина:

– Доказваме A.

– От тук заключаваме, че е вярно „A или B“.

Втори начин:

– Доказваме B.

– От тук заключаваме, че е вярно „A или B“.

Също и на хаскел ако x е обект от тип A, тогава

`Left(`x`)`

е обект от тип „`Either` A W“, където на мястото на W може да стои произволен тип. Значи ако си мислим x като доказателство на A, тогава „`Left(`x`)`“ представлява доказателство на „A или B“.

Аналогично, ако y е обект от тип B, тогава

`Right(`y`)`

е обект от тип „`Either` W B“, където на мястото на W може да стои произволен тип. Ако си мислим y като доказателство на B, тогава „`Right(`y`)`“ представлява доказателство на „A или B“.

Обратно, ако вече сме доказали „A или B“, начинът да използваме това съждение е да разглеждаме случаи:

– Първи случай — вярно е A. Използвайки A, доказваме някакво съждение C.

– Втори случай — вярно е B. Използвайки B, доказваме някакво съждение C.

– От тук заключаваме, че C е вярно.

Аналогично на това, и на хаскел ако вече разполагаме с обект z от тип `Either` A B, един начин да го използваме е следната конструкция:

```
u = case z of
   Left(x) →
      използвайки x намираме обект от тип C
   Right(y) →
      използвайки y намираме обект от тип C
```

Импликация

Съждението „ако A, то B“ има следния смисъл:

> „Ако можем да намерим мегапираня, то ще можем да намерим хипопозавър.“

Следователно съждението „ако A, то B“ ни дава метод, посредством който ако ни бъде даден метод за откриване на мегапираня, ще можем да получим метод за откриване на хипопозавър.

На съждението „ако A, то B“ съответства типът данни „функция, чийто аргумент е от тип A (т.е. метод за откриване на мегапираня), а стойността — от тип B (т.е. метод за откриване на хипопозавър)“. На хаскел този тип се обозначава така:

A `->` B

В математиката доказваме импликацията по следния начин:

– Допускаме, че A е вярно.

– Започваме да правим разсъждения, използващи A.

– В края на тези разсъждения доказваме B

– От тук заключаваме, че от A следва B.

На доказателство от този вид съответства приблизително следният код на хаскел:

```
\x →
   let ...
       ...използвайки x дефинираме разни обекти
       ...
   in
      израз от тип B
```

Този код представлява анонимна функция с аргумент `x` от тип A, която връща стойност от тип B. За да можем да дадем на хаскел по-конкретен пример, да разгледаме съждението „ако A и B, то B и A“. Това съждение е очевидно вярно. На хаскел на него съответства функция, която взема като аргумент наредена двойка `(a,b)` и връща като стойност наредената двойка `(b,a)`:

```
\x →                  -- x е аргументът на функцията
   let a = fst(x)     -- приемаме, че x е
       b = snd(x)     -- наредената двойка (a,b)
   in
      (b,a)           -- връщаме като стойност (b,a)
```

Ако вече сме доказали съжденията A и „ако A, то B“, то от тях ще получим като следствие и съждението B.

Аналогично на това, и на хаскел ако x е обект от тип A и f е функция от тип A`->`B, то $f(x)$ ще бъде обект от тип B.

Отрицание

Отрицанието в интуиционистката логика има две особености, които ще приемем без да ги обосноваваме.

Първата особеност на отрицанието е следната: приемаме, че ако дадено съждение не е вярно, то от него следва всяко съждение. Тази особеност на интуиционистката логика е присъща и на класическата логика, а значи и на цялата математика.

Втората особеност на отрицанието е следната. Да си представим на ум, че по някакъв начин сме се сдобили с мегапираня, и да допуснем, че от тук можем да стигнем до противоречие. С други думи, допускането, че по някакъв начин сме намерили мегапираня, води до противоречие. Ще можем ли да стигнем до противоречие ако допуснем, че съществува мегапираня без да допускаме, че сме намерили тази мегапираня? От правилата на интуиционистката логика можем да заключим, че отговорът на този въпрос трябва да е положителен.

Смисълът на съждението „не A“ е следният:

> „Не съществува мегапираня.“

Начинът, по който доказваме такова съждение е следният:

– Допускаме, че A е вярно.

– Започваме да правим разсъждения, използващи A.

– В края на тези разсъждения стигаме до противоречие

– От тук заключаваме, че A не е вярно.

На хаскел няма непосредствен начин за представяне на отрицанието. Но тъй като съжденията „A не е вярно“ и „от A следва противоречие“ са еквивалентни, то може да считаме, че на съждението „не A“ съответства типът `A -> Impossible`, където `Impossible` е празният тип, т.е. тип който няма нито една стойност. Празният тип не е вграден, но в хаскел 2010 той може да се дефинира така:

```
data Impossible
```

Квантор за всеобщност

Да разгледаме съждението

> „За всеки зъб на мегапираня може да се намери мегапираня, която го е притежавала.“

Това съждение ни дава метод, посредством който ако ни бъде даден зъб от мегапираня, ще можем да получим метод за намиране на мегапиранята, която е притежавала този зъб. На това съответства типът

данни „функция, чийто аргумент е обект x от тип „мегапирански зъб“, а стойността от тип „мегапираня, притежател на x“.

На пръв поглед, разгледан като тип данни, кванторът за всеобщност се интерпретира по същия начин, както и импликацията — като функция. Има обаче и една съществена разлика. При импликацията типовете на аргумента и на стойността са фиксирани и отнапред известни. При квантора за всеобщност аргументът също е има фиксиран тип, но типът на върнатата стойност зависи от аргумента, защото в израза „мегапираня, притежател на x“ се споменава x. Такива типове се наричат *зависими* (dependent types).

Един друг пример за зависим тип ни дава функцията, която получава като аргумент някакво естествено число n и връща като стойност n-мерен нулев вектор. Аргументът на тази функция е от тип „естествено число“, а връщаната стойност — „n-мерен вектор“ е зависим тип, защото зависи от n.

Хаскел няма вградена поддръжка за зависими типове,* но често няма проблем да пишем програми и да си мислим, че те използват зависими типове — просто компилаторът ще пресметне някакви по-общи типове. Например разгледаната по-горе функция, която връща n-мерен нулев вектор, ще връща стойност от тип „вектор“ без да се уточнява, че векторът е n-мерен.

В математиката доказваме съждение от вида „за всяко x е вярно $C(x)$“ по следния начин:

- Избираме произволен обект x.
- Започваме да правим разсъждения за x.
- В края на тези разсъждения доказваме $C(x)$.
- От тук заключаваме, че за всяко x е вярно $C(x)$.

На доказателство от този вид съответства приблизително следният код на хаскел:

```
\x →
   let ...
       ... използвайки x дефинираме разни обекти
       ...
   in
      израз от тип C(x)
```

Този код представлява анонимна функция с аргумент `x`, която връща стойност от тип $C(x)$.

*Има начин поведението им донякъде да се имитира, вж. [13].

Ако вече сме доказали съждението „за всяко x е вярно $C(x)$“, то за всеки конкретен обект x ще можем да получим като следствие $C(x)$.

Аналогично на това, и на хаскел ако f е функция която по аргумент x ни връща обект от тип $C(x)$, то $f(x)$ ще бъде обект от тип $C(x)$.

Квантор за съществуване

Да разгледаме съждението

> „Съществува мегапираня, която е загубила зъб в река Ориноко.“

От това съждение можем да получим следната информация:

– някоя мегапираня;

– зъб, загубен от тази мегапираня в река Ориноко.

На това съответства типът данни наредена двойка, чийто пръв елемент x е от тип „мегапираня“, а вторият — от тип „зъб, загубен от x в река Ориноко“.

На пръв поглед, разгледан като тип данни, кванторът за съществуване се интерпретира по същия начин както конюнкцията — като наредена двойка. Има обаче и една съществена разлика — типът на втория елемент на тази наредена двойка зависи от x и значи е зависим тип.

Няколко примера на хаскел

В следващите няколко примера най-напред ще дадем програмен код на хаскел, а след това ще коментираме как той може да се интерпретира като математическо доказателство.

Когато на хаскел дефинираме някакъв обект A, с инструкцията `:t` A можем да получим неговия тип. Навсякъде в дадените примери `Prelude` е командният показалец на интерпретатора — текстът след него се въвежда от нас. Текстът пък, който не е предшестван от команден показалец, се отпечатва от интерпретатора. Например в следния пример

```
Prelude> let x = 'q'
Prelude> :t x
x :: Char
```

на първия ред дефинираме променлива `x` със стойност `'q'`. На втория ред питаме интерпретатора какъв е типът на `x`. На третия ред интерпретаторът отговаря, че типът на `x` е `Char`.

Първият пример, с който ще илюстрираме съответствието между програми и доказателства, е следният:

```
Prelude> let composition(f,g) = h
Prelude|         where h(x) = g(f(x))
Prelude|
Prelude> :t composition
composition :: (t2 -> t1, t1 -> t) -> t2 -> t
```

От програмна гледна точка тук дефинираме функцията `composition`, която получава като аргумент наредена двойка от две функции `f` и `g` и връща като стойност тяхната композиция `h`. Ако превърнем типа на функцията `composition` в логическа формула ще получим формулата

$$(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow (\varphi \Rightarrow \chi)$$

в която φ, ψ и χ са формулите, отговарящи съответно на автоматичните типове `t2`, `t1` и `t`. Да забележим, че тази формула е тавтология. Функцията `f`, която `composition` получава като аргумент, е от тип `t2->t1`, т.е. $\varphi \Rightarrow \psi$, а функцията `g` — от тип `t1->t`, т.е. $\psi \Rightarrow \chi$. Стойността `h`, която `composition` трябва да върне е от тип $\varphi \Rightarrow \chi$. За да получим `h`, да допуснем, че притежаваме обект `x` от тип φ. Ако към `x` приложим `f`, ще получим обект от тип ψ. Ако към този обект приложим `g`, ще получим обект от тип χ. По този начин получаваме функция, която получава аргумент от тип φ и връща стойност от тип χ и значи типът на тази функция е $\varphi \Rightarrow \chi$.

Това е „програмистката“ интерпретация на случващото се във функцията `composition`. Да видим сега какво доказателство съответства на тази функция. Тъй като аргументът ѝ е от тип $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$, то доказателството започва с изречението „Да допуснем, че е вярно $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$“. В дефиницията на функцията `composition`, аргументът ѝ е наредената двойка `(f,g)`. Това е все едно в доказателството за удобство да означим $(\varphi \Rightarrow \psi)$ с `f` и $(\psi \Rightarrow \chi)$ с `g`. Тъй като `composition` връща като стойност функцията `h`, то доказателството продължава според дефиницията на функцията `h`. Аргументът `x` на `h` е от тип φ, и значи доказателството продължава с изречението „Да допуснем, че е вярно φ“. След това h връща `g(f(x))`. На това отговаря следното изречение „От φ и `f` получаваме ψ, а от тук и `g` получаваме χ“.

И така, на дадената по-горе програма отговаря следното доказателство:

Да допуснем, че $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$. Тогава

$$\varphi \Rightarrow \psi \qquad (\texttt{f})$$

и

$$\psi \Rightarrow \chi \qquad (\texttt{g})$$

Да допуснем, че е вярно φ. От φ и `f` получаваме ψ, а от ψ и `g` получаваме χ.

Допуснахме φ и доказахме χ. Значи е вярно

$$\varphi \Rightarrow \chi \qquad (\texttt{h})$$

Допуснахме $(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi)$ и доказахме $\varphi \Rightarrow \chi$, значи е вярно

$$(\varphi \Rightarrow \psi)\ \&\ (\psi \Rightarrow \chi) \Rightarrow \varphi \Rightarrow \chi \qquad (\texttt{composition})$$

Да разгледаме друг пример.

```
Prelude> let dis(x) = Left x
Prelude> :t dis
dis :: a -> Either a b
```

От програмна гледна точка тук дефинираме функцията `dis`, която получава обект от произволен тип `a` и връща като стойност обекта `Left(x)` от тип `Either a b`. Да си припомним, че на типа `Either a b` съответства формулата $\varphi \vee \psi$, където φ е формулата, съответстваща на типа `a`, а ψ е формулата, съответстваща на типа `b`. Следователно формулата, съответстваща на типа на функцията `dis` е

$$\varphi \Rightarrow \varphi \vee \psi$$

Да забележим, че тази формула е тавтология.

Извикването на функцията `dis` се свежда до извикването на конструктора `Left`. Типът `Either a b` има още един конструктор — `Right`, с чиято помощ можем да докажем и формулата $\psi \Rightarrow \varphi \vee \psi$.

Разгледаният пример с дизюнкция дава тривиално от математическа гледна точка доказателство. Ето един по-интересен пример, също използващ дизюнкция:

```
Prelude> let cases(f,g) = h
Prelude|         where
Prelude|             h(x) = case x of
Prelude|                   Left(y) -> f(y)
Prelude|                   Right(z) -> g(z)
Prelude|
Prelude> :t cases
cases :: (t1 -> t, t2 -> t) -> Either t1 t2 -> t
```

Преведен като формула, типът на функцията `cases` е

$$(\varphi \Rightarrow \chi) \,\&\, (\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi)$$

където φ, ψ и χ са формулите, съответстващи на автоматичните типове `t1`, `t2` и `t`. Функцията `h`, която функцията `cases` връща като стойност, има за аргумент обект `x` от тип `Either t1 t2`, т.е. $\varphi \vee \psi$. Тази функция разглежда два случая в зависимост от това дали `x` има вида `Left(y)`, където `y` е от тип φ, или `Right(z)`, където `y` е от тип ψ. В първия случай `h` връща като стойност `f(y)`, а във втория — `g(z)`.

Да видим как да интерпретираме това като математическо доказателство на формулата $(\varphi \Rightarrow \chi) \,\&\, (\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi)$. Щом като аргументът на `cases` е от тип $(\varphi \Rightarrow \chi) \,\&\, (\psi \Rightarrow \chi)$, то значи доказателството започва с думите „Да допуснем, че $(\varphi \Rightarrow \chi) \,\&\, (\psi \Rightarrow \chi)$“. Аргументът на `h` има вида `(f,g)`, което е все едно в математическото доказателство за краткост да означим $\varphi \Rightarrow \chi$ с `f` и $\psi \Rightarrow \chi$ с `g`. Функцията `cases` връща като стойност функцията h, така че доказателството продължава според дефиницията на `h`. Функцията `h` има аргумент от тип $\varphi \vee \psi$, което значи, че доказателството продължава с изречението „Да допуснем, че е вярно $\varphi \vee \psi$“. След това в `h` се разглеждат случаи според вида на аргумента на `h`. В математическото доказателство това отговаря на това да разгледаме случаи в зависимост от това дали е вярно φ или ψ. При първия случай пресмятаме `f(y)`, където `y` е от тип φ, а `f` е от тип $\varphi \Rightarrow \chi$. В математическото доказателство това отговаря на изречението: „Щом като φ е вярно, а според `f` от φ следва χ, то значи χ е вярно.“ Във втория случай пресмятаме `g(z)`, където `z` е от тип ψ, а `g` е от тип $\psi \Rightarrow \chi$. В математическото доказателство това отговаря на изречението: „Щом като ψ е вярно, а според `g` от ψ следва χ, то значи и χ е вярно.“

И така, на дадената по-горе програма отговаря следното доказателство:

Да допуснем, че $(\varphi \Rightarrow \chi) \,\&\, (\psi \Rightarrow \chi)$ Тогава

$$\varphi \Rightarrow \chi \qquad (\texttt{f})$$

и

$$\psi \Rightarrow \chi \qquad (\texttt{g})$$

Да допуснем, че

$$\varphi \vee \psi \qquad (\texttt{x})$$

и да разгледаме случаи в зависимост от това дали е вярно φ или ψ.

В първия случай, когато е вярно φ, от `f` получаваме χ.

Във втория случай, когато е вярно ψ, от `g` отново получаваме χ.

И в двата случая получихме χ.
Допуснахме $\varphi \vee \psi$ и доказахме χ, значи е вярно

$$\varphi \vee \psi \Rightarrow \chi \qquad (\texttt{h})$$

Допуснахме $(\varphi \Rightarrow \chi)\,\&\,(\psi \Rightarrow \chi)$ и доказахме $\varphi \vee \psi \Rightarrow \chi$, значи е вярно

$$(\varphi \Rightarrow \chi)\,\&\,(\psi \Rightarrow \chi) \Rightarrow (\varphi \vee \psi \Rightarrow \chi) \qquad (\texttt{cases})$$

3.7. Нормални форми

В много дялове на математиката се използват т.н. нормални форми. Когато всеки обект от някой тип може да бъде представен посредством израз, имащ определен вид, казваме, че този израз е нормалната форма на обекта. Когато искаме да дефинираме в компютърна програма тип, чиито обекти притежават нормална форма, не е нужно да се грижим за компютърно представяне на изрази, които не са в нормална форма.

Ето няколко примера.

– От дискретната математика знаем, че булевите функции могат да се представят в конюнктивна нормална форма, в дизюнктивна нормална форма и като полиноми.

– Всеки алгоритъм, обработващ естествени числа, може да се реализира посредством компютърна програма, имаща единствено оператори за присвояване, оператори за четене и запис на входни и изходни данни, само един оператор за цикъл и никакви други оператори. Това е известно като нормална форма на Клѝни.

– В алгебрата всяка матрица може да се представи в жорданова нормална форма.

– Представянето на всяко естествено число като произведение на прости числа също може да се разглежда като нормална форма.

– В ламбда-смятането се дефинира понятието редукция на ламбда-термове. Когато един ламбда-терм бъде доведен посредством редукция до ламбда-терм, който не може повече да се редуцира, казваме, че сме получили бета нормална форма. Бета нормалната форма е резултатът от пресмятането (следователно зациклящите се изчислителни процеси не притежават бета нормална форма).

В този раздел ще дефинираме няколко нормални форми за формули класическата предикатна логика от първи ред.* Нещата, които са верни

*Няма хубави нормални форми за интуиционистката предикатна логика.

само при класическата логика, но не и при интуиционистката логика, ще бъдат отбелязвани с **(клас)**.

3.7.1. Да изберем някоя затворена формула, която е невярна във всяка структура, и да я означим с $\bot$. (Такава формула винаги съществува. Нека φ е произволна затворена формула. Тогава нека $\bot$ бъде формулата $\varphi \,\&\, \neg\varphi$.)

3.7.2. Твърдение. *За произволна формула φ:*

а) $\vDash \neg\varphi \Leftrightarrow (\varphi \Rightarrow \bot)$

б) $\vDash \neg\neg\varphi \Leftrightarrow \varphi$ **(клас)**

Доказателство. **(а)** Да си припомним, че $\neg\varphi \Leftrightarrow (\varphi \Rightarrow \bot)$ е съкращение на формулата $(\neg\varphi \Rightarrow (\varphi \Rightarrow \bot)) \,\&\, ((\varphi \Rightarrow \bot) \Rightarrow \neg\varphi)$. Следователно трябва да докажем, че $\vDash \neg\varphi \Rightarrow (\varphi \Rightarrow \bot)$ и $\vDash (\varphi \Rightarrow \bot) \Rightarrow \neg\varphi$.

Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$. Формулата $\neg\varphi \Rightarrow (\varphi \Rightarrow \bot)$ е вярна в структурата $\mathbf{M}$ при оценка v тогава и само тогава, когато е вярно съждението

ако A е лъжа, то от A следва лъжа

Това съждение е очевидно вярно.

Формулата $(\varphi \Rightarrow \bot) \Rightarrow \neg\varphi$ е вярна в структурата $\mathbf{M}$ при оценка v тогава и само тогава, когато е вярно съждението

ако от A следва лъжа, то A е лъжа

Това съждение също е очевидно вярно.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$. Трябва да докажем, че съждението „не не A“ еквивалентно на A. При използване на класическата логика това е тривиално вярно. ∎

3.7.3. Твърдение. *За произволни формули φ и ψ:*

а) $\vDash \neg(\varphi \vee \psi) \Leftrightarrow \neg\varphi \,\&\, \neg\psi$

б) $\vDash \neg(\varphi \,\&\, \psi) \Leftrightarrow \neg\varphi \vee \neg\psi$* **(клас)**

в) $\vDash \neg(\varphi \Rightarrow \psi) \Leftrightarrow \varphi \,\&\, \neg\psi$** **(клас)**

*Интуиционистки е вярно $\vDash \neg(\varphi \,\&\, \psi) \Leftrightarrow \neg\neg(\neg\varphi \vee \neg\psi)$.

**Интуиционистки е вярно $\vDash \neg(\varphi \Rightarrow \psi) \Leftrightarrow \neg\neg\varphi \,\&\, \neg\psi$.

Доказателство. **(а)** Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, а B е съждението $\mathbf{M} \vDash \psi[v]$.

Най-напред да докажем, че $\mathbf{M} \vDash \neg(\varphi \vee \psi) \Rightarrow \neg\varphi \,\&\, \neg\psi[v]$. За целта да допуснем, че не е вярно съждението „A или B“. Трябва да докажем, че не е вярно A и не е вярно B. За да докажем, че не е вярно A, да допуснем, че A е вярно. Щом A е вярно, значи е вярно и съждението „A или B“. Но това е противоречие, значи A не е вярно. Аналогично се вижда, че и B не е вярно.

За да докажем, че $\mathbf{M} \vDash \neg\varphi \,\&\, \neg\psi \Rightarrow \neg(\varphi \vee \psi)[v]$, да допуснем, че не е вярно A и не е вярно B. Трябва да докажем, че не е вярна дизюнкцията „A или B“. За да докажем, че тази дизюнкция не е вярна, да допуснем, че тя е вярна и да разгледаме случаи в зависимост от това дали е вярно A или B. Когато е вярно A получаваме противоречие, защото знаем, че A не е вярно. Но когато е вярно B също получаваме противоречие, защото знаем, че B не е вярно. И в двата случая стигаме до противоречие. То се дължи на допускането, че дизюнкцията „A или B“ е вярна.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, а B е съждението $\mathbf{M} \vDash \psi[v]$.

Най-напред да докажем, че $\mathbf{M} \vDash \neg(\varphi \,\&\, \psi) \Rightarrow \neg\varphi \vee \neg\psi[v]$. За целта да допуснем, че не е вярно съждението „A и B“. Трябва да докажем, че A е невярно или B е невярно. От конструктивна гледна точка това означава, че ни е дадена функция, която получава аргументи от типове A и B и връща стойност от тип „противоречие“. Имайки само такава функция няма начин да направим функция, която получава само един аргумент от тип A и връща противоречие, нито пък функция, която получава само B и връща противоречие. Следователно няма как да конструираме обект от тип $\neg\varphi \vee \neg\psi$. Това ни подсказва, че ще трябва да допуснем противното.

И така, да допуснем, че не е вярно съждението „A не е вярно или B не е вярно“. Ако допуснем, че A е невярно, ще получим че е вярно съждението, за което току-що допуснахме, че е невярно. Значи съждението A не може да не е вярно. Аналогично се вижда, че и B не може да не е вярно. Това обаче противоречи на дизюнкцията, казваща, че A не е вярно или B не е вярно.

За да докажем обратната посока, да допуснем, че не е вярно A или не е вярно B. Трябва да докажем, че не е вярно съждението „A и B“. За целта да допуснем, че съждението „A и B“ е вярно. Значи съжденията A и B са верни, но това веднага ни дава противоречие с дизюнкцията, според която не е вярно A или не е вярно B. ∎

3.7.4. Твърдение. *За произволна формула* φ

а) $\vDash \neg\exists \mathtt{x}\, \varphi \Leftrightarrow \forall \mathtt{x}\, \neg\varphi$

б) $\vDash \neg\forall \mathtt{x}\, \varphi \Leftrightarrow \exists \mathtt{x}\, \neg\varphi$[*] **(клас)**

Доказателство. **(а)** Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. За произволен елемент μ на универсума на A, нека $A(\mu)$ е съждението $\mathbf{M} \vDash \varphi[v|\mathtt{x} := \mu]$.

Най-напред ще докажем, че $\mathbf{M} \vDash \neg\exists \mathtt{x}\, \varphi \Rightarrow \forall \mathtt{x}\, \neg\varphi[v]$. За целта да допуснем, че не съществува $\mu \in |\mathbf{M}|$, за което $A(\mu)$ е истина. Трябва да докажем, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е лъжа. Да изберем произволно $\mu \in |\mathbf{M}|$ и да допуснем, че $A(\mu)$ е истина. Вижда се, че това противоречи на допускането, че не съществува такова μ.

За да докажем обратната посока, да допуснем, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е лъжа. Трябва да докажем, че не съществува $\mu \in |\mathbf{M}|$, за което $A(\mu)$ е истина. Да допуснем, че такова μ съществува. Тогава за това μ съждението $A(\mu)$ ще бъде истина, което противоречи на допускането, че за всяко μ съждението $A(\mu)$ е лъжа.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. За произволен елемент μ на универсума на A, нека $A(\mu)$ е съждението $\mathbf{M} \vDash \varphi[v|\mathtt{x} := \mu]$.

Най-напред ще докажем, че $\mathbf{M} \vDash \neg\forall \mathtt{x}\, \varphi \Rightarrow \exists \mathtt{x}\, \neg\varphi[v]$. Да допуснем, че не е вярно, че за всеки елемент μ на универсума на $\mathbf{M}$ е вярно $A(\mu)$. От конструктивна гледна точка това означава, че ни е дадена функция, която ще ни върне обект от тип противоречие, ако ѝ дадем като аргумент функция, която за всяко μ връща обект от тип $A(\mu)$. Не се вижда как, разполагайки с такава функция, ще можем да намерим обект μ, за който не е вярно $A(\mu)$. Следователно съждението не може да се докаже конструктивно и значи трябва да допуснем противното.

И така, да допуснем, че не съществува $\mu \in |\mathbf{M}|$, за което не е вярно $A(\mu)$. Използвайки доказаното в а), можем да заключим, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е вярно.[**] Това е противоречие.

За да докажем обратната посока, да допуснем, че не съществува елемент μ на универсума на $\mathbf{M}$, за който е вярно съждението $A(\mu)$. Трябва да докажем, че за всеки елемент μ на универсума на $\mathbf{M}$, съждението $A(\mu)$ е невярно. Това е очевидно. ∎

[*] Интуиционистки е вярно $\vDash \neg\forall \mathtt{x}\, \varphi \Leftrightarrow \neg\neg\exists \mathtt{x}\, \neg\varphi$.

[**] Всъщност от доказаното в а) можем да заключим, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ не може да не е вярно. Използвайки класическата логика, от тук, разбира се, получаваме, че за всяко $\mu \in |\mathbf{M}|$ съждението $A(\mu)$ е вярно.

3.7.5. Твърдение. *За произволни формули φ и ψ:*

$$\vDash (\varphi \Rightarrow \psi) \Leftrightarrow (\neg\varphi \vee \psi)^* \qquad \textbf{(клас)}$$

Доказателство. Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, а B е съждението $\mathbf{M} \vDash \psi[v]$.

Най напред ще докажем, че $\mathbf{M} \vDash (\varphi \Rightarrow \psi) \Rightarrow (\neg\varphi \vee \psi)[v]$. Да допуснем, че от A следва B. Трябва да докажем, че A не е вярно или B е вярно. От конструктивна гледна точка това означава, че имаме функция, на която ако ѝ дадем аргумент от тип A, ще ни върне стойност от тип B. Не се вижда как имайки такава функция това ще ни позволи да установим кой от членовете на исканата дизюнкция е верен. Това ни подсказва, че желаното свойство не може да бъде доказано конструктивно и значи трябва да допуснем противното.

И така, да допуснем, че не е вярна дизюнкцията, казваща, че A е невярно или B е вярно. Това означава, че A е истина и B е лъжа (вж. доказателството на твърдение 3.7.3 а)). Но щом A е истина и от A следва B, значи B е истина. Стигнахме до противоречие.

За да докажем обратната посока, да допуснем, че A е невярно или B е вярно. Трябва да докажем, че от A следва B. За целта да допуснем, че A е вярно. Щом A е вярно, то значи дизюнкцията „A е невярно или B е вярно“ ни казва, че B е вярно. И така, допуснахме, че A е вярно и доказахме, че B е вярно. Следователно от A следва B. ∎

3.7.6. Преди много хилядолетия в град Лагаш, намиращ се в днешен Ирак, живял бог Нингирсу, който бил юначен ловец и воин. Веднъж, по време на един от ловните си походи, Нингирсу влязал в бой с многоглавия дракон Мушмаху. Всеки път когато Нингирсу отрязвал една от главите на Мушмаху, на него му пораствали много нови глави. Отначало Нингирсу решил, че работата е безнадеждна, но после забелязал, че всеки път когато отрязвал някоя глава, шиите на новопоникналите глави били най-малко с един сантиметър по-къси. Понеже разбирал от математика, Нингирсу съобразил, че ако не се отказва, със сигурност ще се стигне до момент, когато драконът ще остане без глави. След като победил, Нингирсу се отправил към град Нипур, където се намирал зигуратът Екур — жилището на боговете. Нингирсу бил изпълнен с боен плам, надавал викове, пеел песни, с които прославял геройството си, и извършвал разни вандалства с цел да му обръщат внимание. Боговете в Нипур се уплашили и му казали, че ако се поуспокои малко, ще

*Интуиционистки не е възможно да изразим импликацията посредством останалите съждителни операции. Вярно е обаче следното: $\vDash \neg\neg(\varphi \Rightarrow \psi) \Leftrightarrow \neg\neg(\neg\varphi \vee \psi)$.

го възнаградят. Като стигнал Екур, той показал на удивените богове бойните си трофеи.

Логиците използват т.н. *ординални числа* с цел да „измерят“ колко е сложно дадено разсъждение, използващо индукция. Ако използваме обикновената индукция за естествени числа по възможно най-естествения начин, казваме, че сме използвали индукция до ординала ω. Ако обаче докато правим индукционната стъпка, вътре в нея за втори път започваме да правим индукция, казваме че сме използвали индукция до ординала ω^2. Ако пък и в индукционната стъпка на втората индукция пак правим индукция, тогава използваме индукция до ω^3. Според „великото предположение“ на Харви Фридман, всяка теорема, публикувана в математическо списание като „Annals of Mathematics“, чиято формулировка използва само крайни обекти, може да се докаже с индукция до ω^3. Въпреки това, оказва се, че само с индукция до ω^3 не можем да докажем, че драконът Мушмаху ще бъде победен. За целта ни е нужна индукция до $\omega^\omega = \sup\{\omega, \omega^2, \omega^3, \omega^4, \dots\}$.

Въпреки че драконът Мушмаху бил убит, той си оставил достоен потомък — Лернейската хидра. Също както и при Мушмаху, и на Лернейската хидра за всяка отсечена глава ѝ порастват нови глави. Новите глави на Хидрата обаче растяли по малко по-различни правила, така че понякога се случвало шиите на новите глави да не са по-къси от шията на току-що отсечената глава. Хидрата била убита от Херкулес, защото отново се оказало, че без значение как ѝ режем главите, след краен брой стъпки тя ще остане без глави. Това обаче е изключително трудно да се докаже, защото изисква индукция до ординала $\varepsilon_0 = \sup\{\omega, \omega^\omega, \omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \dots\}$.* Оказва се, че в стандартната аксиоматична теория на аритметиката — Аритметиката на Пеано — е невъзможно да правим толкова сложни индукции и затова в тази теория не може да се докаже, че Хидрата ще бъде убита от Херкулес.

В следващата дефиниция да си мислим, че редицата от числа $(a_n)_{n=1}^\infty$ измерва броя на главите на дракона Мушмаху. Числото a_1 е равно на броя на главите, чиято шия е дълга 1 сантиметър, a_2 е равно на броя на главите, чиято шия е равна на 2 сантиметра и т.н.

3.7.7. Дефиниция. Нека $(a_n)_{n=1}^\infty$ е редица от естествени числа и редицата $(b_n)_{n=1}^\infty$ се получава като извършим следните промени в редицата $(a_n)_{n=1}^\infty$:

*Всеки може сам да се постави на мястото на Херкулес и да се опита да победи Лернейската хидра, използвайки джава аплет от адрес `http://math.andrej.com/2008/02/02/the-hydra-game/`.

– избираме такова естествено число i, че $a_i \neq 0$;

– намаляваме a_i;

– заменяме $a_1, a_2, \ldots, a_{i-1}$ с произволни естествени числа.

В такъв случай казваме, че редицата $(a_n)_{n=1}^{\infty}$ се *конвертира* в редицата $(b_n)_{n=1}^{\infty}$. Също ще казваме, че редицата $(b_n)_{n=1}^{\infty}$ се получава от редицата $(a_n)_{n=1}^{\infty}$ посредством *конверсия*. Числото i ще наричаме *ранг на конверсията*.

3.7.8. Лема за фундираност на конверсията. *Нека $(a_n)_{n=1}^{\infty}$ е редица от естествени числа, в която има само краен брой ненулеви елементи. Да разгледаме следния процес: прилагаме конверсия към редицата $(a_n)_{n=1}^{\infty}$, после прилагаме конверсия към новополучената редица, след това отново прилагаме конверсия и т.н. Без значение какви точно конверсии извършваме, със сигурност след краен брой стъпки ще стигнем до редица, съдържаща само нули.*

Доказателство. Да забележим, че не е възможно да прилагаме безброй много конверсии, чийто ранг е 1. Наистина, след всяка такава конверсия първият член на редицата намалява, а това не може да се случва до безкрайност. Това означава, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне 2.

Да забележим също, че не е възможно да прилагаме безброй много конверсии, чийто ранг е 1 или 2. Наистина, вече видяхме, че не можем да прилагаме безброй конверсии само от ранг 1, следователно ако прилагаме само конверсии от ранг 1 или 2, ще трябва да прилагаме безброй пъти конверсии от ранг 2. Това обаче е невъзможно, защото след всяка конверсия от ранг 2 вторият член на редицата намалява, а това не може да се случва до безкрайност (да забележим, че конверсиите от ранг 1 не променят втория член на редицата). Това означава, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне 3.

Обаче не е възможно също да прилагаме безброй много конверсии, чийто ранг е 1, 2 или 3. Наистина, вече видяхме, че не можем да прилагаме безброй конверсии само от рангове 1 или 2, следователно ако прилагаме само конверсии от ранг 1, 2 или 3, ще трябва да прилагаме безброй пъти конверсии от ранг 3. Това обаче е невъзможно, защото след всяка конверсия от ранг 3 третият член на редицата намалява, а това не може да се случва до безкрайност (да забележим, че конверсиите от рангове 1 или 2 не променят третия член на редицата). Това

означава, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне 4.

Разсъждавайки по подобен начин, можем да стигнем до извода, че както и да прилагаме конверсиите, след краен брой стъпки или ще стигнем до редица, съдържаща само нули, или ще стигнем до конверсия от ранг поне k, където k е произволно предварително избрано естествено число. Нека числото k е такова, че всички ненулеви членове в $(a_n)_{n=1}^{\infty}$ са измежду $a_1, a_2, \ldots, a_{k-1}$. В такъв случай няма да бъде възможна конверсия от ранг k или по-голям, следователно както и да прилагаме конверсиите, със сигурност ще стигнем до редица, съдържаща само нули. ■

3.7.9. Дефиниция. Казваме, че дадена формула е в *отрицателна нормална форма*, ако:

- във формулата няма други логически операции, освен конюнкция ($\&$), дизюнкция ($\vee$), отрицание ($\neg$) и кванторите за всеобщност ($\forall$) и съществуване ($\exists$);
- всички отрицания във формулата се намират пред атомарни подформули.

3.7.10. Твърдение. *За всяка формула може да се намери еквивалентна на нея (според класическата логика) формула, която е в отрицателна нормална форма.*

Доказателство. Нека φ е произволна формула. Най-напред да елиминираме импликациите във φ по следния начин: да заменяме докато може всяка подформула във φ, имаща вида $\psi' \Rightarrow \psi''$, с формулата $\neg\psi' \vee \psi''$. Съгласно твърдение 3.7.5 след всяка такава замяна получаваме еквивалентна формула. Тъй като след всяка замяна броят на импликациите намалява, след краен брой стъпки ще получим формула без импликации, което значи, че във формулата няма други логически операции, освен конюнкция ($\&$), дизюнкция ($\vee$), отрицание ($\neg$) и кванторите за всеобщност ($\forall$) и съществуване ($\exists$).

Остава да „преместим“ отрицанията пред атомарни формули. За

целта да прилагаме докато може замени от следния вид:

$$\neg\neg\psi \longmapsto \psi \tag{19}$$
$$\neg(\psi' \vee \psi'') \longmapsto \neg\psi' \,\&\, \neg\psi'' \tag{20}$$
$$\neg(\psi' \,\&\, \psi'') \longmapsto \neg\psi' \vee \neg\psi'' \tag{21}$$
$$\neg\exists \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, \neg\psi \tag{22}$$
$$\neg\forall \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, \neg\psi \tag{23}$$

Съгласно твърдения 3.7.2 б), 3.7.3 а), 3.7.3 б), 3.7.4 а) и 3.7.4 б), след всяка такава замяна получаваме еквивалентна формула. Ако след краен брой стъпки получим формула, към която не можем да приложим никоя от тези замени, то това означава, че сме стигнали до формула в отрицателна нормална форма. Това ни дава т.н. частична коректност на алгоритъма за привеждане в отрицателна нормална форма. Остава да видим защо този алгоритъм никога не се зацикля, т.е. трябва да докажем, че можем да прилагаме замени от горния вид само краен брой пъти. Това ще направим по два начина.

(I начин) Ще използваме фундираността на конверсията (лема 3.7.8).

Нека χ е произволна формула. *Височина* на χ ще наричаме дължината на най-дългата редица от вида

$$\chi_1, \chi_2, \chi_3, \ldots, \chi_k$$

където $\chi_1 = \chi$, $\chi_i \neq \chi_{i+1}$ и χ_{i+1} е подформула на χ_i. Ако си мислим формулата като дърво, тогава височината ѝ е равна на дължината на най-дългия клон в това дърво.

За всяка формула φ ще дефинираме редица $(a_n)_{n=1}^{\infty}$, която ще наречем *редица на* φ, по следния начин: нека a_i е равно на броя на подформулите на φ, които започват с отрицание и са с височина точно i.

Може да се забележи, че ако формулата φ' се получава от φ посредством някоя от замените (19)–(23), тогава редицата на φ' може да се получи от редицата на φ посредством конверсия. Съгласно лемата за фундираност на конверсията, след краен брой стъпки или ще стигнем формула, към която не можем да приложим никоя от замените (19)–(23), или ще стигнем формула, чиято редица се състои само от нули. Последното би означавало, че във формулата няма нито едно отрицание, но в този случай също е невъзможно да приложим замените (19)–(23).

(II начин) Да разгледаме следния начин, по който от формула получаваме аритметичен израз. Да заменим всяка атомарна формула с числото две. Да заменим всяка подформула от вида $\psi' \,\&\, \psi''$, $\psi' \vee \psi''$ и $\psi' \Rightarrow \psi''$ с $x + y$, където x и y са изразите, получени съответно от ψ' и ψ''. Да заменим всяка подформула от вида $\neg\psi$ с x^2, където x е изразът, получен от ψ. Да заменим всяка подформула от вида $\forall \mathtt{x}\, \psi$ и $\exists \mathtt{x}\, \psi$ с $1 + x$, където x е изразът, получен от ψ. Може да се забележи, че така полученият израз е естествено число, не по-малко от 2. Тъй като за всеки естествени числа по-големи или равни на 2 е изпълнено $(x^2)^2 > x$, $(x + y)^2 > x^2 + y^2$ и $(1 + x)^2 > 1 + x^2$, то стойността на получения израз ще намалява след всяка замяна от горния вид, а това не може да продължи до безкрайност. ∎

3.7.11. ***Забележка:*** Когато използваме първия начин за да докажем, че алгоритъмът за привеждане в отрицателна нормална форма спира, не е нужно да се използва толкова сложно твърдение като фундираността на конверсията. Нека редицата, съответстваща на някоя формула е $(a_n)_{n=1}^{\infty}$ и да разгледаме числото

$$\sum_{i=1}^{\infty} a_i 3^i$$

(сумата е коректна, защото само краен брой от числата a_i са ненулеви). Докато при дракона Мушмаху когато някоя негова глава бъде отсечена, може да му пораснат произволен брой нови глави, то тук, когато вкараме някое отрицание „навътре“ във формулата, то се заменя най-много с две „по-малки“ отрицания. Това означава, че след всяка замяна от вида (19)–(23), така дефинираното число ще намалява, а това не може да продължи до безкрайност.

3.7.12. Твърдение. *За произволни формули φ и ψ, ако* x *не е свободна променлива на φ, то:*

а) $\vDash \varphi \,\&\, \exists \mathtt{x}\, \psi \Leftrightarrow \exists \mathtt{x}\, (\varphi \,\&\, \psi)$ *и* $\vDash \exists \mathtt{x}\, \psi \,\&\, \varphi \Leftrightarrow \exists \mathtt{x}\, (\psi \,\&\, \varphi)$*;*

б) $\vDash \varphi \,\&\, \forall \mathtt{x}\, \psi \Leftrightarrow \forall \mathtt{x}\, (\varphi \,\&\, \psi)$ *и* $\vDash \forall \mathtt{x}\, \psi \,\&\, \varphi \Leftrightarrow \forall \mathtt{x}\, (\psi \,\&\, \varphi)$*;*

в) $\vDash \varphi \vee \exists \mathtt{x}\, \psi \Leftrightarrow \exists \mathtt{x}\, (\varphi \vee \psi)$ *и* $\vDash \exists \mathtt{x}\, \psi \vee \varphi \Leftrightarrow \exists \mathtt{x}\, (\psi \vee \varphi)$*;*

г) $\vDash \varphi \vee \forall \mathtt{x}\, \psi \Leftrightarrow \forall \mathtt{x}\, (\varphi \vee \psi)$ *и* $\vDash \forall \mathtt{x}\, \psi \vee \varphi \Leftrightarrow \forall \mathtt{x}\, (\psi \vee \varphi)$. **(клас)**

Доказателство. **(а)** Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \vDash \psi[v|\mathtt{x} := \mu]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемнт μ на универсума

на $\mathbf{M}$, съждението $\mathbf{M} \vDash \varphi[v|\mathtt{x} := \mu]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \vDash \varphi \,\&\, \exists \mathtt{x}\, \psi \Rightarrow \exists \mathtt{x}\, (\varphi \,\&\, \psi)[v]$. Да допуснем, че съждението A е вярно и съществува такова $\mu \in |\mathbf{M}|$, че съждението $B(\mu)$ е вярно. В такъв случай очевидно съществува такова $\mu \in |\mathbf{M}|$, че съждението „A и $B(\mu)$“ е вярно.

За да докажем обратната посока, да допуснем, че съществува такова $\mu \in |\mathbf{M}|$, че съждението „A и $B(\mu)$“ е вярно. В такъв случай очевидно съждението A ще бъде вярно и освен това ще съществува $\mu \in |\mathbf{M}|$, за което съждението $B(\mu)$ е вярно.

(б) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \vDash \psi[v|\mathtt{x} := \mu]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемнт μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \vDash \varphi[v|\mathtt{x} := \mu]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \vDash \varphi \,\&\, \forall \mathtt{x}\, \psi \Rightarrow \forall \mathtt{x}\, (\varphi \,\&\, \psi)[v]$. Да допуснем, че е вярно A и че за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. В такъв случай очевидно за всяко $\mu \in |\mathbf{M}|$ ще бъде вярно съждението „A и $B(\mu)$“.

За да докажем обратната посока, да допуснем, че за всяко $\mu \in |\mathbf{M}|$ е вярно съждението „A и $B(\mu)$“. Тъй като универсумът на $\mathbf{M}$ е непразен, като приложим току-що допуснатото за някой елемент на универсума, ще получим, че съждението A е вярно. Освен това очевидно за всяко $\mu \in |\mathbf{M}|$ ще бъде вярно съждението $B(\mu)$.

(в) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \vDash \psi[v|\mathtt{x} := \mu]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемнт μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \vDash \varphi[v|\mathtt{x} := \mu]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \vDash \varphi \vee \exists \mathtt{x}\, \psi \Rightarrow \exists \mathtt{x}\, (\varphi \vee \psi)[v]$. Да допуснем, че е вярно A или за някое $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. В първия случай, когато е вярно A, ще съществува $\mu \in |\mathbf{M}|$, за което е вярно съждението „A или $B(\mu)$“, защото кой да е елемент μ на универсума ще ни свърши работа (а такъв има, защото универсъмът е непразно множество). Във втория случай, когато съществува $\mu \in |\mathbf{M}|$, за което е вярно $B(\mu)$, очевидно също ще съществува $\mu \in |\mathbf{M}|$, за което е вярно съждението „A или $B(\mu)$“.

За да докажем обратната посока, да допуснем, че съществува $\mu \in |\mathbf{M}|$, за което е вярно съждението „A или $B(\mu)$“. Ако за това μ, е вярно A, то A е вярно (това съждение не зависи от μ). Ако пък за съществуващото μ е вярно $B(\mu)$, то значи съществува $\mu \in |\mathbf{M}|$, за което е вярно $B(\mu)$. Следователно A е вярно или съществува $\mu \in |\mathbf{M}|$, за което е вярно $B(\mu)$.

(г) Да изберем произволни структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$ и за произволен елемент μ на универсума на $\mathbf{M}$, нека $B(\mu)$ е съждението $\mathbf{M} \vDash \psi[v|\mathtt{x} := \mu]$. Тъй като $\mathtt{x}$ не е свободна променлива на φ, то за всеки елемент μ на универсума на $\mathbf{M}$, съждението $\mathbf{M} \vDash \varphi[v|\mathtt{x} := \mu]$ е вярно тогава и само тогава, когато е вярно съждението A (което очевидно не зависи от μ).

Най-напред ще докажем, че $\mathbf{M} \vDash \varphi \vee \forall \mathtt{x}\, \psi \Rightarrow \forall \mathtt{x}\, (\varphi \vee \psi)[v]$. За целта да допуснем, че е вярно A или за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. В първия случай очевидно за всяко $\mu \in |\mathbf{M}|$ ще бъде вярно съждението „A или $B(\mu)$“. Във втория случай — също.

За да докажем обратната посока, да допуснем, че за всяко $\mu \in |\mathbf{M}|$ е вярно съждението „A или $B(\mu)$“. От конструктивна гледна точка това означава, че ни е дадена функция, която по дадено μ ни казва дали е вярно A или $B(\mu)$. Ако решим „да пробваме“ тази функция за различни елементи на универсума и всеки път получаваме, че е вярно $B(\mu)$, това няма да означава, че $B(\mu)$ е вярно винаги — възможно е просто да не сме имали късмет и ако бяхме улучили „правилното“ μ, щяхме да установим, че е вярно A. Следователно няма как имайки такава функция да разберем дали е вярно A, или за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. Това ни подсказва, че съждението не е вярно конструктивно и значи трябва да допуснем противното.

И така, да допуснем, че не е вярно съждението „A е вярно или за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$“. Това означава (вж. доказателството на съждение 3.7.3 а)), че съждението A не е вярно и съждението „за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$“ също не е вярно. Но ние знаем, че за всяко $\mu \in |\mathbf{M}|$ е вярно „A или $B(\mu)$“. Тъй като току-що установихме, че A не е вярно, то значи за всяко $\mu \in |\mathbf{M}|$ е вярно $B(\mu)$. Това е противоречие. ■

3.7.13. Твърдение. *За произволни формули φ и ψ, ако $\mathtt{x}$ не е свободна променлива на φ, то:*

а) $\vDash (\varphi \Rightarrow \forall \mathtt{x}\, \psi) \Leftrightarrow \forall \mathtt{x}\, (\varphi \Rightarrow \psi)$;

б) $\vDash (\varphi \Rightarrow \exists \mathtt{x}\, \psi) \Leftrightarrow \exists \mathtt{x}\, (\varphi \Rightarrow \psi)$; **(клас)**

в) $\vDash (\exists \mathtt{x}\, \psi \Rightarrow \varphi) \Leftrightarrow \forall \mathtt{x}\, (\psi \Rightarrow \varphi)$;

г) $\vDash (\forall \mathtt{x}\, \psi \Rightarrow \varphi) \Leftrightarrow \exists \mathtt{x}\, (\psi \Rightarrow \varphi)$. **(клас)**

3.7.14. Дефиниция. Казваме, че една формула е в *пренексна нормална форма*, ако тя има вида

$$\mathfrak{O}_1 \mathtt{x}_1 \mathfrak{O}_2 \mathtt{x}_2 \ldots \mathfrak{O}_n \mathtt{x}_n (\varphi)$$

където $\mathfrak{O}_1, \mathfrak{O}_2, \ldots, \mathfrak{O}_n$ са квантори ($\forall$ или $\exists$), а формулата φ не съдържа квантори.

3.7.15. Твърдение. *За всяка формула може да се намери еквивалентна на нея (според класическата логика) формула, която е в пренексна нормална форма.*

Доказателство. Нека ни е дадена произволна формула. Съгласно лема 2.7.12 може да намерим конгруентна, а значи и еквивалентна на нея формула, в която всички квантори имат различни променливи и никоя кванторна променлива не е свободна променлива във формулата. Да прилагаме в така получената формула докато може замени от следния вид:

$$\begin{gathered}
\varphi \,\&\, \exists \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, (\varphi \,\&\, \psi) \\
\exists \mathtt{x}\, \psi \,\&\, \varphi \longmapsto \exists \mathtt{x}\, (\psi \,\&\, \varphi) \\
\varphi \,\&\, \forall \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, (\varphi \,\&\, \psi) \\
\forall \mathtt{x}\, \psi \,\&\, \varphi \longmapsto \forall \mathtt{x}\, (\psi \,\&\, \varphi) \\
\varphi \vee \exists \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, (\varphi \vee \psi) \\
\exists \mathtt{x}\, \psi \vee \varphi \longmapsto \exists \mathtt{x}\, (\psi \vee \varphi) \\
\varphi \vee \forall \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, (\varphi \vee \psi) \\
\forall \mathtt{x}\, \psi \vee \varphi \longmapsto \forall \mathtt{x}\, (\psi \vee \varphi) \\
\varphi \Rightarrow \forall \mathtt{x}\, \psi \longmapsto \forall \mathtt{x}\, (\varphi \Rightarrow \psi) \\
\varphi \Rightarrow \exists \mathtt{x}\, \psi \longmapsto \exists \mathtt{x}\, (\varphi \Rightarrow \psi)
\end{gathered}$$

$$\begin{gathered}
\exists \mathtt{x}\, \psi \Rightarrow \varphi \longmapsto \forall \mathtt{x}\, (\psi \Rightarrow \varphi) \\
\forall \mathtt{x}\, \psi \Rightarrow \varphi \longmapsto \exists \mathtt{x}\, (\psi \Rightarrow \varphi) \\
\neg \exists \mathtt{x}\, \varphi \longmapsto \forall \mathtt{x}\, \neg \varphi \\
\neg \forall \mathtt{x}\, \varphi \longmapsto \exists \mathtt{x}\, \neg \varphi
\end{gathered}$$

Ако имаме подформула от вида напр. $\varphi \,\&\, \exists \mathtt{x}\, \psi$, то във φ променливата $\mathtt{x}$ няма да бъде свободна, защото ако $\mathtt{x}$ бе свободна във φ, то тъй като

работим с формула, в която няма два квантора с една и съща променлива, то променливата $\mathtt{x}$ би била свободна и в цялата формула, а пък си осигурихме да няма кванторна променлива, която да е свободна в цялата формула. Това означава, че съгласно твърдения 3.7.12 б), 3.7.12 а), 3.7.12 г), 3.7.12 в), 3.7.13 а), 3.7.13 б), 3.7.13 в), 3.7.13 г), 3.7.4 а) и 3.7.4 б) при всяка една от тези замени ще получаваме еквивалентни формули. Ако след краен брой стъпки стигнем формула, към която не може да прилагаме никоя от по-горните замени, то значи сме получили формула в пренексна нормална форма. Това ни дава частичната коректност на алгоритъма за привеждане в пренексна нормална форма. Остава да докажем, че алгоритъмът винаги спира.

(I начин) За произволна формула χ да наречем *дълбочина* на подформулата χ' на χ дължината на най-дългата редица от вида

$$\chi_1, \chi_2, \chi_3, \ldots, \chi_k$$

където $\chi_1 = \chi$, $\chi_k = \chi'$, $\chi_i \neq \chi_{i+1}$, χ_{i+1} е подформула на χ_i и формулите $\chi_1, \chi_2, \ldots, \chi_{k-1}$ не започват с квантор. Ако си мислим формулата като дърво, тогава дълбочината на една подформула е равна на разстоянието между корена на поддървото и корена на цялото дърво, като при това пропускаме възлите от дървото, в които стои квантор.

Да забележим, че замените от горния вид не променят броя на подформулите, започващи с квантор. Всяка една такава замяна обаче намалява дълбочината на някоя от подформулите, започващи с квантор, а това не може да продължи до безкрайност.

(II начин) Да разгледаме следния начин, по който от формула получаваме аритметичен израз. Да заменим всяка атомарна подформула с числото 2. Да заменим всяка подформула от вида $\chi' \,\&\, \chi'$, $\chi' \vee \chi''$ и $\chi' \Rightarrow \chi''$ с $x + y$, където x и y са изразите, получени съответно от χ' и χ''. Да заменим всяка подформула от вида $\neg\chi$ с $1 + x$, където x е изразът, получен от χ. Да заменим всяка подформула от вида $\forall\mathtt{x}\,\chi$ и $\exists\mathtt{x}\,\chi$ с x^2, където x е изразът, получен от χ. Може да се забележи, че така полученият израз е естествено число, не по-малко от 2. Тъй като за всеки естествени числа, не по-малки от 2, е изпълнено $x + y^2 < (x + y)^2$, $x^2 + y < (x + y)^2$ и $1 + x^2 < (1 + x)^2$, то след всяко преобразование на формулата по описание по-горе начин, стойността на съответния аритметичен израз ще се увеличи. Това обаче не може да продължи до безкрай, защото броят на символите в тези аритметични изрази се запазва след всяко такова преобразование и значи може да се получат само краен брой аритметични изрази. ∎

3.7.16. ***Забележка*:** Въпреки че правилата за замяна в алгоритъма за привеждане в пренексна нормална форма са много на брой, те се помнят лесно. Нека да забележим, че когато кванторът излиза пред отрицание или се е намирал отляво на импликация, той се променя, т.е. от $\forall$ става $\exists$ и от $\exists$ става $\forall$. Във всички останали случаи кванторът просто се премества без промяна.

3.7.17. Твърдение. *За всяка формула може да се намери еквивалентна на нея (според класическата логика) формула, която е едновременно в пренексна нормална форма и в отрицателна нормална форма.*

Доказателство. Непосредствено се проверява, че ако преобразуваме според алгоритъма за привеждане в пренексна нормална форма формула, която се намира в отрицателна нормална форма, то ще получаваме формули в отрицателна нормална форма. Това означава, че можем просто да приведем формулата в отрицателна нормална форма и след това да приложим алгоритъмът за привеждане в пренексна нормална форма.

Също така можем непосредствено да проверим, че ако преобразуваме според алгоритъма за привеждане в отрицателна нормална форма формула, която се намира в пренексна нормална форма, то ще получаваме формули в пренексна нормална форма. Това означава, че можем просто да приведем формулата в пренексна нормална форма и след това да приложим алгоритъмът за привеждане в отрицателна нормална форма.

Също така, напълно допустимо е да прилагаме разбъркано замените от двата алгоритъма. В този случай процесът също със сигурност ще бъде краен и ще даде желания резултат, но тук няма да доказваме това. ■

3.7.18. ***Забележка*:** Ако формулата вече е приведена в отрицателна нормална форма и приложим към нея алгоритъма за привеждане в пренексна нормална форма, няма да ни се наложи да прилагаме нито веднъж замени, при които кванторът се променя, т.е. от $\forall$ става $\exists$ или от $\exists$ става $\forall$. Това означава, че не е нужно да работим стъпка по стъпка, а можем просто да преместим наведнъж всички квантори отпред на формулата (обаче без да променяме реда им!) и ще получим пренексна нормална форма.

3.7.19. Твърдение. *Формула от вида $\forall \mathtt{x}\,(\varphi)$ е тъждествено вярна в някоя структура $\mathbf{M}$ тогава и само тогава, когато в $\mathbf{M}$ е тъждествено вярна формулата φ.*

Доказателство. Да допуснем, че $\mathbf{M} \vDash \forall \mathtt{x}\,(\varphi)$, т.е. за всяка оценка v в $\mathbf{M}$ е вярно $\mathbf{M} \vDash \forall \mathtt{x}\,(\varphi)[v]$. По дефиниция това означава, че за всяка оценка v в $\mathbf{M}$ и за всеки елемент μ на универсума на $\mathbf{M}$ е вярно $\mathbf{M} \vDash \varphi[v|\mathtt{x} \coloneqq \mu]$. В частност, ако изберем $\mu = v(\mathtt{x})$, оценката $v|\mathtt{x} \coloneqq \mu$ ще съвпадне с v и значи за всяка оценка v в $\mathbf{M}$ ще бъде вярно $\mathbf{M} \vDash \varphi[v]$. Следователно φ е тъждествено вярна в $\mathbf{M}$.

Обратната посока се вижда още по-лесно. Да допуснем, че φ е тъждествено вярна в $\mathbf{M}$. Това значи, че φ ще бъде вярна при произволна оценка. В частност φ ще бъде вярна и при всяка модифицирана оценка $(v|\mathtt{x} \coloneqq \mu)$, и значи формулата $\forall \mathtt{x}\,\varphi$ е тъждествено вярна в $\mathbf{M}$. ∎

Прилагателното „скулемов“ от следващата дефиниция произлиза от името на откривателя на преобразованието, наречено *скулемизация* — норвежкия логик Тӯралф Скӯлем.

При скулемизацията ще използваме субституции, които променят една единствена променлива. За удобство ще използваме следното означение (подобно на означението, което имаме за оценки): ако $\mathtt{x}$ е променлива, а τ — терм, то с $\mathtt{x} \coloneqq \tau$ ще означаваме субституцията, която заменя $\mathtt{x}$ с τ и оставя всички останали променливи непроменени.

3.7.20. Дефиниция. а) Нека е дадена формула от вида $\exists \mathtt{x}\,(\varphi)$, в която няма свободни променливи, а $\mathtt{c}$ е символ за константа. Формулата $\varphi[\mathtt{x} \coloneqq \mathtt{c}]$ се нарича *скулемово усилване* (от първи вид) на $\exists \mathtt{x}\,(\varphi)$.

б) Нека е дадена формула от вида $\exists \mathtt{x}\,(\varphi)$, чиито свободни променливи са $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$, а $\mathtt{f}$ е n-местен функционален символ. Формулата $\varphi[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)]$ се нарича *скулемово усилване* (от втори вид) на $\exists \mathtt{x}\,(\varphi)$.

в) Символът за константа $\mathtt{c}$ от а) и функционалният символ $\mathtt{f}$ от б) се наричат *скулемов символ за константа* и *скулемов функционален символ*.

3.7.21. Пример. Да разгледаме формулата $\varphi = \exists \mathtt{x}\,\mathtt{p}(\mathtt{x})$. Едно нейно скулемово усилване е формулата $\varphi' = \mathtt{p}(\mathtt{c})$. Във всяка структура $\mathbf{M}$ формулата φ „казва“, че в универсума на $\mathbf{M}$ има елемент, за който предикатът $\mathtt{p}^{\mathbf{M}}$ е истина. Формулата φ' пък казва, че предикатът $\mathtt{p}^{\mathbf{M}}$ е истина не за някой неопределен елемент, а за $\mathtt{c}^{\mathbf{M}}$. Следователно скулемовото усилване φ' казва повече, отколкото φ и ако формулата φ' е вярна в някоя структура, то и φ ще бъде вярна. По-долу твърдение 3.7.23 ще покаже, че това се случва винаги, а не само при този конкретен пример.

Ако имаме право сами да решим каква да бъде интерпретацията на символа $\mathtt{c}$ и формулата φ е вярна в $\mathbf{M}$, то ще можем да интерпретираме $\mathtt{c}$ така, че $\mathtt{c}^{\mathbf{M}}$ да бъде оня елемент от универсума, чието съществуване се твърди от формулата φ. Ако променим $\mathbf{M}$ по този начин, формулата φ' ще стана вярна. По-долу твърдение 3.7.24 ще покаже, че това се случва винаги при скулемово усилване от първи вид, а не само при този конкретен пример.

3.7.22. Пример. Да разгледаме формулата $\varphi = \exists \mathtt{y}\, \mathtt{p}(\mathtt{x}, \mathtt{y})$. Едно нейно скулемово усилване е формулата $\varphi' = \mathtt{p}(\mathtt{x}, \mathtt{f}(\mathtt{x}))$. Във всяка структура $\mathbf{M}$ формулата φ „казва“, че за всеки елемент μ на универсума на $\mathbf{M}$ съществува такъв елемент ν, че $\mathtt{p}^{\mathbf{M}}(\mu, \nu)$ е истина. Формулата φ' пък казва, че $\mathtt{p}^{\mathbf{M}}(\mu, \nu)$ е истина не за някое неопределено ν, а за $\nu = \mathtt{f}^{\mathbf{M}}(\mu)$. Следователно скулемовото усилване φ' казва повече, отколкото φ и ако формулата φ' е вярна в някоя структура, то и φ ще бъде вярна. По-долу твърдение 3.7.23 ще покаже, че това се случва винаги, а не само при този конкретен пример.

Ако имаме право сами да решим каква да бъде интерпретацията на символа $\mathtt{f}$ и формулата φ е вярна в $\mathbf{M}$, то ще можем да интерпретираме $\mathtt{f}$ така, че $\mathtt{f}^{\mathbf{M}}(\mu)$ да бъде оня елемент ν от универсума, чието съществуване се твърди от формулата φ. Ако променим $\mathbf{M}$ по този начин, формулата φ' ще стана вярна. По-долу твърдение 3.7.24 ще покаже, че това се случва винаги при скулемово усилване от втори вид, а не само при този конкретен пример.

Да припомним твърдение 3.3.18, съгласно което за произволни субституция s и оценка v в структура $\mathbf{M}$ съществува такава оценка w, че за всяка променлива $\mathtt{z}$

$$w(z) = [\![s(\mathtt{z})]\!]^{\mathbf{M}}(v)$$

и за произволна формула φ

$$\mathbf{M} \vDash \varphi[w] \longleftrightarrow \mathbf{M} \vDash \varphi[s][v] \tag{24}$$

Нека свободните променливи на φ са $\mathtt{x}_1, \ldots, \mathtt{x}_n$ и оценката s е $\mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq \tau_1, \ldots, \tau_n$. Тогава може да преформулираме (24) по следния начин:

$$\mathbf{M} \vDash \varphi[[\![\tau_1]\!]^{\mathbf{M}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{M}}(v)] \longleftrightarrow \mathbf{M} \vDash \varphi[\mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq \tau_1, \ldots, \tau_n][v] \tag{25}$$

Следващото твърдение обяснява защо скулемовото усилване е наречено „усилване“ — скулемовото усилване на една формула винаги казва повече неща, отколкото самата формула.

3.7.23. Твърдение. *Ако скулемовото усилване на една формула е тъждествено вярно в структура* $\mathbf{M}$*, то и самата формула е тъждествено вярна в* $\mathbf{M}$*.*

Доказателство. Нека $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са свободните променливи на φ; тогава свободните променливи на $\exists \mathtt{x}\,(\varphi)$ са измежду $\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$.

Нека $\varphi[\mathtt{x} \coloneqq \tau]$ е скулемово усилване на $\exists \mathtt{x}\,(\varphi)$ и $\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \tau]$. За да докажем $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, да изберем произволна оценка v в $\mathbf{M}$. Трябва да докажем $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[v]$. От $\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \tau]$ следва

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \tau][v]$$

Съгласно (25) това е еквивалентно на

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq [\![\tau]\!]^{\mathbf{M}}(v), v(\mathtt{x}_1), v(\mathtt{x}_2), \ldots, v(\mathtt{x}_n)]$$

което съгласно дефиниция 3.3.2 ж) ни дава

$$\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq v(\mathtt{x}_1), v(\mathtt{x}_2), \ldots, v(\mathtt{x}_n)]$$

т.е.

$$\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[v]$$

■

3.7.24. Твърдение. *Нека* $\varphi[s]$ *е скулемово усилване на* $\exists \mathtt{x}\,(\varphi)$ *и скулемовият символ не се среща никъде във формулата* φ*. Ако* $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$*, то съществува структура* $\mathbf{K}$*, която съвпада с* $\mathbf{M}$ *във всичко, освен може би при интерпретацията на скулемовия символ, такава, че* $\mathbf{K} \vDash \varphi[s]$*.*

Доказателство. Доказателството ще извършим по отделно в зависимост от това дали скулемовото усилване е от първи или втори вид.

(първи вид) Съгласно дефиниция 3.7.20 субституцията s има вида $\mathtt{x} \coloneqq \mathtt{c}$, където символът за константа $\mathtt{c}$ не се среща никъде във формулата φ, а формулата $\exists \mathtt{x}\,(\varphi)$ няма свободни променливи. Тъй като $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, то съгласно дефиниция 3.3.2 ж) получаваме, че съществува такова $\mu \in |\mathbf{M}|$, че

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mu]$$

Нека структурата $\mathbf{K}$ е идентична с $\mathbf{M}$ във всичко, освен в интерпретацията на символа $\mathtt{c}$ — нека $\mathtt{c}^{\mathbf{K}} = \mu$.

За да установим, че $\mathbf{K} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{c}]$, да изберем произволна оценка v. Трябва да докажем

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{c}][v]$$

Съгласно (25) това е еквивалентно на

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq [\![\mathtt{c}]\!]^{\mathbf{M}}(v)]$$

т.е. на

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mu]$$

което вече видяхме, че е вярно.

(втори вид) Съгласно дефиниция 3.7.20 субституцията s има вида $\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, където функционалният символ $\mathtt{f}$ не се среща никъде във формулата φ, а $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ са свободните променливи на $\exists \mathtt{x}\,(\varphi)$. Тъй като $\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)$, то за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$

$$\mathbf{M} \vDash \exists \mathtt{x}\,(\varphi)[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu_1, \mu_2, \ldots, \mu_n]$$

откъдето съгласно дефиниция 3.3.2 ж) получаваме, че за произволни $\mu_1, \mu_2, \ldots, \mu_n \in |\mathbf{M}|$ съществува такова $\mu \in |\mathbf{M}|$, че

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \mu, \mu_1, \mu_2, \ldots, \mu_n] \tag{26}$$

Нека $f\colon |\mathbf{M}|^n \to |\mathbf{M}|$ е онази функция, която на така избрани $\mu_1, \mu_2, \ldots, \mu_n$ съпоставя така намереното μ.*

Нека структурата $\mathbf{K}$ е идентична с $\mathbf{M}$ във всичко, освен в интерпретацията на символа $\mathtt{f}$ — нека $\mathtt{f}^{\mathbf{K}} = f$.

За да установим, че $\mathbf{K} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)]$, да изберем произволна оценка v. Трябва да докажем

$$\mathbf{M} \vDash \varphi[\mathtt{x} \coloneqq \mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)][v]$$

Съгласно (25) това е еквивалентно на

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq [\![\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)]\!]^{\mathbf{M}}(v), [\![\mathtt{x}_1]\!]^{\mathbf{M}}(v), \ldots, [\![\mathtt{x}_n]\!]^{\mathbf{M}}(v)]$$

т.е. на

$$\mathbf{M} \vDash \varphi[\mathtt{x}, \mathtt{x}_1, \ldots, \mathtt{x}_n \coloneqq f(v(\mathtt{x}_1), \ldots, v(\mathtt{x}_n)), v(\mathtt{x}_1), \ldots, v(\mathtt{x}_n)]$$

Последното е истина предвид (26) и дефиницията на функцията f. ∎

*На това място използваме т.н. *аксиома за избора*.

Забележка: В доказателството на твърдение 3.7.24 за втория вид скулемово усилване използвахме т.н. *аксиома за избора*. Един начин да формулираме тази аксиома е следният:

> Ако за всяко $x \in X$ съществува $y \in Y$, за които е верен някакъв предикат $p(x, y)$, то тогава съществува такава функция f, че за всяко $x \in X$ е вярно $p(x, f(x))$.

Ако тук интерпретираме квантора „съществува“ конструктивно, тази аксиома е съвсем естествена, защото в този случай ще разполагаме с конкретен метод, посредством който по дадено x можем да получим нужното y. Когато интерпретираме квантора класически, не разполагаме с никакъв метод, посредством който по x можем да получим y и затова аксиомата за избора дълго време е била една от най-оспорваните аксиоми на теория на множествата. Всъщност методът на скулемизацията, може да се обоснове и без да се използва аксиомата за избора, но доказателството става значително по-усложнено.

3.7.25. Дефиниция. Множество от формули е *изпълнимо*, ако съществува структура, в която са тъждествено верни всички формули от множеството. Множество от формули е *неизпълнимо*, ако не е изпълнимо.

3.7.26. (скулемизация) Нека ни е дадено крайно множество от формули Γ в пренексна нормална форма и да разгледаме следния процес, който ще наречем *скулемизация*. Избираме произволна формула от Γ, която съдържа квантор. Ако формулата започва с квантор $\forall$, то отстраняваме този квантор. Съгласно твърдение 3.7.19 ще получим такова множество от формули Γ', че Γ е изпълнимо тогава и само тогава, когато Γ е изпълнимо. Ако пък избраната формула започва с квантор $\exists$, то да заменим тази формула с такова нейно скулемово усилване, че СКУЛЕМОВИЯТ ФУНКЦИОНАЛЕН СИМВОЛ ДА НЕ СЕ СРЕЩА В НИКОЯ ФОРМУЛА ОТ Γ.* Съгласно твърдение 3.7.23, ако Γ' е изпълнимо, то и Γ ще е изпълнимо. Ще докажем и обратното — ако Γ е изпълнимо, то и Γ' е изпълнимо. Да допуснем, че Γ е изпълнимо и нека $\mathbf{M}$ е някоя структура, в която са тъждествено верни формулите от Γ. Съгласно твърдение 3.7.24 или 3.7.24 съществува структура $\mathbf{K}$, в която е вярно скулемовото усилване на формулата, започваща с $\exists$. Останалите формули са идентични в Γ и Γ'. Те обаче не съдържат скулемовия символ, а структурите $\mathbf{M}$ и $\mathbf{K}$ съвпадат за всички символи, освен евентуално

*Разбира се, може да направим това само ако в сигнатурата има достатъчно символи.

за скулемовия. Тъй като тези формули са тъждествено верни в $\mathbf{M}$, то те са тъждествено верни и в $\mathbf{K}$.

Тъй като на всяка стъпка от описания процес изчезва по един квантор, то след краен брой стъпки ще получим множество, в което всички формули са безкванторни. При това, така намереното множество не се различава от първоначалното по отношение на своята изпълнимост — ако първоначалното множество е изпълнимо, то и новополученото ще е неизпълнимо, и ако новополученото е изпълнимо, то значи и първоначалното множество е било изпълнимо.

И така, доказахме следната теорема:

3.7.27. Теорема. *Нека* Γ *е крайно*[*] *множество от формули. Ако в сигнатурата има достатъчно символи, то ще можем да намерим такова крайно множество* Γ' *от безкванторни формули, че* Γ *да бъде изпълнимо тогава и само тогава, когато е изпълнимо* Γ'.

Доказателство. Да приведем формулите от Γ в пренексен вид и след това да приложим скулемизация (вж. 3.7.26). ∎

3.7.28. Дефиниция. Казваме, че една формула е в *скулемова нормална форма*, ако:

- формулата е в пренексна нормална форма;
- формулата не съдържа нито един квантор $\exists$;
- формулата не съдържа свободни променливи.

3.7.29. Теорема за скулемизацията. *Нека* Γ *е крайно*[**] *множество от формули. Ако в сигнатурата има достатъчно символи, то ще можем да намерим такова крайно множество* Γ' *от формули в скулемова нормална форма, че* Γ *да бъде изпълнимо тогава и само тогава, когато е изпълнимо* Γ'.

Доказателство. Най-напред, използвайки теорема 3.7.27 можем да получим крайно множество Δ от безкванторни формули, което е изпълнимо тогава и само тогава, когато е изпълнимо Γ. Ако пред всяка от безкванторните формули в Δ сложим достатъчно квантори за всеобщност, ще получим множество Γ' в скулемова нормална форма. Освен това, съгласно твърдение 3.7.19, формулите от множеството Γ' са тъждествено верни точно в онези структури, в които са тъждествено верни и формулите от Δ. Следователно Δ е изпълнимо тогава и само тогава, когато е изпълнимо Γ'. ∎

[*]Теоремата е вярна и за безкрайно Γ, но тук няма да обосноваваме това.

[**]Теоремата е вярна и за безкрайно Γ, но тук няма да обосноваваме това.

3.7.30. Твърдение. *За всеки три формули φ, ψ и χ:*

а) $\vDash \varphi \vee (\psi \,\&\, \chi) \Leftrightarrow (\varphi \vee \psi) \,\&\, (\varphi \vee \chi)$

б) $\vDash (\psi \,\&\, \chi) \vee \varphi \Leftrightarrow (\psi \vee \varphi) \,\&\, (\chi \vee \varphi)$

Доказателство. Да изберем произволна структура $\mathbf{M}$ и оценка v в $\mathbf{M}$. Нека A е съждението $\mathbf{M} \vDash \varphi[v]$, B е съждението $\mathbf{M} \vDash \psi[v]$ и C е съждението $\mathbf{M} \vDash \chi[v]$. Трябва да докажем, че са верни твърденията

> „Съждението (A или (B и C)) е вярно тогава и само тогава, когато е вярно ((A или B) и (A или C))“

и

> „Съждението ((B и C) или A) е вярно тогава и само тогава, когато е вярно ((B или A) и (C или A))“

И двете твърдения са очевидни. ∎

3.7.31. Дефиниция. а) *Литерал* означава атомарна формула или отрицание на атомарна формула.

б) *Елементарна дизюнкция* означава безквантворна формула, в която единствените логически операции са дизюнкция и отрицание и всяко отрицание се намира пред атомарна формула.

в) Една безквантворна формула е в *конюнктивна нормална форма*, ако представлява конюнкция от елементарни конюнкции.[*]

3.7.32. Твърдение. *За всяка безквантворна формула можем да намерим еквивалентна на нея формула в конюнктивна нормална форма.*

Доказателство. Нека φ е произволна безквантворна формула. Да я приведем в отрицателна нормална форма.[**] По този начин ще получим безквантворна формула, в която единствените логически операции са конюнкция, дизюнкция и отрицания и всяко отрицание се намира пред атомарна формула. Да прилагаме към така получената формула докато може замени от следния вид:

$$\varphi \vee (\psi \,\&\, \chi) \longmapsto (\varphi \vee \psi) \,\&\, (\varphi \vee \chi) \tag{27}$$

$$(\psi \,\&\, \chi) \vee \varphi \longmapsto (\psi \vee \varphi) \,\&\, (\chi \vee \varphi) \tag{28}$$

[*] По точно, можем да дадем следната индуктивна дефиниция на това какво значи *конюнктивна нормална форма*:

а) всяка елементарна дизюнкция е в конюнктивна нормална форма;

б) конюнкция на две формули в конюнктивна нормална форма е формула в конюнктивна нормална форма.

[**] Да припомним, че за целта е достатъчно да прилагаме докато може замени от

Съгласно твърдение 3.7.30 при замени от този вид ще получаваме еквивалентни формули. Освен това след всяка такава замяна формулата ще остава безквантorна, няма да се появят други операции, освен конюнкция, дизюнкция и отрицание и всяко отрицание ще си остане пред атомарната си формула. Може да забележим обаче, че ако след краен брой стъпки стигнем до формула, към която не можем да приложим замяна от горния вид, това ще означава, че сме получили формула в конюнктивна нормална форма. Това показва частичната коректност на алгоритъма за привеждане в конюнктивна нормална форма. Остава да видим защо алгоритъмът винаги свършва след краен брой стъпки.

(I начин) За произволна формула χ да наречем *дълбочина* на подформулата χ' на χ дължината на най-дългата редица от вида

$$\chi_1, \chi_2, \chi_3, \ldots, \chi_k$$

където $\chi_1 = \chi$, $\chi_k = \chi'$, $\chi_i \neq \chi_{i+1}$, χ_{i+1} е подформула на χ_i и формулите $\chi_1, \chi_2, \ldots, \chi_{k-1}$ не са от вида $\chi' \,\&\, \chi''$. Ако си мислим формулата като дърво, тогава дълбочината на една подформула е равна на разстоянието между корена на поддървото и корена на цялото дърво, като при това пропускаме възлите от дървото, в които стои конюнкция.

Може да се забележи, че ако формулата ψ се получава от φ посредством някоя от замените (27) и (28), то броят на подформулите от вида $\chi' \,\&\, \chi''$ не се променя, а дълбочината на някоя подформула от този вид намалява. Това не може да продължи до безкрайност.

(II начин) Да разгледаме следния начин, по който от безквантorна формула в отрицателна нормална форма получаваме аритметичен израз. Да заменим всеки литерал с числото 2. Да заменим всяка подформула от вида $\chi' \,\&\, \chi'$ с $x + y$, където x и y са изразите, получени съответно от χ' и χ''. Да заменим всяка подформула от вида $\chi' \vee \chi'$ с $x.y$, където x и y са изразите, получени съответно от χ' и χ''. След всяко

следния вид:

$$\begin{aligned}
\varphi \Leftrightarrow \psi &\longmapsto (\varphi \Rightarrow \psi) \,\&\, (\psi \Rightarrow \varphi) \\
\varphi \Rightarrow \psi &\longmapsto \neg\varphi \vee \psi \\
\neg\neg\varphi &\longmapsto \varphi \\
\neg(\varphi \vee \psi) &\longmapsto \neg\varphi \,\&\, \neg\psi \\
\neg(\psi \,\&\, \varphi) &\longmapsto \neg\psi \vee \neg\varphi
\end{aligned}$$

преобразование на формулата от горния вид, стойността на съответния аритметичен израз няма да се промени, защото $x.(y + z) = x.y + x.z$ и $(y + z).x = y.x + z.x$. Същевременно след всяко такова преобразование се увеличава броят на числата 2 в така получения аритметичен израз. Това не може да продължи до безкрайност, защото за числа не по-малки от 2 е вярно $x + y \leqq x.y$ и значи стойността на всеки такъв аритметичен израз със сигурност е не по-малка от удвоения брой на числата 2 в него. ■

Глава 4

Логическо програмиране

4.1. Логическо програмиране с ограничения

4.1.1. В този и следващите раздели ще считаме, че $\top$ е някоя произволно избрана затворена безквантorна формула, която е винаги вярна (т.е. предикатна тавтология), а $\bot$ е произволно избрана затворена безквантorна формула, която е винаги невярна. Например ако φ е коя да е затворена безквантorна формула, може да положим $\top$ да бъде формулата $(\varphi \Rightarrow \varphi)$, а $\bot$ да бъде формулата $\neg\top$.*

4.1.2. Дефиниция. Израз от вида

$$\psi \text{ :- } \varphi_1, \varphi_2, \ldots, \varphi_n \tag{29}$$

където $n \geqq 0$, а $\psi, \varphi_1, \varphi_2, \ldots, \varphi_n$ са атомарни формули, ще наричаме *клауза*. Следвайки синтаксиса на пролог, когато $n = 0$ ще си позволяваме да пишем просто ψ вместо ψ :- .

Ще използваме следната дефиниция, с цел да избегнем нуждата от предефиниране на много неща, които вече сме дефинирали за формули:

*Ако сигнатурата не съдържа нито един символ за константа, такива затворени безквантorни формули може и да не съществуват, но тук няма да се интересуваме от този случай. Всъщност на практика ограничението да имаме поне един символ за константа изобщо не е ограничително. Дори и да нямаме такъв символ, можем да променим сигнатурата като добавим в нея някакъв символ за константа, защото тази промяна няма да доведе до никакви съществени изменения.

4.1.3. Дефиниция. При $n \geqq 1$ формулата

$$\varphi_1 \,\&\, \varphi_2 \,\&\, \ldots \,\&\, \varphi_n \Rightarrow \psi$$

ще наричаме *формула* на клаузата (29). Когато $n = 0$, формулата на клаузата (29) е ψ.

Използваме дефиниция 4.1.3 по следния начин. Ще считаме, че една клауза е вярна в структура при оценка, ако нейната формула е вярна. Ще считаме, че една клауза е тъждествено вярна в структура, ако е тъждествено вярна нейната формула. Ще считаме, че някоя структура е модел за множество от клаузи, ако тази структура е модел за формулите на клаузите от множеството. И т.н.

Да забележим, че съгласно дефиниция 4.1.3 без значение дали $n = 0$ или $n \geqq 1$, клаузата (29) е вярна в структура $\mathbf{M}$ при оценка v тогава и само тогава, когато е вярно твърдението

> *Ако всяка от формулите* $\varphi_1, \varphi_2, \ldots, \varphi_n$ *е вярна в* $\mathbf{M}$ *при оценка* v*, то формулата* ψ *също е вярна в* $\mathbf{M}$ *при оценка* v

4.1.4. Дефиниция. Израз от вида

$$\texttt{?-}\varphi_1, \varphi_2, \ldots, \varphi_n \tag{30}$$

където $n \geqq 0$, а $\varphi_1, \varphi_2, \ldots, \varphi_n$ са атомарни формули, ще наричаме *запитване*.

Също както при клаузите, и тук за да избегнем нуждата да даваме голямо количество нови дефиниции, ще въведем понятието формула на запитване:

4.1.5. Дефиниция. При $n \geqq 1$ формулата

$$\varphi_1 \,\&\, \varphi_2 \,\&\, \ldots \,\&\, \varphi_n$$

ще наричаме *формула* на запитването (30). Когато $n = 0$, формулата на запитването (30) е $\top$.

Във всяка логическа програма се използват два вида предикатни символи — предикатни символи, чийто смисъл се дефинира посредством клаузите от програмата, и предикатни символи, които са вградени в използвания език за логическо програмиране и не само, че не е нужно, но не е и позволено да се предефинират.

4.1.6. Нека ВГРАД е сигнатура, която съдържа всички символи за константи, всички функционални символи и някои предикатни символи от сигнатурата $\mathfrak{sig}$. За предикатния символ за равенство задължително ще поискаме да бъде сред символите на ВГРАД. Ще считаме, че сигнатурата ВГРАД съдържа вградените предикатни символи в използвания език за логическо програмиране.

4.1.7. ***Забележка:*** а) Това, че ВГРАД съдържа всички символи за константи и всички функционални символи, означава, че разработваме теория на езици за логическо програмиране, в които програмистът има право да дефинира нови предикати, но не и нови функции.

б) За краткост, в много книги за пролог предикатните символи се наричат предикати. В този курс обаче ще внимаваме да използваме тези термини правилно. Предикатните символи са символи, с които се означават определени предикати. Логическите програми са синтактични обекти, следователно в тях се използват предикатни символи, а не предикати. Например `append` е вграден предикатен символ във всички версии на пролог. В пролог този вграден предикатен символ се интерпретира като това, което обикновено наричаме „вграден предикат `append`“, т.е. триаргументният предикат, който казва че ако конкатенираме списъците, подадени като първи и втори аргумент, ще получим списъка, подаден като трети аргумент. Да си представим обаче език за логическо програмиране, в който `append` прави нещо друго. Тогава в този език за програмиране също ще има вграден предикатен символ `append`, но този предикатен символ няма да се интерпретира като вградения предикат `append` на пролог, а като някакъв друг вграден предикат.

4.1.8. Дефиниция. *Логическа програма* означава крайно множество от клаузи, в които отляво на символа `:-` не се използват предикатни символи от сигнатурата ВГРАД.

4.1.9. Дефиниция. Нека сигнатурата $\mathfrak{sig}_2$ съдържа всички символи от сигнатурата $\mathfrak{sig}_1$. Казваме, че структурата $\mathbf{M}_2$ за $\mathfrak{sig}_2$ е *обогатяване* на структурата $\mathbf{M}_1$ за $\mathfrak{sig}_1$, ако:

- двете структури имат един и същ универсум и
- двете структури интерпретират по един и същ начин символите за константи, функционалните и предикатните символи от $\mathfrak{sig}_1$.

Да забележим, че ако $\mathbf{M}$ е обогатяване на $\mathbf{K}$, то една функция е оценка в $\mathbf{M}$ тогава и само тогава, когато тя е оценка в $\mathbf{K}$.

4.1.10. Да фиксираме структура $\mathbf{B}$ за сигнатурата ВГРАД. Тази структура „казва“ как са интерпретирани в езика за логическо програмиране символите за константи, функционалните символи и вградените предикатни символи. Когато от тук нататък споменем някоя структура без да уточняваме коя е нейната сигнатура, ще считаме (както и досега), че това е структура за $\mathfrak{sig}$ (т.е. в нея са дефинирани всички предикатни символи, а не само вградените).

Когато пишем някоя логическа програма, ние си мислим, че дефинираме определени предикати. С други думи, нашата логическа програма дефинира определена структура. Какви свойства има тази структура? От една страна искаме в тази структура вградените предикатни символи да се интерпретират както в $\mathbf{B}$. Това означава, че структурата, която дефинираме, трябва да бъде обогатяване на $\mathbf{B}$. От друга страна, в тази структура трябва да са тъждествено верни клаузите от програмата. Достатъчни ли са тези две изисквания, за да определят еднозначно структурата, която имаме предвид? Следващият пример показва, че не.

4.1.11. Пример. Нека структурата $\mathbf{M}$ има същия универсум като $\mathbf{B}$ и символите за константи, функционалните символи и вградените предикатни символи се интерпретират също като в структурата $\mathbf{B}$. Нека невградените предикатни символи са винаги истина в $\mathbf{M}$ (при произволни аргументи). По дефиниция структурата $\mathbf{M}$ ще бъде обогатяване на $\mathbf{B}$. Освен това може да се забележи, че тя ще бъде модел за абсолютно всички логически програми. Наистина, нека

$$\psi \text{ :- } \varphi_1, \varphi_2, \ldots, \varphi_n$$

е произволна клауза от програмата. Съгласно дефиниция 4.1.8 предикатният символ на атомарната формула ψ не е вграден, следователно формулата ψ е вярна в $\mathbf{M}$ при произволна оценка, а от тук следва, че и цялата клауза е вярна при произволна оценка.

Но въпреки, че една логическа програма може да има много модели, които са обогатявания на $\mathbf{B}$, все пак когато пишем програмата, ние имаме предвид един точно определен модел. Нека например сигнатурата ВГРАД съдържа триместен предикатен символ `append` и нека структурата $\mathbf{B}$ интерпретира този символ като пролог. Да разгледаме

логическата програма, която се състои от следната клауза:*

$$\mathtt{p(x, y)\ :-\ append(x, x, y)}$$

Интуитивно ни се иска да считаме, че тази програма дефинира предикат $p(x, y)$, който казва, че ако конкатенираме списъка x със себе си, ще получим списъка y.

Как можем да определим структурата, „която имаме предвид“? Едно очевидно свойство на „лошата“ структура $\mathbf{M}$ от пример 4.1.11 е това, че в нея има твърде много верни неща. Всъщност измежду всички структури, които са едновременно обогатявания на $\mathbf{B}$ и модели на логическата програма, в структурата $\mathbf{M}$ ще бъдат верни възможно най-много атомарни формули. Когато обаче един програмист пише логическа програма, той иска за дефинираните от него предикати да е истина само това, което изрично е посочено в програмата като истина, а всичко останало да е лъжа. Възниква въпросът: не може ли да дефинираме структура, която също е обогатяване на $\mathbf{B}$ и която също е модел на логическата програма, но в която са верни възможно най-малко атомарни формули? Оказва се, че отговорът на този въпрос е положителен. Тази структура се нарича *най-малък модел* на логическата програма. Именно най-малкият модел е структурата, която програмистът има предвид, когато пише логическа програма.

4.1.12. Теорема за най-малкия модел. *Нека* Γ *е логическа програма. Тогава съществува структура, която ще означаваме* $T_\Gamma^\infty(\tilde{B})$ *и ще наричаме* най-малък модел *на* Γ*, която има следните свойства:*

- $T_\Gamma^\infty(\tilde{B})$ *е обогатяване на* $\mathbf{B}$*;*
- $T_\Gamma^\infty(\tilde{B})$ *е модел на* Γ*;*
- *ако*

  $$\mathtt{p}^{T_\Gamma^\infty(\tilde{B})}(\beta_1, \beta_2, \ldots, \beta_n)$$

  е истина за някои $\beta_1, \beta_2, \ldots, \beta_n \in |\mathbf{B}|$*, то за всяка структура* $\mathbf{M}$*, която също като* $T_\Gamma^\infty(\tilde{B})$ *е едновременно обогатяване на* $\mathbf{B}$ *и модел на* Γ*, ще бъде истина*

  $$\mathtt{p}^{\mathbf{B}}(\beta_1, \beta_2, \ldots, \beta_n)$$

4.1.13. Следствие. *Нека* Γ *е логическа програма и* φ *е запитване. Ако запитването* φ *е вярно в най-малкия модел на* Γ *при оценка* v*, тогава то е вярно при оценка* v *във всички модели на* Γ*, които са обогатявания на* $\mathbf{B}$*.*

*Тук не се съобразяваме с изискването на пролог, според което променливите трябва да започват с главна буква.

Доказателство. Съгласно теоремата, ако една атомарна формула е вярна при дадена оценка в най-малкия модел на Γ, тогава тя ще бъде вярна във всеки модел на Γ, който е обогатяване на **B**. От тук следва исканото, защото запитванията представляват конюнкции от атомарни формули. ∎

4.1.14. Дефиниция. Нека е дадена логическа програма Γ. Казваме, че при дадената програма *запитването* φ *се удовлетворява с отговор* v, ако това запитване е вярно при оценка v във всеки модел на Γ, който е обогатяване на **B**.

Да забележим, че тази дефиниция зависи от конкретния избор на структурата **B**. Това е естествено — ако вградените предикатни символи почнат да работят по друг начин (т.е. структурата **B** се смени), нормално е да очакваме, че ще получаваме различни отговори на запитванията към дадена логическа програма.

4.1.15. От следствие 4.1.13 следва, че при програма Γ запитването φ се удовлетворява с отговор v тогава и само тогава, когато запитването φ е вярно в най-малкия модел на Γ при оценка v. С други думи, запитването е вярно при дадена оценка в структурата, „за която програмистът си мисли“, тогава и само тогава, когато то е вярно във всички модели на програмата, в които вградените предикати са реализирани „както трябва“. Това още веднъж потвърждава интуицията, че структурата, „за която програмистът си мисли“, е структурата с възможно най-малко верни неща — верни са само запитванията, чиято вярност изрично следва от това, което програмистът е казал в програмата, а всички останали запитвания са неверни.

Пристъпваме към дефиницията на алгоритъм, който е достатъчно ефективен, за да се използва на практика, и чрез който може да намираме отговорите, при които дадено запитване се удовлетворява при дадена програма. Описаният алгоритъм реализира това, което се нарича *логическо програмиране с ограничения*, на английски — constraint logic programming.

4.1.16. Дефиниция. *Ограничение* ще наричаме формула, която или е равна на $\top$, или представлява конюнкция от атомарни формули от сигнатурата ВГРАД.

Ако условно приемем, че $\top$ е конюнкция на нула атомарни формули от сигнатурата ВГРАД, то тогава може да кажем, че ограничение означава конюнкция на нула или повече атомарни формули от ВГРАД.

4.1.17. Ще приемем, че разполагаме с алгоритъм, посредством който за произволно ограничение може да проверяваме дали то е изпълнимо в структурата $\mathbf{B}$, или не. Няма да уточняваме какъв точно е този алгоритъм, защото това разбира се зависи от това какви вградени предикати са реализирани в $\mathbf{B}$.*

4.1.18. Дефиниция. *Състояние* означава наредена двойка от запитване и ограничение. Използваме следното означение за състояние от запитване φ и ограничение ψ:

$$\langle \varphi \parallel \psi \rangle$$

4.1.19. Езиците за логическо програмиране с ограничения работят по следния начин. Да допуснем, че сме задали на компютъра запитване φ. Започваме изпълнението на логическата програма от състояние $\langle \varphi \parallel \top \rangle$. След това на всяка стъпка преобразуваме текущото състояние по начина, описан в следващата дефиниция, като идеята на това преобразуване е следната — постепенно превеждаме нещата, които са „казани“ от запитването φ, като използваме само предикати от $\mathbf{B}$. По този начин все по-голяма и по-голяма част от информацията, съдържаща се в състоянието, ще се премества от запитването в ограничението. Ако след краен брой такива преобразувания получим състояние от вида $\langle \texttt{?-} \parallel \psi \rangle$, т.е. състояние, в което запитването съдържа нула формули, то значи цялата информация вече се съдържа в ограничението, а за ограничението в 4.1.17 приехме, че имаме алгоритъм, с който можем да проверяваме за изпълнимост (обикновено разполагаме с алгоритъм, с който не само може да проверяваме за изпълнимост, но също така може да намерим и при кои стойности на променливите ограничението ще бъде вярно.).

4.1.20. Дефиниция. Нека Г е логическа програма и ни е дадено състояние

$$\langle \texttt{?-p}(\tau_1, \ldots, \tau_n), \varphi_1, \varphi_2, \ldots, \varphi_k \parallel \psi \rangle$$

Тогава:

а) Ако $\texttt{p}$ е символ от ВГРАД, то даденото състояние *се свежда* до

$$\langle \texttt{?-}\varphi_1, \varphi_2, \ldots, \varphi_k \parallel \texttt{p}(\tau_1, \ldots, \tau_n) \,\&\, \psi \rangle$$

*Може да считаме, че алгоритъмът за унификация, който ще дефинираме в раздел 4.3, е алгоритъмът, който пролог използва, за да провери дали дадено ограничение е изпълнимо.

ако ограничението в новото състояние е изпълнимо в $\mathbf{B}$. С други думи, ако първата атомарна формула от текущата цел е с вграден предикатен символ, то просто я прехвърляме в ограничението.

б) Ако $\mathtt{p}$ не е символ от ВГРАД, а клаузата

$$\mathtt{p}(\sigma_1, \ldots, \sigma_n) \texttt{ :- } \chi_1, \chi_2, \ldots, \chi_m$$

се получава като преименуваме променливите в някоя клауза от логическата програма Γ, то даденото състояние се свежда до

$$\langle \texttt{?-}\chi_1, \chi_2, \ldots, \chi_m, \varphi_1, \varphi_2, \ldots, \varphi_k \parallel \tau_1 = \sigma_2 \,\&\, \ldots \,\&\, \tau_n = \sigma_n \,\&\, \psi \rangle$$

ако ограничението в новото състояние е изпълнимо в $\mathbf{B}$.

4.1.21. ***Забележка*:** а) Да забележим, че тази дефиниция е формулирана по такъв начин, че без значение дали предикатният символ $\mathtt{p}$ е от ВГРАД, или не, се грижим новото състояние да има изпълнимо ограничение. Правим това, защото ако получим състояние, в което ограничението няма решение, то всяко следващо състояние няма да има решение и значи няма смисъл да смятаме.

б) Когато предикатният символ $\mathtt{p}$ не е от ВГРАД, в общия случай текущото състояние се преобразува по недетерминиран начин. Това е така, защото в програмата може да има много клаузи от вида

$$\mathtt{p}(\sigma_1, \ldots, \sigma_n) \texttt{ :- } \chi_1, \chi_2, \ldots, \chi_m$$

По принцип това означава, че при многопроцесорни или многоядрени компютри имаме възможност да изпълняваме паралелно тези алтернативни свеждания на текущото състояние. За съжаление езикът пролог не се възползва от паралелизма на съвременните компютри — при него клаузите се обработват последователно според реда, в който са написани в програмата.

в) Причината, поради която преименуваме променливите в клаузите от програмата, е следната — променливите, които се срещат в текущото състояние, нямат по смисъл нищо общо с променливите, които са били използвани, когато пишем програмата. Затова, за да не възникнат проблеми, работим не непосредствено с клаузите от програмата, а с варианти на клаузите, в които променливите са преименувани, така че да се различават от променливите в състоянието.

Следващата дефиниция формализира това, което в 4.1.19 изказахме неформално.

4.1.22. Дефиниция. а) Редица от състояния, всяко от които се свежда към следващото, се нарича *извод*.

б) Нека е дадена логическа програма Г. Казваме, че при дадената програма *запитването φ се удовлетворява с обратен извод с отговор v*, ако съществува извод, в който първото състояние е $\langle \varphi \| \top \rangle$, последното има вида $\langle \texttt{?-} \parallel \psi \rangle$ и $\mathbf{B} \vDash \psi[v]$.

Да илюстрираме с конкретен пример работата на език за логическо програмиране с ограничения.

4.1.23. Пример. Нека универсумът на структурата **B** се състои от реалните числа, двуместните функционални символи `+`, `-`, `*` и `/` се интерпретират съответно като събиране, изваждане, умножение и деление на реални числа и имаме два предикатни символа `=` и `<`, които се интерпретират съответно като „равно“ и „по-малко“. За удобство ще считаме, че освен това в **B** са дефинирани символи за константи за реални числа. Например нека символът 1 се интерпретира като числото 1, 5.5 се интерпретира като числото $5\frac{1}{2}$ и т.н.*

Важно: Забележете, че в структурата **B** има вградени функции за събиране, изваждане, умножение и деление на реални числа. В пролог няма такива вградени функции и вместо тях се използва вграденият предикат `is`. Следователно повечето логически програми, използващи тази структура, а в частност и по-долната програма, **не са** коректни програми на пролог.

При такава структура **B**, може да дефинираме предикат за факториел по следния начин:

```
fact(0,1).
fact(N,N*X) :- 0<N, fact(N-1,X).
```

Нека видим как при горната програма ще се обработи запитването `?-fact`$(3, X)$. Началното състояние ще бъде

$$\langle \texttt{?-fact}(3, X) \parallel \top \rangle$$

Ако се опитаме към това състояние да приложим факта `fact`$(0, 1)$, трябва да получим състоянието

$$\langle \texttt{?-} \parallel 3 = 0 \,\&\, X = 1 \,\&\, \top \rangle$$

*Един възможен метод за проверка на изпълнимост на ограничения при такава структура **B** е да използваме симплекс-метода и алгоритъма на Гаус – Жордан, за да елиминираме линейните равенства и неравенства и да отложим обработката на останалите условия, докато те станат линейни.

Ограничението на това състояние е неизпълнимо в $\mathbf{B}$, защото формулата $3 = 0$ е невярна в $\mathbf{B}$, и затова не го използваме.

Ако вместо това използваме втората клауза, то успешно ще сведем първоначалното състояние към следното:

$$\langle \texttt{?-}0 < N_1, \texttt{fact}(N_1 - 1, X_1) \parallel 3 = N_1 \,\&\, X = N_1 * X_1 \,\&\, \top \rangle$$

Забележете, че сме преименували променливите в клаузата — вместо N имаме N_1 и вместо X имаме X_1.

Ако спазваме точно дефиниции 4.1.20 и 4.1.22, сега би трябвало да приложим някоя клауза към така полученото състояние. На практика обаче всички реализации на логически езици в този момент опростяват състоянието. В нашия случай, ако елиминираме променливата N_1 като я заменим навсякъде с 3 и махнем $\top$ от ограничението, ще получим следното състояние:

$$\langle \texttt{?-}0 < 3 - 1, \texttt{fact}(3 - 1, X_1) \parallel X = 3 * X_1 \rangle$$

Тъй като първата атомарна формула в това състояние е с вграден предикатен символ, то просто я прехвърляме в ограничението:

$$\langle \texttt{?-fact}(3 - 1, X_1) \parallel 0 < 3 - 1 \,\&\, X = 3 * X_1 \rangle$$

И отново опростяваме като премахнем от ограничението формулата $0 < 3 - 1$, която е вярна в $\mathbf{B}$:

$$\langle \texttt{?-fact}(3 - 1, X_1) \parallel X = 3 * X_1 \rangle$$

Ако към това състояние се опитаме да приложим факта $\texttt{fact}(0, 1)$, трябва да получим състоянието

$$\langle \texttt{?-} \parallel 3 - 1 = 0 \,\&\, X_1 = 1 \,\&\, X = 3 * X_1 \rangle$$

Ограничението на това състояние е неизпълнимо в $\mathbf{B}$, защото формулата $3 - 1 = 1$ е невярна в $\mathbf{B}$, и затова не го използваме.

Ако вместо това приложим втората клауза, то успешно ще сведем състоянието до следното:

$$\langle \texttt{?-}0 < N_2, \texttt{fact}(N_2 - 1, X_2) \parallel 3 - 1 = N_2 \,\&\, X_1 = N_2 * X_2 \,\&\, X = 3 * X_1 \rangle$$

И отново опростяваме:

$$\langle \texttt{?-}0 < 3 - 1, \texttt{fact}(3 - 1 - 1, X_2) \parallel X = 3 * 2 * X_2 \rangle$$

Първата атомарна формула в това състояние е с вграден предикатен символ, така че просто я прехвърляме в ограничението:

$$\langle \texttt{?-fact}(3-1-1, X_2) \parallel 0 < 3-1 \,\&\, X = 3*2*X_2 \rangle$$

И опростяваме, като махнем тази формула от ограничението, защото тя е вярна в $\mathbf{B}$:

$$\langle \texttt{?-fact}(3-1-1, X_2) \parallel X = 3*2*X_2 \rangle$$

Към това състояние не може да приложим факта $\texttt{fact}(0,1)$, защото, както и по-горе, ще получим състояние с неизпълнимо ограничение. Ако вместо това приложим втората клауза, свеждаме горното състояние до следното:

$$\langle \texttt{?-}0 < N_3, \texttt{fact}(N_3 - 1, X_3) \| 3-1-1 = N_3 \,\&\, X_2 = N_3 * X_3 \,\&\, X = 3*2*X_2 \rangle$$

Опростяваме ограничението:

$$\langle \texttt{?-}0 < 3-1-1, \texttt{fact}(3-1-1-1, X_3) \parallel X = 3*2*1*X_3 \rangle$$

Първата атомарна формула е с вграден предикатен символ. Прехвърляме я в ограничението, след което забелязваме, че тя е вярна в $\mathbf{B}$ и затова я махаме:

$$\langle \texttt{?-fact}(3-1-1-1, X_3) \parallel X = 3*2*1*X_3 \rangle$$

Ако към това състояние приложим факта $\texttt{fact}(0,1)$, ще получим състоянието

$$\langle \texttt{?-} \parallel 3-1-1-1 = 0 \,\&\, X_3 = 1 \,\&\, X = 3*2*1*X_3 \rangle$$

Опростяваме ограничението:

$$\langle \texttt{?-} \parallel X = 6 \rangle$$

По този начин изчислението завършва успешно с отговор $X = 6$, което наистина е факториелът на числото 3.

И така, имаме две дефиниции — 4.1.14 и 4.1.22 б) — в които се дефинират понятията „удовлетворява“ и „удовлетворява с обратен извод“. Двете дефиниции са очевидно различни. Първата дефиниция — за „удовлетворява“ — е семантична и при нея по-лесно може да си представим какво всъщност означава едно запитване да се удовлетворява

с даден отговор. Втората — за „удовлетворява с обратен извод“ — се свежда до изпълнението на някакъв алгоритъм, но изобщо не е очевидно кога всъщност този алгоритъм ще определи, че дадено запитване се удовлетворява. За щастие, оказва се, че тези две дефиниции са еквивалентни.

Твърдението, което казва, че отговорите, получени алгоритмично (т.е. отговорите според втората дефиниция), са верни (от гледна точка на първата дефиниция), се нарича „теорема за коректност“. А твърдението, което казва, че всички верни отговори (от гледна точка на първата дефиниция) могат да се получат алгоритмично (т.е. според втората дефиниция), се нарича теорема за пълнота.

4.1.24. Теорема за коректност. *Нека е дадена логическа програма* Г. *Ако при тази програма запитването* φ *се удовлетворява с обратен извод с отговор* v, *то* φ *се удовлетворява с отговор* v.

Обратното твърдение в общия случай не е вярно — ако някое запитване се удовлетворява с отговор v, не е сигурно, че това запитване ще се удовлетвори с обратен извод с отговор v. Причината за това е следната — оценката v дава стойности на всички променливи по напълно произволен начин, докато алгоритъмът за намиране на отговори се интересува единствено от променливите, които са споменати в запитването. Може да формулираме теоремата за пълнота по следния начин:

4.1.25. Теорема за пълнота. *Нека е дадена логическа програма* Г. *Ако при тази програма запитването* φ *се удовлетворява с отговор* v, *то съществува оценка* w, *която съвпада с* v *за всички променливи, които се срещат в запитването* φ, *и такава, че* φ *се удовлетворява с обратен извод с отговор* w.

4.2. Ербранови структури

В раздел 4.1 се уговорихме, че разполагаме с фиксирана структура $\mathbf{B}$. Тази структура определя какви вградени функции и предикати има в езика за логическо програмиране, а от нейния универсум разбираме с какви обекти може да работим в програмата. За логическата програма може да кажем, че надгражда над структурата $\mathbf{B}$, т.е. приемаме, че имаме наготово всичко, предоставено от $\mathbf{B}$ и използваме логика, за да дефинираме нови предикати. Затова почти всички дефиниции в раздел 4.1 споменаваха структурата $\mathbf{B}$. Така например за една цел φ

казахме, че се удовлетворява при дадена логическа програма с отговор v, ако φ е вярна при оценка v във всички структури, които не само бяха модел на програмата, но също са и обогатявания на $\mathbf{B}$ (вж. дефиниция 4.1.14).

Въпреки че когато програмираме този подход е естествен, от логическа гледна точка е по-естествено да разгледаме вариант на логическото програмиране, при който няма структура $\mathbf{B}$. С други думи, интересен е вариант на логическото програмиране, при който не използваме логиката, за да надграждаме над нещата, предоставени от $\mathbf{B}$, а абсолютно всичко свеждаме към логика. В този случай за една цел φ казваме (вж. дефиниция 3.4.12), че се удовлетворява, ако φ е изпълнима във всички модели на логическата програма (а не само при моделите, които са обогатявания на $\mathbf{B}$). Разбира се, при този подход няма как да кажем с какви отговори се удовлетворява целта, защото конкретните отговори зависят от универсума на структурата, а нямаме структура $\mathbf{B}$, която да ни каже кой е универсума.

За щастие, оказва се, че не се налага да разработваме нов вид теория на логическото програмиране. Френският логик Жак Ербран (1908 – 1931) е установил, че вместо да разглеждаме произволни структури с кой знае какъв универсум, е достатъчно да се ограничим със структури, чийто универсум се състои от всички термове без променливи, а стойността на кой да е терм без променливи в тази структура е самият терм. Такива структури се наричат *ербранови*.

Нека отбележим, че ербрановите структури не винаги съществуват. Съгласно задача 14 ако в сигнатурата няма нито един символ за константи, то няма да съществуват термове без променливи, а универсумите на структурите не е позволено да бъдат празното множество. Ако пък в сигнатурата има поне един символ за константа, то този символ сам си е терм без променливи, така че в този случай ербранови структури ще съществуват.

4.2.1. Всичко в този раздел ще правим при предположението, че в сигнатурата има поне един символ за константи. Всъщност на практика ограничението да имаме поне един символ за константа изобщо не е ограничително. Дори и да нямаме такъв символ, можем да променим сигнатурата като добавим в нея някакъв символ за константа, защото тази промяна няма да доведе до никакви съществени изменения.

4.2.2. Дефиниция. Структурата $\mathbf{H}$ е *ербранова*, ако:

– Универсумът на $\mathbf{H}$ е множеството от всички термове, които не съдържат променливи.

– За всеки символ за константа $\mathtt{c}$, стойността му е самата константа, т.е. $\mathtt{c}^{\mathbf{H}} = \mathtt{c}$. (Да забележим, че $\mathtt{c}$ е терм без променливи.)

– За всеки n-местен функционален символ $\mathtt{f}$ и елементи $\tau_1, \tau_2, \ldots, \tau_n$ на универсума на $\mathbf{H}$ имаме

$$\mathtt{f}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_2) \tag{31}$$

(Да забележим, че щом $\tau_1, \tau_2, \ldots, \tau_n$ са от универсума на $\mathbf{H}$, то те са термове без променливи, а значи и $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_2)$ е терм без променливи, т.е. елемент на универсума.)

4.2.3. Дефиниция. Множеството от всички термове без променливи ще наричаме *ербранов универсум*.

4.2.4. Обърнете внимание, че скобите от двете страни на равенство (31) имат напълно различен смисъл. Скобите отляво ги използваме, за да покажем, че даваме $\tau_1, \tau_2, \ldots, \tau_n$ като аргументи на функцията $\mathtt{f}^{\mathbf{H}}$. Скобите отдясно пък са просто символи — вторият и последният символ на терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Също така напълно различен е и смисълът на запетаите. Използваме запетаите отляво, за да разделим аргументите на функцията $\mathtt{f}^{\mathbf{H}}$. За разлика от тях, запетаите отдясно са просто символи, които се срещат някъде в терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$.

***Задача* 47:** Нека сигнатурата съдържа поне един символ за константа. Докажете подробно, че съществуват ербранови структури.

Решение:

Бърз преглед на дефиницията на структура 3.1.7 показва, че за да дефинираме коя да е структура (вкл. ербранова структура) трябва да определим следното: кой е универсумът, как се интерпретират символите за константите, как се интерпретират функционалните символи и как се интерпретират предикатните символи.

В случая можем да вземем универсъмът на структурата да бъде множеството от всички термове без променливи. Да забележим, че това множество е непразно, защото сигнатурата съдържа поне един символ за константа и този символ е терм без променливи.

Интерпретацията на символите за константи и функционалните символи на всяка една ербранова структура е еднозначно определена, така че не се налага да я измисляме: $\mathtt{c}^{\mathbf{H}} = \mathtt{c}$ и $\mathtt{f}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Няма никакви ограничения за интерпретацията на предикатните символи. За определеност може да считаме, че $\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_N)$ е истина за всеки предикатен символ $\mathtt{p}$ и елементи $\tau_1, \tau_2, \ldots, \tau_n$ на универсума на $\mathbf{H}$.

Тази дефиниция на ербрановата структура $\mathbf{H}$ можем да изкажем по-формално по следния начин.

Нека $\mathbf{H} = \langle \Omega, \mathtt{i} \rangle$, където Ω е множеството от всички термове без променливи.

За всеки символ за константа $\mathtt{c}$ нека $\mathtt{i}(\mathtt{c}) = \mathtt{c}$.

За всеки n-местен функционален символ $\mathtt{f}$ нека $\mathtt{i}(\mathtt{f}) = f$, където $f \colon \Omega^n \to \Omega$ е функцията

$$f(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$$

(Отляво на горното равенство скобите и запетаите разграничават аргументите на функцията f, докато отдясно те са просто символи, срещащи се някъде в терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$.)

За всеки n-местен предикатен символ $\mathtt{p}$ нека $\mathtt{i}(\mathtt{p})$ е n-местният предикат p, за който

$$p(\tau_1, \tau_2, \ldots, \tau_n) \longleftrightarrow 1 = 1$$

Така дефинирана, структурата $\mathbf{H}$ ще бъде ербранова.

Край на решението.

Тъй като елементите на универсума на една ербранова структура са термове без променливи, то всяка оценка в тази структура съпоставя на променливите термове без променливи. Това означава, че всяка оценка в ербранова структура представлява субституция и значи можем да я прилагаме по два различни начина:

- можем да пресметнем стойността на един терм при тази оценка;
- можем да си мислим, че оценката е субституция и да я приложим към този терм.

Следващото твърдение показва, че и в двата случая ще получим един и същ резултат, т.е. $[\![\tau]\!]^{\mathbf{H}}(v) = \tau[v]$.

4.2.5. Твърдение. *Нека $\mathbf{H}$ е ербранова структура и v е оценка в $\mathbf{H}$. Тогава стойността на кой да е терм τ в $\mathbf{H}$ при оценка v е равна на резултата от прилагането на субституцията v към терма τ.*

Доказателство. Ще докажем твърдението с индукция по терма τ. Да припомним, че с $[\![\tau]\!]^{\mathbf{H}}(v)$ означаваме стоността на τ при оценка v в структурата $\mathbf{H}$, а $\tau[v]$ е резултатът от прилагането на субституцията v към τ.

Ако $\tau = \mathtt{x}$ е променлива, то стойността на τ при оценка v е стойността на $\mathtt{x}$ при оценка v, т.е. $v(\mathtt{x})$. Да приложим субституцията v към $\mathtt{x}$ означава да заместим $\mathtt{x}$ с $v(\mathtt{x})$ и значи отново получаваме $v(\mathtt{x})$.

Ако $\tau = \mathtt{c}$ е символ за константа, то по дефиниция 3.2.2 стойността на τ в $\mathbf{H}$ при коя да е оценка е $\mathtt{c}^{\mathbf{H}} = \mathtt{c}$. Резултатът от прилагането на коя да е субституция към терма $\mathtt{c}$ също е $\mathtt{c}$.

Нека $\tau = \mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$. Стойността на τ в $\mathbf{H}$ при оценка v е

$$[\![\tau]\!]^{\mathbf{H}}(v) = \mathtt{f}^{\mathbf{H}}([\![\tau_2]\!]^{\mathbf{H}}(v), [\![\tau_2]\!]^{\mathbf{H}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{H}}(v))$$

Понеже структурата е ербранова, последното е равно на

$$\mathtt{f}([\![\tau_1]\!]^{\mathbf{H}}(v), [\![\tau_2]\!]^{\mathbf{H}}(v), \ldots, [\![\tau_n]\!]^{\mathbf{H}}(v))$$

което съгласно индукционното предположение е равно на

$$\mathtt{f}(\tau_1[v], \tau_2[v], \ldots, \tau_n[v]) = (\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n))[v] = \tau[v]$$

■

4.2.6. Следствие. *Ако* $\mathbf{H}$ *е ербранова структура и* τ *е терм без променливи, то стойността на* τ *в* $\mathbf{H}$ *при коя да е оценка е* τ*.*

Доказателство. Нека v е произволна оценка в $\mathbf{H}$. Съгласно твърдение 4.2.5 стойността на τ в $\mathbf{H}$ при оценка v е равна на резултата от прилагането на субституцията v към τ. Но τ не съдържа променливи, които субституциите могат да заместват и значи като приложим v към τ получаваме пак τ. ■

***Задача* 48:** Нека $\mathbf{H}$ е ербранова структура и оценката v в $\mathbf{H}$ е такава, че $v(\mathtt{x}) = \mathtt{f}(\mathtt{f}(\mathtt{c}))$ и $v(\mathtt{y}) = \mathtt{g}(\mathtt{c}, \mathtt{f}(\mathtt{a}))$. Кои от скобите и кои от запетаите в следния израз са символи и кои не са символи:

$$\mathtt{f}(\mathtt{g}^{\mathbf{H}}(v(\mathtt{x}), (\mathtt{g}(\mathtt{a}, \mathtt{y}))[v]))$$

Колко леви скоби и колко запетаи се съдържат в стойността на този израз?

Решение:

Ако означаваме скобите и запетаите, които са символи, с ❟, ❨ и ❩, тогава можем да препишем израза от условието на задачата така:

$$\mathtt{f}\big(\mathtt{g}^{\mathbf{H}}(v(\mathtt{x}), (\mathtt{g}\big(\mathtt{a}❟\mathtt{y}\big))[v])\big)$$

Стойността на този израз в $\mathbf{H}$ е термът $\mathtt{f}(\mathtt{g}(\mathtt{f}(\mathtt{f}(\mathtt{c})), \mathtt{g}(\mathtt{a}, \mathtt{g}(\mathtt{c}, \mathtt{f}(\mathtt{a})))))$. С просто преброяване установяваме, че той съдържа 7 леви скоби и 3 запетаи.

Край на решението.

За атомарните формули вместо твърдение 4.2.5 можем да докажем следното твърдение:

4.2.7. Твърдение. *Нека* $\mathbf{H}$ *е ербранова структура,* v *е оценка в* $\mathbf{H}$ *и* φ *е атомарна формула. Тогава формулата* φ *е вярна в* $\mathbf{H}$ *при оценка* v *тогава и само тогава, когато формулата* $\varphi[v]$ *е вярна в* $\mathbf{H}$.

Доказателство. Нека $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$. Тъй като структурата е ербранова, съгласно твърдение 4.2.5 стойността на τ_i в $\mathbf{H}$ при оценка v е $\tau_i[v]$. Термовете $\tau_i[v]$ не съдържат променливи и значи съгласно твърдение 4.2.6 тяхната стойност също е $\tau_i[v]$. Следователно както атомарната формула $\varphi = \mathtt{p}(\tau_1, \tau_2, \ldots, \tau_n)$, така и атомарната формула $\varphi[v] = \mathtt{p}(\tau_1[v], \tau_2[v], \ldots, \tau_n[v])$ е вярна в $\mathbf{H}$ при оценка v тогава и само тогава, когато е истина $\mathtt{p}^{\mathbf{H}}(\tau_1[v], \tau_2[v], \ldots, \tau_n[v])$. Остава само да отбележим, че съгласно твърдение 3.4.9, да бъде формулата $\varphi[v]$ вярна при оценка v е същото, каквото тя да бъде вярна при коя да е оценка. ∎

4.2.8. Лема. *Нека* $\mathbf{M}$ *е произволна структура. Тогава съществува ербранова структура* $\mathbf{H}$ *и силен хомоморфизъм* $h\colon \mathbf{H} \to \mathbf{M}$.

Доказателство. Дефиниция 4.2.2 за ербранова структура ни казва какъв е универсумът на $\mathbf{H}$ — множеството от всички термове, които не съдържат променливи. Пак от дефиницията разбираме и как в $\mathbf{H}$ се интерпретират символите за константи и функционалните символи. Остава единствено да определим как в $\mathbf{H}$ се интерпретират предикатните символи. Да отложим засега това.

Хомоморфизмът h трябва да изобразява всеки терм без променливи в елемент на универсума на $\mathbf{M}$. Да дефинираме $h(\tau)$ да бъде стойността на τ в $\mathbf{M}$ (тъй като τ не съдържа променливи, не е нужно да уточняваме коя е оценката, вж. твърдение 3.2.6). Да докажем, че това наистина е силен хомоморфизъм.

Първо, по дефиниция h изобразява елементите на универсума на $\mathbf{H}$ е елементи на универсума на $\mathbf{M}$.

Второ, за всеки символ за константа $\mathtt{c}$:

$$h(\mathtt{c}^{\mathbf{H}}) = h(\mathtt{c}) = \mathtt{c}$$

Освен това, за всеки n-местен функционален символ $\mathtt{f}$ (в долните

равенства с $[\![\ldots]\!]^{\mathbf{M}}$ е означена стойността в $\mathbf{M}$ на терм без променливи):

$$\begin{aligned} h(\mathtt{f}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n)) &= h(\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)) \\ &= [\![\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)]\!]^{\mathbf{M}} \\ &= \mathtt{f}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}, [\![\tau_2]\!]^{\mathbf{M}}, \ldots, [\![\tau_n]\!]^{\mathbf{M}}) \\ &= \mathtt{f}^{\mathbf{M}}(h(\tau_1), h(\tau_2), \ldots, h(\tau_n)) \end{aligned}$$

И накрая, за всеки n-местен предикатен символ $\mathtt{p}$ може просто да дефинираме

$$\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) \overset{\text{def}}{\longleftrightarrow} \mathtt{p}^{\mathbf{M}}([\![\tau_1]\!]^{\mathbf{M}}, [\![\tau_2]\!]^{\mathbf{M}}, \ldots, [\![\tau_n]\!]^{\mathbf{M}})$$

защото тогава

$$\mathtt{p}^{\mathbf{H}}(\tau_1, \tau_2, \ldots, \tau_n) \longleftrightarrow \mathtt{p}^{\mathbf{M}}(h(\tau_1), h(\tau_2), \ldots, h(\tau_2))$$

■

4.2.9. Теорема („малка“ теорема на Ербран). *а) Нека* Γ *е множество от безкванторни формули и* φ *е безкванторна формула. Тогава* φ *е изпълнима във всеки модел на* Γ *тогава и само тогава, когато* φ *е изпълнима във всеки ербранов модел на* Γ.

б) Нека Γ *е множество от безкванторни формули. Тогава* Γ *има модел тогава и само тогава, когато* Γ *има ербранов модел.*

Доказателство. **(а)** Ясно е, че ако φ е изпълнима във всеки модел на Γ, то в частност φ ще бъде изпълнима и във всеки ербранов модел на Γ. За да докажем обратната посока, да допуснем, че φ е изпълнима във всеки ербранов модел на Γ и да изберем произволен (не задължително ербранов) модел на Γ. Трябва да докажем, че φ е изпълнима в $\mathbf{M}$.

От лема 4.2.8 намираме ербранова структура $\mathbf{H}$ и хомоморфизъм $h\colon \mathbf{H} \to \mathbf{M}$. От твърдение 3.5.9 а) следва, че $\mathbf{H}$ е модел на Γ. Тъй като φ е изпълнима във всеки ербранов модел на Γ, то значи φ е изпълнима в $\mathbf{H}$, следователно от твърдение 3.5.9 б) получаваме, че φ е изпълнима в $\mathbf{M}$.

(б) Може да сведем към (а). Нека $\varphi = \bot$. Тъй като φ е неизпълнима формула, то φ е изпълнима във всеки модел на Γ тогава и само тогава, когато Γ няма модели и φ е изпълнима във всеки ербранов модел на Γ тогава и само тогава, когато Γ няма ербранови модели. ■

4.2.10. Може да използваме „малката“ теорема на Ербран, за да сведем логическото програмиране без структура **B** към логическото програмиране с ограничения, разгледано в раздел 4.1. Нека единственият предикатен символ в сигнатурата ВГРАД е равенството. И нека **B** е ербрановата структура за тази сигнатура, в която символът за равенство се интерпретира като равенство (т.е. универсумът на **B** съдържа термовете без променливи, а символите за константи и функционалните символи се интерпретират съгласно дефиниция 4.2.2). При така дефинирана структура **B** е вярно следното следствие:

4.2.11. Следствие. *Нека* Г *е логическа програма, която не съдържа символа за равенство и* φ *е запитване, което също не съдържа символа за равенство. Тогава* φ *се удовлетворява от* Г *(вж. дефиниция 3.4.12) тогава и само тогава, когато съществува отговор, с който при тази програма запитването* φ *се удовлетворява (вж. дефиниция 4.1.14).*

Доказателство. φ се удовлетворява от Г тогава и само тогава, когато φ е изпълнима във всеки модел на Г. Съгласно малката теорема на Ербран това се случва тогава и само тогава, когато φ е изпълнима във всеки ербранов модел на Г. Тъй като Г и φ не съдържат символа за равенство, то интерпретацията на този символ е без значение и значи последното се случва тогава и само тогава, когато φ е изпълнима във всеки ербранов модел на Г, в който символа за равенство се интерпретира като равенство. Но една структура е ербранова и символа за равенство в нея се интерпретира като равенство тогава и само тогава, когато тя е обогатяване на **B**. Следователно горното се случва тогава и само тогава, когато φ е изпълнима във всеки модел на Г, който е обогатяване на **B**. Последното е равносилно с това да съществува отговор, с който при програма Г запитването φ се удовлетворява. ■

4.3. Алгоритъм за унификация

Да припомним, че в 4.2.10 дефинирахме **B** да бъде ербранова структура, в която единственият предикатен символ е = и той се интерпретира по обичайния начин, т.е. като равенство. Теорема 4.2.9 и следствие 4.2.11 показват, че тази структура **B** е много интересна от теоретична гледна точка. Тя обаче е интересна и от практическа гледна точка, защото общо взето може да считаме, че точно такава структура **B** се използва в езика пролог. Всъщност в езика пролог има немалко вградени предикати, за които не можем да се даде никаква логическа семантика (такива са например предикатите **!** и ==). Ако обаче разг-

ледаме версия на езика пролог, от която са били „изхвърлени“ всички „нелогически“ предикати, то може да считаме, че такъв пролог реализира логическо програмиране с ограничения на основата на такава структура **B**.

Всъщност дори и да изхвърлим „нелогическите“ предикати, пак в пролог ще останат много вградени предикати освен равенството. Това обаче от теоретична гледна точка не е от значение, защото се оказва, че тези предикати може да се дефинират с логическа програма и значи не е нужно да считаме, че са вградени предикати.

4.3.1. ***Забележка*:** Разбира се, вградените предикати на пролог са твърде много, така че нито е практически възможно, нито пък има смисъл тук да се опитваме да докажем, че те наистина могат да се дефинират с логическа програма. Донякъде интуиция, че това е така, се дава от упражненията по логическо програмиране във ФМИ, защото единствените вградени предикати, които се използват по съществен начин на упражненията са аритметичните предикати `is`, `<`, `=<`, `>=` и `>`.*

В този раздел ще опишем алгоритъм, който може да се използва за решаване на ограничения, когато структурата **B** е дефинирана както в 4.2.10. От съображения за ефективност повечето версии на пролог използват леко видоизменен вариант на този алгоритъм, в който не се извършва т.н. occurs check (вж. **??**).

Да припомним, че съгласно дефиниция 4.1.16 ограниченията представляват конюнкции от атомарни формули от сигнатурата ВГРАД. В нашия случай единственият предикатен символ в сигнатурата ВГРАД е $=$, което означава, че ограниченията представляват конюнкции от равенства между термове:

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \ldots \,\&\, \tau_n = \sigma_n$$

Затова може да си мислим, че ограниченията представляват системи от нула (когато ограничението е $\top$) или повече от уравнения, които решаваме в структурата **B**.

4.3.2. Дефиниция. Казваме, че оценката v е *решение на ограничението* φ, ако $\mathbf{B} \vDash \varphi[v]$.

Разбира се, v е решение на дадено ограничение тогава и само тогава, когато всички уравнения в него са верни при оценка v.

*Може би не е очевидно, че на пролог може да се дефинират аритметични предикати без да се разчита на вградените аритметични предикати, но това е така.

4.3.3. Дефиниция. За две ограничения казваме, че са *еквивалентни* в **B**, ако имат едни и същи решения.

4.3.4. Дефиниция. Казваме, че ограничението φ е *решено* относно променливата $\mathtt{x}$, ако тази променлива се среща точно веднъж в φ и то в уравнение от вида $\mathtt{x} = \tau$. (От тук в частност следва, че $\mathtt{x}$ не се среща в τ, нито в което и да е друго уравнение в ограничението). За уравнението $\mathtt{x} = \tau$ казваме, че е *решаващо*.

Алгоритъмът за унификация, към чиято дефиниция пристъпваме сега, представлява метод, посредством който за всяко ограничение може да намерим еквивалентно на него ограничение, в което всички уравнения са решаващи.

4.3.5. Дефиниция. Следните преобразувания на ограничения ще наричаме *решаващи преобразувания*:

Първо решаващо преобразувание. Ако в ограничението има уравнение от вида

$$\tau = \mathtt{x}$$

където $\mathtt{x}$ е променлива, а термът τ не е променлива, то заменяме това уравнение с

$$\mathtt{x} = \tau$$

Второ решаващо преобразувание. Ако в ограничението има уравнение от вида

$$\mathtt{x} = \mathtt{x}$$

където $\mathtt{x}$ е променлива, то отстраняваме това уравнение от ограничението.

Трето решаващо преобразувание. Ако в ограничението има уравнение от вида

$$\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

то заменяме това уравнение с

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \ldots \,\&\, \tau_n = \sigma_n$$

Ако в ограничението има уравнение от вида

$$\mathtt{c} = \mathtt{c}$$

където $\mathtt{c}$ е символ за константа, то отстраняваме това уравнение от ограничението.

Четвърто решаващо преобразуване. Ако в ограничението има уравнение от вида

$$\mathtt{x} = \tau$$

което не е решаващо и променливата $\mathtt{x}$ не се среща в терма τ, то заместваме навсякъде в останалите уравнения променливата $\mathtt{x}$ с τ.

4.3.6. Алгоритъмът за унификация се състои в следното: да прилагаме решаващи преобразувания към ограничението по произволен начин докато може. Следващото твърдение показва, че ако към някое ограничение не може повече да прилагаме решаващи преобразувания, то в него със сигурност или всички уравнения ще бъдат решаващи, или ограничението ще съдържа уравнение, което няма да има решения, и значи цялото ограничение няма да има решения.

4.3.7. Твърдение. *Ако към някое ограничение не може да се прилагат решаващи преобразувания, то в него със сигурност или всички уравнения са решаващи, или ограничението съдържа уравнение, което няма решения.*

Доказателство. Ако разгледаме какво представляват решаващите преобразувания, може да забележим, че ако към някое ограничение не може повече да прилагаме такива преобразувания, то със сигурност всяко уравнение в ограничението ще бъде или решаващо, или ще има някой от следните пет вида:

$$\begin{aligned}
\mathtt{a} &= \mathtt{b} && (\mathtt{a} \text{ и } \mathtt{b} \text{ са различни символи})\\
\mathtt{c} &= \mathtt{f}(\dots) &&\\
\mathtt{f}(\dots) &= \mathtt{c} &&\\
\mathtt{f}(\dots) &= \mathtt{g}(\dots) && (\mathtt{f} \text{ и } \mathtt{g} \text{ са различни символи})\\
\mathtt{x} &= \tau && (\tau \text{ не е променлива и променливата } \mathtt{x} \text{ се съдържа в } \tau)
\end{aligned}$$

Всяко уравнение от тези пет вида няма решение. За първите четири това е очевидно, защото $\mathbf{B}$ е ербранова структура, а за петото да забележим, че съгласно твърдение 4.2.5 стойността на τ при коя да е оценка v се получава като заменим всяка променлива в τ, включително променливата $\mathtt{x}$, с $v(\mathtt{x})$. Това означава, че броят на символите в стойността на τ при оценка v със сигурност е по-голям, от колкото броят на символите в $v(\mathtt{x})$. ∎

За да има полза от твърдение 4.3.7, трябва да докажем следните две неща: 1) че от каквото и ограничение да започнем и както и да прилагаме решаващите преобразувания, то със сигурност след краен брой

стъпки ще получим ограничение, към което повече не може да прилагаме решаващи преобразувания (т.е. алгоритъмът за унификация не се зацикля) и 2) след всяко прилагане на решаващо преобразувание, новото ограничение е еквивалентно в $\mathbf{B}$ на първоначалното (т.е. алгоритъмът за унификация е коректен). Тези две неща са доказани в твърдения 4.3.8 и 4.3.9.

4.3.8. Твърдение. *Не е възможно към едно ограничение да прилагаме решаващи преобразувания до безкрайност.*

Доказателство. Първо да забележим, че ако системата е решена относно някоя променлива, то и след прилагането на някое решаващо преобразувание тя ще остане решена относно тази променлива. От друга страна след прилагането на четвъртото решаващо преобразувание броят на променливите, относно които ограничението е решено, се увеличава с една. Тъй като в ограничението се срещат само краен брой променливи, ясно е, че няма как до безкрайност да увеличаваме броят на променливите, относно които ограничението е решено. Следователно четвъртото решаващо преобразувание може да се прилага само краен брой пъти. След като приложим четвъртото решаващо преобразувание за последен път получаваме ограничение, към което повече не се прилагат решаващи преобразувания от четвърти вид.

Сега да забележим, че решаващите преобразувания от първи, втори и трети вид не увеличават броя на срещанията на функционални символи и символи за константи в ограничението. От друга страна прилагането на решаващо преобразувание от трети вид със сигурност намалява този брой. Ясно е, че това няма как да става безброй много пъти, следователно третото решаващо преобразувание може да се прилага само краен брой пъти. След като го приложим за последен път, получаваме ограничение, към което повече не се прилагат решаващи преобразувания от трети и четвърти вид.

Самата дефиниция на преобразуванията от първи и втори вид изключва възможността да прилагаме само такъв вид преобразувания безброй пъти. ∎

4.3.9. Твърдение. *След прилагане на решаващо преобразувание получаваме ограничение, което е еквивалентно в* $\mathbf{B}$ *на първоначалното.*

Доказателство. Твърдението е очевидно по отношение на решаващите преобразувания от първи и втори вид.

Ако е било приложено решаващо преобразувание от трети вид за уравнението

$$\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n) = \mathtt{f}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

то също няма нищо сложно — тъй като структурата е ербранова, то това уравнение е вярно точно при онези оценки, при които е вярна конюнкцията

$$\tau_1 = \sigma_1 \,\&\, \tau_2 = \sigma_2 \,\&\, \ldots \,\&\, \tau_n = \sigma_n$$

Ако е било приложено решаващо преобразуване от трети вид за уравнението $\mathtt{c} = \mathtt{c}$, то твърдението отново е очевидно.

Нека е било приложено решаващо преобразуване от четвърти вид за уравнението

$$\mathtt{x} = \tau$$

Ако v е решение на ограничението (без значение дали преди или след преобразуването), то $v(\mathtt{x}) = [\![\tau]\!]^{\mathbf{H}}(v)$. Съгласно лемата за субституциите (твърдение 3.3.15), за произволен терм σ

$$[\![\sigma]\!]^{\mathbf{H}}(w) = [\![\sigma[\mathtt{x} \coloneqq \tau]]\!]^{\mathbf{H}}(v)$$

където оценката w е дефинирана с равенството

$$w(\mathtt{z}) = [\![\mathtt{z}[\mathtt{x} \coloneqq \tau]]\!]^{\mathbf{H}}(v)$$

Но $v(\mathtt{x}) = [\![\tau]\!]^{\mathbf{H}}(v)$, а когато $\mathtt{z} \neq \mathtt{x}$, то $\mathtt{z}[\mathtt{x} \coloneqq \tau] = \mathtt{z}$ и значи оценките v и w съвпадат. Следователно за произволен терм σ

$$[\![\sigma]\!]^{\mathbf{H}}(v) = [\![\sigma[\mathtt{x} \coloneqq \tau]]\!]^{\mathbf{H}}(v)$$

т.е. стойността на кой да е терм при оценка v си остава същата, ако към терма приложим субституцията $\mathtt{x} \coloneqq \tau$. Тъй като четвъртото решаващо преобразуване представлява прилагане на точно тази субституция, то значи v или е решение на ограничението както преди, така и след преобразуването, или v не е решение на ограничението нито преди, нито след преобразуването. ∎

4.3.10. В логическото програмиране алгоритъмът за унификация се използва по следния начин. След като го приложим към ограничението на текущото състояние, то или получаваме ограничение, за което знаем, че няма решение, или получаваме ограничение, в което всяко уравнение е решаващо. В първия случай цялото състояние няма решение и повече не го използваме. Във втория случай нека сме получили състояние от вида

$$\langle \mathtt{?\text{-}}\varphi_1, \varphi_2, \ldots, \varphi_k \parallel \mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \ldots \,\&\, \mathtt{x}_n = \tau_n \rangle$$

Да приложим към всяка от формулите $\varphi_1, \varphi_2, \ldots, \varphi_k$ субституцията $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n$. Ако означим така получените формули с $\varphi'_1, \varphi'_2, \ldots, \varphi'_k$, то ще получим състоянието

$$\langle \mathtt{?\text{-}}\varphi'_1, \varphi'_2, \ldots, \varphi'_k \parallel \mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \ldots \,\&\, \mathtt{x}_n = \tau_n \rangle$$

в което никоя от променливите $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ не се среща в запитването ?-$\varphi'_1, \varphi'_2 \ldots, \varphi'_k$. Това означава, че изпълнението на програмата може да продължи без да се интересуваме повече от ограничението. Чак когато пролог намери отговор на първоначално поставеното запитване от потребителя, може да използваме равенствата в ограничението, за да изведем отговор. И тъй като може да прилагаме алгоритъма за унификация винаги, когато решим, това означава, че изчисленията може да се извършват, използвайки състояния, от чиито ограничения не се интересуваме преди да завърши работата на програмата.

Тук няма да доказваме, че замяната на формулите $\varphi_1, \varphi_2, \ldots, \varphi_k$ с $\varphi'_1, \varphi'_2, \ldots, \varphi'_k$ по описания по-горе начин не променя по същество начина, по който работи една логическа програма, спрямо това, което вече бе описано в раздел 4.1. Ще се задоволим да докажем семантичната коректност на тази замяна. Тази коректност следва от следното твърдение:

4.3.11. Твърдение. *Нека променливите* $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ *не се срещат в термовете* $\tau_1, \tau_2, \ldots, \tau_n$. *Ако структурата* $\mathbf{M}$ *е обогатяване на* $\mathbf{B}$, *то формулата*

$$\varphi \,\&\, \mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \ldots \,\&\, \mathtt{x}_n = \tau_n$$

е вярна в $\mathbf{M}$ *при някоя оценка* v *тогава и само тогава, когато при същата оценка е вярна формулата*

$$\varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n] \,\&\, \mathtt{x}_1 = \tau_1 \,\&\, \mathtt{x}_2 = \tau_2 \,\&\, \ldots \,\&\, \mathtt{x}_n = \tau_n$$

Доказателство. Доказателството е аналогично на разглеждането на решаващите преобразувания от четвърти тип в твърдение 4.3.9, само че прилагаме лемата за субституциите за формули, а не за термове (в случая това е твърдение 3.3.18, а не 3.3.15).

Ако някоя от двете формули в условието на твърдението е вярна при оценка v, то $v(\mathtt{x}_i) = [\![\tau_i]\!]^{\mathbf{H}}(v)$. Съгласно лемата за субституциите (твърдение 3.3.18), за формулата φ е вярно

$$\mathbf{M} \vDash \varphi[w] \longleftrightarrow \mathbf{M} \vDash \varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n][v]$$

където оценката w е дефинирана с равенството

$$w(\mathtt{z}) = [\![\mathtt{z}[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n]]\!]^{\mathbf{H}}(v)$$

Но $v(\mathtt{x}_i) = [\![\tau_i]\!]^{\mathbf{H}}(v)$, а когато $\mathtt{z} \notin \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$, то

$$\mathtt{z}[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n] = \mathtt{z}$$

и значи оценките v и w съвпадат. Следователно за формулата φ е имаме

$$\mathbf{M} \vDash \varphi[v] \longleftrightarrow \mathbf{M} \vDash \varphi[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \coloneqq \tau_1, \tau_2, \ldots, \tau_n][v]$$

От тук следва искането. ∎

Казаното в 4.3.10 означава, че по същество пролог работи по начина, който вече илюстрирахме в пример 4.1.23, с тази разлика, че в пример 4.1.23 не бяхме уточнили какъв точно алгоритъм се използва, когато опростяваме състоянията, докато при пролог знаем, че за целта се използва алгоритъмът за унификация.

Да разгледаме с конкретен пример как работи пролог.

4.3.12. Пример. Нека програмата се състои от следните клаузи:

$$\mathtt{p(a)}\ \mathtt{:-}\ \mathtt{p(c)}. \tag{32}$$

$$\mathtt{p}(X)\ \mathtt{:-}\ \mathtt{r}(X). \tag{33}$$

$$\mathtt{r(a)}. \tag{34}$$

Ще приложим тази програма към запитването

$$\mathtt{?\text{-}p(a)}$$

Клауза (32) свежда първоначалното състояние

$$\langle \mathtt{?\text{-}p(a)} \parallel \top \rangle \tag{35}$$

към състоянието

$$\langle \mathtt{?\text{-}p(c)} \parallel \mathtt{a} = \mathtt{a} \rangle$$

Като приложим третото решаващо преобразуване към $\mathtt{a} = \mathtt{a}$, това състояние се опростява до

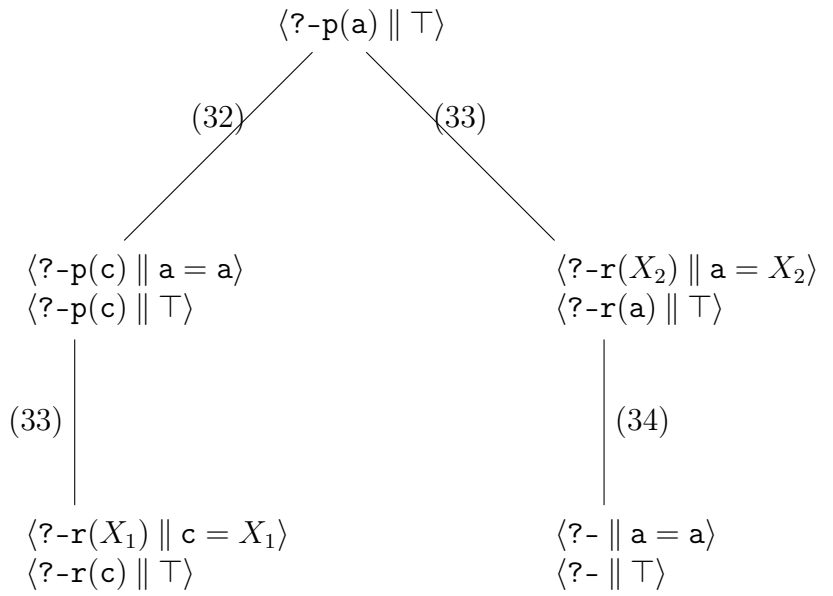
$$\langle \mathtt{?\text{-}p(c)} \parallel \top \rangle$$

Единствената клауза, която не свежда това състояние към състояние с неизпълнимо ограничение, е клауза (33). Посредством нея то се свежда до

$$\langle \mathtt{?\text{-}r}(X_1) \parallel \mathtt{c} = X_1 \rangle$$

Алгоритъмът за унификация свежда това ограничение към $X_1 = \mathtt{c}$. След прилагане на субституцията $X_1 \coloneqq \mathtt{c}$ към запитването, променливата X_1 става излишна, така че състоянието се опростява до

$$\langle \mathtt{?\text{-}r(c)} \parallel \top \rangle$$



Фиг. 11

Никоя клауза не може да се използва за това запитване, така че то остава неудовлетворено.

За запитването (35) имаме още една възможност — да приложим клауза (33). Посредством нея то се свежда до

$$\langle \texttt{?-r}(X_2) \parallel \texttt{a} = X_2 \rangle$$

Алгоритъмът за унификация свежда това ограничение към $X_2 = \texttt{a}$. След прилагане на субституцията $X_1 := \texttt{a}$ към запитването, променливата X_2 става излишна, така че състоянието се опростява до

$$\langle \texttt{?-r(a)} \parallel \top \rangle$$

Клауза (34) свежда това състояние до

$$\langle \texttt{?-} \parallel \texttt{a} = \texttt{a} \rangle$$

Третото решаващо преобразуване елиминира равенството $\texttt{a} = \texttt{a}$, така че получаваме успешното състояние

$$\langle \texttt{?-} \parallel \top \rangle$$

Всички свеждания от пример 4.3.12 могат да бъдат изобразени с дърво, както това е направено във фигура 11. Във върховете на това

дърво сме записали по две състояния. Първото се получава преди да приложим алгоритъмът за унификация, а второто — след това. Пълнотата означава, че ако първоначалната заявка се удовлетворява, то това дърво със сигурност ще съдържа успешен клон (т.е. клон, чиито състояния представляват извод).

Интерпретаторът на пролог обхожда това дърво в дълбочина, започвайки от най-левите клонове. В случая пролог ще обходи най-напред левия клон на дървото, където няма да намери решение на задачата, и след това ще се прехвърли на десния клон.

Когато в дървото има безкраен ляв клон, пролог ще се зацикли без да открие успешен клон. Да разгледаме още един пример.

4.3.13. Пример. Нека програмата този път се състои от следните клаузи:

$$\texttt{p}(X)\ \texttt{:-}\ \texttt{p}(\texttt{c}). \tag{36}$$

$$\texttt{p}(X)\ \texttt{:-}\ \texttt{r}(X). \tag{37}$$

$$\texttt{r}(\texttt{a}). \tag{38}$$

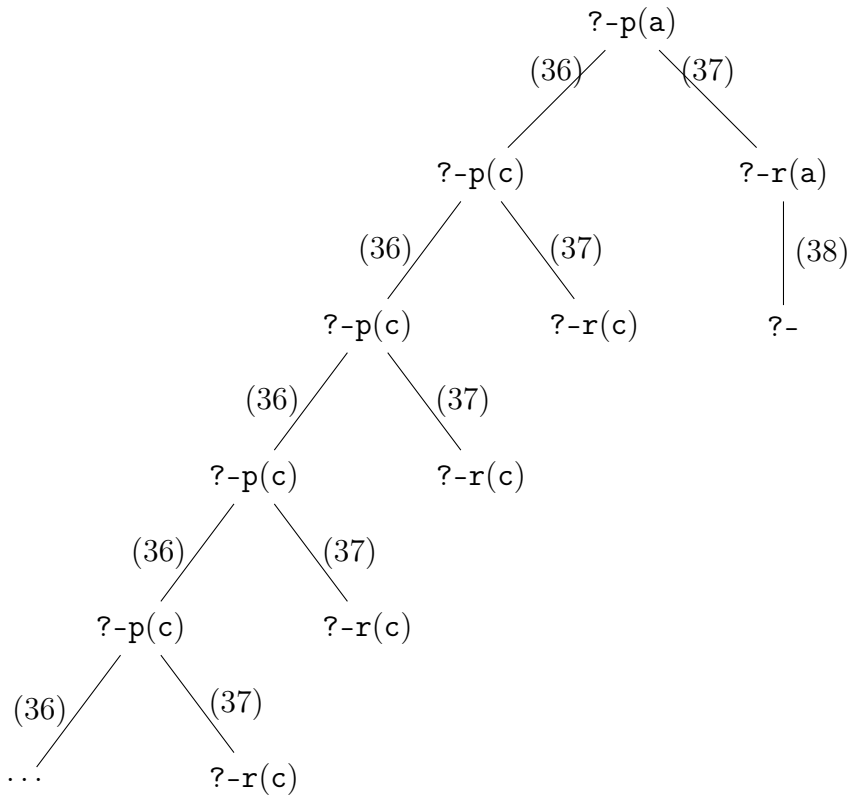
Дървото на извод, което се получава от тази програма при запитване `?-p(a)`, е изобразено на фигура 12. За по-добра прегледност, в това дърво са изобразени само запитванията в състоянията, които се получават след прилагане на алгоритъма за унификация. Вижда се, че това дърво съдържа безкраен ляв клон. Пролог ще тръгне по този клон и никога няма да открие десния и успешен клон.

Това показва, че при пролог е от значение редът, в който са записани клаузите. Когато дървото на извод е крайно, тогава при лошо подреждане на клаузите пролог може да работи значително по-бавно. Например при дървото на фигура 11 пролог ненужно хаби време, за да обхожда левия и неуспешен клон. Много по-лошо е положението, когато дървото е безкрайно. В този случай пролог може до безкрайност да се движи по някой безкраен клон, както се случи при дървото от фигура 12.

В първата от тези програми проблемът се оправя, ако се разменят клаузи (32) и (33), а във втората — като се разменят клаузи (36) и (37). За съжаление обаче, в някои случаи пролог се зацикля без значение в какъв ред подреждаме клаузите.

Да разгледаме още един пример.* Нека програмата се състои от

*Автор на примера е проф. Димитър Скордев [26].



Фиг. 12

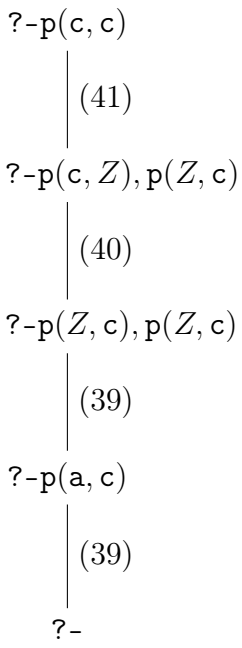
следните клаузи:

$$\texttt{p(a, c)}. \tag{39}$$

$$\texttt{p}(X, Y)\ \texttt{:-}\ \texttt{p}(Y, X). \tag{40}$$

$$\texttt{p}(X, Y)\ \texttt{:-}\ \texttt{p}(X, Z),\ \texttt{p}(Z, Y). \tag{41}$$

По принцип запитването `?-p(c, c)` може да се удовлетвори с обратен извод. Едно успешно свеждане на това запитване до празното запитване е дадено на фигура 13. Въпреки това, както и да подреждаме клаузите в тази програма, винаги ще се оказва, че отляво на този успешен клон, а и изобщо отляво на кой да е успешен клон, в дървото ще има безкраен клон. Затова при тази програма пролог ще се зацикля без значение как подреждаме клаузите в програмата.



Фиг. 13

Приложение A

Термове

A.1. Термовете в пролог

Синтаксис

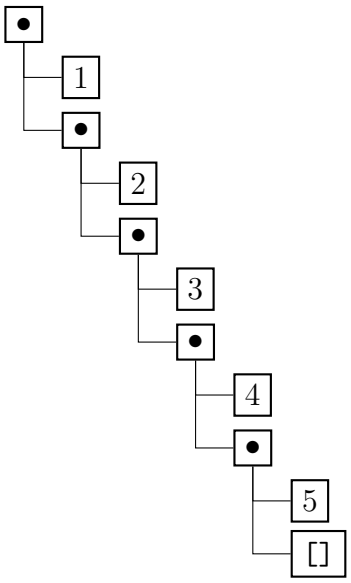
A.1.1. Езикът пролог използва синтаксис, подобен на математическия, със следните по-важни различия:

Първо, функционалните символи и символите за константи на пролог могат да се претоварват. Така например $\texttt{f}(\texttt{f}(\texttt{f},\texttt{f}))$ е правилен терм. В него първият символ `f` е едноместен, вторият — двуместен, а третият и четвъртият — символи за константи. Въпреки че са означени по един и същ начин, пролог интерпретира тези символи като напълно различни един от друг.[*]

Второ, пролог не използва математическата уговорка, съгласно която напр. `b` трябва да бъде символ за константа, а `x` — променлива. Вместо това, пролог използва следното правило: ако името започва с главна буква, то тогава то е променлива, а ако започва с малка буква, то е символ за константа или функционален символ.[**] Например `x(C)` е правилно построен терм, в който `x` е едноместен функционален символ, а `C` е променлива.

[*]За избягване на недоразумения, в документацията на пролог арността се обозначава с наклонена черта /. Например `f`/2 е двуместният функционален символ `f`, а `c`/0 — символът за константа `c`.

[**]Броят на аргументите показва дали е символ за константа или функционален символ.



Фиг. 14. Синтактично дърво за списъка `[1,2,3,4,5]`

Трето, както във всеки нормален език за програмиране, имената в пролог могат да бъдат многобуквени. Например `abc(abc,Abc)` е терм, в който първото `abc` е двуместният функционален символ `abc`, второто `abc` е символът за константа `abc`, а `Abc` е променлива.

В езика пролог не се прави разлика между символи за константи, функционални символи и предикатни символи, а атомарните формули са просто вид термове.

Списъци

Списъците са една от най-важните структури данни в програмирането. Ако представим списъка `[1,2,3,4,5]` във вид на свързан списък, в паметта на компютъра ще се образува структура, която донякъде наподобява синтактичното дърво, показано във фигура 14. Като се вземе предвид това, че термовете са универсален тип данни, фактът, че можем да използваме синтактични дървета (т.е. термове), за да представяме списъци, никак не е изненадващ.

Следователно ако искаме да използваме списъка `[1,2,3,4,5]` на пролог, вместо него можем да използваме терма

```
•(1,•(2,•(3,•(4,•(5,[])))))
```

В този терм `[]` е символ за константа, представящ списъка с нула елементи (т.н. *празен списък*),* а $\bullet$ е двуместен функционален символ като смисълът на $\bullet(A, X)$ е списък, чийто пръв елемент е A, а X е списък от всички елементи без първия.

Първият елемент на даден списък се нарича *глава* на списъка, а списъкът от всички елементи без първия — *опашка*. Например 1 е глава на списъка `•(1,•(2,•(3,•(4,•(5,[])))))`, а `•(2,•(3,•(4,•(5,[]))))` — опашка.

Използването на този синтаксис за работа със списъци е възможно, но неудобно. Затова в пролог е предвидено следното улеснение: вместо да пишем списъци като `•(1,•(2,•(3,•(4,•(5,[])))))`, можем да използваме далеч по-четливия запис `[1,2,3,4,5]`. Този начин за записване на списъци обаче не е нищо повече от синтактична захар, тъй като не добавя никакви нови възможности към езика, а само прави програмите по-кратки и по-четливи, улеснявайки по този начин живота на програмистите.

Вместо да пишем $\bullet(\alpha, \chi)$, пролог позволява да използваме записа `[`α`|`χ`]`. Тук α е главата, т.е. първият елемент на `[`α`|`χ`]`, а χ — опашката. Освен това можем да пишем и изрази като `[`$\alpha_1, \alpha_2, \alpha_3$`|`$\chi$`]` вместо $\bullet(\alpha_1, \bullet(\alpha_2, \bullet(\alpha_3, \chi)))$. Тук α_1 е първият елемент на `[`$\alpha_1, \alpha_2, \alpha_3$`|`$\chi$`]`, α_2 — вторият елемент, α_3 — третият, а χ е списъкът от всички елементи от четвъртия нататък.**

Ще казваме, че даден запис на спъсък на пролог е *ненормален*, ако в него скобата `[` се среща непосредствено след вертикална черта `|`. В противен случай записът е *нормален*. Винаги има начин да трансформираме един ненормален запис на списък в нормален. Например списъкът `[1|[2|[]]]` е равен на `[1,2]`, а списъкът `[[1,2]|[3,4]]` е равен на `[[1,2],3,4]`. Нарисувайте синтактични дървета, за да се убедите, че това е така.***

***Задача* 49:** Уверете се, че термът

`•(•(•(1,[]),•(2,[])),•(•(3,•(4,[])),•(5,•(6,[]))))`

*Тук се вижда разликата между „константа“ и „символ за константа“. Празният списък е константа (в случая нещо, което може би се намира някъде в паметта на компютъра), а `[]` е символ (т.е. означение) за тази константа.

**Ако списъкът съдържа само трите елемента α_1, α_2 и α_3, то χ е празният списък.

***Нормалният запис на списъците е много по-четлив, така че никога няма смисъл да записваме списъците по ненормален начин. Затова, когато на писмения изпит някоя програма на пролог съдържа ненормални списъци, проверяващите започват да стават подозрителни.

е същият като списъка с ненормален запис

```
[[[1|[]]|[2|[]]]|[[3|[4|[]]]|[5|[6|[]]]]]
```

Нарисувайте синтактично дърво за този списък. Как можем да го запишем по нормален начин? Забележете колко по-четлив е нормалният запис.

Оператори

> Операторите се използват с цел да направят програмите по-четливи. Хората са доста гъвкави и могат да се приспособят към странни среди — например те могат да свикнат да четат програми на лисп и фортран. Въпреки това, ние считаме, че синтаксисът е важен; силата на добрата нотация е добре известна в математиката. Съществена част от хубавия синтаксис на пролог е възможността да се описват и използват оператори.
>
> Лион Стерлинг и Ехуд Шапиро [19]

Да си припомним, че когато математиците напишат терм във вида $3/4+(2-4*5)$, те всъщност имат предвид терма $+(/(3,4),-(2,*(4,5)))$. И пролог прави същото — изразът `3/4+(2-4*5)` ще бъде изтълкуван така, сякаш вместо това сме записали `+(/(3,4),-(2,*(4,5)))`.*

Нещо интересно обаче, което отличава пролог от почти всички останали езици за програмиране, е това, че пролог позволява на програмиста да дефинира и много други оператори. Използвайки тази възможност, на пролог могат да се дефинират езици с доста сложен синтаксис. Например в приложение Б е дадена пълната реализация на един императивен език за програмиране, наречен ИМПЕРАТОР, като това е така направено, че ИМПЕРАТОР да стане част от езика пролог. Ето как например може да дефинираме предикат, за пресмятането на факториела на дадено число:

```
% по дадено естествено число N записва в F неговия факториел
факториел(N,F) :- император
   f := 1;
   for i in 2..N loop
      f := f * i;
   F <== f.
```

*Личи си, че пролог е измислен във Франция, нали?

Дефинирането на нови оператори на пролог става посредством вградения предикат `op`. Например след изпълнението на*

```
:- op(700,xf, [е_животно]).
:- op(700,xfx,[има]).
:- op(600,fy, [бащата_на]).
```

пролог ще интерпретира изразите

```
бащата_на лиско
лиско е_животно
бащата_на лиско е_животно
бащата_на бащата_на лиско има козина
```

все едно, че сме написали термовете

```
бащата_на(лиско)
е_животно(лиско)
е_животно(бащата_на(лиско))
има(бащата_на(бащата_на(лиско)), козина)
```

Тук числото, което даваме като пръв аргумент на `op`, е приоритетът на оператора като колкото по-малко е числото, толкова по-голям е приоритетът. Да забележим, че ако на оператора `бащата_на` дадем приоритет 800 вместо 600, то пролог ще интерпретира третия от горните изрази като

```
бащата_на(е_животно(лиско))
```

което, разбира се, е правилен терм, но лишен от смисъл.

Вторият аргумент на `op` определя типа на оператора. Съществуват следните типове:

fx Едноместни префиксни оператори, в които аргументът не може да има същият приоритет. Например ако дефинираме оператор `вали` от тип `fx`, то изразът `вали дъжд` ще се интерпретира като `вали(дъжд)`, а `вали вали дъжд` ще бъде невалиден израз.

fy Едноместни префиксни оператори, в които аргументът може да има същият приоритет. Например `бащата_на бащата_на лиско`.

xf *и* ***yf*** Аналогично на `fx` и `fy`, но за постфиксни оператори. Например `лиско е_животно`.

*Не всяка реализация на пролог позволява директното използване на кирилица във функционалните символи, символите за константи и променливите. SWI-пролог е една от най-популярните реализации на пролог и тя позволява това. Вижте `http://www.swi-prolog.org`. Кирилица може да се използва и при строубери пролог. Вижте `http://dobrev.com`.

оператор	*тип*	*приоритет*
унарни + и -	`fy`	200
^	`xfy`	200
* и /	`yfx`	400
бинарни + и -	`yfx`	500
= и <	`xfx`	700
,	`xfy`	1000
;	`xfy`	1100
:- и ?-	`xfx`	1200

Таблица 4. Приоритет и тип на някои от предефинираните оператори на пролог

yfx Лявоасоциативни инфиксни оператори. Например изразът `2+3+4` се интерпретира като `(2+3)+4`.

xfy Дясноасоциативни инфиксни оператори. Например изразът `2^3^4` се интерпретира като `2^(3^4)`.

xfx Двуместни инфиксни оператори, при които нито отляво, нито отдясно се позволяват оператори със същия приоритет. Например изразите 2=3=4 и 2<3<4 са недопустими, защото операторите = и < са от тип `xfx`.

При определяне на приоритета на операторите е добре да се имат предвид приоритетите на операторите, които в пролог са предефинирани. По-важните от тях са дадени в таблица 4.

Операторът запетая в пролог е особен, защото се използва хем като инфиксен оператор (като смисълът му е конюнкция), хем като разделител между аргументите. Например ако искате да кажете, че не е вярно, че „5=5 и 3=4“ и напишете израза `not(5=5,3=4)`, пролог ще се оплаче, че не е ясен смисълът на `not`, когато се използва с два аргумента. Проблемът се решава с една двойка допълнителни скоби: `not((5=5,3=4))`.

***Задача* 50:** В пролог `not` е обикновен функционален символ, а не оператор. В следствие на това трябва да пишете `not(5=4)` вместо `not 5=4` и `not((5=5,3=4))` вместо `not (5=5,3=4)`. Обаче, ако дефинирате `not` като оператор, изразите `not 5=4` и `not (5=5,3=4)` ще станат коректни.* Как можете да направите това?

***Задача* 51:** Дефинирайте на пролог оператори `и`, `или`, `не`, благодарение на които ще можем да програмираме на български например така:

*Забележете интервала между `not` и отворената скоба.

```
ремонт(Кола) :-
   не добри_спирачки(Кола) или
   шумна(Кола) или пуши(Кола) или
   външна_повреда(Кола,Повреда) и не драскотина(Повреда) или
   течове(Кола, Количество) и Количество > 194.8.
```

А.2. Термовете във функционалните езици

Функционалните езици образуват две големи семейства — семейството на диалектите на лисп, към което спада напр. скийм, и семейството на статичнотипизираните функционални езици като хаскел, о-камъл, скала и кложър.

Също както при пролог в общи линии всички неща са термове, така на лисп пък всички неща са списъци. За списъците се използва следният синтаксис: $(\alpha_1\ \alpha_2\ \ldots\ \alpha_n)$. С други думи забелязват се следните разлики спрямо синтаксиса, използван от пролог: списъците се заграждат в кръгли, а не в квадратни скоби и между елементите не се поставят запетаи, а интервали.

Въпреки че лисп няма директна поддръжка за термове, оказва се, че никак не е трудно да използваме списъци, за да записваме термове — това става като вместо терма $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ използваме списъка $(\mathtt{f}\ \tau_1\ \tau_2\ \ldots\ \tau_n)$. С други думи първият елемент на списъка показва кой е функционалният символ, а следващите елементи са аргументите. Например термът $\mathtt{f}(\mathtt{g}(\mathtt{a}, \mathtt{b}), \mathtt{c})$ може да бъде записан на лисп така:

```
(f (g a b) c)
```

В раздел 2.7 видяхме, че променливите — както в математиката, така и в програмирането — биват два вида: свободни и свързани. Термовете във функционалните езици, и в частност при лисп и диалектите му, не могат да съдържат свободни променливи.

За сметка на това обаче диалектите на лисп позволяват да се използват т.н. ламбда-термове, които съдържат свързани променливи. Например на скийм

```
(lambda (f g) (lambda (x) (f (g x))
```

е ламбда-терм, съдържащ трите свързани променливи `f`, `g` и `x`. Този ламбда-терм представя функцията, която взема като аргументи две функции `f` и `g` и връща като стойност тяхната композиция (което, раз-

бира се, също е функция). Математиците* записват този ламбда-терм така: $\lambda fg.\lambda x.f(gx)$.

На пролог свързани променливи няма.**

* * *

При другата голяма група от функционални езици — тази на статичнотипизираните езици като хаскел, о-камъл, скала и кложър — в следствие на строгата типизация не е удобно „да се хитрува“ като се използват списъци вместо термове. Затова тези езици имат по-непосредствена поддръжката за термове, отколкото лисп.

В много от статичнотипизираните функционални езици за термовете се използва същият синтаксис, както и при лисп, с тази разлика, че не е задължително да пишем най-външните кръгли скоби (въпреки това в примерите по-долу най-външните скоби са слагани). Например `(f` α_1 α_2 `...`α_n`)` е термът с функционален символ `f` и аргументи $\alpha_1, \alpha_2, \ldots, \alpha_n$.

Да допуснем, че искаме да разполагаме с функционални символи двуместен `Point`, двуместен `Rectangle`, едноместен `Circle` и триместен `Positioned_figure`.

Смисълът на `(Point` x y`)` е точка с координати (x, y).

Смисълът на `(Rectangle` a b`)` е геометричната фигура правоъгълник със страни a и b като позицията на правоъгълника не е фиксирана.

Смисълът на `(Circle` r`)` е геометричната фигура окръжност с радиус r като позицията на окръжността не е фиксирана.

Смисълът на `(Positioned_figure` A φ σ`)` е геометричната фигура σ (която може да бъде правоъгълник или окръжност) поставена в точката A и завъртяна на ъгъл φ.***

Например

```
(Positioned_figure (Point 1 5) 30 (Rectangle 3 2))
```

представлява правоъгълник със страни 3 и 2, поставен в точка с координати $(1, 5)$ и завъртян на ъгъл 30.****

Преди да можем да дефинираме тези функционални символи, трябва да определим какви типове използват те. Забелязваме, че се използват общо четири типа:

*И по-точно Аланзо Чърч през 1932 г.

**Свързани променливи има например в езика ламбда-пролог.

***Разбира се, да въртим окръжност няма смисъл. Обаче в един по-реалистичен пример освен правоъгълника със сигурност би имало и много други форми, които не са ротационно инвариантни.

****Каквото и да значи изразът „завъртян на ъгъл 30“. Много самолети и ракети са падали заради това, че една част от програмата използва радиани, а друга градуси или едната метри, а другата футове.

Float Т.е. реално число. Този тип почти винаги е вграден в езиците и не е нужно да го дефинираме.

Point_type Стойността на функционалния символ `Point` и първият аргумент на `Positioned_figure` има този тип.

Figure Стойността на `Rectangle` и `Circle` има този тип.

Graphic_object Това е типът на стойността на `Positioned_figure`.

След като сме решили кои са функционалните символи и какви са техните типове, можем да ги дефинираме формално. Забележете, че дефиницията на функционалните символи е част от дефиницията на типовете.

Дефиницията на хаскел изглежда така:

```
data Point_type = Point Float Float
data Figure =
   | Rectangle Float Float
   | Circle Float
data Graphic_object = Positioned_figure Point_type Float Figure
```

Същата дефиницията на о-камъл изглежда така (на о-камъл типовете се пишат с малки букви):

```
type point_type = Point of float * float
type figure =
   Rectangle of float * float
   | Circle of float
type graphic_object =
   Positioned_figure of point_type * float * figure
```

Приложение Б

Реализация на език за програмиране ИМПЕРАТОР

В това приложение ще видим как можем да напишем на пролог интерпретатор на прост императивен език за програмиране и то така, че този език да стана част от пролог. Понеже всеки език за програмиране трябва да си има име, нека дадем на нашия език името ИМПЕРАТОР.

Най-напред ще определим синтаксиса на ИМПЕРАТОР. Всяка програма на ИМПЕРАТОР ще се състои просто от ключовата дума `император`, следвана от списък от команди, разделени със символа точка и запетая. Всяка команда ще има един от следните пет вида.

Първо, оператор за присвояване. Отляво трябва да стои идентификатор, който си го мислим като име на променлива на ИМПЕРАТОР:

```
променлива := израз
```

Второ, присвояване на стойност на променлива на пролог. Отляво трябва да стои име на променлива на пролог, която все още не е получила стойност:

```
променлива <== израз
```

Трето, изпълнение на предикат на пролог (също както в програма на пролог можем да изпълняваме код, написан на ИМПЕРАТОР, така и в програма на ИМПЕРАТОР можем да извикваме предикати на пролог):

```
пролог p(T1,T2,...,Tn)
```

Четвърто, оператор за цикъл while. Ще считаме, че цикълът се изпълнява докато стойността на контролиращия израз е различна от нула:

```
while израз loop (
   оператор 1;
   оператор 2;
   ...
   оператор n
)
```

Пето, оператор за цикъл for:

```
for i in A..B loop (
   оператор 1;
   оператор 2;
   ...
   оператор n
)
```

Шесто, условен оператор. Приемаме, че изразът е истина, ако стойността му е различна от нула:

```
if израз then (
   оператор 1;
   оператор 2;
   ...
   оператор n
)
```

или

```
if израз then (
   оператор 1;
   оператор 2;
   ...
   оператор n
) else (
   оператор 1;
   оператор 2;
   ...
   оператор m
)
```

Използвайки този език, ето как можем да дефинираме на пролог предикат за пресмятане на факториела на едно число:

```
% по дадено естествено число N записва в F неговия факториел
факториел(N,F) :- императоp
   f := 1;
   for i in 2..N loop (
      f := f * i
   );
   F <== f.
```

В последния ред от тази програма присвояваме на променливата с име F на пролог стойността, която в този момент променливата с име f на ИМПЕРАТОР има.

Пълната реализация на ИМПЕРАТОР е дадена малко по-нататък. Ще коментираме някои по-трудни за разбиране неща. Първо, да видим как е реализиран синтаксисът на ИМПЕРАТОР.

Операторите `император`, `:=`, `<==` и `пролог` са дефинирани така:

```
:- op(1150, fx, [император]).
:- op(990, xfx, [':=', '<==']).
:- op(990, fx, [пролог]).
```

Тук приоритетът 1150 е подбран така, че да бъде по-малко число (т.е. по-голям приоритет) от приоритета на `:-`, но по-голямо число от приоритета на оператора точка и запетая. Първото ни гарантира, че няма нужда да ограждаме програмата на ИМПЕРАТОР в скоби така: `p :- (император ...)`. Второто ни позволява да не слагаме скоби след `император` само заради това, че в програмата се съдържат команди, разделени с точка и запетая ето така: `p :- император (... ; ...)`.*

Приоритетът 990 е подбран така, че бъде по-голямо число (т.е. по-малък приоритет) от приоритетите на всички аритметични операции в пролог, но по-малко число, от приоритета на оператора точка и запетая. Първото ни гарантира, че няма нужда да пишем скоби в израза след `:=` и `<==`, например `i := (1+2)`. Второто пък означава, че няма нужда да пишем скоби около самия оператор за присвояване, например `(i:=1);(f:=1)`.

Синтаксисът на оператора за цикъл while на ИМПЕРАТОР използва две различни ключови думи — `while` и `loop`. Всяка една от тях трябва

*Обаче скоби около програмата на ИМПЕРАТОР все пак трябва да се поставят, ако тя не е единственото нещо, което се прави в предиката на пролог:

```
p(X) :- q(X),
        (император
            тук е програмата на ИМПЕРАТОР
        ),
        r(X).
```

да бъде дефинирана като префиксен, постфиксен или инфиксен оператор на пролог със свой собствен приоритет. Един начин това да бъде направено е да дефинираме `while` като префиксен оператор, а `loop` — като инфиксен с два аргумента. Ще дадем на `while` по-голям приоритет, отколкото на `loop`, така че в израза

`while` израз `loop (` команди `)`

пролог да постави скобите по следния начин:

`(while` израз `) loop (` команди `)`

Записан като терм същият цикъл изглежда така:

`loop(while(`израз`),` команди`)`

Ще изберем така приоритетите на `while` и `loop`, че те да бъдат по-големи от приоритета на оператора точка и запетая, но по-малки от приоритетите на операторите `:=`, `<==` и `пролог`:

```
:- op(994, fx, [while]).
:- op(995, xfx, [loop]).
```

Операторът за цикъл for използва операторите `for`, `in`, `..` и `loop`. Така ще ги дефинираме, че в израза

`for` i `in` a`..`b `loop (` команди `)`

пролог да постави скобите по следния начин:

`(for (`i `in (`a`..`b`))) loop (` команди `)`

Записан като терм същият цикъл изглежда така:

`loop(for(in(i,(a..b))),` команди`)`

Ще изберем така приоритетите на `for`, `in` и `..`, че те да бъдат по-големи от приоритета на `loop` и по-малки от приоритетите на аритметичните оператори:

```
:- op(994, fx, [for]).
:- op(993, xfx, [in]).
:- op(992, xfx, ['..']).
```

Условният оператор на ИМПЕРАТОР използва две или три ключови думи — `if`, `then` и може би също `else`:

```
:- op(995, fx, [if]).
:- op(993, xfx, [then]).
:- op(994, xfx, [else]).
```

Така избраните приоритети означават, че пролог ще интерпретира изразите

```
if израз then ( команди )
if израз then ( команди1 ) else ( команди2 )
```

като термовете

```
if(then(израз,команди))
if(else(then(израз,команди1),команди2)
```

За разлика от пролог, на ИМПЕРАТОР променливите могат да си променят стойността. Това означава, че не можем да помним стойността на променливите на ИМПЕРАТОР в променливи на пролог. Вместо това ще използваме базата знания на пролог.

Да припомним, че вграденият предикат `assert` може да се използва, за да добавим факти към базата знания на пролог. Например след изпълнението на `assert(a(5))` пролог „ще знае“, че `a(5)` е вярно. Предикатът `retract` пък прави обратното, например след изпълнението на `retract(a(5))` пролог „ще забрави“, че `a(5)` е вярно.

За да запомним, че стойността на променливата `x` е 5, ще добавим към базата знания на пролог факт от вида

```
стойност_на_променлива(x, 5)
```

За да дадем на променливата `x` нова стойност, например 7, първо трябва да премахнем от базата знания старата стойност, т.е. факта, имащ вида `стойност_на_променлива(x, _)` и едва след това да добавим новия факт `стойност_на_променлива(x, 7)`.

Следва пълната реализация на ИМПЕРАТОР:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% © 2015 Антон Зиновиев
%
% Тази програма може бъде използвана без ограничения от
% всеки, който е получил нейно копие, в това число да бъде
% разпространявана, изменяна, лицензирана и прелицензирана
% при условие, че тази бележка за авторските права не се
% трие.  ПРОГРАМАТА СЕ ПРЕДОСТАВЯ БЕЗ КАКВИТО И ДА Е
```

```
% ГАРАНЦИИ, ЧЕ СТАВА ЗА НЕЩО СМИСЛЕНО И НЯМА ДА НАВРЕДИ.
% Без изрично разрешение, имената на носителите на
% авторските права не могат да бъдат използвани за
% популяризиране на продукти, базирани на тази програма.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(1150, fx, [император]).
:- op(990, xfx, [':=', '<==']).
:- op(990, fx, [пролог]).

:- op(994, fx, [while]).
:- op(995, xfx, [loop]).

:- op(994, fx, [for]).
:- op(993, xfx, [in]).
:- op(992, xfx, ['..']).

:- op(995, fx, [if]).
:- op(993, xfx, [then]).
:- op(994, xfx, [else]).

%% стойност(Израз, Стойност) -- пресмята в Стойност
%%            стойността на Израз
стойност(Пром, Стойност) :-
        atom(Пром),
        стойност_на_променлива(Пром, Стойност).
стойност(Число, Число) :-
        number(Число).
стойност(A + B, Z) :-
        стойност(A, X), стойност(B, Y),
        Z is X + Y.
стойност(A - B, Z) :-
        стойност(A, X), стойност(B, Y),
        Z is X - Y.
стойност(A * B, Z) :-
        стойност(A, X), стойност(B, Y),
        Z is X * Y.
стойност(A / B, Z) :-
        стойност(A, X), стойност(B, Y),
        Z is X / Y.
стойност(A = B, Z) :-
```

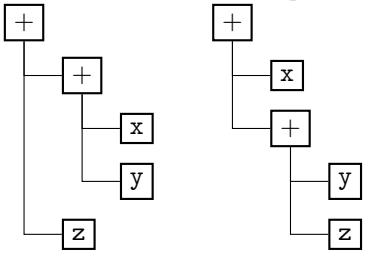
```
        стойност(A, X), стойност(B, Y),
        (X = Y -> Z = 1 ; Z = 0).
стойност(A\= B, Z) :-
        стойност(A, X), стойност(B, Y),
        (X \= Y -> Z = 1 ; Z = 0).
стойност(A < B, Z) :-
        стойност(A, X), стойност(B, Y),
        (X < Y -> Z = 1 ; Z = 0).
стойност(A > B, Z) :-
        стойност(A, X), стойност(B, Y),
        (X > Y -> Z = 1 ; Z = 0).
стойност(A =< B, Z) :-
        стойност(A, X), стойност(B, Y),
        (X =< Y -> Z = 1 ; Z = 0).
стойност(A >= B, Z) :-
        стойност(A, X), стойност(B, Y),
        (X >= Y -> Z = 1 ; Z = 0).

император X ; Y :-
        (император X), (император Y).
император V := E :-
        стойност(E, EVal), !,
        (
         %% премахваме старата стойност на V
         retract(стойност_на_променлива(V, _))
        ;
         true
        ), !,
        %% добавяме новата стойност
        assert(стойност_на_променлива(V, EVal)).
император V <== E :-
        стойност(E, V), !.
император пролог Цел :-
        call(Цел).
император if E then X :-
        стойност(E, EVal), !,
        (EVal \= 0 -> (император X) ; true).
император if E then X else Y :-
        стойност(E, EVal), !,
        (EVal \= 0 -> (император X) ; (император Y)).
император while E loop X :-
```

```
        стойност(E, EVal), !,
        (
         EVal \= 0
        ->
         (императ X), !, (императ while E loop X)
        ;
         true
        ).
императ for I in A..B loop X :- императ
        I := A;
        while I =< B loop (
           X;
           I := I + 1
        ).
```

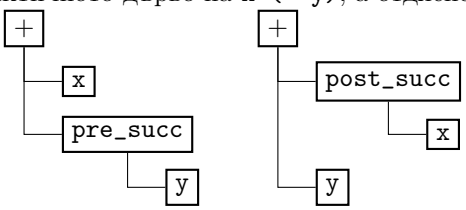
Приложение B

Решения на задачите

Задача 1. Отляво е синтактичното дърво на `(x+y)+z`, `x+y+z`, `((x+y))+z` и `(x)+y+z`, а отдясно — синтактичното дърво на `x+(y+z)`:



Задача 2. Тъй като на си има префикснен `++` и постфиксен `++`, за префиксния ще използваме етикет `pre_succ`, а за постфиксния — `post_succ`. В такъв случай отляво е синтактичното дърво на `x+(++y)`, а отдясно — на `(x++)+y`:

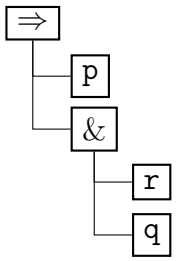


Компилаторите на си ще интерпретират `x+++y` като `(x++)+y`,* но не знам дали това е произволно решение или е предписание на стандарта на си.

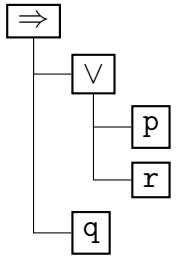
Задача 3. Тъй като изразът (p⇔p) е съкратен запис за формулата ((p⇒p)&(p⇒p)), то значи тази формула съдържа 13 символа.

Задача 4. Първата формула е $(\mathtt{p} \Rightarrow (\mathtt{r} \,\&\, \mathtt{q}))$, а синтактичното ѝ дърво е изобразено във фиг. 15. Втората формула е $((\mathtt{p} \vee \mathtt{r}) \Rightarrow \mathtt{q})$, а синтактичното ѝ дърво е изобразено

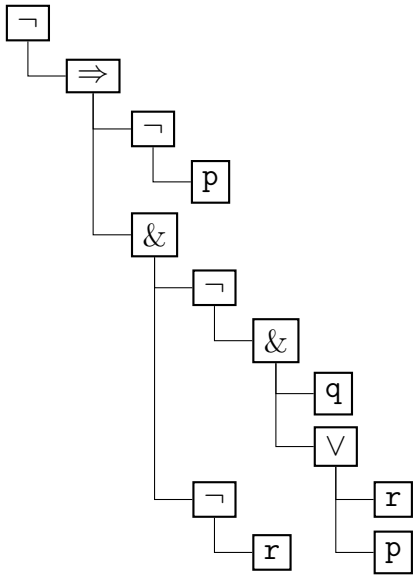
*И също `x+++++y` като `((x++)++)+y`.



Фиг. 15. Синтактично дърво за $\mathtt{p} \Rightarrow \mathtt{r} \,\&\, \mathtt{q}$



Фиг. 16. Синтактично дърво за $\mathtt{p} \vee \mathtt{r} \Rightarrow \mathtt{q}$



Фиг. 17. Синтактично дърво за $\neg(\neg\mathtt{p} \Rightarrow \neg(\mathtt{q} \,\&\, (\mathtt{r} \vee \mathtt{p})) \,\&\, \neg\mathtt{r})$

във фиг. 16. Третата формула е $(\neg(\neg\mathtt{p} \Rightarrow (\neg(\mathtt{q} \,\&\, (\mathtt{r} \vee \mathtt{p})) \,\&\, \neg\mathtt{r})))$, а синтактичното ѝ дърво е изобразено във фиг. 17.

Задача 5. Първото разсъждение по първо правило, второто по второ, третото по трето, четвъртото по четвърто и петото по пето.

Задача 6. `f` е двуместен функционален символ, `g` и `h` са едноместни функционални символи, `c` и `a` са символи за константи, а `x` и `y` са променливи.

Задача 7. $\mathtt{f(c)}$ е терм, $\mathtt{c(f)}$ не е терм, защото константата $\mathtt{c}$ не може да има аргументи, а функционалният символ $\mathtt{f}$ трябва да има аргументи, $\mathtt{c}$ е терм, $\mathtt{f}$ не е терм, защото функционалният символ $\mathtt{f}$ трябва да има поне един аргумент, $\mathtt{x(c)}$ не е терм, защото променливите не могат да имат аргументи, $\mathtt{f(f(x))}$ е терм, $\mathtt{f(f(f(c)))}$ е терм, $\mathtt{f(f(f(c,c)))}$ не е терм, защото $\mathtt{f}$ не може да бъде едновременно едноместен и двуместен, $\mathtt{g(g(x,x),g(x,y))}$ е терм.

Задача 8. $\mathtt{c}$ е символ за константа и значи терм, а не атомарна формула, $\mathtt{f(c)}$ е терм без променливи, а не атомарна формула, $\mathtt{p(c)}$ и $\mathtt{p(f(f(f(c))))}$ са атомарни формули без променливи, $\mathtt{p(f(x))}$ е атомарна формула, която съдържа променливи, $\mathtt{f(p(x))}$ и $\mathtt{p(p(x))}$ не са нито термове, нито атомарни формули, $\mathtt{f(f(x))}$ не е атомарна формула, а терм, който съдържа променливи.

Задача 9. Тъй като $\mathtt{c}$ е символ за константа константа, от правило 2.4.3 б) получаваме, че $\mathtt{c}$ е терм.

Тъй като $\mathtt{x}$ е променлива, от правило 2.4.3 а) получаваме, че $\mathtt{x}$ е терм.

Тъй като вече сме доказали, че $\mathtt{c}$ и $\mathtt{x}$ са термове, а $\mathtt{f}$ е двуместен функционален символ, от правило 2.4.3 в) получаваме, че $\mathtt{f(c,x)}$ е терм.

Тъй като вече сме доказали, че $\mathtt{c}$ и $\mathtt{f(c,x)}$ са термове, а $\mathtt{f}$ е двуместен функционален символ, от правило 2.4.3 в) получаваме, че $\mathtt{f(c,f(c,x))}$ е терм.

Задача 10. Термът $\mathtt{f(c)}$ съдържа скоби, но не съдържа запетаи. Няма термове, които съдържат запетаи, но не съдържат скоби. Във всички термове има равен брой леви и десни скоби.

Задача 11. Например $\mathtt{f(c,c,c,c,c,c,c)}$ и $\mathtt{g(g(h(c,c,c)))}$. Във втория от тези два терма може ли да заменим $\mathtt{h}$ с $\mathtt{g}$?

Задача 12. Ако заменим константата с $\mathtt{c}$, едноместния функционален символ с $\mathtt{f}$ и двуместния с $\mathtt{g}$, то тези изрази ще добият следния вид:

$$\boxed{\mathtt{c}}\ \boxed{\mathtt{f(c)}}\ \boxed{\mathtt{f(f(c))}}\ \boxed{\mathtt{g(c,c)}}\ \boxed{\mathtt{g(f(c),g(c,c))}}$$

Задача 13. Термът $+(\mathtt{c},*(\mathtt{x},\mathtt{a}))$ съдържа 4 скоби — две леви и две десни.

Задача 14. С индукция по построението на терма ще докажем, че всеки терм съдържа поне една променлива.

а) Ако $\mathtt{x}$ е променлива, то очевидно $\mathtt{x}$ съдържа променлива.

б) Символи за константи няма.

в) Нека $\mathtt{f}$ е n-местен функционален символ и $\tau_1, \tau_2, \ldots, \tau_n$ са термове, всеки от които съдържа поне една променлива.[*] В такъв случай очевидно и термът $\mathtt{f}(\tau_1, \tau_2, \ldots, \tau_n)$ ще съдържа поне една променлива (тук неявно използваме, че $n \geqq 1$).

Задача 15. а) Ако $\mathtt{x}$ е променлива, то $\mathtt{x}$ съдържа равен брой запетаи и десни скоби (нула), защото променливите не са запазени символи, а запетаята и дясната скоба са.

б) Ако $\mathtt{c}$ е символ за константа, то $\mathtt{c}$ съдържа равен брой запетаи и десни скоби (нула), защото символите за константи не са запазени символи, а запетаята и дясната скоба са.

в) Нека $\mathtt{f}$ е функционален символ (по услови той трябва да бъде двуместен). Да допуснем, че τ_1 и τ_2 са термове с равен брой запетаи и десни скоби.[**] Нека k_i е

[*] Това е индукционното предположение.

[**] Това е индукционното предположение.

броят на запетаите в τ_i (същото число е равно и на броя на десните скоби). Тъй като $\mathtt{f}$ не е запазен символ, то $\mathtt{f}$ не запетая или скоба. Следователно броят на запетаите в терма $\mathtt{f}(\tau_1, \tau_2)$ е $1 + k_1 + k_2$ — една запетая между τ_1 и τ_2 плюс запетаите в τ_1 и τ_2. Броят на десните скоби е същият — затворената скоба в края на терма плюс скобите в τ_1 и τ_2.

Задача 16. Индуктивният принцип казва следното:

> Нека $\mathfrak{p}$ е свойство, за което е вярно, че:
>
> 1. свойството $\mathfrak{p}$ е вярно за 5;
> 2. свойството $\mathfrak{p}$ е вярно за 8;
> 3. ако свойството $\mathfrak{p}$ е вярно за буртантите n и m, то то ще бъде вярно и за nm^2.
>
> Тогава свойството $\mathfrak{p}$ ще бъде вярно за всички буртанти.

Ще използваме този индуктивен принцип в случая, когато $\mathfrak{p}$ е свойството, че числото събрано с 1 се дели на 3 без остатък. За целта трябва да докажем, че са верни трите изисквания в индуктивния принцип за буртанти.

1. Очевидно 5 събрано с 1 се дели на 3 без остатък.
2. Очевидно 8 събрано с 1 се дели на 3 без остатък.
3. Да допуснем, че n и m са такива буртанти, че $n+1$ и $m+1$ се делят на 3 без остатък.* Нека $n + 1 = 3k$ и $m + 1 = 3l$. Тогава

$$\begin{aligned} nm^2 + 1 &= ((n+1) - 1)((m+1) - 1)^2 + 1 \\ &= (3k - 1)(3l - 1)^2 \\ &= 3k(3l - 1)^2 - (3l - 1)^2 + 1 \\ &= 3k(3l - 1)^2 - 9l^2 + 6l - 1 + 1 \\ &= 3(k(3l - 1)^2 - 3l^2 + 2l) \end{aligned}$$

   Следователно $nm^2 + 1$ се дели на три без остатък.

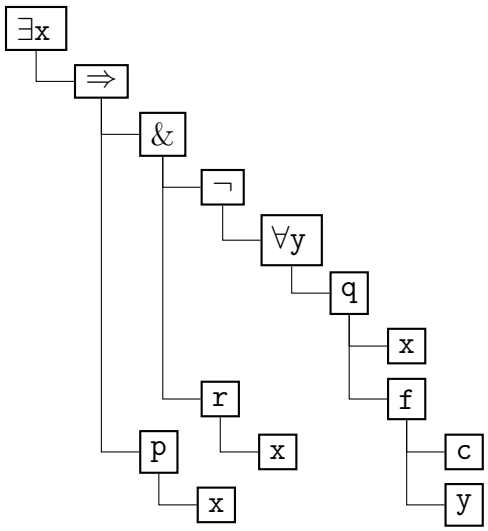
Доказахме, че трите условия от принципа за индукция за буртанти са изпълнени, от където следва, че всички буртанти събрани с единица се делят на 3 без остатък.

Задача 17. Вж. фиг. 18.

Задача 18. В следващия списък са подчертани подформулите:

$$\begin{aligned} &\underline{\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))} \\ &\forall \mathtt{x}\,\underline{(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x}))} \\ &\forall \mathtt{x}\,(\underline{\mathtt{p}(\mathtt{x})} \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x})) \\ &\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \underline{\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \mathtt{r}(\mathtt{x})}) \\ &\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \underline{\exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y}))} \,\&\, \mathtt{r}(\mathtt{x})) \\ &\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\underline{\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y}))} \,\&\, \mathtt{r}(\mathtt{x})) \\ &\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \mathtt{y})) \,\&\, \underline{\mathtt{r}(\mathtt{x})}) \end{aligned}$$

*Това че $n+1$ и $m+1$ се делят на 3 без остатък е индукционното предположение.



Фиг. 18. Синтактично дърво за формулата .
$\exists \mathtt{x}\,(\neg\forall \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}))$

В следващия списък са подчертани термове, които също са са поддървета на синтактичното дърво от фигура 9, но не са подформули, защото са термове:

$$\forall \mathtt{x}\,(\mathtt{p}(\underline{\mathtt{x}}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\underline{\mathtt{x}},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\underline{\mathtt{f}(\mathtt{c},\mathtt{y})})\ \&\ \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\underline{\mathtt{c}},\mathtt{y}))\ \&\ \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\underline{\mathtt{y}}))\ \&\ \mathtt{r}(\mathtt{x}))$$
$$\forall \mathtt{x}\,(\mathtt{p}(\mathtt{x}) \Rightarrow \exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\underline{\mathtt{x}}))$$

Задача 19. Тук е подчертана областта на действие на първия квантор:

$$\underline{\forall \mathtt{x}\,(\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))}$$

Тук променливите, управлявани от първия квантор са подчертани:

$$\forall \underline{\mathtt{x}}\,(\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\underline{\mathtt{x}}) \Rightarrow \mathtt{p}(\underline{\mathtt{x}},\mathtt{y}))$$

Тук е подчертана областта на действие на втория квантор:

$$\forall \mathtt{x}\,(\underline{\forall \mathtt{x}\,\exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))}\ \&\ \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Тук променливите, управлявани от втория квантор са подчертани:

$$\forall \mathtt{x}\,(\forall \underline{\mathtt{x}}\,\exists \mathtt{y}\,\mathtt{q}(\underline{\mathtt{x}},\mathtt{f}(\mathtt{c},\mathtt{y}))\ \&\ \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Тук е подчертана областта на действие на третия квантор:

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\underline{\exists \mathtt{y}\,\mathtt{q}(\mathtt{x},\mathtt{f}(\mathtt{c},\mathtt{y}))}\ \&\ \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x},\mathtt{y}))$$

Тук променливите, управлявани от третия квантор са подчертани:

$$\forall \mathtt{x}\,(\forall \mathtt{x}\,\exists \underline{\mathtt{y}}\,\mathtt{q}(\mathtt{x}, \mathtt{f}(\mathtt{c}, \underline{\mathtt{y}})) \,\&\, \mathtt{r}(\mathtt{x}) \Rightarrow \mathtt{p}(\mathtt{x}, \mathtt{y}))$$

Последната променлива y е единствената свободна променлива, а всички други променливи са свързани. Множеството от свободните променливи на тази формула е $\{\mathtt{y}\}$.

Задача 20. **(а)** Да. Можем да използваме следните еднократни преименувания:

$$\forall \mathtt{x}\,\forall \mathtt{y}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x}, \mathtt{y})$$
$$\forall \mathtt{x}\,\forall \mathtt{z}\,\forall \mathtt{z}\,\mathtt{p}(\mathtt{x}, \mathtt{z})$$
$$\forall \mathtt{x}\,\forall \mathtt{z}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x}, \mathtt{y})$$

(б) Да. Можем да използваме следните еднократни преименувания:

$$\forall \mathtt{x}\,\forall \mathtt{y}\,\forall \mathtt{y}\,\mathtt{p}(\mathtt{x}, \mathtt{y})$$
$$\forall \mathtt{x}\,\forall \mathtt{y}\,\forall \mathtt{z}\,\mathtt{p}(\mathtt{x}, \mathtt{z})$$
$$\forall \mathtt{x}\,\forall \mathtt{t}\,\forall \mathtt{z}\,\mathtt{p}(\mathtt{x}, \mathtt{z})$$
$$\forall \mathtt{y}\,\forall \mathtt{t}\,\forall \mathtt{z}\,\mathtt{p}(\mathtt{y}, \mathtt{z})$$
$$\forall \mathtt{y}\,\forall \mathtt{x}\,\forall \mathtt{z}\,\mathtt{p}(\mathtt{y}, \mathtt{z})$$
$$\forall \mathtt{y}\,\forall \mathtt{x}\,\forall \mathtt{x}\,\mathtt{p}(\mathtt{y}, \mathtt{x})$$

(в) Не. В първата от формулите първият аргумент на p се свързва от първия квантор, а във втората формула същият аргумент се свързва от втория квантор. Преименуванията на свързани променливи не могат да доведат до подобни изменения.

Задача 21. При всяко еднократно преименуване на свързана променлива, ние заменяме променливи с променливи, т.е. символи със символи. Това не променя броя на символите във формулата. Конгруентните формули се получават с няколко последователни еднократни преименувания, всяко от които не променя броя на символите.

Задача 22. Нека z е свободна променлива на φ и $\mathtt{z} \neq \mathtt{x}$. Да разгледаме някое свободно срещане на z в φ и съответното срещане на z в ψ. Ако допуснем, че срещането на z в ψ не е свободно, то значи това срещане в ψ ще попада в областта на действие на някой квантор $\forall \mathtt{z}$ или $\exists \mathtt{z}$. При преименуването обаче по никакъв начин не влияем на променливата z, следователно областта на действие на същия този квантор в φ ще бъде същата и значи ще включва и срещането на z, за което допуснахме, че е свободно. Това е противоречие.

Задача 23. Щом ψ и ψ' са конгруентни, то значи има редица от формули $\psi = \psi_0, \psi_1, \psi_2, \ldots, \psi_n = \psi'$, всяка от които се получава от предходната посредством еднократно преименуване на свързана променлива.

Да дефинираме формули $\varphi = \varphi_0, \varphi_1, \ldots, \varphi_n = \varphi'$ по следния начин: формулата φ_i се получава от φ като заменим подформулата ψ с ψ_i. Всяка от формулите ψ_i се получава от предходната като в областта на действие на някой квантор $\forall \mathtt{z}$ или $\exists \mathtt{z}$ заменим всяко срещане на z с някоя променлива $\mathtt{z}'$. Тъй като областта на

действие на един квантор е дефинирана като подформулата, започваща с този квантор и подформула на подформула е подформула, то значи подформулата, в която преименуваме $\mathtt{z}$ на $\mathtt{z}$ е също област на действие и в голямата формула φ_i. Доказахме, че всяка от формулите φ_i се получава от предходната посредством еднократно преименуване на свързана променлива.

Задача 25. Има редица от формули $\chi_0, \chi_1, \ldots, \chi_n$, която показва как с последователност от няколко еднократни преименувания на свързани променливи от $\forall \mathtt{x}\, \varphi$ можем да получим $\forall \mathtt{x}\, \psi$. От тази редица трябва да получим аналогична редица, само че за формулите φ и ψ. Да отстраним квантора, с който започва всеки член на редицата $\chi_0, \chi_1, \ldots, \chi_n$. Новополучената редица $\chi'_0, \chi'_1, \ldots, \chi'_k$ ще започва с φ и ще завършва с ψ. За съжаление обаче тя може и да не бъде редица от еднократни преименувания, защото ако някъде в първоначалната редица сме преименували променливата на началния квантор от $\mathtt{x}$ на някоя друга променлива, то няма как да направим аналогично преименуване след като махнем квантора, защото тази променлива ще стане свързана. За да елиминираме този проблем, нека просто в тези случаи в новата редица не преименуваме онези срещания на променливата $\mathtt{x}$, които повече не попадат в областта на действие на никой квантор — щом като накрая в ψ пак трябва да имаме $\mathtt{x}$, то няма смисъл някъде да заменим $\mathtt{x}$ с друга променлива и по-нататък да заменим обратно другата променлива пак с $\mathtt{x}$. Нека така получената редица е $\chi''_0, \chi''_1, \ldots, \chi''_m$; тук $m \leqq n$, защото в новата редица не правим някои от преименуванията.

Дали така получената редица ще се получава с еднократни преименувания на свързани променливи? За съжаление отново може и да не е така. Съгласно дефиницията, правим еднократни преименувания по следния начин — в някоя подформула от вида $\forall \mathtt{z} \ldots$ или $\exists \mathtt{z} \ldots$ заменяме всяко срещане на $\mathtt{z}$ с някоя променлива $\mathtt{z}'$, която не се среща никъде в тази подформула. Ако $\mathtt{z}' \neq \mathtt{x}$, то няма да има проблеми. Да допуснем обаче, че $\mathtt{z}' = \mathtt{x}$ и да си припомним как променихме формулите χ''_i — тъй като при тях се отказахме да преименуваме променлива $\mathtt{x}$ на началния квантор, то значи в тези формули на някои места може да се срещат променливи $\mathtt{x}$, които в съответната формула от първоначалната редица $\chi_0, \chi_1, \ldots, \chi_n$ не са се срещали, защото сме били преименували временно $\mathtt{x}$. Ако това се случи, нека се уговорим да преименуваме $\mathtt{z}$ не на $\mathtt{x}$, а на някоя променлива $\mathtt{x}'$,която не се среща никъде в тази редица.

По този начин най-после получаваме редица $\chi'''_0, \chi'''_1, \ldots, \chi'''_m$, която се получава с еднократни преименувания, но възниква нов проблем. В новата редица на някои места вместо променливата $\mathtt{x}$ се среща променливата $\mathtt{x}'$. Дали тази редица завършва с ψ (което ни трябва, за да покажем $\varphi \equiv \psi$)? Оказва се, че да. Налагаше се вместо $\mathtt{x}$ да използваме $\mathtt{x}'$ в следния случай — когато в областта на действие на някой квантор искахме да преименуваме всяко срещане на някоя променлива $\mathtt{t}$ на $\mathtt{x}$ и не можехме, защото $\mathtt{x}$ вече се срещаше в тази област на действие. Но щом в първоначалната редица $\chi_0, \chi_1, \ldots, \chi_n$ сме могли да направим такова преименуване, значи там $\mathtt{x}$ не се е срещала. Единственият начин това да се случи е в първоначалната редица временно да сме били преименували променливата $\mathtt{x}$ на началния квантор. Следователно променливата $\mathtt{x}$, която в новата редица ще ни пречи да преименуваме $\mathtt{t}$ на $\mathtt{x}$, ще се управлява от началния квантор, който в новата редица отсъства. В такъв случай същата променлива $\mathtt{x}$ ще се управлява от началния квантор и в последната формула $\forall \mathtt{x}\, \psi$, което означава, че в тази формула (а също и във формулата ψ) кванторът, чиято променлива от $\mathtt{t}$ е станала на $\mathtt{x}$ не може да продължи да бъде с променлива $\mathtt{x}$, защото в такъв случай ще „хване“ чужда променлива. Така виждаме, че това че на

някое място не можем да заменим $\mathtt{t}$ на $\mathtt{x}$ не е проблем, защото някъде по-нататък тази променлива $\mathtt{x}$ и без това ще се замени с някоя друга.

Задача 26.

$$\begin{aligned}
\mathtt{f(x,y)[x,y := y,y]} &= \mathtt{f(y,y)} \\
\mathtt{f(x,y)[x,y := g(x),x]} &= \mathtt{f(g(x),x)} \\
\mathtt{f(g(x),y)[x,y := g(x),x]} &= \mathtt{f(g(g(x)),x)} \\
\mathtt{f(x,g(y))[x,y := g(x),x]} &= \mathtt{f(g(x),g(x))}
\end{aligned}$$

Задача 27. Тъй като субституцията не променя променливата $\mathtt{y}$, ще обясним какво се случва само с променливите $\mathtt{x}$.

(а) Първата променлива $\mathtt{x}$ е свързана и затова не я пипаме. Втората променлива $\mathtt{x}$ можем да заменим без проблеми. Отговор: $\forall \mathtt{x}\, \mathtt{p(x,y)} \vee \mathtt{q(f(y))}$.

(б) Ако просто заменим променливата $\mathtt{x}$ с $\mathtt{f(y)}$ ще получим грешка от втори вид, защото променливата $\mathtt{y}$ в $\mathtt{f(y)}$ ще се свърже с квантора $\forall \mathtt{y}$. Затова най-напред трябва да преименуваме свързаната променлива $\mathtt{y}$ например на $\mathtt{z}$ и след това да заменим $\mathtt{x}$ с $\mathtt{f(y)}$. Отговор: $\forall \mathtt{z}\, \mathtt{p(f(y),z)} \vee \mathtt{q(y)}$. Забележете, че свободната променлива $\mathtt{y}$ не се преименува!

Задача 28. При прилагане на s към първата формула получаваме

$$\mathtt{p(f(x_1 + x_2, z), (t + x_3) + x_1)} \vee \mathtt{f(x_1 + x_2, z) + (y + x_3) = f(x_1 + f(x_2, t), f(y, y))}$$

Тъй като останалите формули съдържат квантори, то трябва да проверим дали няма квантори с опасни променливи.

Свободни променливи на втората формула са: $\mathtt{t}, \mathtt{x}_1$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{t}), s(\mathtt{x}_1)$, т.е. променливите $\mathtt{x}_1, \mathtt{x}_2, \mathtt{t}, \mathtt{z}$. И двата квантора са с променливи, които не са измежду опасните, следователно не се налага да преименуваме. Както обикновено, когато прилагаме субституцията, трябва да заменяме само свободните променливи. Получаваме

$$\forall \mathtt{x}_3(\mathtt{p(x_3, x_1 + f(x_2, t))} \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f(y_{178}, x_3) = f(x_1 + x_2, z) + (x_1 + f(x_2, t))}))$$

Свободни променливи на третата формула са: $\mathtt{z}, \mathtt{x}_1$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{z}), s(\mathtt{x}_1)$, т.е. променливите $\mathtt{t}, \mathtt{x}_3, \mathtt{x}_1, \mathtt{x}_2, \mathtt{z}$. Кванторът $\forall \mathtt{x}_3$ е с опасна променлива и трябва да го преименуваме. Например ако преименуваме неговата променлива $\mathtt{x}_3$ на $\mathtt{x}_{352}$ ще получим

$$\forall \mathtt{x}_{352}(\mathtt{p(x_{352}, z)} \Rightarrow \exists \mathtt{y}\, (\mathtt{f(y, x_{352})} = \mathtt{x_1 + z}))$$

Към тази формула вече може да прилагаме субституцията s (заменяме само свободните променливи). Получаваме

$$\forall \mathtt{x}_{352}(\mathtt{p(x_{352}, (t + x_3) + x_1)} \Rightarrow \exists \mathtt{y}\, (\mathtt{f(y, x_{352})} = \mathtt{f(x_1 + x_2, z) + (x_1 + f(x_2, t))}))$$

Свободни променливи на четвъртата формула са: $\mathtt{z}, \mathtt{x}_2$. Следователно опасни променливи са променливите, срещащи се в термовете $s(\mathtt{z}), s(\mathtt{x}_2)$, т.е. променливите

$\mathtt{t}, \mathtt{x}_3, \mathtt{x}_1, \mathtt{y}$. И двата квантора се оказват с опасна променлива и трябва да ги преименуваме. Например ако преименуваме променлива $\mathtt{x}_3$ на $\mathtt{x}_{352}$ и $\mathtt{y}$ на $\mathtt{y}_{178}$ ще получим

$$\forall \mathtt{x}_{352}(\mathtt{p}(\mathtt{x}_{352}, \mathtt{z}) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_{352}) = \mathtt{x}_2 + \mathtt{z}))$$

Към тази формула вече може да прилагаме субституцията s (заменяме само свободните променливи). Получаваме

$$\forall \mathtt{x}_{352}(\mathtt{p}(\mathtt{x}_{352}, (\mathtt{t} + \mathtt{x}_3) + \mathtt{x}_1) \Rightarrow \exists \mathtt{y}_{178}(\mathtt{f}(\mathtt{y}_{178}, \mathtt{x}_{352}) = (\mathtt{y} + \mathtt{x}_3) + (\mathtt{x}_1 + \mathtt{f}(\mathtt{x}_2, \mathtt{t}))))$$

Задача 33. Съгласно задача 32 $\varphi' \equiv \varphi[\mathtt{x} := \mathtt{x}']$ и $\psi' \equiv \psi[\mathtt{x} := \mathtt{x}']$. Пак от същата задача следва, че е достатъчно да докажем, че $\varphi' \,\&\, \psi' \equiv (\varphi \,\&\, \psi)[\mathtt{x} := \mathtt{x}']$. Последното е вярно, защото съгласно задача 31 $(\varphi \,\&\, \psi)[\mathtt{x} := \mathtt{x}'] \equiv \varphi[\mathtt{x} := \mathtt{x}'] \,\&\, \psi[\mathtt{x} := \mathtt{x}']$.

Задача 34. (10) За произволни реални числа x, y и z ако $x < y$ и $y < z$, то $x < z$.

(11) За произволни реални числа x, y и z ако $x < y$, то $x + z < y + z$.

(12) Ако $3 < 3$, то всяко реално число е по-малко от себе си.

Задача 35. 1. $\mathtt{p}(\mathtt{x}, \mathtt{y})$ `-:` $\mathtt{p}(\mathtt{y}, \mathtt{x})$

2. `-:` $\mathtt{p}(\mathtt{x}, \mathtt{y})$

Задача 36. Тъй като универсумът на $\mathbf{M}$ е непразно множество, то той съдържа поне един елемент μ. Да дефинираме оценката v така: $v(\mathtt{x}) = \mu$ за всяка променлива $\mathtt{x}$.

Задача 37. Тъй като τ не съдържа променливи, то $v(\mathtt{x}) = v'(\mathtt{x})$ за всяка променлива $\mathtt{x}$, която се среща в τ. Затова искането следва от твърдение 3.2.6.

Задача 38. Нека термът τ е $\mathtt{f}(\mathtt{g}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{x}_2)$.

Задача 39. Нека термът τ е $\mathtt{f}(\mathtt{g}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{x}_2)$ и формулата φ е $\mathtt{p}(\mathtt{g}(\mathtt{x}_1, \mathtt{x}_1), \mathtt{f}(\mathtt{x}_2, \mathtt{x}_2))$.

Задача 40. Непосредствено от дефиницията следва, че φ не е изпълнима в $\mathbf{M}$ тогава и само тогава, когато φ е неизпълнима в $\mathbf{M}$.

Ако φ е неизпълнима, то не съществува оценка, при която φ е вярна, следователно каквато и оценка да изберем, φ няма да е вярна при тази оценка и значи φ е невярна при всяка оценка, т.е. φ е тъждествено невярна.

Ако φ е тъждествено невярна в $\mathbf{M}$, то φ е невярна при всяка оценка, значи няма как да намерим оценка, при която φ е вярна, следователно φ е неизпълнима.

Задача 41. Непосредствено от дефиницията следва, че φ не е изпълнима тогава и само тогава, когато φ е неизпълнима.

Ако φ е неизпълнима, то не съществуват структура и оценка, при които φ е вярна, следователно каквито и структура и оценка да изберем, φ няма да е вярна при тези структура и оценка и значи φ е невярна при произволно избрани структура и оценка, т.е. φ е тъждествено невярна.

Ако φ е тъждествено невярна, то φ е невярна при всички структура и оценка, значи няма как да намерим структура и оценка, при които φ е вярна, следователно φ е неизпълнима.

Задача 42. Да. Например атомарната формула $\mathtt{x} = \mathtt{c}$ е изпълнима във всяка структура с равенство, защото е вярна когато оценката дава на $\mathtt{x}$ същата стойност, каквато структурата дава на $\mathtt{c}$. Но от друга страна тази формула няма да бъде тъждествено вярна, ако в универсума на структурата се съдържат поне два елемента, защото тогава ще съществуват оценки, при които стойността на $\mathtt{x}$ се различава от стойността на $\mathtt{c}$.

Задача 43. Ако φ е тъждествено вярна, то φ е вярна при всяка оценка. Нека v е произволно избрана оценка. Тогава φ е вярна при оценка v, значи φ е изпълнима в структурата и следователно не е неизпълнима в структурата.

В това доказателство използваме, че универсумът на структурата е непразно множество, по следния начин: за да можем да изберем произволна оценка v, е нужно да сме сигурни, че съществува поне една оценка. Ако ни е даден поне един елемент от универсума, тогава със сигурност ще съществува поне една оценка — да вземем например оценката, която на всички променливи дава като стойност дадения елемент от универсума. Ако обаче универсумът бе празното множество, то не би съществувала нито една оценка, защото за произволна променлива $\mathtt{x}$, $v(\mathtt{x})$ трябва да бъде елемент на универсума.

Задача 44. Задачата може да се реши лесно, използвайки свойствата на хомоморфизмите, които ще докажем по-нататък, но всъщност тя има много и най-различни решения. Ето едно (упътване):

Нека $\mathbf{M}$ е модел за Г. За всяко естествено число m ще дефинираме различни по между си структури $\mathbf{K}_m$, които са модели за Г. Универсум на $\mathbf{K}_m$ е $|\mathbf{M}| \times \{m\}$. За всеки символ за константа нека $\mathtt{c}^{\mathbf{K}_m} = \langle \mathtt{c}^{\mathbf{K}}, m\rangle$. За всеки функционален символ нека $\mathtt{f}^{\mathbf{K}_m}(\langle\mu_1, m\rangle, \langle\mu_2, m\rangle, \ldots, \langle\mu_n, m\rangle) = \langle \mathtt{f}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n), m\rangle$. И за всеки предикатен символ нека $\mathtt{p}^{\mathbf{K}_m}(\langle\mu_1, m\rangle, \langle\mu_2, m\rangle, \ldots, \langle\mu_n, m\rangle) = \mathtt{p}^{\mathbf{M}}(\mu_1, \mu_2, \ldots, \mu_n)$.

За произволна оценка v в $\mathbf{K}_m$ да означим с v' оценката в $\mathbf{M}$, такава че за произволна променлива $\mathtt{x}$, $v'(\mathtt{x})$ е равно на първия елемент на наредената двойка $v(\mathtt{x})$.

С индукция по термовете докажете, че за произволен терм τ и оценка v в $\mathbf{K}_m$, ако μ е стойността на τ в $\mathbf{M}$ при оценка v', то $\langle\mu, m\rangle$ е стойността на τ в $\mathbf{K}_m$ при оценка v.

След това докажете още, че за произволна атомарна формула φ и оценка v в $\mathbf{K}_m$, стойността на φ в $\mathbf{M}$ при оценка v' е еквивалентна на стойността на φ в $\mathbf{K}_m$ при оценка v.

От тук заключете, че ако някоя формула е тъждествено вярна в $\mathbf{M}$, то тя ще бъде тъждествено вярна и в $\mathbf{K}_m$ и обратно.

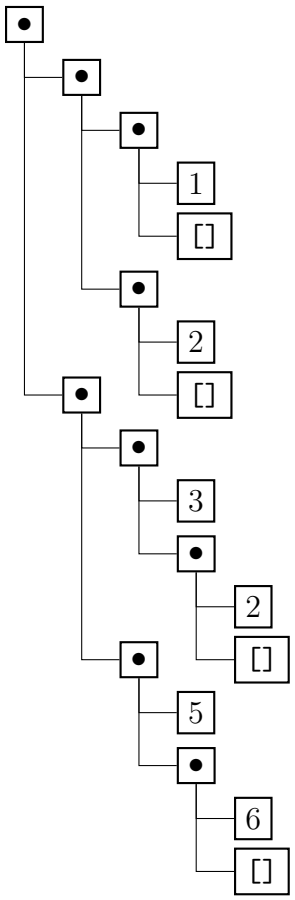
Задача 45. Разбира се, че е възможно. В твърдение 3.4.9 се говори за изпълнимост и тъждествена вярност в конкретна структура, а не за изпълнимост и тъждествена вярност по принцип. Например формулата $\forall\mathtt{x}\,\forall\mathtt{y}\,(\mathtt{x.y} = \mathtt{y.x})$ е изпълнима, защото е вярна във всяка структура, която е абелева група или комутативен пръстен, но не е тъждествено вярна (не е предикатна тавтология), защото не е вярна в структура, която е неабелева група или некомутативен пръстен.

Задача 46.

1. Ако структурата $\mathbf{M}$ е такава, че в нея са тъждествено верни всички елементи на Г, то в частност и φ ще бъде тъждествено вярно в $\mathbf{M}$.
2. следва от в).
3. Да допуснем, че φ следва от Δ. Да изберем произволна структура, в която са тъждествено верни елементите на Г. Тъй като всеки от елементите на Δ следва от Г, то в тази структура ще бъдат тъждествено верни и елементите на Δ. Но φ следва от Δ, значи в тази структура и φ е тъждествено вярно.

Задача 49. `[[[1],2],[3,4],5,6]`. За синтактичното дърво вж. фиг. 19.

Задача 50. `:- op(800,fy,[not]).`



Фиг. 19. Синтактично дърво за
`[[[1|[]]|[2|[]]]|[[3|[4|[]]]|[5|[6|[]]]]]` и
`•(•(•(1,[]),•(2,[])),•(•(3,•(4,[])),•(5,•(6,[]))))`

Задача 51.

```
:- op(800,fy,[не]).
:- op(1000, xfy, [и]).
:- op(1100, xfy, [или]).
не Нещо :- not(call(Нещо)).
Едно и Друго :- call(Едно), call(Друго).
Едно или Друго :- call(Едно); call(Друго).
```

Приложение Г

Задължителни неща

На изпита по логическо програмиране се очаква всички студенти да са научили нещата от списъка, даден по-долу. Познаването само на тези неща не гарантира получаването висока оценка. Не е нужно да се помнят условията на задачите, включени в този списък, но трябва да може да се решава всяка задача с подобно условие.

При твърденията, които трябва да се знаят без доказателство, от студентите се очаква да се сещат сами за съответното твърдение, ако то е нужно за решаването на някоя задача. Да разгледаме например следната задача:

> Нека структурата $\mathbf{M}$ е с универсум множеството на естествените числа, а оценките v и w в $\mathbf{M}$ са такива, че $v(\mathtt{x}) = 5$, $v(\mathtt{y}) = 8$, $v(\mathtt{z}) = 3$, $w(\mathtt{x}) = 7$, $w(\mathtt{y}) = 8$, $w(\mathtt{z}) = 9$. Да се докаже, че стойността на формулата
>
> $$\forall \mathtt{x}\, \mathtt{p}(\mathtt{x}, \mathtt{y}) \Rightarrow \exists \mathtt{z}\, \forall \mathtt{x}\, (\neg \mathtt{q}(\mathtt{z}, \mathtt{x}, \mathtt{y}) \,\&\, \mathtt{p}(\mathtt{x}, \mathtt{y}))$$
>
> в $\mathbf{M}$ при оценка v е еквивалентна на стойността ѝ в $\mathbf{M}$ при оценка w.

От студентите се очаква да могат да дадат приблизително следното решение:

Променливата $\mathtt{y}$ е единствената свободна променлива в тази формула. Тъй като $v(\mathtt{y}) = w(\mathtt{y}) = 8$, то исканата еквивалентност следва от следното твърдение:

Твърдение. *Нека v и v' са оценки в структура $\mathbf{M}$. Ако v и v' съвпадат за всички свободни променливи на формулата φ, то $\mathbf{M} \vDash \varphi[v]$ е еквивалентно на $\mathbf{M} \vDash \varphi[v']$.*

Не е нужно твърденията да могат да се цитират точно по начина, по който са били дадени на лекциите или са формулирани в записките. Разбира се, твърденията, които се цитират, трябва да са верни, а ако има съществени разлики между цитираното твърдение, и твърдението, формулирано в записките, тогава се очаква при поискване формулираното твърдение да може да се докаже.

Библиография

> Нека се занимаем сега със списъка на автори, какъвто има в други книги и какъвто липсва във вашата. Лек за това се намира много лесно, ще трябва само да намерите книга, която да съдържа пълен списък с автори от първата до последната буква на азбуката, както сам вие казахте. Същия този азбучен списък поставете във вашата книга и не се тревожете никак дори ако се открие измамата, поради това, че вие не сте имали никаква нужда да черпите знания от тези автори. Все ще се намери някой наивен читател, който да повярва, че сте използвали всички тези автори за вашата проста и обикновена история. Накрая, ако този дълъг поменик от автори не послужи за друго, то поне ще даде тежест на вашата книга. Още повече че никой не ще седне да проверява дали наистина сте чели, или не тези автори, защото от това няма да му стане нито по-топло, нито по-студено.
>
> Мигел де Сервантес Сааведра [25]

[1] Jesse B. Wright Arthur W. Burks, Don W. Warren. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8(46):53–57, 1954.

[2] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012.

[3] J.A. Bergstra and J.V. Tucker. A characterisation of computable data types by means of a finite, equational specification method. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland,*

July 14-18, 1980, Proceedings, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1980.

[4] J.A. Bergstra and J.V. Tucker. The completeness of the algebraic specification methods for computable data types. *Information and Control*, 54(3):186 – 200, 1982.

[5] Garrett Birkhoff. On the structure of abstract algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31:433–454, 10 1935.

[6] Susanne Bobzien. Ancient logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Center for the Study of Language and Information (CSLI), Stanford University, spring 2016 edition, 2016.

[7] Chin-Liang Chang and Richard C. T. Lee. *Symbolic logic and mechanical theorem proving*. Computer science classics. Academic Press, 1973.

[8] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[9] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM J. Res. Dev.*, 4(1):28–35, January 1960.

[10] L. L. Ivanov. *Algebraic recursion theory*. Ellis Horwood series in mathematics and its applications: Statistics and operational research. E. Horwood, 1986.

[11] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[12] Michael G. Main and David B. Benson. Free semiring-representations and nondeterminism. *Journal of Computer and System Sciences*, 30(3):318 – 328, 1985.

[13] Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[14] I. McGilchrist. *The Master and His Emissary: The Divided Brain and the Making of the Western World*. Yale University Press, 2009.

[15] Dauda Odunayo Olanloye. An expert system for diagnosing faults in motorcycle. *International Journal of Engineering and Applied Sciences*, 5(06), 2014.

[16] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[17] Dimiter Skordev. *Computability in combinatory spaces. An algebraic generalization of abstract first order computability.* Kluwer Academic Publishers, Dordrecht-Boston-London, 1992.

[18] Ivan Soskov. On the computational power of the logic programs. In P. Petkov, editor, *Heyting'88 (Varna)*, pages 117–137. Plenum Press, 1990.

[19] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, MA, USA, 1986.

[20] Robert D. Tennent. *Semantics of programming languages.* Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[21] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[22] Eliezer Yudkowsky. Cognitive biases potentially affecting judgment of global risks. In Nick Bostrom and Milan Cirkovic, editors, *Global Catastrophic Risks.* Oxford University Press, August 2006.

[23] Анатолий Иванович Мальцев. Несколько замечаний о квазимногообразиях алгебраических систем. *Алгебра и логика (семинар) 5*, 3:3–9, 1966.

[24] Анатолий Иванович Мальцев. *Алгебраические системы.* Современная алгебра. Наука, 1970.

[25] Мигел де Сервантес Сааведра. *Знаменитият идалго Дон Кихот де ла Манча.* Библиотека за ученика. Държавно издателство „Отечество“, София, 1986. Четвърто съкратено издание. Превод на Тодор Нейков.

[26] Димитър Скордев. Логическо програмиране. Записки. `http://http://www.fmi.uni-sofia.bg/fmi/logic/skordev/ln/lp/new/sydyrzha.htm`.

[27] Димитър Скордев. *Комбинаторные пространства и рекурсивность в них.* Изд. на БАН, 1980.

[28] Иван Сосков. Пример полного по Московакису базиса, которы не является программного полным. In А. П. Ершов, editor, *Математическая теория и практика систем программного обеспечения (Новосибирск).* ВЦ СОАН, 1982.

Именен указател

Предметен указател