Языки программирования и методы трансляции

(1 часть)

ВВЕДЕНИЕ

1 Понятие языка программирования (неформально)

Язык программирования это система обозначений для описания вычислений.

ЯП = алфавит + синтаксис + семантика .

Алфавит это символы для записи конструкций языка.

Синтаксис это правила записи конструкций языка.

Семантика это смысл конструкций языка.

Способы задания синтаксиса

Формы Бэкуса-Наура (БНФ)

Синтаксические диаграммы

Формы Бэкуса - Наура

Разработаны Джоном Бэкусом в 1960 г. и использованы Петером Науром для описания синтаксиса языка программирования Алгол-60.

БНФ включают:

Терминальные символы, которые образуют алфавит языка.

Нетерминальные символы – заключаются в угловые скобки <...> и используются для обозначения синтаксических конструкций языка. Например, <двоичное число>, <метка>, <арифметическое выражение>

Метасимволы

| - или,

::= - по определению.

Пример: множество целых чисел:

<целое число> ::= <целое без знака> | + <целое без знака> | - <целое без знака>

<целое без знака> ::= <цифра>|<целое без знака > <цифра>

<цифра>::= 0|1|2|3|4|5|6|7|8|9

Две последние формы дают пример рекурсии в БНФ

Дополнительные метасимволы (расширение БНФ (РБНФ))

[…] - повторение символа 0 или 1 раз

{…} - повторение символа произвольное число раз (в т.ч. нуль)

Пример: РБНФ

Множество целых чисел:

<целое число> ::= [+ | -] <целое без знака>

<целое без знака> ::= <цифра> {<цифра>}

<цифра> ::= 0|1|2|3|4|5|6|7|8|9

Синтаксические диаграммы

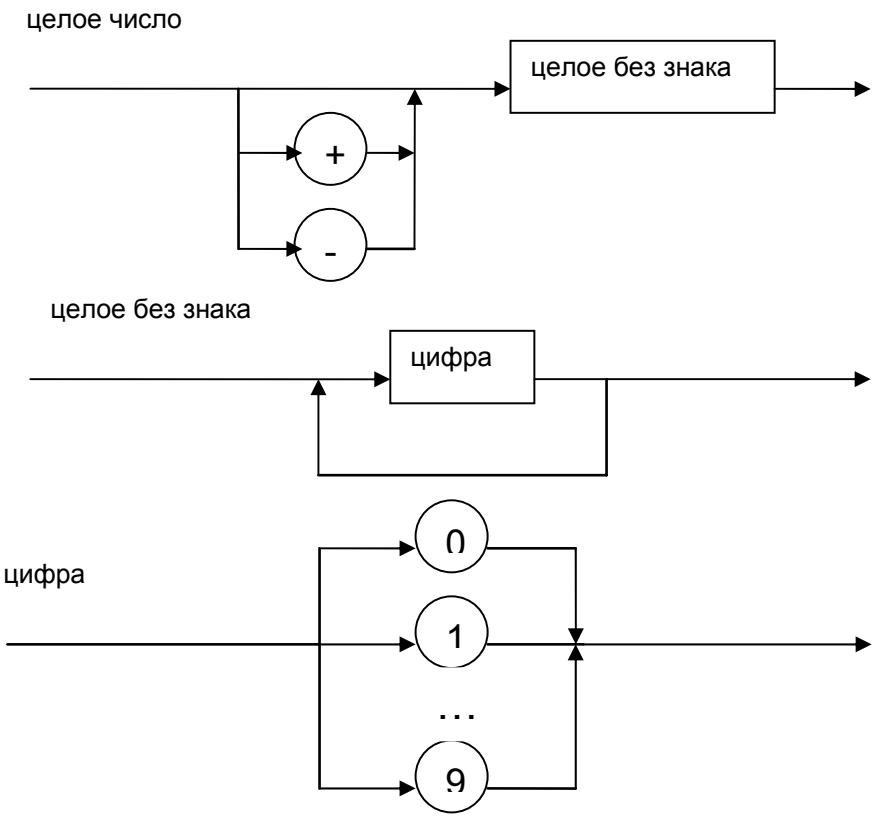
Синтаксические диаграммы это графическое изображение БНФ (Никлаус Вирт).

Нетерминальные символы изображаются в виде прямоугольников.

Терминальные символы изображаются в виде овалов.

Метасимволы заменяются композициями направленных дуг.

Пример: множество целых чисел:



Имеет место эквивалентность БНФ и синтаксических диаграмм.

2 Эволюция языков программирования

Машинные команды

Недостатки:

Необходимость знания архитектуры машины.

Простые программы из-за ограниченных возможностей машин и сложности разработки и отладки программ.

Имеется возможность встраивания данных в код программы как команд.

Ассемблер

Обеспечивает переход к символическому кодированию машинных команд .

Имеется возможность использования макросов и меток.

Программа может быть представлена в исходном тексте и в откомпилированном виде.

Имеются специальные программы-дизассемблеры.

Сравнение:

Машинные языки	Ассемблер
“-” сложность написания программ	“+” наглядность
“-” Нет переносимости программ на др. платформы	

Фортран (Fortran)

1954 , IBM , Джон Бэкус (John Backus)

Первый язык программирования высокого уровня.

Достоинства:

Абстрагирование от особенностей машинной архитектуры.

Концепция подпрограмм.

Недостатки:

Пробелы не являлись разделителями.

Цикл for до метки 10 при изменении индекса от 1 до 100

```
DO 10 I=1,100
```

Оператор присваивания

```
DO10I = 1.100
```

Cobol - 1960 г., для коммерческих приложений.

PL/1 – 1964 г., IBM, пришел на смену Cobol и Fortran .

Появилась обработка исключительных ситуаций и поддержка параллелизма.

Пробелы стали использоваться как синтаксические разделители, но ключевые слова не были зарезервированы.

```
IF ELSE=THEN THEN THEN; ELSE ELSE
```

BASIC (Beginners' All-Purpose Symbolic Instruction Code - многоцелевой язык символических инструкций для начинающих) - 1963 г.

Язык задумывался в первую очередь как средство обучения и как первый изучаемый язык программирования.

Algol – 1964 г., разработан Петером Науром (Peter Naur) .

Этот язык дал начало целому семейству Алгол-подобных языков (важнейший представитель - Pascal).

Pascal-подобные языки

PASCAL - 1970 г., Никлаус Вирт.

Достоинства:

Впервые оператор безусловного перехода перестал играть основополагающую роль при управлении порядком выполнения операторов.

Внедрена строгая проверка типов, что позволило выявлять многие ошибки на этапе компиляции.

Недостатки:

Отсутствие средств для разбиения программы на модули.

Modula-2 - 1978 г., Никлаус Вирт.

Модуль стал одной из ключевых концепций языка .

Modula-3 - 1988 г., Никлаус Вирт.

Появились объектно-ориентированные черты.

С-подобные языки

C - 1972 г., Керниган и Ритчи, язык для разработки операционной системы UNIX.

Недостатки:

Компилятор С очень слабо контролирует типы.

C++ - 1986 г., Бьярн Страуструп.

Появились объектно-ориентированные черты.

Java - 1995 г., корпорация Sun Microsystems. Разработчики Кен Арнольд и Джеймс Гослинг.

Особенности:

Компиляция в код некоей абстрактной машины, для которой затем пишется эмулятор (Java Virtual Machine) для реальных систем.

Нет указателей и множественного наследования.

C# - 1999-2000 гг., Microsoft.

Ориентирован, в основном, на разработку многокомпонентных Интернет-приложений.

Языки Ada и Ada 95

Ada - 1983 г. для Министерства Обороны США.

Имеет достоинсва:

Выявление множества ошибок на этапе компиляции.

Параллелизм, обработка исключений.

В 1995 году был принят стандарт языка Ada 95 .

Добавлена объекно-ориентированность.

Имеет недостатки:

Сложность освоения языка и громоздкий синтаксис.

Процедурно-ориентированные языки (Fortran, BASIC, Pascal, C)

Имеют достоинства:

Простота написания программ.

Наглядность представление программы.

Наличие встроенных базовых функций, процедур.

Переносимость программ на др. платформы

Иметя недостатки:

Большое количество строк в коде программы.

Не отражается модель реального мира

Объектно-ориентированные языки (Object Pascal, C++) устраняют перечисленные недостатки процедурных языков.

Языки обработки данных

APL (Application Programming Language) – 1957 г., язык описания математической обработки данных.

Допускает использование математических символов.

Snobol – 1962 г., а в 1974 его преемник **Icon.**

Синтаксис Icon напоминает C и Pascal одновременно.

Имеет встроенные функции работы со строками.

Современным аналогом Icon и Snobol является Perl - язык обработки строк и текстов.

SETL – 1969 г., язык для описания операций над множествами.

Lisp – 1958 г., язык для обработки списков. Потомки: Planner (1967), Scheme (1975), Common Lisp (1984).

Скриптовые языки

Характеризуются свойствми:

Интерпретируемость.

Простота синтаксиса.

Легкая расширяемость.

JavaScript (LiveScript) - Netscape Communications .

Интерпретируется браузером во время отображения веб-страницы.

По синтаксису схож с Java и (отдаленно) с C/C++.

VBScript – Microsoft .

Синтаксически похож на Visual Basic.

Perl – создан для обработки различного рода текстов и выделения нужной информации.

Является интерпретируемым языком.

Python – 1990 - 91 гг. Гвидо ван Россум (Guido van Rossum)

Интерпретируемый объектно-ориентированный язык программирования.

Объектно-ориентированные языки

Simula – 1967 г., язык для моделирования различных объектов и процессов .

Smalltalk – 1972 г., я зык для проектирования сложных графических интерфейсов .

Eiffel - 1986 г., чистый язык объектно-ориентированного программирования.

Языки параллельного программирования

Occam – 1982 г.,

предназначен для программирования транспьютеров - многопроцессорных систем распределенной обработки данных.

Модель параллельных вычислений Linda – 1985 г.

организует взаимодействие между параллельно выполняющимися процессами.

Языки программирования разделяются на следующие классы:

Императивные языки. Программы на них представляют собой пошаговое описание решения задачи.

Неимперативные языки. Программа описывает только постановку проблемы. Решение задачи выполняет компилятор.

Существуют 2 подхода: функциональное и логическое программирование.

Функциональные языки

Поддерживают представление программы в виде математических функций.

Имеются 2 типа языков: с ленивой и с энергичной семантикой.

Язык с энергичной семантикой – ML и два его диалекта Standard ML (SML) и CaML.

Языки с ленивой семантикой – Haskell и Clean.

Языки логического программирования

Программы на них выражены как формулы математической логики .

Prolog – 1971 г.

Потомки:

Parlog – 1983 г. , ориентирован на параллельные вычисления .

Delta Prolog.

3 Классификация языков программирования

По поколениям – Generation Language (GL):

1GL – система команд

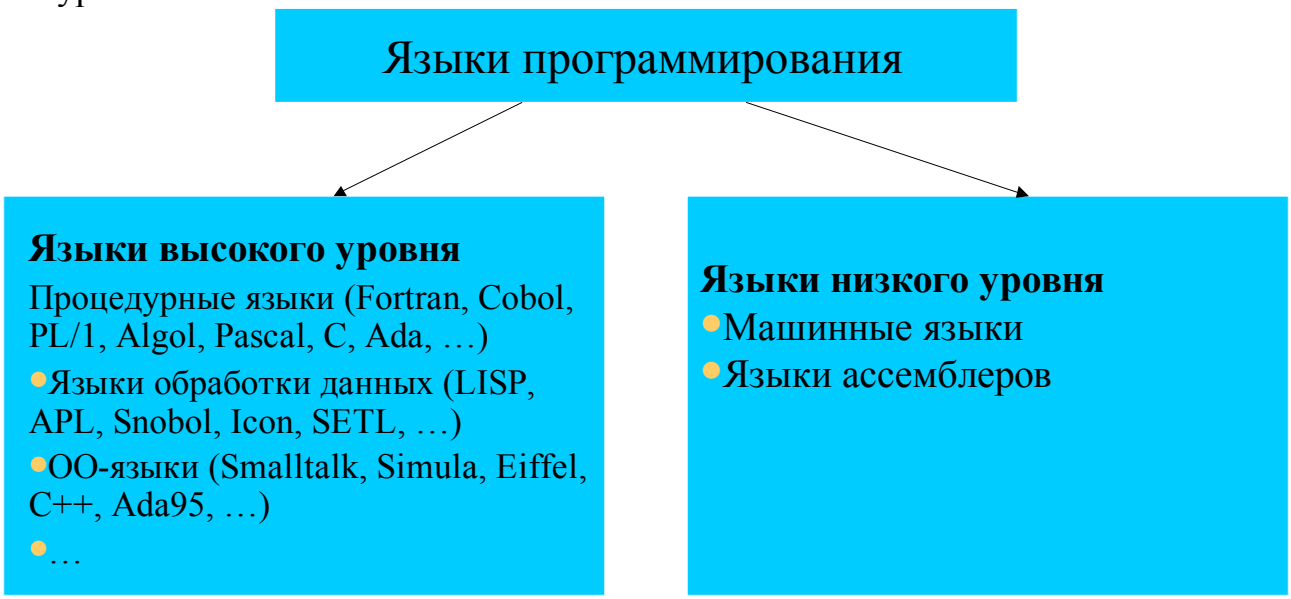
2GL – машинно-ориентированные языки

3GL – языки высокого уровня

4GL – объектно-ориентированные языки

5GL – интеллектуальные языки

По уровню:



Языки высокого уровня характеризуются

наличием понятия типа данных,

независимостью от архитектуры конкретного компьютера (мобильностью программ),

развитыми управляющими структурами и средствами описания структур данных,

близостью к естественному языку.

Языки низкого уровня характеризуются

отсутствием понятия типа данных,

зависимостью от архитектуры конкретного компьютера (отсутствием мобильности программ),

примитивными управляющими структурами и средствами описания структур данных,

близостью к машинному языку.

4 Среда программирования

4.1 Понятие среды программирования

Среда программирования это совокупность программ, обеспечивающих технологический цикл разработки программ: **анализ, спецификация, проектирование, кодирование** (редактирование, компиляция, компоновка)**, тестирование, отладка.**

Базовые компоненты среды

Редактор – средство создания и изменения исходных файлов с текстом программы.

Компилятор – транслирует исходный файл в объектный файл, содержащий команды в машинном коде для конкретного компьютера.

Компоновщик (редактор связей) – собирает объектные файлы программы и формирует исполняемый файл (разрешая внешние ссылки между объектными файлами).

Отладчик – средство управления выполнением исполняемого файла на уровне отдельных операторов программы для диагностики ошибок.

Прочие компоненты среды

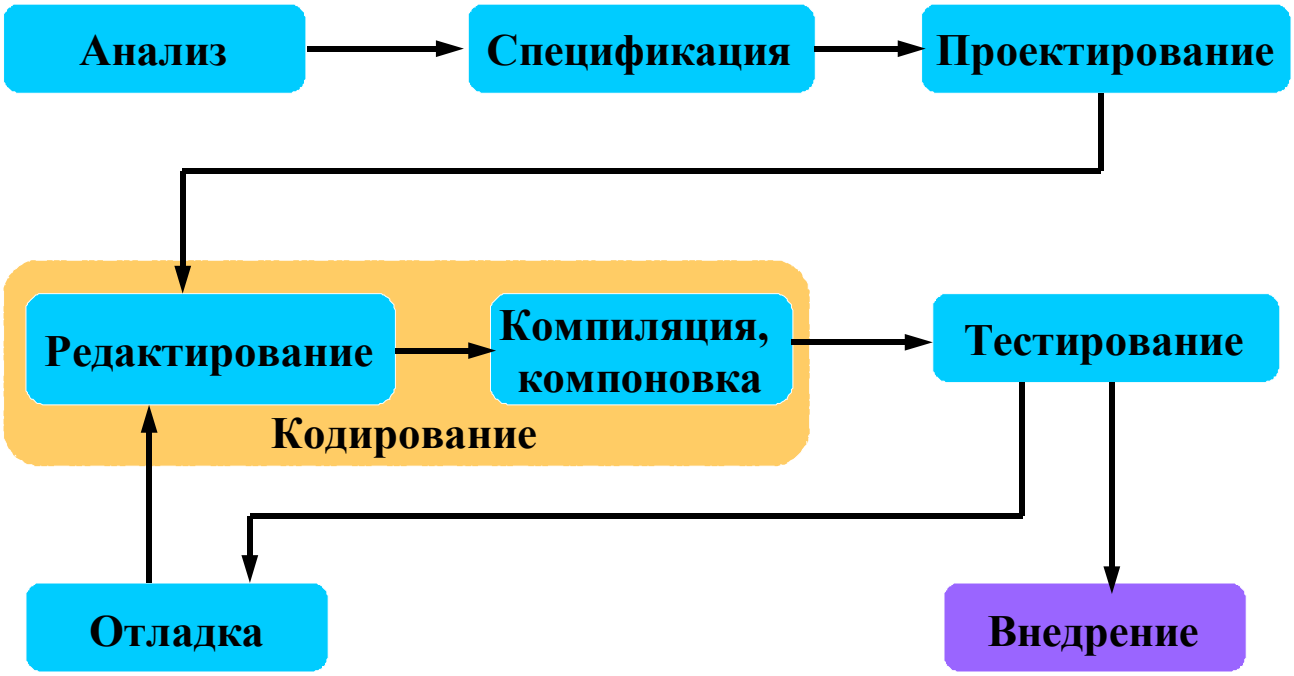
Библиотекарь – средство ведения совокупностей объектных файлов (библиотек).

Профилировщик – средство измерения времени выполнения программных компонент для последующей оптимизации критических компонентов.

Загрузчик – копирует исполняемый файл с диска в память и осуществляет его запуск.

4.2 Техника разработки программ

Цикл разработки программы может быть представлен следующей схемой:



Анализ – определение того, что должна делать программа (но не как она это должна делать).

Спецификация – описание требований к программе в формальном виде.

Проектирование – разработка структуры и алгоритма программы.

Кодирование = редактирование + компиляция + компоновка.

Тестирование – подготовка эталонных входных и соответствующих выходных данных (тестов), запуск программы и сравнение полученных данных с эталонными.

Отладка – выявление и исправление ошибок.

4.3 Классификация ошибок в программе

Синтаксическая ошибка

Связана с нарушением синтаксических правил.

Ошибка времени выполнения (run-time error)

происходит при выполнении (синтаксически верной) программы, когда она производит какое-либо недопустимое действие (деление на ноль и др.).

Логическая ошибка

Ошибка при разработке и написании алгоритма. При этом программа не содержит ни синтаксических ошибок, ни ошибок времени выполнения, но делает не то, что хотел автор программы.

4.4 Отладка

Отладка (Debugging) это процесс локализации и исправления ошибок, выявленных во время тестирования программы.

Разновидности отладки:

«Сухая» отладка – по листингу (тексту) программы, без использования компьютера, отладчика и др.

Использование отладчика (Debugger) – программы, позволяющей отслеживать процесс выполнения программы по ее исходному тексту и просматривать текущие значения переменных.

Основные функции отладчика

Трассировка – пошаговое выполнение программы. Шагу соответствует *одна строка исходного текста* (в которой может быть более одного оператора).

Режим "без трассы подпрограмм" – пошаговое выполнение программы, при котором вызов подпрограммы отрабатывается как один оператор.

Режим "трасса подпрограмм" – пошаговое выполнение программы, при котором трасса включает все операторы подпрограмм.

Точка останова (breakpoint) приостанавливает выполнение программы. Может быть установлена только на выполняемом операторе (не на комментарии и др.).

Возможен *просмотр значений переменных* при пошаговом выполнении программы.

Языки прграммирования

5. Основные виды языков программирования

Программы на традиционных языках программирования, таких как Си, Паскаль, Java и т.п. состоят их последовательности модификаций значений некоторого набора переменных, который называется *состоянием.* Если не рассматривать операции ввода-вывода, а также не учитывать того факта, что программа может работать непрерывно (т.е. без остановок, как в случае серверных программ), можно сделать следующую абстракцию. До начала выполнения программы состояние имеет некоторое начальное значение σ_0, в котором представлены входные значения программы. После завершения программы состояние имеет новое значение σ', включающее в себя то, что можно рассматривать как «результат» рабо-

ты программы. Во время исполнения каждая команда изменяет состояние; следовательно, состояние проходит через некоторую конечную последовательность значений:

$$\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_n = \sigma'$$

Состояние модифицируется с помощью команд *присваивания,* записываемых в виде v = E или v := E, где v - переменная, а E - некоторое выражение. Эти команды следуют одна за другой; операторы, if и while, позволяют изменить порядок выполнения этих команд в зависимости от текущего значения состояния. Такой стиль программирования называют *императивным или процедурным.*

Функциональное программирование представляет парадигму, отличную от рассмотренной модели. Функциональная программа представляет собой некоторое выражение (в математическом смысле); выполнение программы означает вычисление значения этого выражения. Считая, что результат работы императивной программы полностью и однозначно определен ее входом, можно сказать, что финальное состояние (или любое промежуточное) представляет собой некоторую функцию (в математическом смысле) от начального состояния, т.е. $\sigma' = f(\sigma)$. В функционально программировании используется именно такая точка зрения: программа представляет собой выражение, соответствующее функции *f*. Функциональные языки программирования поддерживают построение таких выражений.

При сравнении функционального и императивного подхода к программированию можно отметить следующие свойства функциональных программ:

- Функциональные программы не используют переменные в том смысле, в котором они используются в императивном программировании. В функциональных программах не используется оператор присваивания.
- Как следствие из предыдущего пункта, в функциональных программах нет циклов.
- Выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.

• Функции можно передавать в другие функции в качестве аргументов и возвращать в качестве результата, и проводить вычисления, результатом которых будет функция.

• Вместо циклов функциональные программы широко используют рекурсивные функции.

Многие характеристики императивных языков программирования являются результатом абстрагирования от машинных кодов к языкам ассемблера, а затем к языкам типа Фортран и т.д. Более привлекателен подход, при котором языки программирования появляются как абстрактные системы для записи алгоритмов, а затем происходит их перевод на императивный язык компьютера.

Функциональный подход имеет ряд преимуществ перед императивным. Функциональные программы более непосредственно соответствуют математическим объектам, и следовательно, позволяют проводить строгие рассуждения. Установить значение императивной программы, т.е. той функции, вычисление которой она реализует, в общем случае довольно трудно. Значение функциональной программы может быть выведено практически непосредственно.

Следует сделать замечание относительно употребления термина «функция» в императивных языках. В математическом смысле «функции» императивного языка не являются функциями, поскольку:

• Их значение может зависеть не только от аргументов;

• Результатом их выполнения могут быть разнообразные *побочные эффекты* (например, изменение значений глобальных переменных)

• Два вызова одной и той же функции с одними и теми же аргумен тами могут привести к различным результатам.

Функции в функциональных программах являются функциями в математическом смысле. Из этого следует, что вычисление любого выражения не может иметь никаких побочных эффектов, и порядок вычисления его подвыражений не оказывает влияния на результат. Функциональные программы легко поддаются распараллеливанию, поскольку отдельные компоненты выражений могут вычис-

ляться одновременно.

6 Лямбда-исчисление как формализация функциональных языков

Теория машин Тьюринга является основой императивных языков программирования, лямбда-исчисление служит базисом и математическим «фундаментом», на котором основаны функциональные языки программирования.

Лямбда-исчисление было изобретено в начале 30-х годов логиком А. Чёрчем, который использовал его в качестве формализма для обоснования математики.

В настоящее время лямбда-исчисление является основной формализацией, применяемой в исследованиях связанных с языками программирования. Это, связано, со следующими факторами:

- Это единственная формализация, которая может быть непосредственно использована для написания программ.
- Лямбда-исчисление дает простую и естественную модель для таких важных понятий, как рекурсия и вложенные среды.
- Большинство конструкций традиционных языков программирования может быть отображено в конструкции лямбда-исчисления.
- Функциональные языки являются удобной формой синтаксической записи для конструкций различных вариантов лямбда-исчисления. Некоторые современные языки например Haskell имеют полное соответствие своей семантики с семантикой подразумеваемых конструкций лямбда-исчисления.

Лямбда-выражением будем называть конструкцию вида

$$\lambda x.E$$

где E — некоторое выражение, возможно, использующее переменную x.
Пример. $\lambda x.x^2$ представляет собой функцию, возводящую свой аргумент в квадрат.

Использование лямбда-нотации позволяет четко разделить случаи, когда под выражением вида $f(x)$ мы понимаем саму функцию f, а когда ее значение в точке x.

Применение функции f к аргументу x мы будем обозначать как $f\,x$. Будем считать, что применение функции к аргументу ассоциативно влево, т.е. $f\,x\,y$ означает $(f(x))(y)$. В качестве сокращения для выражений вида $\lambda x.\ \lambda y.E$ будем использовать запись $\lambda xy.E$ (аналогично для большего числа аргументов). Будем считать, что «область действия» лямбда-выражения простирается вправо насколько возможно, т.е., например, $\lambda x.x\,y$ означает $\lambda x.(xy)$, а не $(\lambda x.x)\,y$.

Существует операция *каррирования,* позволяющая записывать функции многих переменных в обычной лямбда-нотации. Она заключается в том, чтобы выражения вида $\lambda x\,y.x + y$ рассматривать как функцию $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, т.е. если его применить к одному аргументу, результатом будет функция, которая принимает другой аргумент. Таким образом:

$$(\lambda x\,y.x + y)\ 1\ 2\ = (\lambda y.1 + y)\ 2 = 1 + 2.$$

Переменные в лямбда-выражениях могут быть *свободными* и *связанными.* В выражении вида $x^2 + x$ переменная x является свободной; его значение зависит от значения переменной x и в общем случае ее нельзя переименовать. В подвыражении переменная может быть свободной, в то время как во всем выражении она может быть связана какой-либо *операцией связывания переменной,* такой как операция суммирования. Та часть выражения, которая находится «внутри» операции связывания, называется *областью видимости* переменной.

В лямбда исчислении выражения $\lambda x.E[x]$ и $\lambda y.E[y]$ считаются эквивалентными (это называется α-эквивалентностью, и процесс преобразования между такими парами называют α-преобразованием). При этом, необходимо наложить условие, что y не является свободной переменной в $E[x]$.

7 Лямбда-исчисление как формальная система

Лямбда-исчисление основано на формальной нотации лямбда-терма, составляемого из переменных и некоторого фиксированного набора констант с использованием операции применения функции и лямбда-абстрагирования. Это означает, что все лямбда-выражения можно разделить на четыре категории:

1. **Переменные:** обозначаются произвольными строками, составленными из букв и цифр.

2. **Константы:** также обозначаются строками; отличие от переменных будем определять из контекста.

3. **Комбинации:,** т.е. применения функции *S* к аргументу *T;* и *S* и *T* могут быть произвольными лямбда-термами. Комбинация записывается как *ST.*

4. **Абстракции** произвольного лямбда-терма *S* по переменной *x,* обозначаемые как *λx.S.*

Лямбда-терм определяется рекурсивно и его грамматику можно представить в виде следующей формы Бэкуса-Наура:

$$Exp = Var \mid Const \mid Exp\ Exp \mid \lambda\ Var\ .\ Exp$$

В соответствие с этой грамматикой лямбда-термы представляются в виде синтаксических деревьев, а не в виде последовательности символов. Отсюда следует, что соглашения об ассоциативности операции применения функции, эквивалентность выражений вида *λxy.S* и *λx. λy.S,* неоднозначность в именах констант и переменных проистекают только из необходимости представления лямбда-термов в удобном человеку виде, и не являются частью формальной системы.

7.1 Свободные и связанные переменные

Формализуем интуитивное представление о свободных и связанных переменных. Множество свободных переменных *FV(S)* лямбда-терма *S* можно определить рекурсивно следующим образом:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(c) &= \varnothing \\ FV(ST) &= FV(S) \cup FV(T) \\ FV(\lambda x.S) &= FV(S) \setminus \{x\} \end{aligned}$$

Аналогично множество связанных переменных *BV(S)* определяется следующими формулами:

$$\begin{aligned} BV(x) &= \varnothing \\ BV(c) &= \varnothing \end{aligned}$$

$$BV(ST) = BV(S) \cup BV(T)$$
$$BV(\lambda x.S) = BV(S) \cup \{x\}$$

Здесь предполагается, что *c* — некоторая константа.

Пример. Для терма S = *(λx y.x) (λx.z x)* можно показать, что *FV(S)* = {*z*} и *BV(S)* = {*x,y*}.

7.2 Подстановки

Интуитивно ясно, что применение терма λ*x.S* как функции к аргументу *T* дает в результате терм *S,* в котором все свободные вхождения переменной *x* заменены на *T*.

Будем обозначать операцию подстановки терма *S* вместо переменной *x* в другом терме *T* как *T*[*x := S*]. Правила подстановки можно определить рекурсивно. При этом необходимо наложить дополнительные ограничения, позволяющие избегать конфликта в именах переменных.

$$\begin{aligned}
x[x := T] &= T \\
y[x := T] &= y, \text{ если } x \neq y \\
c[x := T] &= c \\
(S_1\ S_2)[x := T] &= S_1[x := T]\ S_2[x := T] \\
(\lambda x.S)[x := T] &= \lambda x.S \\
(\lambda y.S)[x := T] &= \lambda y.(S[x := T]), \text{ если } x \neq y \text{ и } x \notin FV(S), \text{ либо } y \notin FV(T) \\
(\lambda y.S)[x := T] &= \lambda z.(S[y := z][x := T]) \text{ иначе, где } z \notin FV(S) \cup FV(T)
\end{aligned}$$

7.3 Конверсия

Лямбда-исчислене основано на трех операциях конверсии, которые позволяют переходить от одного терма к другому, эквивалентному ему. Эти конверсии обозначают греческими буквами *a, (β* и *η)*. Они определяются следующим образом:

- α-конверсия: $\lambda x.S \xrightarrow{\alpha} \lambda y.S[x := y]$ при условии, что $y \notin FV(S)$. Например, $\lambda u.u\ v \xrightarrow{\alpha} \lambda w.w\ u$.
- β-конверсия: $(\lambda x.S)\ T \xrightarrow{\beta} S[x := T]$.
- η-конверсия: $\lambda x.T\ x \xrightarrow{\eta} T$, если $x \notin FV(T)$. Например, $\lambda u.v\ u \xrightarrow{\eta} v$.

β-конверсия соответствует вычислению значения функции от аргумента, *α*-конверсия является вспомогательным механизмом для того, чтобы изменять имена связанных переменных, *η*-конверсия используется в основном при рас-

смотрении лямбда-исчисления с точки зрения логики.

7.4 Равенство лямбда-термов

Используя введенные правила конверсии, можно формально определить понятие равенства лямбда-термов. Два терма равны, если от одного из них можно перейти к другому с помощью конечной последовательности конверсии. Определим понятие равенства следующими выражениями, в которых горизонтальные линии следует понимать как «если утверждение над чертой выполняется, то выполняется и утверждение под ней»:

$$\frac{S \xrightarrow{\alpha} T \text{ или } S \xrightarrow{\beta} T \text{ или } S \xrightarrow{\eta} T}{S = T}$$

$$\frac{}{T = T}$$

$$\frac{S = T}{T = S}$$

$$\frac{S = T \text{ и } T = U}{S = U}$$

$$\frac{S = T}{S\ U = T\ U}$$

$$\frac{S = T}{U\ S = U\ T}$$

$$\frac{S = T}{\lambda x.S = \lambda x.T}$$

Следует отличать понятие равенства, определяемое этими формулами, от понятия синтаксической эквивалентности, которую будем обозначать специальным символом ≡. Например, $\lambda x.x \neq \lambda y.y$, но $\lambda x.x = \lambda y.y$. Можно рассматривать синтаксическую эквивалентность термов с точностью до α-конверсий. Такую эквивалентность обозначают символом $\equiv_a$. Это отношение определяется так же, как равенство лямбда-термов, за тем исключением, что из всех конверсий допустимы только α -конверсии. Таким образом, $\lambda x.x \equiv_a \lambda y.y$.

7.5 Экстенсиональность

η-конверсия в лямбда-исчислении выражает собой принцип *экстенсиональности.* В общефилософском смысле два свойства называются экстенсионально эк-

вивалентными, если они принадлежат в точности одним и тем же объектам. В математике принят экстенсиональный взгляд на множества, т.е. два множества считаются одинаковыми, если они содержат одни и те же элементы. Аналогично две функции равны, если они имеют одну и ту же область определения, и для любых значений аргумента из этой области определения вычисляют один и тот же результат.

В силу η-конверсии отношение равенства лямбда-термов экстенсионально. Действительно, если $f\ x$ и $g\ x$ равны для любого x, то в частности $f\ y = g\ y$, где y не является свободной переменной ни в f, ни в g. Следовательно, по последнему правилу в определении равенства лямбда-термов, имеем $\lambda y.f\ y = \lambda y.\partial\ y$. Если несколько раз применить η-конверсию, можно получить, что $f = g$. И обратно, экстенсиональность дает то, что каждое применение η-конверсии приводит к равенству, поскольку по правилу β-конверсии $(\lambda x.T\ x)\ y = T\ y$ для любого y, если x не свободна в T.

7.6 Редукция лямбда-термов

Отношение равенства лямбда-термов симметрично. Оно отражает понятие эквивалентности лямбда-термов. С вычислительной точки зрения более интересно отношение редукции (обозначаемое символом —>) следующим образом:

$$\frac{S \xrightarrow{\alpha} T \text{ или } S \xrightarrow{\beta} T \text{ или } S \xrightarrow{\eta} T}{S \to T}$$

$$\frac{}{T \to T}$$

$$\frac{S \to T \text{ и } T \to U}{S \to U}$$

$$\frac{S \to T}{S\ U \to T\ U}$$

$$\frac{S \to T}{U\ S \to U\ T}$$

$$\frac{S \to T}{\lambda x.S \to \lambda x.T}$$

Отношение редукции служит основой для вычисления терма, последовательно вычисляя комбинации $f(x)$, где f-некоторая лямбда-абстракция. Если для терма невозможно сделать никакую редукцию, за исключением α-

преобразования, говорят, что терм находится в *нормальной форме.*

7.7 Редукционные стратегии

Функциональная программа представляет собой *выражение,* а выполнение ее означает вычисление этого выражения. Можно сказать, что начиная с некоторого подтерма (редекса *от англ, redex* (*REDucible Expression*)) мы последовательно применяем редукции до тех пор, пока это возможно. При этом возникает вопрос, какую редукцию применять на каждом конкретном шаге? Отношение редукции не детерминистично, т.е. для некоторого терма *t* существует несколько различных t_i, таких что $t \longrightarrow t_i$. Иногда выбор между ними означает выбор между конечной и бесконечной последовательности редукций, т.е. между завершением и зацикливанием программы. Например, если мы начнем редукцию с самого внутреннего *редекса* в выражении *(λx.y) ((λx.x x x) (λx.x x x))*, то получим бесконечную последовательность редукций:

(λx.y) ((λx.x x x) (λx.x x x))

—> *(λx.y) ((λx.x x x) (λx.x x x) (λx.x x x))*

—> *(λx.y) ((λx.x x x) (λx.x x x) (λx.x x x) (λx.x x x))*

—> . . .

Однако если мы начнем с самого внешнего редекса, то сразу получим: *y* и больше нельзя применить никакую редукцию.

Следующая теорема утверждает, что это обстоятельство имеет место и в общем случае.

Теорема 1. *Если S —> T, где T находится в нормальной форме, то последовательность редукций, начинающаяся с S, построенная таким образом, чтобы для редукции всегда выбирался самый левый и самый внешний редекс, гарантированно завершится и приведет терм в нормальную форму.*

Понятие «самого левого и самого внешнего» редекса можно определить индуктивно: для терма *(λx.S) T* это будет сам терм; для любого другого терма *S T* это будет самый левый и самый внешний редекс в *S,* и для абстракции *λx.S* это будет самый левый самый внешний редекс в *S.* В терминах рассмотренного

синтаксиса редуцируется редекс, чей символ λ находится левее всего.

Теорема Черча-Россера, утверждает, что если начиная с терма T провести две произвольные конечные последовательности редукций, то существуют еще две последовательности редукций, которые приводят к одному и тому же терму (он может не находиться в нормальной форме).

Теорема 2. *Если* $t \longrightarrow s_1$ *и* $t \longrightarrow s_2$*, то существует терм* u*, такой, что* $s_1 \longrightarrow u$ и $s_2 \longrightarrow u$.

Эта теорема имеет несколько важных следствий.

Следствие 1. *Если* $t_1 = t_2$*, то существует терм* u *такой, что* $t_1 \longrightarrow u$ и $t_2 \longrightarrow u$.

Следствие 2. *Если* $t = t_1$ *и* $t = t_2$*, где* t_1 *и* t_2 *находятся в нормальной форме, то* $t_1 \equiv_a t_2$*, т.е.* t_1 *и* t_2 *равны с точностью до переименования переменных.*

Следовательно, нормальная форма, если она существует, единственна с точностью до α-конверсии.

С вычислительной точки зрения это означает следующее. Стратегия редуцирования самого левого самого внешнего редекса (ее называют *нормализованной стратегией)* является наилучшей, поскольку она приводит к результату, если он достижим с помощью какой-либо стратегии. С другой стороны, *любая* завершающаяся последовательность редукций всегда приводит к одному и тому же результату.

8 Комбинаторы

Теория комбинаторов была разработана до создания лямбда-исчисления, однако мы будем рассматривать ее в терминах лямбда-исчисления. *Комбинатором* будем называть лямбда-терм, не содержащий свободных переменных. Такой терм является *замкнутым*; т.е. он имеет фиксированный смысл независимо от значения любых переменных.

В теории комбинаторов установлено, что с помощью нескольких базовых комбинаторов и переменных можно выразить любой терм без применения операции лямбда-абстракции. В частности, замкнутый терм можно выразить только через

эти базовые комбинаторы. Определим эти комбинаторы следующим образом:

$$\begin{aligned} I &= \lambda x.x \\ K &= \lambda x\ y.x \\ S &= \lambda f\ g\ x.f\ x\ (g\ x) \end{aligned}$$

I является функцией идентичности, которая возващает свой аргумент. *K* служит для создания постоянных (константных) функций: применив его к аргументу *a*, получим функцию *λx.a,* которая возвращает *a* независимо от переданного ей аргумента. Комбинатор *S* является «разделяющим»: он использует две функции и аргумент и «разделяет» аргумент между функциями.

Теорема 3. *Для любого лямбда-терма t существует терм t', не содержащий лямбда-абстракций и составленный из комбинаторов S, K, I и переменных, такой что FV(t') = FV(t)* и *t' = t.*

Эту теорему можно усилить, поскольку комбинатор *I* может быть выражен в терминах *S* и *K.* Действительно, для любого *A* выполняется:

$$\begin{aligned} S\ K\ A\ x &= K\ x\ (A\ x) \\ &= (\lambda y.x)\ (A\ x) \\ &= x \end{aligned}$$

Применяя *η*-конверсию, получаем, что *I = S K A* для любого *A.* Будем использовать *A = K.* Таким образом, *I = S K K,* и *I* можно исключать из выражений, составленных из комбинаторов.

Мы представили комбинаторы как некоторые лямбда-термы, однако имеется теория, в которой они являются базовым понятием. Так же, как и в лямбда-исчислении, определяется формальный синтаксис, который вместо лямбда-абстракций содержит комбинаторы. Вместо правил *α*, *β* и *η* -конверсии, вводятся правила конверсии для выражений, содержащих комбинаторы, например *K x y -> x.* Как независимая теория, теория комбинаторов обладает многими аналогиями с лямбда-исчислением, в частности, для нее выполняется теорема Черча-Россера. Однако эта теория менее интуитивно понятна.

Комбинаторы представляют не только теоретический интерес. Лямбда-исчисление может рассматриваться как простой функциональный язык программирования, составляющий ядро реальных языков программирования, таких как ML или Haskell. Теорема 3 показывает, что лямбда-исчисление может быть «скомпилировано» в «машинный код» комбинаторов. Комбинаторы используются как метод реализации функциональных языков на уровне программного, а так же аппаратного обеспечения.

9 Лямбда-исчисление как язык программирования

9.1 Представление данных в лямбда-исчислении

Программы предназначены для обработки данных, поэтому вначале определим лямбда-выражения, представляющие данные. Затем определим базовые операции на этих данных. В реальных языках программирования строка *s,* представляющая некоторое описание в читаемом человеком формате может быть закодирована в виде лямбда-выражения *s′*. Эта процедура известна как «синтаксический сахар». Будем записывать такую процедуру в виде

$$s \triangleq s'$$

Где символ $\triangleq$ означает «равно по определению».

Эту процедуру можно трактовать как определение некоторой константы s, обозначающей операцию, которую можно использовать в терминах лямбда-исчисления.

9.1.1 Булевские значения и условия

Для кодирования значений true и false можно использовать любые неравные лямбда-термы, но удобнее определить их следующим образом:

$$\text{true} \triangleq \lambda x\, y.\ x$$

$$\text{false} \triangleq \lambda x\, y.\ y$$

Используя эти определения, можно определить условное выражение:

$$\text{if } E \text{ then } E_1 \text{ else } E_2 \triangleq E\ E_1 E_2.$$

Действительно при E = true имеем:

$$\text{if true then } E_1 \text{ else } E_2 \triangleq \text{true } E_1\ E_2$$
$$= (\lambda x\ y.\ x)\ E_1\ E_2$$
$$= E_1$$

а при E = false получаем

$$\text{if false then } E_1 \text{ else } E_2 \ = \ \text{true } E_1\ E_2$$

Определим стандартные логические операторы:

$$\text{not } p \triangleq \text{if } p \text{ then false else true}$$

$$p \text{ and } q \triangleq \text{if } p \text{ then } q \text{ else false}$$

$$p \text{ or } q \triangleq \text{if } p \text{ then true else } q$$

9.1.2 Пары и кортежи

Упорядоченную пару можно определить следующим образом:

$$(E_1,E_2) \triangleq \lambda f.\ f\ E_1\ E_2$$

Функции для извлечения компонентов пары можно определить как:

$$\begin{aligned} \text{fst}\, p &\triangleq p\ \text{true} \\ \text{snd}\, p &\triangleq p\ \text{false} \end{aligned}$$

Эти определения удовлетворяют соотношениям:

$$\begin{aligned} \text{fst}(p,q) &= (p,q)\ \text{true} = \\ &= (\lambda f.f\ p\ q)\ \text{true} = \\ &= \text{true}\ p\ q = \\ &= (\lambda x\ y.x)\ p\ q = \\ &= p \end{aligned}$$

и

$$\begin{aligned} \text{snd}(p,q) &= (p,q)\ \text{false} = \\ &= (\lambda f.f\ p\ q)\ \text{false} = \\ &= \text{false}\ p\ q = \\ &= (\lambda x\ y.y)\ p\ q = \\ &= q \end{aligned}$$

Тройки, четверки и произвольные n-кортежи можно построить с помощью пар:

$$(E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots, E_n)).$$

При этом следует ввести соглашение, что оператор «запятая» ассоциативен вправо.

Функции fst и snd можно расширить на случай n-кортежей. Определим функцию селектора, которая возвращает *i*-й компонент кортежа *p*. Будем записывать ее как $(p)_i$. Тогда $(p)_1$ = fst *p* и $(p)_i$ = fst ($snd^{i-1}p$), при $1<i<n$
и $(p)_n = snd^{n-1}p$.

9.1.3 Натуральные числа

Натуральное число *n* можно представить в виде:

$$n = \lambda f x.\ f^n\ x$$

т.е. $0 = \lambda f x.\ x,\ 1 = \lambda f x.\ f x,\ 2 = \lambda f x.\ f\ (f\ x)$ и т.д. Данное представление называется «цифрами по Черчу». Цифры по Черчу обладают рядом удобных формальных свойств.

Можно определить операцию следования (увеличения на 1):

$$\text{SUC} = \lambda\ n f x.\ n f\ (f x)$$

В самом деле:

$$\begin{aligned}
\text{SUC}\ n &= (\lambda n\ f\ x.n\ f\ (f\ x))\ (\lambda f\ x.f^n\ x) = \\
&= \lambda f\ x.(\lambda f\ x.f^n\ x)\ f\ (f\ x) = \\
&= \lambda f\ x.(\lambda x.f^n\ x)\ (f\ x) = \\
&= \lambda f\ x.f^n\ (f\ x) = \\
&= \lambda f\ x.f^{n+1}\ x = \\
&= n+1
\end{aligned}$$

Определим функцию проверки числа на равенство нулю:

$$\text{ISZERO} \triangleq \lambda\ n.\ n\ (\lambda\ x\ .\text{false})\ \text{true}$$

поскольку

$$\text{ISZERO}\ 0 = (\lambda f x.x)\ (\lambda\ x.\text{false})\ \text{true} = \text{true}$$

$$\begin{aligned}\text{ISZERO } n+1 &= (\lambda f\, x.f^{n+1}\, x)(\lambda\, x.\ \text{false})\ \text{true} = \\ &= (\lambda\, x.\text{false})^{n+1}\ \text{true} = \\ &= (\lambda\, x.\ \text{false})((\lambda\, x\ .\text{false})^{n}\ \text{true}) = \\ &= \text{false}\end{aligned}$$

Определим сложение и умножение:

$$\begin{aligned} m+n &\triangleq \lambda f\ x.m\ f\ (n\ f\ x) \\ m*n &\triangleq \lambda f\ x.m\ (n\ f)\ x \end{aligned}$$

Действительно:

$$\begin{aligned} m+n &= \lambda f\ x.m\ f\ (n\ f\ x) = \\ &= \lambda f\ x.(\lambda f\ x.f^m\ x)\ f\ (n\ f\ x) = \\ &= \lambda f\ x.(\lambda x.f^m\ x)\ (n\ f\ x) = \\ &= \lambda f\ x.f^m\ (n\ f\ x) = \\ &= \lambda f\ x.f^m\ ((\lambda f\ x.f^n\ x)\ f\ x) = \\ &= \lambda f\ x.f^m\ ((\lambda x.f^n\ x)\ x) = \\ &= \lambda f\ x.f^m\ (f^n\ x) = \\ &= \lambda f\ x.f^{m+n}\ x \end{aligned}$$

и

$$\begin{aligned} m*n &= \lambda f\ x.m\ (n\ f)\ x = \\ &= \lambda f\ x.(\lambda f\ x.f^m\ x)\ (n\ f)\ x = \\ &= \lambda f\ x.(\lambda x.(n\ f)^m\ x)\ x = \\ &= \lambda f\ x.(n\ f)^m\ x = \\ &= \lambda f\ x.((\lambda f\ x.f^n\ x)\ f)^m\ x = \\ &= \lambda f\ x.(\lambda x.f^n\ x)^m\ x = \\ &= \lambda f\ x.(f^n)^m\ x = \\ &= \lambda f\ x.f^{mn}\ x \end{aligned}$$

Более сложной является операция вычиания единицы от натурального числа. Требуется найти лямбда-выражение (обозначим его PRE), такое, что PRE 0 = 0 и PRE $n + 1 = n$. Эту задачу решил Клини в 1935 году. Следуя Клини, определим вначале функцию PREFN, удовлетворяющую условиям:

$$\text{PREFN}\ f\ (\text{true}, x) = (\text{false}, x)$$

и

$$\text{PREFN}\ f\ (\text{false}, x) = (\text{false}, f\ x)$$

Тогда (PREFN f)$^{n+1}$ (true, x) = (false, f^n x) и функцию предшествования можно задать следующим образом:

$$\text{PRE } n \triangleq \lambda f\ x.\,\text{snd}(n\ (\text{PREFN}\ f)\ (\text{true}, x))$$

Функцию PREFN можно определить как:

$$\text{PREFN } \triangleq \lambda f\ p.(\text{false}, \text{if } \text{fst}\, p \text{ then } \text{snd}\, p \text{ else } f(\text{snd}\, p))$$

9.2 Рекурсивные функции

Ключевым моментом для определения рекурсивных функций в лямбда-исчислении является существование так называемых *комбинаторов неподвижной точки.* Замкнутый лямбда-терм Y называется комбинатором неподвижной точки, если для любого лямбда-терма f выполняется: $f(Y\ f) = Y\ f$. Таким образом, комбинатор неподвижной точки по заданному терму f возвращает неподвижную точку f, т.е. терм x, такой что $f(x) = x$. Первый такой комбинатор, найденный Карри, обычно обозначается как Y. Его называют также «парадоксальным комбинатором». Он определяется следующим образом:

$$Y \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Данное выражение действительно определяет комбинатор неподвижной точки, поскольку:

$$\begin{aligned} Y\ f &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ f = \\ &= (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) = \\ &= f((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) = \\ &= f(Y\ f) \end{aligned}$$

Комбнатор неподвижной точки Кари решает задачу математичеки. Однако, с вычислительной очки зрения такое определение вызывает трудности, поскольку выполняемые проебразования используют лямбда-равенство, а не редукцию, а также *обратную* β-конверсию. Поэтому для вычислительных целей более предпочтительно определение комбинатора неподвижной точки, принадлежащее Тьюрингу:

$$T \triangleq (\lambda x\ y.y\ (x\ x\ y))\ (\lambda x\ y.y\ (x\ x\ y))$$

Будем обозначать комбинатор неподвижной точки Тьюринга как *Y*. Он может быть исползован в определении рекурсивных функций. Рассмотрим в качестве примера функцию факториал:

$$\mathrm{fact}(n) = \mathrm{if}\ \ \mathrm{ISZERO}\, n\ \ \mathrm{then}\ 1\ \mathrm{else}\ n * \mathrm{fact}(\mathrm{PRE}\, n)$$

Преобразуем это выражение к следующему эквивалентному виду:

$$\mathrm{fact} = (\lambda f\ n.\mathrm{if}\ \ \mathrm{ISZERO}\, n\ \ \mathrm{then}\ 1\ \mathrm{else}\ n * f(\mathrm{PRE}\, n))\ \ \mathrm{fact}$$

Т.о. fact = F fact, где

$$F = \lambda f\ n.\mathrm{if}\ \ \mathrm{ISZERO}\, n\ \ \mathrm{then}\ 1\ \mathrm{else}\ n * f(\mathrm{PRE}\, n)$$

Отсюда можно заключить, что fact является неподвижной точкой функции F т.е., fact = Y F.

Данная техника обобщается для определения взаимно рекурсивных функций, т.е. набора функций, определения которых взаимно зависят друг от друга.

$$\begin{aligned} f_1 &= F_1\ f_1 \ldots f_n \\ f_2 &= F_2\ f_1 \ldots f_n \\ \ldots &= \ldots \\ f_n &= F_n\ f_1 \ldots f_n \end{aligned}$$

Эти выражения, используя кортежи, можно преобразовать к одному равенству:

$$(f_1, f_2, \ldots, f_n) = (F_1\ f_1 \ldots f_n, F_2\ f_1 \ldots f_n, \ldots, F_n\ f_1 \ldots f_n)$$

Положив $t = (f_1, f_2, \ldots, f_n)$ и введя $F = \lambda\, q.\ (F_1 q, F_2 q, \ldots, F_n q)$ получим

$t = F\ t$.

Отсюда, снова с помощью селекторов $f_i = (t)_i$, можно получить отдельные компоненты кортежа *t*.

9.3 Именованные выражения

Возможность опеделять безымянные функции является преимуществом лямбда-исчисления. Было показано, что рекурсивные функции можно определить, не вводя каких имен. Тем не менее, возможность давать имена выражениям полезна, поскольку позволяет избегать переписывания именованных выражений. Простая форма именования может быть введена как форма «синтаксическо-

го сахара» над чистым лямбда-исчисление:

$$\text{let } x = S \text{ in } T \triangleq (\lambda x.T)\ S$$

Можно связать несколько выражений с переменными в последовательном или параллельном стиле. В первом случае let-конструкции вкладываются друг в друга. Во втором используется запись:

$$\text{let } \{x_1 = S_1,\ x_2 = S_2,\ \ldots,\ x_n = S_n\} \text{ in } T$$

Это выражение можно рассматривать как «синтаксический сахар» для:

$$(\lambda(x_1\ ,\ \ldots\ ,\ x_n)T)(S_1\ \ldots\ S_n)$$

Вместо префиксной формы с let можно ввести постфиксный вариант:

$$T \text{ where } x\ =\ S$$

С помощью let-нотации можно определять функции, введя соглашение, что

$$\text{let } f\ x_1\ x_2\ \ldots\ x_n = S \text{ in } T$$

означает

$$\text{let } f = \lambda x_1\ x_2\ \ldots\ x_n.S \text{ in } T$$

Для определения рекурсивных функций можно ввести соглашение по использованию комбинатора неподвижной точки, т.е. let f = F f in S означает

let $f = Y\ F$ in S.

Рассмотренная система «синтаксического сахара» позволяет поддерживать понимаемый человеком синтаксис для чистого лямбда-исчисления, и с ее помощью можно писать программы на языке, близком к настоящему языку программирования, такому как Haskell.

Программа представляет собой единственное выражение. Однако, имея в распоряжении механизм let для связывания подвыражений с именами, более естественно рассматривать программу как набор *определений* вспомогательных функций, за которыми следует само выражение. В Haskell введено соглашение опускать конструкцию let в определениях верхнего уровня.

Определения вспомогательных функций можно интерпретировать как систему уравнений в обычном математическом смысле. Они не задают явных указа-

ний, каким образом вычислять выражения. По этой причине функциональное программирование часто объединяют с логическим и включают в класс *декларативных* методов программирования. Программа определяет ряд желаемых свойств результата и оставляет машине найти способ его вычисления.

В действительности выражение-программа вычисляется с помощью «развертывания» всех конструкций до уровня чистого лямбда-исчисления и последующего применения β-конверсий. Хотя в самой программе нет информации по ее выполнению, при ее составлении подразумевается некоторая стратегия исполнения.

Кроме того, необходимо ввести соглашение по используемым редукционным стратегиям, поскольку выбор различных β-редексов может привести к различному поведению программы в смысле ее завершаемости.

10 Типы

Причина введения типов в лямбда-исчисление и в языки программирования возникает как с точки зрения логики так и пограммироваия.

Лямбда-исчисление разрабатывалось для формализации языка математики. Черч предполагал включить в лямбда-исчисление теорию множеств. По заданному множеству S можно определить его характеристическую функцию χ_s, такую что:

$$\chi_S(x) = \begin{cases} \text{true}, & x \in S \\ \text{false}, & x \notin S \end{cases}$$

С другой стороны, имея унарный предикат P, можно определить множество таких x, что $P(x)$ = true. Однако определение предикатов (и, следовательно, множеств) в виде произвольных лямбда-выражений может привести к противоречиям.

Рассмотрим парадокс Рассела. Определим множество R, состоящее из всех множеств, которые не содержат себя в качестве элемента:

$$R = \{x | x \notin x\}$$

Тогда $R \in R \Leftrightarrow R \notin R$, что является противоречием. В терминах лямбда-исчисления можно определить предикат $R = \lambda x.\neg(x\ x)$ и получим противо-

речие $R\ R = \neg(R\ R)$. Выражение *R R* является неподвижной точкой оператора отрицания.

Парадокс Рассела в лямбда-исчислении возникает из-за того, что мы применяем функцию к самой себе. Однако это не обязательно приводит к парадоксу: например, функция идентичности *λx. x* или константная функция *λx. y* не приводят к противоречиям. Более четкое представлене функций, обозначаемых лямбда-термами, требует точного зння области их определения и значений и применения только к аргументам, принадлежащим областям их определения. По этим причинам Рассел предложил ввести понятие типа.

Типы возникли также в языках программирования. Одной из причин этого была эффективность: зная о типе переменной, можно генерировать более эффективный код и более эффективно использовать память.

Помимо эффективности типы предоставляют возможность статической проверки программ. Типы можно использовать также для достижения модульности и скрытия данных.

Существуют также безтиповые языки, как императивные, так и функциональные. Некоторые языки являются *слабо типизированными,* когда компилятор допускает некоторое несоответствие в типах и сам делает необходимые преобразования. Существуют языки (например, Python), которые выполняют проверку типов *динамически,* во время выполнения программы, а не во время компиляции.

Определение системы типов, которая позволяет выполнять статические проверки и в то же время не накладывает жестких ограничений - сложная задача. Система типов, использующаяся в таких языках, как Haskell - предоставляет возможность *полиморфизма,* когда одна и та же функция может использоваться с различными типами. Это оставляет возможность статических проверок, предоставляя в то же время преимущества слабой или динамической типизации. Более того, программист не обязан указывать типы в Haskell: компилятор может вывести наиболее общий тип любого выражения и отвергнуть выражения, не имеющие типа.

10.1 Типизированное лямбда-исчисление

Модифицируем лямбда-исчисление введя в него понятие типа. Каждый лямбда-терм должен иметь *тип,* причем терм *S* можно применить к терму *T* в комбинации *S T,* если их типы правильно соотносятся друг с другом, т.е. *S* имеет тип функции $\sigma \longrightarrow \tau$ и *T* имеет тип σ. В результате, *S T,* имеет тип τ. Это свойство называется *сильной типизацией.* Приведение типов не допускается.

Будем использовать запись вида $T :: \sigma$ для утверждения «*T* имеет тип σ».

10.1.1 Базовые типы

Предположим, что имеется некоторый набор *базовых,* типов, таких как Bool или Integer. Из них можно конструировать составные типы с помощью *конструкторов типов,* являющихся, по сути, функциями. Дадим следующее индуктивное определение множества типов Ty_C, основывающихся на множестве базовых типов *C*:

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\sigma \in Ty_C, \tau \in Ty_C}{\sigma \to \tau \in Ty_C}$$

Предполагается, что функциональная стрелка —> ассоциативна вправо, т. е. $\sigma \longrightarrow \tau \longrightarrow \nu$ означает $\sigma \longrightarrow (\tau \longrightarrow \nu)$.

Можно расширить систему типов двумя способами. Во-первых, можно ввеси понятие *типовых, переменных,* являющихся средством для реализации полиморфизма. Во-вторых, можно ввести дополнительные конструкторы типов, помимо функциональной стрелки, например конструктор `x` для типа пары значений. В этом случае необходимо добавить правило:

$$\frac{\sigma \in Ty_C, \tau \in Ty_C}{\sigma \times \tau \in Ty_C}$$

Можно ввести именованные конструкторы произвольной арности. Будем использовать запись $Con\ (\alpha_1, \ldots, \alpha_n)$ для применения *n*-арного конструктора *Con* к аргументам α_i.

Важным свойством типов является то, что $\sigma \longrightarrow \tau \neq \sigma$ т.е. тип не может совпа-

дать ни с каким своим синтаксически правильным типовым подвыражением. Это исключает возможность применения терма к самому себе.

10.1.2 Типизации по Черчу и Карри

Существуют два основных подхода к определению типизированного лямбда-исчисления. Первый подход, принадлежащий Черчу это *явная* типизация. Каждому терму сопоставляется единственный тип. Это означает, что в процессе конструирования термов, нетипизированные термы модифицируются с помощью дополнительной характеристики - типа. Для констант этот тип задан заранее, но переменные могут иметь любой тип. Правила корректного формирования термов выглядят следующим образом:

$$\frac{}{v :: \sigma}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c :: \sigma}$$

$$\frac{S :: \sigma \to \tau, T :: \sigma}{S\,T :: \tau}$$

$$\frac{v :: \sigma, T :: \tau}{\lambda v.T :: \sigma \to \tau}$$

Однако мы будем использовать для типизации подход Карри, который является *неявным.* Термы могут иметь или не иметь типа, и если терм имеет тип, то он может быть не единственным. Например, функция идентичности *λx. x* может иметь любой тип вида *σ —> σ.* Такой подход более предпочтителен, поскольку, во-первых, он соответствует используемому в языках типа Haskell понятию полиморфизма, а во-вторых, позволяет не задавать типы явным образом.

Определим понятие типизируемости по отношению к *контексту,* т. е. конечному набору предположений о типах переменных. Будем записывать:

$$? \vdash T :: \sigma$$

чтобы обозначить, что «в контексте ? терм *T* может иметь тип *σ».* Будем употреблять запись $\vdash T :: \sigma$ или просто $T :: \sigma$, если суждение о типизации выполняется в пустом контексте. Элементы контекста ? имеют вид *v* :: *σ,* т. е. они представляют собой предположения о типах переменных. Будем предполагать, что в ? нет противоречивых предположений.

10.1.3 Формальные правила типизации

$$\frac{v :: \sigma \in ?}{? \vdash v :: \sigma}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c :: \sigma}$$

$$\frac{? \vdash S :: \sigma \to \tau, ? \vdash T :: \sigma}{? \vdash S\,T :: \tau}$$

$$\frac{? \cup \{v :: \sigma\} \vdash T :: \tau}{? \vdash \lambda v.T :: \sigma \to \tau}$$

Эти правила являются индуктивным определением отношения типизируемости. Для примера рассмотрим типизирование функции идентичности *λx. x.* По правилу для переменных имеем:

$$\{x :: \sigma\} \vdash x :: \sigma$$

и отсюда по последнему правилу получаем:

$$\emptyset \vdash \lambda x.x :: \sigma \to \sigma$$

В силу соглашения о пустых контекстах получаем *λx. x* :: *σ* —> *σ.* Можно показать, что все преобразования лямбда-термов сохраняют типы.

10.2 Полиморфизм

Система типов по Карри обеспечивает *полиморфизм,* в том смысле, что терм может иметь различные типы. Необходимо различать концепции полиморфизма и *перегрузки.* Оба этих термина означают, что выражение может иметь несколько типов. Однако в случае полиморфизма все типы сохраняют струкурное сходство. Например, функция идентичности может иметь тип *σ* —> *σ, τ* —> *τ* или (*σ* —> *τ*) —> (*σ* —> *τ*). При перегрузке функция может иметь различные типы, не связанные друг с другом структурным сходством.

10.2.1 let-полиморфизм

Рассмотренная система типов приводит к некоторым ограничениям на полиморфизм. Например, приемлемо следующее выражение:

if *(λx. x)* true then *(λx.x)* 1 else 0

Если использовать правила типизации, то можно получить, что это выражение

имеет тип int. Два экземпляра функции идентичности имеют типы bool —>bool и int —> int соответственно.

Рассмотрим выражение:

$$\text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0$$

Согласно определению, это иной способ записи для

$$(\lambda I.\ \text{if } I \text{ true then } I\ 1 \text{ else } 0)\ (\lambda x.\ x)$$

Однако это выражение не может быть типизировано. В нем присутствует *единственный* экземпляр функции идентичности, и он должен иметь единственный тип.

Для преодоления этого ограничения добавим правило типизации, в котором let-конструкции рассматривается как первичная:

$$\frac{? \vdash S :: \sigma \text{ и } ? \vdash T[x := S] :: \tau}{? \vdash \text{let } x = S \text{ in } T :: \tau}$$

Это правило определяет *let-полиморфизм.*

10.2.2 Наиболее общие типы

Некоторые выражения не имеют типа, например, $\lambda f.\ f\ f$ или $\lambda f.$ (f true, f.1) Типизируемые выражения обычно имеют много типов, хотя некоторые имеют только один.

Имеет место утверждение: для каждого типизируемого выражения существует *наиболее общий тип* или *основной тип,* и все возможные типы выражения являются экземплярами этого наиболее общего типа. Прежде чем сделать это утверждение строгим, необходимо ввести некоторую терминологию.

Введем понятие *типовых переменных.* Типы могут быть сконструированы с помощью применения конструкторов типа либо к типовым константам, либо к переменным. Будем использовать буквы α и β для типовых переменных, а σ и τ - для произвольных типов. Определим понятие замены типовой переменной на некоторый тип. Будем использовать ту же нотацию, что и при подстановке термов. Например:

$$(\sigma \text{ —> boo1})[\ \sigma := (\sigma \text{ —> } \tau)] = (\sigma \text{ —> } \tau) \text{—> bool}$$

Расширим это определение, добавив параллельные подстановки:

$$\begin{aligned}
\alpha_i[\alpha_1 := \tau_1, \ldots, \alpha_k := \tau_k] &= \tau_i \\
\beta[\alpha_1 := \tau_1, \ldots, \alpha_k := \tau_k] &= \beta, \text{ если } \alpha_i \neq \beta \text{ для } 1 \leq i \leq k \\
\text{Con } (\sigma_1, \ldots, \sigma_n)[\theta] &= \text{Con } (\sigma_1[\theta], \ldots, \sigma_n[\theta])
\end{aligned}$$

Можно рассматривать типовые константы как 0-арные конструкторы, т. е. считать, что int задается как int (). Имея определение подстановки, можно счиать, что тип σ является *более общим,* чем тип σ' и записывать этот факт как $\sigma \preceq \sigma'$. Это отношение выполняется тогда и только тогда, когда существует набор подстановок θ такой, что $\sigma' = \sigma\,\theta$. Например:

$$\begin{aligned}
\alpha &\preceq \alpha \\
\alpha \to \alpha &\preceq \beta \to \beta \\
\alpha \to \text{bool} &\preceq (\beta \to \beta) \to \text{bool} \\
\beta \to \alpha &\preceq \alpha \to \beta \\
\alpha \to \alpha &\not\preceq (\beta \to \beta) \to \beta
\end{aligned}$$

Имеет место:

Теорема 4. *Каждый типизируемый терм имеет некоторый основной тип, т. е. если $T :: \tau$, то существует некоторый σ, такой что $T :: \sigma$ и для любого σ', если $T :: \sigma'$, то $\sigma \preceq \sigma'$.*

Доказательство этой теоремы конструктивно: оно дает конкретную процедуру для поиска основного типа. Эта процедура известна как *алгоритм Хиндли-Милнера.* Все реализации языков программирования типа Haskell включают в себя вариант этого алгоритма. Выражения в них могут быть сопоставлены их основному типу либо отвергнуты как нетипизируемые.

10.3 Сильная нормализация

Справедлива следующая теорема о сильной нормализации:

Теорема 5. *Каждый типизируемый терм имеет нормальную форму и каждая возможная последовательность редукций, начинающаяся с типизируемого терма, завершается.*

Функциональная программа, соблюдающая дисциплину типизации, может быть вычислена любым образом, и она всегда завершится в единственной нор-

мальной форме (единственность следует из теоремы Черча-Россера, которая выполняется и для типизированного лямбда-исчисления). Однако возможность создавать незавершающиеся программы существенна для полноты по Тьюрингу для определения произвольных вычислимых и полных функций.

Чтобы обеспечить полноту по Тьюрингу, добавим способ определения произвольных рекурсивных функций, которые *являются* правильно типизированными. Для этого определим полиморфный оператор рекурсии для всех типов вида:

$$Rec :: ((\sigma \longrightarrow \tau) \longrightarrow (\sigma \longrightarrow \tau)) \longrightarrow \sigma \longrightarrow \tau$$

Кроме того, добавим правило редукции:

$$Rec\ F \longrightarrow F\ (Rec\ F)$$

для любого $F :: (\sigma \longrightarrow \tau) \longrightarrow (\sigma \longrightarrow \tau)$.

Будем считать, что рекурсивные определения функций интерпретируются с использованием этих операторов рекурсии.

11 Отложенные вычисления

С теоретической точки зрения, нормальный порядок редукции выражений наиболее предпочтителен, поскольку если какая либо стратегия завершается, завершится и она. Такая стратегия известна как *вызов по имени*. С практической точки зрения такая стратегия имеет существенные недостатки.

Рассмотрим выражение

$$(\lambda x.\ x + x + x)\ (10 + 5)$$

Нормальная редукция приводит это выражение к следующему виду:

$$(10 + 5) + (10 + 5) + (10 + 5)$$

Таким образом, необходимо трижды вычислять значение одного и того же выражения. На практике это недопустимо. Существуют два основных подхода к решению этой проблемы.

При первом подходе отказываются от нормальной стратегии и вычисляют аргументы функций до того, как передать их значения в функцию. Такой подход называют *передачей по значению*. При этом вычисление некоторых выражений, завершающееся при нормальной стратегии, может привести к бесконечной по-

следовательности редукций. Однако такая стратегия позволяет получать эффективный машинный код.

При другом подходе, использующемся в Haskell, нормальная стратегия редукций сохраняется. Однако различные возникающие подвыражения разделяются и никогда не вычисляются более одного раза. Такой способ вызова называется *ленивым* или *вызовом по необходимости,* поскольку выражения вычисляются только тогда, когда их значения действительно необходимы для работы программы.

Оптимизирующие компиляторы языка Haskell выявляют места программы, в которых можно обойтись без отложенных вычислений.

Конструкторы данных в Haskell можно считать функциями (единственное отличие от функций заключается в том, что их можно использовать в шаблонах при сопоставлении с образцом.) В сочетании с «ленивым» вызовом это позволяет определять бесконечные структуры данных. Примером служит бесконечный список целых чисел, начиная с числа *n*:

numsFrom *n* = *n* : numsFrom (*n*+1)

Программа, реализующая отложенные вычисления, может быть представлена в виде конструкции *g* (*f* input), где *g* и *f* - некоторые функции. Они выполняются строго синхронно, *f* запускается только тогда, когда *g* требуется прочитать данные. После этого *f* приостанавливается, и невыполняется, до тех пор пока *g* вновь не попытается прочитать следующую группу входных данных. Если *g* заканчивается, не прочитав весь вывод *f*, то *f* прерывается, *f* может быть не завершаемой программой, создающей бесконечный вывод, так как она будет остановлена, как только завершится д. Это позволяет отделить условия завершения от тела цикла, что является мощным средство модуляризации.

Этот метод называется "ленивыми вычислениями" так как *f* выполняется настолько редко, насколько это возможно. Он позволяет осуществить модуляризацию программы как генератора, который создает большое количество возможных ответов, и селектора, который выбирает подходящие.

Рассмотрим алгоритм Ньютона-Рафсона для вычисления квадратного корня. Этот алгоритм вычисляет квадратный корень числа *z,* начиная с начального при-

ближения a_0. Он уточняет это значение на каждом последующем шаге, используя правило:

$$a_{n+1} = (a_n + z/a_n)/2$$

Если приближения сходятся к некоторому пределу a, то $a = (a + z/a)/2$, то есть $a \bullet a = z$ или $a = \sqrt{z}$.

Программа проводит проверку на точность (eps) и останавливается, когда два последовательных приближения отличаются меньше чем на eps. При императивном подходе алгоритм обычно программируется следующим образом (на языке Си):

```
x = a0;
do{
    y = x;
    x = (x + z/x) / 2;
} while (abs(x-y) > eps)
// теперь x = квадратному корню из z
```

Предсавим эту программу в модульной форме, используя ленивые вычисления.

Так как алгоритм Ньютона-Рафсона вычисляет последовательность приближений, будем использовать для этого в программе список приближений. Каждое приближение получено из предыдущего с помощью функции

next $z\ x = (\ x + z\ /\ x\ )\ /2$

Данная функция, преобразует каждое текущее приближение в следующее. Обозначим эту функцию f, тогда последовательность приближений будет: [a_0, $f\ a_0$, $f(f\ a_0)$, $f(f(f\ a_0))$, . . .] . Определим функцию, вычисляющую данную последовательность:

iterate $f\ x = x$: iterate $f\ (f\ x)$

Тогда список приближений можно вычислить следующим образом:

iterate (next z) $a0$

iterate это пример функции с «бесконечным» выводом. Однако фактически

будет вычислено не больше приближений, чем требуется остальным частям программы. Таким образом, мы имеем потенциальную бесконечность.

Завершающая часть программы это функция within, которая использует оценку точности eps и список приближений. Она просматривает список и находит два последовательных приближения, отличающихся не более чем на eps.

```
within eps (a:b:rest) =
    if abs(a-b) <= eps
          then b
          else within eps (b : rest)
```

Окончательно, получаем:

```
sqrt a0 eps z = within eps (iterate (next z) a0)
```

Мы можем объединять полученные части программы различными способами. Одна из модификаций заключается в использовании относительной погрешности вместо абсолютной. Она применима как для очень малых чисел (когда различие между последовательными приближениями маленькое), так и для очень больших (при округлении ошибки могут быть намного большими, чем eps). Заменим within функцией

```
relative eps (a:b:rest) =
  if abs(a-b) <= eps*abs(b)
       then b else
       relative eps (b:rest)
```

Т.о. получим новую версию sqrt

```
relative sqrt a0 eps z = relative eps (iterate (next z) a0)
```

Нет необходимости переписывать часть, которая генерирует приближения.

Запишем алгоритм определения корня функции *f* методом касательных. Если начальное приближение равно x0, то следующее приближение вычисляется по формуле $x_1 = x - f(x) / f'(x)$. На языке Haskell этот алгоритм можно представить следующим образом:

```
root f diff x0 eps = within eps (iterate (next f diff) x0)
                              where next f diff x = x - f x / diff x
```

Для работы этого алгоритма необходимо указать две функции: *f*, вычисляющню значение функции *f* и diff, вычисляющую значение *f* '. Например, чтобы найти положительный корень функции $f(x) = x^2 - 1$ с начальным приближением 2, составим следующее выражение:

```
root f diff 2 0.001 where f x = x^2 - 1
                            diff x = 2*x
```

12 Классы типов

Классы типов позволяют указывать типы являющиеся *экземплярами* классов, и предоставлять определения ассоциированных с классом *операций,* перегруженных для каждого типа. Например, класс, содержащий оператор равенства имеет вид:

```
class Eq a where
  (= =) :: a —> a —> Bool
```

Здесь Eq - имя определяемого класса. = = - операция, определенная в классе. Это определение означает, что «тип а является экземпляром класса Eq, если для него существует перегруженный оператор = =».

Консрукция Eq *a* выражает ограничение на тип и называется *контекстом.* Контексты располагаются перед описаниями типов.

Например:

```
elem :: (Eq a) => a —> [a] —> Bool
```

Эта запись выражает факт, что функция elem определена только для тех тпов, для которых определен оператор равенства. Указать типы принадлежащие классу Eq и определить функции сравнения для этих типов можно с помощью *определения принадлежности:*

```
instance Eq Integer where
    x = = y = x `integerEq` y
```

Определение оператора = = называется *методом.* Функция integerEq сравнивает на равенство два целых числа. В общем случае в правой части определения допус-

тимо любое выражение, как и при обычном определении функции.

При определении принадлежности можно использовать контекст. Пусть определен рекурсивный тип для дерева:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Тогда можно определить оператор поэлементного сравнения деревьев:

```
instance (Eq a) => Eq (Tree a) where
    Leaf a == Leaf b = a == b
    (Branch l1 r1) == (Branch l2 r2) = (l1 == l2) && (r1 == r2)
    _ == _ = False
```

В стандартных библиотеках языка Haskell определено много классов типов. Класс Eq в действительности определен следующим образом:

```
class Eq a where
    (==), (/=) :: a —> a —> Bool
    x /= y = not (x == y)
```

Здесь определены две операции: равенство и неравенство. При этом использовано определение *метода по умолчанию* для неравенства. Если метод для какого-либо экземпляра класса пропущен в определении принадлежности, используется метод, определенный в классе, если он существует.

Haskell также поддерживает *расширения класса.* Например, можно определить класс Ord, который *наследует* все операции класса Eq, и кроме того, определяет новые операторы сравнения и функции минимум и максимум:

```
class (Eq a) => Ord a where
   (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
```

Говорят, что Eq является *суперклассом* класса Ord (соответственно, Ord является *подклассом* Eq).

13 Монады

Понятие монад является одним из важнейших в языке Haskell.

Одной из простейших монад является тип Maybe. Он определен следующим об-

разом:

```
data Maybe a = Nothing | Just a
```

Этот тип может использоваться для представления значения функций, включая случай, когда значение отсутсвует. Например, функция *f* с сигнатурой

```
f :: a —> Maybe b
```

принимает значение типа *a* и возвращает результат типа *b* (обернутый в конструктор Just), либо может не вычислить значения и тогда вернет значение Nothing.

Типичным примером такого рода функций служат функции для запросов к базе данных. В случае, если данные, удовлетворяющие критериям запроса, существуют, их следует предоставить; в противном случае возвращается Nothing.

Рассмотрим базу данных, содержащую информацию об адресах людей. Она устанавливает соответствие между полным именем человека и его адресом. Для простоты предположим, что имя и адрес задаются строками. Тогда базу данных можно описать следующим образом:

```
type AddressDB = [(String, String)]
```

Таким образом, база данных представляет собой список пар, первым компонентом которых будет имя, а вторым - адрес. Определим функцию

getAddress, которая по заданному имени возвращает адрес:

```
getAddress :: AddressDB —> String —> Maybe String
getAddress [] _ = Nothing
getAddress ((name,address):rest) n | n == name = Just address
                                   | otherwise = getAddress rest
```

n

Для имен, присутствующих в базе, функция возвращает соответствующий адрес. Если такого имени в базе нет, возвращается Nothing.

Предположим, что имеется еще одна база данных, содержащая информвцию об адресах и номерах телефонов:

```
type PhoneDB = [(String,Integer)]
```

Пусть имеется функция getPhone, по адресу возвращающая телефон, реализованная с помощью типа Maybe аналогично функции getAddress. Требуется определить функцию, возвращающую телефон по имени владельца.

Значение Nothing эта функция может вернуть в следующих случаях:

1. Указанное имя не содержится в базе адресов.
2. Адрес, соответствующий указанному имени, существует, однако он не содержится в базе телефонов.

Исходя из этих соображений, функцию getPhoneByName можно определить следующим образом:

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> Maybe Integer

getPhoneByName a p n =

    case (getAddress a n )  of
        Nothing —> Nothing
        Just address —> case (getPhone p address) of
                            Nothing —> Nothing
                            Just phone —> Just phone
```

Использованный стиль программирования может привести к ошибкам. В случае, когда уровень вложенности запросов увеличивается, растет и объем повторяющегося кода. Поэтому определим вспомогательную функцию thenMB, которая реализует использованный в функции getPhoneByName шаблон связывания функций, возвращающих значение типа Maybe.

```
thenMB :: Maybe a —> (a —> Maybe b) —> Maybe b

thenMB mB f = case mB of

                Nothing —> Nothing

                Just a —> f a
```

Эта функция принимает два аргумента: значение типа Maybe *a* и функцию, отображающую значение типа *a* в значение типа Maybe *b*. Если первый аргу-

мент содержит значение Nothing, второй аргумент игнорируется. Если первый аргумент содержит реальное значение, обернутое в конструктор Just, оно извлекается и передается в функцию, являющуюся вторым аргументом. Переопределим функцию getPhoneByName следующим образом:

```
getPhoneByName a p n = getAddress a n      'thenMB' \address —
>
                                    getPhone p address 'thenMB' phone —>
                                    Just phone
```

Вычисление этой функции можно понимать как передачу результата левого аргумента оператора 'thenMB' переменной из лямбда-абстракции правого аргумента.

Таким образом, мы определили функцию thenMB, комбинирующую вычисления, которые могут либо предоставить результат, либо отказ от его вычисления. Функция thenMB не зависит от того, какие вычисления она комбинирует, требуется только, чтобы они удовлетворяли ее сигнатуре. Этуфункцию можно использовать для любых аналогичных задач. Она определяет правило комбинирования вычислений в виде цепочки, заключающееся в том, что если одно из вычислений не выполнилось, не выполняется и вся цепочка.

Усовершенствуем рассматриваемый пример. Потребуем, чтобы в случае неудачи функция getPhone сообщала о причине по которой она не нашла телефон. Для этого определим функции, getAddress и getPhoneByName таким образом, чтобы они возвращали значение типа Value, который определим следующим образом:

```
data Value a = Value a   |   Error String
```

Значение типа Value a представляет собой либо значение типа a, обернутое в конструктор Value, либо строковое сообщение об ошибке, содержащееся в конструкторе Error. Функцию get Address определим следующм образом:

```
getAddress :: AddressDB -> String -> Value String
getAddress [] _ = Error "no address"
getAddress ((name,address):rest) n | n == name = Value address
```

```
                                    | otherwise = getAddress rest n
```

В случае ошибки getAddress вернет значение Error "no address". Аналогично можно определить и функцию getPhone, которая в случае ошибки вернет значение Error "no phone". Тогда функцию getPhoneByName можно определить следующим образом:

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> Value Integer

getPhoneByName a p n = case (getAddress a n ) of
                              Error s -> Error s
                              Value address -> case (getPhone p address) of
                                                    Error s -> Error s Value
                                                    phone -> Value phone
```

Здесь имеет место проблема, как и в предыдущем варианте. Для ее решения: определим вспомогательную функцию:

```
thenV :: Value a -> (a -> Value b) -> Value b

thenV mV f = case mV of
                    Error s -> Error s
                    Value v -> f v
```

С использованием этой функции можно упростить определение getPhoneByName:

```
getPhoneByName a p n = getAddress a n `thenV` \address ->
                              getPhone p address `thenV` \phone ->
                              Value phone
```

Имеется сходство в определении функций thenMB и thenV, а также в определениях функции getPhoneByName. Тип Value также представляет пример монады.

Обобщим рассмотренную задачу. Предположим, что одному человеку может соответствовать несколько адресов, а одному адресу - несколько телефонов. Тогда функции get Phone, getAddress и getPhoneByName должны возвращать списки. Их сигнатуры имеют вид:

```
getAddress :: AddressDB -> String -> [String]
getPhone :: PhoneDB -> String -> [Integer]
```

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> [Integer]
```

Пусть функции getPhone и getAddress в случае неудачи возвращают пустые списки, а в случае успешного поиска - списки значений. Используя эти функции можно определить функцию getPhoneByName. Однако вначале определим вспомогательную функцию:

```
thenL :: [a] -> (a -> [b]) -> [b]

thenL mL f = case mL of
                  [] -> []
                  (x:xs) -> f x ++ thenL xs f
```

Тогда:

```
getPhoneByName a p n = getAddress a n `thenL` \address ->
                              getPhone p address `thenL` \phone ->
                              [phone]
```

Список также является монадой.

Монада - это тип данных, подразумевающий определенную стратегию комбинирования вычислений

С другой стороны, монаду можно рассматривать как контейнерный тип, т. е. как тип, значения которого содержат в себе некоторое количество элементов другого типа.

В стандартной библиотеке языка Haskell определен класс типов, являющихся монадами.

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a

    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

Таким образом тип m является монадой, если для него определены указанные функции и операторы. При этом необходимо определить только функцию return и оператор »= ; для остальных функций и операторов даны определения по умолчанию.

Инфиксный оператор »= служит обозначением таких функций как thenMB, thenV и thenL. Функция return предназначена для создания монадического типа. Она «вкладывает» значение в монаду, которая рассматривается как контейнер.

Для типа Maybe имеется определение принадлежности вида:

```
instance Monad Maybe where

    (»=) = thenMB

    return a = Just a
```

Таким образом, функцию getPhoneByName, возвращающую значения типа Maybe Integer, можно определить следующим образом:

```
getPhoneByName a p n = getAddress a n    »= \address ->
                       getPhone p address »= \phone ->
                       return phone
```

Можно дать определения принадлежности и для других рассмотренных монад:

```
instance Monad Value where

    (»=) = thenV

    return a = Value a

instance Monad [ ] where

    (»=) = thenL

    return a = [a]
```

Оператор » используется для комбинирования вычислений, которые не

зависят друг от друга.

Поддержка монад встроена непосредственно в язык Haskell. Так называемая do-нотация облегчает запись монадических вычислений. Она позволяет записывать псевдо-императивный код с именованными переменными. Результат монадического вычисления может быть «присвоен» переменной с помощью оператора <-. Выражение справа от этого оператора должно иметь монадический тип m a. Выражение слева представляет собой образец, с которым сопоставляется значение *внутри* монады. В последующих монадических вычислениях можно использовать переменные, связанные в результате этого сопоставления.

Функция getPhoneByName при использовании do нотации имеет вид:

```
getPhoneByName a p n =

    do address <- getAddress a n
       phone <- getPhone p address
       return phone
```

Ключевое слово do вводит правило выравнивания, заключающееся в том, что первый символ, появляющийся после этого слова, задает колонку, в которой должны начинаться последующие строки. Использование символов {, } и ; позволяет не использоваь правило выравнивания:

do-нотация использует простые правила преобразования в стандартную нотацию:

1. Конструкция вида do {el; e2} преобразуется в el » e2

2. Конструкция вида do {x <- el; e2} преобразуется в el »= \x -> e2

Понятие монад происходмит из области математики, которая называется теорией категорий. В ней монадами являются объекты, удовлетворяющие набору аксиом; аналогично, для монад языка Haskell должен выполняться ряд законов. Компилятор языка не проверяет их выполнение (это практически невозможно сделать автоматически) и их правильность должна быть га-

рантирована программистом. Эти законы имеют вид:

```
return x »= f ≡ f x

f »= return ≡ f

f »= (\x  -> g x »= h) ≡ (f »= g) »= h
```

Первые два закона утверждают, что return является левой и правой единицей для оператора »=, а третий закон утверждает, что оператор »= обладает свойством ассоциативности. Рассмотрим эти законы подробнее.

Если рассматривать монады как вычисления, то return x создает тривиальное вычисление, всегда возвращающее значение x. Если связать его с вычислением f, то это эквивалентно непосредственному выполнению f от x. Если этот закон выполняется, то следующие две программы должны быть эквивалентны:

```
do x <-  return a f x

do f a
```

Второй закон утверждает, что если выполнено некоторое вычисление f и его результат передан в тривиальное вычисление, то это эквивалентно просто выполнению этого вычисления. Следующие две программы должны быть эквивалентны:

```
do x <- f
        return x

do f
```

Третий закон утверждает, что независимо от того, в каком порядке сгруппированы действия (слева направо или справа налево), результат не должен меняться. Следующие программы должны быть эквивалентны:

```
do x <- f
   do y <- g x
      h y

do y <- do x <- f
```

```
        g x
    h y
```

В классе Monad определена функция fail. Она вызывается в том случае, если при сопоставлении с образцом в do-нотации произошла ошибка. По умолчанию эта функция печатает сообщение и завершает программу, однако в некоторых монадах ее имеет смысл переопределить. Например, для монады Maybe ее определяют следующим образом:

```
fail _ = Nothing
```

Аналогично для списка:

```
fail _ = []
```

Рассмотрим так называемые монады состояния. Императивные программы можно представить как ряд последовательных изменений состояния программы. В императивных языках состояние присутствует неявно; в функциональных языках его необходимо явно передавать в качестве аргумента функции.

Рассмотрим задачу генерации псевдослучайных чисел. В компьютере псевдослучайные числа генерируются с помощью рекуррентных соотношений вида $x_k = f(x_{k-1})$, где x_k - k-е число. Примером такого датчика, генерирующего вещественные числа в диапазоне от 0 до 1, в простейшем случае может служить соотношение

$$x_k = \{11\, x_{k-1} + \pi\},$$

где {} - оператор взятия дробной части.

В императивном языке можно реализовать функцию без аргументов, при каждом вызове возвращающей новое псевдослучайное число. При этом состояние датчика, представляющее в данное случае предыдущее значение последовательности, можно хранить в глобальной переменной. В языке Haskell значение функции полностью определяется ее аргументами и состояние (предыдущее значение последовательности) необходимо передавать явно. Таким образом, функция, вычисляющая объем параллелепипеда со случайными сторонами, запишется следующим образом:

```
random x = frac (pi + x * 11)

cube x0 = let x1 = random  x0
              x2 = random  x1
              x3 = random  x2
          in x1 * x2 * x3
```

Здесь в качестве параметра функции cube выступает начальное значение для генератора, а состояние передается из одного вызова функции random в другой явно.

В данном случае состояние датчика и его возвращаемого значения совпадают. Однако это не обязательно. Датчик может учитывать при вычислении очередного значения не только предыдущее значение, но и предшествующее ему. Можно разделить состояние, передаваемое от одного вызова к другому и возвращаемое значение. Это позволяет рассматривать состояние отдельно от возвращаемого значения. Сделаем следующие определения:

```
type State = ... -- Определение типа состояния

data StateT a = State -> (a, State)

random :: StateT Float
```

Тип StateT определяет значения, являющиеся *преобразователями состояния,* т. е. функциями, которые преобразуют заданное состояние в другое состояние, возвращая при этом некоторое значение. Таким образом, функция random фактически имеет тип State -> (Float,State). По заданному состоянию она возвращает два значения: очередное псевдослучайное число и новое состояние. Функция cube запишется таким образом:

```
cube s0 = let (x1,s1) = random      s0
              (x2,s2)  =  random s1
              (x3,s3)  =  random s2
          in x1 * x2 * x3
```

Тип StateT можно объявить монадой, если сделать следующие опре-

деления:

```
instance Monad StateT where
  st »= f = \s -> let (x,s1) = st s
                  in f x s1
  return a = \s -> (a,s)
```

Оператор »= связывает состояния, передавая их из одного вычисления в другое, а функция return возвращает функцию, которая оставляет состояние неизменным и возвращает указанное значение. Применяя do-нотацию, получим:

```
cube :: StateT Float
cube = do x1 <- random
          x2 <- random
          x3 <- random
          return (x1 * x2 * x3)
```

Монады состояния являются основой для определения операций ввода-вывода в языке Haskell. Пусть имеется функция getChar, которая по заданному дескриптору файла типа FileHandle возвращает очередной прочитанный из него символ:

```
getChar :: FileHandle -> Char
```

Эта функция не возвращает одно и то же значения, будучи вызвана с одним и тем же аргументом. Ее результат зависит не только от аргумента, но и от состояния файла, которое меняется при вызове этой функции.

Одним из решений этой проблемы будет явная передача этого состояния в функции, осуществляющие ввод-вывод. Таким образом, каждая из функций ввода-вывода будет принимать дополнительный параметр типа World, описывающим состояние внешнего мира, и возвращать, помимо результата, измененное состояние мира.

В Haskell определен тип IO *a*, аналогичный типу StateT, при этом вместо типа State используется World. Определение оператора связывания >>= гарантирует, что это значение не будет использоваться более одного раза. Определяется ряд *базовых операций,* осуществляющих основные операции ввода-

вывода: открыть файл, считать символ из файла и т. п. Важным свойством этих операций является то, что они возвращают значение типа IO. Например, функция для считывания символа из файла имеет тип

```
getChar :: FileHandle -> IO Char
```

Программа на языке Haskell представляет собой функцию main :: IO (), содержащую последовательность операций ввода-вывода (их называют *действиями)*. Эта функция неявно передает состояние мира в последовательность действий.

Необходимо отметить два важных свойства типа IO.

1. Это абстрактный тип данных: пользователю недоступны конструкторы данных этого типа, невозможно извлечь значение из него без использования монадических конструкций и т. д.
2. Любая функция, использующая значение, помеченное конструктором IO, должна возвращать значение, также помеченное этим конструктором. Если функция не содержит в своей сигнатуре типа IO, значит, она не выполняет никаких операций ввода-вывода: это гарантируется системой проверки типов. Таким образом, части программы, которые выполняют ввод-вывод, явно отделены от чисто функционального ядра.

Использование монад позволяет определить небольшой под-язык в рамках языка Haskell и программировать на нем практически в императивном стиле.