Э. А. Опалева
В. П. Самойленко

Языки программирования и методы трансляции

- *Грамматика и распознающий автомат – формальные модели описания синтаксиса*
- *Синтаксический анализ – ядро транслятора, определяющего его основные свойства*
- *Атрибутные грамматики – простой и удобный способ описания семантики языка*

УЧЕБНОЕ ПОСОБИЕ

BHV

Э. А. Опалева, В. П. Самойленко

Языки программирования и методы трансляции

Рекомендовано УМО по университетскому политехническому образованию для студентов высших учебных заведений, обучающихся по специальности 220400 (230105) – Программное обеспечение вычислительной техники и автоматизированных систем

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068+800.92
ББК 32.973-018.1я73
О-60

Опалева Э. А., Самойленко В. П.

О-60 Языки программирования и методы трансляции. — СПб.: БХВ-Петербург, 2005. — 480 с.: ил.

ISBN 5-94157-327-8

Учебное пособие содержит систематическое изложение теоретических основ перевода и компиляции. Рассмотрены общие вопросы разработки, описания и реализации языков программирования, формальные методы описания синтаксиса и семантики языков программирования, методы синтаксического анализа современных языков программирования. Приводится методика разработки описания перевода и пример использования этой методики для построения атрибутной транслирующей грамматики.

Для студентов и преподавателей вузов

УДК 681.3.068+800.92
ББК 32.973-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Нина Седых*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Ксения Вальская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Рецензенты:
Бузюков Л.Б., д.т.н., профессор, декан факультета сетей связи, систем коммутации и ВТ, зав. кафедрой цифровой вычислительной техники и информатики Санкт-Петербургского государственного университета телекоммуникаций.

Парфенов В.Г., д.т.н., профессор, декан факультета информационных технологий и программирования, зав. кафедрой информационных систем Санкт-Петербургского государственного университета информационных технологий, механики и оптики.

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 01.06.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 38,7.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5 Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-327-8

Оглавление

Введение

В учебных планах подготовки дипломированных специалистов, включенных в Государственный образовательный стандарт подготовки студентов по направлениям 654600 — "Информатика и вычислительная техника" (специальность 220400 — "Программное обеспечение вычислительной техники и автоматизированных систем") и 657100 — "Прикладная математика" (специальность 073000 — "Прикладная математика"), бакалавров по направлениям 552800 — "Информатика и вычислительная техника" и 510200 — "Прикладная математика и информатика" значительное место занимают дисциплины, связанные с изучением теории формальных грамматик, языков и автоматов, методов построения языковых процессоров (компиляторов и интерпретаторов).

Учебный материал, положенный в основу данного учебного пособия, читается авторами с середины 70-х годов в Санкт-Петербургском государственном электротехническом университете (ЛЭТИ им. В. И. Ульянова (Ленина)) в рамках таких дисциплин, как "Теория языков программирования и методы трансляции", "Теория вычислительных процессов и структур", "Теория формальных грамматик и автоматов", "Системное программное обеспечение". Учебное пособие отличается от аналогичных книг тем, что содержит систематическое изложение теоретических основ перевода и компиляции, ориентированное на новые образовательные стандарты и может служить основой базового курса по перечисленным ранее дисциплинам. Наиболее полно содержание данного учебного пособия соответствует учебной программе специальной дисциплины "Теория языков программирования и методы трансляции" Государственного образовательного стандарта, который изучается студентами в VII семестре. К этому времени студенты уже освоили основы информатики, дискретной математики, организации ЭВМ, алгоритмические языки и программирование, структуры данных, поэтому изучение материала данного учебного пособия не должно вызвать у них особых затруднений. Однако учебное пособие может быть полезно не только студентам, изучающим дисциплину "Теория языков программирования и методы трансляции", но и студентам других направлений и специальностей, а также тем, кто самостоятельно собирается создавать несложные языки программирования, в том числе и такие, которые используются в системах автоматизации различных прикладных областей, и разрабатывать языковые процессоры для языков программирования различных уровней сложности.

Издание подобного учебного пособия представляется полезным и своевременным, поскольку имеющаяся литература по теоретическим основам построения языковых процессоров ориентирована в основном на более подготовленных читателей. При изложении материала в учебном пособии широко используются формальные описания. Такой подход, с одной стороны, позволяет наиболее полно описывать алгоритмы, а с другой стороны, прививает студентам навыки, необходимые для самостоятельной работы с научной литературой. Материал учебного пособия излагается по принципу "от простого к сложному", все изучаемые понятия связаны в единую систему, изложение теоретического материала чередуется с рассмотрением большого количества примеров для небольших грамматик, описывающих синтаксические конструкции реальных языков программирования, и экспериментальных грамматик с небольшим числом терминалов, нетерминалов и правил вывода. Каждая глава заканчивается списком вопросов для самоконтроля и заданий для самостоятельной работы, способствующих активному усвоению материала учебного пособия.

Учебное пособие может быть использовано для односеместрового или двухсеместрового курса по теоретическим вопросам проектирования языковых процессоров. Оно состоит из введения и двенадцати глав, разделенных на четыре части.

В первой части книги рассматриваются общие вопросы разработки, описания и реализации языков программирования. Это те вопросы, которыми должен владеть любой высококвалифицированный программист, т. к. их глубокое понимание позволяет создавать более надежные и эффективные программы. Для будущих специалистов по разработке языковых процессоров вопросы, рассматриваемые в первой части, позволят получить прочные знания как о математических моделях, на которых основываются языки программирования, так и о методах и способах их реализации.

Первая часть учебного пособия состоит из двух глав. В *главе 1* приводятся основные парадигмы языков программирования и критерии, наиболее часто используемые для критической оценки существующих и проектируемых языков программирования. В этой главе рассматриваются основные конструкции языков программирования: объекты данных и механизмы их типизации, типы и структуры данных, выражения и управляющие структуры. При рассмотрении основных конструкций языков программирования основное внимание уделяется не синтаксису этих конструкций, а их семантике. В *главе 2* обсуждаются основные методы формального описания синтаксиса языков программирования (форма Бэкуса-Наура и ее модификации, синтаксические диаграммы Вирта), рассматриваются вопросы, связанные с описанием контекстных условий языков программирования. Также в этой главе дается представление о наиболее известных методах формального определения динамической семантики: W-грамматике, операционной, аксиоматической и денотационной семантике.

Вторая часть учебного пособия посвящена формальным методам описания синтаксиса языка. В *главе 3* определяются формальные грамматики и формальные языки, приводится классификация формальных грамматик по Хомскому, рассматриваются эквивалентные преобразования КС-грамматик (исключение бесполезных символов, ε-правил, цепных правил, левой рекурсии), неоднозначность грамматик, свойства КС-языков и свойства детерминированных КС-языков. В *главе 4* дается определение распознающего автомата, рассматриваются типы распознающих автоматов и языки, допускаемые распознающими автоматами. Подробно рассматриваются конечные автоматы и преобразователи, способы их задания, связь автоматных грамматик и конечных автоматов, решение проблемы принадлежности, пустоты и эквивалентности для конечных автоматов. В *главе 5* определяются автоматы с магазинной памятью и их разновидности (расширенные МП-автоматы, детерминированные МП-автоматы), преобразователи с магазинной памятью, рассматривается связь между МП-автоматами и КС-грамматиками. В учебном пособии рассматриваются только те аспекты теории автоматов, которые имеют отношение к построению языковых процессоров, поэтому ряд ее понятий опущен, и, следовательно, учебное пособие нельзя использовать как исчерпывающий курс по теории автоматов. Однако студенты, изучившие материал данного учебного пособия, впоследствии значительно проще воспримут курс теории автоматов, и наоборот: знание основ теории автоматов поможет студентам быстрее освоить материал, изложенный в данном учебном пособии.

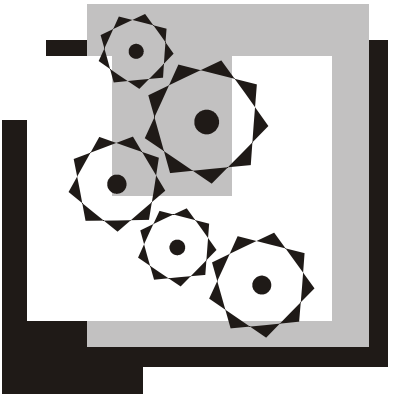
Третья часть учебного пособия посвящена методам синтаксического анализа современных языков программирования. Грамматики, используемые для задания синтаксиса этих языков, позволяют описывать большинство синтаксических конструкций и вместе с тем допускают применение простых детерминированных алгоритмов синтаксического анализа. В *главе 6* даются определение и алгоритмы нисходящего и восходящего разбора, рассматриваются вопросы моделирования работы недетерминированных преобразователей с магазинной памятью, лежащих в основе нисходящих и восходящих синтаксических анализаторов, приводится малоизвестный детерминированный алгоритм Эрли, позволяющий построить правый разбор для входной цепочки, порождаемой произвольной КС-грамматикой. В *главе 7* рассматриваются алгоритмы разбора для $LL(1)$-грамматик ("1-предсказывающий" алгоритм и метод рекурсивного спуска), в *главе 8* — алгоритм разбора "перенос-свертка" и его использование для некоторых подклассов $LR(k)$-грамматик ($LR(0)$- и $SLR(1)$-грамматик). В *главе 9* рассматривается использование алгоритма "перенос-свертка" для грамматик предшествования (простого, слабого, операторного). В конце главы приводится описание языка Флойда-Эванса, ориентированного на разработку синтаксических анализаторов, и рассматривается пример применения этого языка для построения детерминированного восходящего анализатора. При отборе и изложении материала *глав 7—9* большое внимание было уделено не просто методам

синтаксического разбора, а возможности их расширения для описания синтаксически управляемого атрибутного перевода с помощью атрибутных транслирующих грамматик.

В четвертой части учебного пособия излагаются формальные методы описания и реализации синтаксически управляемого перевода. *Глава 10* посвящена промежуточным формам представления программы после этапа синтаксического анализа. В ней рассматриваются как традиционные (польская инверсная запись, тетрады, триады, косвенные триады), так и современные (byte-code JVM) формы представления. В *главе 11* дается формальное определение перевода. В качестве формальных моделей описания перевода рассматриваются схемы синтаксически управляемого перевода, транслирующие грамматики и атрибутные транслирующие грамматики. Приводится методика разработки описания перевода и пример использования этой методики для построения атрибутной транслирующей грамматики. В *главе 12* рассмотрены вопросы разработки и реализации синтаксически управляемого перевода. В этой главе описано, каким образом нужно доработать рассмотренные в третьей части учебного пособия алгоритмы нисходящего и восходящего разбора и каким требованиям при этом должна удовлетворять атрибутная транслирующая грамматика, чтобы можно было выполнять синтаксически управляемый перевод заданным методом.

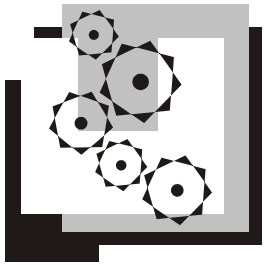
Введение, *главы 1*, *2*, *4*, *5*, *12*, *разд. 7.5*, *10.4*, *11.5—11.6* написаны Эльвирой Александровной Опалевой, а *главы 3*, *6*, *8*, *9*, *разд. 7.1—7.4*, *10.1—10.3*, *11.1—11.4* — Владимиром Петровичем Самойленко. Общее редактирование учебного пособия выполнено Э. А. Опалевой.

В заключение авторы хотят выразить признательность своим родным за их помощь и поддержку в течение всего времени работы над книгой, благодаря которым и смог появиться на свет этот труд. Также хочется выразить благодарность сотрудникам издательства "БХВ-Петербург" Елене Кондаковой, Людмиле Еремеевской и Нине Седых за их терпеливое и доброжелательное отношение в процессе подготовки текста учебного пособия и его корректуры, которые авторам не всегда удавалось делать в срок.



Часть I

Языки программирования



Глава 1

Основные концепции языков программирования

Любую систему обозначений и согласованную с ней систему понятий, которую можно использовать для описания алгоритмов и структур данных, в первом приближении можно считать языком программирования. Однако в настоящем учебнике речь пойдет только об универсальных языках программирования, которые нашли широкое применение при разработке программ в различных областях человеческой деятельности. Каждый из этих языков создавался для определенных целей и имеет свои достоинства и недостатки. Для того чтобы эффективно использовать и реализовывать языки программирования, необходимо хорошо знать фундаментальные понятия, лежащие в основе их построения.

Знание концептуальных основ языков программирования с точки зрения использования и реализации базовых языковых конструкций позволит:

- более обоснованно выбрать язык программирования для реализации конкретного проекта;
- разрабатывать более эффективные алгоритмы;
- систематически пополнять набор полезных языковых конструкций;
- ускорить изучение новых языков программирования;
- использовать полученные знания как методологическую основу для разработки новых языков программирования;
- получить базовые знания, необходимые для разработки трансляторов для языков программирования, поддерживающих разные вычислительные модели.

Используемые программистами языки программирования отличаются как своим синтаксисом, так и функциональными возможностями. Различия в синтаксисе играют незначительную роль при изучении концептуальных основ языка программирования, в то время как наличие или отсутствие тех или иных функциональных возможностей существенно влияет на реализацию и область применения языка.

1.1. Парадигмы языков программирования

На сегодняшний день имеются четыре основные парадигмы языков программирования [36, 38], отражающие вычислительные модели, с помощью которых описывается большинство существующих методов программирования:

- императивная;
- функциональная;
- декларативная;
- объектно-ориентированная.

1.1.1. Императивные языки

Императивные (*процедурные*) *языки* — это языки программирования, управляемые командами, или операторами языка. Основной концепцией императивного языка является состояние компьютера — множество всех значений всех ячеек (слов) памяти компьютера. Данная модель вытекает из особенностей аппаратной части компьютера стандартной архитектуры, названной "фон-неймановской" в честь одного из ее авторов — американского математика Джона фон Неймана. В таком компьютере данные, подлежащие обработке, и программы хранятся в одной памяти, называемой *оперативной*. Центральный процессор получает из оперативной памяти очередную команду на входном языке, декодирует ее, выбирает из памяти указанные в качестве операндов входные данные, выполняет команду и возвращает в память результат.

Программа на императивном языке представляет собой последовательность команд (операторов), которые выполняются в порядке их написания. Выполнение каждой команды приводит к изменению состояния компьютера. Основными элементами императивных языков программирования, ориентированных на фон-неймановскую архитектуру, являются переменные, моделирующие ячейки памяти компьютера, и операторы присваивания, осуществляющие пересылку данных. Выполнение оператора присваивания может быть представлено как последовательность обращений к ячейкам памяти за операндами выражения из правой части оператора присваивания, передача их процессору, вычисление выражения и возвращение результата вычисления в ячейку памяти, представляющую собой переменную из левой части оператора присваивания.

Императивные языки поддерживают, как правило, один или несколько итеративных циклов, различающихся синтаксисом. Итеративные циклы в фон-неймановской архитектуре выполняются быстро за счет того, что команды программы хранятся в соседних ячейках памяти компьютера.

Большинство императивных языков включает в себя конструкции, позволяющие программировать рекурсивные алгоритмы, но их реализация на

компьютерах с фон-неймановской архитектурой не эффективна, что связано с необходимостью программного моделирования стековой памяти.

К императивным языкам относятся такие распространенные языки программирования, как ALGOL-60 [1], BASIC [15, 51], FORTRAN [13, 14], PL/1 [56], Ada [16], Pascal [19, 21], C [57], C++ [53], Java [33, 60].

1.1.2. Языки функционального программирования

В *языках функционального программирования* (*аппликативных* языках) вычисления в основном производятся путем применения функций к заданному набору данных. Разработка программ заключается в создании из простых функций более сложных, которые последовательно применяются к начальным данным до тех пор, пока не получится конечный результат. Типичная программа, написанная на функциональном языке, имеет следующий вид:

$$функция_n\ (\ ...\ функция_2\ (функция_1\ (данные))\ ...\ ).$$

На практике наибольшее распространение получили язык функционального программирования LISP [50, 62] и два его диалекта: язык Common LISP [64] и язык Scheme. Основной структурой данных языка LISP являются связные списки, элементами которых могут быть либо атомы (идентификаторы или числовые константы), либо другие связные списки. При этом список

$$(K\ L\ M\ N)$$

в терминах данных интерпретируется как список из четырех элементов *K*, *L*, *M*, *N*, а в терминах программ — как функция *K* с аргументами *L*, *M* и *N*.

Несмотря на то, что многие ученые в области компьютерных наук указывают на преимущества языков функционального программирования по сравнению с императивными, языки функционального программирования не получили широкого распространения из-за невысокой эффективности реализации их на компьютерах с фон-неймановской архитектурой. На компьютерах с параллельной архитектурой трансляторы для функциональных языков реализуются более эффективно, однако они еще не конкурентоспособны по сравнению с реализациями императивных языков.

Кроме языка LISP, основной областью применения которого являются системы искусственного интеллекта, известны и другие языки функционального программирования: ML (MetaLanguage) [58, 63], Miranda и Haskell [37, 39]. Язык ML наряду с функциональным программированием поддерживает императивное программирование, но, в отличие от императивных языков, функции в языке ML могут быть полиморфными и передаваться между подпрограммами как параметры.

Замечание

Программирование как на императивных, так и на функциональных языках является *процедурным*. Это означает, что программы на этих языках содержат указания, *как* нужно выполнять вычисления.

1.1.3. Декларативные языки

Декларативные языки программирования — это языки программирования, в которых операторы представляют собой объявления или высказывания в символьной логике. Типичным примером таких языков являются *языки логического программирования* (языки, основанные на системе правил).

В программах на языках логического программирования соответствующие действия выполняются только при наличии необходимого разрешающего условия. Программа на языке логического программирования схематично выглядит следующим образом:

разрешающее условие 1 → *последовательность операторов 1*

разрешающее условие 2 → *последовательность операторов 2*

...

разрешающее условие n → *последовательность операторов n*

В отличие от императивных языков, операторы программы на языке логического программирования выполняются не в том порядке, как они записаны в программе. Порядок выполнения операторов определяется системой реализации правил.

Характерной особенностью декларативных языков является их *декларативная семантика*. Основная концепция декларативной семантики заключается в том, что смысл каждого оператора не зависит от того, как этот оператор используется в программе. Так, смысл заданного высказывания в языке логического программирования можно точно определить по самому оператору. Декларативная семантика намного проще семантики императивных языков, что может рассматриваться как преимущество декларативных языков над императивными.

Наиболее распространенным языком логического программирования является язык Prolog [40, 59]. Основными областями применения языка Prolog являются экспертные системы, системы обработки текстов на естественных языках и системы управления реляционными базами данных. К языкам программирования, основанным на системе правил, можно отнести языки синтаксического разбора (например, YACC — Yet Another Compiler Compiler), в которых синтаксис анализируемой программы рассматривается в качестве разрешающего условия.

1.1.4. Объектно-ориентированные языки

Концепция объектно-ориентированного программирования складывается из трех ключевых понятий: *абстракция данных*, *наследование* и *полиморфизм*.

Абстракция данных позволяет *инкапсулировать* множество объектов данных (члены класса) и набор абстрактных операций над этими объектами данных

(методы класса), ограничивая доступ к данным только через определенные абстрактные операции. Инкапсуляция позволяет изменять реализацию класса без плохо контролируемых последствий для программы в целом.

Наследование — это свойство классов создавать из базовых классов производные, которые наследуют свойства базовых классов и могут содержать новые элементы данных и методы. Наследование позволяет создавать иерархии классов и является эффективным средством внесения изменений и дополнений в программы.

Полиморфизм означает возможность одной операции или имени функции ссылаться на любое количество определений функций, зависящих от типа данных параметров и результатов. Это свойство объектно-ориентированных языков программирования обеспечивается динамическим связыванием сообщений (вызовов методов) с определениями методов.

В основе объектно-ориентированного программирования лежит объектно-ориентированная декомпозиция. Разработка объектно-ориентированных программ заключается в построении иерархии классов, описывающих отношения между объектами, и в определении классов. Вычисления в объектно-ориентированной программе задаются сообщениями, передаваемыми от одного объекта к другому.

Объектно-ориентированная парадигма программирования является попыткой объединить лучшие свойства других вычислительных моделей. Наиболее полно объектно-ориентированная концепция реализована в языке Smalltalk [61]. Поддержка объектно-ориентированной парадигмы в настоящее время включена в такие популярные императивные языки программирования, как Ada 95, Java и C++.

1.2. Критерии оценки языков программирования

Каждый из языков программирования, используемых в настоящее время (FORTRAN, Ada, C, Pascal, Java, ML, LISP, Perl, Postscript, Prolog, C++, Smalltalk, Forth, APL, BASIC, HTML, XML), имеет свои преимущества и недостатки, но, тем не менее, все они относительно удачны по сравнению с сотнями других языков, которые были разработаны и реализованы, использовались какое-то время, но так и не нашли широкого применения.

Некоторые успехи или неудачи языка могут быть внешними по отношению к нему. Так, например, использование языка Ada в США для разработки приложений в проектах Министерства обороны было регламентировано Правительством. Аналогично часть успеха языка FORTRAN можно отнести к большой поддержке его различными производителями вычислительной техники, которые потратили много усилий на его реализацию и подробное

описание. Широкое распространение таких языков программирования, как LISP и Pascal, объясняется их использованием в качестве объектов теоретического изучения студентами, специализирующимися в области языков программирования и методов их реализации.

Рассмотрим свойства, которыми должны в той или иной мере обладать языки программирования.

1.2.1. Понятность

Понятность (удобочитаемость) конструкций языка — это свойство, обеспечивающее легкость восприятия программ человеком. Это свойство языка программирования зависит от целого ряда факторов, начиная с выбора ключевых слов и заканчивая возможностью построения модульных программ.

Понятность конструкций языка зависит от выбора такой нотации языка, которая позволяла бы при чтении текста программы легко выделять основные понятия каждой конкретной части программы, не обращаясь к другой документации на программу.

Высокая степень понятности конструкций языка полезна с различных точек зрения:

- уменьшаются требования к документированию проекта, если текст программы является центральным элементом документации;
- понятность конструкций языка позволяет легче понимать программу и, следовательно, быстрее находить ошибки;
- высокая степень понятности конструкций языка позволяет легче сопровождать программу.

Это особенно справедливо для программ с большим жизненным циклом, когда поддержание обновляемой сопроводительной документации в условиях неизбежного множества последовательных модификаций может оказаться весьма трудоемким делом.

Очевидно, что реализация требований понятности конструкций языка во многом зависит от программиста, который должен по возможности лучше структурировать свою программу и так располагать ее текст, чтобы подчеркнуть структуру программы. Вместе с тем важную роль играет синтаксис и структура языка, используемого программистом. Слишком лаконичный синтаксис может оказаться удобным при написании программ, но вместе с тем усложнить их модификацию. Так программы на APL [39] настолько непонятны, что даже авторы спустя несколько месяцев после завершения работы над программой затрудняются в их интерпретации. Понятный язык программирования характеризуется тем, что конструкции, обозначающие разные понятия, выглядят по-разному, т. е. семантические различия языка отражаются в его синтаксисе.

На нижних уровнях программных конструкций язык должен обеспечивать возможность четкой спецификации того, какие объекты данных подвергаются обработке и как они используются. Эта цель достигается выбором идентификаторов и спецификацией типов данных. В язык нельзя вводить такие ограничения, как максимальная длина идентификаторов или только определенные фиксированные типы данных.

Алгоритмические структуры должны выражаться в терминах легко понимаемых структур управления, таких как `if ... then ... else` и т. п. Ключевые слова не следует сводить к аббревиатурам, символы операций должны отображать их смысл (следствием этого является перегрузка операций в языках программирования).

На более высоких уровнях программных конструкций язык должен обеспечивать возможность реализации различных абстракций (определять типы данных, структуры данных и операции над ними), а также разбиения программы на модули и управления областями действия имен.

Как правило, неизбежной платой за выполнение требования высокой степени понятности конструкций языка программирования является увеличение длины программы.

1.2.2. Надежность

Под надежностью понимается степень автоматического обнаружения ошибок, которое может быть выполнено транслятором или операционной средой, в которой выполняется программа. Надежный язык позволяет выявлять большинство ошибок во время трансляции программы, а не во время ее выполнения. Это желательно по двум причинам:

- чем раньше при разработке программы обнаружена ошибка, тем меньше стоимость самого проекта;
- трансляция может быть выполнена на любой машине, воспринимающей входной язык, в то время как тестирование оттранслированной программы должно выполняться на целевой машине либо с использованием программ интерпретации, специально разработанных для тестирования.

Существует несколько способов проверки правильности выполнения программой своих функций: использование формальных методов верификации программ, проверка путем чтения текста программы, прогон программы с тестовыми наборами данных. На практике для проверки правильности программ, как правило, используют некоторую комбинацию этих способов. При этом для обнаружения ошибок во время прогона программы необходимо включать в нее дополнительные операторы вывода промежуточных результатов, которые после отладки программы должны быть удалены.

Принципиальным средством достижения высокой надежности языка, поддерживаемым на этапе трансляции, является *система типизации данных*.

Предположим, что массив целых чисел `A`, содержащий 100 элементов, индексируется целой переменной `i`. Надежный язык должен обеспечить выполнение условия `1 ≤ i ≤ 100` в любом месте программы, где встречается `A[i]`. Это можно сделать двумя способами:

- включить в программу явную проверку значений индекса перед каждым обращением к элементу массива;
- специфицировать область значений при описании переменной `i`. В этом случае проверка значения переменной `i` будет выполняться во время присваивания ей значения.

Второй способ является более предпочтительным, т. к. во многих случаях законность изменений значения переменной `i` может быть проверена во время компиляции программы. Одновременно увеличивается удобочитаемость программы, т. к. область значений, принимаемых каждой переменной, устанавливается явно.

Безусловно, имеется предел числа ошибок, которые могут быть обнаружены любым транслятором. Например, логические ошибки в программе не могут быть обнаружены автоматически. Однако ошибок такого рода будет возникать меньше, если сам язык программирования поощряет программиста писать ясные, хорошо структурированные программы, предоставляя ему возможность выбора подходящих языковых конструкций. Отсюда следует, что надежность языка программирования связана с его удобочитаемостью.

1.2.3. Гибкость

Гибкость языка программирования проявляется в том, сколько возможностей он предоставляет программисту для выражения всех операций, которые требуются в программе, не заставляя его прибегать к вставкам ассемблерного кода или различным ухищрениям.

Как правило, при разработке языка обеспечивается достаточная гибкость для соответствующей выбранной области применения, но не больше.

Замечание

Требование гибкости языка конфликтует с требованием надежности, поэтому при выборе языка программирования для решения конкретной задачи необходимо понимать, на основании каких требований сделан выбор и на какие компромиссы при этом придется идти. Например, критерий гибкости особенно существенен при решении задач в режиме реального времени, где может потребоваться работа с широким спектром нестандартного периферийного оборудования. В тех же случаях, когда сбой в программе может привести к тяжелым последствиям, например при управлении работой атомной электростанции, на первый план выступает такая характеристика языка, как надежность.

1.2.4. Простота

Простота языка обеспечивает легкость понимания семантики языковых конструкций и запоминания их синтаксиса. Простой язык предоставляет ясный, простой и единообразный набор понятий, которые могут быть использованы в качестве базовых элементов при разработке алгоритма. При этом желательно иметь минимальное количество различных понятий с как можно более простыми и систематизированными правилами их комбинирования — язык должен обладать свойством *концептуальной целостности*. Концептуальная целостность языка включает в себя три взаимосвязанных аспекта: экономию, ортогональность и единообразие понятий.

Экономия понятий языка предполагает использование минимального числа понятий.

Ортогональность понятий означает, что между ними не должно быть взаимного влияния. В ортогональном языке любые языковые конструкции можно комбинировать по определенным правилам. Например, выражение и условный оператор некоторого языка программирования ортогональны, если любое выражение можно использовать внутри условного оператора.

Единообразие понятий требует согласованного, единого подхода к описанию и использованию всех понятий.

Для достижения простоты не обязательно избегать сложных языковых конструкций, но следует стараться не накладывать случайных ограничений на их использование. Так, при реализации массивов следует дать возможность программисту объявлять массивы любого типа данных, допускаемого языком. Если же, наоборот, в языке запретить использование массивов некоторого типа, то в результате язык окажется скорее сложным, чем простым, т. к. основные правила языка изучить и запомнить проще, чем связанные с ними ограничения.

Простота уменьшает затраты на обучение программистов и вероятность совершения ошибок, возникающих в результате неправильной интерпретации программистом языковых конструкций. Естественно, упрощать язык можно до определенного предела. Язык высокого уровня с неадекватными управляющими операторами и структурами данных вряд ли можно назвать хорошим.

Наиболее простыми являются языки функционального программирования, т. к. они основаны на использовании одной конструкции — вызова функции, который может легко комбинироваться с другими вызовами функций.

1.2.5. Естественность

Язык должен содержать такие структуры данных, управляющие структуры и операции, а также иметь такой синтаксис, которые позволяли бы отражать в программе логические структуры, лежащие в основе реализуемого алгоритма.

Наличие различных парадигм программирования напрямую связано с необходимостью реализации последовательных, параллельных и логических алгоритмов. Язык, соответствующий определенному классу алгоритмов, может существенно упростить создание программ для конкретной предметной области.

1.2.6. Мобильность

Язык, независимый от аппаратуры, предоставляет возможность переносить программы с одной платформы на другую с относительной легкостью. Это позволяет распределить высокую стоимость программного обеспечения на ряд платформ.

На практике добиться мобильности довольно трудно, особенно в системах реального времени, в которых одна из задач проектирования языка заключается в максимальном использовании преимуществ базового машинного оборудования. Особенно трудно поддаются решению проблемы, связанные с различающимися длинами слов памяти.

На мобильность значительно влияет уровень стандартизации языка. Для языков, имеющих стандартное определение, таких как Ada, FORTRAN, C и Pascal, все реализации языка должны основываться на этом стандарте. Стандарт является единственным способом обеспечения единообразия различных реализаций языка. Международные стандарты разрабатываются Организацией Международных Стандартов (ISO — International Standards Organization). Стандарты обычно создаются для популярных, широко используемых языков программирования.

1.2.7. Стоимость

Суммарная стоимость использования языка программирования складывается из нескольких составляющих. В нее входят:

- стоимость обучения языку;
- стоимость создания программы;
- стоимость трансляции программы;
- стоимость выполнения программы;
- стоимость сопровождения программы.

Стоимость обучения языку определяется степенью сложности языка.

Стоимость создания программы зависит от языка и системы программирования, выбранных для реализации конкретного приложения. Наличие в языке программирования развитых структур данных и конструкций позволяет эффективно использовать язык только при условии его надежной, эффективной и хорошо документированной реализации. Для сокращения времени создания программы необходимо иметь систему программирования, которая включает в себя специализированные текстовые редакторы, тестирующие

пакеты, средства для поддержки и модификации нескольких версий программы, развитый графический интерфейс и др.

Стоимость трансляции программы тесно связана со стоимостью ее выполнения. Совокупность методов, используемых транслятором для уменьшения объема и/или сокращения времени выполнения оттранслированной программы, называется *оптимизацией*. Чем выше степень оптимизации, тем качественнее получается результирующая программа и сильнее уменьшается время ее выполнения, но при этом возрастает стоимость трансляции. Разрешение конфликта между стоимостью трансляции и стоимостью выполнения программы осуществляется на этапе создания транслятора в результате анализа области его применения. Так для трансляторов, разрабатываемых для учебных целей, оптимизация не требуется, в то время как для промышленных трансляторов, с помощью которых создаются многократно используемые программы, требуется высокая степень оптимизации.

Язык программирования, стоимость реализации которого велика, потенциально имеет меньше шансов на широкое распространение. Одной из причин повсеместного распространения языка Java является бесплатное распространение его реализаций сразу после разработки первой версии языка.

Стоимость выполнения программы существенна для программного обеспечения систем реального времени. Системы реального времени должны обеспечивать высокую пропускную способность, чтобы не нарушать ограничений, накладываемых управляемым производственным или технологическим процессом или внешним оборудованием. Поскольку необходимо гарантировать определенное время реакции системы, следует избегать языковых конструкций, ведущих к непредсказуемым издержкам времени выполнения программы (например, при сборке мусора в схеме динамического распределения памяти). Для обычных приложений все снижающаяся стоимость машинного оборудования и все возрастающая стоимость разработки программ позволяют считать, что скорость выполнения программ на сегодняшний день не столь критична.

В *стоимость сопровождения программы* входят затраты на исправление дефектов и модификацию программы в связи с обновлением аппаратуры или расширением функциональных возможностей. Стоимость сопровождения программы в первую очередь зависит от удобочитаемости языка, поскольку сопровождение обычно выполняется лицами, не являющимися разработчиками программного обеспечения.

1.3. Объекты данных в языках программирования

Любая программа для ЭВМ, базирующейся на архитектуре фон Неймана, представляет собой набор операций, которые применяются к определенным данным в заданной последовательности. При программировании на машин-

ном языке необходимо точно знать, как данные представляются в виде последовательности битов в памяти машины и какие машинные команды необходимо использовать для реализации требуемых операций. Язык программирования предоставляет программисту абстрактную модель, в которой объекты данных и операции специфицированы в проблемно-ориентированных терминах. Под объектом данных будем понимать один или несколько однотипных элементов данных, объединенных в одно целое. Объект данных называется *элементарным*, если представляющее его значение является единым целым. В противном случае, если объект данных представляет собой совокупность некоторых других объектов, будем называть его *структурным*. Рассмотрим более подробно вопросы, связанные с определением и использованием объектов данных в языках программирования.

1.3.1. Имена

Имя (*идентификатор*) — это строка символов, используемая для обозначения некоторой сущности в программе. Такими сущностями могут быть переменные, типы, метки, подпрограммы, формальные параметры и другие конструкции языков программирования. В общем случае идентификаторы не имеют какого-либо смысла, а используются только в качестве имен программных объектов или их атрибутов.

В большинстве языков программирования идентификатор представляет собой конечную последовательность букв и цифр, которая начинается с буквы и может содержать соединительный символ подчеркивания '_'. В некоторых языках (C, C++, Java) различаются строчные и прописные буквы. При таком подходе увеличивается пространство возможных имен, но ухудшается надежность языка.

Ключевые слова — это имена, имеющие особое значение только в определенном контексте, например: **`begin`**, **`end`**, **`if`**. В некоторых языках программирования (Pascal, Ada, C, C++, Java) ключевые слова являются зарезервированными, т. е. не могут использоваться в качестве имен. В других языках (FORTRAN, PL/1, APL, BASIC) разрешается переопределять ключевые слова. Возможность переопределения ключевых слов ухудшает надежность и удобочитаемость языка, усложняет процесс компиляции, поэтому при разработке нового языка лучше объявлять ключевые слова зарезервированными.

Многие языки программирования содержат *предопределенные* имена, имеющие конкретный смысл, но не являющиеся ключевыми словами. Это, как правило, имена встроенных типов данных или функций. Для того чтобы компилятор мог правильно определять атрибуты предопределенных имен, их объявления должны быть видимы компилятору. В некоторых языках программирования, например в C и C++, программисты могут использовать имена, предопределенные в библиотеках. Доступ компилятора к таким именам возможен через соответствующие заголовочные файлы.

Предопределенные имена можно переопределять. Например, в языке Pascal идентификатор `sin` считается именем функции, значение которой равно синусу ее аргумента. Программист может определить в своей программе другой идентификатор `sin`, но в этом случае предопределенная в языке функция `sin` будет не доступна. Так же, как и в случае переопределения ключевых слов, переопределение предопределенных имен в большинстве случаев нежелательно.

1.3.2. Константы

Константа — это объект данных, имя которого связано со значением (значениями) в течение всего времени жизни. В языках программирования используются константы двух видов:

- литералы;
- именованные константы.

Литерал представляет собой буквальную запись значения константы. Например, 25 — это десятичная форма записи целочисленной константы, представляющей собой объект данных со значением 25. Форма записи значений литералов предопределенного типа задается в языке.

Именованная константа (константа, определяемая программистом) — это объект данных, который связывает имя с буквальным значением константы. Значения именованных констант известны во время компиляции, поэтому компилятор будет обнаруживать все ошибки, связанные с попыткой присвоения именованной константе нового значения.

1.3.3. Переменные

Переменная — это объект данных, который явным образом определен и именован в программе. *Простая переменная* — это именованный элементарный объект данных. Переменные можно характеризовать с помощью следующих атрибутов:

- имя;
- адрес;
- значение;
- тип;
- время жизни;
- область видимости.

Имя переменной — это идентификатор, используемый в программах для ссылки на значение переменной. Связывание объекта данных с одним или несколькими именами, с помощью которых можно ссылаться на объект

данных, осуществляется при помощи объявлений и может изменяться при входе и выходе из подпрограмм (блоков).

Переменная представляет собой абстракцию области памяти — ячейки или совокупности ячеек памяти компьютера. *Адрес* переменной — это адрес области памяти, с которой связана данная переменная. Для связывания с переменой свободная область памяти соответствующего размера извлекается из пула доступной памяти. Этот процесс называется *выделением* памяти. Разрыв связи между некоторой областью памяти и переменной называется *освобождением* памяти. При освобождении памяти ее область, с которой разрывается связь, возвращается обратно в пул доступной памяти. Выделение и освобождение памяти выполняется специальными *программами управления памятью*, которые недоступны программисту.

Во многих языках программирования одно и то же имя можно связать с разными адресами памяти. Например, в программе могут быть определены две подпрограммы `sub1` и `sub2`, в каждой из которых определяется переменная с одним и тем же именем, например `temp`. В этом случае имя `temp` определяет две *разные* переменные, которые будут связаны с разными областями памяти. Аналогичным образом решается проблема определения переменных в блочных структурах, когда переменная с одним и тем же именем описана во внешнем и внутреннем блоках.

Возможен случай, когда несколько имен переменных связаны с одной и той же областью памяти. Такие переменные называются *альтернативными* (alias-именами).

Например, в языке FORTRAN альтернативные имена могут создаваться с помощью оператора `EQUIVALENCE`, а в языках Ada и Pascal — с помощью вариантных записей. Альтернативные имена широко используются в языках манипулирования данными в системах управления базами данных для получений виртуальных копий одной и той же таблицы. Альтернативные имена экономят память, позволяя связывать с одной областью памяти несколько переменных. Однако такой подход ухудшает удобочитаемость и надежность программ, создает серьезные трудности при верификации и модернизации программ, содержащих альтернативные имена.

Значение переменной — это содержимое ячейки или совокупности ячеек памяти (определенная комбинация битов), связанных с данной переменной. Связывание переменной со своим значением осуществляется в процессе выполнения программы, обычно в результате выполнения оператора присваивания. Переменная сохраняет присвоенное ей значение до тех пор, пока этой переменной не будет присвоено новое значение, при этом предыдущее значение переменной теряется безвозвратно. С каждой переменной связывается определенный тип значений, которые она может принимать.

Тип переменной связывает переменную с множеством значений, которые она может принимать. Переменная должна быть связана с определенным типом до того, как к ней можно будет обращаться.

Время жизни переменной — это время, в течение которого переменная связана с определенной областью памяти (*более подробную информацию см. в разд. 1.5*).

Область видимости переменной — это последовательность операторов программы, из которых можно обратиться к этой переменной (*более подробную информацию см. в разд. 1.6*).

1.4. Механизмы типизации

Типы могут определяться статически и динамически. При *статическом* определении типа связывание осуществляется при трансляции программы, а при *динамическом* — во время выполнения программы.

1.4.1. Статические и динамические типы данных

Статическое определение типа может быть выполнено путем явного или неявного объявления переменных. *Явное объявление* имеет место, если в программу включен специальный оператор объявления типа, устанавливающий тип для переменных, перечисленных в операторе. *Неявное объявление* связывает переменные с типами посредством принятых по умолчанию соглашений. При этом первое появление имени в программе расценивается как объявление переменной.

Большинство языков программирования требует явного объявления переменных. В случае неявного объявления в языке должны быть заданы правила, с помощью которых определяется тип переменных. Например, в программах на языке FORTRAN, если идентификатор не был объявлен явно, по умолчанию считается, что он имеет тип `INTEGER`, если он начинается с одной из букв I, J, K, L, M или N; в противном случае идентификатор является именем переменной типа `REAL`. В языке Perl имена, начинающиеся с символа '@', являются именами массивов, а идентификаторы, начинающиеся с символа '$', могут именовать только переменные числового и строкового типов. Неявные объявления ухудшают надежность программ, т. к. необъявленные по ошибке переменные будут неправильно связаны с типами, устанавливаемыми по умолчанию, что может привести к непредсказуемым ошибкам, которые сложно обнаружить.

При динамическом связывании переменная связывается с типом в момент присваивания переменной значения. При этом тип переменной будет идентичен типу присваиваемого значения. Основным преимуществом динамического связывания переменных с типом является то, что в этом случае обеспечивается высокая гибкость языка. Например, программист может написать программу сортировки элементов массива, который может содержать данные любого типа. Переменные, предназначенные для хранения элементов массива, будут связываться с соответствующим типом во время

ввода данных. Классическим примером динамического связывания переменных с типом является язык APL, в котором корректной является следующая последовательность операторов:

```
A ← 1, 100, 1000
A ← 2.5
```

Независимо от предыдущего значения типа переменная `A` после выполнения первого оператора присваивания будет представлять собой целочисленный массив, содержащий 3 элемента: 1, 100 и 1000. В результате выполнения второго оператора присваивания переменная `A` станет простой переменной, содержащей вещественное число 2,5.

В языках функционального программирования (ML, Miranda, Haskell) распространен механизм логического вывода типа. Например, в языке ML объявление функции:

```
Fun circ_length (r) = 6.28318 * r;
```

определяет функцию, аргумент и результат которой имеют вещественный тип. Вещественный тип логически выводится из типа константы, входящей в выражение.

Динамическое связывание типов имеет ряд недостатков:

- снижается возможность обнаружения транслятором ошибок по сравнению со статическим связыванием типов, т. к. при динамическом связывании во время трансляции отсутствует информация о типе переменных;
- при реализации динамического связывания вся информация о типах переменных должна сохраняться в течение всего времени работы программы, что требует значительных дополнительных ресурсов памяти, связанных с необходимостью хранить данные различных типов;
- динамическое связывание типов приводит к увеличению времени работы программы за счет программной реализации механизмов связывания.

Языки программирования с динамическим механизмом связывания типов часто реализуются в виде интерпретаторов в связи со сложностью реализации динамического изменения типов на машинном языке.

Ключевым фактором, обеспечивающим необходимый уровень надежности языка программирования, является механизм типизации, реализованный в языке.

Различают следующие механизмы типизации:

- слабая типизация;
- строгая типизация.