Краткое содержание

ОТ АВТОРОВ ПЕРЕВОДА 10

ВВЕДЕНИЕ 12

ГЛАВА 1. ВВЕДЕНИЕ 15

ГЛАВА 2. ЯЗЫК И СИНТАКСИС 19

ГЛАВА 3. РЕГУЛЯРНЫЕ ЯЗЫКИ 27

ГЛАВА 4. АНАЛИЗ КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ 33

ГЛАВА 5. АТРИБУТНЫЕ ГРАММАТИКИ И СЕМАНТИКИ 45

ГЛАВА 6. ЯЗЫК ПРОГРАММИРОВАНИЯ ОБЕРОН-0 51

ГЛАВА 7. СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР ДЛЯ ОБЕРОНА-0 55

ГЛАВА 8. УЧЕТ КОНТЕКСТА, ЗАДАННОГО ОБЪЯВЛЕНИЯМИ 65

ГЛАВА 9. RISC-АРХИТЕКТУРА КАК ЦЕЛЬ 75

ГЛАВА 10. ВЫРАЖЕНИЯ И ПРИСВАИВАНИЯ 81

ГЛАВА 11. УСЛОВНЫЕ И ЦИКЛИЧЕСКИЕ ОПЕРАТОРЫ И ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ 95

ГЛАВА 12. ПРОЦЕДУРЫ И КОНЦЕПЦИЯ ЛОКАЛИЗАЦИИ 109

ГЛАВА 13. ЭЛЕМЕНТАРНЫЕ ТИПЫ ДАННЫх 125

ГЛАВА 14. ОТКРЫТЫЕ МАССИВЫ, УКАЗАТЕЛЬНЫЙ И ПРОЦЕДУРНЫЙ ТИПы 131

ГЛАВА 15. МОДУЛИ И РАЗДЕЛЬНАЯ КОМПИЛЯЦИЯ 141

ГЛАВА 16. ОПТИМИЗАЦИЯ И СТРУКТУРА ПРЕ/ПОСТПРОЦЕССОРА 153

ПРИЛОЖЕНИЕ А. СИНТАКСИС 164

ПРИЛОЖЕНИЕ В. НАБОР СИМВОЛОВ ASCII 167

ПРИЛОЖЕНИЕ С. КОМПИЛЯТОР ОБЕРОН-0 168

ЛИТЕРАТУРА 191

Содержание

От авторов перевода ... 10
О книге ... 10
О переводе ... 10

Введение ... 12
Предисловие ... 12
Благодарности ... 14

Глава 1. Введение ... 15

Глава 2. Язык и синтаксис ... 19
2.1. Упражнения ... 24

Глава 3. Регулярные языки ... 27
3.1. Упражнение ... 32

Глава 4. Анализ контекстно-свободных языков ... 33
4.1. Метод рекурсивного спуска ... 34
4.2. Таблично-управляемый нисходящий синтаксический анализ ... 38
4.3. Восходящий синтаксический анализ ... 40
4.4. Упражнения ... 42

Глава 5. Атрибутные грамматики и семантики ... 45
5.1. Правила типов ... 46
5.2. Правила вычислений ... 47
5.3. Правила трансляции ... 48
5.4. Упражнение ... 49

Глава 6. Язык программирования Оберон-0 ... 51
6.1. Упражнение ... 54

Глава 7. Синтаксический анализатор для Оберона-0 55
7.1. Лексический анализатор 56
7.2. Синтаксический анализатор 57
7.3. Устранение синтаксических ошибок 59
7.4. Упражнения 64

Глава 8. Учет контекста, заданного объявлениями 65
8.1. Объявления 66
8.2. Записи о типах данных 68
8.3. Представление данных во время выполнения 69
8.4. Упражнения 73

Глава 9. RISC-архитектура как цель 75
9.1. Ресурсы и регистры 76

Глава 10. Выражения и присваивания 81
10.1. Прямая генерация кода по принципу стека 82
10.2. Отсроченная генерация кода 84
10.3. Индексированные переменные и поля записей 89
10.4. Упражнения 94

Глава 11. Условные и циклические операторы и логические выражения 95
11.1. Сравнения и переходы 96
11.2. Условные и циклические операторы 97
11.3. Логические операции 101
11.4. Присваивание логическим переменным 105
11.5. Упражнения 106

Глава 12. Процедуры и концепция локализации 109
12.1. Организация памяти во время выполнения 110
12.2. Адресация переменных 112
12.3. Параметры 114
12.4. Объявления и вызовы процедур 116

12.5. Стандартные процедуры 121
12.6. Процедуры-функции 122
12.7. Упражнения 123

Глава 13. Элементарные типы данных 125
13.1. Типы REAL и LONGREAL 126
13.2. Совместимость между числовыми типами данных 127
13.3. Тип данных SET 129
13.4. Упражнения 130

Глава 14. Открытые массивы, указательный и процедурный типы 131
14.1. Открытые массивы 132
14.2. Динамические структуры данных и указатели 133
14.3. Процедурные типы 136
14.4. Упражнения 138

Глава 15. Модули и раздельная компиляция 141
15.1. Принцип скрытия информации 142
15.2. Раздельная компиляция 143
15.3. Реализация символьных файлов 145
15.4. Адресация внешних объектов 149
15.5. Проверка конфигурационной совместимости 150
15.6. Упражнения 152

Глава 16. Оптимизация и структура пре/постпроцессора 153
16.1. Общие соображения 154
16.2. Простые оптимизации 155
16.3. Исключение повторных вычислений 156
16.4. Распределение регистров 157
16.5. Структура пре/постпроцессорного компилятора 158
16.6. Упражнения 162

Приложение А. Синтаксис 164
А1. Оберон-0 164
А2. Оберон 164
А3. Символьные файлы 166

Приложение B. Набор символов ASCII 167

Приложение C. Компилятор Оберон-0 168
C.1. Лексический анализатор 169
C.2. Синтаксический анализатор 172
C.3. Генератор кода 182

Литература 191

От авторов перевода

О книге

Давно известно, что лучший способ постичь секреты мастерства – это наблюдать за работой мастера. Эта небольшая, но насыщенная информацией книжка, по сути дела, представляет собой отчет о такой работе. Ну а то, что ее автор – настоящий мастер своего дела, сомнению не подлежит, потому что имя профессора Никлауса Вирта ни в каких дополнительных рекомендациях не нуждается. Эта книга – своего рода мастер-класс, который дает своим ученикам всемирно известный маэстро. Она не является ни «тяжелой» теоретической монографией, ни сборником наставлений и поучений увенчанного лаврами мэтра. Эта книжка – *практическое* пособие для всех тех любознательных людей, кто желает разобраться и понять, что такое компилятор и как он устроен. По мнению автора, без этого ни один программист не может называть себя квалифицированным специалистом.

В отличие от многочисленных книг, которые исчерпывающе описывают и теорию, и разнообразные методы синтаксического анализа, перевода и компиляции, эта книжка посвящена реализации одного-единственного компилятора *современного* языка программирования для *конкретного* компьютера. Но это нисколько не умаляет ее достоинства. Если обычные книги после прочтения почти всегда оставляют читателя наедине с вопросом «А что же дальше? Где же результат?» или с загадочными, полными опечаток текстами готовых программ, то эта небольшая книжка расставляет практически все точки над i, проводя читателя от *самого начала* до *самого конца* процесса разработки компилятора, попутно предупреждая его о неверных шагах и давая ему в руки богатый практический материал. Автор придерживается принципа «Делай со мной. Делай, как я. Делай лучше меня».

Таким образом, книга Н. Вирта – безусловно, не только прекрасное дополнение к многочисленным и столь же прекрасным фундаментальным трудам по этой теме, но может и должна использоваться в качестве практического пособия по изучению компиляторов. Кроме того, простота и доступность преподнесения довольно сложного материала снимает с него покров таинственности и делает его доступным практически каждому любителю программирования. Остается только сожалеть о том, что эта книга не была своевременно переведена и издана у нас.

Для практического использования текст компилятора Оберон-0, о котором идет речь в книге, адаптирован к системе БлэкБокс (BlackBox Component Builder – вариант системы Оберон). Оригинальные и адаптированные исходные тексты компилятора можно найти на сайте www.oberoncore.ru.

О переводе

Несколько слов о переводе.
В силу того, что мы имеем дело не с развернутой монографией, а с конспектом лекций, каждая фраза, часто облекаемая в форму тезиса, до предела насыщена

информацией. Поэтому наша основная задача при переводе состояла в том, чтобы сохранить лаконичность и информационную насыщенность авторского текста и при этом максимально точно довести его суть до читателя, не поддаваясь искушению сдобрить его отсебятиной.

Несмотря на царящие до сих пор «разброд и шатания» в терминологии по этой теме, мы при переводе, следуя за автором, отдавали предпочтение наиболее устоявшимся, хотя и не всегда правильным и точным, терминам.

В связи с этим нельзя не упомянуть о терминах «front-end» и «back-end». Они уже давно употребляются в разнообразной англоязычной технической литературе, но тем не менее до сих пор не находят адекватных русскоязычных эквивалентов. Чаще всего их перевод зависит от контекста. Применительно к компиляторам наиболее точными их русскими аналогами являются, пожалуй, «машинно-независимая часть» и «машинно-зависимая часть» соответственно. Однако мы, теперь уже следуя авторской лаконичности, предпочли им более абстрактные и менее точные, но более короткие термины – «препроцессор» и «постпроцессор» соответственно.

Кроме того, список литературы пронумерован, и именные ссылки на него в тексте заменены номерными. К списку литературы добавлено несколько более поздних публикаций.

Авторы перевода выражают благодарность В. Н. Лукину за прочтение перевода и сделанные замечания.

Введение

Предисловие

Эта книга появилась из моих конспектов лекций по вводному курсу проектирования компиляторов в ETH (Федеральном технологическом институте) в Цюрихе. Несколько раз меня просили объяснить необходимость этого курса, так как проектирование компиляторов рассматривается как некий эзотерический предмет, применяемый только в нескольких узкоспециализированных программистских фирмах. Поскольку в наши дни все, что не приносит немедленной выгоды, должно быть оправдано, я должен попробовать объяснить, почему я вообще считаю этот предмет важным и уместным для студентов, изучающих информатику.

Основой любого академического образования является то, что передается не только знание и, в случае инженерного образования, «ноу-хау», но и понимание сути явления и способность проникнуть в его суть. В частности, в информатике мало поверхностного знания системы, необходимо еще и понимание ее содержания. Каждый образованный программист должен знать возможности компьютера, понимать способы и методы представления и интерпретации программ. Компилятор преобразует текст программы во внутренний код, это мост, соединяющий программное обеспечение и аппаратные средства.

Однако кому-то может показаться, что нет необходимости знать методы трансляции для понимания связи между исполняемой программой и кодом и еще менее важно знать, как на самом деле пишется компилятор. Личный опыт преподавателя подсказывает мне, что глубокое понимание предмета лучше всего приходит при всестороннем проникновении как в общую идею системы, так и в детали ее реализации. В нашем случае таким проникновением будет написание реального компилятора.

Конечно, мы должны сосредоточиться на основах. В конце концов, эта книга – вводный курс, а не справочник для специалистов. Наше первое ограничение касается входного языка. Было бы неуместным рассматривать проектирование компиляторов для больших языков. Язык должен быть небольшим, но тем не менее должен содержать все поистине фундаментальные элементы языков программирования. Для наших целей мы выбрали подмножество языка Оберон. Второе ограничение касается целевого компьютера. Он должен иметь обычную структуру и простой набор команд. Наиболее важна практичность обучающих понятий. Оберон – это общецелевой гибкий и мощный язык, а наш целевой компьютер идеальным образом отражает удачную RISC-архитектуру. И наконец, третье ограничение состоит в отказе от изощренных методов оптимизации кода. При таких условиях можно объяснить весь компилятор в деталях и даже создать его в ограниченные рамками курса сроки.

В главах 2 и 3 рассматриваются основы языка и синтаксиса. Глава 4 посвящена синтаксическому анализу, то есть методу разбора предложений и программ. Мы

сосредоточили внимание на простом, но удивительно мощном методе рекурсивного спуска, который используется в нашем иллюстративном компиляторе. Мы рассматриваем синтаксический анализ как средство для достижения цели, но не как самоцель. Глава 5 готовит нас к переходу от синтаксического анализатора к компилятору, а выбранный метод ставится в зависимость от атрибутов синтаксических конструкций.

После знакомства в главе 6 с языком Оберон-0 в главе 7 приводится разработка его синтаксического анализатора методом рекурсивного спуска. Из практических соображений обсуждается также обработка синтаксических ошибок. В главе 8 мы объясняем, почему языки, содержащие объявления и, следовательно, зависимость от контекста, могут тем не менее обрабатываться как контекстно-свободные.

До этого момента не было необходимости в рассмотрении целевого компьютера и набора его команд. Но поскольку последующие главы посвящены теме генерации кода, то становится неизбежной спецификация целевого компьютера (глава 9). Это RISC-архитектура с небольшим набором команд и набором регистров. В связи с этим центральная тема разработки компилятора – генерация последовательностей команд – разнесена по трем главам: код для выражений и присваиваний (глава 10), код для условных операторов и операторов цикла (глава 11), код для объявлений процедур и обращений к ним (глава 12). Вместе они покрывают все конструкции языка Оберон-0.

Последующие главы посвящены нескольким дополнительным важным конструкциям языков программирования общего назначения. Их трактовка поверхностна и не затрагивает деталей, но подкреплена несколькими упражнениями в конце соответствующих глав. Рассматриваются следующие темы: элементарные типы данных (глава 13), открытые массивы, динамические структуры данных и процедурные типы, называемые методами в объектно-ориентированной терминологии (глава 14).

Глава 15 касается модульного конструирования и принципов скрытия информации. Это приводит к теме разработки программного обеспечения в команде, основанной на определении интерфейсов и последующей независимой реализации частей (модулей). Методика основана на раздельной компиляции модулей с полным контролем совместимости типов всех компонентов интерфейса. Такая методика имеет первостепенное значение для разработки программного обеспечения в целом и для современных языков программирования в частности.

Наконец, глава 16 дает краткий обзор проблем оптимизации кода. Она необходима, с одной стороны, из-за семантической пропасти между исходными языками и архитектурами компьютеров, а с другой – из-за нашего желания как можно полнее использовать все доступные ресурсы компьютеров.

Благодарности

Я выражаю мои искренние благодарности всем, кто способствовал своими предложениями и критикой этой книги, которая созрела за многие годы преподавания курса проектирования компиляторов в ETH в Цюрихе. В частности, я обязан Ханспетеру Месенбоку и Михаэлю Францу, которые внимательно прочли рукопись и подвергли ее критическому разбору. Кроме того, я благодарю Штефана Геринга, Штефана Людвига и Джозефа Темпла за их ценные комментарии и сотрудничество в курсе обучения.

Никлаус Вирт
Декабрь 1995

Глава 1

Компьютерные программы пишутся на языке программирования и определяют классы вычислительных процессов. Однако компьютеры выполняют не тексты программ, а последовательности отдельных команд. Поэтому текст программы должен быть оттранслирован в соответствующую последовательность команд, прежде чем он может быть выполнен компьютером. Эта трансляция может быть автоматизирована, то есть она сама может быть описана программой. Транслирующая программа называется *компилятором*, а текст, который должен транслироваться, называется *исходным текстом* (или иногда *исходным кодом*).

Нетрудно видеть, что этот процесс трансляции от исходного текста до последовательности команд требует значительных усилий и должен подчиняться сложным правилам. Построение первого компилятора для языка *Фортран* (formula translator) примерно в 1956 году было смелым предприятием, в успехе которого мало кто был уверен. Создание компилятора потребовало приблизительно 18 человеко-лет и поэтому считалось одним из крупнейших программных проектов того времени.

Запутанность и сложность процесса трансляции могли быть уменьшены только при выборе ясно определенного, хорошо структурированного исходного языка. Это произошло впервые в 1960 году с появлением языка Алгол 60, который заложил технические основы проектирования компиляторов, имеющие значение и по сей день. Также впервые для определения структуры языка была применена формальная система записи [12].

Процесс трансляции теперь управляется структурой анализируемого текста. Текст разбирается на грамматические компоненты согласно заданному *синтаксису*. Для простейших компонентов их семантика распознается, а значение (семантика) составных компонентов выводится из семантики их составляющих. Смысл исходного текста конечно же должен быть сохранен при трансляции.

В сущности, процесс трансляции состоит из следующих частей:

1. Последовательность литер исходного текста транслируется в соответствующую последовательность *символов* словаря языка. Например, идентификаторы, состоящие из букв и цифр, числа, состоящие из цифр, разделители и операторы, состоящие из специальных литер, распознаются на этом этапе, который называется *лексическим анализом*.
2. Последовательность символов преобразуется в представление, которое непосредственно отражает синтаксическую структуру исходного текста и делает эту структуру легко узнаваемой. Этот этап называется *синтаксическим анализом* (грамматическим разбором).
3. Языки высокого уровня характеризуются тем, что объекты программ, например переменные и функции, классифицируются согласно их типу. Поэтому в дополнение к синтаксическим правилам в языке определяют правила совместимости типов операций и операндов. Следовательно, дополнительная обязанность компилятора – проверка соблюдения программой этих правил. Такая проверка называется *контролем типов*.
4. На основе представления, полученного на шаге 2, генерируется последовательность команд из системы команд целевого компьютера. Этот этап назы-

вается *генерацией кода*. Вообще, это наиболее запутанная часть компилятора и не в последнюю очередь потому, что системам команд многих компьютеров недостает желаемой регулярности. Зачастую из-за этого генерация кода разделяется еще на несколько фаз.

Разбиение процесса компиляции на как можно большее число частей было преобладающей технологией приблизительно до 1980 года, потому что доступная память была слишком мала, чтобы вместить весь компилятор. Только отдельные части компилятора, будучи подогнанными по размеру к памяти, могли загружаться последовательно одна за другой. Части назывались *проходами*, а все вместе называлось *многопроходным компилятором*. Число проходов было обычно от 4 до 6, но в одном из известных автору случаев (для PL/I) достигло 70. Как правило, выход прохода k служил входом для прохода k+1, а диск служил промежуточной памятью (рис. 1.1). Очень частое обращение к дисковой памяти приводило к длительной компиляции.

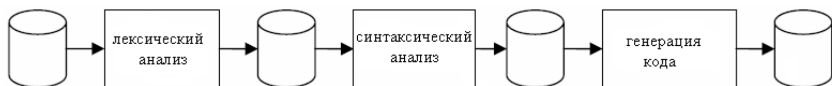

Рис. 1.1. Многопроходная компиляция

Современные компьютеры с их практически неограниченными объемами памяти дают возможность исключить промежуточное хранение данных на диске. Вместе с этим можно отказаться как от сложного процесса линеаризации структуры данных на выходе, так и от воссоздания ее на входе. Поэтому *однопроходные компиляторы* могут увеличить скорость компиляции в несколько тысяч раз. Вместо того чтобы цепляться одна за другую в строго последовательном порядке, различные части (задачи) чередуются. Например, генерация кода не ждет, пока завершатся все подготовительные задачи, а начинается сразу после распознавания первой сентенциальной структуры исходного текста.

Разумным компромиссом является компилятор, состоящий из двух частей, называемых *препроцессором* (*front end*) *и постпроцессором* (*back end*). Первая часть включает лексический и синтаксический анализы с контролем типов и генерирует дерево, представляющее синтаксическую структуру исходного текста. Это дерево хранится в основной памяти и образует интерфейс для второй части, которая выполняет генерацию кода. Основное преимущество такого решения заключается в независимости препроцессора компилятора от целевого компьютера и его системы команд. Это преимущество неоценимо, когда нужны компиляторы одного и того же языка для разных компьютеров, потому что один и тот же препроцессор служит им всем.

Идея разделения исходного языка и целевой архитектуры привела также к созданию проектов с несколькими препроцессорами для различных языков, генерирующими деревья для единственного постпроцессора. Если для трансляции m

языков для n компьютеров раньше было необходимо $m \times n$ компиляторов, то теперь достаточно m препроцессоров и n постпроцессоров (рис. 1.2).

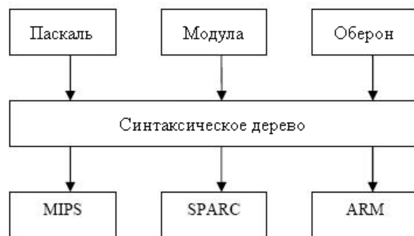

Рис. 1.2. Препроцессор и постпроцессор

Это современное решение задачи переноса компилятора напоминает нам подход, который сыграл значительную роль в распространении Паскаля примерно в 1975 году [15]. Роль структурного дерева была возложена на линеаризованную форму – последовательность команд абстрактного компьютера. Постпроцессор состоял из программы-интерпретатора, реализация которой не вызывала больших трудностей, а последовательность команд называлась P-кодом. Недостатком этого решения была потеря эффективности, свойственная интерпретаторам.

Часто встречаются компиляторы, которые генерируют не двоичный код сразу, а сначала текст ассемблера. Для окончательной трансляции вслед за компилятором запускается еще и ассемблер, в силу чего неизбежно увеличивается время трансляции. Так как эта схема едва ли сулит какие-то преимущества, мы не рекомендуем такой подход.

Более того, языки высокого уровня все чаще используются для программирования микроконтроллеров, применяемых в различных технических устройствах. Подобные устройства используются прежде всего для сбора данных и автоматического управления машинами. В таких случаях объем рабочей памяти обычно небольшой и недостаточен для того, чтобы разместить в нем компилятор. И тогда программное обеспечение генерируется на других компьютерах, способных к компиляции. Компилятор, создающий код для компьютера, отличного от того, на котором выполняется компиляция, называется *кросс-компилятором*. Сгенерированный код в этом случае передается, или загружается, по линии передачи данных.

В следующих главах мы сосредоточимся на теоретических основах проектирования компиляторов, а затем – на разработке настоящего однопроходного компилятора.

Глава 2

Язык и синтаксис

2.1. Упражнения 24

Каждый язык обладает структурой, называемой грамматикой, или синтаксисом. Например, правильное предложение (в английском языке – *Прим. перев.*) всегда состоит из подлежащего и следующего за ним сказуемого. Под правильным здесь понимается *правильно составленное* предложение. Это можно описать следующей формулой:

```
предложение = подлежащее сказуемое.
```

Если мы добавим к этой формуле еще две

```
подлежащее  = "Джон" | "Мария".
сказуемое  = "ест" | "говорит".
```

то с их помощью получим ровно четыре возможных предложения, а именно:

```
Джон ест          Мария ест
Джон говорит      Мария говорит
```

где символ | должен произноситься как *или*. Мы назовем эти формулы *синтаксическими правилами*, *продукциями* или просто *синтаксическими уравнениями*. Подлежащее и сказуемое – это синтаксические классы. Краткая запись этих формул пренебрегает смыслом идентификаторов:

```
S = AB.          L = {ac, ad, bc, bd}
A = "a" | "b".
B = "c" | "d".
```

Мы будем использовать такую сокращенную запись в последующих кратких примерах. Множество L предложений, которые могут быть сгенерированы этим способом, то есть повторяющейся заменой левых частей уравнений правыми, называется *языком*.

Приведенный выше пример, очевидно, определяет язык, состоящий только из четырех предложений. Обычно язык содержит бесконечно много предложений. Следующий пример показывает, что бесконечное множество может быть очень просто определено конечным числом уравнений. Символ ∅ обозначает пустую последовательность.

```
S = A.            L = {∅, a, aa, aaa, aaaa,...}
A = "a" A | ∅.
```

Метод, позволяющий выполнять подстановку (здесь "a"A вместо A) бесконечное число раз, называется *рекурсией*.

Наш третий пример опять основан на применении рекурсии. Но он генерирует не только предложения, состоящие из произвольной последовательности одного и того же символа, но и вложенные предложения:

```
S = A.            L = {b, abc, aabcc, aaabccc,...}
A = "a" A "c" | "b".
```

Понятно, что таким образом может быть выражена произвольно глубокая вложенность (здесь – для A), что особенно важно в определении структурированных языков.

Наш четвертый, и последний, пример показывает структуру выражений. Символы E, T, F и V обозначают выражение, слагаемое, множитель и переменную соответственно.

```
E = T | "+" T.
T = F | T "*" F.
F = V | "(" E ")".
V = "a" | "b" | "c" | "d".
```

Из этого примера видно, что синтаксис не только определяет множество предложений языка, но и наделяет их структурой. Синтаксис раскладывает предложения на составляющие, как показано в примере на рис. 2.1. Графические представления называются *структурными деревьями*, или *синтаксическими деревьями*.

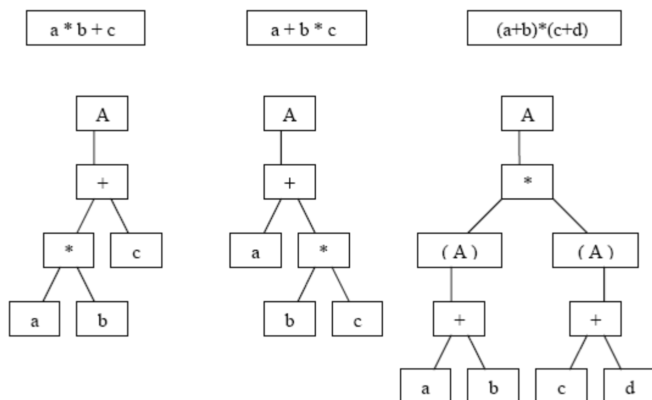

Рис. 2.1. Структура выражений

Сформулируем представленные выше понятия более строго.

Язык (порождающая язык грамматика – *Прим. перев.*) определяется следующим образом:

1. Множество *терминальных символов*. Это символы, которые появляются в предложениях. Говорят, что они терминальные, потому что не могут быть заменены никакими другими символами. Процесс подстановки заканчивается терминальными символами. В нашем первом примере это множество состоит из элементов a, b, c и d. Это множество также называется *словарем*.
2. Множество *нетерминальных символов*. Они обозначают синтаксические классы и могут замещаться в результате подстановок. В нашем первом примере это множество состоит из элементов S, A и B.
3. Множество *синтаксических уравнений* (также называемых *продукциями*). Они определяют возможные подстановки нетерминальных символов. Уравнение задается для каждого нетерминального символа.

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине
«Электронный универс»
e-Univers.ru