ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Разработка программ «калькуляторного» типа с применением приложения Bison

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по специальностям
230105-«Программное обеспечение вычислительной техники и автоматизированных систем»
010503-«Математическое обеспечение и администрирование информационных систем»

Автор

Коледов Леонид Викторович

Ростов-на-Дону, 2013

Аннотация

Данная разработка может быть использована в качестве основного учебного материала по дисциплинам: «Теория языков программирования и методы трансляции» и «Теория вычислительных процессов»

Автор

Коледов Леонид Викторович, к. ф.-м. н., доцент, профессор кафедры

Область научных интересов

Информационные технологии, системы искусственного интеллекта

Оглавление

ВВЕДЕНИЕ

Bison -- это генератор лексических анализаторов общего назначения, который преобразует описание контекстно-свободной LALR(1) грамматики в программу на языке C для разбора этой грамматики. Если вы овладеете Bison, вы сможете использовать его для разработки анализаторов языков достаточно широкого класса: от используемых в простых настольных калькуляторах до сложных языков программирования.

Bison обратно совместим с Yacc: все правильные грамматики Yacc должны без изменений работать с Bison. Любой человек, хорошо знающий Yacc, не должен иметь больших проблем при использовании Bison. Вам нужно иметь навык программирования на C для того, чтобы использовать Bison и чтобы понимать это руководство.

Мы начнём с учебных глав, которые объясняют основные принципы Bison и содержат три полностью завершённых примера с объяснениями. Если вы не знаете ни Bison, ни Yacc, начните с них. Затем следуют главы, детально описывающие специфические особенности Bison.

Bison написан, в основном, Робертом Корбеттом (Robert Corbett). Ричард Столлмен (Richard Stallman) сделал его совместимым с Yacc. Вильфред Хансен (Wilfred Hansen) из Carnegie Mellon Univerisity добавил поддержку многосимвольных литералов и другие возможности.

Эта редакция относится к Bison версии 1.35.

1. УСЛОВИЯ ИСПОЛЬЗОВАНИЯ BISON

Начиная с версии 1.24 Bison мы изменили условия распространения yyparse, разрешив использовать продукт работы Bison в несвободных программах. Ранее анализаторы, сгенерированные Bison, могли быть использованы только в программах, являющихся свободным программным обеспечением.

Другие инструменты GNU для программирования, такие как компилятор C GNU, никогда не содержали такого требования. Они всегда могли использоваться в несвободном программном обеспечении. Bison отличался от них не из-за какого-то особого политического решения, просто ко всему исходному коду Bison применялась обычная Универсальная Общественная Лицензия (GPL).

Выход Bison -- файл анализатора Bison -- содержит точную копию значительной части Bison в качестве кода функции yyparse (все действия вашей грамматики вставляются в эту функцию в одном месте, остальная часть функции при этом не изменяется). В результате применения условий GPL к коду yyparse использование выхода Bison было ограничено свободным программным обеспечением.

Мы не изменяли условия из-за нашего отношения к людям, желающим делать программы частно-собственными (проприетарными). **Программы должны быть свободными.** Но мы поняли, что ограничение использования Bison свободным программным обеспечением не слишком способствует производству с его помощью других свободных программ. И мы решили сделать практические условия использования Bison теми же, что и для других инструментов GNU.

2. ПРИНЦИПЫ BISON

В этой главе вводятся многие основные понятия, без которых детальное описание Bison не будет иметь смысла. Если вы ещё не знаете, как использовать Bison или Yacc, мы предлагаем вам начать с внимательного чтения этой главы.

2.1 Языки и контекстно-свободные грамматики

Для того, чтобы Bison мог разобрать программу на каком-то языке, этот язык должен быть описан *контекстно-свободной грамматикой*. Это означает, что вы определяете одну или более *синтаксических групп* и задаёте правила их сборки из составных частей. Например, в языке C одна из групп называется `выражение'. Правило для составления выражения может выглядеть так: "Выражение может состоять из знака `минус' и другого выражения". Другое правило: "Выражением может быть целое число". Как вы может видеть, правила часто бывают рекурсивными, но должно быть по крайней мере одно правило, выводящее из рекурсии.

Наиболее распространённой формальной системой для представления таких правил в удобном для человека виде является *форма Бэкуса-Наура* (БНФ, Backus-Naur Form, BNF), которая была разработана для описания языка Algol 60. Любая грамматика, выраженная в форме Бэкуса-Наура является контекстно-свободной грамматикой. Bison принимает на вход, в сущности, особый вид БНФ, адаптированный для машинной обработки.

Bison может работать не со всеми контекстно-свободными грамматиками, а только с грамматиками класса LALR(1). Коротко, это означает, что должно быть возможно определить, как разобрать любую часть входа, заглядывая вперёд не более, чем на одну лексему. Строго говоря, это описание LR(1)-грамматики, класс LALR(1) имеет дополнительные ограничения, которые не так просто объяснить. Но в обычной практике редко встречаются LR(1)-грамматики, которые не являются LALR(1). См. раздел 6.7 Загадочные конфликты свёртка/свёртка, для получения большей информации.

В правилах формальной грамматики языка каждый вид синтаксических единиц или групп назы- вается *символом*. Те из них, ко-

торые формируются группировкой меньших конструкций в соответствии с правилами грамматики, называются *нетерминальными символами*, а те, что не могут разбиты -- *терминальными символами* или *типами лексем*. Мы называем часть входного текста, соответствующую одному терминальному символу *лексемой*, а соответствующую нетерминальному символу -- *группой*.

Для примера терминальных и нетерминальных символов можно использовать язык C. Лексемами C являются идентификаторы, константы (числовые и строковые), и различные ключевые слова, знаки арифметических операций и пунктуации. Таким образом, терминальные символы грамматики C это: `идентификатор', `число', `строка' и по одному символу на каждое ключевое слово, знак операции или пунктуации: `if', 'return', `const', `static', `int', `char', `знак плюс', `открывающая скобка', `закрывающая скобка', `запятая' и многие другие (эти лексемы могут быть разбиты на литеры, но это уже вопрос составления словарей, а не грамматики).

Вот простая функция на языке C, разбитая на лексемы:

@ifnotinfo

```
int             /* ключевое слово `int' */
square (int x)  /* идентификатор, открывающая круглая скобка, */
                /* идентификатор, идентификатор, закрывающая  */
                /* круглая скобка */
{               /* открывающая фигурная скобка */
  return x * x; /* ключевое слово `return', идентификатор, звёздочка, */
                /* идентификатор, точка с запятой */
}               /* закрывающая фигурная скобка */
```

Синтаксические группы C это: выражение, оператор, объявление и определение функции. Они представлены в грамматике C нетерминальными символами `выражение', `оператор', `объявление' и `определение функции'. Полная грамматика, для того, чтобы выразить смысл этих четырёх, использует десятки дополнительных языковых конструкций, каждой из которых соответствует свой нетерминальный символ. Пример выше является определением функции, он содержит одно объявление и один оператор. В

операторе каждое `x', так же, как и `x * x' являются выражениями.

Каждому нетерминальному символу должны быть сопоставлены правила грамматики, показывающие, как он собирается из более простых конструкций. Например, одним из операторов C является оператор return, это может быть описано правилом грамматики, неформально читающимся так:

`Оператор' может состоять из ключевого слова `return', `выражения' и `точки с запятой'.

Должно существовать множество других правил для `оператор', по одному на каждый вид оператора C.

Один нетерминальный символ должен быть отмечен как специальный, определяющий завершённое высказывание на языке. Он называется *начальным символом*. В компиляторе это означает полную программу на входе. В языке C эту роль играет нетерминальный символ `последовательность определений и объявлений'.

Например, `1 + 2' является правильным выражением C -- правильной частью программы на C -- но не является правильной *целой* программой на C. В контекстно-свободной грамматике C это следует из того, что `выражение' не является начальным символом.

Анализатор Bison читает на входе последовательность лексем и группирует их, используя правила грамматики. Если вход правилен, конечным результатом будет свёртка всей последовательности лексем в одну группу, которой соответствует начальный символ грамматики. Если мы используем грамматику C, весь входной текст в целом должен быть `последовательностью определений и объявлений'. Если это не так, анализатор сообщит о синтаксической ошибке.

2.2 От формальных правил к входному тексту Bison

Формальная грамматика -- это математическая конструкция. Чтобы определить язык для Bison, вы должны написать файл, описывающий грамматику в синтаксисе Bison -- файл *грамматики Bison*. См. раздел 4. Файлы грамматики Bison.

Нетерминальный символ формальной грамматики на входе Bison представляется идентификатором, таким же как идентификатор C. По соглашению их нужно записывать в нижнем регистре, например: expr, stmt или declaration.

Представление в Bison нетерминальных символов также называется *типом лексем*. Типы лексем также могут быть представлены идентификаторами в стиле C. По соглашению эти идентификаторы следует записывать в верхнем регистре, чтобы отличить их от нетерминалов, например, INTEGER, IDENTIFIER, IF, или RETURN. Терминальный символ, соответствующий конкретному ключевому слову языка следует называть так же, как это ключевое слово выглядит в верхнем регистре. Терминальный символ error зарезервирован для восстановления после ошибок. См. раздел 4.2 Символы, терминальные и нетерминальные.

Терминальный символ также может быть представлен как однолитерная константа, как однолитерная константа C. Вам стоит делать так всегда, когда лексема представляет собой просто одиночную литеру (скобку, знак плюс и т.д.) -- используйте ту же литеру в качестве терминального символа для этой лексемы.

Третий способ представления терминального символа -- представление строковой константой C из нескольких литер. См. раздел 4.2 Символы, терминальные и нетерминальные, для получения большей информации.

Правила грамматики также содержат выражение в синтаксисе Bison. Например, вот правило Bison для оператора C return. Точка с запятой в кавычках является однолитерной лексемой, представляющей часть синтаксиса оператора C, а отдельная точка с запятой и двоеточие являются знаками пунктуации Bison, используемыми во всех правилах.

```
stmt:   RETURN expr ';'
        ;
```

См. раздел 4.3 Синтаксис правил грамматики.

2.3 Семантические значения

Формальная грамматика выбирает лексемы только по их виду, например, если в правиле упоминается терминальный символ `целочисленная константа', это означает, что в этой позиции грамматически допустима *любая* целочисленная константа. Точное значение константы не имеет значения для разбора -- если `x+4' грамматически допустимо, то `x+1' или `x+3989' равно допустимы.

Но точное значение очень важно, чтобы после разбора определить, что означает входной текст. Компилятор, не могущий различить в программе константы 4, 1 и 3989, бесполезен! Поэтому каждая лексема в грамматике Bison характеризуется как типом лексемы, так и *семантическим значением*. См. раздел 4.5 Определение семантики языка.

Тип лексемы -- это терминальный символ, определённый в грамматике, такой как INTEGER, IDENTIFIER или ','. Он даёт всю информацию, необходимую для принятия решения, где допустимо появления лексемы и как группировать её с другими лексемами. Правила грамматик не знают о лексемах ничего, кроме их типов.

Семантическое значение несёт всю остальную информацию о смысле лексемы, такую как значение целого или имя идентификатора (такие лексемы как ',', просто знаки пунктуации, не нуждаются в каком-либо семантическом значении).

Например, входная лексема может классифицироваться как лексема типа INTEGER и иметь семантическое значение 4. Другая входная лексема может иметь тот же тип INTEGER, но значение 3989. Если правило грамматики говорит, что допустима лексема типа INTEGER, будет принята любая из этих двух лексем, потому что обе они имеют тип INTEGER. Когда анализатор принимает лексему, он отслеживает её семантическое значение.

Каждая группа, так же как и её нетерминальный символ, может иметь семантическое значение. Например, в калькуляторе выражение обычно имеет семантическое значение, представляющее собой число. В компиляторе языка программирования выражение обычно имеет семантическое значение в виде дерева, описывающего смысл выражения.

2.4 Семантические действия

Чтобы быть полезной, программа должна делать нечто большее, чем разбор входного текста -- она должны также создавать некий выход, основанный на входе. В грамматике Bison правило грамматики может содержать *действие*, состоящее из операторов C. Каждый раз, когда анализатор распознаёт текст, соответствующий правилу, выполняется его действие. См. раздел 4.5.3 Действия.

Чаще всего целью действия является вычисление семантического значения всей конструкции по семантическим значениям её частей. Предположим, например, что у нас есть правило, гласящее, что выражение может быть суммой двух выражений. Когда анализатор распознаёт такую сумму, каждое из подвыражений имеет семантическое значение, описывающее, как оно построено. Действию этого правила следует создать значение подобного вида для только что распознанного большего выражения.

Например, вот правило, говорящее, что выражение может быть суммой двух подвыражений:

```
expr: expr '+' expr   { $$ = $1 + $3; }
    ;
```

Действие сообщает, как получить семантическое значение выражения суммы из значений двух подвыражений.

2.5 Положения

Многие приложения, такие как интерпретаторы или компиляторы, должны генерировать подробные и информативные сообщения об ошибках. Для обеспечения этого должно быть возможно отслеживать *позицию в тексте* или *положение* каждой синтаксиче-

ской конструкции. Bison предоставляет механизм работы с такими положениями.

Каждая лексема имеет семантическое значение. Аналогично, каждой лексеме сопоставлено положение, но тип положений одинаков для всех лексем и групп. Более того, создаваемый анализатор снабжён структурой данных для информации о положениях, задаваемой по умолчанию (см. раздел 4.6 Отслеживание положений, для получения дальнейшей информации).

Как и с семантическими значениями, в действиях можно получить доступ к положениям, используя специальный набор конструкций. В приведённом выше примере положение группы в целом -- @$, в то время как положения подвыражений -- @1 и @3.

Когда обнаруживается текст, соответствующий правилу, для вычисления семантического значения его левой части используется действие по умолчанию (см. раздел 4.5.3 Действия). Точно так же, для положений используется другое действие по умолчанию. Однако действия для положений в большинстве случаев достаточно, в том смысле, что обычно не нужно описывать формирование @$ для каждого правила. При вычислении нового положения для данной группы по умолчанию анализатор берёт начало первого символа и конец последнего.

2.6 Выходной текст Bison: файл анализатора

Когда вы запускаете Bison, вы подаёте ему на вход файл грамматики Bison. Выходным текстом является исходный текст на C, осуществляющий разбор языка, описываемого грамматикой. Этот файл называется *анализатором Bison*. Имейте в виду, что утилита Bison и анализатор Bison -- это две разные программы: утилита Bison -- это программа, создающая на выходе анализатор Bison, который затем становится частью вашей программы.

Задачей анализатора Bison является сборка лексем в группы в соответствии с правилами грамматики, например, объединение идентификаторов и знаков операций в выражения. По мере выполнения этой задачи анализатор выполняет действия, сопоставленные используемым правилам грамматики.

Лексемы поступают из функции, называемой *лексическим анализатором*, которую вы должны каким-либо образом предоставить (например, написав её на C). Анализатор Bison вызывает лексический анализатор каждый раз, когда ему нужна новая лексема. Он не знает, что находится "внутри" лексемы (хотя её семантическое значение может отражать это). Обычно лексический анализатор получает лексемы анализом литер текста, но Bison не зависит от этого. См. раздел 5.2 Функция лексического анализатора yylex.

Файл анализатора Bison -- это код на C, определяющий функции yyparse, реализующую грамматику. Эта функция не образует целую программу на C -- вы должны предоставить некоторые дополнительные функции. Одна из них -- лексический анализатор. Другая -- функция, вызываемая анализатором для сообщения об ошибке. Кроме того, выполнение программы на C должно начинаться с функции main: вы должны создать её и вызывать из неё yyparse, иначе анализатор никогда не заработает. См. раздел 5. Интерфейс анализатора на C.

Все имена переменных и функций в файле анализатора Bison, помимо определённых в написанных вами действиях и имён типов лексем, начинаются с `yy' или `YY'. Сюда входят интерфейсные функции, такие как функция лексического анализатора yylex, функция сообщения об ошибке yyerror и сама функция анализатора yyparse. Также это относится к многочисленным идентификаторам, используемым во внутренних целях. Поэтому вам следует избегать использования идентификаторов C, начинающихся с `yy' или `YY' в грамматике Bison, за исключением определённых в этом руководстве.

В некоторых случаях файл анализатора Bison включает системные заголовки, и тогда при написании вашего кода следует учитывать, что некоторые идентификаторы зарезервированы такими заголовками. На некоторых не-GNU системах включаются заголовки <alloca.h>, <stddef.h> и <stdlib.h>, поскольку это необходимо для объявления функций выделения памяти и связанных типов. Другие системные заголовки могут быть включены, если вы придадите ненулевое значение YYDEBUG (см. раздел 9. Отладка вашего анализатора).

2.7 Этапы использования Bison

Реальный процесс разработки языка с использованием Bison, от спецификации грамматики до работающего компилятора или интерпретатора, содержит следующие этапы:

1. Формально описать грамматику в виде, распознаваемом Bison (см. раздел 4. Файлы грамматики Bison). Для каждого правила грамматики языка описать действия, которые должны выполняться при распознавании текста, соответствующего этому правилу. Действие описывается последовательностью операторов C.
2. Написать лексический анализатор для обработки входного текста и передачи лексем анализатору. Лексический анализатор может быть написан вручную на C (см. раздел 5.2 Функция лексического анализатора yylex). Он также может быть создан с помощью Lex, но использование Lex в этом руководстве не обсуждается.
3. Написать управляющую функцию, вызывающую анализатор, созданный Bison.
4. Написать процедуру сообщения об ошибках.

Чтобы превратить этот исходный код в работающую программу, вы должны выполнить следующие шаги:

1. Обработайте описание грамматики Bison чтобы получить анализатор.
2. Скомпилируйте код, созданный Bison, так же, как любой другой файл с исходным кодом.
3. Соберите объектные файлы чтобы получить конечный продукт.

2.8 Обзор схемы грамматики Bison

Входной файл утилиты Bison -- это *файл грамматики Bison*. Общий вид файла грамматики Bison следующий:

%{
Объявления C
%}

Объявления Bison

%%
Правила грамматики
%%
Дополнительный код на C

`%%', `%{' и `%}' -- это знаки пунктуации, присутствующие в любом файле грамматики Bison для разделения его секций.

Объявления C могут определять типы и переменные, используемые в действиях. Вы также можете использовать команды препроцессора для определения используемых там макросов и #include для включения файлов заголовков, делающих всё вышеперечисленное.

Объявления Bison задают имена терминальных и нетерминальных символов и могут также описывать приоритет операций и типы данных семантических значений различных символов.

Правила грамматики определяют, как каждый нетерминальный символ собирается из своих частей.

Дополнительный код на C может содержать любой код на C, который вы хотите использовать. Часто здесь находится определение лексического анализатора yylex и подпрограммы, вызываемые действиями правил грамматики. В простых программах здесь может находится и вся остальная часть программы.

3. ПРИМЕРЫ

Сейчас мы приведём и объясним три простые программы, написанные с использованием Bison: калькулятор обратной польской нотации, калькулятор алгебраической (инфиксной) нотации, и многофункциональный калькулятор. Все три протестированы под BSD Unix 4.3, каждая из них даёт пригодный для использования, хотя и ограниченный, интерактивный настольный калькулятор.

Эти примеры просты, но грамматики Bison для реальных языков программирования пишутся таким же образом.

3.1 Калькулятор обратной польской нотации

Первый пример -- это простой калькулятор с двойной точностью для выражений в *обратной польской нотации* (использующий постфиксные операции). Этот пример является хорошей отправной точкой, поскольку приоритеты операций не используются. Обработка приоритетов будет показана во втором примере.

Исходный код этого калькулятора называется `rpcalc.y'. По соглашению для входных файлов Bison используется расширение `.y'.

3.1.1 Объявления для rpcalc

Это объявления C и Bison для калькулятора обратной польской нотации. Как и в C, комментарии помещаются между `/*...*/'.

```
/* Калькулятор обратной польской нотации. */

%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM

%% /* Далее следуют правила грамматики и действия */
```

Секция объявлений C (см. раздел 4.1.1 Секция объявлений C содержит две директивы препроцессора.

Директива #define определяет макрос YYSTYPE. Это задаёт тип данных C для семантических значений как лексем, так и групп (см. раздел 4.5.1 Типы данных семантических значений). Анализатор Bison будет использовать любой тип, заданный YYSTYPE, а если вы не определили его -- тип по умолчанию int. Поскольку мы указали double, с каждой лексемой и каждым выражением будет ассоциировано вещественное число.

Директива #include используется для объявления функции возведения в степень pow.

Из второй секции, объявлений Bison, Bison получает информацию о типах лексем (см. раздел 4.1.2 Секция объявлений Bison). Здесь должен быть объявлен любой терминальный символ, не являющийся однолитерной константой (они, как правило, не нуждаются в объявлении). В этом примере все арифметические операции обозначаются однолитерными константами, поэтому нужно объявить только терминальный символ NUM, тип лексемы для числовых констант.

3.1.2 Правила грамматики для rpcalc

Это правила грамматики для калькулятора обратной польской нотации.

```
input:    /* пусто */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM           { $$ = $1;         }
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        | exp exp '*'     { $$ = $1 * $2;    }
        | exp exp '/'     { $$ = $1 / $2;    }
```

```
      /* возведение в степень */
        | exp exp '^'     { $$ = pow ($1, $2); }
      /* унарный минус         */
        | exp 'n'         { $$ = -$1;        }
;
%%
```

Здесь определены группы "языка" rpcalc: выражение (названное exp), строка ввода (line), и законченный входной текст (input). У каждого из этих нетерминальных символов имеется несколько альтернативных правил, объединённых знаком `|', читающимся "или". В последующих разделах объясняется, что означают эти правила.

Семантика языка определяется действиями, предпринимаемыми при распознавании группы. Действия -- это код на C, находящийся между фигурными скобками. См. раздел 4.5.3 Действия.

Вы должны писать эти действия на C, однако Bison предоставляет способ передачи семантических значений между правилами. В каждом действии псевдопеременная $$ обозначает семантическое значение группы, которую собирает это правило. Присвоение $$ значения -- основная работа большинства действий. На семантические значения компонентов правила можно ссылаться как на $1, $2 и т.д.

3.1.2.1 Объяснение input

Рассмотрим определение input:

```
input:    /* пусто */
        | input line
;
```

Это определение читается следующим образом: "Законченный входной текст представляет собой пустую строку либо законченный входной текст, за которым следует входная строка". Обратите внимание, что "законченный входной текст" определяется в терминах самого себя. Это определение называется *леворекурсивным*, поскольку input всегда является самым левым символом по-

следовательности. См. раздел 4.4 Рекурсивные правила.

Первая альтернатива пуста, поскольку между двоеточием и первым знаком `|' нет символов. Это означает, что input может соответствовать пустой входной строке (без лексем). Мы пишем так правило, потому что допустимо нажатие клавиш Ctrl-d сразу после запуска калькулятора. По соглашению пустая альтернатива ставится первой и в ней пишется комментарий `/* пусто */'.

Второе альтернативное правило (input line) описывает любой нетривиальный входной текст. Оно означет "После прочтения любого количества строк, прочитать ещё одну, если это возможно". Левая рекурсия заставляет это правило выполняться в цикле. Поскольку первая альтернатива соответствует пустому входному тексту, цикл будет выполняться ноль или более раз.

Функция анализатора yyparse продолжает обработку входного текста до тех пор, пока не будет обнаружена грамматическая ошибка или лексические анализатор не скажет, что входных лексем больше нет -- мы условимся, что это произойдёт по достижении конца файла.

3.1.2.2 Объяснение line

Теперь рассмотрим определение line:

```
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;
```

Первая альтернатива -- это лексема литеры новой строки, это означает, что rpcalc принимает пустую строку (и игнорирует её, поскольку там нет никакого правила). Вторая альтернатива -- это выражение, за которым следует литера новой строки. Именно эта альтернатива несёт основную пользу rpcalc. Семантическое значение группы exp -- это значение $1, потому что искомое exp -- первый символ альтернативы. Действие выводит это значение, которое является результатом вычислений, заданных пользователем.

Это действие необычно, потому что оно не присваивает значе-

ния $$. Вследствие этого семантическое значение line не инициализируется (значение будет непредсказуемым). Было бы ошибкой в программе, если бы это значение когда-либо использовалось, но мы не пользуемся им -- после того, как rpcalc вывел значение введённой пользователем входной строки, оно больше не нужно.

3.1.2.3 Объяснение expr

Группа exp имеет несколько правил, по одному на каждый вид выражений. Первое правило обрабатывает наиболее простым выражениям -- отдельным числам. Второе обрабатывает выражение сложения, которые выглядит как два выражения, за которыми следует знак `плюс'. Третье обрабатывает вычитание и т.д.

```
exp:      NUM
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        ...
        ;
```

Мы используем `|' чтобы объединить все правила для exp, но мы могли бы с тем же успехом написать их отдельно:

```
exp:      NUM ;
exp:      exp exp '+'     { $$ = $1 + $2;    } ;
exp:      exp exp '-'     { $$ = $1 - $2;    } ;
        ...
```

У большей части правил есть действия, вычисляющие значение выражения из значений его частей. Например, в правиле для сложения $1 относится к первому компоненту exp, а $2 -- ко второму. Третий компонент, '+' не имеет осмысленного ассоциированного семантического значения, но если бы он имел его, на него можно было ссылаться как на $3. Когда yyparse, используя это правило, распознаёт выражение-сумму, сумма значений двух подвыражений даст значение всего выражения. См. раздел 4.5.3 Действия.

Вы не обязаны приписывать действие каждому правилу. Когда у правила нет действия, по умолчанию Bison копирует значение $1 в $$. Именно это происходит в первом правиле (используюшем

NUM).

Показанный здесь способ форматирования -- рекомендуемое соглашение, но Bison не требует этого. Вы можете добавлять или изменять промежутки по своему усмотрению. Например, такая запись:

```
exp   : NUM | exp exp '+' {$$ = $1 + $2; } | ...
```

означает то же, что и:

```
exp:      NUM
        | exp exp '+'    { $$ = $1 + $2; }
        | ...
```

Однако последняя намного более наглядна.

3.1.3 Лексический анализатор rpcalc

Задачей лексического анализатора является низкоуровневый разбор -- преобразование литер или последовательностей литер входного текста в лексемы. Анализатор Bison получает эти лексемы, вызывая лексический анализатор. См. раздел 5.2 Функция лексического анализатора yylex.

Для работы калькулятора обратной польской нотации нужен весьма простой лексический анализатор. Он пропускает пробелы и знаки табуляции, читает числа как значения типа double и возвращает их как лексемы типа NUM. Любые другие литеры, не являющиеся частью числа, считаются отдельными лексемами. Имейте в виду, что код лексемы для таких однолитерных лексем -- это сама литера.

Возвращаемое значение функции лексического анализатора -- это числовой код, представляющий тип лексемы. Текст, используемый в правилах Bison для обозначения типа лексемы, также является выражением C для числового кода этого типа. Это может работать двумя способами. Если тип лексемы является литерой, её числовым кодом будет ASCII-код этой литеры, вы можете использовать в лексическом анализаторе в качестве числа ту же литеру. Если тип лексемы -- идентифи- катор, этот идентификатор

Bison определяет как макрос C, определением которого будет подходящее число. В этом примере, поэтому, NUM становится макросом, используемым yylex.

Семантическое значение лексемы (если оно у неё есть) сохраняется в глобальной переменной yylval, где её и ищет анализатор Bison (тип данных yylval -- YYSTYPE, определяемый в начале грамматики; см. раздел 3.1.1 Объявления для rpcalc).

Нулевой код типа лексемы возвращается, если обнаружен конец файла (Bison считает указателем конца входного файла любые неположительные значения).

Ниже приведён код лексического анализатора:

```
/* Лексический анализатор возвращает вещественное число
   с двойной точностью в стеке и лексему NUM, или прочитанную
   литеру ASCII, если это не число. Все пробелы и знаки
   табуляции пропускаются, в случае конца файла возвращается
0. */

#include <ctype.h>

int
yylex (void)
{
  int c;

  /* пропустить промежутки  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* обработка чисел       */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* вернуть конец файла  */
  if (c == EOF)
    return 0;
```

```
  /* вернуть одну литеру */
  return c;
}
```

3.1.4 Управляющая функция

В соответствии с духом этого примера управляющая функция сведена к явному минимуму. Единственное требование -- она должна вызывать yyparse чтобы запустить процесс разбора.

```
int
main (void)
{
  return yyparse ();
}
```

3.1.5 Подпрограмма сообщения об ошибках

Когда yyparse обнаруживает синтаксическую ошибку, она вызывает функцию сообщения об ошибках yyerror для вывода сообщения об ошибке (обычно, но не всегда, "ошибка разбора"). Предоставить эту функцию должен программист (см. раздел 5. Интерфейс анализатора на C), поэтому приведём используемое нами определение:

```
#include <stdio.h>

void
yyerror (const char *s)  /* вызывается yyparse в случае ошибки */
{
  printf ("%s\n", s);
}
```

После возврата из функции yyerror анализатор Bison може произвести восстановление после ошибки и продолжить разбор, если грамматика содержит подходящее правило ошибки (см. раздел 7. Восстановление после ошибок). В противном случае yyparse вернёт ненулевое значение. В этом примере мы не писали никаких правил ошибки, поэтому любой неправильный входной текст приведёт к завершению работы калькулятора. Это поведение неудачно для настоящего калькулятора, но вполне подходит для первого примера.

3.1.6 Запуск Bison для создания анализатора

Перед запуском Bison для создания анализатора нам нужно решить, хранить весь исходный код в одном или нескольких файлах. Для такого простого примера проще всего будет поместить всё в один файл. Определение yylex, yyerror и main находятся в конце файла, в секции "дополнительного кода на C" (см. раздел 2.8 Обзор схемы грамматики Bison).

В больших проектах, скорее всего, у вас будет несколько файлов с исходным кодом, и вы будет использовать make для их перекомпиляции.

Если весь исходный код находится в одном файле, используйте следующую команду для преобразования его в файл анализатора:

bison *имя_файла*.y

В этом примере файл называется `rpcalc.y' ("Reverse Polish CALCulator" -- калькулятор обратной польской нотации). Bison создаёт файл `*имя_файла*.tab.c', убирая `.y' из названия исходного файла. Выходной файл Bison содержит исходный код yyparse. Дополнительные функции (yylex, yyerror и main) в точности копируются из входного файла в выходной.

3.1.7 Компиляция файла анализатора

Так нужно компилировать и запускать файл анализатора:

```
# Перечислить файлы в текущем каталоге.
$ ls rpcalc.tab.c  rpcalc.y

# Компиляция анализатора Bison.
# `-lm' указывает компилятору искать pow в
# математической библиотеке.
$ cc rpcalc.tab.c -lm -o rpcalc

# Снова перечислить файлы.
$ ls
rpcalc  rpcalc.tab.c  rpcalc.y
```

Файл `rpcalc' теперь содержит исполняемый код. Вот пример сеанса работы с rpcalc.

```
$ rpcalc
4 9 +
13
3 7 + 3 4 5 *+-
-13
3 7 + 3 4 5 * + - n            Обратите внимание на унарный минус
                               `n'
13
5 6 / 4 n +
-3.166666667
3 4 ^                          Возведение в степень
81
^D                             Признак конца файла
$
```

3.2 Калькулятор инфиксной нотации: calc

Теперь мы модифицируем rpcalc для обработки инфиксных, а не постфиксных операций. Инфиксная нотация предполагает принцип приоритета операций и необходимость обработки скобок произвольной глубины вложенности. Вот код Bison файла `calc.y' -- инфиксного настольного калькулятора.

```
/* Калькулятор для выражени в инфиксной нотации -- calc */

%{
#define YYSTYPE double
#include <math.h>
%}

/* Объявления BISON */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* обращение -- унарный минус */
%right '^'    /* возведение в степень       */

/* Далее следует грамматика */
%%
```

```
input:    /* пустая строка */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

Функции yylex, yyerror и main могут быь теми же, что и раньше.

В этом коде показаны две новые важные возможности.

Во второй секции (объявления Bison) %left объявляет типы лексем и говорит, что они обозначают левоассоциативные операции. Объявления %left и %right (правая ассоциативность) используются вместо %token, использующегося для объявления имени типа лексемы без ассоциативности (эти лексемы являются однолитерными константами, которые обычно не требуют объявления. Мы объявляем их здесь для указания ассоциативности).

Приоритет операций определяется порядком строк с объявлениями -- чем больше номер строки (чем ниже она на странице или на экране), тем выше приоритет. Отсюда, возведение в степень имеет наивысший приоритет, далее идут унарный минус (NEG), `*' и `/' и т.д. См. раздел 6.3 Приоритет операций.

Другая новая возможность -- %prec в секции грамматики для операции унарного минуса. %prec просто указывает Bison, что правило `| '-' exp' имеет тот же приоритет, что и NEG -- в данном случае следующий за наивысшим. См. раздел 6.4 Контекстно-

зависимый приоритет.

Вот пример работы с `calc.y':

```
$ calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
-56 + 2
-54
3 ^ 2
9
```

3.3 Простое восстановление после ошибок

До сих пор это руководство не касалось вопроса *восстановления после ошибок* -- как продолжить разбор после того, как анализатор обнаружил синтаксическую ошибку. Всё, что мы предпринимали -- это выдача сообщения об ошибке функцией yyerror. Вспомним, что по умолчанию после вызова yyerror yyparse завершает работу. Это значит, что неправильный ввод приведёт к завершению работы калькулятора. Сейчас мы покажем, как исправить этот недостаток.

Язык Bison содержит зарезервированное слово error, которое можно включить в правило грамматики. В следующем примере оно добавлено к одной из альтернатив line:

```
line:     '\n'
        | exp '\n'   { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok;                  }
;
```

Это добавление к грамматике допускает простое восстановление после ошибок в случае ошибки разбора. Если читается выражение, которое не может быть вычислено, третьим правилом для line будет распознана ошибка, и разбор продолжится (однако функция yyerror по-прежнему используется для вывода сообщения). Действие выполняет оператор yyerrok, автоматически определяемый Bison макрос. Он означает, что восстановление после ошибки завершено (см. раздел 7. Восстановление после ошибок). Обратите внимания на различие между yyerrok и yyerror -- ни то,

ни другое не опечатка.

Этот вид восстановления после ошибок работает с синтаксическими ошибками. Есть другие виды ошибок, например, деление на ноль, вызывающее сигнал исключения, обычно являющегося фатальным. Настоящая программа калькулятора должна обрабатывать этот сигнал и использовать longjmp для возврата в функцию main и продолжения разбора входных строк. Ей следует также отбросить остаток текущей строки входа. Мы не будем далее обсуждать этот вопрос, потому что он не характерен для программ Bison.

3.4 Калькулятор с отслеживаием положений: ltcalc

Этот пример расширяет калькулятор инфиксной нотации отслеживанием положений. Это свойство будет использоваться для улучшения сообщений об ошибках. Для большей ясности этот пример -- простой целочисленный калькулятор, поскольку большая часть изменений, необходимых для использования положений, будет сделана в лексическом анализаторе.

3.4.1 Объявления ltcalc

Объявления C и Bison для калькулятора с отслеживанием положение те же самые, что и для калькулятора инфиксной нотации.

```
/* Калькулятор с отслеживанием положений.  */

%{
#define YYSTYPE int
#include <math.h>
%}

/* Объявления Bison.  */
%token NUM

%left '-' '+'
%left '*' '/'
%left NEG
%right '^'
```

%% /* Далее следует грамматика */

Заметьте, что специальных объявлений для работы с положениями нет. Определять тип данных для сохранения положений не нужно, мы будем использовать тип, заданный по умолчанию (см. раздел 4.6.1 Тип данных положений), являющийся структурой с четырьмя следующими целочисленными полями: first_line, first_column, last_line и last_column.

3.4.2 Правила грамматики ltcalc

Обрабатываются положения или нет, не влияет на синтаксис вашего языка. Поэтому правила грамматики языка в этом примере будут очень похожи на правила в предыдущем примере, мы только модифицируем их для работы с новой информацией.

В этом примере мы будем использовать положения для сообщения о делении на ноль, и определения места неправильных выражений и подвыражений.

```
input   : /* пусто */
        | input line
;

line    : '\n'
        | exp '\n' { printf ("%d\n", $1); }
;

exp     : NUM           { $$ = $1; }
        | exp '+' exp   { $$ = $1 + $3; }
        | exp '-' exp   { $$ = $1 - $3; }
        | exp '*' exp   { $$ = $1 * $3; }
        | exp '/' exp
            {
              if ($3)
                $$ = $1 / $3;
              else
                {
                  $$ = 1;
                  fprintf (stderr, "%d.%d-%d.%d: деление на ноль",
                           @3.first_line, @3.first_column,
```

```
                @3.last_line, @3.last_column);
        }
      }
    | '-' exp %prec NEG     { $$ = -$2; }
    | exp '^' exp           { $$ = pow ($1, $3); }
    | '(' exp ')'           { $$ = $2; }
```

Этот код показывает, как получать значения положений изнутри семантических действий, используя псевдопеременные @*n* для компонентов правила и @$ -- для групп.

Нам не нужно присваивать значение @$, создаваемый анализатор делает это автоматически. По умолчанию перед выполнением кода на C для каждого действия @$ правила с *n* компонентами сопоставляется интервал от начала @1 до конца @*n*. Это поведение может быть переопределено (см. раздел 4.6.3 Действие по умолчанию для положений), а для очень специфических правил @$ может вычисляться вручную.

3.4.3 Лексический анализатор ltcalc.

До сих пор для отслеживания положений мы полагались на значения Bison по умолчанию. Следующим шагом мы перепишем лексический анализатор и сделаем его способным передавать в анализатор положения лексем, как он уже делает это с их семантическими значениями.

Для этой цели мы должны принимать в расчёт каждую литеру входного текста, чтобы избежать того, что вычисленные положения будут неверными.

```
int
yylex (void)
{
  int c;

  /* пропустить промежутки */
  while ((c = getchar ()) == ' ' || c == '\t')
    ++yylloc.last_column;

  /* шаг */
```

```
  yylloc.first_line = yylloc.last_line;
  yylloc.first_column = yylloc.last_column;

  /* обработка чисел */
  if (isdigit (c))
    {
      yylval = c - '0';
      ++yylloc.last_column;
      while (isdigit (c = getchar ()))
        {
          ++yylloc.last_column;
          yylval = yylval * 10 + c - '0';
        }
      ungetc (c, stdin);
      return NUM;
    }

  /* вернуть конец файла */
  if (c == EOF)
    return 0;

  /* вернуть однц литеру и обновить положение */
  if (c == '\n')
    {
      ++yylloc.last_line;
      yylloc.last_column = 0;
    }
  else
    ++yylloc.last_column;
  return c;
}
```

В основном, лексический анализатор делает то же, что и раньше: пропускает пробелы и знаки табуляции и читает числа или однолитерные лексемы. Дополнительно он обновляет yylloc, глобальную переменную (типа YYLTYPE), содержащую положение лексемы.

Теперь, каждый раз, когда функция возвращет лексему, в распоряжении анализатора находятся как её номер, так и семантическое значение и положение в тексте. Последнее необходимое изменение -- инициализация yylloc, например, в управляю-

щей функции.

```
int
main (void)
{
  yylloc.first_line = yylloc.last_line = 1;
  yylloc.first_column = yylloc.last_column = 0;
  return yyparse ();
}
```

Помните, что вычисление положений не касается синтаксиса. Каждая литера должна быть ассоциирована с обновлением информации о положении, независимо от того, находится ли она в правильном входном тексте, в комментариях, в строковой константе и т.д.

3.5 Многофункциональный калькулятор: mfcalc

Теперь, когда мы уже обсудили основы Bison, пришло время перейти к более сложной задаче. Приведённые выше калькуляторы поддерживали только пять функций: `+', `-', `*', `/' и `^'. Было бы приятно иметь калькулятор, предоставляющий другие математические функции, такие как sin, cos и т.д.

Добавить новые операции в инфиксный калькулятор, до тех пор, пока они обозначаются односимвольными константами, несложно. Лексический анализатор yylex возвращает все нечисловые литеры как лексемы, так что для добавления операции достаточно введения нового правила грамматики. Но мы хотим иметь нечто более гибкое, встроенные функции, синтаксис которых выглядит так:

имя_функции (*аргумент*)

В то же время, мы добавим в калькулятор память, допуская создание именованных переменных, сохранение в них значений и последующее использование. Вот пример сеанса работы многофункционального калькулятора:

```
$ mfcalc
pi = 3.141592653589
3.1415926536
```

```
sin(pi)
0.0000000000
alpha = beta1 = 2.3
2.3000000000
alpha
2.3000000000
ln(alpha)
0.8329091229
exp(ln(beta1))
2.3000000000
$
```

Обратите внимание, что допускаются множественное присваивание и вложенные вызовы функций.

3.5.1 Объявления mfcalc

Вот объявления C и Bison для многофункционального калькулятора.

```
%{
#include <math.h>  /* Математические функции: cos(), sin() и т.д.
*/
#include "calc.h"  /* Содержит определение `symrec'            */
%}
%union {
double     val;  /* Чтобы возвращать числа.                   */
symrec  *tptr;   /* Чтобы возвращать указатели таблицы символов
*/
}

%token <val>  NUM        /* Простое число двойной точности   */
%token <tptr> VAR FNCT   /* Переменная и функция             */
%type  <val>  exp

%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* Обращение -- унарный минус */
%right '^'    /* Возведение в степень       */
```

```
/* Далее следует грамматика */

%%
```

Вышеприведённая грамматика вводит только две новые возможности языка Bison. Эти возможности позволяют семантическим значениям иметь различные типы данных (см. раздел 4.5.2 Несколько типов значений).

Объявление %union задаёт весь список возможных типов, это употребляется вместо определения YYSTYPE. Допустимые типы теперь: вещественные числа двойной точности (для exp и NUM) и указатели на элементы таблицы символов. См. раздел 4.7.3 Набор типов значений.

Поскольку значения теперь могут иметь различные типы, необходимо ассоциировать тип с каждым символом грамматики, семантическое значение которого используется. Эти символы: NUM, VAR, FNCT и exp. Их объявления дополнены информацией об их типах данных (помещённой в угловых скобках).

Конструкция Bison %type используется для объявления нетерминальных символов, так же как %token используется для объявления типов лексем. Ранее мы не использовали %type, потому что нетерминальные символы обычно неявно объявляются правилами, определяющими их. Но exp должно быть объявлено явно чтобы можно было задать тип его значения. См. раздел 4.7.4 Нетерминальные символы.

3.5.2 Правила грамматики mfcalc

Вот правила грамматики многофункционального калькутора. Большая их часть напрямую скопирована из calc. Введено три новых правила, использующие VAR и FNCT.

```
input:   /* пусто */
       | input line
;

line:
         '\n'
```

```
        | exp '\n'   { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok;                  }
;

exp:      NUM               { $$ = $1;                         }
        | VAR             { $$ = $1->value.var;              }
        | VAR '=' exp       { $$ = $3; $1->value.var = $3;     }
        | FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
        | exp '+' exp       { $$ = $1 + $3;                    }
        | exp '-' exp       { $$ = $1 - $3;                    }
        | exp '*' exp       { $$ = $1 * $3;                    }
        | exp '/' exp       { $$ = $1 / $3;                    }
        | '-' exp  %prec NEG { $$ = -$2;                       }
        | exp '^' exp       { $$ = pow ($1, $3);               }
        | '(' exp ')'       { $$ = $2;                         }
;
/* End of grammar */
%%
```

3.5.3 Таблица символов mfcalc

Многофункциональному калькулятору требуется таблица символов для отслеживания имён и значений переменных и функций. Это не влияет на правила грамматики (за исключением действий) или объявления Bison, но требует введения некоторых дополнительных функций на C.

Сама таблица символов состоит из связанного списка записей. Её определение, находящееся в заголовке `calc.h', приведено далее. Оно позволяет размещать в таблице как функции, так и переменные.

```
/* Тип функций.                                */
typedef double (*func_t) (double);

/* Тип данных для связей в цепочке символов.        */
struct symrec
{
  char *name;  /* имя символа                       */
  int type;    /* тип символа: либо VAR, либо FNCT   */
  union
  {
    double var;                    /*    значение VAR     */
```

```
    func_t fnctptr;             /* значение FNCT    */
  } value;
  struct symrec *next;    /* поле связи             */
};

typedef struct symrec symrec;

/* Таблица символов: цепочка `struct symrec'.       */
extern symrec *sym_table;

symrec *putsym (const char *, func_t);
symrec *getsym (const char *);
```

Новая версия main включает вызов init_table, функции, инициализирующей таблицу символов. Вот текст этих двух функций

```
#include <stdio.h>

int
main (void)
{
  init_table ();
  return yyparse ();
}

void
yyerror (const char *s)  /* Вызывается yyparse в случае ошибки */
{
  printf ("%s\n", s);
}

struct init
{
  char *fname;
  double (*fnct)(double);
};

struct init arith_fncts[] =
{
  "sin",  sin,
  "cos",  cos,
  "atan", atan,
```

```
  "ln",   log,
  "exp",  exp,
  "sqrt", sqrt,
  0, 0
};

/* Таблица символов: цепочка `struct symrec'.  */
symrec *sym_table = (symrec *) 0;

/* Поместить арифметические функции в таблицу. */
void
init_table (void)
{
  int i;
  symrec *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++)
    {
      ptr = putsym (arith_fncts[i].fname, FNCT);
      ptr->value.fnctptr = arith_fncts[i].fnct;
    }
}
```

Вы можете добавить к калькулятору дополнительные функции, просто редактируя список инициализации и включая необходимые файлы заголовков.

Две важные функции позволяют просматривать символы в таблице и вводить новые. Функции putsym передаётся имя и тип (VAR или FNCT) заносимого объекта. Объект включается в начало списка, и возвращается указатель на объект. Функции getsym передаётся имя искомого символа. Если он найден, возвращается указатель на него, иначе же ноль.

```
symrec *
putsym (char *sym_name, int sym_type)
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof (symrec));
  ptr->name = (char *) malloc (strlen (sym_name) + 1);
  strcpy (ptr->name,sym_name);
  ptr->type = sym_type;
  ptr->value.var = 0; /* set value    to 0 even if fctn.  */
```

```
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec *
getsym (const char *sym_name)
{
  symrec *ptr;
  for (ptr = sym_table; ptr != (symrec *) 0;
       ptr = (symrec *)ptr->next)
    if (strcmp (ptr->name,sym_name) == 0)
      return ptr;
  return 0;
}
```

Функция yylex теперь должна распознавать переменные, числовые значения и односимвольные арифметические операции. Строки алфавитноцифровых литер, начинающиеся не с цифры, распознаются или как переменные, или как функции, в зависимости от того, что говорится о них в таблице символов.

Строка передаётся функции getsym для поиска в таблице символов. Если имя встречается в таблице, в yyparse возвращаются указатель на его положение и его тип (VAR или FNCT). Если в таблице его ещё нет, оно заносится как VAR, используя putsym. Опять же, указатель и тип (который должен быть VAR) возвращаются в yyparse.

Для обработки числовых значений и арифметических операций изменения в yylex не нужны.

```
#include <ctype.h>

int
yylex (void)
{
  int c;

  /* Игнорировать промежутки, получить первый непробельный символ.  */
```

```
  while ((c = getchar ()) == ' ' || c == '\t');

  if (c == EOF)
    return 0;

  /*  С  литеры  начинается  число  =>  разобрать  число.
*/
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval.val);
      return NUM;
    }

  /*  С  литеры  начинается  идентификатор  =>  читать  имя.
*/
  if (isalpha (c))
    {
      symrec *s;
      static char *symbuf = 0;
      static int length = 0;
      int i;

      /* Первоначально сделать буфер достаточно большим
         для имени символа из 40 литер.  */
      if (length == 0)
        length = 40, symbuf = (char *)malloc (length + 1);

      i = 0;
      do
        {
          /* Если буфер полон, расширить его.          */
          if (i == length)
            {
              length *= 2;
              symbuf = (char *)realloc (symbuf, length + 1);
            }
          /* Добавить эту литеру в буфер.              */
          symbuf[i++] = c;
          /* Получить следующую литеру.                */
          c = getchar ();
        }
```

```
    while (c != EOF && isalnum (c));

    ungetc (c, stdin);
    symbuf[i] = '\0';

    s = getsym (symbuf);
    if (s == 0)
      s = putsym (symbuf, VAR);
    yylval.tptr = s;
    return s->type;
  }

 /* Любая другая литера сама по себе является лексемой.        */
 return c;
}
```

Эта программа одновременно и достаточно мощна, и гибка. Вы можете легко добавлять новые функции и также несложно модифицировать код для введения предопределённых переменных, таких как pi и e.

3.6 Упражнения

1. Добавьте несколько новых функций из `math.h' в список инициализации.
2. Добавьте ещё один массив, содержащий константы и их значения. Потом модифицируйте init_table чтобы внести эти константы в таблицу символов. Проще всего будет придать константам тип VAR.
3. Заставьте программу выводить сообщение об ошибке, если пользователь ссылается на неинициализированную переменную каким-либо образом, кроме присвоения ей значения.

4. ФАЙЛЫ ГРАММАТИКИ BISON

Bison принимает на вход спецификацию контекстно-свободной грамматики и создаёт функцию на языке C, которая распознаёт правильные предложения этой грамматики.

Имя входного файла грамматики Bison по соглашению заканчивается на `.y'. См. раздел 10. Вызов Bison.

4.1 Структура грамматики Bison

Файл грамматики Bison содержит четыре основные секции, показанные здесь с соответствующими разделителями.

%{
Объявления C
%}

Объявления Bison

%%
Правила грамматики
%%

Дополнительный код на C

Комментарии, заключённые в `/* ... */', могут появляться в любой секции.

4.1.1 Секция объявлений C

Секция *объявлений C* содержит макроопределения и объявления функций и переменных, используемых в действиях правил грамматики. Они копируются в начало файла анализатора так, чтобы они предшествовали определению yyparse. Вы можете использовать `#include' для получения объявлений из файлов заголовков. Если вам не нужны какие-либо объявления C, вы можете опустить ограничивающие эту секцию `%{' и `%}'.

4.1.2 Секция объявлений Bison

Секция *объявлений Bison* содержит объявления, определяющие терминальные и нетерминальные символы, задающие приоритет и т.д. В некоторых простых грамматиках вам могут быть не нужны никакие объявления. См. раздел 4.7 Объявления Bison.

4.1.3 Секция правил грамматики

Секция *правил грамматики* содержит одно или более правил грамматики Bison и ничего более. См. раздел 4.3 Синтаксис правил грамматики.

Должно быть по меньшей мере одно правило грамматики, и первый ограничитель `%%' (предшествующий правилам грамматики) не может быть опущен, даже если это первая строка файла.

4.1.4 Секция дополнительного кода на C

Секция *дополнительного кода на C* в точности копируется в конец файла анализатора, точно так же, как секция *объявлений C* в начало. Это наиболее удобное место, чтобы поместить что-либо, что вы хотите иметь в файле анализатора, но что не нужно помещать перед определением yyparse. Например, сюда часто помещаются определения yylex и yyerror. См. раздел 5. Интерфейс анализатора на C.

Если последняя секция пуста, вы можете опустить `%%', отделяющее её от правил грамматики.

Сам анализатор Bison содержит множество статических переменных с именами, начинающимися с `yy', и макросов с именами, начинающимися с `YY'. Не использовать такие имена, за исключением описанных в этом руководстве, в секции дополнительного кода C файла грамматики -- хорошая идея.

4.2 Символы, терминальные и нетерминальные

Символы в грамматиках Bison представляют грамматическую

классификацию языка.

Терминальный символ (также известный как *тип лексемы*) представляет класс синтаксически эквивалентных лексем. Вы используете такой символ в правилах грамматики, чтобы обозначить, что допустима лексема этого класса. В анализаторе Bison символ представляется числовым кодом, и функция yylex возвращает код типа лексемы чтобы показать, лексема какого вида прочитана. Вам не нужно знать, каково значение кода, для его обозначения можно использовать сам символ.

Нетерминальный символ обозначает класс синтаксически эквивалентных групп. Имя символа используется при написании правил грамматики. По соглашению оно должно быть записано в нижнем регистре.

Имена символов могут содержать буквы, цифры (но не начинаться с цифры), знаки подчёркивания и точки. Точки имеют значение только в нетерминалах.

Есть три способа записи терминальных символов в грамматике:

- *Именованный тип лексемы* записывается идентификатором, как идентификаторы C. По соглашению он должен быть записан в верхнем регистре. Каждое такое имя должно быть определено в объявлении Bison, например, %token. См. раздел 4.7.1 Имена типов лексем.
- *Тип однолитерной лексемы* (*лексема-однолитерная константа*) записывается в грамматике с использованием того же синтаксиса, что используется в C для литерных констант: например, '+' -- это тип однолитерной лексемы. Тип однолитерной лексемы не надо объявлять, если только вам не нужно задать тип его семантического значения (см. раздел 4.5.1 Типы данных семантических значений), ассоциативность или приоритет (см. раздел 6.3 Приоритет операций). По соглашению тип однолитерной лексемы используется только для представления лексемы, состоящей из этой конкретной литеры. Так, тип лексемы '+' используется для представления литеры `+' в качестве лексемы. Ничто не обязывает вас придерживаться этого соглашения, но если вы не будете этого делать, ваша программа

будет путать других читателей. В Bison также могут быть использованы все обычные escape-последовательности, использующиеся в литерных константах C, но вы не должны использовать нулевой символ в качестве однолитерной константы, поскольку его код ASCII -- ноль -- это код, возвращаемый yylex при обнаружении конца файла. (см. раздел 5.2.1 Соглашения о вызове yylex).

- *Строковый литерал (строковая лексема)* записывается как строковая константа C: например, строковым литералом является "<=". Строковый литерал не надо объявлять, если только вам не нужно задать тип его семантического значения (см. раздел 4.5.1 Типы данных семантических значений), ассоциативность или приоритет (см. раздел 6.3 Приоритет операций). Вы можете связать строковую лексему с символическим именем (псевдонимом), используя объявление %token (см. раздел 4.7.1 Имена типов лексем). Если вы не сделали этого, лексический анализатор должен отыскать номер строковой лексемы в таблице yytname (см. раздел 5.2.1 Соглашения о вызове yylex). **ВНИМАНИЕ:** В Yacc строковые лексемы не работают. По соглашению строковые лексемы используются только для представления лексемы, состоящей из этой конкретной строки. Так, тип лексемы "<=" используется для представления строки `<=' в качестве лексемы. Bison не обязывает вас придерживаться этого соглашения, но если вы не будете этого делать, люди, читающие вашу программу, запутаются. В Bison также могут быть использованы все escape-последовательности, использующиеся в строковых константах C. Строковая лексема должна содержать две или более литеры, для лексем из одной литеры используйте однолитерные лексемы (см. выше).

Выбранный вами способ записи терминального символа не влияет на его грамматический смысл. Оно зависит только от того, где он встречается в правилах, и когда функция анализатора его возвращает.

Значение, возвращаемое yylex, -- всегда один из терминальных символов (или 0 в случае конца входного текста). Каким бы способом вы не записывали тип лексемы в правилах грамматики, в определениия yylex он должен быть описан тем же образом. Числовой код типа однолитерной лексемы -- это просто код ASCII

этой литеры, и таким образом yylex может для получения необходимого кода использовать ту же литерную константу. Каждый именованный тип лексемы в файле анализатора становится макросом C, и yylex может для обозначения кода использовать то же имя (именно поэтому точки в терминальных символах не имеют значения). См. раздел 5.2.1 Соглашения о вызове yylex.

Если yylex определяется в отдельном файле, вам необходимо сделать доступными ей макроопределения типов лексем. Используйте параметр `-d' при запуске Bison, и эти макроопределения будут записаны в отдельный файл заголовка `*имя*.tab.h', который вы можете включать в другие нуждающиеся в нём файлы исходного кода. См. раздел 10. Вызов Bison.

Символ error -- это терминальный символ, зарезервированный для восстановления после ошибок. Вы не должны использовать его для каких-то других целей. В частности, yylex никогда не должна возвращать это значение.

4.3 Синтаксис правил грамматики

Правило грамматики Bison имеет следующий общий вид:

результат: *компоненты*...
    ;

где *результат* -- это описываемый правилом нетерминальный символ, а *компоненты* -- различные терминальные и нетерминальные символы, объединяемые этим правилом (см. раздел 4.2 Символы, терминальные и нетерминальные).

Например:

```
exp:    exp '+' exp
    ;
```

говорит о том, что две группы типа exp и лексема `+' между ними могут быть объединены в более крупную группу типа exp.

Пробельные литеры в правилах только разделяют символы. Вы

можете по своему усмотрению вставлять дополнительные промежутки.

Между компонентами могут быть разбросаны *действия*, определяющие семантику правила. Действие выглядит так:

{*операторы C*}

Обычно в правиле только одно действие, и оно следует после всех компонентов. См. раздел 4.5.3 Действия.

Для одного *результата* можно написать несколько правил, отдельно или же соединённых литерой вертикальной черты `|' как здесь:

```
результат:    компоненты первого правила...
        | компоненты второго правила...
        ...
        ;
```

В любом случае правила рассматриваются как различные, даже если они таким образом объединены.

Есть в правиле нет *компонентов*, это означает, что *результат* может соответствовать пустой строке. Например, вот как определяется последовательность нуля или более групп exp, разделённых запятыми:

```
expseq:   /* пусто */
        | expseq1
        ;

expseq1:  exp
        | expseq1 ',' exp
        ;
```

Традиционно в каждом правиле, не содержащем компонентов, пишется комментарий `/* пусто */'.

4.4 Рекурсивные правила

Правило называется *рекурсивным*, если нетерминал его *результата* появляется также в его правой части. Почти все грамматики Bison должны использовать рекурсию, потому что это единственный способ определить последовательность из произвольного числа элементов. Рассмотрим рекурсивное определение последовательности одного или более выражений, разделённых запятыми:

```
expseq1:  exp
        | expseq1 ',' exp
        ;
```

Поскольку рекурсивный символ expseq1 -- самый левый в правой части, мы называем это *левой рекурсией*. И наоборот, вот та же конструкция, определённая с использованием *правой рекурсии*:

```
expseq1:  exp
        | exp ',' expseq1
        ;
```

Последовательность любого вида может быть определена с использованием как левой, так и правой рекурсии, но вам следует всегда использовать леворекурсивные правила, потому что они могут разобрать последовательность из любого числа элементов, используя ограниченое стековое пространство. Размер используемого праворекурсивными правилами стека Bison пропорционален числу элементов последовательности, поскольку все эти элементы должны быть помещены в стек перед тем, как правило будет применено в первый раз. См. раздел 6. Алгоритм анализатора Bison, по поводу дальнейшего объяснения этого факта.

Косвенная или *взаимная* рекурсия возникает, когда результат правила не появляется непосредственно в правой части, но встречается в правилах для других нетерминалов, появляющихся в его правой части.

Например:

```
expr:     primary
```

```
        | primary '+' primary
        ;

primary:  constant
        | '(' expr ')'
        ;
```

определяет два взаимнорекурсивных правила, поскольку каждое из них ссылается на другое.

4.5 Определение семантики языка

Правила грамматики языка определяют только его синтаксис. Семантика определяется семантическими значениями, сопоставленными различным лексемам и группам, и правилами, выполняющимися при распознавании различных групп.

Например, калькулятор производит правильные вычисления, потому что с каждым выражением сопоставлено числовое значение. Он правильно складывает, потому что действие для группы `*x* + *y*` складывает числа, сопоставленные с *x* и *y*.

4.5.1 Типы данных семантических значений

В простой программе может быть достаточно использование одного и того же типа данных для семантических значений всех языковых конструкций. Это верно для примеров постфиксного и инфиксного калькулятора (см. раздел 3.1 Калькулятор обратной польской нотации).

По умолчанию Bison использует для всех семантических значений тип int. Чтобы задать другой тип, определите макрос YYSTYPE, как здесь:

```
#define YYSTYPE double
```

Это макроопределение должно находиться в секции объявлений C файла грамматики (см. раздел 4.1 Структура грамматики Bison).

4.5.2 Несколько типов значений

В большинстве программ вам будут нужны разные типы данных для разных видов лексем и групп. Например, числовой константе может быть нужен тип int или long, строковой -- тип char *, а идентификатору -- указатель на элемент таблицы символов.

Чтобы в анализаторе можно было использовать несколько типов семантических значений, Bison требует сделать две вещи:

- Задать весь набор возможных типов данных в объявлении Bison %union (см. раздел 4.7.3 Набор типов значений).
- Выбрать для каждого символа (терминального или нетерминального), для которого используются семантическое значение, один из этих типов. Для лексем это делается в объявлении Bison %token (см. раздел 4.7.1 Имена типов лексем), а для групп -- в объявлении Bison %type (см. раздел 4.7.4 Нетерминальные символы).

4.5.3 Действия

Действие сопровождает синтаксическое правило и содержит код на C, который должен выполняться при каждом распознавании текста, соответствующего этому правилу. Задачей большинства действий является вычисление семантического значения группы, собираемой правилом, исходя из семантических значений, сопоставленных лексемам или меньшим группам.

Действие состоит из операторов C, окружённых фигурными скобками, как составной оператор C. Оно может быть помещено в любой точке правила, и выполняется в этой точке. Большинство правил имеют только одно действие в конце правила, после всех компонентов. Действия внутри правила непросты, и используются только в специальных целях (см. раздел 4.5.5 Действия внутри правил).

Код на C действия может ссылаться на семантические значения компонентов, связываемых правилом с помощью конструкции $*n*, обозначающей значение *n*-го компонента. Семантическое значение собираемой группы -- $$ (Bison, копируя действия в файл анализатора, превращает обе эти конструкции в ссылки на эле-

менты массива).

Вот типичный пример:

```
exp:    ...
        | exp '+' exp
          { $$ = $1 + $3; }
```

Это правило собирает exp из двух меньших групп exp, соединённых лексемой `знак плюс'. В действии $1 и $3 ссылаются на семантические значения двух групп-компонентов exp, являющихся первым и третьим символами в правой части правила. Сумма сохраняется в $$, становясь тем самым семантическим значением выражения сложения, только что распознанного правилом. Если бы с лексемой `+' было сопоставлено полезное семантическое значение, на него можно было бы ссылаться как на $2.

Если вы не задаёте действие для правила, Bison применяет действие по умолчанию: $$ = $1. То есть значение первого символа правила становится значением всего правила. Конечно, действие по умолчанию допустимо только если эти два типа данных соответствуют друг другу. Для пустого правила нет осмысленного действия по умолчанию, каждое пустое правило должно иметь явное действие, за исключением случая, когда его значение никогда не будет использоваться.

$*n* с нулевым или отрицательным *n* допустима для ссылки на лексемы и группы, находящиеся в стеке *перед* соответствующими текущему правилу лексемами и группами. Это очень рискованный приём, и чтобы надёжно пользоваться им, вы должны быть уверены, что знаете контекст применения правила. Вот случай, в котором вы можете безопасно использовать его:

```
foo:      expr bar '+' expr  { ... }
        | expr bar '-' expr  { ... }
        ;

bar:      /* пусто */
 */
        { previous_expr = $0; }
        ;
```

Пока bar используется только так, как здесь показано, $0 всегда будет ссылаться на expr, предшествующий bar в определении foo.

4.5.4 Типы данных значений в действиях

Если вы выбрали единственный тип данных семантических значений, конструкции $$ и $*n* всегда имеют этот тип.

Если вы используете %union для задания множества типов данных, вы должны определить выбор из этих типов для каждого терминального или нетерминального символа, который может иметь семантическое значение. Тогда каждый раз, когда вы используете $$ или $*n*, тип данных определяется тем, на какой символ в правиле они ссылаются. В этом примере,

```
exp:    ...
        | exp '+' exp
          { $$ = $1 + $3; }
```

$1 и $3 ссылаются на экземпляры exp, поэтому все они имеют тип данных, объявленный для нетерминального символа exp. Если бы использовался $2, то он имел бы тип данных, объявленный для терминального символа '+', каким бы он ни был.

Вместо этого вы можете указывать тип данных, когда вы ссылаетесь на значение, вставляя `<*тип*>' после `$' в начале ссылки. Например, если вы определите типы так:

```
%union {
  int itype;
  double dtype;
}
```

вы можете написать $<itype>1 чтобы сослаться на первый компонент правила как на целое, или $<dtype>1 -- чтобы сослаться как на вещественное число с двойной точностью.

4.5.5 Действия внутри правил

Время от времени полезно помещать действия внутрь правила.

Эти действия записываются точно так же, как обычные действия в конце правил, но выполняются до того, как анализатор распознает следующие компоненты.

Действие внутри правила может с помощью $*n* ссылаться на компоненты, предшествующие ему, но не на последующие, поскольку выполняется до их разбора.

Действие внутри правила само по себе считается одним из компонентов правила. Это имеет значение, если позднее в том же правиле присутствует ещё одно действие (и, обычно, ещё одно в конце правила): вы должны при вычислении *n* в конструкции $*n* учитывать действия вместе с правилами.

Действие внутри правила также имеет семантическое значение. Действие может установить его значение присваиванием $$, а последующие действия в правиле могут ссылаться на него, используя $*n*. Поскольку для именования действия нет символа, нет способа объявить заранее тип данных его значения, поэтому вы должны использовать конструкцию `$<...>*n*' для задания типа данных каждый раз, когда вы ссылаетесь на это значение.

Установить значение всего правила действием внутри правила невозможно, поскольку присваивание $$ не будет этого делать. Единственный способ задать значение всего правила -- использовать обычное действие в конце правила.

Приведём пример гипотетического компилятора, обрабатывающего оператор let, имеющий вид: `let (*переменная*) *оператор*' и позволяющий создать переменную *переменная* только на время выполнения *оператора*. Для разбора этой конструкции мы должны поместить *переменную* в таблицу символов на время разбора *оператора*, а после этого удалить её. Вот, как это делается:

```
stmt:   LET '(' var ')'
              { $<context>$ = push_context ();
                declare_variable ($3); }
        stmt   { $$ = $6;
                 pop_context ($<context>5); }
```

Как только распознано `let (*переменная*)', выполняется первое

действие. Оно сохраняет копию текущего семантического контекста (список доступных переменных) в качестве своего семантического значения, используя вариант типа данных context. Потом оно вызывает declare_variable, чтобы добавить в этот список новую переменную. Когда работа первого действия завершена, может быть разобран вложенный оператор stmt. Имейте в виду, что действие внутри правила -- это пятый компонент, поэтому `stmt' -- шестой.

После разбора вложенного оператора его семантическое значение становится значением всего оператора let. Потом для восстановления прошлого списка переменных используется семантическое значение предыдущего действия. Это удалит из списка временную let-переменную, и она не будет существовать во время разбора остальной части программы.

Применение действий до того, как правило полностью распознано, часто приводит к конфликтам, поскольку анализатор должен принять определённый вариант разбора чтобы обработать действие. Например, следующие два правила без внутреннего действия могут сушествовать совместно в работающем анализаторе, потому что анализатор может сдвинуть лексему открывающей фигурной скобки и посмотреть, что следует за ней, перед принятием решения, есть там объявление или нет.

```
compound: '{' declarations statements '}'
        | '{' statements '}'
        ;
```

Но если мы добавим внутреннее действие, как показано ниже, правила перестанут работать:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | '{' statements '}'
        ;
```

Теперь анализатор вынужден решать, запускать или нет внутреннее действие, когда он прочитал ещё только открывающую скобку. Другими словами, он должен принять решение об использовании того или иного правила, не имея информации, достаточной

для того, чтобы сделать это правильно (лексема открывающей фигурной скобки в этот момент -- это то, что называется лексемой, *увиденной впереди*, поскольку анализатор всё ещё решает, что с ней делать. См. раздел 6.1 Предпросмотренные лексемы.).

Вы можете думать, что можно решить проблему, расположив в обоих правилах одинаковые действия, как здесь:

```
compound: { prepare_for_local_variables (); }
        '{' declarations statements '}'
      | { prepare_for_local_variables (); }
        '{' statements '}'
      ;
```

Но это не поможет, потому что Bison не осознает, что эти два действия идентичны (Bison никогда не пытается понимать код на C в действиях).

Если грамматика такова, что определение можно отличить от оператора по первой лексеме (что верно для C), то одним из работающих решений будет поместить действие после открывающей скобки, как здесь:

```
compound: '{' { prepare_for_local_variables (); }
        declarations statements '}'
      | '{' statements '}'
      ;
```

Теперь первая лексема следующего определения или оператора, которая в любом случае должна сообщить Bison, какое правило использовать, действительно может это сделать.

Другое решение -- вынести действие в отдельный нетерминальный символ, служащий подпрограммой:

```
subroutine: /* пусто */
        { prepare_for_local_variables (); }
      ;

compound: subroutine
        '{' declarations statements   '}'
```

```
| subroutine
  '{' statements '}'
;
```

Теперь Bison может выполнить действие в правиле для subroutine, не принимая решения, какое из правил для compound использовать в конце концов. Имейте в виду, что действие теперь находится в конце правила. Любые действия внутри правил могут быть таким образом превращены в действия в конце правил, и именно это Bison на самом деле делает для реализации действий внутри правил.

4.6 Отслеживание положений

Хотя правил грамматики и семантических действий и достаточно, чтобы написать полностью функциональный анализатор, может быть полезно обрабатывать некоторую дополнительную информацию, особенно положение символов.

Способ обработки положений определяется указанием типа данных и действия, которые должны выполняться при разборе правил.

4.6.1 Тип данных положений

Определение типа данных для положений гораздо проще, чем для семантических значений, поскольку все лексемы и группы всегда используют один и тот же тип.

Тип положений задаётся определением макроса YYLTYPE. Если YYLTYPE не определён, Bison по умолчанию использует структуру из четырёх членов:

```
struct
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
}
```

4.6.2 Действия и положения

Действия полезны не только для определения семантики языка, но и для описания поведения анализатора, касающегося положений.

Наиболее очевидный способ получения положения синтаксической группы очень похож на способ вычисления семантических значений. Для получения доступа в конкретном правиле к связываемым элементам может использоваться несколько конструкций. Положение *n*-го компонента правой части правила обозначается @*n*, а положение группы левой части -- @$.

Вот простой пример, использующий для положений тип данных по умолчанию:

```
exp:    ...
        | exp '/' exp
           {
             @$.first_column = @1.first_column;
             @$.first_line = @1.first_line;
             @$.last_column = @3.last_column;
             @$.last_line = @3.last_line;
             if ($3)
               $$ = $1 / $3;
             else
               {
                 $$ = 1;
                 printf("Деление на ноль, l%d,c%d-l%d,c%d",
                        @3.first_line, @3.first_column,
                        @3.last_line, @3.last_column);
               }
           }
```

Как и для семантических значений, есть действие по умолчанию для положений, выполняемое при каждом разборе правила. Оно устанавливает начало @$ на начало первого символа, а конец -- на конец последнего символа.

При использовании действия по умолчанию отслеживание по-

ложений может быть полностью автоматическим. Вышеприведённый пример можно переписать так:

```
exp:    ...
        | exp '/' exp
          {
            if ($3)
              $$ = $1 / $3;
            else
              {
                $$ = 1;
                printf("Деление на ноль, l%d,c%d-l%d,c%d",
                       @3.first_line, @3.first_column,
                       @3.last_line, @3.last_column);
              }
          }
```

4.6.3 Действие по умолчанию для положений

На самом деле действия -- не лучшее место для вычисления положений. Поскольку положения гораздо более общи, чем семантические значения, в анализаторе есть место, где можно переопределить действие по умолчанию для каждого правила. Макрос YYLLOC_DEFAULT вызывается при каждом разборе правила, перед запуском связанного с ним действия.

В большинстве случаев этого макроса, в общем, достаточно, чтобы избавиться от специального кода в семантических действиях.

Макрос YYLLOC_DEFAULT принимает три параметра. Первый -- положение группы (результат вычисления). Второй -- массив, содержащий положения всех элементов правой части связываемого правила. Последний -- размер правой части правила.

По умолчанию он определён следующим образом:

```
#define YYLLOC_DEFAULT(Current, Rhs, N)         \
  Current.last_line   = Rhs[N].last_line;       \
  Current.last_column = Rhs[N].last_column;
```

При определении YYLLOC_DEFAULT вы должны считать, что:

- У аргументов нет побочных действий. Однако, YYLLOC_DEFAULT может изменять только первый из них (результат).
- Перед выполнением YYLLOC_DEFAULT анализатор устанавливает @$ равным @1.
- Для обеспечения последовательности с реализацией семантических действий, правильные индексы массива -- от 1 до *n*.

4.7 Объявления Bison

Секция *объявлений Bison* грамматики Bison определяет символы, используемые при формулировке грамматики и типы данных семантических значений. См. раздел 4.2 Символы, терминальные и нетерминальные.

Все имена типов лексем (но не однолитерные лексемы, такие как '+' и '*') должны быть объявлены. Нетерминальные символы должны быть объявлены, если вам нужно задать используемый тип данных семантического значения (см. раздел 4.5.2 Несколько типов значений).

По умолчанию первое правило файла также задаёт начальный символ. Если вы хотите, чтобы начальным символом был какой-то другой, вы должны объявить его явно (см. раздел 2.1 Языки и контекстно-свободные грамматики).

4.7.1 Имена типов лексем

Основной способ объявления имени типа лексемы (терминального символа) следующий:

%token *имя*

В анализаторе Bison превратит это в директиву #define, так что функция yylex (если она присутствует в файле) сможет использовать имя *имя* для обозначения кода этого типа лексемы.

Если вы хотите задать ассоциативность и приоритет, вместо %token вы можете использовать %left, %right или %nonassoc. См. раздел 4.7.2 Приоритет операций.

Вы можете явно задать числовой код типа лексемы, добавив целочисленное значение непосредственно после имени лексемы:

%token NUM 300

Тем не менее, в общем случае лучше позволить Bison определить числовые коды для всех типов лексем самому. Bison автоматически выберет коды, не конфликтующие друг с другом и с литерами ASCII.

В случае, если тип значения -- объединение, вы дожны включить в %token или другие объявления лексем тип данных, заключённый в угловые скобки (см. раздел 4.5.2 Несколько типов значений).

Например:

```
%union {              /* определение набора типов */
  double val;
  symrec *tptr;
}
%token <val> NUM      /* определение лексемы NUM и её типа */
```

Вы можете связать строковую лексему с именем типа лексемы, записав строку в конце объявления %token, объявляющего это имя. Например:

%token arrow "=>"

Например, грамматика языка C может задавать такие имена и соответствующие строковые лексемы:

```
%token  <operator>  OR      "||"
%token  <operator>  LE 134  "<="
%left  OR  "<="
```

После того, как вы сопоставили друг с другом строку и тип лексемы, они становятся взаимозаменяемы в дальнейших объявлениях или правилах грамматики. Функция yylex может использовать имя лексемы или строку для получения числового кода типа лексемы (см. раздел 5.2.1 Соглашения о вызове yylex).

4.7.2 Приоритет операций

Для одновременного объявления лексемы и задания её приоритета и ассоциативности используйте объявления %left, %right или %nonassoc. Они называются *объявлениями приоритета*. См. раздел 6.3 Приоритет операций, для общей информации о приоритете операций.

Синтаксис объявления приоритета тот же, что и %token: или

%left *символы*...

или

%left <*тип*> *символы*...

В самом деле, любое из этих определений служит тем же целям, что и %token. Но кроме того, они задают ассоциативность и относительный приоритет всех *символов*:

- Ассоциативность операции *op* определяет разбор повторяющихся операторов: будут ли при разборе `*x op y op z*` вначале сгруппированы *x* и *y*, или же *y* и *z*. %left задаёт левую ассоциативность (вначале группируются *x* и *y*), а %right -- правую (наоборот, *y* и *z*). %nonassoc задаёт неассоциативную операцию, т.е. `*x op y op z*` рассматривается как синтаксическая ошибка.
- Приоритет операции определяет, как она соотносится с другими операциями. Все лексемы, объявленные в одном объявлении приоритета, имеют одинаковый приоритет, и разбираются вместе в соответствии с их ассоциативностью. При объединении двух лексем, объявленных в разных объявлениях приоритета, объявленная позднее имеет более высокий приори- тет и группируется

раньше.

4.7.3 Набор типов значений

Объявление %union задаёт весь набор возможных типов данных семантических значений. За ключевым словом %union следуют фигурные скобки, содержимое которых имеет тот же вид, что и содержимое структуры union C.

Например:

```
%union {
  double val;
  symrec *tptr;
}
```

Это объявление говорит о том, что есть два альтернативных типа: double и symrec *. Им даны имена val и tptr, эти имена используются в объявлениях %token и %type для указать одного из этих типов для терминального или нетерминального символа (см. раздел 4.7.4 Нетерминальные символы).

Имейте в виду, что, в отличие от объявления union в C, вы не ставите точку с запятой после закрывающей фигурной скобки.

4.7.4 Нетерминальные символы

Если вы используете %union для задания множества типов значений, вы должны объявить тип значения каждого нетерминального символа, семантическое значение которого используется. Это делает объявление %type:

%type <*тип*> *нетерминал*...

Здесь *нетерминал* -- имя нетерминального символа, а *тип* -- имя, данное в %union желаемой альтернативе (см. раздел 4.7.3 Набор типов значений). Вы можете задать в одном объявлении %type любое количество нетерминальных символов, если у них одинаковый тип значения. Для разделения между собой имён символов

используйте пробелы.

Вы можете также объявить тип значения терминального символа. Для этого используйте ту же конструкцию <*тип*> в объявлении терминального символа. Все варианты объявления лексемы допускают наличие <*тип*>.

4.7.5 Подавление сообщений о конфликтах

В норме Bison сообщает, если в грамматике есть какие-либо конфликты (см. раздел 6.2 Конфликты сдвиг/свёртка), но большинство реальных грамматик содержат безвредные конфликты сдвиг/свёртка, разрешаемые предсказуемым образом, и которые сложно исключить. Желательно подавить сообщения об этих конфликтах, пока их число не изменяется. Вы можете сделать это объявлением %expect.

Это объявление выглядит так:

%expect *n*

Здесь *n* -- десятичное целое число. Это объявление говорит, что не нужно выдавать предупреждений, если грамматика содержит *n* конфликтов сдвиг/свёртка и не содержит конфликтов свёртка/свёртка. Если конфликтов меньше или больше, или если есть конфликты свёртка/свёртка, вместо обычного предупреждения выдаётся сообщение об ошибке.

В общем случае использование %expect включает следующие этапы:

- Скомпилируйте вашу грамматику без %expect. Используйте параметр `-v' чтобы получить подробный список конфликтов и мест их возникновения. Bison также выведет число конфликтов.
- Проверьте каждый из конфликтов, чтобы удостовериться в том, что решение Bison по умолчанию -- это то, чего вы на самом деле хотите. Если нет, перепишите грамматику и вернитесь к началу.
- Добавьте объявление %expect, взяв число *n* равным вы-

веденному Bison.

Теперь Bison перестанет раздражать вас сообщениями о проверенных вами конфликтах, но вновь начнёт выдавать сообщения, если изменения в грамматике повлекут появление новых конфликтов.

4.7.6 Начальный символ

По умолчанию Bison полагает начальным символом грамматики первый нетерминал, заданный в секции определения грамматики. Программист может переопределить его объявлением %start, как здесь:

%start *символ*

4.7.7 Чистый (повторно входимый) анализатор

Повторно входимая программа -- это программа, которая не меняется в процессе выполнения. Другими словами, она полностью состоит из *чистого* кода (кода только для чтения). Повторная входимость важна всегда, когда возможно асинхронное выполнение, например, не повторно входимая программа может быть ненадёжной при вызове её из обработчика сигнала. В системах с несколькими потоками управления, не повторно входимая программа может быть вызвана только внутри критического участка.

В норме Bison генерирует не повторно входимый анализатор. Это подходит в большинстве случаев, и даёт совместимость с YACC (стандартные интерфейсы YACC не повторно входимы по своей природе, потому что они используют для взаимодействия с yylex статически выделяемые переменные, включая yylval и yylloc).

В качестве альтернативы вы можете создать чистый, повторно входимый анализатор. Объявление Bison %pure_parser говорит, что вы хотите получить повторно входимый анализатор. Оно выглядит так:

%pure_parser

В результате переменные взаимодействия yylval и yylloc становятся локальными переменными yyparse и используются другие соглашения о вызове функции лексического анализатора yylex. См. раздел 5.2.4 Соглашения о вызове для чистых анализаторов, для прояснения деталей. Переменная yynerrs также становится локальной переменной yyparse (см. раздел 5.3 Функция сообщения об ошибках yyerror). Соглашения о вызове самой функции yyparse не изменяются.

Будет ли анализатор чистым, никак не влияет на правила грамматики. Вы можете создать как чистый, так и не повторно входимый анализатор из любой правильной грамматики.

4.7.8 Обзор объявлений Bison

Здесь приведено краткое изложение используемых при определении грамматики объявлений:

%union
: объявляет набор типов данных, которые могут иметь семантические значения (см. раздел 4.7.3 Набор типов значений).

%token
: Объявляет терминальный символ (имя типа лексемы) без указание приоритета или ассоциативности (см. раздел 4.7.1 Имена типов лексем).

%right
: Объявляет терминальный символ (имя типа лексемы), являющийся знаком правоассоциативной операции (см. раздел 4.7.2 Приоритет операций).

%left
: Объявляет терминальный символ (имя типа лексемы), являющийся знаком левоассоциативной операции (см. раздел 4.7.2 Приоритет операций).

%nonassoc
: Объявляет терминальный символ (имя типа лексемы), являющийся знаком неассоциативной операции (использование его там, где требуется ассоциативность, является синтаксической ошибкой) (см. раздел 4.7.2 Приоритет операций).

%type

Объявляет тип семантического значения нетерминального символа (см. раздел 4.7.4 Нетерминальные символы).

%start

Задаёт начальный символ грамматики (см. раздел 4.7.6 Начальный символ).

%expect

Объявляет ожидаемое число конфликтов сдвиг/свёртка (см. раздел 4.7.5 Подавление сообщений о конфликтах).

Для изменения поведения bison используйте следующие директивы:

%debug

В файле анализатора определяет макрос YYDEBUG как 1, если он ещё не определён, так что компилируются возможности отладки. См. раздел 9. Отладка вашего анализатора.

%defines

Создаёт дополнительный выходной файл, содержащий макроопределения имён типов лексем, определённых в грамматике, и типа семантического значения YYSTYPE, а также несколько объявлений внешних (extern) переменных. Если выходной файл анализатора называется `*имя*.c', то этот файл будет называться `*имя*.h'. Этот выходной файл необходим, если вы хотите поместить определение yylex в отдельный файл исходного кода, потому что функция yylex должна иметь возможность обращаться к кодам типов лексем и переменной yylval. См. раздел 5.2.2 Семантические значения лексем.

%file-prefix="*префикс*"

Задаёт префикс имён всех выходных файлов Bison. Имена выбираются таким образом, как если бы входной файл назывался `*префикс*.y'.

%locations

Создаёт код с обработкой положений (см. раздел 5.4 Специальные возможности, используемые в действиях). Этот режим включается всегда, когда грамматика использует специальные лексемы `@*n*', но если ваша грамматика их не использует, `%locations' позволит обеспе-

чить более аккуратный вывод сообщений об ошибках разбора.

%name-prefix="*префикс*"

Переименовывает используемые анализатором внешние символы так, что они начинаются с *префикс* вместо `yy'. Точный список переименовываемых символов: yyparse, yylex, yyerror, yynerrs, yylval, yychar и yydebug. Например, если вы используете `%name-prefix="c_"', их имена примут вид: c_parse, c_lex и т.д. См. раздел 4.8 Несколько анализаторов в одной программе.

%no-parser

Не включает никакого кода на C в файл анализатора, только создаёт таблицы. Файл анализатора будет содержать только директивы #define и объявления статических переменных. Этот параметр также указывает Bison записать в файл `*имя_файла*.act' код на C действий грамматики, окружённых фигурными скобками, пригодный для оператора switch.

%no-lines

Не создаёт никаких команд препроцессора #line в файле анализатора. Обычно Bison записывает эти команды в файл анализатора так, что компилятор C и отладчики будут связывать ошибки и объектный код с вашим исходным файлом (файлом грамматики). Эта директива заставит их связывать ошибки с файлом анализатора, рассматривая его как независимый исходный файл.

%output="*имя_файла*"

Задаёт *имя_файла* для файла анализатора.

%pure-parser

Запрашивает чистую (повторно входимую) программу анализатора (см. раздел 4.7.7 Чистый (повторно входимый) анализатор).

%token_table

Создать массив имён лексем в файле анализатора. Имя массива -- yytname, yytname[*i*] -- имя лексемы с внутренним кодом Bison *i*. Первые три элемента yytname всегда "$", "error и "$illegal", после них идут символы, определённые в файле грамматики. Для однолитерных и строковых лексем имя в таблице включает одинарные или двойные кавычки: например, "'+'" -- однолитерная константа, а "\"<=\"" -- строковая лексема. Все литеры строковой лексемы точно в том же виде содержатся в строке,

находящейся в таблице, даже двойные кавычки не экранируются. Например, Если лексема состоит из трёх литер `*"*', её строка в yytname содержит `"*"*"'. (На C это будет записываться "\"*\"*\""). Когда вы задаёте %token_table, Bison также создаёт макроопределения YYNTOKENS, YYNNTS, YYNRULES, и YYNSTATES:

YYNTOKENS

Самый большой номер лексемы плюс один.

YYNNTS

Число нетерминальных символов.

YYNRULES

Число правил грамматики.

YYNSTATES

Число состояний анализатора (см. раздел 6.5 Состояния анализатора).

%verbose

Создаёт дополнительный выходной файл, содержащий подробные описания состояний анализатора, и какие действия в этом состоянии выполняются при предпросмотре каждого типа лексем. Этот файл также описывает все конфликты, как разрешаемые приоритетом операций, так и не разрешаемые. Имя файла получается заменой `.tab.c' или `.c' в имени выходного файла анализатора на `.output'. Поэтому, если входной файл называется `foo.y', файл анализатора по умолчанию будет называться `foo.tab.c'. Вследствие этого, подробный выходной файл будет называться `foo.output'.

%yacc

%fixed-output-files

Как будто был задан параметр --yacc, т.е. имитирует Yacc, включая его соглашения об именах. См. раздел 10.1 Параметры Bison.

4.8 Несколько анализаторов в одной программе

Большая часть программ, использующих Bison, разбирает только один язык, и поэтому содержит только один анализатор Bison. Но что делать, если вы хотите одной программой анализировать более одного языка? Тогда вам нужно разрешить конфликты имён

между различными определениями yyparse, yylval и т.д.

Простой способ сделать это -- использовать параметр `-p *префикс*' (см. раздел 10. Вызов Bison). Тогда имена интерфейсных функций и переменных анализатора Bison будут начинаться с *префикс*, а не с yy. Это можно использовать, чтобы дать всем анализаторам различные не конфликтующие имена.

Точный список переименуемых символов: yyparse, yylex, yyerror, yynerrs, yylval, yychar и yydebug. Например, если вы используете `-p c', эти имена превратятся в cparse, clex и т.д.

Никакие другие переменные и макросы, связанные с Bison, не переименуются. Они не глобальны, и если одни и те же имена будут использоваться в разных анализаторах, конфликта не возникнет. Например, макрос YYSTYPE не будет переименован, но различное его определение в разных анализаторах не вызовет проблем (см. раздел 4.5.1 Типы данных семантических значений).

Параметр `-p' добавляет макроопределения в начало выходного файла анализатора, определяя yyparse как *префикс*parse и т.д. Это фактически подставляет во всём файле анализатора одно имя вместо другого.

5. ИНТЕРФЕЙС АНАЛИЗАТОРА НА C

Анализатор Bison -- это на самом деле функция на C, называющаяся yyparse. Здесь мы опишем соглашения по интерфейсу yyparse и других необходимых ей функций.

Имейте в виду, что для своих внутренних целей анализатор использует множество идентификаторов C, начинающихся с `yy' и `YY'. Если вы используете такой идентификатор (кроме тех, что описаны в настоящем руководстве) в действии или дополнительном коде на C в файле грамматики, вероятно, вы столкнётесь с неприятностями.

5.1 Функция анализатора yyparse

Вы вызываете функцию yyparse для запуска анализа. Эта функция читает лексемы, выполняет действия, и в конце концов завершает работу, когда встречает конец входного текста или сталкивается с невосстановимой синтаксической ошибкой. Вы можете также написать действие, которое укажет yyparse завершить работу немедленно, без продолжения чтения.

Если разбор завершён успешно (возврат вызван концом входного текста), yyparse возвращает значение 0.

Значение 1 возвращается, если разбор не удался (возврат вызван синтаксической ошибкой).

В действии вы можете потребовать немедленного возврата из yyparse, используя следующие макросы:

YYACCEPT
: Немедленный возврат со значением 0 (сообщение об удачном разборе).

YYABORT
: Немедленный возврат со значением 1 (сообщение об ошибке).

5.2 Функция лексического анализатора yylex

Функция *лексического анализатора* yylex распознаёт лексемы во входном потоке и передаёт их анализатору. Bison не создаёт эту функцию автоматически, вы должны написать её так, чтобы yyparse могла вызывать её. Эту функцию иногда называют лексическим сканером.

В простых программах yylex часто определяется в конце файла грамматики Bison. Если yylex определена в отдельном исходном файле, вам нужно сделать доступными там макроопределения типов лексем. Для этого используйте параметр `-d' при запуске Bison, чтобы он записал эти макроопределения в отдельный файл заголовка `*имя*.tab.h', который вы можете включить в другие исходные файлы, которым он нужен. См. раздел 10. Вызов Bison.

5.2.1 Соглашения о вызове yylex

Значение, возвращаемое yylex должно быть числовым кодом типа только что встреченной лексемы, или 0 для обозначени конца входного текста.

Если в правилах грамматики на лексему ссылаются по имени, это имя становится в файле анализатора макросом C, определением которого будет числовой код, соответствующий этому типу лексемы. Таким образом, yylex может использовать для обозначения типа это имя. См. раздел 4.2 Символы, терминальные и нетерминальные.

Если в правилах грамматики на лексему ссылаются с помощью однолитерной константы, числовой код этой литеры также является кодом типа лексемы. Таким образом yylex может просто вернуть код этой литеры. Нулевая литера не должна использоваться таким образом, потому что её код -- ноль, означающий конец входного текста.

Приведём пример, иллюстрирующий это:

```
int
yylex (void)
{
  ...
  if (c == EOF)     /* Проверка конца файла. */
    return 0;
  ...
  if (c == '+' || c == '-')
    return c;      /* Полагаем, что тип лексемы `+' -- '+'. */
  ...
  return INT;      /* Вернуть тип лексемы. */
  ...
}
```

Этот интерфейс разрабатывался так, чтобы выход утилиты lex мог быть без изменений использован как определение yylex.

Если грамматика использует строковые лексемы, есть два спосо-

ба, которыми yylex может определить коды их типов лексем:

- Если грамматика определяет символические имена лексем как псевдонимы строковых лексем, yylex может использовать эти символические имена как и все остальные. В этом случае использование строковых лексем в файле грамматики не окажет влияния на yylex.
- yylex может найти многолитерную лексему в таблице yytname. Индекс лексемы в этой таблице -- это код типа лексемы. Имя многолитерной лексемы записывается в yytname в виде: двойная кавычка, литеры лексемы, вторая двойная кавычка. Литеры лексемы никаким образом не экранируются, они дословно переносятся в содержимое строки в таблице. Приведём код поиска лексемы в yytname, полагая, что литеры лексемы находятся в массиве token_buffer.

```
  for (i = 0; i < YYNTOKENS; i++)
    {
      if (yytname[i] != 0
          && yytname[i][0] == '"'
          && strncmp (yytname[i] + 1, token_buffer,
                      strlen (token_buffer))
          && yytname[i][strlen (token_buffer) + 1] == '"'
          && yytname[i][strlen (token_buffer) + 2] == 0)
        break;
    }
```

Таблица yytname создаётся только если вы используете объявление %token_table. См. раздел 4.7.8 Обзор объявлений Bison.

5.2.2 Семантические значения лексем

В обычном (не повторно входимом) анализаторе семантические значения лексем должны помещаться в глобальную переменную yylval. Если вы используете единственный тип данных для семантических значений, yylval имеет этот тип. Так, если этот тип int (по умолчанию), вы можете написать в yylex:

```
...
yylval = value;  /* Поместить значение на вершину стека Bison. */
```

```
  return INT;      /* Вернуть тип лексемы. */
  ...
```

Если вы используете множественные типы данных, тип yylval -- объединение типов, полученное из объявления %union (см. раздел 4.7.3 Набор типов значений). Так, если вы сохраняете значение лексемы, вы должны использовать правильный элемент объединения. Если объявление %union выглядит так:

```
%union {
  int intval;
  double val;
  symrec *tptr;
}
```

то код в yylex может выглядеть так:

```
  ...
  yylval.intval = value; /* Поместить значение на вершину */
                         /* стека Bison. */
  return INT;            /* Вернуть тип лексемы. */
  ...
```

5.2.3 Позиции лексем в тексте

Если вы используете в действиях `@*n*'-свойства (см. раздел 4.6 Отслеживание положений) для отслеживания положений лексем и групп в тексте, ваша функция yylex должна предоставить эту информацию. Функция yyparse ожидает, что положение только что разобранной лексемы в тексте находится в глобальной переменной yylloc. Таким образом, yylex должна поместить в эту переменную правильные данные.

По умолчанию значение yylloc -- это структура, и вам нужно только проинициализировать её элементы, которые вы собираетесь использовать в действиях. Эти четыре элемента называются, first_line, first_column, last_line и last_column. Отметим, что использование этих свойств делает анализатор заметно более медленным.

Тип данных yylloc называется YYLTYPE.

5.2.4 Соглашения о вызове для чистых анализаторов

Если вы используете объявление Bison %pure_parser, требующее создания чистого, повторно входимого анализатора, глобальные переменные взаимодействия yylval и yylloc использовать нельзя (см. раздел 4.7.7 Чистый (повторно входимый) анализатор). В таких анализаторах эти две глобальные переменные замещаются указателями, передаваемыми в качестве аргументов функции yylex. Вы должны объявить их, как здесь показано, и передавать информацию назад, помещая её по этим указателям.

```
int
yylex (YYSTYPE *lvalp, YYLTYPE *llocp)
{
  ...
  *lvalp = value;  /* Поместить значение на вершину стека Bison.
*/
  return INT;      /* Вернуть тип лексемы.  */
  ...
}
```

Если файл грамматики не использует конструкции `@' для ссылок на позиции в тексте, тип YYLTYPE не будет определён. В этом случае опустите второй аргумент, yylex будет вызываться только с одним аргументом.

Если вы используете повторно входимый анализатор, вы можете (необязательно) передавать ему информацию о дополнительных параметрах повторно входимым способом. Для этого определите макрос YYPARSE_PARAM как имя переменной. Это изменит функцию yyparse чтобы она принимала один аргумент с этим именем типа void *.

При вызове yyparse передайте адрес объекта, приведя его к типу void *. Действия грамматики могут ссылаться на сожержимое объекта, приводя значение указателя обратно к его правильному типу, и затем разыменовывая его. Приведём пример. Напишите в анализаторе:

```
%{
struct parser_control
```

```
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
%}
```

Затем вызовите анализатор следующим образом:

```
struct parser_control
{
  int nastiness;
  int randomness;
};

...

{
  struct parser_control foo;
  ...  /* Поместить правильные данные в foo.  */
  value = yyparse ((void *) &foo);
  ...
}
```

В действиях грамматики используйте для обращения к данным выражения наподобие следующего:

```
((struct parser_control *) parm)->randomness
```

Если вы хотите передать данные дополнительных параметров функции yylex, определите макрос YYLEX_PARAM тем же способом, что и для YYPARSE_PARAM, как показано ниже:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};
```

```
#define YYPARSE_PARAM parm
#define YYLEX_PARAM parm
%}
```

Затем вам следует определить yylex, чтобы она принимала дополнительный аргумент -- значение parm (всего будет два или три аргумента, в зависимости от того, передаётся ли аргумент типа YYLTYPE). Вы можете объявить аргумент как указатель на правильный тип объекта, или же объявить его как void * и получать доступ к содержимому как показано выше.

Вы можете использовать `%pure_parser' и потребовать создания повторно входимого анализатора, не используя при этом YYPARSE_PARAM. Тогда вам следует вызывать yyparse без аргументов, как обычно.

5.3 Функция сообщения об ошибках yyerror

Анализатор Bison обнаруживает *ошибку разбора* или *синтаксическую ошибку* каждый раз, когда читает лексему, которая не может удовлетворять никакому синтаксическому правилу. Действие в грамматике может также явно сообщить об ошибке, используя макрос YYERROR (см. раздел 5.4 Специальные возможности, используемые в действиях).

Анализатор Bison рассчитывает сообщить об ошибке, вызывая функцию сообщения об ошибке yyerror, которую должны предоставить вы. Она вызывается функцией yyparse каждый раз при обнаружении синтаксической ошибки, и принимает один аргумент. В случае ошибки разбора это обычно строка "parse error".

Если вы определите макрос YYERROR_VERBOSE в секции объявлений Bison (см. раздел 4.1.2 Секция объявлений Bison), Bison будет давать более подробные и обстоятельные строки сообщений об ошибках, вместо обычного "parse error". Не имеет значения, какое определение вы используете для YYERROR_VERBOSE, только то, определили ли вы его.

Анализатор может обнаружить ещё один тип ошибки -- переполнение стека. Это происходит, когда входной текст содержит конструкции слишком большой глубины вложенности. Маловероятно,

что вы столкнётесь с этим, поскольку анализатор Bison расширяет свой стек автоматически до очень больших пределов. Но если переполнение всё же происходит, yyparse вызывает обычным образом yyerror, за исключением того, что аргументом будет строка "parser stack overflow".

Следующего определения достаточно для простых программ:

```
void
yyerror (char *s)
{
  fprintf (stderr, "%s\n", s);
}
```

После возвращения из yyerror в yyparse последняя попытается произвести восстановление после ошибки, если в грамматике вы написали подходящие правила восстановления после ошибок (см. раздел 7. Восстановление после ошибок). Если восстановление невозможно, yyparse немедленно завершит работу, вернув 1.

Переменная yynerrs содержит число обнаруженных до сих пор синтаксических ошибок. Обычно эта переменная глобальная, но если вы требуете создания чистого анализатора (см. раздел 4.7.7 Чистый (повторно входимый) анализатор), это локальная переменная, к которой могут иметь доступ только действия.

5.4 Специальные возможности, используемые в действиях

Ниже представлена таблица конструкций Bison, переменных и макросов, которые могут быть полезны в действиях.

`$$'
: Играет роль переменной, содержащей семантическое значение группы, собираемой текущим правилом. См. раздел 4.5.3 Действия.

`$*n*'
: Играет роль переменной, содержащей семантическое значение *n*-го компонента текущего правила. См. раздел 4.5.3 Действия.

`$<*тип_альт*>$'

Аналогична $$, но задаёт альтернативу *тип_альт* в объединении, заданном объявлением %union. См. раздел 4.5.4 Типы данных значений в действиях.

`$<*тип_альт*>*n*'

Аналогична *n*, но задаёт альтернативу *тип_альт* в объединении, заданном объявлением %union. См. раздел 4.5.4 Типы данных значений в действиях.

`YYABORT;'

Немедленно завершает работу yyparse, сообщая об ошибке. См. раздел 5.1 Функция анализатора yyparse.

`YYACCEPT;'

Немедленно завершает работу yyparse, сообщая об удачном разборе. См. раздел 5.1 Функция анализатора yyparse.

`YYBACKUP (*лексема*, *значение*);'

Отмена сдвига лексемы. Этот макрос допустим только в правилах, которые выполняют свёртку единственного значения, и только когда нет предпросмотренной лексемы. Он устанавливает для предпросмотренной лексемы тип *лексема* и семантическое значение *значение*. Затем он отбрасывает значение, которое должно быть свёрнуто по этому правилу. Если макрос используется, когда его применение недопустимо, как например, когда уже есть предпросмотренная лексема, он сообщает о синтаксической ошибке сообщением `cannot back up' и производит обычное восстановление после ошибки. В любом случае оставшаяся часть правила не выполняется.

`YYEMPTY'

Значение, помещаемое в yychar, когда там нет предпросмотренной лексемы.

`YYERROR;'

Немедленно вызывает синтаксическую ошибку. Этот оператор запускает восстановление после ошибки, как если бы ошибку обнаружил сам анализатор, и не выводит никакого сообщения. Если вы хотите вывести сообщение об ошибке, перед оператором `YYERROR;' вызовите явно yyerror. См. раздел 7. Восстановление после ошибок.

`YYRECOVERING'

Этот макрос заменяет выражение, имеющее значение 1 когда анализатор выполняет восстановление после синтаксической ошибки, и 0 всё остальное время. См. раздел 7. Восстановление после ошибок.

`yychar'

Переменная, содержащая текущую предпросмотренную лексему (в чистом анализаторе это на самом деле локальная для yyparse переменная). Когда предпросмотренной лексемы нет, в неё помещается значение YYEMPTY. См. раздел 6.1 Предпросмотренные лексемы.

`yyclearin;'

Отбросить текущую предпросмотренную лексему. Это полезно, прежде всего, в правилах обработки ошибок. См. раздел 7. Восстановление после ошибок.

`yyerrok;'

Немедленно взобновляет создание сообщений об ошибках для последующих синтаксических ошибок. Это полезно, прежде всего, в правилах обработки ошибок. См. раздел 7. Восстановление после ошибок.

`@$'

Играет роль структурной переменной, содержащей информацию о позиции в тексте группы, создаваемой текущим правилом. См. раздел 4.6 Отслеживание положений.

`@*n*'

Играет роль структурной переменной, содержащей информацию о позиции в тексте *n*-го компонента текущего правила. См. раздел 4.6 Отслеживание положений.

6. АЛГОРИТМ АНАЛИЗАТОРА BISON

Когда Bison читает лексемы, он помещает их в стек вместе с их семантическими значениями. Стек называется *стеком анализатора*. Помещение лексемы в стек традиционно называется *сдвигом*.

Например, предположим, что инфиксный калькулятор прочёл `1 + 5 *', а далее во входном тексте следует `3'. Стек будет содержать четыре элемента, по одному на каждую лексему, для которой был выполнен сдвиг.

Но стек не всегда содержит по элементу для каждой прочитанной лексемы. Когда последние *n* лексем и групп, для которых был выполнен сдвиг, соответствуют компонентам правила грамматики, они могут быть объединены в соответствии с этим правилом. Это называется *свёрткой*. Эти лексемы и группы заменяются в стеке одной группой, символ которой является результатом (левой частью) этого правила. Выполнение действия правила -- часть процесса свёртки, потому что именно тогда вычисляется семантическое значение полученной группы.

Например, если стек анализатора инфиксного калькулятора содержит:

1 + 5 * 3

и следующая лексема -- это литера новой строки, то последние три элемента могут быть свёрнуты до 15 правилом:

expr: expr '*' expr;

После этого стек будет содержать только три элемента:

1 + 15

В этот момент можно произвести ещё одну свёртку, получая единственное значение 16. Затем можно выполнить сдвиг для лексемы новой строки.

Анализатор пытается сдвигами и свёртками свернуть весь вход-

ной текст до единственной группы, символом которой является начальный символ грамматики (см. раздел 2.1 Языки и контекстно-свободные грамматики).

Этот тип анализаторов известен в литературе как анализатор снизу вверх.

6.1 Предпросмотренные лексемы

Анализатор Bison *не всегда* выполняет свёртку немедленно, как только последние *n* лексем и групп соответствуют правилу. Причина этого в том, что такая простая стратегия не подходит для обработки большинства языков. Вместо этого, когда свёртка возможна, анализатор иногда "смотрит вперёд" на следующую лексему, чтобы решить, что делать.

Когда лексема прочитана, сдвиг не выполняется сразу же, сначала она становится *предпросмотренной лексемой*, которая не помещена в стек. Теперь анализатор может выполнить одну или более свёрток лексем и групп в стеке, в то время, как предпросмотренная лексема остаётся за его пределами. Когда больше свёрток выполнять не следует, предпросмотренная лексема сдвигается в стек. Это не означает, что произведены все возможные свёртки, это зависит от типа предпросмотренной лексемы, некоторые правила могут предпочесть отложить своё применение.

Вот простой случай, когда нужен предпросмотр. Эти три правила определяют выражения, содержащие бинарную операцию сложения и постфиксную унарную операцию факториала (`!'), а также допускают группировку скобками.

```
expr:     term '+' expr
        | term
        ;

term:     '(' expr ')'
        | term '!'
        | NUMBER
        ;
```

Предположим, что прочитаны и сдвинуты лексемы `1 + 2', что

следует делать дальше? Если далее следует лексемы `)', то первые три лексемы должны быть свёрнуты до expr. Это единственный правильный вариант, поскольку сдвиг `)' создаст последовательность символов term ')', но ни одно правило этого не допускает.

Если же далее следует лексема `!', она должна быть немедленно сдвинута, чтобы `2 !' можно было свернуть до term. Если бы вместо этого анализатор выполнил свёртку ранее сдвига, `1 + 2' стало бы expr. Тогда было бы невозможно сдвинуть `!', потому что сдвиг создал бы в стеке последовательность символов expr '!'. Ни одно правило не допускает такой последовательности.

Текущая предпросмотренная лексема находится в переменной yychar. См. раздел 5.4 Специальные возможности, используемые в действиях.

6.2 Конфликты сдвиг/свёртка

Предположим, мы разбираем язык, имеющий операторы if-then и if-then-else, с помощью такой пары правил:

```
if_stmt:
        IF expr THEN stmt
      | IF expr THEN stmt ELSE stmt
      ;
```

Здесь мы предполагаем, что IF, THEN и ELSE -- терминальные символы лексем конкретных ключевых слов.

Когда лексемы ELSE прочитана и стала предпросмотренной лексемой, содержимое стека (предполагая, что входной текст правилен), как раз подходит для свёртки по первому правилу. Но законно также и сдвинуть ELSE, потому что это приведёт к последующей свёртке по второму правилу.

Такая ситуация, когда допустимы как сдвиг, так и свёртка, называется *конфликтом сдвиг/свёртка*. Bison разработан так, что он разрешает эти конфликты, выбирая сдвиг, за ислючением случаев, когда объявления приоритета операций указывают обратное.

Чтобы понять причину этого решения, сравним его с другой возможной альтернативой.

Поскольку анализатор предпочитает сдвинуть ELSE, в результате предложение else будет относиться к самому внутреннем оператору if, что сделает два следующих входных текста эквивалентными:

if x then if y then win (); else lose;

if x then do; if y then win (); else lose; end;

Но если анализатор выбирал при возможности свёртку, а не сдвиг, в результате предложение else относилось бы к самому внешнему оператору if, что сделает эквивалентными следующие два входных текста:

if x then if y then win (); else lose;

if x then do; if y then win (); end; else lose;

Конфликт возникает потому что грамматика написана неоднозначно: правомерен любой способ разбор простого вложенного оператора if. Общепринятое соглашение состоит в том, что такие неоднозначности разрешаются отнесением предложения else к самому внутреннему оператору if. Именно это делает Bison, выбирая сдвиг, а не свёртку (идеальным было бы написать однозначную грамматику, но в данном случае это сделать очень трудно). Эта конкретная неоднозначность впервые встретилась в спецификации Algol 60 и называется неоднозначностью "кочующего else".

Чтобы избежать предупреждений Bison о предсказуемых, законных конфликтах сдвиг/свёртка используйте объявление %expect *n*. Тогда предупреждений не последует до тех пор, пока число конфликтов сдвиг/свёртка в точности равно *n*. См. раздел 4.7.5 Подавление сообщений о конфликтах.

Определение if_stmt выше лишь несёт ответственность за возникновение конфликта, но на самом деле конфликт не возникнет без дополнительных правил. Вот полный фходной файл Bison, дей-

ствительно обнаруживающий конфликт:

```
%token IF THEN ELSE variable
%%
stmt:     expr
        | if_stmt
        ;

if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;

expr:     variable
        ;
```

6.3 Приоритет операций

Другая ситуация, когда возникают конфликты сдвиг/свёртка, -- это в арифметических выражениях. Здесь сдвиг -- не всегда предпочтительное решение, объявления Bison приоритета операций позволят вам указать, когда выполняется сдвиг, а когда свёртка.

6.3.1 Когда необходим приоритет

Рассмотрим следующий фрагмент неоднозначной грамматики (неоднозначной, потому что входной текст `1 - 2 * 3' может юбыть разобран двумя разными способами):

```
expr:     expr '-' expr
        | expr '*' expr
        | expr '<' expr
        | '(' expr ')'
        ...
        ;
```

Предположим, что анализатор рассмотрел лексемы `1', `-' и `2'. Должен ли он выполнить свёртку по правилу для операции вычитания? Это зависит от следующей лексемы. Конечно, если далее следует лексема `)', мы должны выполнить свёртку, сдвиг будет

неверным решением, потому что ни одно правило не может свернуть ни последовательность лексем `- 2 )', ни что-либо, начинающееся с неё. Но если следующая лексема -- `*' или `<', у нас есть выбор: как сдвиг, так и свёртка позволят удачно завершить разбор, но с разными результатами.

Чтобы решить, что должен делать Bison, мы должны рассмотреть результаты. Если сдвинуть следующую лексему знака операции *op*, она должна быть свёрнута первой, чтобы дать возможность свернуть разность. Результатом (на самом деле) будет `1 - (2 *op* 3'. С другой стороны, если вычитание свернуть до сдвига *op*, результатом будет `(1 - 2) *op* 3'. Поэтому ясно, что выбор сдвига или свёртки должен зависеть от относительного приоритета операций `-' и *op*: `*' должна быть сначала сдвинута, а `<' -- нет.

А что можно сказать о таком входном тексте, как `1 - 2 - 5', должен ли он означать `(1 - 2) - 5' или `1 - (2 - 5'? Для большинства операций мы предпочтаем первый вариант, называемый *левой ассоциативностью*. Второй вариант -- *правая ассоциативность* желателен для операций присваивания. Выбор левой или правой ассоциативности -- это вопрос о том, будет анализатор выбирать сдвиг или свёртку, когда стек содержит `1 - 2' и предпросмотренная лексема -- `-'. Сдвиг даёт правую ассоциативность.

6.3.2 Задание приоритета операций

Bison позволяет вам указать требуемый выбор с помощью объявлений приоритета операций %left и %right. Каждое такое объявление содержит список лексем, являющихся операциями, приоритет и ассоциативность которых определяется объявлением. Объявление %left делает эти все эти операции левоассоциативными, а %right -- правоассоциативными. Третий вариант -- %nonassoc, устанавливающий, что появление одной операции два раза "подряд" будет синтаксической ошибкой.

Относительный приоритет различных операций управляется порядком, в котором они объявляются. Первое в файле объявление %left или %right задаёт операции самого низкого приоритета, следующее такое объявление задаёт операции, приоритет которых немного выше и т.д.

6.3.3 Примеры приоритета

В нашем примере нам нужны были следующие объявления:

```
%left '<'
%left '-'
%left '*'
```

В более законченном примере, поддерживающем также другие операции, нам следует объявлять их группами равного приоритета. Например, '+' объявляется вместе с '-':

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

(здесь NE и т.п. соответствуют операциям "не равно" и т.д. Мы полагаем, что эти лексемы содержат более одной литеры, и поэтому представляются именами, а не строками).

6.3.4 Как работает приоритет

Во-первых, объявление приоритета присваивает уровни приоритета объявленным терминальным символам. Во-вторых, присваиваются уровни приоритета определённым правилам: каждое правило получает приоритет, равный приоритету последнего терминального символа среди его компонентов (вы можете также явно задать приоритет правила. См. раздел 6.4 Контекстно-зависимый приоритет.)

Наконец, конфликт разрешается сравнением приоритета рассматриваемого правила с приоритетом предпросмотренной лексемы. Если приоритет лексемы выше, выполняется сдвиг, если ниже -- свёртка. Если приоритеты одинаковы, выбор делается исходя из ассоциативности этого уровня приоритета. Подробный выходной файл, создаваемый при использовании параметра `-v' (см. раздел 10. Вызов Bison), объясняет, как был разрешён каждый конфликт.

Не все правила и не все лексемы имеют приоритет. Если у правила или у предпросмотренной лексемы нет приоритета, по умолча-

нию производится сдвиг.

6.4 Контекстно-зависимый приоритет

Часто приоритет операции зависит от контекста. На первый взгляд, это звучит странно, но на самом деле это очень распространённый случай. Например, знак "минус" обычно имеет очень высокий приоритет как унарная операция, и несколько меньший (меньший, чем умножение) как бинарная операция.

Объявления приоритета Bison -- %left, %right и %nonassoc -- для данной лексемы могут использоваться только один раз, поэтому лексема имеет только один приоритет, объявленный таким образом. Чтобы воспользоваться контекстно-зависимым приоритетом, вам нужно использовать дополнительный механизм: модификатор правил %prec.

Модификатор %prec объявляет приоритет конкретного правила, указывая терминальный символ, приоритет которого следует использовать для этого правила. Этот символ не обязательно должен появляться в самом правиле. Синтаксис модификатора таков:

%prec *терминальный_символ*

Он ставится после всех компонентов правила. В результате правилу вместо приоритета, который должен быть присвоен ему обычным способом, присваивается приоритет символа *терминальный_символ*. Затем изменённый приоритет правила влияет на разрешение конфликтов с участием этого правила (см. раздел 6.3 Приоритет операций).

Вот как %prec решает проблему унарного минуса. Во-первых, объявим приоритет фиктивного терминального символа UMINUS. Лексем такого типа нет, этот символ служит только для определения его приоритета.

```
...
%left '+' '-'
%left '*'
%left UMINUS
```

Теперь приоритет UMINUS можно использовать в правилах:

```
exp:    ...
        | exp '-' exp
        ...
        | '-' exp %prec UMINUS
```

6.5 Состояния анализатора

Функция yyparse реализована с использованием машины с конечным числом состояний. Значения, помещаемые в стек анализатора -- не просто коды типов лексем, они представляют всю последовательность терминальных и нетерминальных символов на вершине стека или около неё. Текущее состояния содержит всю информацию о более раннем входном тексте, относящуюся к решению вопрос, что делать далее.

Каждый раз, когда читается предпросмотренная лексема, текущее состояние анализатора и тип предпросмотренной лексемы ищутся в таблице. Эта ячейка таблицы может говорить, например: "Выполнить сдвиг для предпросмотренной лексемы". В этом случае она также задаёт новое состояние анализатора, помещаемое на вершину стека. Или же она может говорить: "Выполнить свёртку, используя правило номер *n*". Это означает, что определённое количество лексем и групп снимаются с вершины стека и заменяются одной группой. Другими словами, из стека вынимаются несколько состояний, и в него помещается одно новое.

Есть ещё одна возможность -- таблица может сказать, что предпросмотренная лексема в текущем состоянии недопустима. Это запускает процесс обработки ошибок (см. раздел 7. Восстановление после ошибок).

6.6 Конфликты свёртка/свёртка

Конфликт свёртка/свёртка возникает, когда есть два правила или более, применимых к одной последовательности входного текста. Это обычно свидетельствует о серьёзной ошибке в грамматике.

Например, вот ошибочная попытка определить последователь-

ность нуля или более групп word:

```
sequence: /* пусто */
            { printf ("пустая последовательность\n"); }
        | maybeword
        | sequence word
            { printf ("добавлено слово %s\n", $2); }
        ;

maybeword: /* пусто */
            { printf ("maybeword пусто\n"); }
        | word
            { printf ("одиночное слово %s\n", $1); }
        ;
```

Ошибка состоит в неоднозначности: есть более одного способа разобрать одиночное слово word в sequence. Оно может быть свёрнуто до maybeword, а затем до sequence по второму правилу. Или же "совсем ничто" может быть свёрнуто до sequence по первоум правилу, и объединено с word, используя третье правило для sequence.

Есть также более одного способа свёртки "совсем ничего" в sequence. Это может быть непосредственно сделано по первому правилу, или косвенно через maybeword, и затем применением второго правила.

Вы можете думать, что это безразлично, потому что это не влияет на то, является какой-либо входной текст правильным, или нет. Но это влияет на то, какие правила выполняются. Один порядок разбора выполнит действие второго правила, другой -- действия первого и третьего правила. В этом примере изменится вывод программы.

Bison разрешает конфликты свёртка/свёртка, выбирая правило, появляющееся в грамматике раньше, но полагаться на это очень рискованно. Каждый конфликт свёртка/свёртка должен быть изучен и обычно исключён. Вот правильный способ определения sequence:

```
sequence: /* пусто */
```

```
            { printf ("пустая последовательность\n"); }
        | sequence word
            { printf ("добавлено слово %s\n", $2); }
        ;
```

Вот ещё одна распространённая ошибка, приводящая к конфликтам свёртка/свёртка:

```
sequence: /* пусто */
        | sequence words
        | sequence redirects
        ;

words:    /* пусто */
        | words word
        ;

redirects:/* пусто */
        | redirects redirect
        ;
```

Здесь сделана попытка определить последовательность, которая может содержать либо группы word, либо redirects. Сами по себе определения sequence, words и redirects не содержат ошибок, но все вместе создают тонкую неоднозначность: даже пустая строка может быть разобрана бесконечным числом способов!

А именно: "совсем ничто" может быть words. Или оно может быть двумя words подряд, тремя, или любым другим количеством. Или оно может быть words, за которым следуют три redirects, а затем ещё одно words. И так далее.

Есть два способа исправить эти правила. Во-первых, сделать последовательность одноуровнневой:

```
sequence: /* пусто */
        | sequence word
        | sequence redirect
        ;
```

Во-вторых, запретить, чтобы words или redirects были пу-

стыми:

```
sequence: /* пусто */
        | sequence words
        | sequence redirects
        ;

words:     word
        | words word
        ;

redirects:redirect
        | redirects redirect
        ;
```

6.7 Загадочные конфликты свёртка/свёртка

Иногда могут возникать конфликты свёртка/свёртка, выглядящие неоправданными. Вот пример:

```
%token ID

%%
def:    param_spec return_spec ','
        ;
param_spec:
             type
        |    name_list ':' type
        ;
return_spec:
             type
        |    name ':' type
        ;
type:        ID
        ;
name:        ID
        ;
name_list:
             name
        |    name ',' name_list
        ;
```

Может показаться, что эта грамматика может быть разобрана с предпросмотром только на одну лексему: если прочитано param_spec, то ID является name, если далее следует запятая или двоеточие, и type, если следует ID. Другими словами, эта грамматика является LR(1)-грамматикой.

Тем не менее, Bison, как и большинство генераторов анализаторов, на самом деле может обрабатывать не все LR(1)-грамматики. В этой грамматике два контекста -- после ID в начале param_spec и контекст в начале return_spec достаточно похожи, чтобы Bison полагал их одинаковыми. Они оказываются похожими, потому что должен действовать один и тот же набор правил -- правило свёртки до name и правило свёртки до type. В этой стадии обработки Bison неспособен определить, что в разных контекстах правилам потребуются разные предпросмотренные лексемы, и поэтому создаёт одно состояние анализатора для них обоих. Объединение этих двух контекстов позднее вызовет конфликт. В терминологии синтаксических анализаторов, этот случай означает, что грамматика не является LALR(1).

В общем, лучше устранить недостатки, чем документировать их. Но этот конкретный недостаток по его природе трудно устранить -- генератор анализаторов, который может обрабатывать LR(1)-грамматики трудно написать, и он будет создавать очень большие анализаторы. На практике Bison более полезен в том виде, в котором он существует сейчас.

Когда возникает эта проблема, вы часто можете решить её, выявив два состояния анализатора, которые были смешаны, и добавляя что-нибудь, чтобы они выглядели по-разному. В вышеприведённом примере добавление одного правила к return_spec, как указано ниже, устранит проблему:

```
%token BOGUS
...
%%
...
return_spec:
          type
      |   name ':' type
      /* Это правило никогда не используется.  */
```

```
    |   ID BOGUS
    ;
```

Это решит проблему, поскольку появляется возможность использования ещё одного правила в контексте после ID в начале return_spec. Это правило не действует в соответствующем контекста в param_spec, поэтому эти два контекста получат разные состояния анализатора. До тех пор, пока лексема BOGUS не генерируется yylex, добавленное правило не изменит способ разбора ни одного реального входного текста.

В этом конкретном примере есть и другой способ решения проблемы: переписать правило для return_spec, чтобы оно использовало ID непосредственно, а не через name. Это также приведёт к тому, что два путающихся контекста будут иметь различные наборы действующих правил, потому что контекст return_spec задействует изменённое правило для return_spec, а не правило для name.

```
param_spec:
          type
    |     name_list ':' type
    ;
return_spec:
          type
    |     ID ':' type
    ;
```

6.8 Переполнение стека и как его избежать

Стек анализатора Bison может переполниться, если для слишком многих лексем выполнен сдвиг и не выполнена свёртка. Когда это происходит: функция анализатора yyparse завершает работу и возвращает ненулевое значение, лишь вызвав перед этим yyerror чтобы сообщить об ошибке.

Определяя макрос YYMAXDEPTH, вы можете указать, какой глубины может достичь стек анализатора перед тем, как произойдёт переполнение стека. Определите этот макрос целым значением. Это значение -- максимальное число лексем, для которых может быть выполнен сдвиг (и не выполнена свёртка) до того, как произойдёт переполнение. Оно должно быть константным вы-

ражением, значение которого известно во время компиляции.

Не обязательно выделять всё допустимое для стека пространство. Если вы задаёте большое значение YYMAXDEPTH, анализатора на самом деле сначала выделяет небольшой стек, и затем постепенно увеличивает его по мере необходимости. Это последовательное выделение происходит автоматически и незаметно. Поэтому вам не нужно делать YYMAXDEPTH болезненно малым только для того, чтобы сберечь память для обычных входных текстов, не нуждающихся в большом объёме стека.

Значение YYMAXDEPTH по умолчанию, если вы не определили его, составляет 10000.

Вы можете управлять размером стека, выделяемого при инициализации, определяя макрос YYINITDEPTH. Это значение также должно быть целочисленной константой, доступной во время компиляции. Значение по умолчанию составляет 200.

7. ВОССТАНОВЛЕНИЕ ПОСЛЕ ОШИБОК

Обычно программа, заканчивающая работу при ошибке разбора, неприемлема. Например, компилятору следует восстановиться настолько, чтобы разобрать остаток входного текста и проверить наличие в ней ошибок, калькулятор должен обработать следующее выражение.

В простом интерактивном анализаторе команд, где каждый вход -- одна строка, может быть достаточно разрешить yyparse вернуть 1 при ошибке, а вызывающей программе -- проигнорировать остаток входной строки, в которой она произошла (и затем вызвать yyparse снова). Но это не годится для компилятора, потому что в этом случае он забудет весь синтаксический контекст, приведший к ошибке. Синтаксическая ошибка глубоко внутри функции во входном тексте компилятора не должна заставлять компилятор рассматривать следующую строку как начало исходного файла.

Вы можете определять способ восстановления после синтаксической ошибки, составляя правила, распознающие специальную лексему error. Это терминальный символ, который всегда определён (вам не нужно объявлять его) и зарезервирован для обработки ошибок. Анализатор Bison генерирует лексему error каждый раз, когда обнаружена синтаксическая ошибка. Если вы предусмотрели правило для распознавания этой лексемы в текущем контексте, разбор может быть продолжен.

Например:

```
stmnts:  /* пустая строка */
        | stmnts '\n'
        | stmnts exp '\n'
        | stmnts error '\n'
```

Четвёртое правило в этом примере говорит, что ошибка, за которой следует переход на новую строку, является допустимым дополнением для любого stmnts.

Что случится, если синтаксическая ошибка будет обнаружена внутри exp? Правило восстановления после ошибки, если его ин-

терпретировать строго, применимо к последовательности, состоящей в точности из stmnts, error и перехода на новую строку. Если ошибка обнаружена внутри exp, вероятно, в стеке после последнего stmnts будут находиться некоторые дополнительные лексемы и подвыражения, а до литеры новой строки нужно будет прочитать ещё несколько лексем. Поэтому это правило обычным способом неприменимо.

Но Bison может принудительно привести ситуацию к правилу, отбрасывая часть семантического контекта и часть входного текста. Во-первых, он отбрасывает состояния и объекты в стеке до тех пор, пока не вернётся к правилу, в котором приемлема лексема error (это означает, что уже разобранные подвыражения будут отброшены, вплоть до последнего завершённого stmnts). В этот момент может быть выполнен сдвиг лексемы error. Потом, если нельзя выполнить сдвиг старой предпросмотренной лексемы, анализатор читает лексемы и отбрасывает их до тех пор, пока не найдёт подходящую лексему. В данном примере, Bison читает и отбрасывает входной текст, пока не обнаружит литеру новой строки, так что можно применить четвёртое правило.

Выбор правил грамматики для ошибок -- это выбор стратегии восстановления после ошибки. Простая и полезная стратегия -- при обнаружении ошибки просто пропустить остаток текущей входной строки или текущего оператора.

```
stmnt: error ';'  /* при ошибке пропускать, пока не будет считана ';' */
```

Также полезно восстанавливать до закрывающего ограничителя, соответствующего уже разобранному открывающему ограничителю. В противном случае закрывающий ограничитель, вероятно, оказался бы без пары, и вызвал новое, ложное сообщение об ошибке.

```
primary:  '(' expr ')'
        | '(' error ')'
        ...
        ;
```

Стратегии восстановления после ошибки неизбежно связаны с

догадками. Когда догадка неверна, одна синтаксическая ошибка часто приводит к появлению других. В вышеприведённом примере, правило восстановления после ошибки предполагает, что ошибка вызвана неправильным входным текстом внутри одного stmnt. Предположим, что вместо этого внутрь правильного stmnt вставлена точка с запятой. После того, как правило восстановления после ошибки произведёт восстановление от первой ошибки, сразу же будет обнаружена другая синтаксическая ошибка, поскольку текст, следующий за лишней точкой с запятой также не является верным stmnt.

Чтобы предотвратить поток сообщений об ошибках, анализатор не будет выводить сообщения об ошибках, произошедших вскоре после первой. Возобновит он их вывод только после того, как будет успешно произведён сдвиг трёх лексем подряд.

Имейте в виду, что правила, принимающие лексему error, могут содержать действия, так же как и любые другие правила.

Вы можете возобновить вывод сообщений об ошибках немедленно, используя в действиях макрос yyerrok. Если вы сделаете это в правиле действии правила обработки ошибки, сообщения об ошибках не будут подавляться. Этот макрос не требует аргументов, `yyerrok;' -- это правильный оператор C.

Сразу после обнаружения ошибки предыдущая предпросмотренная лексема анализируется заново. Если это невозможно, можно использовать макрос yyclearin для очистки этой лексемы. Напишите оператор `yyclearin;' в действии правила обработки ошибки.

Например, предположим, что при ошибке разбора вызывается подпрограмма обработки ошибки, продвигающаяся по входному потоку до некоторой точки, где разбор снова может быть начат. Предыдующая предпросмотренная лексема должна быть отброшена с помощью `yyclearin;'.

Макрос YYRECOVERING обозначает выражение, значение которого равно 1, если анализатор производит восстановление после синтаксической ошибки, и 0 всё остальное время. Значение 1 обозначает, что сообщеия о новых синтаксических ошибках в данный

момент подавляются.

8. ОБРАБОТКА КОНТЕКСТНЫХ ЗАВИСИМОСТЕЙ

Основной принцип Bison в том, чтобы сначала разбираются лексемы, а затем они группируются в более крупные синтаксические единицы. Во многих языках значение лексемы зависит от её контекста. Хотя это и нарушает принципы Bison, существуют определённые приёмы (известные как *кладжи*), которые дают вам возможность писать анализаторы Bison для таких языков.

(На самом деле, "кладж" -- это всякий приём, который решает поставленную задачу, но не является ни чистым, ни надёжным.)

8.1 Семантическая информация в типах лексем

Язык C содержит контекстную зависимость -- способ использования идентификаторов зависит от его текущего смысла. Например, рассмотрим:

foo (x);

Это выглядит как оператор вызова функции, но если foo -- имя определения типа, то на самом деле это объявление x. Как анализатор Bison для C может решить, как разбирать такой входной текст?

В GNU C используется метод, состоящий в том, чтобы иметь два разных типа лексем: IDENTIFIER и TYPENAME. Когда yylex обнаруживает идентификатор, она ищёт текущее объявление идентификтора чтобы решить, какой тип лексемы возвращать: TYPENAME, если идентификатор объявлен как определение типа, и IDENTIFIER в противном случае.

Правила грамматики могут затем выражать контекстную зависимость выбором типа лексемы при распознавании. IDENTIFIER допустима в выражении, а TYPENAME -- нет. Объявление может начинаться с TYPENAME, но не с IDENTIFIER. В контекстах, где смысл идентификатора *не имеет значения*, таких как в объявлениях, которые могут затенять имя определения типа, принимают-

ся как TYPENAME, так и IDENTIFIER -- для каждого из двух типов лексем есть своё правило.

Этот приём просто использовать, если решение, какой тип идентификаторов допустим, принимается в месте, близком к тому, где разбирается этот идентификатор. Но в C это не всегда так: C допускает, чтобы объявление переопределяло имя определения типа, при условии, что явный тип был задан ранее:

```
typedef int foo, bar, lose;
static foo (bar);        /* переопределить bar
                            как статическую переменную */
static int foo (lose);   /* переопределить foo
                            как функцию */
```

К сожалению, объявляемое имя отделено от самой объявляющей конструкции сложной синтаксической структурой -- "объявлятелем".

В результате часть анализатора Bison для C должна быть продублирована с изменением имён всех нетерминалов: одно для разбора объявления, в котором имя определения типа может быть переопределено, и другое для разбора объявлений, в которых это невозможно. Приведём часть такого дублирования, действия в котором опущены для краткости:

```
initdcl:
          declarator maybeasm '='
          init
        | declarator maybeasm
        ;

notype_initdcl:
          notype_declarator maybeasm '='
          init
        | notype_declarator maybeasm
        ;
```

Здесь initdcl может переопределить имя определения типа, а notype_initdcl -- нет. Та же разница между declarator и

notype_declarator.

Есть нечто общее между этим приёмом и лексической увязкой (описанной ниже), в том, что информация, изменяющая ход лексического анализа, изменяется во время разбора другим частями программы. Различие в том, что здесь эта информация глобальна, и используется в программе для других целей. Истинная лексическая увязка имеет флаг специального назначения, управляемый синтаксическим контекстом.

8.2 Лексическая увязка

Одним из способов обработки контекстной зависимости является *лексическая увязка* -- флаг, устанавливаемый действиями Bison, предназначенный для изменения способа разбора лексем.

Например, предположим, что у нас есть язык, смутно похожий на C, но со специальной конструкцией `hex (*шестнадцатеричное_выражение*)'. После ключевого слова hex идёт выражение в скобках, в котором все целые числа записаны в шестнадцатеричной системе счисления. В частности, при появлении в этом контекта лексема `a1b' должна рассматриваться как целое, а не как идентификатор. Вот как вы можете это сделать:

```
%{
int hexflag;
%}
%%
...
expr:   IDENTIFIER
        | constant
        | HEX '('
                { hexflag = 1; }
          expr ')'
                { hexflag = 0;
                  $$ = $4; }
        | expr '+' expr
                { $$ = make_sum ($1, $3); }
        ...
        ;
```

```
constant:
        INTEGER
      | STRING
      ;
```

Здесь мы полагаем, что yylex проверяет значение hexflag, когда оно ненулевое, все числа рассматриваются как шестнадцатеричные, и лексемы, начинающиеся в букв, разбираются как целые, если это возможно.

Объявление hexflag, показанное в секции объявлений C файла анализатора, необходимо чтобы оно было доступно для действий (см. раздел 4.1.1 Секция объявлений C). Вы также должны написать код yylex так, чтобы анализ подчинялся этому флагу.

8.3 Лексическая увязка и восстановление после ошибок

Лексическая увязка предъявляет строгие требования ко всем имеющимся у вас правилам восстановления после ошибок. См. раздел 7. Восстановление после ошибок.

Причина этого в том, что целью правил восстановления после ошибок является прервать анализ одной конструкции и возобновить анализ более крупных конструкций. Например, в языках типа C типичное правило восстановления после ошибки пропускает лексемы до следующей точки с запятой, и затем начинает разбор нового оператора, как здесь:

```
stmt:   expr ';'
      | IF '(' expr ')' stmt { ... }
      ...
      error ';'
            { hexflag = 0; }
      ;
```

Если внутри конструкции `hex (*выражение*)' есть синтаксическая ошибка, будет применено это правило обработки ошибок, и тогда действия для законченного `hex (*выражение*)' никогда не будут выполнены. Поэтому hexflag останется установленным для всего остального входного текста или до следующего ключевого

слова hex, в результате чего идентификаторы будут ошибочно интерпретироваться как целые числа.

Чтобы избежать этой проблемы, правило восстановления после ошибки само сбрасывает hexflag.

Также могут существовать правила восстановления после ошибок, работающие внутри выражений. Например, может быть правило, применяющееся внутри скобок и пропускающее всё до закрывающей скобки:

```
expr:   ...
        | '(' expr ')'
                { $$ = $2; }
        | '(' error ')'
        ...
```

Если это правило действует внутри конструкции hex, оно не приводит к выходу из этой конструкии (поскольку оно применяется к самому внутреннем уровню скобок внутри конструкции). Поэтому не нужно очищать флаг, оставшаяся часть конструкции hex должна быть разобрана пока флаг всё ещё действует.

Что если существует правило восстановления после ошибок, которое может прервать или не прерывать разбор конструкции hex, в зависимости от обстоятельств? Способа написать правило, определяющее, был ли прерван разбор конструкции hex, нет. Поэтому, если вы используете лексиечскую увязку, лучше быть уверенным в том, что ваши правила восстановления после ошибок не таковы. Каждое правило должно быть таким, что вы можете быть уверены в том, что оно всегда будет или никогда не будет сбрасывать флаг.

9. ОТЛАДКА ВАШЕГО АНАЛИЗАТОРА

Если грамматика Bison компилируется правильно, но при запуске делает не то, чего вы хотите, помочь вам выяснить, почему это происходит, может средство трассировки анализатора yydebug.

Чтобы включить компиляцию возможностей трассировки, вы должны определить макрос YYDEBUG как ненулевое значение при компиляции анализатора. Вы можете использовать параметр компилятора `-DYYDEBUG=1' или поместить `#define YYDEBUG 1' в секцию объявлений C файла грамматики (см. раздел 4.1.1 Секция объявлений C). Вместо этого можно использовать параметр `-t' при запуске Bison (см. раздел 10. Вызов Bison) или объявление %debug (см. раздел 4.7.8 Обзор объявлений Bison). Мы полагаем, что вы всегда будете определять YYDEBUG, так что отладка всегда будет возможна.

Средство трассировки выводит собщения, используя макровызовы вида YYFPRINTF (stderr, *формат*, *аргументы*, где *формат* и *аргументы* -- обычные формат и аргументы функции printf. Если вы определяете YYDEBUG как ненулевое значение, но не определяете YYFPRINTF, автоматически включается <stdio.h> и YYFPRINTF определяется как fprintf.

После того, как вы скомпилировали программу с использованием средств трассировки, чтобы потребовать выполнения трассировки, нужно поместить ненулевое значение в переменную yydebug. Вы можете сделать это, заставив это делать код на C (возможно, в функции main), или изменить это значение отладчиком C.

Каждый шаг, предпринимаемый анализатором, когда yydebug не равно нулю, даёт одну или две строки информации о трассировке, выдаваемой на stderr. Сообщения трассировки говорят о следующем:

- При каждом вызове yylex: лексема какого вида прочитана.
- При каждом сдвиге: глубина и всё содержимое стека состояний (см. раздел 6.5 Состояния анализатора).
- При каждой свёртке: по какому правилу произведена свёртка, и полное содержимое стека после

этого.

Для осмысления этой информации полезно обратиться к файлу листинга, выдаваемому параметром Bison `-v' (см. раздел 10. Вызов Bison). Этот файл показывает смысл каждого состояния в терминах позиций в различных правилах, а также, что будет происходить в каждом состоянии при каждой возможной входной лексеме. Читая последовательные сообщения трассировки, вы можете видеть, что анализатор функционирует в соответствии с его спецификацией в файле листинга. В конце концов вы дойдёте до места, где происходит что-либо нежелательное, и увидите, какие части грамматики несут за это ответственность.

Файл анализатора -- это программа на C, и вы можете использовать отладчики C, но объяснить, что она делает непросто. Функция анализатора -- это интерпретатор машины с конечным числом состояний, и за пределами действий она выполняет один и тот же код снова и снова. В каком месте грамматики она работает, показывают только значения переменных.

Отладочная информация обычно содержит тип каждой прочитанной лексемы, но не её семантическое значение. Вы можете также определить макрос YYPRINT, чтобы предоставить способ вывода значения. Если вы определяете YYPRINT, он должен принимать три аргумента. Анализатор будет передавать ему стандартный поток ввода/вывода, числовой код типа лексемы и значение лексемы (из yylval).

Приведём пример YYPRINT, подходящего для многофункционального калькулятора (см. раздел 3.5.1 Объявления mfcalc):

```
#define YYPRINT(file, type, value)   yyprint (file, type, value)

static void
yyprint (FILE *file, int type, YYSTYPE value)
{
  if (type == VAR)
    fprintf (file, " %s", value.tptr->name);
  else if (type == NUM)
    fprintf (file, " %d", value.val);
}
```

10. ВЫЗОВ BISON

Обычный способ вызова Bison следующий:

bison *вх_файл*

Здесь *вх_файл* -- имя файла грамматики, обычно заканчивающееся `.y'. Имя файла анализатора получается заменой `.y' на `.tab.c'. Так, `bison foo.y' даст `foo.tab.c', а `bison hack/foo.y' -- `hack/foo.tab.c'. Также возможно, если вы в своём файле грамматики пишете код на C++, а не на C, назвать его `foo.ypp' или `foo.y++'. Тогда выходные файлы будут иметь расширение, аналогичное расширению входного (соответственно `foo.tab.cpp' и `foo.tab.c++'). Эта возможность влияет на все параметры, обрабатывающие имена файлов, такие как `-o' или `-d'.

Например:

bison -d *вх_файл.yxx*

создаст `вх_файл.tab.cxx' и `вх_файл.tab.hxx', а

bison -d *вх_файл.y* -o *вых_файл.c++*

создаст `вых_файл.c++' и `вых_файл.h++'.

10.1 Параметры Bison

Bison поддерживает как традиционные однобуквенные параметры, так и длинные мнемонические имена параметров. Длинные имена параметров помечаются `--' вместо `-'. Сокращение имён параметров допустимо до такой длины, пока они остаются уникальными. Если длинный параметр принимает аргумент, как, например, `--file-prefix', соедините имя параметра и аргумент знаком `='.

Вот список параметров, которые можно использовать с Bison, в алфавитном порядке коротких параметров. Далее следует перекрёстный указатель в алфа- витном порядке длинных па-

раметров.

Режимы работы:

-h
--help
Вывести обзор параметров командной строки Bison и завершить работу.
-V
--version
Вывести версию Bison и завершить работу.
-y
--yacc
--fixed-output-files
Эквивалентно `-o y.tab.c'. Выходной файл анализатора называется `y.tab.c', другие выходные файлы: `y.output' и `y.tab.h'. Назначение этого параметра в имитации соглашений по именованию выходных файлов Yacc. Так, следующий скрипт shell может заменить Yacc:
bison -y $*

Подстройка анализатора:

-S *файл*
--skeleton=*файл*
Задать используемый скелет. Вероятно, вам не понадобится использовать этот параметр, если вы не разработчик Bison.
-t
--debug
В файле анализатора определить макрос YYDEBUG равным 1, если он ещё не определён, так что средства отладки будут скомпилированы. См. раздел 9. Отладка вашего анализатора.
--locations
Как будто было задано %locations. См. раздел 4.7.8 Обзор объявлений Bison.
-p *префикс*
--name-prefix=*префикс*
Как будто было задано %name-prefix="*префикс*". См. раздел 4.7.8 Обзор объявлений Bison.

-l
--no-lines
Не помещать в файл анализатора команд препроцессора #line. Обычно Bison помещает их в файл анализатора, так что компилятор и отладчики C будут связывать ошибки с вашим исходным файлом, файлом грамматики. Этот параметр заставить их связывать ошибки с файлом анализатора, рассматривая его как самостоятельный независимый исходный файл.
-n
--no-parser
Как будто было задано %no-parser. См. раздел 4.7.8 Обзор объявлений Bison.
-k
--token-table
Как будто было задано %token-table. См. раздел 4.7.8 Обзор объявлений Bison.

Регулирование вывода:

-d
--defines
Как будто было задано %defines, т.е. записывает дополнительный выходной файл, содержащий макроопределения имён типов лексем, определённых в грамматике, и типа семантических значений YYSTYPE как несколько объявлений переменных extern. См. раздел 4.7.8 Обзор объявлений Bison.
--defines=*файл_определений*
То же, что и предыдущий, но записывает в файл *файл_определений*.
-b *префикс_файла*
--file-prefix=*префикс*
Как будто было задано %file-prefix, т.е. задаёт префикс имён всех выходных файлов Bison. См. раздел 4.7.8 Обзор объявлений Bison.
-v
--verbose
Как будто было задано %verbose, т.е. записывает дополнительный выходной файл, содержащий подробное описание грамматики и анализатора. См. раздел 4.7.8 Обзор объявлений Bison.

-o *имя_файла*
--output=*имя_файла*
Именем файла анализатора будет *имя_файла*. Имена других выходных файлов получаются из *имя_файла* как описано для параметров `-v' и `-d'.

-g
Вывести определение VCG автомата LALR(1)-грамматики, вычисленного Bison. Если файл грамматики называется `foo.y', выходной файл VCG будет `foo.vcg'.

--graph=*файл_графа*
Поведение *--graph* то же, что и `-g'. Единственное различие в том, что у него есть необязательный аргумент -- имя выходного файла графа.

10.2 Переменные среды

Вот список переменных среды, оказывающих влияние на работу Bison.

`BISON_SIMPLE'
`BISON_HAIRY'
Многие анализаторы, генерируемые Bison, дословно копируются из файла `bison.simple'. Если Bison не может найти этот файл или если вы хотите указать Bison использовать другую его копию, установка переменной среды BISON_SIMPLE на путь к файлу заставит Bison использовать эту копию. Когда используется объявление `%semantic_parser', Bison копирует анализатор из файла `bison.hairy'. Расположение этого файла также может быть задано или переопределено аналогичным образом, переменной среды BISON_HAIRY.

10.3 Перекрёстный список параметров

Здесь приведён список параметров в алфавитном порядке длинных параметров, чтобы помочь вам найти соответствующий короткий параметр.

10.4 Ограничения на расширения под DOS

В системах DOS/Windows 9X возможность использования расширений имён выходных файлов, таких как `.tab.c', зависит от ис-

пользуемой файловой системы. Простая файловая система DOS содержит ограничение на длину имени файла, не допускает использование определённого множества незаконных литер и не допускает более одной точки в имени файла.

Порт bison DJGPP определяет во время выполнения, доступны ли длинные имена файлов (ДИФ). Поддержка ДИФ будет доступны в сеансе DOS под Windows 9X и последующих версий. Для Windows NT 4.0 для правильной поддержки ДИФ в сеансе DOS нужен специальный драйвер (`ntlfn09b.zip' или более поздний, доступный на любом зеркале simtelnet в каталоге /v2misc). Если поддержка ДИФ доступна, порт bison DJGPP будет использовать для выходных файлов стандартные расширения имён файлов POSIX. Если поддержка ДИФ недоступна, этот порт будет использовать расширения имён файлов для DOS.

В этой таблице указаны используемые расширения:

расширени ДИФ (Win9X)	расширение КИФ (просто DOS)
`.tab.c'	`_tab.c'
`.tab.h'	`_tab.h'
`.tab.cpp'	`_tab.cpp'
`.tab.hpp'	`_tab.hpp'
`.output'	`.out'
`.stype.h'	`.sth'
`.guard.c'	`.guc'

10.5 Вызов Bison под VMS

Синтаксис командной строки Bison под VMS является вариацией обычного синтаксиса команды Bison адаптированной для соответствия соглашениям VMS.

Чтобы найти VMS-эквивалент параметра Bison, возьмите длинную форму параметра, подставьте `/' вместо начального `--', `_' -- вместо каждого `-' в имени длинного параметра. Например, такой вызов под VMS:

bison /debug/name_prefix=bar foo.y

соответствует следующей команде под POSIX:

bison --debug --name-prefix=bar foo.y

Файловая система VMS не допускает такие имена файлов, как `foo.tab.c'. В вышеприведённом примере выходной файл будет называться `foo_tab.c'.

A. СИМВОЛЫ BISON

error

Имя лексемы, зарезервированной для обработки ошибок. Эта лексема может использоваться в правилах грамматики, чтобы позволить анализатору Bison распознавать ошибки в грамматике без остановки процесса разбора. В результате предложение, содержащее ошибку, может быть распознано как правильное. В случае ошибки разбора лексема error становится текущей предпросмотренной лексемой. Затем выполняются действия, соответствующие error и предпросмотренной лексемой становится та, которая первоначально вызвала ошибку. См. раздел 7. Восстановление после ошибок.

YYABORT

Макрос, работающий как если бы была обнаружена невосстановимая синтаксическая ошибка, немедленно завершая работу yyparse и возвращая 1. Функция сообщения об ошибке yyerror не вызывается. См. раздел 5.1 Функция анализатора yyparse.

YYACCEPT

Макрос, работающий как если бы было прочитано полное предложение языка, немедленно завершая работу yyparse и возвращая 0. См. раздел 5.1 Функция анализатора yyparse.

YYBACKUP

Макрос, отбрасывающий значение из стека анализатора, и "подделывающий" предпросмотренную лексему. См. раздел 5.4 Специальные возможности, используемые в действиях.

YYERROR

Макрос, работающий как если бы была обнаружена синтаксическая ошибка: вызывает yyerror и затем производит обычное восстановление после ошибки, если это возможно (см. раздел 7. Восстановление после ошибок), или (если невозможно) yyparse завершает работу и возвращает 1. См. раздел 7. Восстановление после ошибок.

YYERROR_VERBOSE

Макрос, который вы определяете директивой #define в секции объявлений Bison, требующий, чтобы при вызове yyerror строки сообщений об ошибках содержали подробную информацию.

YYINITDEPTH

Макрос для задания первоначального размера стека анализатора. См. раздел 6.8 Переполнение стека и как его избежать.

YYLEX_PARAM

Макрос для задания дополнительного аргумента (или списка дополнительных аргументов), которые функция yyparse передаёт yylex. См. раздел 5.2.4 Соглашения о вызове для чистых анализаторов.

YYLTYPE

Макрос типа данных yylloc, структура из четырёх элементов. См. раздел 4.6.1 Тип данных положений.

yyltype

Значение YYLTYPE по умолчанию.

YYMAXDEPTH

Макрос для задания максимального размера стека анализатора. См. раздел 6.8 Переполнение стека и как его избежать.

YYPARSE_PARAM

Макрос для задания имени параметра, который должен принимать yyparse. См. раздел 5.2.4 Соглашения о вызове для чистых анализаторов.

YYRECOVERING

Макрос, значение которого указывает, производит ли в данный момент анализатор восстановление после синтаксической ошибки. См. раздел 5.4 Специальные возможности, используемые в действиях.

YYSTACK_USE_ALLOCA

Макрос, используемый для управлени использованием alloca. Если определён как `0', анализатор при попытке расширить внутренние стеки будет использовать не alloca, а malloc. *Не определяйте* YYSTACK_USE_ALLOCA как что-либо другое.

YYSTYPE

Макрос типа данных семантических значений, по умолчанию int. См. раздел 4.5.1 Типы данных семантических значений.

yychar

Внешняя целочисленная переменная, содержащая целое значение текущей предпросмотренной лексемы (в чистом анализаторе это локальная переменная yyparse). Действия восстановления после ошибок могут проверять

значение этой переменной. См. раздел 5.4 Специальные возможности, используемые в действиях.

yyclearin

Макрос, используемый в действиях правил восстановления после ошибок. Очищает предыдущую предпросмотренную лексему. См. раздел 7. Восстановление после ошибок.

yydebug

Внешняя целочисленная переменная, по умолчанию установленная в ноль. Если yydebug присвоено ненулевое значение, анализатор будет выводить информацию о входных символах и собственных действиях. См. раздел 9. Отладка вашего анализатора.

yyerrok

Макрос, заставляющий анализатор немедленно вернуться в нормальный режим после ошибки разбора. См. раздел 7. Восстановление после ошибок.

yyerror

Предоставляемая пользователем функция, вызываемая yyparse в случае ошибки. Функция принимает один аргумент, указатель на строку, содержащую сообщение об ошибке. См. раздел 5.3 Функция сообщения об ошибках yyerror.

yylex

Предоставляемая пользователем функция лексического анализатора, вызываемая без аргументов и возвращающая следующую лексему. См. раздел 5.2 Функция лексического анализатора yylex.

yylval

Внешняя переменная, в которую yylex должна помещать семантическое значение, связанное с лексемой (в чистом анализаторе это локальная переменная yyparse, и её адрес передаётся yylex). См. раздел 5.2.2 Семантические значения лексем.

yylloc

Внешняя переменная, в которую yylex должна помещать номера строки и колонки, связанных с лексемой (в чистом анализаторе это локальная переменная yyparse, и её адрес передаётся yylex). Вы можете игнорировать эту переменную, если вы не используете возможности `@' в действиях грамматики. См. раздел 5.2.3 Позиции лексем в тексте.

yynerrs

Глобальная переменная, которую Bison увеличивает на 1 при каждой ошибке разбора (в чистом анализаторе это локальная переменная yyparse). См. раздел 5.3 Функция сообщения об ошибках yyerror.

yyparse

Функция анализатора, создаваемая Bison. Вызывайте эту функцию для запуска процесса разбора. См. раздел 5.1 Функция анализатора yyparse.

%debug

Готовит анализатор к отладке. См. раздел 4.7.8 Обзор объявлений Bison.

%defines

Объявление Bison для создания файла заголовка, нужного сканеру. См. раздел 4.7.8 Обзор объявлений Bison.

%file-prefix="*префикс*"

Объявление Bison, устанавливающее префикс выходных файлов. См. раздел 4.7.8 Обзор объявлений Bison.

%left

Объявление Bison, устанавливающее левую ассоциативность лексем(ы). См. раздел 4.7.2 Приоритет операций.

%name-prefix="*префикс*"

Объявление Bison, переименовывающее внешние символы. См. раздел 4.7.8 Обзор объявлений Bison.

%no-lines

Объявление Bison, отменяющее создание директив #line в файле анализатора. См. раздел 4.7.8 Обзор объявлений Bison.

%nonassoc

Объявление Bison, устанавливающее неассоциативность лексем(ы). См. раздел 4.7.2 Приоритет операций.

%output="*имя_файла*"

Объявление Bison, устанавливающее имя файла анализатора. См. раздел 4.7.8 Обзор объявлений Bison.

%prec

Объявление Bison, устанавливающее приоритет отдельного правила. См. раздел 6.4 Контекстно-зависимый приоритет.

%pure-parser

Объявление Bison, требующее создания чистого

(повторно входимого) анализатора. См. раздел 4.7.7 Чистый (повторно входимый) анализатор.

%right

Объявление Bison, устанавливающее правую ассоциативность лексем(ы). См. раздел 4.7.2 Приоритет операций.

%start

Объявление Bison, задающее начальный символ. См. раздел 4.7.6 Начальный символ.

%token

Объявление Bison, объявляющее лексем(ы) без задания приоритета. См. раздел 4.7.1 Имена типов лексем.

%token-table

Объявление Bison, включающее таблицу имён лексем в файл анализатора. См. раздел 4.7.8 Обзор объявлений Bison.

%type

Объявление Bison для нетерминалов. См. раздел 4.7.4 Нетерминальные символы.

%union

Объявление Bison, задающее несколько возможных типлв данных семантических значений. См. раздел 4.7.3 Набор типов значений.

Знаки пунктуации и ограничители, используемые во входном тексте Bison:

`%%'

Ограничитель, используемый для отделения секции правил грамматики от секции объявлений Bison или секции дополнительного кода на C. См. раздел 2.8 Обзор схемы грамматики Bison.

`%{ %}'

Весь код между `%{' и `%}' дословно копируется в выходной файл. Такой код образует секцию "объявлений C" входного файла. См. раздел 4.1 Структура грамматики Bison.

`/*...*/'

Ограничители комментариев, как в C.

`:'

Разграничивает результат правила и его компоненты. См. раздел 4.3 Синтаксис правил грамматики.

`;'

Завершает правило. См. раздел 4.3 Синтаксис правил грамматики.

`|'

Разграничивает альтернативные правила для одного результирующего нетерминала. См. раздел 4.3 Синтаксис правил грамматики.

В. ГЛОССАРИЙ

LALR(1)

Класс контекстно-свободных грамматик, которые может обрабатывать Bison (как и большинство других генераторов анализаторов), подмножество LR(1). См. раздел 6.7 Загадочные конфликты свёртка/свёртка.

LR(1)

Класс контекстно-свободных грамматики, в которых. чтобы разрешить неоднозначность разбора любого куска входного текста, необходимо не более одной предпросмотренной лексемы.

Анализатор

Функция, распознающая правильные последовательности языка путём анализа синтаксической структуры переданного ей лексическим анализатором множества лексем.

Бэкуса-Наура форма (BNF, БНФ)

Формальный метод определения контекстно-свободных грамматики. БНФ впервые была использована в сообщении о языке *ALGOL-60*. См. раздел 2.1 Языки и контекстно-свободные грамматики.

Входной поток

Непрерывный поток данных между устройствами или программами.

Группа

Грамматическая конструкция, которая (в общем случае) грамматически делима, например, "выражение" или "объявление" в C. См. раздел 2.1 Языки и контекстно-свободные грамматики.

Динамическое выделение

Выделение памяти, производимое во время выполнения программы, а не во время компиляции или при входе в функцию.

Инфиксная операция

Арифметическая операция, знак которой помещается между операндами, к которым она применяется.

Контекстно-свободные грамматики

Грамматики, определённые правилами, которые могут быть применены независимо Так, если есть правило, говорящее, что целое число может использоваться как выражение, целые числа допустимы *где угодно*, где допу-

стимы выражения. См. раздел 2.1 Языки и контекстно-свободные грамматики.

Левая ассоциативность

Левоассоциативные операции анализируются слева направо: `a+b+c' сначала вычисляет `a+b', а затем объединяет с `c'. См. раздел 6.3 Приоритет операций.

Левая рекурсия

Правило, результирующий символ которого является также символом его первого компонента. Например: `expseq1 : expseq1 ',' exp;'. См. раздел 4.4 Рекурсивные правила.

Лексема

Базовая, грамматически неделимая единица языка. В грамматике лексему описывает терминальный символ. На вход анализатора Bison поступает последовательность лексем от лексического анализатора. См. раздел 4.2 Символы, терминальные и нетерминальные.

Лексическая увязка

Флаг, устанавливаемый действиями правил грамматики и изменяющий способ разбора лексем. См. раздел 8.2 Лексическая увязка.

Лексический анализатор (сканер)

Функция, читающая входной поток и возвращающая лексемы по одной. См. раздел 5.2 Функция лексического анализатора yylex.

Машина с конечным числом состояний

"Машина", имеющая дискретные состояния, в одном из которых она находится в каждый момент времени. По мере обработки входного текста машина переходит из состояния в состояние, что определяется логикой машины. В случае анализатора входной файл -- это разбираемый текст, а состояния соответствуют различным стадиям в правилах грамматики. См. раздел 6. Алгоритм анализатора Bison.

Начальный символ

Нетерминальный символ, соответствующий целому законченному предложению в разбираемом языке. Начальный символ в спецификации языка обычно записывается первым нетерминальным символом. См. раздел 4.7.6 Начальный символ.

Нетерминальный символ

Символ грамматики, соответствующей грамматика-

ческой конструкции, которая может быть выражена правилами в терминах меньших конструкций. Другими словами, конструкция, не являющаяся лексемой. См. раздел 4.2 Символы, терминальные и нетерминальные.

Обратная польская нотация

Язык с постфиксными операциями.

Однолитерная лексема

Единственная лексема, распознаваемая и обрабатываемая сама как есть. См. раздел 2.2 От формальных правил к входному тексту Bison.

Ошибка разбора

Ошибка, произошедшая во время разбора входного потока из-за неверного синтаксиса. См. раздел 7. Восстановление после ошибок.

Постфиксная операция

Арифметическая операция, знак которой помещается после операндов, к которым она применяется.

Повторная входимость

Повторно входимая подпрограмма -- это подпрограмма, которая может быть вызвана любое число раз одновременно, различные вызовы которой не пересекаются. См. раздел 4.7.7 Чистый (повторно входимый) анализатор.

Правая рекурсия

Правило, результирующий символ которого является также символом его последнего компонента. Например: `expseq1 : exp ',' expseq1;'. См. раздел 4.4 Рекурсивные правила.

Предпросмотренная лексема

Лексема, уже прочитанная, но ещё не сдвинутая. См. раздел 6.1 Предпросмотренные лексемы.

Пустая строка

Аналогично пустому множеству в теории множеств, пустая строка -- это строка литер нулевой длины.

Разбор слева направо

Разбор предложения языка путём последовательного анализа его лексем слева направо. См. раздел 6. Алгоритм анализатора Bison.

Свёртка

Замена строки нетерминалов и/или терминалов одним нетерминалом, в соответсвии с правилом грамматики. См. раздел 6. Алгоритм анализатора Bison.

Сдвиг

Говорят, что анализатор производит сдвиг, когда он принимает решение анализировать дальнейший входной поток вместо того, чтобы выполнить немедленную свёртку по какому-либо уже распознанному правилу. См. раздел 6. Алгоритм анализатора Bison.

Семантика

В компьютерных языках семантика определяется действиями, предпринимаемыми для каждого текста на языке, т.е., смыслом каждого оператора. См. раздел 4.5 Определение семантики языка.

Строковая лексема

Лексема, состоящая из двух или более фиксированных литер. См. раздел 4.2 Символы, терминальные и нетерминальные.

Таблица символов

Структура данных, где во время разбора хранятся имена символов и связанные данные, чтобы дать возможность распознавания и использования существующей информации при повторном использовании символа. См. раздел 3.5 Многофункциональный калькулятор: mfcalc.

Терминальный символ

Символ грамматики, для которого в грамматике не существует правил, и поэтому грамматически неделимый. Представляемый им кусок текста является лексемой. См. раздел 2.1 Языки и контекстно-свободные грамматики.

Языковая конструкция

Одна из типичных используемых в языке схем. Например, одной из конструкций языка C является оператор if. См. раздел 2.1 Языки и контекстно-свободные грамматики.

УКАЗАТЕЛЬ

м

н

о

п