Введение

Обычно входная информация, читаемая программой, всегда обладает некоторой структурой; про любую программу, читающую входной поток мы можем сказать, что она задает некоторый входной язык. Входной язык может быть либо сложным, как язык программирования, либо простым, выглядящим как последовательность чисел. Но к сожалению, обычные средства ввода ограничены по возможностям, трудны в использовании и зачастую не содержат механизмов проверки корректности.

YACC представляет собой универсальный инструмент для описания входного потока программ. Это имя является сокращением фразы "yet another compiler compiler" ("еще один компилятор компиляторов"). Пользователь задает как структуру входного потока, так и фрагменты программ, вызываемые при распознавании объектов в потоке. Компилятор компиляторов (или генератор программ синтаксического разбора, далее просто генератор) переводит спецификацию в некоторую подпрограмму, управляющую процессом ввода. Часто оказывается удобным осуществлять управление пользовательской задачей с помощью этой подпрограммы.

Подпрограмма, построенная генератором, для чтения базовой входной лексемы вызывает предоставляемую пользователем функцию. Таким образом, пользователь может описывать входной поток либо в терминах отдельных символов, либо более высокоуровневыми конструкциями (именами, числами). Пользовательская функция может обрабатывать и некоторые особенности входного потока, такие, как комментарии и соглашения о продолжении, что обычно облегчает грамматическую спецификацию. Класс спецификаций довольно широк: это грамматики LALR с набором правил.

Генератор используется как для разработки компиляторов широко распространенных языков (языки С, АПЛ, Паскаль, Ратфор и пр.), так и для нетрадиционных приложений (язык управления фотонаборной установкой, языки настольных калькуляторов, система доступа к документам, отладчик Фортрана).

Генератор предоставляет широкие возможности для задания структуры входного потока программы. Пользователь YACC задает спецификацию, управляющую процессом ввода, к которой относятся правила для описания структуры потока, фрагменты программ, вызываемые при распознавании этих правил, низкоуровневые функции для выполнения первичного ввода. По этой спецификации генератор строит функцию, управляющую процессом ввода. Эта функция, называемая синтаксическим анализатором, вызывает низкоуровневую пользовательскую подпрограмму (*лексический анализатор*), для выделения базовых элементов (лексем) из входного потока. Лексемы обрабатываются в соответствии с правилами, описывающими входной поток (грамматическими правилами). При распознавании такого правила вызывается соответствующий фрагмент пользовательской программы. Обратите внимание, что при этом можно возвращать значения, которые могут применяться в других фрагментах. Сам генератор написан на мобильном диалекте языка С, все действия и генерируемые подпрограммы также записываются на С. Более того, большинство синтаксических соглашений также соответствуют языку С.

Сердцем входной спецификации является набор грамматических правил. Каждое правило описывает допустимую структуру и присваивает ей имя. Например, правилом могла бы быть следующая строка:

```
date:month_name day '.' year ;
```

Date, month_name, day, year представляют собой некторые объекты входного потока, подразумевается, что они где-то определены. Символ запятой заключен в апострофы; это означает, что она должна восприниматься непосредственно. Двоеточие и точка с запятой выполняют роль знаков пунктуации в правилах и не влияют на распознавание входного потока. Учитывая приведенное определение, следующая строка

```
July 4, 1776
```

могла бы быть выделена этим правилом.

Важная часть обработки входного потока выполняется лексическим анализатором. Эта подпрограмма читает входной поток, выделяет в нем низкоуровневые структуры и передает эти лексемы синтаксическому анализатору. Объекты, распознаваемые лексическим анализатором, называются терминальными символами, в отличие от объектов, распознаваемых синтаксическим анализатором, которые называются нетерминальными символами. Во избежание путаницы будет называть терминальные символы лексемами.

При принятии решения, будут ли входные структуры распознаваться лексическими или грамматическими правилами, пользователю предоставляется значительная свобода. Например, для предыдущего примера можно записать следующие правила:

```
month_name: 'J''a''n';
month_name: 'F''e''b';
.
.
.
month_name: 'D''e''c';
```

Тогда лексический анализатор распознавал бы только отдельные символы, а month_name был бы нетерминальным символом. Такие низкоуровневые правила ведут к трате времени и пространства и могут сильно усложнить спецификацию (вплоть до невозможности ее обработки генератором). Обычно лексический анализатор распознает имена месяцев и возвращает соответствующую информацию. В этом случае month_name считается лексемой. Символы с непосредственным значением, такие, как запятая, также должны проходить через лексический анализатор и считаются лексемами.

Способы спецификация весьма гибки. К приведенному примеру сравнительно несложно добавить следующее правило:

```
date:month:'/'day'/'year ;
```

Оно позволяет ввод фразы

```
7/4/1776
```

в качестве синонима для

```
July 4, 1776
```

В большинстве случаев новое правило может быть включено в работающую систему с минимальными усилиями и достаточно безопасно. Читаемый входной поток может не соответствовать спецификации. Ошибки обнаруживаются с максимальной быстротой, какую теоретически может предоставить просмотр слева направо. Таким образом, во-первых, снижается вероятность обработки неправильных данных, во-вторых, такие данные обычно можно быстро обнаружить. Процедуры обработки ошибок, задаваемые как часть входной спецификации, позволяют повторный ввод при неверных данных, либо продолжение процесса ввода после пропуска некорректной информации.

В некоторых случаях генератор не может построить анализатор из заданного набора спецификаций. Например, в них может содержаться противоречие, или требование более мощного механизма распознавания, нежели доступный. Первый случай соответствует ошибкам проектирования, последний может устраняться либо усложнением лексического анализатора, либо переписью грамматических правил. Хотя уасс и не может обработать все возможные спецификации, по мощности он сопоставим с аналогичными системами. Более того, если некоторые конструкции для него представляют сложности, часто эти конструкции будут сложны и для вас. Пользователи сообщают, что благодаря дисциплине формулировки правильных спецификаций, применяемой в уасс, концептуальные ошибки проектирования выявляются на ранних этапах разработки.

Спецификации

К *нетерминальным символам* или лексемам обращаются по именам. *Yacc* требует непосредственного объявления имен лексем. В дополнение, по причинам, объясняемым ниже, часто желательно включение лексического анализатора как части файла спецификации. Может оказаться полезным и включения ряда других программ. Таким образом, любой файл спецификации состоит из трех частей: объявлений, правил и программ. Части (или разделы) разделяются двойным знаком процента (%%). (Символ процента часто применяется в спецификациях в виде специального символа.)

Другими словами, полная спецификация может быть записана следующим образом:

```
объявления
%%
правила
%%
```

программы

Раздел объявлений может быть пустым. Более того, если опускается раздел программ, то второй разделитель %% можно не указывать. Тогда минимальная спецификация выглядит как

```
%%
правила
```

Пробелы, табуляции и переводы строк игнорируются. Они также не могут появляться в именах или многолитерных зарезервированных символах. Комментарии могут появляться в любой позиции имени, их синтаксис совпадает с синтаксисом комментариев в С.

Раздел правил состоит из одного или более грамматических правил. Грамматическое правило записывается в формате

```
A : BODY ;
```

A представлет собой нетерминальное имя, BODY -последовательность имен и литералов (возможно пустую).

Имена могут быть произвольной длины и составляются из букв, точки, подчеркивания и цифр. Цифры в начале имени не допускаются. Прописные и строчные буквы считаются различными. Имена, используемые в теле грамматического правила, могут являться как лексемами, так и нетерминальными символами.

Литерал представляет собой символ, заключенный в апострофы. Так же, как и в С, обратная дробная черта служит механизмом экранирования внутри литералов, распознаются все специальные последовательности языка С:

\n	**Перевод строки**
\r	**Возврат каретки**
\'	**Апостроф**
****	**Обратная дробная черта**
\t	**Табуляция**
\b	**Шаг назад**
\f	**Перевод формата**
\xxx	**Восьмеричное число xxx**

По ряду причин символ NUL (ПУС, \0 или 0) никогда не должен использоваться в грамматических правилах. Если у нескольких правил одинаковая левая часть, во избежание ее повторения может применяться символ |. Точка с запятой в конце правила перед вертикальной чертой может опускаться. Таким образом, следующие правила:

```
A:B C D;
A:E F ;
A:G ;
```

могут быть записаны как

```
A:B C D;
|E F
|G
;
```

Хотя и необязательно, чтобы все правила с одинаковой левой частью находились рядом, это делает спецификации более читаемыми и облегчает внесение изменений. Если нетерминальный символ соответствует пустой строке, можно записать следующую конструкцию:

```
empty:;
```

Имена, представляющие лексемы, должны объявляться явно. Это можно сделать в разделе объявлений:

```
%token name1 name2 ...
```

(Более подробно это описано в разделах 3, 5 и 6 этой главы.) Каждый нетерминальный символ должен появиться в левой части хотя бы одного правила. Из всех нетерминальных символов один, называемый начальным, играет особую роль. Анализатор строится так, чтобы распознавать начальный символ; таким образом, он должен описывать самую большую, наиболее общую структуру, представляемую грамматическими правилами. По умолчанию, начальным символом считается левая часть первого грамматического правила в разделе правил. Возможно и желательно явно объявить начальный символ в разделе объявлений с помощью ключевого слова %start:

```
%start symbol
```

Конец ввода анализатора отмечается специальной лексемой, называемой конечным маркером. Если лексемы вплоть до конечного маркера (но не включая его) образуют структуру, удовлетворяющую определению начального символа, функция анализатора возвращает управление

вызывающей программе. Если конечный маркер распознается в другом контексте, это считается ошибкой. Возврат конечного маркера - задача разрабатываемой пользователем функции лексического анализа; об этом подробно изложено в разделе 3. Обычно конечный маркер соответствует некоторому очевидному состоянию ввода-вывода: концу файла или концу записи.

Действия

С каждым правилом может быть связано действие, выполняемое при распознавании во входном потоке объекта, удовлетворяющего правилу. Эти действия могут возвращать значения и воспринимать значения, возвращаемые другими действиями. Более того, при желании лексический анализатор может возвращать значения для выделяемых *лексем*.

Действие - это произвольный оператор языка С. В нем можно выполнять ввод-вывод, вызывать подпрограммы и изменять внешние переменные или массивы. Действие указывается как один или несколько операторов в фигурных скобках. Например, следующие фрагменты представляют собой грамматические правила с действиями:

```
A:'('B')'        { hello(1, "abc"); }
XXX:YYYZZZ       { printf("a message\n"); flag=25;}
```

Для упрощения связи между действиями и анализатором операторы действий слегка изменяются. В качестве механизма сигнализации в этом контексте используется знак доллара. Для возврата значения действие обычно присваивает псевдопеременной $$ какое-либо значение. Например, действие, которое ничего не выполняет кроме возврата 1:

```
{$$=1;}
```

Для получения значений, возвращаемых предыдущими действиями и лексическим анализатором, действие может пользоваться псевдопеременными $1, $2, ..., которые соответствуют значениям, возвращаемым правой частью правила, слева направо. Тогда, если правило выглядит как

```
A:B C D                  ;
```

то $2 - значение, возвращаемое С, $3 - значение, возвращаемое D.

В качестве более конкретного примера рассмотрим правило:

```
expr:'('expr')'          ;
```

Значение, возвращаемое этим правилом - выражение expr в скобках. Это можно указать как

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

По умолчанию, значением правила служит значение его первого элемента ($1). Таким образом, грамматическое правило вида

```
A:B
```

может не содержать явных действий.

В приведенных примерах все действия указаны в конце соответствующих правил. Иногда требуется получить управление до завершения полного распознавания правила. Генератор позволяет записывать действия как в середине правила, так и в конце. Подразумевается, что правило возвращает значения посредством механизма $ через действия справа от знака. Но в свою очередь оно может получать значения, возвращаемые символами слева от этого знака. Таким образом, в правиле

```
A:B          { $$=1; }
C            { x=$2; y=$3; }
             ;
```

x получит значение 1, y - значение, возвращаемое C.

Действия, не завершающие правило, на самом деле обрабатываются созданием нового имени нетерминального символа и нового правила, связывающего это имя с пустой строкой. Внутреннее действие выполняется при распознавании этого правила. На самом деле предыдущий пример рассматривается следующим образом:

```
$ACT:        /*empty*/
             { $$=1;}
             ;
A:B $ACT C
             { x=$2 y=$3;}
             ;
```

В большинстве применений действия не выполняют непосредственного вывода. Обычно в памяти строится некоторая структура данных (например, дерево разбора), над которой и осуществляются преобразования. Особенно легко создавать деревья разбора при наличии функций построения и

манипуляции древовидными структурами. Например, предположим, что существует функция node, следующий вызов которой

```
node(L, n1, n2)
```

создает узел с меткой L и потомками n1 и n2 и возвращает.P Для действий пользователь может определять дополнительные переменные. Объявления и определения помещаются в разделе объявлений между ограничителями %{ и %}. Эти объявления считаются глобальными, поэтому они известны всем действиям и лексическому анализатору. Например, строка

```
%{ int variable=0; %}
```

помещенная в раздел объявлений, делает переменную variable доступной для всех действий. Анализатор yacc использует только имена, начинающиеся с yy; пользователь должен избегать применения подобных имен. В приведенных примерах все значения имеют тип целый. Рассмотрение значений других типов проводится ниже.

Лексический анализ

Для чтения входного потока и передачи *лексем* (при необходимости, со значениями) программе разбора пользователь должен разработать лексический анализатор. Лексический анализатор - функция, возвращающая целое, с именем yylex. Функция возвращает целое число, называемое номером лексемы, которое обозначает тип прочитанной лексемы. Если с лексемой связано значение, оно должно присваиваться внешней переменной yylval. Для нормального взаимодействия между программой разбора и синтаксическим анализатором номера лексем должны быть согласованы. Номера выбираются либо yacc, либо пользователем. В любом случае для символического обмена именами применяется механизм директивы #define. Например, предположим, что лексема DIGIT определена в разделе объявлений файла спецификаций. Соответствующая часть лексического анализатора могла бы выглядеть так:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch(c){
        ...
        case '0':
```

```
        case '1':
        ...
        case '9':
                        yylval = c-'0';
                        return (DIGIT);
        ...
        }
    ...
```

Нам нужно вернуть лексему с номером DIGIT и числовое значение. Если фрагмент, связанный с лексическим анализатором, помещен в программный раздел файла спецификаций, идентификатор DIGIT определяется как индекс созданного узла. Этот механизм ведет к построению легко понимаемых и модифицируемых лексических анализаторов. Единственное ограничение состоит в необходимости избегать имен лексем, совпадающих с зарезервированными словами языка Си или анализатора. Например, использование лексем if или while наверняка приведет к серьезным трудностям при компиляции. Лексема error зарезервирована для обработки ошибок и должна применяться осознанно. Как уже упоминалось, номера лексем могут выбираться либо самим построителем, либо пользователем. По умолчанию они выбираются построителем. Номер по умолчанию для литерального символа - числовое значение его кода в наборе символов. Другие имена получают номера, начиная с 257. Для явного присвоения лексеме номера после первого вхождения имени в разделе определений необходимо указать неотрицательное целое число. Это число считается номером имени или литерала. Имена или литералы, не затронутые этим механизмом, сохраняют значения по умолчанию. Важно отметить, что все номера должны быть различными.

По историческим причинам, конечный маркер должен нумероваться либо нулем, либо отрицательным числом. Этот номер не должен переопределяться пользователем. Таким образом, все лексические анализаторы должны при достижении конца входного потока возвращать либо 0, либо отрицательное число в качестве номера лексемы.

Весьма полезный инструмент для построения лексических анализаторов, lex, рассматривается в предыдущей главе. Лексические анализаторы строятся так, чтобы их поведение было согласовано с yacc. При спецификации используются не грамматические правила, а регулярные выражения. Lex легко может применяться для построения довольно сложных лексических анализаторов, но существуют языки (например, ФОРТРАН), не удовлетворяющие ни одной теоретической модели, и анализаторы для них приходится разрабатывать вручную.

Как работает построитель

YACC переводит спецификацию в программу на язык C, которая и обрабатывает входной поток в соответствии с заданными правилами. Алгоритм получения программы разбора по спецификации довольно сложен и здесь рассматриваться не будет. Сама же программа разбора относительно несложна, и понимание принципов ее работы хотя и не обязательно, может

существенно облегчить процедуру построения функций обработки ошибок и анализ возможных неоднозначностей.

Получаемая программа разбора является стековым конечным автоматом. Также существует возможность чтения и запоминания следующей входной лексемы, называемой очередной. Текущее состояние всегда находится в вершине стека. Состояниям конечного автомата присвоены метки в виде небольших целых чисел. В начале работы автомат находится в состоянии 0 и стек содержит только метку 0; очередная *лексема* не прочитана. Автомат может выполнять четыре типа действий: сдвиг, свертка, ввод и ошибка. Операция программы разбора выполняется следующим образом:

1. Основываясь на текущем состоянии, программа разбора определяет, нужна ли для выполняемого действия очередная лексема. Если да, а она не прочитана, для ее ввода вызывается функция yylex.
2. Используя текущее состояние и, при необходимости, очередную лексему, программа разбора определяет следующее действие и выполняет его. Это может привести к помещению состояний в стек или их извлечению из стека, а также к обработке или обходу очередной лексемы.

Наиболее распространенным действием служит сдвиг. Для этого действия всегда нужна очередная лексема. Например, в состоянии 56 может выполняться следующее действие:

```
IF shift 34
```

Это означает, что если очередная лексема есть IF, состояние 56 заталкивается в стек, а текущим состоянием (верхушка стека) становится 34. Очередная лексема обнуляется.

Свертка нужна для ограничения роста стека. Это действие уместно при обнаружении правой части грамматического правила и подготовке к замене ее левой частью. Иногда для выяснения необходимости свертки нужно проверить очередную лексему, но чаще всего без этого можно обойтись. Фактически, действием по умолчанию (обозначаемым символом '.') обычно служит свертка.

Свертка часто связывается с отдельными грамматическими правилами. Эти правилам также присваиваются небольшие целые числа, что ведет к путанице. Действие

```
. reduce 18
```

ссылается на правило 18, а действие

```
IF shift 34
```

ссылается на состояние 34.

Предположим, что свертываемое правило выглядит следующим образом:

```
A: x y z;
```

Свертка зависит от символа в левой части (в данном случае A) и количества символов в правой части (в данном случае три). Для свертки из стека выталкиваются три состояния. (В общем случае, количество выталкиваемых состояний равно количеству символов в правой части.) Фактически, эти действия были помещены в стек при распознавании x, y и z и больше они не нужны. После этого текущим состоянием становится состояние, в котором находился распознаватель перед обработкой правила. С помощью этого состояния и символа в левой части правила выполним сдвиг A. Полученное новое состояние помещается в стек, и разбор продолжается. Однако, существуют значительные различия между обработкой символа в левой части и обычным сдвигом лексемы, поэтому это действие называется переходом. В частности, очередная лексема при сдвиге очищается, а при переходе нет. В любом случае новое состояние содержит строку вида

```
A goto 20
```

вследствие чего состояние 20 помещается в стек м становится текущим.

Фактически, свертка переводит стрелку часов распознавателя назад, выталкивая состояния из стека и приводя его к моменту первого обнаружения правой части правила. Распознаватель введет себя так, как если бы он впервые увидел левую часть правила. Если правая часть правила пуста, состояния из стека не выталкиваются, выявленное состояние становится текущим.

Свертка также существенна при обработке задаваемых пользователем значений и действий. При свертывании правила программный фрагмент, связанный с ним, выполняется перед выравниваем стека. В дополнение к стеку, содержащему состояния, существует стек, в котором содержатся значения, возвращаемые лексическим анализатором и действиями. При сдвиге ввнешняя переменная yylval копируется в стек значений. Свертка выполняется после возврата из пользовательского фрагмента. При переходе в стек значений копируется внешняя переменная yyval. К стеку значений можно обращаться по именам псевдопеременных $1, $2 и т.д.

Два других действия распознавателя значительно проще. Ввод означает, что распознана входная информация, удовлетворяющая спецификации. Это действие выполняется только если очередная лексема является конечным маркером, и означает успешное завершение работы. Действие по ошибке, напротив, сигнализирует, что распознаватель больше не может продолжать обработку спецификации. Входная лексема вместе с очередной не удовлетворяют ни одному правилу. Распознаватель сообщает об ошибке и пытается возобновить работу. Восстановление после ошибок (в отличие от их обнаружения) описано в следующем разделе. Рассмотрим следующий пример:

```
%token DING DONG DELL
%%
```

```
rhyme: sound place
          ;
          sound: DING DONG ;
                              place: DELL
;
```

При вызове yacc с флагом -v выдается файл y.output, содержащий читаемое описание распознавателя. Ниже приведен файл, соответствующий данной грамматике:

```
state 0
          $accept : _rhyme $end
                                    DING  shift 3
      . error
          rhyme  goto 1         sound  goto 2
state 1
                                    $accept :  rhyme_$end
          $end  accept
      . error
state 2
                                    rhyme :  sound_place
          DELL  shift 5
      . error
          place  goto 4
state 3
                                    sound :  DING_DONG
          DONG  shift 6
      . error
state 4
          rhyme :  sound place_    (1)
  . reduce 1
state 5
                                    place :  DELL_    (3)

  . reduce 3
      state 6
                  sound :  DING DONG_    (2)
```

```
. reduce 2
```

Заметьте, что в дополнение к действиям для каждого состояния описаны правила разбора в каждом состоянии. Подчеркивание используется для обозначения того, что уже распознано и что предстоит распознать в каждом правиле. Предположим, на ввод подается

```
DING DONG DELL
```

Полезно рассмотреть по шагам все действия распознавателя. Первоначальное состояние 0. Для выбора действий состояния 0 распознаватель должен прочесть что-либо со входа. Читается первая лексема DING, которая становится очередной. При чтении DING в состоянии 0 выполняется действие shift 3, поэтому состояние 3 заносится в стек, и очередная лексема очищается. Следующей лексемой становится DONG. При чтении DONG в состоянии 3 выполняется действие shift 6, поэтому состояние 6 заносится в стек, и очередная лексема очищается. В стеке содержится 0,3 и 6. В состоянии 6, не обращаясь к очередной лексеме, распознаватель выполняет свертку по правилу 2.

```
sound: DING DONG
```

У правила в правой части 2 символа, поэтому состояния 6 и 3 выталкиваются из стека, выявляя состояние 0. Затем проверяется описание состояния 0 в поисках перехода по sound:

```
sound goto 2
```

При этом состояние 2 заносится в стек и становится текущим. В состоянии 2 должна быть прочитана следующая лексема DELL. Действием служит shift 5, поэтому состояние 5 заносится в стек, который теперь содержит 0,2 и 5, и очередная лексема очищается. В состоянии 5 единственным действием служит свертка по правилу 3. У него в правой части только один символ, поэтому из стека выталкивается состояние 5, открывая состояние 2. Переход в состоянии 2 на place, левую часть правила 3, дает состояние 4. Теперь стек содержит 0,2 и 4. В состоянии 4 единственное действие - свертка по правилу 1. В правой части 2 символа, поэтому выталкиваются два верхних значения, открывая состояние 0. В состоянии 0 выполняется переход по rhyme, переводя распознаватель в состояние 1. В этом состоянии при вводе обнаруживается конечный маркер, указываемый в файле с помощью $end. Выполняется действие состояния 1, и распознаватель успешно заканчивает работу.

Рекомендуется самостоятельно рассмотреть случаи некорректного ввода: DING DONG DONG, DING DONG, DING DONG DELL и пр. Потратив на анализ подобных примеров несколько минут, вы сэкономите себе время в более сложных случаях.

Неоднозначности и конфликты

Набор граматических правил неоднозначен, если входная строка может интерпретироваться по-разному. Например, правило:

```
expr: expr'-'expr
```

естественный путь записи арифметического выражения, как двух выражений со знаком минус между ними. К сожалению, данное правило не указывает способа обработки более сложных конструкций. Например, если на входе будет

```
expr-expr-expr
```

правило позволяет интерпретировать ввод как

```
(expr-expr)-expr
```

так и как

```
expr-(expr-expr)
```

Первый случай - левая ассоциативность, второй - правая. Yacc при попытке построения распознавателя такие случаи обнаруживает. Полезно рассмотреть действия, выполняемые распознавателем при обнаружении подобных конструкций. При чтении второго выражения строка expr-expr удовлетворяет правой части приведенного правила.Таким образом, можно выполнить свертку. После нее на входе остается expr (левая часть правила). Затем читается оставшаяся часть выражения -expr, после чего снова выполняется свертка. В результате получается левоассоциативная интерпретация.

И наоборот, при чтении expr-expr можно было бы отложить немедленное применение правила и читать дальше до обнаружения expr-expr-expr. тогда свертнуты будут два последних символа, что приведет к правой ассоциативности. Таким обазом, прочитав expr-expr, распознаватель может выполнить два равноправных действия, свертку и сдвиг, и не существует способа выбрать одно из них. Может случиться, что нужно будет выбирать между двумя правомочными свертками, это называется конфликтом свертка-свертка. Заметьте, что конфликтов сдвиг-сдвиг не существует. При обнаружении приведенных конфликтов yacc все равно строит распознаватель. Это выполняется на основе одного из возможных вариантов. Правило, описывающее какие действия предпринимать в данной ситуации, называется правилом однозначности. По умолчанию применяются два правила однозначности:

1. В конфликте сдвиг-свертка предрочтение отдается сдвигу.
2. В конфликте свертка-свертка предпочтение отдается первой встреченной свертке.

Первое правило говорит о том, что применение свертки откладывается в пользу сдвига. Правило 2 дает пользователю негибкий метод управления, поэтому рекомендуется избегать подобных конфликтов.

Конфликты могут возникать либо вследствие ошибок во входной спецификации, либо потому, что для обработки корректных правил нужен распознаватель более сложный, нежели генерируемый уасс. К конфликтам может привести использование действий внутри правил, если действие применяется до того, как распознаватель выявит правило. В этом случае применение правил однозначности неуместно и ведет к некорректному распознавателю. По этой причине уасс всегда сообщает количество разрешенных по двум правилам конфликтов.

В общем случае, если возможно применить правила однозначности, всегда можно переписать грамматику так, чтобы конфликты не возникали. По этой причине, большинство генераторов рассматривали конфликт как неустранимую ошибку. По нашему мнению, перепись грамматики выглядит неестественно и приводит к медленным распознавателям. Таким образом, уасс всегда строит распознаватель, даже при наличии конфликтов. Как пример мощности правил однозначности, рассмотрим фрагмент программы с конструкцией if-then-else:

```
stat: IF'('cond')' stat
                          | IF'('cond')'stat ELSE stat
                ;
```

Здесь IF и ELSE лексемы, cond - нетерминал, описывающий условные выражения, stat - *нетерминал*, описывающий операторы. Первое правило назовем простым, второе составным. Эти правила приводят к неоднозначностям, так как входная строка вида

```
IF (C1) IF (C2) S1 ELSE S2
```

может структурироваться двумя путями:

```
IF (C1) {
          IF (C2) S1
} ELSE S2
```

либо

```
IF (C1) {
          IF (C2) S1
ELSE S2 }
```

Второй вариант наиболее распространен. Каждый ELSE связывается с последним IF, непосредственно предшествующим ELSE. Рассмотрим ситуацию, когда на входе IF (C1) IF(C2) S1, и распознаватель ищет ELSE. Можно сразу выполнить свертку по правилу для простого оператора и получить IF (C1) stat, а затем прочесть оставшийся ввод ELSE S2 и выполнить свертку по правилу составного оператора. С другой стороны, ELSE может быть сдвинут, прочитан S2, а правая часть

```
IF (C1) IF (C2) S1 ELSE S2
```

будет свернута по правилам составного оператора. Это ведет ко второму варианту группирования, что наиболее желательно.

Конфликты сдвиг-свертка возникают только при чтении определенного входного символа, ELSE и уже распознанной конструкции, как например

```
IF (C1) IF (C2) S1
```

В общем случае, конфликтов может быть много, и каждый из них будет связан со входным символом и набором уже прочитанных строк. Прочитанные строки характеризуются состояниями распознавателя.

Сообщения о конфликтах лучше всего разбирать по выходному файлу y.output. Возможный пример выдачи приведен ниже:

```
23: shift/reduce conflict (shift 45, reduce 18)
on ELSE
state 23
            stat: IF (cond) stat_ (18)
            stat: IF (cond) stat_ELSE stat
            ELSE shift 45                    . reduce 18
```

Первая строка описывает конфликт, определяя состояние и входной символ. Далее идет обычное описание состояния, в котором указано активное правило и действия. Вспомните, что символ подчеркивания отмечает уже прочитанные правила. Распознаватель может выполнить два возможных действия. Если входной символ ELSE, можно выполнить сдвиг в состояние 45. В состоянии 45 будет следующая строка:

```
stat: IF (cond) stat_ELSE stat
```

Заметьте, что в этом состоянии ELSE всегда сдвигается. В состоянии 23 альтернативное действие, описываемое `.' выполняется в том случае, если входной символ явно в правилах не указан. В этом случае, если входной символ не есть ELSE распознаватель выполняет свертку по правилу 18.

```
stat: IF'('cond')' stat
```

Не забудьте, что числа после команд свдига указывают на состояния, а числа после команд свертки относятся к правилам. В файле y.output после сворачиваемых правил указываются их номера. В каждом состоянии можно выполнить только одну свертку, которая и служит действием по умолчанию. Пользователь, столкнувшийся с неожиданными конфликтами, по-видимому, захочет заглянуть в файл y.output, чтобы убедиться в приемлемости действий по умолчанию. В серьезных случаях ему понадобится значительно больше информации, чем приводится в этом документе. В этом случае лучше обратиться к приведенному списку литературы или помощи знающего пользователя.

Неоднозначности и конфликты

Набор граматических правил неоднозначен, если входная строка может интерпретироваться по-разному. Например, правило:

```
expr: expr'-'expr
```

естественный путь записи арифметического выражения, как двух выражений со знаком минус между ними. К сожалению, данное правило не указывает способа обработки более сложных конструкций. Например, если на входе будет

```
expr-expr-expr
```

правило позволяет интерпретировать ввод как

```
(expr-expr)-expr
```

так и как

```
expr-(expr-expr)
```

Первый случай - левая ассоциативность, второй - правая. Yacc при попытке построения распознавателя такие случаи обнаруживает. Полезно рассмотреть действия, выполняемые распознавателем при обнаружении подобных конструкций. При чтении второго выражения строка expr-expr удовлетворяет правой части приведенного правила.Таким образом, можно выполнить свертку. После нее на входе остается expr (левая часть правила). Затем читается оставшаяся часть выражения -expr, после чего снова выполняется свертка. В результате получается левоассоциативная интерпретация.

И наоборот, при чтении expr-expr можно было бы отложить немедленное применение правила и читать дальше до обнаружения expr-expr-expr. тогда

свертнуты будут два последних символа, что приведет к правой ассоциативности. Таким обазом, прочитав expr-expr, распознаватель может выполнить два равноправных действия, свертку и сдвиг, и не существует способа выбрать одно из них. Может случиться, что нужно будет выбирать между двумя правомочными свертками, это называется конфликтом свертка-свертка. Заметьте, что конфликтов сдвиг-сдвиг не существует. При обнаружении приведенных конфликтов уасс все равно строит распознаватель. Это выполняется на основе одного из возможных вариантов. Правило, описывающее какие действия предпринимать в данной ситуации, называется правилом однозначности. По умолчанию применяются два правила однозначности:

1. В конфликте сдвиг-свертка предрочтение отдается сдвигу.
2. В конфликте свертка-свертка предпочтение отдается первой встреченной свертке.

Первое правило говорит о том, что применение свертки откладывается в пользу сдвига. Правило 2 дает пользователю негибкий метод управления, поэтому рекомендуется избегать подобных конфликтов.

Конфликты могут возникать либо вследствие ошибок во входной спецификации, либо потому, что для обработки корректных правил нужен распознаватель более сложный, нежели генерируемый уасс. К конфликтам может привести использование действий внутри правил, если действие применяется до того, как распознаватель выявит правило. В этом случае применение правил однозначности неуместно и ведет к некорректному распознавателю. По этой причине уасс всегда сообщает количество разрешенных по двум правилам конфликтов.

В общем случае, если возможно применить правила однозначности, всегда можно переписать грамматику так, чтобы конфликты не возникали. По этой причине, большинство генераторов рассматривали конфликт как неустранимую ошибку. По нашему мнению, перепись грамматики выглядит неестественно и приводит к медленным распознавателям. Таким образом, уасс всегда строит распознаватель, даже при наличии конфликтов. Как пример мощности правил однозначности, рассмотрим фрагмент программы с конструкцией if-then-else:

```
stat: IF'('cond')' stat
                          | IF'('cond')'stat ELSE stat
                ;
```

Здесь IF и ELSE лексемы, cond - нетерминал, описывающий условные выражения, stat - *нетерминал*, описывающий операторы. Первое правило назовем простым, второе составным. Эти правила приводят к неоднозначностям, так как входная строка вида

```
IF (C1) IF (C2) S1 ELSE S2
```

может структурироваться двумя путями:

```
IF (C1) {
          IF (C2) S1
} ELSE S2
```

либо

```
IF (C1) {
          IF (C2) S1
ELSE S2 }
```

Второй вариант наиболее распространен. Каждый ELSE связывается с последним IF, непосредственно предшествующим ELSE. Рассмотрим ситуацию, когда на входе IF (C1) IF(C2) S1, и распознаватель ищет ELSE. Можно сразу выполнить свертку по правилу для простого оператора и получить IF (C1) stat, а затем прочесть оставшийся ввод ELSE S2 и выполнить свертку по правилу составного оператора. С другой стороны, ELSE может быть сдвинут, прочитан S2, а правая часть

```
IF (C1) IF (C2) S1 ELSE S2
```

будет свернута по правилам составного оператора. Это ведет ко второму варианту группирования, что наиболее желательно.

Конфликты сдвиг-свертка возникают только при чтении определенного входного символа, ELSE и уже распознанной конструкции, как например

```
IF (C1) IF (C2) S1
```

В общем случае, конфликтов может быть много, и каждый из них будет связан со входным символом и набором уже прочитанных строк. Прочитанные строки характеризуются состояниями распознавателя.

Сообщения о конфликтах лучше всего разбирать по выходному файлу y.output. Возможный пример выдачи приведен ниже:

```
23: shift/reduce conflict (shift 45, reduce 18)
on ELSE
state 23
          stat: IF (cond) stat_ (18)
          stat: IF (cond) stat_ELSE stat
          ELSE shift 45                    . reduce 18
```

Первая строка описывает конфликт, определяя состояние и входной символ. Далее идет обычное описание состояния, в котором указано активное правило и действия. Вспомните, что символ подчеркивания отмечает уже прочитанные правила. Распознаватель может выполнить два возможных действия. Если входной символ ELSE, можно выполнить сдвиг в состояние 45. В состоянии 45 будет следующая строка:

```
stat: IF (cond) stat_ELSE stat
```

Заметьте, что в этом состоянии ELSE всегда сдвигается. В состоянии 23 альтернативное действие, описываемое `.' выполняется в том случае, если входной символ явно в правилах не указан. В этом случае, если входной символ не есть ELSE распознаватель выполняет свертку по правилу 18.

```
stat: IF'('cond')' stat
```

Не забудьте, что числа после команд свдига указывают на состояния, а числа после команд свертки относятся к правилам. В файле y.output после сворачиваемых правил указываются их номера. В каждом состоянии можно выполнить только одну свертку, которая и служит действием по умолчанию. Пользователь, столкнувшийся с неожиданными конфликтами, по-видимому, захочет заглянуть в файл y.output, чтобы убедиться в приемлемости действий по умолчанию. В серьезных случаях ему понадобится значительно больше информации, чем приводится в этом документе. В этом случае лучше обратиться к приведенному списку литературы или помощи знающего пользователя.

Предшествование

Существует одна распространенная ситуация, в которой приведенных выше правил разрешения конфликтов недостаточно: это распознавание арифметических выражений. Большинство используемых конструкций естественно описываются с помощью понятия предшествования операторов и левой или правой ассоциативности. Оказывается, что неоднозначная грамматика с правилами однозначности приводит к построению распознавателей, работающих быстрее, нежели распознаватели, созданные по однозначной грамматике. Основой этого подхода служит запись всех правил для бинарных и унарных операций в виде

```
expr: expr OP expr
expr: UNARY expr
```

При этом получается сильно неоднозначная грамматика с большим количеством конфликтов. В качестве правила однозначности задается предшествование для всех операций и ассоциативность бинарных операций. Этой информации достаточно для разрешения конфликтов и построения распознавателя, понимающего предшествование и ассоциативность.

Предшествование и ассоциативность связываются с лексемами в разделе определений. Это выполняется с помощью ключевых слов %left, %right и %noassoc, за которыми идет список лексем. Все *лексемы* на одной строке обладают одним уровнем предшествования и ассоциативности. Строки перечисляются в порядке возрастания приоритета. Таким образом:

```
%left '+' '-'
%left '*' '/'
```

определяют предшествование и ассоциативность четырех арифметических операций. Плюс и минус имеют левую ассоциативность и меньший приоритет, чем звездочка и дробная черта. Ключевое слово %right описывает правую ассоциативность, ключевое слово %noassoc описывает операторы, аналогичные оператору .LT. ФОРТРАНА, которые нельзя использовать в выражении подряд (A.LT.B.LT.C). Приведем в качестве примера следующее описание:

```
%right '='
%left  '+' '-' %left '*' '/'
%%
expr: expr'='expr | expr '+' expr | expr '-' expr | expr '*' expr | expr '/' expr | NAME ;
```

Если на вход подается строка

```
a=b=c*d-e-f*g
```

она будет распознана следующим образом:

```
a=(b=(((c*d)-e)-(f*g)))
```

При использовании этого механизма приоритет выше у унарных операций. Иногда унарная и бинарная операции имеют одинаковый синтаксис, но разный приоритет. Примером может служить минус: унарный минус имеет такую же силу, как и умножение, тогда как приоритет бинарного минуса ниже приоритета умножения. Ключевое слово %prec меняет уровень приоритета, связанный с данным правилом. Оно ставится непосредственно после тела правила перед действием или завершающим символом `;', после него идет имя лексемы или литерала. Это приводит к тому, что приоритет правила становится равным приоритету указанной лексемы или литерала. Например, чтобы приоритеты унарного минуса и умножения совпадали, строки могут выглядеть следующим образом:

```
%left  '+' '-'
%left '*' '/'
%%
expr: expr'='expr | expr '+' expr | expr '-' expr | expr '*'
expr | expr '/' expr | '-' expr        %prec '*'| NAME;
```

Лексемы, указанные в аргументах директив %left, %right и %noassoc не обязательно, хотя и не запрещено указывать в директиве %token. Предшествование и ассоциативность используются уасс для разрешения конфликтов с помощью правил однозначности. Формально, правила работают следующим образом:

1. Для соответствующих лексем и литералов записывается предшествование и ассоциативность.
2. С каждым грамматическим правилом связывается предшествование и ассоциативность: они совпадают с предшествованием и ассоциативностью последнего литерала или лексемы в теле правила. Конструкция %prec переопределяет правила по умолчанию. Для некотрых правил предшествование и ассоциативность могут отсутствовать.
3. При наличии конфликтов свертка-свертка или сдвиг-свертка при отсутствии либо у входного символа, либо у правила предшествования или ассоциативности, применяются описанные ранее правила однозначности. Также сообщается обо всех конфликтах.
4. При конфликте сдвиг-свертка и наличии предшествования и ассоциативности как у правила, так и у входного символа, конфликт разрешается в пользу действия (сдвига или свертки), чей приоритет выше. Если приоритеты одинаковы, применяется ассоциативность: левая ассоциативность подразумевает свертку, права сдвиг. Отсутствие ассоциативности считается ошибкой.

Конфликты, разрешенные по приоритетам, в число сообщаемых уасс конфликтов не попадают. Это означает, что ошибки при задании предшествования могут скрыть ошибки во входной спецификации. Лучше всего использовать задание предшествования так, как сказано в руководстве, пока вы не приобретете достаточного опыта. Для выяснения, что же на самом деле делает распознаватель, очень полезен файл y.output.

Обработка ошибок

Обработка ошибок довольно сложное дело, так как большинство ситуаций связано с семантикой. При обнаружении ошибок может понажобиться, например, освободить память для дерева разбора, удалить или изменить строки в таблице символов и, что чаще всего, установить некоторые флаги для подавления генерации выходной информации.

При обнаружении ошибок прекращение обработки обычно неприемлемо. Более полезным является продолжение просмотра для обнаружения возможных ошибок. Это ведет к необходимости повторного запуска распознавателя после ошибки. Существует общий класс алгоритмов для этих действий, который включает в себя отбрасывание из входной строки ряда лексем и попытки изменить состояние распознавателя для продолжения обработки.

Для того, чтобы пользователь мог управлять этим процессом, уасс предоставляет простое, но достаточно универсальное средство. Для обработки ошибок зарезервирована лексема с именем error. Это имя может использоваться в грамматических правилах: им отмечаются места, где может встретиться ошибка и где необходимо провести восстановление. Распознаватель выталкивает состояния из стека до тех пор, пока не найдет состояние, в котором error допустимо. Далее считается что error - очередная лексема, и выполняются соответствующие действия. Затем значение очередной лексемы устанавливается равным лексеме, вызвавшей ошибку. Если никаких других правил не указано, при обнраужении ошибки обработка прекращается.

Для предотвращения потока сообщений об ошибках распознаватель после обнаружения ошибки остается в этом состоянии, пока не будут прочитаны и обработаны три следующие лексемы. Если ошибка обнаружена в момент, когда распознаватель находится в состоянии обработки ошибок, сообщений не выводится и входная лексема удаляется. Рассмотрим в качестве примера правило:

```
stat: error
```

Оно означает, что в случае ошибки распознаватель попытается пропустить предложение, в котором она встретилась. Более точно, распознаватель продолжит чтение в поисках трех *лексем*, которые допустимы после предложения, и начнет обработку первой из них. Если начало предложения недостаточно хорошо различимо, распознаватель может сбиться и начать обработку в его середине. При этом будет выведено сообщение об ошибке, хотя ее на самом деле нет. С правилами обработки ошибок могут быть связаны действия. Они могут выполнять операции по повторной инициализации таблиц, освобождению памяти и пр. Приведенные правила обработки ошибок довольно универсальны, но трудны в управлении.

Несколько легче использовать правила вида:

```
stat: error';'
```

При обнаружении ошибки распознаватель попытается пропустить предложение, но будет это делать, пока не встретит символ ';'. Все лексемы после ошибки и вплоть до символа ';' не могут сдвигаться и отбрасываются. При обнаружении этого символа правило свертывается и выполняются соответствующие действия.

Еще один вариант обработки ошибок применяется в интерактивных системах для обеспечения возможности повторного ввода строки в случае ошибки. Правило может выглядеть следующим образом:

```
input: error'0 {printf("Введите строку:"); }
input  {$$=$4;}
```

В этом подходе есть одна потенциальная трудность: распознаватель должен корректно обработать три следующих лексемы перед тем, как восстановить ситуацию после ошибки. Если повторно вводимая строка содержит в двух первых лексемах ошибки, они удаляются без выдачи сообщения, что совершенно неприемлемо. По этой причине существует механизм, заставляющий распознаватель поверить в то, что все последствия ошибки устранены. Оператор yyerrok, употребляемый в действии, переводит распознаватель в нормальное состояние. Перепишем последний пример:

```
input: error'0
              { yyerrok;
                printf("Введите строку:"); }

                input
                {$$=$4;}
                ;
```

Как уже было сказано, лексема, прочитанная непосредственно после символа error, считается лексемой, содержащей ошибку. Иногда это неприемлемо, например, действие по обработке ошибок само может определить место возобновления обработки. В этом случае очередная лексема должна очищаться. Этого эффекта можно достичь, применяя в действии оператор yyclearin. Предположим, например, что действие после ошибки заключается в вызове написанной пользователем функции, пытающейся синхронизироваться и установить указатель на начало следующего правильного оператора. После ее вызова следующая возвращаемая yylex лексема будет скорее всего, первой лексемой правильного оператора. Старая неправильная лексема должна быть отброшена, и состояние ошибки должно быть отменено. Этого можно достичь следующими правилами:

```
stat: error
            { resynch();
              yyerrok;
              yyclearin; }
```

Нужно признать, что эти механизмы довольно негибки, но они в действительности позволяют выполнить эффективное восстановление после большинства ошибок. Более того, пользователь может получить управление для обработки ошибок другими участками программы.

Среда выполнения YACC

Если на вход yacc подать спецификацию, на выходе получается файл с программой на языке C, чаще всего называемый y.tab.c. В нем содержится функция, возвращающая целое, по имени yyparse(). Для получения входных лексем эта функция вызывает функцию *лексического анализатора* yyerror(). Далее, либо будет обнаружена ошибка и в этом случае (если не задано действий по обработке ошибок) yyparse() вернет 1, либо лексический анализатор вернет конечный маркер и распознаватель завершит обработку возвратом 0. Для получения работающей программы пользователь должен снабдить распознаватель некоторой средой выполнения. Например, как у любой программы на C должна существовать функция main, всегда вызывающая yyparse(). Далее, для печати сообщений об ошибках должна вызываться функция yyerror().

Эти функции в той или иной форме должны задаваться пользователями. Для облегчения этой задачи существует библиотека, содержащая версии по умолчанию для функций main() и yyerror(). Имя библиотеки зависит от системы, во многих системах она задается флагом -ly компоновщика. Эти программы очень просты, их текст приведен ниже:

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

и

```
#include <stdio.h>
yyerror(s) char *s; {
fprintf(stderr, "%s0, s);
}
```

Аргументом функции yyerror() служит строка, содержащая сообщение об ошибке. Прикладная программа наверняка должна выводить некоторую конкретную фразу. Обычно отслеживаются номера строк и при ошибке выводится номер строки ошибочного оператора. Во внешней переменной yychar хранится номер очередной лексемы в момент обнаружения ошибки. Это может помочь при выдаче полезной диагностики. Так как функция main() обычно задается пользователем (для обработки аргументов и пр.), библиотека yacc полезна либо для небольших проектов, либо на ранних стадиях разработки. Внешняя переменная yydebug первоначально равна 0. Если ее значение отлично от 0, распознаватель будет подробно сообщать обо всех действиях, включая информацию о прочитанных лексемах и выполняемых операциях. В некоторых операционных системах эту переменную можно устанавливать с помощью отладчика.

Подготовка спецификаций

Этот раздел содержит рекомендации по подготовке эффективных, легко изменяемых и читаемых спецификаций. Следующие подразделы более или менее независимы.

Стиль записи

Довольно трудно добиться правильных и одновременно хорошо читаемых спецификаций. Поэтому постарайтесь соблюдать приводимые ниже соглашения:

1. Для имен *лексем* используйте прописные буквы, для имен нетерминалов - строчные. Это правило поможет вам быстро найти виновника ошибки.
2. Действия и правила размещайте на отдельных строках. Это позволит вам менять их независимо.
3. Располагайте правила с одинаковой левой частью вместе. Левую часть записывайте только один раз, а правые части отделяйте вертикальной чертой.
4. Точку с запятой помещайте на отдельной строке и только после последнего правила с заданной левой частью. Это позволит легко добавлять новые правила.
5. Для тела правил делайте отступ в 2 табуляции, для тела действий - в 3.

Приведенные в этом разделе примеры следуют этому стилю (там, где позволяет пространство). У пользователя могут быть свои идеи по поводу стиля оформления, однако, в любом случае, нужно стремиться к тому, чтобы отделить правила от действий.

Левая рекурсия

Алгоритм, применяемый распознавателем, стимулирует употребление так называемой леворекурсивной формы:

```
name: name rest_of_rule
```

Эти правила часто употребляются при обработке последовательностей и списков:

```
list: item
            | list ',' item
            ;
```

и

```
seq: item
             | seq item
             ;
```

В каждом из этих правил первое правило сворачивается только для первого элемента, второе применяется ко второму и прочим элементам. В случае правой рекурсии

```
seq: item
            | item seq
            ;
```

распознаватель будет больше по размеру и свертка будет производиться справа налево. Что более серьезно, существует опасность переполнения внутреннего стека распознавателя при чтении слишком длинных входных последовательностей. Поэтому при возможности следует использовать левую рекурсию. Имеет смысл рассмотреть последовательность нулевой длины: если она допустима, запишите пустое правило:

```
seq:        /* empty */
            | seq item
            ;
```

Как и ранее, первое правило сворачивается только один раз перед чтением первого элемента, второе правило сворачивается для каждого прочитанного элемента. Допущение пустых последовательностей обычно увеличивает универсальность. Однако, могут возникнуть конфликты при попытке определить, какого типа читаемая пустая последовательность!

Взаимодействие с лексическим анализатором

Выполнение некоторых лексических действия зависит от контекста. Например, обычно лексический анализатор должен удалять пробелы, но не делать этого в случаях строк, заключенных в кавычки. Имена должны заноситься в таблицу символов, если они встречаются в определениях, но не в выражениях. Одним из способов решения данной задачи служит использование глобального флага, читаемого лексическим анализатором и устанавливаемого действиями. Предположим, например, что программа состоит из 0 или более определений, за которыми следует 0 или более операторов.

```
%{
int dflag
%}
%%
prog: decls stats
                ;
decls: /* пусто */
                { dflag=1; }
                | определения
                ;
stats: /* пусто */
                { dflag=0; }
                | операторы
                ;
```

Флаг dflag равен 0 при чтении операторов и 1 при чтении определений, за исключением первой лексемы первого оператора. Распознаватель должен распознать эту лексему перед тем, как сообщить, что кончился раздел определений и начался раздел операторов. В большинстве случаев это единственное исключение не влияет на процесс лексического анализа. Используя такой подход, можно, конечно и переусердствовать. Тем не менее, это иллюстрирует достижение определенных целей, которые трудно достижимы другими способами.

Обработка зарезервированных слов

Некоторые языки программирования допускают использование таких слов, как if в качестве меток или имен переменных, если это не противоречит другим именам в языке. В контексте уасс очень трудно обрабатывать подобные ситуации: лексический анализатор должен получать информацию такого рода - "это вхождение if переменная, а это - ключевое слово". Можно попытаться написать такие программы, но это довольно сложно. Лучше считать, что ключевые слова зарезервированы, и их нельзя использовать для имен переменных.

Эмуляция ошибок и конца ввода

Действия по обработке ошибок и концу ввода могут быть смоделированы в действиях с помощью макросов YYACCEPT и YYERROR. Первый заставлет функцию yyparse() вернуть значение 0, второй заставляет распознаватель вести себя так, как если бы он обнаружил ошибку: вызывается yyerror() и производится восстановление после ошибки. Эти механизмы могут использоваться для моделирования распознавателей с несколькими конечными маркерами и контекстно-чувствительным синтаксисом.

Доступ к значениям охватывающих правил

Действию может понадобиться доступ к значениям, возвращаемым действиями, находящимися слева от текущего правила. Здесь используется такой же механизм, как и в случае обычных действий, знак $ и число, но в этом случае число может быть меньше или равно нулю. Например:

```
sent: adj noun verb adj noun
                { анализ предложения } ;
adj: THE {$$=THE;}
                |YOUNG {$$=YOUNG;}
              ...
                ;
    noun: DOG {$$=DOG;}
                |CRONE {if($0==YOUNG){
                     printf("what\n");
                   }
                   $$=CRONE;
                }
                ;
              ...
```

В действии, идущем после слова CRONE, проверяется, равна ли предыдущая лексема слову YOUNG. Очевидно, что это возможно только в том случае, когда точно известно, что может предшествовать слову noun во входном потоке. К тому же, подход явно неструктурный. Тем не менее, иногда он может сэкономить немало усилий, в особенности, когда нужно сделать несколько исключений из регулярной структуры.

Значения произвольных типов

По умолчанию, значения, возвращаемые действиями и лексическим анализатором, считаются целыми. Yacc также поддерживает значения и других типов, в том числе структурных. В дополнение к этому, yacc следит за значениями типов и подставляет при необходимости соответствующие имена полей объединений, что приводит к получению распознавателей, в которых соблюдается контроль типов. Стек в yacc объявлен как объединение, допускающее значения различных типов. Объединение определяется пользователем и каждое его поле связывается с лексемой или нетерминалом. При обращении к значению при помощи конструкций $$ или $n yacc автоматически вставляет соответствующее имя объединения, чтобы избежать неверных конструкций приведения. Это также снижает число диагностических сообщений верификатора программ.

Для обеспечения подобной типизации существуют три механизма. Во-первых, задание объединения, которое должно выполняться пользователем, так как ряд программ, в особенности лексический анализатор, должны знать

имена полей. Во-вторых, существует способ связи имен полей объединения с и лексемами и нетерминалами. И наконец, есть механизм описания типов того небольшого количества значений, для которых уасс не может определить тип. Объединение определяется в разделе объявлений:

```
%union          {
                  тело объединения
                }
```

Это приводит к тому, что стек значений и внешние переменные yyval и yylval будут иметь тип этого объединения. Если уасс запущен с флагом -d, определение объединения копируется в файл y.tab.h. В другом варианте объединение может определяться в макрофайле, а для ссылок используется переменная YYSTYPE, задаваемая оператором typedef. В макрофайле должны помещаться следующие строки:

```
typedef union   {
                  тело определения
                } YYSTYPE
```

Макрофайл включается в раздел объявлений с помощью конструкции %{ %}. Если YYSTYPE определено, все имена полей объединения должны быть связаны с именами терминальных и нетерминальных символов. Для идентификации поля объединения используется конструкция

```
< name >
```

Если она помещается непосредственно после директив %left, %right, %token или %noassoc, имя поля связывается с именем описанной лексемы. Таким образом, строка

```
%left <optype> '+' '-'
```

помечает любую ссылку на значения, возвращаемые этими лексемами, именем поля объединения optype. Для связи имен полей и нетерминалов также применяется директива %type. Поэтому можно написать следующую строку:

```
%type <nodetype> expr stat
```

Существует ряд ситуаций, в которых этого механизма недостаточно. Если правило содержит внутри себя действие, тип возвращаемого этим действием

значения заранее не определен. Аналогично, обращение к левому контексту затрудняет определение типа. В этом случае ссылке можно присвоить некоторый тип путем указания имени поля объединения между символами < и > сразу после первого символа $. Пример:

```
rule: aaa {$<intval>$=3;}bbb {fun($<intval>2,$<other>0);}
;
```

Вряд ли можно пропагандировать эту конструкцию, но к счастью, она нужна двольно редко.

В следующем разделе приводится пример спецификации. Описанные в этом подразделе средства не активны пока к ним не было обращений. Этот механизм, в частности, включается конструкциями %type. При их применении можно достигнуть достаточно серьезного уровня контроля типов. Например, отмечаются диагностическими сообщениями все обращения посредством $ к объектам неопределенного типа. Если эти средства не используются, стек значений содержит значения целого типа, как это было в первых версиях программы.

Простой настольный калькулятор

Приводимый ниже пример содержит полную спецификацию простой программы-калькулятора. Калькулятор содержит 26 регистров, называемых буквами от a до z, и воспринимает арифметические выражения, составленные из операций +, -, *, /, % (остаток), & (побитовое И), | (побитовое ИЛИ), = (присваивание). Если операция верхнего уровня есть присваивание, значение не выводится. Как и в Си, числа, начинающиеся с 0, считаются восьмеричными, в противном случае - десятичными. Данный пример спецификации демонстрирует применение предшествования и неоднозначности, простой способ восстановления после ошибки. Существенное упрощение заключается в том, что фаза лексического анализа гораздо проще, чем в большинстве программ, а вывод генерируется построчно. Обратите внимание на способ ввода десятичных и восьмеричных чисел. Такого рода действия лучше делать в *лексическом анализаторе*.

```
%{
#include <stdio.h>
#include <ctype.h>

int regs[26];
int base;
%}

%start list
%token DIGIT LETTER %left '|'
```

```
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS

%%          /* раздел правил */
list :
          |  list stat '\n'
          |  list error '\n'
              { yyerrok;}
          ;
stat :    expr
              { printf("%d\n", $1); }
          |  LETTER '=' expr
                        { regs[$1] = $3;} ;
expr :    '(' expr ')'
              { $$=$2;}
          |  expr '+' expr
              { $$=$1+$3;}
          |  expr '-' expr
              { $$=$1-$3;}
          |  expr '*' expr
              { $$=$1*$3;}
          |  expr '/' expr
              { $$=$1/$3;}
          |  expr '%' expr
              {$$=$1%$3;}
          |  expr '&' expr
              {$$=$1&$3;}
          |  expr '|' expr
              {$$=$1|$3;}
          |  '-'expr%prec UMINUS
              {$$=-$2;}
          |  LETTER
              {$$=regs[$1];}
          |  number
          ;
```

```
number :  DIGIT {$$=$1;base=($1==0)|8:10;}
          | number DIGIT {$$=base*$1+$2;}
          ;
%%          /* начало программы */

yylex() {

/* Программа лексического анализа возвращает LETTER для
строчной буквы, yylval - от 0 до 25, возвращает DIGIT для
цифры, yylval -от 0 до 9, все остальные символы возвращаются
непосредственно */
int c;

while ((c = getchar()) == ' ') {

   /* пропуск пробелов */
               ;
        if (islower(c)) {
                  yylval = c-'a'; return (LETTER);
        }
        if (isdigit(C)) {
                  yylval = c-'0'; return(DIGIT);
        }
        return(c);
}
```

Описание входного синтаксиса

В этом разделе приведено описание синтаксиса yacc. Зависимость от контекста не рассматривается. Грамматика yacc лучше всего может быть описана как LR(2), основные трудности начинаются при обнаружении идентификатора в правиле, идущем непосредственно после действия. Если после идентификатора стоит двоеточие, это начало следующего правила, иначе это продолжение текущего правила, которое содержит встроенное действие. Реализован алгоритм, при котором лексический анализатор делает опережающий просмотр в поисках идентификатора и определяет, является ли следующая *лексема* двоеточием. Если да, то возвращается значение C_IDENTIFIER. Иначе возвращается IDENTIFIER. Литералы всегда возвращаются как IDENTIFIER и никогда как часть C_IDENTIFIER.

```
/*  grammar  for  the  input  to  Yacc  */
/*  basic  entities  */
```

```
%token IDENTIFIER    /*includes identifiers and literals
*/
%token C_IDENTIFIER    /*identifier (but not literal)
followed by colon    */
%token NUMBER         /*  [0-9]+  */
             /*reserved  words: %type => TYPE, %left =>
LEFT,etc.*/
%token LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION
%token MARK          /*  the  %%  mark  */
%token LCURL         /*  the  %{  mark  */
%token RCURL         /*  the  %}  mark  */
            /*  ascii  character  literals  stand  for
themselves  */
% start    spec
%%
spec :   defs  MARK  rules  tail
         ;
tail :   MARK  {In this action, eat up the rest of the file}
         |    /*  empty:  the  second  MARK  is  optional  */
         ;
defs :   /*  empty  */
         |     defs  def
         ;
def  :   START  IDENTIFIER
         |     UNION  {  Copy union  definition  to  output
}
         |     LCURL  {  Copy  C  code  to  output  file   }
RCURL
         |     ndefs  rword  tag  nlist
         ;
rword :  TOKEN
         |     LEFT
         |     RIGHT
         |     NONASSOC
         |     TYPE
         ;
         tag   :      /*  empty:  union  tag  is  optional
*/
```

```
                |   '<'  IDENTIFIER  '>'
                ;
        nlist :  nmno
                |    nlist  nmno
                |    nlist  ','  nmno
                ;
        nmno  :  IDENTIFIER              /*  NOTE:  literal
illegal  with  %type  */
                |   IDENTIFIER NUMBER   /*  NOTE:  illegal
with  %type  */
                ;
                /*  rules  section  */
        rules :  C_IDENTIFIER  rbody  prec
                |    rules  rule
                ;
        rule  :  C_IDENTIFIER  rbody  prec
                |    '|'  rbody  prec
                ;
        rbody :  /*  empty  */
                |    rbody  IDENTIFIER
                |    rbody  act
                ;
        act   :  '{'  {  Copy  action,  translate  $$,  etc.
}  '}'
                ;
        prec  :  /*  empty  */
                |    PREC  IDENTIFIER
                |    PREC  IDENTIFIER  act
                |    prec  ';'
                ;
```

Устаревшие конструкции

В этом разделе перечислены синонимы и конструкции, которые поддерживаются только для совместимости. Их использование не поощряется.

1. Литералы могту заключаться в апострофы. *Литералы* могут состоять из нескольких символов. Если литерал состоит из букв, цифр и

подчеркивания, его тип определяется, какесли бы он был заключен в кавычки. В противном случае определить его тип затруднительно. Использование многосимвольных литералов обескураживает начинающих пользователей, так как уасс выполняет работу лексического анализатора.
2. Везде, где можно использовать %, допустим символ \. В частности, \\ означает то же, что и %%.
3. Существует еще ряд синонимов: (%< - %left, %> %right, %binary - %noassoc, %term = %token, %= %prec)
4. Действия могут записываться в виде ={...}
5. Фигурные скобки могут опускаться, если действие записывается как один оператор Си.
6. Фрагменты программ между %{ %} можно помещать как в разделе правил, так и в разделе объявлений.