ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗИРОВАННЫХ СИСТЕМ»

Синтаксический анализ транслируемой программы

Методические указания

Волгоград
2009

УДК 004.4'413

Р е ц е н з е н т

д-р техн. наук, профессор *А. В. Заболеева-Зотова*

Издается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Синтаксический анализ транслируемой программы : метод. указания / сост. О. А. Сычев ; ВолгГТУ. – Волгоград, 2009. – 36 с.

Изложены материалы по использованию генератора парсеров bison для разработки синтаксических анализаторов. Описана методика выполнения работы, приведены советы по построению грамматики, описанию структур данных и составлению синтаксического дерева, организации связи между лексическим и синтаксическим анализаторами. Особое внимание уделено способам поиска и устранения конфликтов в грамматике. Предлагается список контрольных вопросов. Методические указания предназначены для студентов специальности 230105.65 «Программное обеспечение вычислительной техники и автоматизированных систем» по дисциплине «Теория языков программирования и методы трансляции».

1 Цель работы

Целью работы является изучение современных средств автоматизации разработки синтаксических анализаторов и разработка собственного синтаксического анализатора с помощью генератора bison.

2 Задание

Необходимо разработать синтаксический анализатор для выбранного языка программирования с помощью генератора синтаксических анализаторов bison.

3 Основы LR-разбора

LR-разбор осуществляется с помощью конечного автомата со стеком и входной последовательностью данных. На вход автомата поступают терминальные символы разбираемого текста. Автомат может выполнить два рода действий:

- сдвиг — текущий терминальный символ помещается на вершину стека, для анализа выбирается следующий терминальный символ;
- свертка — один или несколько символов на вершине стека образуют последовательность, соответствующую правилу грамматики, они «сворачиваются» (т.е. заменяются) соответствующим нетерминальным символом.

Важное значение имеет тот факт, что сворачиваемая последовательность должна находится на вершине стека. Автомат не может свернуть символ (последовательность) в глубине стека — понимание этого поможет вам избежать ошибок в грамматике.

Автомат также меняет свои состояния. Состояние указывает, в каком месте правила мы в данный момент находимся и какие действия возможны сейчас (в зависимости от следующих входных символов) для того, чтобы разбор был успешен.

Цифра в скобках после LR означает количество входных символов, которые автомат должен знать чтобы принять решение, как поступить в данной ситуации (т.е. осуществить сдвиг или свертку, и если свертку, то по какому конкретно правилу). Наиболее распространенный LR(1)-разбор означает, что такое решение должно быть принято зная только один следующий символ. Это следует принимать во внимание при составлении грамматики.

4 Генератор синтаксических анализаторов bison

4.1 Введение

Генератор синтаксических анализаторов Bison предназначен для автоматизации генерации кода синтаксических анализаторов (парсеров) по грамматике языка. Его использование существенно упрощает задачу разработки синтаксического анализатора, поскольку он берет на себя всю наиболее сложную часть анализатора — алгоритм разбора по контекстно-свободной грамматике. Bison может обрабатывать LR(1)-грамматики, хотя наилучшие результаты показывает на LALR(1)-грамматиках.

Входом bison является входной файл (обычно имеющий расширение `y`), содержащий описание грамматики и действий, выполняемых при срабатывании правил грамматики. Результатом работы bison является текст программы на языке Си, определяющий функцию `yyparse`, выполняющую синтаксический анализ входного текста. Этот код может быть скомпилирован и скомпонован с кодом лексического анализатора для получения исполняемого файла.

Входной файл состоит из нескольких секций:

```
%{
Пролог
%}
Секция объявлений
%%
Секция правил грамматики
%%
Секция пользовательского кода
```

Пролог содержит код на языке Си, который должен быть помещен в начале файла с синтаксическим анализатором. Обычно это подключение заголовочных файлов (например для используемых стандартных библиотек), а также объявление внешних (external) переменных для связи с лексическим анализатором и остальной программой.

Секция объявлений содержит объявления символов грамматики и описание их свойств (семантическое значение, приоритет и т.д.). Секция правил содержит описание грамматики (в слегка измененной форме Бэкуса-Науэра) и действий на языке Си, выполняемых при срабатывании этих правил. Секция пользовательского кода содержит необходимые дополнительные функции, которые будут добавлены в конец файла анализатора. Обычно здесь описывается функция `main` (другие необходимые функции, например функции вывода на экран или в файл синтаксического дерева, обычно размещают в отдельных файлах).

Символ грамматики (symbol, не следует путать с более традиционным понятием символа, описываемого английским словом character) является единицей грамматики и грамматического правила. Символы делятся

на терминальные (также называются терминалами или токенами, token) и нетерминальные. Терминальные символы являются простейшими символами грамматики, входом синтаксического анализатора является последовательность терминальных символов, генерируемая лексическим анализатором. Нетерминальные символы образуются путем группировки терминальных и нетерминальных по правилам грамматики. Правило грамматики описывает способ группировки терминальных и нетерминальных символов для получения нового нетерминального символа.

Терминальные символы могут описываться в bison несколькими способами:

– Именованные терминалы — терминальные символы, записываемые словом, подобным имени переменной или функции в языке программирования (состоит из латинских букв, цифр и знаков подчеркивания). Традиционно, чтобы отличить от нетерминальных символов, имена записываются заглавными буквами. Именованные терминалы должны быть объявлены в секции объявлений с помощью директивы `%token`, например

```
%token FOR
%token ID
```

– Символьные терминалы — если терминальный символ грамматики представляет собой один символ (character) входного текста, то его принято записывать как символьную константу языка Си, в одинарных кавычках, например `'*'`. Символьные терминалы не требуют объявления. Не следует употреблять в качестве символьного терминала символы перевода строки и символ с нулевым кодом.

– Строковые литеральные терминалы — в качестве имени терминального символа могут использоваться также строковые константы языка Си, однако при их использовании взаимодействие с лексическим анализатором становится существенно сложнее, поэтому рекомендуется избегать их.

Нетерминальные символы могут быть только именованными, имена традиционно записываются строчными буквами. Они не требуют специального объявления если не требуется указать их дополнительные параметры.

Один из нетерминальных символов грамматики является особым — в него должен свернуться весь исходный текст. Такой символ называют стартовым. По умолчанию bison считает стартовым символ, стоящий в левой части первого правила грамматики, вы можете изменить это с помощью директивы `%start` в секции объявлений.

4.2 Семантические значения символов

Имя символа определяет его тип — например для терминального символа это тип лексемы. Иногда этого хватает — например, если лексема описывает ключевое слово языка или знак операции. Однако во многих случаях для успешной трансляции знать только тип символа недостаточно: если лексема описывает целочисленную константу, то для успешной компиляции необходимо знать также, какое именно число было введено в программе. Аналогично, лексеме, описывающей идентификатор, требуется знать какое, собственно, имя задано. Таким образом, сам символ описывает тип того фрагмента исходного текста, который он описывает. Для обработки более конкретного значения этого фрагмента (чему равно целое число?) в bison с каждым символом грамматики во время разбора связывается семантическое значение.

Семантические значения, как правило, имеют разный тип: для идентификатора это будет строка, для целой константы — число. Целью синтаксического анализатора является построение синтаксического дерева программы, которое на языке Си обычно представляется в виде набора структур (описывающих узлы дерева), ссылающихся друг на друга. Например для описания цикла `while` может применяться следующая структура:

```
struct NWhile {
   struct NExpr * condition;//условие продолжения
                            //цикла
   struct NStmt * body;//тело цикла
};
```

Указатели на подобные структуры являются семантическими значениями нетерминальных символов грамматики. В этом случае семантическим значением стартового символа грамматики будет указатель на структуру — корень дерева, пользуясь которым можно обойти его. Объявления всех необходимых структур обычно размещают в отдельном заголовочном файле, например `tree_nodes.h`

Bison использует один тип семантического значения для всех символов грамматики. Следовательно, если для разных символов требуются разные типы семантических значений, то необходимо использовать объединение (union) их как тип данных семантического значения анализатора. Такое объединение может быть задано с помощью директивы `%union` в секции объявлений

```
%union {
/*объявления всех необходимых типов семантиче-
ских значений*/
}
```

Обратите внимание, что, в отличие от использования ключевого слова union языка Си, точка с запятой в конце директивы %union не ставится.

Если семантическое значение задано объединением, то при объявлении терминальных и нетерминальных символов необходимо указать, какое поле объединения будет использоваться каждым символом. Это делается в угловых скобках перед именем символа (для нетерминальных символов используется директива %type). Например:

```
%union {
int Int;
char *Id;
struct NWhile * While;
struct NExpr * Expr;
}
%token <Int> INT
%token <Id> ID
%type <While> while_loop
%type <Expr> expr
```

Обратите внимание, что в угловых скобках указано имя поля (а не его тип), а структура NWhile содержит значение для нетерминала while_loop (представляющего полное описание цикла в программе), а не терминала WHILE (представляющего ключевое слово while), которому семантическое значение не требуется т.к. он имеет только один вариант написания.

4.3 Правила грамматики и действия по ним

Правила грамматики задаются следующим образом. В начале строки записывается результирующий нетерминальный символ, потом ставится двоеточие, после которого через пробел перечисляются символы правой части правила, заканчивается оно точкой с запятой. Если существует несколько правил с одним нетерминальным символом в левой части, то варианты правой части могут быть записаны через символ вертикальной черты, при этом каждый вариант необходимо писать с новой строки. Например правила для условного оператора языка Си:

```
if_stmt: IF '(' expr ')' stmt
      | IF '(' expr ')' stmt ELSE stmt
;
```

В этом правиле участвуют именованные терминалы IF и ELSE, символьные '(' и ')' и нетерминалы stmt (оператор) и expr (выражение).

Для каждого правила записываются действия, выполняемые анализатором при срабатывании этого правила (свертки по правилу) в виде кода

программы на языке Си. Эти действия записываются в фигурных скобках после правила, если правило содержит несколько компонент, записанных через вертикальную черту, то для каждой компоненты пишется отдельное действие (поэтому каждая компонента начинается с новой строки).

Основная цель действия состоит в вычислении семантического значения нетерминала в левой части правила (результата свертки) по семантическим значениям, находящимся справа. При этом семантическое значение результирующего терминала записывается в коде программы как `$$`, а значения символов правой части как `$1`, `$2` и т.д. (нумерация с единицы). Типы этих обозначений соответствуют типам полей объединения, указанным при объявлении соответствующего символа. Например, если ваш анализатор просто вычисляет значение выражения, состоящего из констант (без построения дерева), то возможно следующее правило (семантическое значение `expr` - `int`):

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

При написании реального синтаксического анализатора семантическим значением будет являться указатель на вершину синтаксического дерева. Соответственно, основной целью действия в большинстве случаев является динамическое выделение памяти под соответствующую структуру и заполнение ее полей. Чтобы не загромождать грамматику, настоятельно рекомендуется выносить такие действия в отдельные функции (размещаемые в заголовочном файле, либо секции пользовательского кода), а в секции правил грамматики записывать только вызов функции. Например для показанной выше структуры `NWhile` необходимо объявить следующую функцию, которая динамически создаст структуру и заполнит ее поля:

```
struct NWhile * CreateWhile
    (struct NExpr * expr, struct NStmt * stmt);
```

Соответственно правило с действием будут выглядеть следующим образом:

```
while_loop: WHILE '(' expr ')' stmt
    {$$=CreateWhile($3,$5); }
;
```

Подробнее создание и заполнение структур при разработке синтаксического анализатора рассматривается далее.

Специальный терминальный символ `error` позволяет создавать правила обработки ошибок. В их отсутствие парсер прекратит разбор после первого же обнаруженного нарушения грамматики. Символ `error` позволяет указать, с какого места и как можно продолжить разбор текста программы, чтобы найти последующие ошибки. Например в языке Си операторы завершаются точкой с запятой. Если в каком-либо операторе встретилась ошибка, будет логично продолжить разбор со следующего операто-

ра, для чего применяется правило (stmt — нетерминальный символ, обозначающий оператор языка):

```
stmt: error ';'
;
```

4.4 Связь лексического и синтаксического анализаторов

Свои входные данные синтаксический анализатор (парсер) получает от лексического (сканера). Входом является поток лексем, то есть сканер должен сообщить тип лексемы и, при необходимости, ее семантическое значение.

Для этого сканеру прежде всего нужна информация о возможных лексемах (терминальных символах), а также типе семантического значения. Для этого при обработке исходного файла парсера необходимо запустить bison с ключом -d, например `bison -d mygrammar.y`

В этом случае bison генерирует дополнительный заголовочный файл с объявлениями терминальных символов и объединения для семантического значения. Этот файл следует подключить к файлам исходного кода обоих анализаторов.

Когда парсеру требуется узнать очередную лексему он вызывает функцию `int yylex()` (обычно генерируемую с помощью инструментальных средств, например flex). Эта функция сообщает данные о лексеме следующим образом. Тип лексемы является возвращаемым значением функции. При этом именованные терминалы объявляются в автоматически сгенерированном заголовочном файле как макросы, подставляющие целые числа (отсюда традиция писать их большими буквами), а при возвращении символьного терминала возвращается символьная константа, которая приводится к целому типу автоматически (при использовании символов с кодом более 127 следует учитывать, что во многих компиляторах тип `char` по умолчанию является знаковым, bison же ожидает положительные коды лексем, поэтому при возвращении такого символа необходимо явно приводить тип к `unsigned char`). Перед возвратом кода лексемы ее семантическое значение (если требуется) записывается в глобальную переменную `yylval`, являющуюся объединением (также объявленную в заголовочном файле, созданном bison'ом). Например при использовании объявленного выше объединения, лексический анализатор может вести себя следующим образом (приводится фрагмент исходного файла для flex)

```
while {return WHILE;}
\( {return '(';}
[1-9][0-9]+ {yylval.Int=atoi(yytext); return
INT;}
```

Соответственно, функция `main` программы, осуществляющей синтаксический анализ, будет (после направления потока `yyin` в необходи-

мый файл) вызывать функцию `yyparse` (а не `yylex`). После завершения работы `yyparse`, обычно, в специальной глобальной переменной (например `root`) оказывается сохранен корень синтаксического дерева, поэтому далее могут вызываться функции для последующей его обработки и/или вывода на экран (в файл).

4.5 Конфликты в грамматике

Если грамматика, введенная в bison, не полностью соответствует требования LR(1)-грамматики, то при выполнении грамматического разбора возникают конфликты. Конфликт означает, что при некоторых входных последовательностях возможны ситуации, где анализатор не в состоянии (просматривая на один символ впереди) однозначно выбрать стратегию поведения: возможны различные варианты действий, которые приведут к разным результатам.

Некоторые конфликты являются нормальной частью грамматик для большинства языков программирования — они достаточно легко обходятся программными настройками, в то время как решение их чисто средствами грамматики является громоздким и неэффективным. Другие конфликты сигнализируют об ошибках в грамматике и, соответственно, сгенерированном анализаторе. Важно научиться различать эти виды конфликтов и работать с ними соответственно.

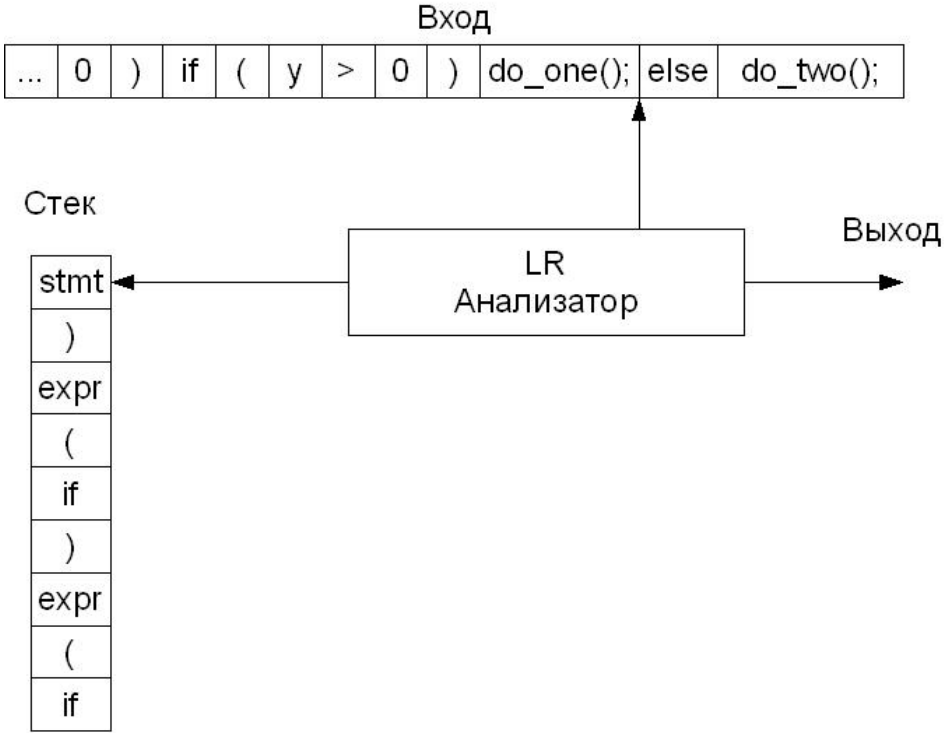


Рисунок 1- Состояние автомата в момент конфликта

При выполнении LR-разбора возможны два типа конфликтов:

1) конфликты сдвиг/свертка — возникают тогда, когда парсер не может выбрать между тем, сдвинуть ли ему очередной символ на стек или произвести предварительно свертку по правилу, при возникновении таких конфликтов непонятно взаимное расположение вершин синтаксического дерева, однако состав вершин известен четко; многие (но не все!) из конфликтов этого типа являются нормальной частью грамматики;

2) конфликты свертка/свертка — возникают, когда в некотором состоянии возможна свертка по нескольким различным правилам, в отличие от конфликтов сдвиг/свертка в этом случае анализатору неясно количество и тип вершин синтаксического дерева - конфликты такого типа обычно свидетельствуют о серьезных проблемах в грамматике.

Одним из типичных примеров конфликтов сдвиг/свертка является так называемый конфликт «болтающегося else» (dangling else). Рассмотрим правила грамматики, задающие условный оператор `if`:

```
if_stmt: IF '(' expr ')' stmt
| IF '(' expr ')' stmt ELSE stmt
;
stmt: if_stmt
........
;
```

Данная грамматика приведет к возникновению конфликта при разборе двух вложенных операторов `if` — в следующем примере непонятно, к какому из условий относится `else`:

```
if(x>0) if(y>0) do_one(); else do_two();
```

Конфликт возникнет когда автомату на вход поступает `else`, а на вершине стека расположено выражение `do_one();` ,свернутое в `stmt` (см. рис. 1).

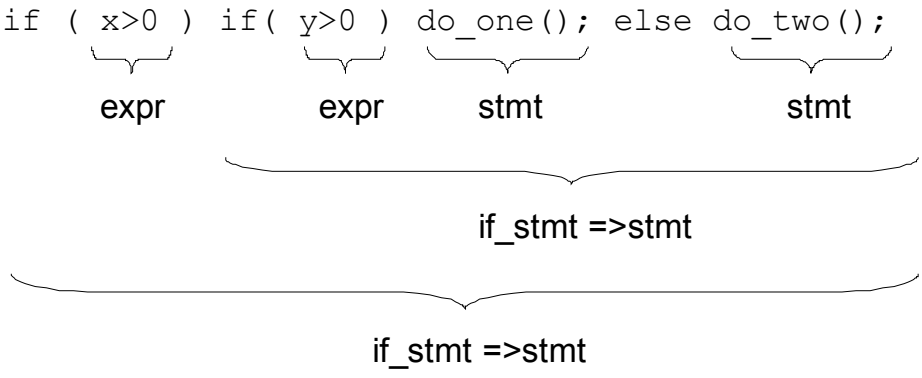
Анализатор может выбрать сдвиг, тогда ему надо будет сначала свернуть по правилу `if...else` оператор `if(y>0)...`, затем свернуть `if(x>0) ....` по правилу без `else` (см. рис. 2 а).

В этом случае вложенность операторов будет понята так:

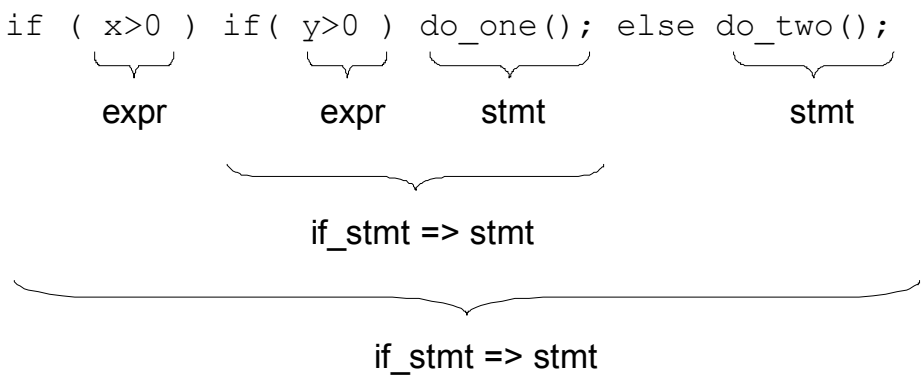
```
if(x>0) {
    if(y>0)
        do_one();
    else
        do_two();
}
```

Однако, не нарушая корректности разбора, анализатор может в этот момент выбрать свертку последовательности `if(y>0) do_one();` по правилу без `else`, после чего свернуть его в `stmt` и свернуть уже

`if(x>0) stmt else do_two();` по правилу `if...else` (см. рис. 2 б).



а)



б)

Рисунок 2 – Свертка выражения

В этом случае код будет понят так:

```
if(x>0)   {
    if(y>0)
        do_one();
} else
    do_two();
```

Таким образом, оба разбора грамматически корректны, но приводят к совершенно разным деревьям и, соответственно, программам. Эта проблема известна с 60-х годов, ее решение формальными методами грамматики является громоздким и неэффективным. В большинстве языков программирования принято в такой последовательности считать правильным первый вариант (т.е. относить `else` в подобной ситуации ко внутреннему условию), поэтому конфликты типа сдвиг/свертка в bison по умолчанию разрешаются в пользу сдвига.

Другой хорошо известный случай конфликтов сдвиг/свертка — это конфликты, возникающие при вычислении выражений. Рассмотрим следующую грамматику:

```
expr: INT
| expr '+' expr
| expr '*' expr
| '(' expr ')'
;
```

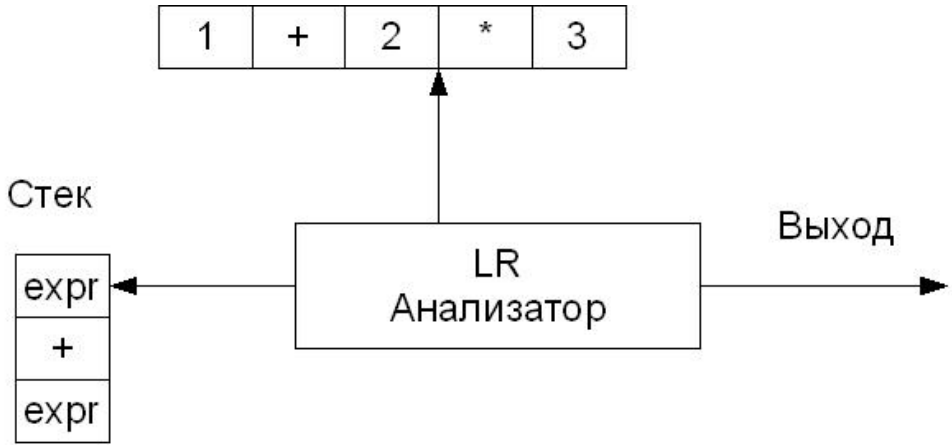


Рисунок 3 – Состояние автомата в момент конфликта

Как видно на рисунке 3, при вычислении выражения 1+2*3 возникает конфликт сдвиг-свертка. В этом случае автомат может выбрать сдвиг умножения, тогда сначала будет произведена свертка операции 2*3, а затем уже прибавление 1 к их произведению (см. рис. 4 а), либо предварительно свернуть 1+2, а затем сдвинуть * и 3, т.е. умножить сумму 1+2 на 3 (см. рис. 4 б).

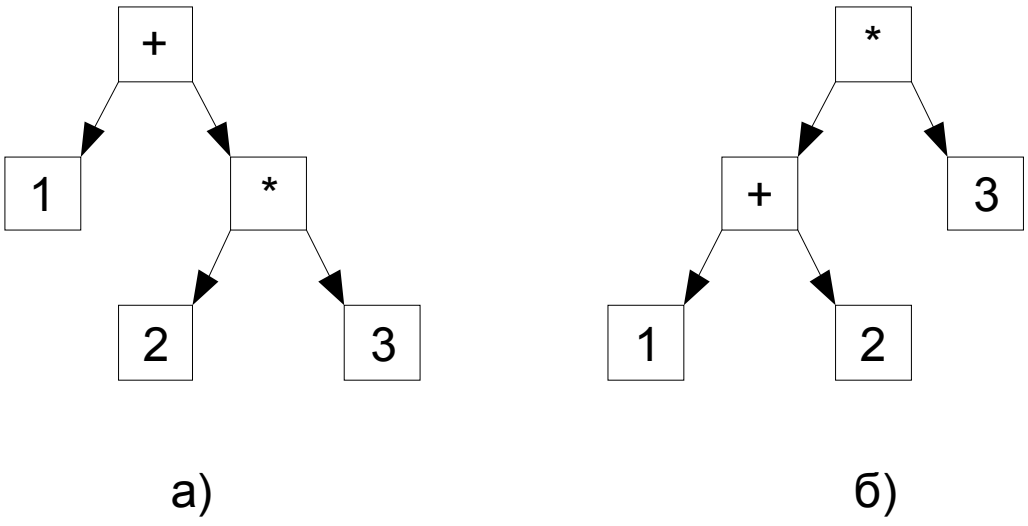


Рисунок 4 – Синтаксические деревья

Очевидно, что подобные случаи нельзя разрешить простым правилом действия по умолчанию, в отличие от случая с `else` — если в приведенном выше примере правильным будет выбрать сдвиг, то для выражения 1*2+3 корректным вариантом будет свертка. Конфликт может быть разрешен средствами грамматики, если для каждого уровня приоритетов операций ввести отдельный нетерминальный символ. Однако, если для приведенного выше примера с тремя уровнями приоритетов (скобки, умножение и сложение) это довольно легко, то в реальных грамматиках это приводит к значительному их усложнению: например язык Си насчитывает 16 (!) уровней приоритета операций. Что значительно хуже, введение новых нетерминальных символов усложняет структуру синтаксического дерева: в худшем случае, каждое (!) число или идентификатор в выражении должны будут пройти по 16 узлам (для каждого нетерминального символа), прежде чем окажутся свернуты в `expr`. Обрабатывать такое дерево крайне затруднительно.

К счастью, если рассматривать ситуацию на уровне конфликтов, проблема решается куда проще: зная приоритеты операций мы можем построить простое правило: если приоритет у левой операции выше, то следует выбрать свертку, если у правой — сдвиг. Такое решение (если анализатору известны приоритеты) позволяет записать все операции выражения с помощью одного нетерминального символа, устраняя все лишние узлы синтаксического дерева. К сведениям о приоритете, однако, необходимо добавить сведения об ассоциативности: она определяет порядок вычисления подряд идущих операций с одинаковым приоритетом, слева-направо (левая ассоциативность) или справа-налево (правая ассоциативность, типичным примером правой ассоциативности является операция присваивания: записанная в цепочку типа `a=b=4` она должна выполняться справа налево — сначала `b=4`, затем `a=b`, но не наоборот!).

Приоритет и ассоциативность может присваиваться терминальным символам грамматики при их объявлении. В этом случае вместо `%token` используется `%left` (левая ассоциативность), `%right` (правая ассоциативность) `%nonassoc` (операция неассоциативна, т.е. не может идти два раза подряд). Приоритет определяется порядком объявления операций: первыми определяются операции с низшим приоритетом (выполняемые последними), последними операции с высшим приоритетом (выполняемые первыми). Если несколько операций имеют одинаковый приоритет, то они перечисляются в одном объявлении через пробел. Пример объявления приоритетов:

```
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc ')'
```

Две операции скобок не могут идти подряд друг за другом без знака арифметической операции между ними, поэтому скобки объявлены как неассоциативная операция. В объявлении использована закрывающая скобка, поскольку приоритет и ассоциативность правила определяются по *последнему* терминальному символу в нем. Правильный выбор приоритетов играет важную роль в существовании языка, поскольку неудачно выбранные приоритеты вынудят программиста перегружать выражения скобками. Примером удачно подобранных приоритетов операций можно считать язык Си, неудачно — Pascal (приоритеты логических операций в Pascal выше чем у операций сравнения, так что каждую операцию сравнения требуется заключать в скобки, то же верно и для соотношения приоритетов между арифметическими операциями и операциями сравнения).

В некоторых случаях приоритет операции может зависеть от правила грамматики, а не только от терминального символа. Наиболее очевидным примером такой ситуации является операция «минус» - существуют две ее формы, причем унарный минус (-5) имеет значительно более высокий приоритет, чем бинарный (3-5). В этом случае следует объявить особый, неиспользуемый терминальный символ для дополнительного приоритета (`UMINUS`) в примере ниже, и указать его в соответствующем правиле с помощью директивы `%prec`:

```
%left '+' '-'
%left '*' '/'
%left UMINUS
........
expr: expr '-' expr
| expr '*' expr
| '-' expr %prec UMINUS
......
;
```

Конфликты типа свертка/свертка означают, что одну и ту же последовательность можно свернуть по нескольким различным правилам — они обычно свидетельствуют о серьезных проблемах в построении грамматики и обязательно должны быть исследованы и разрешены. При запуске bison выдает количество обнаруженных в грамматике конфликтов. Для того, чтобы разобраться в их причине, следует запустить bison с дополнительным ключом --verbose, в этом случае будет сгенерирован файл с подробным описанием синтаксического анализатора (в виде конечного автомата) и указанием конфликтных состояний. Подробнее о разрешении конфликтов рассказывает соответствующий раздел данного методического указания.

4.6 Положения

Если входной текст не соответствует грамматике, то будет выдано сообщение об ошибке. Однако само по себе сообщение о наличии ошибки (даже если в нем будет указан ожидаемый и настоящий символы) мало что скажет вам (или программисту) — крайне желательно снабдить сообщение информацией о месте ошибки в программе — как минимум, номере строки. Получить сообщение о наличии ошибки где-то в большом файле не очень-то приятно, и вполне способно сильно затруднить вам отладку синтаксического анализатора. Для устранения этой проблемы необходимо снабдить bison информацией о положении символов в коде программы. Для подключения работы механизма положений, необходимо указать директиву `%location` в секции объявлений программы.

Положение (location) символа грамматики описывается структурой `YYLTYPE`, состоящей из четырех целых чисел: `first_line`, `first_column`, `last_line` и `last_column` (не забывайте, что нетерминальный символ грамматики может охватывать, например, описание целого класса, занимающее не одну страницу экранного текста). Положения терминальных символов должны быть переданы через переменную `yylloc` (типа `YYLTYPE`), заполненную перед возвращением из функции `yylex` аналогично семантическому значению, для чего необходимо ввести в лексический анализатор счетчики строк и символов в строке. В тексте действий, выполняемых при срабатывании правил грамматики, вы можете ссылаться на положения через символ `@`: `@$` будет обозначать положение результирующего символа правила, `@1` первого символа из левой части и т.д. (например `@1.first_line` дает номер строки начала первого символа левой части правила). Bison способен сам вычислить значение `@$` пользуясь правилом по умолчанию, вам следует упоминать положения в тексте лишь для того, чтобы сохранить необходимую информацию в узлах семантического дерева (например для выдачи сообщений об ошибках в семантическом анализаторе).

5 Методика выполнения работы

1. Изучите язык, для которого вы разрабатываете компилятор, и разработайте LR(1) грамматику языка.
2. По разработанной грамматике создайте скелетный входной файл для Bison, содержащий объявления терминальных символов и правила грамматики (без действий)
3. Запустите bison с созданным файлом и изучите сообщения о конфликтах в грамматике. Если имеются сообщения о неиспользуемых нетерминальных символах, недоступных правилах или конфликтах типа

свертка/свертка, то запустите bison с ключом --verbose и анализируя полученный файл устраните ошибки в грамматике.

4. На основе грамматики разработайте заголовочный файл со структурами, описывающими узлы синтаксического дерева вашего языка и подключите их ко входному файлу bison. Добавьте к грамматике объявления терминальных и нетерминальных символов с указанием типа их семантического значения.

5. Добавьте в конец входного файла bison описания функций заполнения структур, и действия по вызову этих функций при свертке правил грамматики.

6. Переработайте разработанный ранее лексический анализатор для связи с синтаксическим, свяжите файлы анализаторов между собой по внешним переменным (`yylval`, `yylloc`). Запустите bison с ключом -d и подключите к лексическому анализатору полученный заголовочный файл.

7. Перенесите функцию `main` из файла для flex в файл для bison, заменив в ней вызов функции `yylex` на `yyparse`.

8. Напишите функцию, выводящую построенное анализатором дерево на экран или в файл (предпочтительно в XML-формате) и вставьте ее вызов в функцию `main` после `yyparse`.

9. Скомпилируйте полученную программу, проверьте ее работоспособность на тестовом примере. При необходимости отладьте анализатор.

6 Разработка грамматики

При описании грамматики языка программирования обычно используются три вида правил (совокупностей правил):

1) обычные правила (применяются например для описания различных операторов);

2) рекурсивные правила, использующиеся для описания последовательностей (например последовательность операторов, составляющих тело функции, последовательность аргументов функции, последовательность описаний функций в программе);

3) рекурсивные правила, использующиеся для создания иерархических структур (например выражения с учетом приоритета операций).

Мы будем рассматривать их особенности в порядке использования в языке снизу вверх: вычисление выражений, описание последовательностей, описание операторов.

6.1 Выражения

Выражением (expression) называется совокупность операций (operator) и их операндов (operand), которое вычисляется в соответствии с приоритетами и ассоциативностью операций. Вычисление выражений все-

гда дает определенный результат (в противоположность операторам (statement), результатов не имеющих); в некоторых случаях выражение может также иметь побочный эффект (side effect) - т.е. приводить к изменению значений в памяти. Операция характеризуется своей арностью (унарная, бинарная), т.е. количеством аргументов.

Порядок вычисления выражений определяется их приоритетом (precedence) и ассоциативностью. Если две бинарные операции расположены подряд, первой из них вычисляется та, приоритет которой больше (например умножение имеет приоритет перед сложением). Если операции имеют одинаковый приоритет, то порядок вычислений определяется ассоциативностью этой группы операций: ассоциативность может быть левой (слева направо) и правой (справа налево, например операция присваивания).

Приоритеты и ассоциативность операций должны учитываться при синтаксическом разборе, поскольку порядок вычисления операций отражается в синтаксическом дереве. В качестве примера рассмотрим синтаксические деревья для двух выражений, изображенные на рисунке 4.

Учет приоритетов операций может быть выполнен средствами грамматики, однако это влечет за собой ее значительной усложнение: каждый уровень приоритета требует введения отдельного нетерминального символа грамматики, и любое выражение должно при разборе пройти все эти уровни (как правило многократно, поскольку каждое подвыражение должно, в свою очередь, также их пройти). Поскольку в реальных языках программирования количество уровней приоритета превосходит 10, использование такой техники приводит к получению громоздких, с трудом поддающихся обработке синтаксических деревьев даже для простых программ. Поэтому была разработана техника более простого решения данной проблемы.

Если все операции выражения будут использовать один нетерминальный символ, то при LR-разборе выражения, содержащего две подряд идущие бинарные операции возникнет конфликт типа сдвиг/свертка, решение которого определит порядок их выполнения. Генератор анализаторов может самостоятельно разрешить этот конфликт, если ему известны приоритеты и ассоциативность этих операций. Поэтому при составлении грамматики следует все операции выражения записывать в виде одного рекурсивно определенного нетерминального символа со множеством правил типа `expr: expr '+' expr`. При этом следует учитывать следующие моменты:

1) в синтаксическом дереве элементарные операнды (константы, переменные, вызовы функций) можно также рассматривать как выражения, что приводит к значительному упрощению грамматики (правила типа `expr: ID`);

2) круглые скобки, используемые для управления порядком вычисления, рассматриваются как унарная операция с высоким приоритетом;

3) в некоторых языках присваивание является операцией с побочным эффектом (и правой ассоциативностью), в других присваивание рассматривается как оператор;

4) обращение к элементу массива в С-подобных языках является бинарной операцией, задаваемой правилами вида `expr: expr '[' expr ']'`, где первое выражение вычисляется в массив, второе — в индекс, при этом операция имеет левую ассоциативность, что позволяет использовать ее последовательно для работы с многомерными массивами;

5) операциями являются также знаки, используемые для обращения к элементам класса (точка, стрелочка и т.д.), однако это унарные операции: выражение допускается лишь слева от них, справа должен быть идентификатор (поле класса) или вызов функции (метод класса).

6.2 Последовательности

Последовательности часто встречаются в языках программирования: это и общая последовательность классов в объектно-ориентированной программе, и последовательность объявления членов класса, и последовательность операторов в теле функции, и последовательность аргументов функции в ее заголовке и вызове. Последовательности также описываются в грамматике рекурсивными правилами, однако (в отличие от древовидных структур выражений), лишь один символ в правой части правила является рекурсивным.

При составлении грамматики необходимо учитывать определенные особенности описания последовательностей. Для правильного описания последовательности необходимо ответить на несколько вопросов.

Ставится ли разделитель элементов последовательности после последнего ее элемента? Например в языке С разделителем операторов в последовательности является точка с запятой, и она ставится после последнего элемента, в то время как запятая, разделяющая аргументы функции, ставится только между аргументами. Если разделитель ставится после последнего элемента, то синтаксически удобнее сделать разделитель частью элемента последовательности, тогда в описании самой последовательности он не участвует, например:

```
stmt: expr ';'
;
stmt_seq: stmt
| stmt_seq stmt
;
```

Если разделитель ставится только между элементами последовательности, то он участвует прямо в описании последовательности, например:

```
expr_seq: expr
| expr_seq ',' expr
;
```

Второй важный момент касается последовательностей, которые могут быть пустыми. Наивный способ описания таких последовательностей обычно таков:

```
param_list: /*empty*/
| expr
| param_list ',' expr
;
```

Однако такой способ некорректен: он позволяет последовательности, начинающиеся с запятых, поскольку в третьем варианте правила ничто не запрещает рассматривать `param_list` как пустоту (по первому варианту). Для правильного задания последовательностей, которые могут быть пустыми, необходимо использовать два нетерминальных символа:

```
param_list: /*empty*/
| param_listE
;
param_listE: expr
| param_listE ',' expr
;
```

Рекурсивным символом при описании последовательности обычно бывает самый левый или правый символ в этой части правила, поэтому принято говорить о левой и правой рекурсиях. Любую последовательность можно описать как левой, так и правой рекурсией. При описании грамматики для LR-разбора необходимо использовать левую рекурсию, поскольку она позволяет разбирать последовательности любой длины используя ограниченный стек, в то время как правая рекурсия использует стек пропорционально длине последовательности.

Рассмотрим последовательность `expr_seq`:

Левая рекурсия

```
expr_seq: expr
  | expr_seq ',' expr
;
```

Правая рекурсия

```
expr_seq: expr
  | expr ',' expr_seq
;
```

и реальную последовательность входного файла:

```
5, 4+3, a
```

При использовании леворекурсивного правила, анализатор, сдвинув на стек `5` (сворачиваемую как `expr`), должен свернуть его в `expr_seq` (это единственный путь продолжить разбор), в дальнейшем анализатор сдвигает запятую и `4+3` (сворачиваемые в `expr`), после чего по правилу

сворачивается `expr_seq` (5), запятая и `expr` (4+3). Аналогичным образом происходит свертка остатка последовательности: таким образом, при свертке последовательности (не рассматривая свертку выражений внутри нее) используется только 3 ячейки стека (см. табл. 1).

Таблица 1 – Содержимое стека при разборе последовательности, описанной леворекурсивным правилом

Операция	Стек				
сдвиг					
свертка	5				
свертка	`expr`				
сдвиг	`expr_seq`				
сдвиг	`expr_seq`	`,`			
свертка	`expr_seq`	`,`	4		
сдвиг	`expr_seq`	`,`	`expr`		
сдвиг	`expr_seq`	`,`	`expr`	`+`	
свертка	`expr_seq`	`,`	`expr`	`+`	3
свертка	`expr_seq`	`,`	`expr`	`+`	`expr`
свертка	`expr_seq`	`,`	`expr`		
сдвиг	`expr_seq`				
сдвиг	`expr_seq`	`,`			
свертка	`expr_seq`	`,`	`a`		
свертка	`expr_seq`	`,`	`expr`		
	`expr_seq`				

При использовании правой рекурсии для свертки по правилу `expr ',' expr_seq` необходимо, чтобы `expr_seq` находился на стеке справа от запятой. Поэтому единственный способ успешно разобрать приведенную выше последовательность заключается в том, чтобы свернуть в `expr_seq a`, после чего становится возможной свертка выражения `4+3`,

запятой и полученного `expr_seq` (и т.д.). Для этого перед началом свертки на стек необходимо переместить всю последовательность (просмотр осуществляется слева направо), поэтому количество используемых ячеек стека пропорционально длине последовательности, которая может быть очень большой (например последовательность операторов в большой функции или последовательность классов в большой программе).

6.3 Операторы

Разработка правил написания операторов несложна, если учитывать следующие закономерности:

1) в языках типа C простейшим оператором считается выражение, заканчивающееся точкой с запятой, т.е.

```
stmt: expr ';';.
```

В других языках выражение не считается оператором, вместо этого как оператор рассматривается присваивание.

2) составной оператор обычно также рассматривается как оператор, например для языка C

```
stmt: '{' stmt_seq '}' ;
```

3) при соблюдении описанных выше условий контролирующие операторы описываются просто, например цикл «пока» в языке C:

```
stmt: WHILE '(' expr ')' stmt ;
```

Скобки здесь необходимы (если выражение считается оператором), т.к. `stmt` может начинаться с `expr` (см. 1), и отсутствие скобок вызовет конфликт унарного и бинарного минуса, например при разборе следующего выражения:

```
while 2<i-5+3;
```

Неясно, считать ли условием `2<i` (выражение `-5+3`) или `2<i-5` (выражение `+3`). Обратите внимание также на отсутствие терминального символа точка с запятой в описании цикла: если оператором является выражение, то точка с запятой будет уже свернута как его часть (см 1), в противном случае (при использовании составного оператора) точка с запятой не требуется.

6.4 Пример

Рассмотрим простейший язык программирования, содержащий все перечисленные выше виды правил грамматики. Язык позволяет вычислять арифметические выражения с использованием числовых констант и переменных, а также арифметических операций (+,−,*,/) и операции присваивания (=). Возможны также условные выражения с использованием операций сравнения (==, !=, <, >). Язык имеет три оператора: выражение, `print` (печатает данное выражение) и условие. Операторы разделяются

переводом строк, в условном операторе выражение от списка выполняемых операторов отделяется переводом строки, а признаком завершения является ключевое слово `endif` (на отдельной строке).

Используемые терминальные символы: `INT` (числовая константа), `ID` (имя переменной), `ENDL` (перевод строки), `PRINT`, `IF`, `ENDIF` (ключевые слова), `EQ`, `NE` (для операций сравнения == и !=), `UMINUS` (используется для задания контекстно-зависимого приоритета унарного минуса). Введены следующие нетерминальные символы: `program` (стартовый символ грамматики), `stmt_list` (список операторов), `stmt` (оператор), `expr` (выражение), `print_stmt` (оператор печати), `if_stmt` (условный оператор).

Грамматика будет иметь следующий вид:

```
program : stmt_list
;

stmt_list: stmt
| stmt_list stmt
;

stmt : expr ENDL
| if_stmt
| print_stmt
;

expr : INT
| ID
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| expr '=' expr
| expr '<' expr
| expr '>' expr
| expr EQ expr
| expr NE expr
| '(' expr ')'
| '-' expr %prec UMINUS
;
print_stmt : PRINT expr ENDL
;
if_stmt : IF expr ENDL stmt_list ENDIF ENDL
;
```

7 Структуры данных и их заполнение

7.1 Разработка структур данных

На основании построенной грамматики необходимо разработать структуры данных для хранения вершин синтаксического дерева в памяти. При этом следует исходить из следующих принципов:

1) каждому нетерминальному символу грамматики соответствует своя структура, за исключением некоторых служебных символов, не вводящих новых понятий (например для промежуточного символа, используемого для создания списков, которые могут быть пустыми, структуру описывать не нужно);

2) если правила для описываемого структурой символа содержат терминальные символы с семантическим значениями, то в структуре необходимо объявить поля для хранения этих значений;

3) если правила для описываемого структурой символа содержат нетерминальные символы, то в структуре необходимо объявить поля с указателями на структуры, описывающие эти нетерминальные символы;

4) если символ используется для описания последовательности, то необходимо представлять ее компоненты в виде связного списка или динамически изменяющего размер массива (например шаблона vector при использовании C++); использование статических массивов в дереве крайне нежелательно, так как они занимают много памяти (чтобы могла поместится большая последовательность), а узлов дерева может быть очень много;

5) если для описываемого структурой символа существует несколько правил, по которым может быть произведена свертка (разделенных вертикальной чертой), то в структуру обычно бывает полезно ввести поле (предпочтительно перечислимое), указывающее, какое именно правило сработало. В редких случаях это не требуется, так как ситуацию можно определить по другим полям структуры — например наличие или отсутствие `else` в условном операторе можно определить устанавливая указатель на блок для `else` в 0. Этого также не требуется, если пара (или более) правил применяется для описания последовательности.

Рассмотрим структуры данных, необходимые для хранения синтаксического дерева грамматики, изложенной выше. Служебными нетерминальными символами, не требующими дополнительных структур, в данном случае будут `program` (специальный стартовый символ), а также `print_stmt` и `if_stmt` (все данные по операторам будут сведены в общую структуру для оператора — это увеличивает в объеме структуру (в данном случае незначительно), но упрощает дальнейшую обработку дерева).

Мы имеем два нетерминальных символа с вариантами: `expr` и `stmt`. Зададим для них перечисления:

```
enum ExprType {Int, Id, Plus, Minus, Mul, Div,
As, LT, GT, EQ, NE, Prior, UMinus};
enum StmtType {Expr,If,Print};
```

Структура данных, описывающая выражение, будет состоять из типа выражения, полей для значений терминальных символов (идентификатор, целое число) и двух указателей на выражение, описывающих левый (единственный при унарной операции) и правый операнды если в качестве типа указана операция.

```
struct Expression {
    enum ExprType Type;
    int Num;
    char *Name;
    struct Expression *Left;
    struct Expression *Right;
};
```

В грамматике описана последовательность операторов. Используем для реализации последовательности односвязный список. В структуре списка (нетерминальный символ `stmt_list`) хранятся указатели на первый и последний элементы списка (последний значительно упрощает добавление новых элементов при синтаксическом разборе):

```
struct StatementList {
    struct Statement *First;
    struct Statement *Last;
};
```

В структуре для оператора находится его тип, указатель на выражение (используется для хранения условного выражения в операторе `if`, печатаемого или просто вычисляемого выражения), указатель на список операторов (используется в условном операторе для описания тела условия), а также указатель на следующий элемент последовательности операторов (0 для последнего элемента):

```
struct Statement {
    enum StmtType Type;
    struct Expression *Expr;
    struct StatementList *Block;
    struct Statement *Next;
};
```

7.2 Описания в файле Bison

Теперь мы можем заполнить секцию описаний в файле для Bison. Обратите внимание, что вам следует подключить заголовочный файл с

описаниями структур до описаний, в секции пролога. Там же, в прологе, объявляется глобальная переменная для хранения корня дерева — в данном случае ее типом будет список операторов.

Прежде всего задаем объединение для семантического значения, объединяющее все используемые типы данных, как для терминальных, так и для нетерминальных символов грамматики:

```
%union {
    int Int;
    char *Id;
    struct Expression *Expr;
    struct StatementList *SL;
    struct Statement *Stmt;
};
```

Теперь можно описать нетерминальные символы грамматики с указанием типа их семантических значений:

```
%type <Expr> expr
%type <SL> program stmt_list
%type <Stmt> stmt print_stmt if_stmt
```

При описании нетерминальных символов выделяются два вида описаний:

- символы с семантическими значениями:

```
%token <Int> INT
%token <Id> ID
```

- символы операций, описываемые в порядке возрастания приоритета с указанием ассоциативности:

```
%right '='
%left '<' '>' EQ NE
%left '+' '-'
%left '*' '/'
%left UMINUS
%nonassoc ')'
```

Обратите внимание, что приоритет описывается только для закрывающей скобки, поскольку приоритет правила определяется по последнему входящему в него терминальному символу.

7.3 Действия по правилам грамматики

После описания структур и символов грамматики можно приступать к описанию действий, выполняемых при свертке участка анализируемого текста по правилу. В большинстве случаев действием будет являться создание структуры и заполнение ее полей. Из этого правила существует несколько исключений:

1) действие для стартового символа грамматики сохраняет указатель на вершину дерева в глобальной переменной;

2) при описании последовательности с пустотой, а также круглых скобок в выражениях используются промежуточные нетерминальные символы (или правила), не приводящие к возникновению новых узлов дерева — в этих случаях достаточно просто присвоить семантическое значение единственного нетерминального символа в левой части правила результату;

3) при описании действий для последовательности, описанной леворекурсивным правилом, необходимо помнить о действиях LR-парсера при разборе леворекурсивной последовательности. Нетерминальный символ последовательности обычно содержит два правила: одно включает только элемент последовательности (срабатывает однократно в начале работы последовательности — подходящий момент для инициализации списка), другое - собственно рекурсивное правило (срабатывает для всех членов последовательности кроме первого — действие будет заключаться в добавлении элемента к списку).

При описании действий категорически не рекомендуется загромождать описание грамматики сложными действиями. Если требуется более двух операторов (либо длинные вызовы), то действия по правилу следует вынести в специальную функцию (описываемую ниже, в третьей секции файла), а в секции правил оставить лишь вызовы функций. Функции должны получать необходимые ей семантические значения символов левой части правила как параметры и возвращать семантическое значение результата.

Для рассматриваемого нам примера понадобятся функции: создания выражения из константы, идентификатора и операции; создания операторов (операторы печати и вычисления выражения могут использовать одну функцию), а также создания списка операторов и добавления элемента в список.

```
struct Expression * create_const_expr(int token)
{
    struct Expression *Result =
        (struct Expression *) malloc(
            sizeof(struct Expression));

  Result->Type = Int;
  Result->Num=token;
  return Result;
}
struct Expression * create_id_expr(char * token)
{
    struct Expression *Result =
```

```
        struct Expression *)malloc(
            sizeof(struct Expression));

    Result->Type = Id;
    Result->Id=token;
    return Result;
}

struct Expression * create_op_expr
    (enum ExprType Type,
    struct Expression *lhs,
    struct Expression *rhs)
{
     struct Expression *Result =
      (struct Expression *) malloc(
        sizeof(struct Expression));
   Result->Type = Type;
   Result->Left=lhs;
   Result->Right=rhs;
   return Result;
}
struct Statement * create_simple_stmt
    (enum StmtType type,
     struct Expression *expr)
{
     struct Statement *Result =
         (struct Statement *) malloc(
             sizeof(struct Statement));
    Result->Type=type;
    Result->Expr=expr;
    return Result;
}
struct Statement * create_if_stmt
    (struct Expression *expr,
     struct StatementList *block)
{
    struct Statement *Result =
         (struct Statement *) malloc(
             sizeof(struct Statement));
    Result->Type=If;
    Result->Expr=expr;
    Result->Block=block;
    Result->Next=NULL
    return Result;
```

```
}
struct StatementList * create_stmt_list
   (struct Statement *s1)
{
    struct StatementList *Result =
    (struct StatementList *) malloc(
       sizeof(struct StatementList));

  Result->First=s1;
  Result->Last=s1;
  return Result;
}
struct StatementList * add_to_stmt_list
   (struct StatementList *s1,
   struct StatementList *s2)
{
   s1->Last->Next=s2;
   s1->Last=s2;
   return s1;
}
```

После добавления действий с вызовами функций грамматика примет следующий вид:

```
program : stmt_list {root=$1;$$=$1;}
;

stmt_list : stmt {$$= create_stmt_list($1);}
| stmt_list stmt {$$=add_to_stmt_list ($1, $2);}
;

stmt : expr ENDL
{$$=create_simple_stmt(Expr,$1);}
| if_stmt {$$=$1;}
| print_stmt {$$=$1;}
;

expr : INT {$$=create_const_expr($1);}
| ID {$$=create_id_expr($1);}
| expr '+' expr {$$=create_op_expr(Plus,$1,$3);}
| expr '-' expr
{$$=create_op_expr(Minus,$1,$3);}
| expr '*' expr {$$=create_op_expr(Mul,$1,$3);}
| expr '/' expr {$$=create_op_expr(Div,$1,$3);}
```

```
      | expr '=' expr {$$=create_op_expr(As,$1,$3);}
      | expr '<' expr {$$=create_op_expr(LT,$1,$3);}
      | expr '>' expr {$$=create_op_expr(GT,$1,$3);}
      | expr EQ expr {$$=create_op_expr(EQ,$1,$3);}
      | expr NE expr {$$=create_op_expr(NE,$1,$3);}
      | '(' expr ')' {$$=$2;}
      | '-' expr %prec UMINUS {$$= create_op_expr
(UMinus,$2,0);}
      ;

      print_stmt : PRINT expr ENDL
{$$=create_simple_stmt( Print,$2);}
      ;

      if_stmt : IF expr ENDL stmt_list ENDIF ENDL
{$$=create_if_stmt($2,$4);}
      ;
```

8 Связь с лексическим анализатором

Синтаксический анализатор получает исходные данные из лексического. Эти данные включают:

- код лексемы (терминального символа);
- семантическое значение лексемы (терминального символа);
- местоположение (необязательно).

Код лексемы передается в виде значения, возвращаемого из функции `yylex`. Функция эта будет автоматически вызываться из `yyparse` каждый раз, когда синтаксическому анализатору понадобится следующий символ. Терминальные символы, обозначаемые одним символом, обычно возвращаются в виде кода этого символа, т.е. обычной символьной константой (если код велик, то для безопасности его стоит привести к типу `unsigned char`). Остальные терминальные символы обозначаются макросами, соответствующими их именам в синтаксическом анализаторе.

Для получения значений этих макросов следует запустить bison с ключом `-d`. В результате будет создан заголовочный файл aaaa_tab.h, где aaa обозначает имя исходного bison-файла. Этот файл будет содержать все необходимые лексическому анализатору объявления. Его следует подключить к исходному файлу для flex в секции объявлений.

Таким образом, если символ не требует семантического значения, то код для flex необходимо модифицировать следующим образом:

```
FOR {return FOR;}
= {return '=';}
```

Для возврата семантических значений используется глобальная переменная yylval. Эта переменная хранит семантическое значение возвращаемого символа. Она должна быть заполнена до возврата кода лексемы. Если используется директива %union, то yylval будет объединением — необходимо заполнить поле, которое будет объявлено семантическим значением данного терминала в bison. Обратите внимание, что при возвращении текста нельзя возвращать прямо указатель yytext, т.к. его содержимое будет испорчено при дальнейшей работе анализатора. Следует сделать копию возвращаемого текста. Например, если поля объединения для строк и целых чисел называются str и ival:

```
[A-Za-z_][0-9A-Za-z_]* {
   yylval.str=(char *)malloc(strlen(yytext)+1);
   strcpy(yylval.str, yytext);
   return ID;
}
[1-9][0-9]* {
   yylval.ival=atoi(yytext);
   return INT;
}
```

Для возвращения местоположений используется переменная yylloc. Работа с местоположениями не является обязательной, однако их использование значительно облегчает отладку синтаксического анализатора: в случае, если возникнет ошибка распознавания исходного файла (т.е. он не будет соответствовать фактически заданной грамматике), вы увидите сообщение об ошибке с указанием места ее возникновения, а не просто факта ее наличия. Это значительно упрощает поиск ошибки.

Таким образом, для связи между лексическим и синтаксическим анализатором необходим заголовочный файл (генерируется bison), глобальные переменные yylval и yylloc (при работе с местоположениями), а также глобальная функция yylex. Два файла (сгенерированные bison и flex) ни в коем случае не следует напрямую объединять с помощью команды #include (общим является только заголовочный файл). Вместо этого следует добавить к проекту оба файла и объявить функции yyparse и yylex внешними (ключевое слово extern перед заголовком функции) в секциях объявлений обоих файлов (переменные yylval и yylloc будут объявлены таковыми автоматически, последняя при включенной опции использовании местоположений в bison), например:

```
extern int yyparse(void);
```

В функции main, оставшейся с предыдущей лабораторной работы, необходимо заменить вызов yylex на вызов yyparse, после чего вызвать свою процедуру вывода на экран (в файл) полученного синтаксического дерева.

9 Конфликты и ошибки в грамматике

Если грамматика не является LR(1)-грамматикой, то при обработке файла bison выдаст сообщения о конфликтах: сдвиг-свертка (shift/reduce) и свертка-свертка (reduce/reduce). Наиболее опасными из них являются конфликты свертка-свертка, их необходимо полностью устранить из грамматики.

Для упрощения отладки грамматики можно использовать следующие методы:

- использовать анализатор с поддержкой местоположений — в этом случае вы сможете узнать, какое конкретно место входного файла вызвало конфликт;
- использовать директиву %error-verbose во входном файле, чтобы получить более подробное сообщение об ошибке (работает только в последних версиях bison);
- использовать ключ --verbose при запуске bison для получения файла, описывающего результирующий автомат.

При использовании ключа --verbose bison генерирует дополнительный файл с расширением output, содержащий описание созданного автомата и конфликтов в нем. Файл состоит из нескольких секций:

- перечень конфликтов, разрешенных с помощью сведений о приоритете и ассоциативности операций — эти конфликты не требуют вашего внимания;
- перечень состояний с неразрешенными конфликтами — используйте его для обнаружения состояний с конфликтами типа свертка-свертка, которые вам требуется разрешить;
- перечень неиспользуемых символов и правил — поможет вам определить ошибки и конфликты, оставляющие какие-либо правила (символы) вашей грамматики не действующими;
- перечень правил грамматики и используемых в них символах;
- перечень состояний конечного автомата и правил действий в них — основной материал для исследования работы вашего анализатора.

Анализируя явно определенные bison конфликты свертка-свертка вы обычно узнаете состояние, в котором произошел конфликт, из второй секции и анализируете описание этого состояния (последняя секция) для того, чтобы найти причину конфликта. Примеры описания состояния:

```
state 0

    $accept  ->  . exp $    (rule 0)
    NUM          shift, and go to state 1
    exp          go to state 2
```

В это состояние анализатор попадает в начале анализа (`$accept`); точка в правиле указывает положение анализатора в процессе анализа текста: слева ничего, справа ожидается то, что в конечном итоге свернется в `expr`. Если в этом состоянии следующим терминалом является `NUM`, то необходимо сдвинуть его на стек и перейти в состояние 1. Если в этом состоянии была произведена на стеке свертка нетерминала `expr`, то перейти в состояние 2.

```
state 1

    exp  ->  NUM .    (rule 5)
    $default    reduce using rule 5 (exp)
```

Первая строка показывает, что мы пытаемся построить `expr` из терминала `NUM`, используя правило 5 (секция правил `exp -> NUM`). В данном случае возможно только одно действие — свертка по этому правилу в нетерминал `expr`.

```
state 2

    $accept  ->  exp . $   (rule 0)
    exp  ->  exp . '+' exp   (rule 1)
    exp  ->  exp . '-' exp   (rule 2)
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)

    $           shift, and go to state 3
    '+'         shift, and go to state 4
    '-'         shift, and go to state 5
    '*'         shift, and go to state 6
    '/'         shift, and go to state 7
```

Более сложное состояние. Точка в правилах показывает, что в данном случае анализатор находится после выражения (т.е. какое-либо выражение в исходной строке было разобрано, или, в терминах анализатора, свернуто на стеке). В данном случае анализатор может ожидать конец текста (`$`), либо знак арифметической операции — любой другой вариант является синтаксической ошибкой. При обнаружении во входном потоке соответствующего терминального символа, он сдвигается на стек и происходит переход в нужное состояние.

Этой информации обычно достаточно, чтобы разобраться в причинах конфликта и устранить его[1]. Однако устранения конфликтов типа свертка-свертка не всегда достаточно для отладки грамматики. Некоторые конфликты типа сдвиг-свертка также могут служить причиной неправильной работы грамматики.

Например, в языке C++ функция может иметь тип `void`, а переменная — нет. Рассмотрим следующую попытку описать возможность задать как глобальную переменную, так и функцию:

```
def :type ID ';'
    |  f_type ID '(' formal_param_list ')' ';'
;
f_type:type
    |  VOID
;
type :ID
    | INT
;
```

Если не учитывать особенности работы LR(1)-анализатора, то такая грамматика может показаться вполне корректной. Тем не менее она не сможет распознавать прототипы функций с типом, отличным от `void`. Причина этого кроется в том, что сдвинув терминальный символ типа (`ID` или `INT`), и свернув его в `type`, анализатор должен решить, сворачивать ли его в `f_type` или сдвигать следующий за ним `ID`. Информации для этого недостаточно, т.к. LR(1)-анализатор может использовать для этого решения только один последующий символ (т.е. `ID`), который одинаков в обеих случаях (необходимая для разрешения конфликта скобка идет далее, в действительности это LR(2)-грамматика). Поскольку поведением по умолчанию в конфликте сдвиг-свертка является сдвиг, то `ID` будет сдвинут на стек, предотвращая возможность свертки `type` в `f_type` (напоминаю, что свертка может происходить только на вершине стека), при обнаружении далее скобки будет выдано сообщение о синтаксической ошибке, которое может удивить неосторожного автора грамматики. Для разрешения конфликта необходимо устранить нетерминал `f_type`, введя вместо этого дополнительное правило для нетерминала `def`:

```
def : type ID ';'
    | type ID '(' formal_param_list ')' ';'
    | VOID ID '(' formal_param_list ')' ';'
;
```

В этом случае анализатор не должен совершать выбора между переменной и функцией заранее, до достижения скобки (или точки с запятой после идентификатора), поэтому все случаи будут успешно разобраны.

10 Контрольные вопросы

1. Объявление терминальных и нетерминальных символов в генераторе парсеров bison
2. Задание приоритета и ассоциативности операций в генераторе парсеров bison
3. Описание правил в генераторе парсеров bison. Работа с семантическими значениями правил

4. Использование объединения в качестве семантического значения в генераторе парсеров bison

5. Работа с положениями в тексте в генераторе парсеров bison.

6. Устройство C-парсера. Связь между C-парсером и сканером.

7. Устройство C++-парсера. Связь между C++ парсером и сканером.

8. GLR-парсеры. Отличие GLR разбора от LR-разбора.

9. Составьте структуры данных и правила грамматики для фрагмента языка, где индексы многомерных массивов записываются через запятую в одних квадратных скобках.

10. Составьте структуры данных и правила грамматики для фрагмента языка, поддерживающего операции =, +,-,*,/, целые переменные (без объявления) и константы с префиксной нотацией выражений.

11. Составьте структуры данных и правила грамматики для фрагмента языка, поддерживающего операции =, +,-,*,/, целые переменные (без объявления) и константы с постфиксной нотацией выражений.

Учебное издание

Олег Александрович **Сычев**

СИНТАКСИЧЕСКИЙ АНАЛИЗ ТРАНСЛИРУЕМОЙ ПРОГРАММЫ

Методические указания

Темплан выпуска электронных изданий 2009 г., поз. № 30.

На магнитоносителе. Уч.-изд. л. 1,52.
Подписано на «Выпуск в свет» 15.06.2009. Заказ № 11.

Волгоградский государственный технический университет.
400131, г. Волгоград, пр. им. В. И. Ленина, 28, корп. 1.