Лабораторная работа №2. Разработка блока синтаксического анализа транслятора простого языка программирования

Цель: изучение методов синтаксического анализа с их программной реализацией.

Задачи:

1. Изучение теоретического материала по организации синтаксического анализа языков программирования.
2. Составление формального описания синтаксического анализатора.
3. Программная реализация по формальному описанию.

Ход работы:

1. Получить у преподавателя собственный вариант задания, предусматривающего разработку синтаксического анализатора простого языка программирования с возможностью вывода дерева разбора анализируемой программы и/или таблицы лексем на экран, а также обязательной диагностикой синтаксических и лексических ошибок.
2. Составить формальное описание программы синтаксического анализа с использованием математического аппарата формальных контекстно-свободных грамматик. Полученная LALR(1)-грамматика не должна приводить к конфликтам «перенос-свертка» и «свертка-свертка».
3. Произвести автоматическую генерацию исходного текста программы синтаксического анализа с использованием системы bison. Отладить полученную программу.

При разработке программы рекомендуется отделить синтаксический анализатор от вспомогательных модулей, в том числе от лексического анализатора.

4. Написать отчет.

Внимание: в приложениях приведены исходные коды для вывода на экран деревьев разбора строк на простом языке программирования. В рабочий проект достаточно включить *print-tree.c* и файлы, сгенерированные по спецификациям *lexer.ll* и *language.yy*. Используемый компилятор должен соответствовать стандарту C99 языка программирования C.

Ход защиты:

1. Продемонстрировать преподавателю корректную работу программы синтаксического анализа.
2. Пояснить работу изученных механизмов по полученному формальному описанию программы синтаксического анализа.

Во время защиты лабораторной работы необходимо иметь при себе исполняемый модуль программы, исходные тексты, файлы с тестовыми примерами, а также твердую копию отчета.

Содержание отчета:

1. Цель работы с постановкой задачи.
2. Исходный текст распознавателя в соответствии с полученным заданием.
3. Тестовые примеры работы программы[1].

Краткие теоретические сведения

Формальная грамматика – это система, позволяющая определить языки посредством порождающих правил. Например,

S : 1S2 ;

S : ;

Каждое правило, которое называется продукцией, имеет левую и правую части, разделенные двоеточием, обозначающем «есть» и заканчивающееся точкой с запятой. Символы 1 и 2 принадлежат алфавиту языка, а S – вспомогательный символ для обозначения комбинаций символов языка в строках. Вспомогательных символов может быть любое количество. Применение правил в последовательности 1,1,1,2 породит строку 111222. Теперь можно отобразить этот процесс с указанием номеров продукций и получаемыми промежуточными строками $S \Rightarrow_1 1S2 \Rightarrow_1 11S22 \Rightarrow_1 111S222 \Rightarrow_2 111222$. Полученную последовательность строк называют выводом строки (или порождением строки). Символ $\Rightarrow$ обозначает один шаг вывода или порождения, его можно трактовать как «непосредственно выводится из». Каждая из строк, участвующих в порождении, называется сентенциальной формой, а последняя, которая состоит только из терминальных символов, называется предложением языка.

Формальная грамматика определяется как четверка (V_T, V_N, P, S), где приняты следующие обозначения:

– V_T – это алфавит языка, элементы этого множества называются терминальными символами или терминалами.

– V_N – это вспомогательный алфавит, элементы этого множества называются нетерминалами. Множества терминалов и нетерминалов не пересекаются, а их объединение иногда называют словарем грамматики.

– P – это множество продукций грамматики в виде пар $\alpha : \beta;$, где $\alpha \in (V_N \cup V_T)^+$, $\beta \in (V_N \cup V_T)^*$. Иными словами, в левой части правила должны содержать хотя бы один нетерминальный символ.

– S – начальный символ или аксиома грамматики. С аксиомы начинается порождение любой строки языка.

Во множестве продукций грамматики могут встречаться несколько правил, имеющих одинаковые левые части вида $\alpha : \beta_1;, \alpha : \beta_2;, ..., \alpha : \beta_n;$. В этом случае правила объединяют и записывают в виде $\alpha : \beta_1 | \beta_2 | ... | \beta_n;$, где n – это количество продукций. Допускаются некоторые другие усовершенствования:

[1] Не менее трех тестовых примеров.

– круглые скобки означают, что внутри них в данном случае может быть только одна из перечисленных цепочек;

– квадратные скобки означают нуль либо одно вхождение цепочки, указанной внутри скобок;

– фигурные скобки означают нуль, одно и более вхождений цепочки, указанной внутри скобок;

– запятая предназначена для разделения цепочек символов внутри скобок;

– если требуется включить в строку один из перечисленных выше символов, можно использовать кавычки.

Определим продукции грамматики для целых чисел:

$$N : [(+,-)]D\{D\};$$

$$D : 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9;$$

Одно и то же предложение языка можно по-разному вывести (породить) при помощи одних и тех же продукций. Если на каждом шаге порождения заменяется крайний левый нетерминал сентенциальной формы, такое порождения называется левым. Аналогично определяется правое порождение, если на каждом шаге заменяется крайний правый нетерминал.

Одну и ту же строку языка можно генерировать, используя разные грамматики. Грамматики, генерирующие один и тот же язык, называются эквивалентными.

Подавляющее большинство грамматик являются рекурсивными. Рекурсия выражается в том, что некоторые нетерминальные символы в правилах грамматики выражаются через себя. Рекурсия может быть прямой, когда символ определяется через себя в одном правиле, а непрямой – когда то же самое проходит через цепочку правил. В грамматике с продукциями

S : 1 S 2;

S : ;

в продукции 1 имеет место прямая рекурсия, а в грамматике с продукциями

S : 1 D 2;

D : 3 S;

D : 4;

в продукциях 1 и 2 наблюдается непрямая рекурсия. Чтобы рекурсия не была бесконечной, для соответствующего нетерминального символа должны быть предусмотрены правила, помогающие избавиться от бесконечного рекурсивного определения (в приведенной грамматике – это продукция 3). Рекурсия может быть правой, левой и средней, это зависит от того, в каком месте продукции находится нетерминальный символ. Например, в грамматике

S : 1 S;

S : S 1;

S : 1 S 2;

В продукциях имеют место соответственно правая, левая и средняя рекурсии. Наиболее часто встречается средняя рекурсия. В любом случае, ничего страшного в этом интересном явлении нет. Более того, только рекурсия

позволяет с помощью конечного набора продукций выразить бесконечное множество строк языка.

Синтаксический анализ (СА) выполняется для того, чтобы проверить правильность конструкций программы, которые образованы из лексем. Символами входной строки для СА являются лексемы. Иными словами, множество терминалов грамматики, описывающей синтаксические свойства языка, содержат только лексемы.

В процессе компиляции задача СА заключается в нахождении порождения конкретного предложения языка, используя конкретную грамматику. Более точно, требуется не столько найти предложение, сколько найти его порождение или доказать, что его не существует. Как правило, искомыми являются левое (на каждом шаге порождения заменяется крайний левый нетерминал сентенциальной формы) или правое (на каждом шаге порождения заменяется крайний правый нетерминал сентенциальной формы) порождения, являющиеся единственными. Если же порождение не является единственным, то должны существовать правила, указывающие в каком случае какое порождение использовать. Таким образом, результат СА всегда определен.

Синтаксический анализатор – это основная часть компилятора, которая отвечает за СА. Без СА работа компилятора становится бессмысленной. Как правило, синтаксические конструкции языков программирования описываются с помощью КС-грамматик, но иногда они могут описываться с помощью регулярных грамматик.

Существует несколько способов поиска порождения предложения языка, однако наиболее популярны два: нисходящий СА и восходящий СА. В первом случае начинают с аксиомы грамматики и двигаются к предложению (генерация предложения), используя продукции грамматики. Во втором случае начинают с предложения и двигаются к аксиоме грамматике (сворачивание предложения). И в том, и в другом случае обычно предложения, которые предполагается сгенерировать или свернуть, считывают слева направо.

Нисходящий синтаксический анализ

При нисходящем СА требуется найти левое порождение строки языка. Для примера возьмем следующую грамматику.

Рассмотрим язык, который генерируется грамматикой со следующими продукциями.

$S : XY;$

$X : xX|x;$

$Y : yY|y;$

Предложение *xxxyyyy* можно сгенерировать с помощью левого порождения:

$$S \Rightarrow XY \Rightarrow xXY \Rightarrow xxXY \Rightarrow xxxY \Rightarrow xxxyY \Rightarrow xxxyyY \Rightarrow xxxyyyY \Rightarrow xxxyyyy$$

Первый шаг порождения очевиден, так как символ аксиомы находится в левой часть только одной продукции. Крайний левый нетерминал входит в

левую часть двух правил. Значит, на втором шаге порождения требуется выбрать одну из них, но поскольку в генерируемом предложении более одного *x*, то используется продукция $X \to xX$. Аналогичные выводы можно сделать на третьем шаге. На четвертом шаге требуется сгенерировать последний x, значит, нужно использовать продукцию $X \to x$. При выполнении шагов (5-8) для порождения символов *y* используются две последние продукции.

Таким образом, помимо сгенерированных символов требуется знать еще два символа предпросмотра. По определению, это текущий символ входной строки или специальный символ конца строки, который обычно обозначается как $\perp$. Принятие решений при нисходящем СА основывается на символе или группе символов предпросмотра.

Далее будет рассмотрен один большой подкласс КС-грамматик, который поддерживают методы нисходящего СА с одним символом предпросмотра. Эти грамматики считаются однозначными, то есть каждому предложения языка соответствует единственное левое порождение. Для каждого нетерминала из левой части нескольких продукций необходимо найти непересекающиеся множества символов предпросмотра, чтобы каждое из них содержало символы, соответствующие только одной возможной правой части продукции. Выбор продукции зависит от символа предпросмотра и множества, к которому он принадлежит. Если символ предпросмотра не принадлежит ни одному из непересекающихся множеств, делается вывод о синтаксической ошибке.

Синтаксический анализ методом рекурсивного спуска работает нисходящим образом и включает использование рекурсивных процедур, но надо помнить, что все грамматики, допускающие СА методом рекурсивного спуска, являются подклассом LL(1), и существует ряд LL(1)-языков, чьи LL(1)-грамматики не допускают рекурсивный спуск. Тем не менее, данный метод позволяет достаточно просто разрабатывать программы СА и заслуженно стал наиболее популярным способом нисходящего СА.

Для примера возьмем грамматику простейшего языка программирования со следующими продукциями:

$PROG : start\ DECLARATIONS\ comma\ OPERATORS\ finish;$

$DECLARATIONS : decl\ Z;$

$Z : semicolon\ DECLARATIONS \mid ;$

$OPERATORS : oper\ X$

$X : semicolon\ OPERATORS \mid ;$

Данная грамматика является LL(1), и, как известно из теории, для этого надо рассмотреть множества первых порождаемых символов для двух продукций *X* и двух продукций *Z* и проверить их на непересечение.

Использование СА методом рекурсивного спуска (далее РС) состоит из последовательного определения всех символов правых частей продукций. Терминалы, которые обозначены строчными буквами, определяются непосредственно вызовом лексического анализатора. Нетерминалы определяются вызовом соответствующих функций, которые рекомендуется называть также, как и сами нетерминалы. Далее приведен фрагмент далекой от

совершенства программы РС для языка, генерируемого приведенной грамматикой.

```
#include <stdio.h>
void main(), PROG(), DECLARATIONS(), OPERATORS(), X(), Z();

void main()
{
    lexeme = get_token();
    PROG();
}

void PROG()
{
    if (lexeme != start) error();

    lexeme = get_token();
    DECLARATIONS();
    if (lexeme != comma) error();

    lexeme = get_token();
    OPERATORS();
    if (lexeme != finish) error();
}

void DECLARATIONS()
{
    if (lexeme != decl) error();
    lexeme = get_token();
    Z();
}

void Z()
{
    if (lexeme == semicolon)
    {
        lexeme = get_token();
        DECLARATIONS();
    }
    else if (lexeme != comma) error();
}

void OPERATORS()
{
    if (lexeme != oper) error();
    lexeme = get_token();
     X();
}

void X()
{
    if (lexeme == semicolon)
    {
        lexeme = get_token();
        OPERATORS();
    }
    else if (lexeme != finish) error();
}
```

Вызов функции *get_token*() позволяет синтаксическому анализатору получить от лексического анализатора очередной символ. Функция *error*() вызывается после появления ошибки. Терминалы *start*, *finish*, *comma*, *semicolon*,

decl, *oper* являются предварительно объявленными константами, значениями которых являются представления символов СА.

Можно изменить и расширить полученный язык арифметическими выражениями, элементами которых являются целые числа, имена переменных из одного символа, а также знаки операций. Кроме того, можно добавить оператор присваивания и вывода значений переменных по именам. Последовательность операторов разделяется точкой с запятой. Для изменения приоритетов операций можно использовать круглые скобки.

Грамматика для полученного языка может иметь следующие продукции.

```
Program    : Statement
           | Statement ';' Program
           ;
Statement  : Assignment
           | Expression
           | Variable
           ;
Assignment : Variable '=' Expression ;
Expression : Term
           | Term '+' Expression
           | Term '-' Expression
           ;
Term       : Factor
           | Term '*' Factor
           | Term '/' Factor
           ;
Factor     : Variable
           | Number
           | '-' Factor
           | '+' Factor
           | '(' Expression ')'
           ;
```

Лексический анализатор для такого языка также сравнительно прост. Лексемы *Variable* и *Number* можно описать следующими регулярными выражениями.

```
Variable   [a-zA-Z]
Number     [0-9]+
```

Построение СА, использующих LL(k)-грамматики и/или работающих методом рекурсивного спуска, может быть автоматизировано. Примерами таких программных средств являются Coco/R и ANTLR.

Восходящий синтаксический анализ

При восходящем СА требуется найти правое порождение строки языка. Рассмотрим один из таких языков.

$S : XY;\ X : xX|x;\ Y : yY|y;$

Цепочку *xxxyyyy* можно сгенерировать с помощью правого порождения

$S \Rightarrow XY \Rightarrow XyY \Rightarrow XyyY \Rightarrow XyyyY \Rightarrow Xyyyy \Rightarrow xXyyyy \Rightarrow xxXyyyy \Rightarrow xxxyyyy$

Однако при восходящем СА удобнее представить этапы порождения не в указанном, а в обратном порядке.

$xxxyyyy \Rightarrow xxXyyyy \Rightarrow xXyyyy \Rightarrow Xyyyy \Rightarrow XyyyY \Rightarrow XyyY \Rightarrow XyY \Rightarrow XY \Rightarrow S$

На каждом этапе применятся правило грамматики, его правая часть заменяется левой, которая состоит из одного символа. Правые части продукций не распознаются, пока не будут полностью считаны, а значит, частично распознанные части требуется где-то хранить до замены их левыми частями. Для хранения используется наиболее подходящая на этот случай структура данных – стек. Этапы процесса восходящего СА состоят из двух типов действий:

1. Перемещение последнего считанного символа в стек – действие переноса (сдвига, *shift*), то есть символ удаляется их входной строки и размещается в стеке.

2. Замена строки, находящейся на вершине стека, применением правила грамматики – действие свертки (*reduce*), что, естественно, вызывает изменение вершины стека.

3. Прием (*accept*) – окончание ВСА с принятие анализируемой цепочки.

4. Ошибка (*error*) – возникновение синтаксической ошибки

Покажем различные этапы СА.

Строка	Стек	Правило	Сентенциальная форма	Действие
xxxyyyy			xxxyyyy	
~~x~~xxyyyy	x		xxxyyyy	Перенос
~~xx~~xyyyy	xx		xxxyyyy	Перенос
~~xxx~~yyyy	xxx		xxxyyyy	Перенос
~~xxx~~yyyy	xxX	X : x;	xxXyyyy	Свертка
~~xxx~~yyyy	xX	X : xX;	xXyyyy	Свертка
~~xxx~~yyyy	X	X : xX;	Xyyyy	Свертка
~~xxxy~~yyy	Xy		Xyyyy	Перенос
~~xxxyy~~yy	Xyy		Xyyyy	Перенос
~~xxxyyy~~y	Xyyy		Xyyyy	Перенос
~~xxxyyyy~~	Xyyyy		Xyyyy	Перенос
~~xxxyyyy~~	XyyyY	Y : y;	XyyyY	Свертка
~~xxxyyyy~~	XyyY	Y : yY;	XyyY	Свертка
~~xxxyyyy~~	XyY	Y : yY;	XyY	Свертка
~~xxxyyyy~~	XY	Y : yY;	XY	Свертка
~~xxxyyyy~~	S	S : XY;	S	Свертка
~~xxxyyyy~~				Прием

В начале СА сентенциальная форма в точности повторяет сворачиваемое предложение, при этом стек пуст. При выполнении действия переноса символ во входной строке зачеркивается. По окончании СА сентенциальная форма – это символ аксиомы, только он находится в стеке, а вся строка прочитана (все символы зачеркнуты).

В таблице показано каждое конкретное действие СА, но не показывается, когда именно должны производиться свертка или перенос и как осуществлять выбор, если возможны несколько действий свертки. Свертка производится при условии наличия правой части некоторого правила на вершине стека (однако

этого недостаточно), иначе необходимо выполнять перенос. Также на вершине стека могут определяться правые части более одной продукции, то есть на некотором этапе СА могут существовать две и более свертки. В этом случае говорят, что имеет место конфликт свертка-свертка. Если же кажутся возможными действия переноса и свертки, то имеет место конфликт перенос-свертка. Для разрешения названных конфликтов на практике применяется информация, полученная путем предпросмотра, и/или предыстория СА.

В нашем примере для разрешения конфликтов может использоваться один символ предпросмотра. Для применения правила $X : x;$ символом предпросмотра являлся y. Символ конца строки ($\perp$) определяет применение правила $Y : y;$.

Грамматика, все конфликты которой, возникающие при восходящем СА, разрешаются использованием информации, касающейся уже проведенного анализа, и конечного числа символов предпросмотра, называется LR(k)-грамматикой. Символ L означает чтение строки слева направо, R – использование правых порождений, k – количество символов предпросмотра. Языком LR(k) называется язык, который может быть сгенерирован LR(k)-грамматикой. Если требуется один символ предпросмотра, то грамматика и язык относятся к классу LR(1).

Критерий принятия решений относительно предпринимаемого действия может содержаться в таблице, которая называется таблицей синтаксического анализа. В ней каждому состоянию синтаксического анализатора соответствует одна строка, а каждому символу грамматики (нетерминалу и терминалу, а также маркеру конца строки) – один столбец. Например, для грамматики с продукциями (они пронумерованы для иллюстрации):

$1.E \to E + T\ 2.E \to T\ 3.T \to T * F\ 4.T \to F\ 5.F \to (E)\ 6.F \to x$

таблица СА может иметь следующий вид.

	E	T	F	+	*	(	)	x	⊥
1	П2	П5	П8			П9		П12	Прием
2				П3					
3		П4	П8			П9		П12	
4				С1	П6		С1		С1
5				С2	П6		С2		С2
6			П7			П9		П12	
7				С3	С3		С3		С3
8				С4	С4		С4		С4
9	П10	П5	П8			П9		П12	
10				П3			П11		
11				С5	С5		С5		С5
12				С6	С6		С6		С6

В начале процесса СА анализатор находится в состоянии 1, входной символ – это первый введенный символ. Каждый шаг анализа определяется ячейкой таблицы, которая соответствует текущему состоянию, а также входным символом. Используемая для работы анализатора ячейка таблицы может принимать один из двух типов.

1. Перенос вида Пk, анализатор выполняет действие переноса, текущее состояние изменяется на состояние k.

2. Свертка вида Cm, анализатор выполняет действие свертки, при этом используется продукция *m*.

Пустые ячейки соответствуют синтаксическим ошибкам. Следовательно, с каждой из таких ячеек можно связать индивидуальное сообщение об ошибке. Синтаксический анализатор находится в одном из конечного числа состояний, и оно совместно с символом предпросмотра либо только что свернутым нетерминалом определяют элемент таблицы СА. Если он определяет перенос, то выполняются операции:

– символ, соответствующий текущему столбцу, заносится в стек символов;

– состояние запоминается в стеке состояний;

– если входной символ является терминальным, то он принимается, а следующим входным символом становится следующий терминал или символ конца строки.

Если элемент таблицы определяет свертку, то выполняются операции:

– из стеков удаляются P элементов, где P – количество символов в продукции, по которой осуществляется свертка;

– анализатор переходит в состояние на вершине стека состояний;

– входной символ становится символом в левой части продукции, по которой осуществляется свертка.

На ход СА не влияет содержимое стека символов, его можно не принимать во внимание, тогда как стек состояний оказывает влияние на выполнение СА. Кроме того, таблицы СА создаются инструментальными средствами, то пользователям и разработчикам компиляторов нет необходимости понимать принципы ее формирования. Надо отметить, что действия переноса всегда заносятся в таблицу СА до действий свертки. Если в таблице отсутствуют конфликты типа перенос-свертка и свертка-свертка, то грамматика обозначается LR(0), где 0 сигнализирует о том, что символы предпросмотра для разрешения конфликтов не используются.

Однако, при своей простоте LR(0)-грамматики могут генерировать очень небольшое число языков. Гораздо чаще применяются LR(1)-грамматики и их подклассы.

Управляющие таблицы для LR(1)-грамматик заполняются на основании значений не только стека, но и символов предпросмотра для каждого нетерминала. Если таблица содержит конфликты, значит, грамматика не является LR(1).

Таблицы СА для LR(1)-грамматик имеют очень большие размеры и обрабатываются медленно, так как должны учитывать много информации. Для ее сокращения могут применяться SLR(1)-грамматики, но чаще используются LALR(1)-грамматики. Если для внесения в таблицу СА действий свертки учитываются все возможные символы-последователи нетерминалов левой части продукции, то таблица называется SLR(1)-таблицей. Если она не содержит конфликтов, то грамматика называется SLR(1)-грамматикой, то есть простой LR(1)-грамматикой. Все LR(0)-грамматики являются SLR(1)-грамматиками, но не наоборот. Конфликты, возникающие в SLR(1)-таблицах,

могут разрешаться с помощью LALR(1)-таблиц, в которых ограничивается число рассматриваемых при конкретной свертке символов-последователей на основе информации о состояниях. В них определяются только те последователи, которые корректны в данном состоянии. Если LALR(1)-таблица не содержит конфликтов, то грамматика называется LALR(1)-грамматикой, то есть LR(1)-грамматикой с предпросмотром. Класс SLR(1)-грамматик уже класса LALR(1). Оставшиеся после LALR(1) конфликты могут быть разрешены использованием LR(1)-таблиц. Синтаксический анализатор использует таблицы СА одинаково, следовательно, не имеет значения, какой алгоритм применялся для ее создания. В современных версиях bison можно использовать также грамматики GLR(1) и IELR(1), описание которых можно найти в работах, приведенных в библиографическом списке.

Генератор синтаксических анализаторов bison

Автоматизированное построение восходящих синтаксических анализаторов может выполняться при помощи генератора YACC. Существуют свободно-распространяемые варианты YACC, например, BYacc, bison, zubr и другие. Входной файл для этих инструментальных средств имеет следующий вид:

```
определения
%%
правила
%%
функции пользователя
```

Первая и третья секции могут быть пустыми. Если секция функции пользователя опущен, то второй разделитель %% можно не указывать. На выходе bison выдает программу на языке C, но существуют версии и для других языков программирования. Для описания грамматики в bison и других инструментах используется форма, похожая на форму Бэкуса-Наура. Например, язык арифметических выражений будет представлен следующим входом bison.

```
%left '+' '-'
%left '*' '/'
%%
expr    :  expr '+' expr
        |  expr '-' expr
        |  expr '*' expr
        |  expr '/' expr
        |  '(' expr ')'
        |  NUMBER
        ;
```

Терминальные символы заключены в апострофы. Как видно, эта грамматика содержит левую и правую рекурсии. Более того, она является неоднозначной, хотя LR(1)-грамматики не могут быть таковыми. Все дело в том, что здесь применяются другие правила разрешения неоднозначности. во-первых, %left означает, что в частности операции сложения и вычитания выполняются слева направо, а это как раз соответствует порядку выполнения

математических операций. Во-вторых, символы * и / находятся в строке, расположенной под строкой с символами «+» и «-«, а это значит, что умножение и деление имеют приоритет, более высокий, чем сложение и вычитание.

Может потребоваться ввести унарные операции. Например, унарный минус включается в данную грамматику следующим образом.

```
%token NUMBER
%left '+' '-'
%left '*' '/'
%left MINUS
%%
expr    :  expr '+' expr
        |  expr '-' expr
        |  expr '*' expr
        |  expr '/' expr
        |  '-' expr          %prec MINUS
        |  '(' expr ')'
        |  NUMBER
        ;
```

В конец синтаксических правил можно вводить действия, а на практике без этого трудно обойтись. Если для умножения и унарного минуса требуется выполнять действия, то их нужно заключить в фигурные скобки, как в следующем фрагменте кода.

```
expr    :  expr '*' expr                { $$ = $1 * $3;}
        |  '-' expr    %prec MINUS      { $$ = -$2;}
        ;
```

Здесь использована еще одна интересная особенность bison: переменные со знаком доллара. $n – это значение (или атрибут) *n*-го символа правой части продукции, $$ – это значение символа в левой части (его можно рассматривать как синтезируемый атрибут). Терминалы, не соответствующие ровно одному знаку, объявляются с ключевым словом %token, которое обеспечивает взаимосвязь с лексическим анализатором. При выполнении данной работы также необходимо реализовать блок лексического анализа для распознавания чисел и других лексем, которые могут встретиться в выражениях. Например, в секции пользовательских функций можно определить следующую функцию *yylex*(), которая отвечает за распознавание чисел с плавающей точкой двойной точности и передачу очередной распознанной лексемы синтаксическому анализатору:

```
#include <ctype.h>
int yylex(void)
{
    int c;

    /* пропуск пробелов */
```

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

    /* обработка чисел */
    if (c == '.' || isdigit(c))
    {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    /* обработка символа конца файла  */
    if (c == EOF)
        return 0;

    /* обработка символов, состоящих из одного знака */
    return c;
}
```

Минимальная по размеру функция, управляющая синтаксическим анализом, может выглядеть так:

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    return yyparse();
}
```

Аналогично, минимально возможная функция для диагностики синтаксических ошибок, может выглядеть следующим образом:

```
void yyerror(const char *s)
{
    fprintf(stderr, "Line %d - \"%s\"\n", yylineno, err);
}
```

В нашем случае для калькулятора нужно также изменить тип данных, хранимых в стеке по умолчанию на числа с плавающей точкой, для чего в секцию объявлений нужно добавить строки:

```
%{
#define YYSTYPE double
%}
```

Если же необходимо добавить возможность использования переменных, то придется несколько видоизменить тип данных для стека в секции объявлений входной спецификации. Например:

```
%union
{
    double val;/*число с плавающей точкой двойной точности*/
    char name; /* имя переменной – единственный символ */
}
```

```
%token <val> NUMBER
%token <name> VAR
%type <val> expr
```

Объединение с двумя полями устанавливает два типа данных в стеке, для лексемы *NUMBER* с помощью ключевого слова %token и для нетерминала *expr* с помощью ключевого слова %type устанавливается тип *double*, а для лексемы *VAR* – единственный символ, который и будет хранить имя переменной. Придется внести изменения и в функцию *yylex*(), где производится обращение и установка значения различных полей внутренней переменной *yylval*.

```
#include <ctype.h>
int yylex (void)
{
    int c;

    /* пропуск пробелов */
    while ((c = getchar()) == ' ' || c == '\t');

    /* обработка чисел */
    if (c == '.' || isdigit(c))
    {
        ungetc(c, stdin);
        scanf("%lf", &yylval.val);
        return NUMBER;
    }
    /* обработка символа с именем переменной */
    if (isalpha(c)) { yylval.name = c; return VAR; }
    /* обработка символа конца файла  */
    if (c == EOF)
        return 0;
    /* обработка символов, состоящих из одного знака */
    return c;
}
```

Для распознавания условного оператора и цикла с предусловием можно использовать строки, подобные тем, что показаны в приведенном ниже фрагменте кода.

```
%union
{
    double val;/*число с плавающей точкой двойной точности*/
    char name; /*имя переменной – единственный символ */
}
...
%token IF THEN ELSE WHILE DO
%type <val> statement expr if_stmt while-stmt
%%
statement : expr
          | if_stmt
          | while_stmt
          ;
```

```
if_stmt   : IF '(' expr ')' THEN '{' statement '}'
            { printf("Распознан if-then\n");
              $$->val = $6->val;
            }
          | IF '(' expr ')' THEN '{' statement '}'
                            ELSE '{' statement '}'
            { printf("Распознан if-then-else\n");
             $$->val = $4->val + $6->val;
            }
          ;
while_stmt : WHILE '(' expr ')' '{' statement '}'
             { printf("Распознан while-stmt\n");
               $$->val = $6->val;
             }
           ;
...
```

Действия в конце продукций могут быть иными, например, для построения дерева синтаксического разбора, абстрактного синтаксического дерева или любого другого представления программы, необходимого для последующих фаз процесса компиляции. Кроме того, также потребуется модифицировать код лексического анализатора, но такие изменения не должны вызвать трудности при реализации.

Использованы следующие источники:

1. Шилдт, Г. Искусство программирования на C++. / Г.Шидлт. СПб.: БХВ-Петербург, 2005. – 496 с.
2. RedDragon Book (1985).
3. Компиляция. 5: нисходящий разбор - http://habrahabr.ru/blogs/programming/99466/
4. Костельцев, А.В. Построение интерпретаторов и компиляторов / А.В. Костельцев. – СПб.: Наука и Техника, 2001. – 224 с.
5. Levine, J. lex & yacc, Second Edition / J. Levine, T. Mason, D. Brown. – Sebastopol, CA: O'Reilly Media, 1992. – 384 p.
6. Levine, J. flex & bison / J. Levine. – Sebastopol, CA: O'Reilly Media, 2009. – 292 p.
7. http://www.linux.org.ru/books/GNU/bison/bison_toc.html
8. http://epaperpress.com/lexandyacc/index.html
9. Компиляция. 2: грамматики - http://habrahabr.ru/blogs/programming/99298/
10. Компиляция. 3: бизон - http://habrahabr.ru/blogs/programming/99366/
11. Компиляция. 4: игрушечный ЯП - http://habrahabr.ru/blogs/programming/99397/
12. http://cas.ee.ic.ac.uk/people/ccb98/teaching/ee2_software_engineering/Lecture10-Parsing2.pdf
13. http://bytes.com/topic/c/answers/788879-parsing-tree-help
14. http://www.cs.man.ac.uk/~pjj/cs211/ho/node8.html
15. http://web.eecs.utk.edu/~bvz/cs461/notes/parse_tree/

Общая постановка задачи и варианты заданий к лабораторной работе 2

Общие требования:

1) Разработать программу, осуществляющую синтаксический, а также лексический анализ простого языка программирования. Наличие в языке конструкций, отсутствующих в задании, является ошибочным.

2) На вход синтаксического анализатора подается внешний файл с текстом программы на языке, описанном в конкретном варианте задания.

3) На выходе синтаксический анализатор должен выдавать дерево разбора программы и/или информацию об ошибках и/или таблицу всех лексем с важнейшими характеристиками: тип/класс лексемы; ее литеральное написание; порядковый номер строки, содержащей лексему.

4) Используется интерфейс командной строки, т.е. анализатор в общем случае запускается так:

```
user$: parser -parameters input.lng
```

здесь *user$* – подсказка командной строки; *parser* – имя исполняемого модуля программы синтаксического анализа; *-parameters* – 0 или более параметров командной строки; *input.lng* – пример имени внешнего файла с программой на анализируемом языке.

5) С помощью параметра командной строки можно явно отключать и включать функцию вывода дерева разбора. По умолчанию эта функция должна быть отключена. Для вывода дерева на экран:

```
user$: parser –tree input.lng
```

Для отключения вывода дерева на экран:

```
user$: parser –notree input.lng
```

6) С помощью параметра командной строки можно явно отключать и включать функцию вывода таблицы лексем. По умолчанию эта функция должна быть отключена. Для вывода таблицы лексем на экран:

```
user$: parser –lexemes input.lng
```

Для отключения вывода дерева на экран:

```
user$: parser –nolexemes input.lng
```

7) При выводе информации об ошибке обязательно указание на номер строки в исходном файле.

8) Диагностируются как синтаксические, так и лексические ошибки.

Вариант 1. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из

идентификаторов, десятичных чисел с плавающей запятой[1] в обычном и нормализованном экспоненциальном форматах, а также целочисленных констант. Кроме того, элементами арифметических выражений являются знаки присваивания ('='), знаки операций ('+', '-', '*', '/', '%') и круглые скобки.

Вариант 2. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'). Выражения состоят из идентификаторов, нечувствительных к регистру констант *true* и *false*, десятичных чисел с плавающей точкой[2] в обычном формате, знаков присваивания ('='), знаков операций *or*, *xor*, *and*, *not*, арифметических операций и круглых скобок.

Вариант 3. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, десятичных чисел с плавающей запятой[2] в обычном и нормализованном экспоненциальном форматах, строковых литералов, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 4. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, десятичных чисел с плавающей запятой[2] в обычном и нормализованном экспоненциальном форматах, символьных констант[3], знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 5. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из идентификаторов, римских чисел (корректные последовательности заглавных и строчных литер L, X, V и I)[4], десятичных чисел с плавающей точкой в обычном формате[2]. Кроме того, элементами выражений являются знаки присваивания ('='), арифметические операции ('+', '-', '*', '/') и круглые скобки.

Вариант 6. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'). Выражения состоят из идентификаторов, логических констант *0* и *1*, десятичных чисел с плавающей точкой[2] в обычном формате, знаков присваивания ('='), знаков операций *or*, *xor*, *and*, *not* и круглых скобок.

Вариант 7. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, римских чисел (корректные последовательности заглавных и

[1] http://ru.wikipedia.org/wiki/Плавающая_запятая
[2] http://ru.wikipedia.org/wiki/Плавающая_запятая
[3] http://citforum.ru/programming/c/h11.shtml#112
[4] http://ru.wikipedia.org/wiki/Римские_числа

строчных литер L, X, V и I)[4], десятичных чисел с плавающей точкой[2] в обычном формате, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 8. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, римских чисел[4] (корректные последовательности заглавных и строчных литер L, X, V и I), десятичных чисел с плавающей точкой[2] в обычном формате, знаков присваивания ('='), знаков операций сравнения (<, >, =, !=, >=, <=) и круглых скобок.

Вариант 9. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из идентификаторов, целочисленных констант в шестнадцатеричной, восьмеричной и десятичной системах счисления. Кроме того, элементами арифметических выражений являются знаки присваивания ('='), знаки операций ('+', '-', '*', '/', '%') и круглые скобки.

Вариант 10. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Выражения состоят из идентификаторов, целочисленных констант в шестнадцатеричной, десятичной и восьмеричной системах счисления, знаков присваивания ('='), знаков побитовых операций *or*, *xor*, *and*, *not* , арифметических операций и круглых скобок.

Вариант 11. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, целочисленных констант в шестнадцатеричной, десятичной и восьмеричной системах, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 12. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, целочисленных констант в шестнадцатеричной, десятичной и восьмеричной системах, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 13. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из идентификаторов, комплексных чисел[1] (действительная и мнимая части отделяются нечувствительными к регистру символами I или J, и представляют собой два целых и/или десятичных числа с плавающей точкой в обычном формате), целочисленных констант, знаков присваивания ('='), знаков операций ('+', '-', '*', '/') и круглых скобок.

[1] http://ru.wikipedia.org/wiki/Комплексное_число

Вариант 14. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'). Выражения состоят из идентификаторов, нечувствительных к регистру констант *T* и *NIL*, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), знаков присваивания ('='), знаков логических операций *or*, *xor*, *and*, *not* и круглых скобок.

Вариант 15. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, строковых констант (заключенная в двойные кавычки последовательность любых символов, за исключением двойных кавычек), целочисленных констант, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 16. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, целочисленных констант в восьмеричной и шестнадцатеричной системах, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков *b* или *B*), знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 17. Входной язык содержит выражения, разделенные символом точки с запятой (';'). Выражения состоят из идентификаторов, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), десятичных чисел с плавающей точкой[1] с обычном формате, знаков присваивания ('='), знаков операций *or*, *xor*, *and*, *not* и круглых скобок.

Вариант 18. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, дат двух любых (на усмотрение разработчика) форматов, целочисленных констант, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 19. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, временных величин двух любых (на усмотрение разработчика) форматов, целочисленных констант, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 20. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, символьных констант[2], целочисленных констант в

[1] http://ru.wikipedia.org/wiki/Плавающая_запятая
[2] http://citforum.ru/programming/c/h11.shtml#112

десятичной и шестнадцатеричной системе, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=) и круглых скобок.

Вариант 21. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей запятой[6] в обычном формате, целых чисел в восьмеричной и десятичной системах. Кроме того, элементами арифметических выражений являются знаки присваивания (':='), знаки арифметических операций ('+', '-', '*', '/', '%', модуля) и круглые скобки.

Вариант 22. Входной язык содержит логические выражения, разделенные символом точки с запятой (';'). Логические выражения состоят из идентификаторов, нечувствительных к регистру констант *t* (истина) и *f* (ложь), битовых строк (последовательностей из 0 и 1, начинающихся о обязательной пары знаков 0*b* или 0*B*), знаков присваивания (':='), знаков операций *or*, *xor*, *and*, *not*, *стрелка Пирса*[1] (пара знаков '–' и '>' или ключевое слово *peirce*) и круглых скобок.

Вариант 23. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, десятичных чисел с плавающей запятой[2] в нормализованном экспоненциальном формате, целых чисел в восьмеричной и десятичной системах, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<) и круглых скобок.

Вариант 24. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, десятичных чисел с плавающей запятой[9] в обычном и нормализованном экспоненциальном форматах, целочисленных констант, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<) и круглых скобок.

Вариант 25. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из идентификаторов, чисел из римских цифр[3] (корректные последовательности заглавных и строчных литер L, X, V и I). Кроме того, элементами арифметических выражений являются знаки присваивания (':='), знаки операций ('+', '-', '*', '/', '%') и круглые скобки.

Вариант 26. Входной язык содержит логические выражения, разделенные символом точки с запятой (';'). Логические выражения состоят из идентификаторов, логических констант *0* и *1*, целочисленных констант в

[1] http://ru.wikipedia.org/wiki/Стрелка_Пирса
[2] http://ru.wikipedia.org/wiki/Плавающая_запятая
[3] http://ru.wikipedia.org/wiki/Римские_числа

десятичной и шестнадцатеричной системах, знаков присваивания (':='), знаков операций *or*, *xor*, *and*, *not, штрих Шеффера*[1] (знак вертикальной черты или ключевое слово *sheffer*) и круглых скобок.

Вариант 27. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, чисел из римских цифр[10] (корректные последовательности заглавных и строчных литер L, X, V и I), целочисленных констант в восьмеричной и десятичной системах, знаков операций сравнения (>=, <=, =>, =<, ==, <>) и круглых скобок.

Вариант 28. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, чисел из римских цифр[2] (корректные последовательности заглавных и строчных литер L, X, V и I), целочисленных констант в восьмеричной и шестнадцатеричной системах, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<) и круглых скобок.

Вариант 29. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'). Арифметические выражения состоят из идентификаторов, целочисленных констант в двоичной, восьмеричной, десятичной и шестнадцатеричной системах. Кроме того, элементами арифметических выражений являются знаки присваивания (':='), знаки операций ('+', '-', '*', '/', '%', '^') и круглые скобки.

Вариант 30. Входной язык содержит выражения, разделенные символом точки с запятой (';'). Выражения состоят из идентификаторов, целочисленных констант в шестнадцатеричной системе, знаков присваивания (':='), знаков побитовых операций *or*, *xor*, *and*, *not,* <<, >> (сдвиг влево и вправо)[3] и круглых скобок.

Вариант 31. Входной язык содержит операторы условий *if-без-else* и *if-else*, разделенные символом точки с запятой (';'). Операторы условий состоят из идентификаторов, целочисленных констант в шестнадцатеричной системе, знаков присваивания (':='), знаков операций сравнения (<, >, =, ==, <>, !=, >=, <=) и круглых скобок.

Вариант 32. Входной язык содержит операторы цикла с параметром, разделенные символом точки с запятой (';'). Операторы цикла состоят из идентификаторов, целочисленных констант в шестнадцатеричной системе, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), знаков присваивания (':='), знаков операций сравнения

[1] http://ru.wikipedia.org/wiki/Штрих_Шеффера
[2] http://ru.wikipedia.org/wiki/Римские_числа
[3] http://ru.wikipedia.org/wiki/Битовый_сдвиг

(<, >, =, ==, <>, !=, >=, <=) и круглых скобок.

Приложение А. attribute.h. Заголовочный файл с описанием структуры атрибутов узлов дерева разбора

```
#ifndef __ATTRIBUTE_H__
#define __ATTRIBUTE_H__

/**
 * attribute.h
 *     Структуры данных для манипуляции атрибутами, в частности узлов дерева разбора.
 *
 */

/**
 * Простой атрибут, ассоциированный с узлом дерева разбора.
 * Тип атрибута в этом случае придется отслеживать вручную.
 * В качестве альтернативы можно использовать типизированную
 * версию данной структуры.
 */
typedef union Attribute
{
    int    ival;
    char*  sval;
} Attribute;

/**
 * Именованная версия приведенной выше структуры атрибута. Обычно используется
 * для множеств атрибутов.  Мы предполагаем, что имя подразумевает тип,
 * таким образом, именованные атрибуты не имеют явно заданных типов.
 */
typedef struct NamedAttribute
{
    char*     name;
    Attribute val;
} NamedAttribute;

/**
 * Набор атрибутов. Он может быть реализован как hash-таблица.
 * Однако, поскольку обычно требуется небольшое количество атрибутов,
 * то эффективнее использвать всего лишь массив именованных атрибутов.
 */
typedef struct AttributeSet
{
    int capacity;               // Начальное количество атрибутов
    int size;                        // Как много атрибутов хранится
    NamedAttribute* contents;   // Сами атрибуты
} AttributeSet;



// +-----------------------------------------------+
// | Прототипы функций работы с наборами атрибутов |
// +-----------------------------------------------+

/**
 * Создается новый набор атрибутов, количество которых ограничено
 * значением capacity.
 *
 * В случае ошибки возвращает значение NULL.
 */
AttributeSet* CreateAttributeSet(int capacity);
```

```
/**
 * Освобждается память, занятая набором атрибутов
 */
void FreeAttributeSet(AttributeSet* set);

/**
 * Функции, задающие значение атрибута конкретного типа (integer, ...).
 * Возвращает  1 в случае успеха и 0 в случае ошибки.
 */
int SetAttributeValue(AttributeSet* set, char* name, Attribute att);
int SetAttributeValueInteger(AttributeSet* set, char* name, int ival);
int SetAttributeValueString(AttributeSet* set, char* name, char* sval);

/**
 * Функции получения значения атрибута. Если нет атрибута с заданным именем,
 * то возвращается непредсказуемое значение.
 */
Attribute GetAttributeValue(AttributeSet* set, char* name);
int GetAttributeValueInteger(AttributeSet* set, char* name);
char* GetAttributeValueString(AttributeSet* set, char* name);

/**
 * Определяет, имеется ли в наборе атрибут с заданным именем.
 */
int HasAttribute(AttributeSet* set, char* name);

#endif // __ATTRIBUTE_H__
```

Приложение Б. attribute.c. Функции для выполнения действий над атрибутами узлов дерева разбора

```
/**
 * attribute.c
 *   Реализация функциональности структур данных для манипуляции атрибутами,
 *   в частности узлов дерева разбора.
 *
 */

#ifndef YYBISON
#include "attribute.h"
#endif
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

/**
 * Отыскивает индекс атрибута в наборе. Если он не найден, то возвращается -1.
 */
static int FindAttribute(AttributeSet* set, char* name)
{
    int i;
    for (i = 0; i < set->size; ++i)
    {
        if (0 == strcmp(set->contents[i].name, name))
        {
           return i;
        }
    }
  // Этот оператор выполняется, если предыдущий цикл был полностью выполнен.
  return -1;
}



/**
 * Получение атрибута.
 */
Attribute GetAttributeValue(AttributeSet* set, char* name)
{
    return set->contents[FindAttribute(set, name)].val;
}

/**
 * Установка значения атрибута.
 */
int SetAttributeValue(AttributeSet* set, char* name, Attribute att)
{
    // Смотрим, есть ли такой атрибут в наборе.
    int index = FindAttribute(set, name);

    // Если есть, то изменяем значение.
    if (index >= 0)
    {
        set->contents[index].val = att;
        return 1;
    }

    // Если нет места для нового атрибута, то сдаемся.
    if (set->size >= set->capacity)
        return 0;
```

```
    // Размещаем новый атрибут в наборе.
    index = (set->size)++;
    set->contents[index].name = name;
    set->contents[index].val = att;

    return 1;
}



// +-------------------------+
// | Экспортируемые функции |
// +-------------------------+

void FreeAttributeSet(AttributeSet* set)
{
    if (0 < set->capacity)
      {
            free(set->contents);
            set->contents = NULL;
      }
    set->size = 0;
    set->capacity = 0;
    free(set);
      set = NULL;
}

int GetAttributeValueInteger (AttributeSet* set, char* name)
{
    return (GetAttributeValue(set, name)).ival;
}

char* GetAttributeValueString(AttributeSet* set, char* name)
{
    return (GetAttributeValue(set, name)).sval;
}

int HasAttribute(AttributeSet* set, char* name)
{
    return (FindAttribute(set, name) != -1);
}

AttributeSet* CreateAttributeSet(int capacity)
{
    int i;
    AttributeSet* result = (AttributeSet*) malloc(sizeof(AttributeSet));
    if (result == NULL)
        return NULL;
    if (capacity > 0)
    {
        result->contents     =     (NamedAttribute*)     malloc(capacity     *
sizeof(NamedAttribute));
        if (result->contents == NULL)
        {
            free(result);
            return NULL;
        }
        for (i = 0; i < capacity; ++i)
            result->contents[i].name = "";
    }
    result->capacity = capacity;
```

```
    result->size = 0;
    return result;
}

int SetAttributeValueInteger(AttributeSet* set, char* name, int ival)
{
    // Можно воспользоваться приведением типов, но
    // мы пробуем более безопасный способ.
    Attribute att;
    att.ival = ival;
    return SetAttributeValue(set, name, att);
}

int SetAttributeValueString(AttributeSet* set, char* name, char* sval)
{
    Attribute att;
    att.sval = sval;
    return SetAttributeValue(set, name, att);
}
```

Приложение В. parse-tree.h. Структуры данных для узлов дерева разбора

```
#ifndef __PARSE_TREE_H__
#define __PARSE_TREE_H__

/**
 * parse-tree.h
 *   Структуры данных для дерева разбора.
 *
 */

// +-------------+
// | Комментарии |
// +-------------+

/*
  Мы представляем узлы с использованием традиционной для язык С стратегии.
  Каждый объект начинается с целого числа - типа узла. Обычно они генерируются
  случайным образом, однако, т.к. у нас очень мало типов, то используются
  предопределенные числа. Такая стратегия должна бы немного сложнее
  для реализации проверки типов. Это вероятно будет сделано в будущих версиях.
  Наследование поддерживается через включение структур.  Поскольку стандарт С
  говорит, что элементы структур упорядочены, то указывая элемент суперкласса
  первым, мы гарантируем, что его поля будут содержаться внутри.
 */



#include <stdio.h>
#include "attribute.h"



/**
 * У абстрактных узлов есть тип (для объектов), символ (терминал
 * или нетерминал) и набор атрибутов.
 */
extern int TYPE_NODE;
typedef struct Node
{
    int type;
    int symbol;
    AttributeSet* attributes;
} Node;

/**
 * У узлов, содержащих терминалы/лексемы нет дополнительной информации.
 * Тем не менее, обеспечивается отдельный тип для ясности.
 */
extern int TYPE_TNODE;
typedef struct TNode
{
    Node parent;
} TNode;

/**
 * Узлы, содержащие нетерминалы, могут иметь набор прямых потомков.
 */
extern int TYPE_NNODE;
typedef struct NNode
{
    Node parent;
```

```
    int arity;          // Количество прямых потомков
    Node** children;    // Поддеревья
} NNode;



// +-------------------+
// | Прототипы функций |
// +-------------------+

/**
 * Освобождение памяти, занятой под дерево разбора.
 */
void FreeTree(Node* tree);

/**
 * Получение арности корня дерева.
 */
int GetNodeArity(Node* tree);

/*
 * Получение i-го поддерева заданного дерева.
 */
Node* GetNodeChild(Node* tree, int i);

/**
 * Является ли узел нетерминалом (NNode).
 */
int IsNonterminalNode(Node* tree);

/**
 * Является ли узел терминалом (TNode).
 */
int IsTerminalNode(Node* tree);

/**
 * Создает новый нетерминальный узел, который содержит заданную информацию.
 * Возвращает узел в случае успеха и NULL в противном случае.
 */
Node* CreateNonterminalNode(int nonterm, int arity, AttributeSet* attributes);

/**
 * Создает новый терминальный узел, который содержит заданную информацию.
 * Возвращает узел в случае успеха и NULL в противном случае.
 */
Node* CreateTerminalNode(int term, AttributeSet* attributes);

/**
 * Печатает дерево в выходной поток.
 */
void PrintTree(FILE* stream, Node* node);

/**
 * Задает i-го потомка узла дерева.
 * Возвращает 1 в случае успеха и 0 в противном случае.
 */
int SetNodeChild(Node* node, int i, Node* child);

#endif
```

Приложение Г. parse-tree.c. Функции обработки узлов дерева разбора

```
/**
 * parse-tree.c
 *   Функции работы с простой структурой дерева разбора.
 *
 */

// +-------------+
// | Комментарии |
// +-------------+

/*
  1. Данный код зависим от значений определенных Bison/Yacc, поэтому
     этот файл нужно включить директивой #include внутри .y-файла, вместо
     его компиляции.

  2. Этот код зависит от функций определенных в attribute.h и attribute.c,
     поэтому данный файл нужно связать (линкером) с этими двумя файлами.

  3. Из-за этих и других зависимостей, в .y-файле нужно включить
     <stdio.h>, "parse-tree.h" и "attribute.h".
 */



// +---------------------------+
// | Экспортируемые переменные |
// +---------------------------+

int TYPE_NODE = 60000;
int TYPE_TNODE = 70000;
int TYPE_NNODE = 80000;



// +------------+
// | Прототоипы |
// +------------+

static void PrintTreeIndented(FILE* stream, Node* node, int spaces);



/**
 * Вспомогательная функция для PrintAttributes, которая печатает правильный
префикс.
 */
static void PrintPrefix(FILE* stream, int* printed)
{
    if (*printed)
        fprintf(stream, ", ");
    else
        fprintf(stream, " [");
    ++(*printed);
}

/**
 * Печатает набор атрибутов.
 */
static void PrintAttributes(FILE* stream, AttributeSet* attributes)
{
```

```
    int printed = 0;
    if (attributes == NULL)
        return;
    if (HasAttribute(attributes, "name"))
    {
        PrintPrefix(stream, &printed);
        fprintf(stream,     "name:'%s'",     GetAttributeValueString(attributes,
"name"));
    }
    if (HasAttribute(attributes, "text"))
    {
        PrintPrefix(stream, &printed);
        fprintf(stream,     "text:'%s'",     GetAttributeValueString(attributes,
"text"));
    }

    if (printed)
        fprintf(stream, "]");
}

/**
 * Печатает составной узел, дополненный некоторым количеством пробелов.
 * Как в функции PrintTreeIndented, ожидается, что первые дополнения
 * уже напечатаны.
 */
static void PrintNonterminalNode(FILE* stream, NNode* nn, int spaces)
{
    Node* node = (Node *)nn;

    if (node == NULL)
    {
        fprintf(stream, "*** ERROR: Null NNode ***\n");
        return;
    }

    // Печатает текущий узел дерева
     fprintf(stream, "%s/%d", nonterm_names[node->symbol], nn->arity);
     PrintAttributes(stream, node->attributes);
     fprintf(stream, "\n");

    // Печатает всех непосредственных потомков.
    int child_indent = spaces + 4;
    int i;
    for (i = 0; i < nn->arity; ++i)
    {
        fprintf(stream, "%*d ", child_indent, i+1);
        PrintTreeIndented(stream, nn->children[i], child_indent);
    }
}

/**
 * Печатает терминальный узел.
 */
static void PrintTerminalNode(FILE* stream, TNode* tn)
{
    int symbol = ((Node *) tn)->symbol;
    AttributeSet* attributes = ((Node *) tn)->attributes;

    switch (symbol)
    {
    case _INTEGER:
```

```
        fprintf(stream, "INT_CONST");
        break;

    case _IDENTIFIER:
        fprintf(stream, "IDENTIFIER");
        break;

    case _BEGIN:
        fprintf(stream, "BEGIN");
        break;

    case _END:
        fprintf(stream, "END");
        break;

    case _SEMI:
        fprintf(stream, "SEMICOLON");
        break;

    case _PLUSOP:
        fprintf(stream, "PLUS_OPERATOR");
        break;

    default:
        if ((symbol < TOKENS_START) || (symbol > TOKENS_END))
            fprintf(stream, "*invalid*(%d)", symbol);
        else
          fprintf(stream, "%s", yytname[symbol]);
        break;
    }

  PrintAttributes(stream, attributes);
  fprintf(stream, "\n");
}

/**
 * Печатает дерево, дополненное некоторым количеством пробелов.
 * Ожидается, что дополнение уже напечатано на первой строке.
 */
static void PrintTreeIndented(FILE* stream, Node* node, int spaces)
{
    if (node == NULL)
    {
        fprintf(stream, "*** ERROR: Null tree. ***\n");
    }
    else if (IsTerminalNode(node)) // Если это лист дерева.
    {
        PrintTerminalNode(stream, (TNode *) node);
    }
    else if (IsNonterminalNode (node)) // если это составной узел дерева.
    {
      PrintNonterminalNode(stream, (NNode *) node, spaces);
    }
    else
    {
      fprintf(stream, "*** ERROR: Unknown node type %d. ***\n", node->type);
    }
}
```

```
// +------------------------+
// | Экспортируемые функции |
// +------------------------+

void FreeTree(Node* node)
{
    if (NULL == node)
        return;
    if (IsNonterminalNode(node)) // Если это лист.
    {
        int i;
        NNode *nn = ((NNode *) node);
        for (i = 0; i < nn->arity; ++i)
            FreeTree(nn->children[i]);
        if (nn->children != NULL)
            {
            free(nn->children);
                  nn->children = NULL;
            }

    }

    if (node->attributes != NULL)
        FreeAttributeSet(node->attributes);
  free(node);
  node = NULL;
}

int GetNodeArity(Node* node)
{
    if (0 == IsNonterminalNode(node))
       return 0;
    NNode* nn = (NNode *) node;
    return nn->arity;
}

Node* GetNodeChild(Node* node, int i)
{
    if (0 == IsNonterminalNode(node))
        return NULL;
    NNode* nn = (NNode *) node;
    if (nn->arity <= i)
        return NULL;
    return nn->children[i];
}

int IsNonterminalNode(Node* node)
{
    return node->type == TYPE_NNODE;
}

int IsTerminalNode(Node* node)
{
    return node->type == TYPE_TNODE;
}

Node* CreateNonterminalNode(int nonterm, int arity, AttributeSet* attributes)
{
    NNode *nn = (NNode *)malloc (sizeof (NNode));
    if (NULL == nn)
        return NULL;
```

```
    Node* node = (Node *) nn;
    node->type = TYPE_NNODE;
    node->symbol = nonterm;
    node->attributes = attributes;
    nn->arity = arity;
    nn->children = (Node**)malloc(arity * sizeof(Node *));
        return node;
}

Node* CreateTerminalNode(int term, AttributeSet* attributes)
{
    Node* tn = (Node *)malloc(sizeof(TNode));
    if (NULL == tn)
       return NULL;
    Node* node = (Node *) tn;
    node->type = TYPE_TNODE;
    node->symbol = term;
    node->attributes = attributes;
    return node;
}

void PrintTree(FILE* stream, Node* node)
{
    PrintTreeIndented (stream, tree, 0);
}

int SetNodeChild(Node* node, int i, Node* child)
{
    if (0 == IsNonterminalNode(node))
        return 0;
    NNode* nn = (NNode *) node;
    if (nn->arity <= i)
        return 0;
    nn->children[i] = child;
    return 1;
}
```

Приложение Д. nonterminals.c. Макроопределения нетерминалов заданного языка

```
/**
 * nonterminals.c
 *   Список всех нетерминалов заданного языка.  Используется для автоматизации
 *   отображения нетерминальных символов на имена.
 */

{
    NONTERMINAL(compound_statement),
    NONTERMINAL(statement),
    NONTERMINAL(start),
    NONTERMINAL(cons),
    NONTERMINAL(empty_statement),
    NONTERMINAL(epsilon),
    NONTERMINAL(expr),
    NONTERMINAL(program),
    NONTERMINAL(statement_list),
    NONTERMINAL(statement_list_tail),
    NONTERMINAL(NONTERMINALS)

};
```

Приложение Е. lexer.ll. Лексический анализатор заданного языка

```
/**
 * lexer.ll
 *   flex-спецификация лексического анализатора простого языка.
 *
 */

%{
#if defined _WIN32
#include <io.h>              // Для isatty
#elif defined _WIN64
#include <io.h>              // Для isatty
#endif
#ifdef MSVC
#define  isatty _isatty      // В VC isatty назван _isatty
#endif
%}

%option nounistd
%%

        /* Пропускаются все пробельные символы.
           Аналогично будет выглядеть код для комментариев. */
[ \t\n]         { }

        /* Целочисленные константы. */
0               { return _INTEGER; }
[1-9][0-9]*     { return _INTEGER; }

        /* Ключевые слова. */
begin           { return _BEGIN; }
end             { return _END; }

        /* Знаки пунктуации, пока здесь только точка с запятой. */
";"             { return _SEMI; }
"+"             { return _PLUSOP; }

        /* Идентификаторы. Это правило должно идти после, но ни в коем случае
         * до шаблона, представляющего то, что может быть идентификатором,
         * например, ключевые слова.
         */
[a-z][a-z0-9_]* { return _IDENTIFIER; }
.               { yyerror("Not in alphabet."); }
%%
int yywrap() { return 1; }
```

Приложение Ё. language.yy. bison-спецификация синтаксического анализатора заданного языка

```
/**
 * language.yy
 *   Синтаксический анализатор очень простого языка,
 *   выводящий на экран дерево разбора.
 *
 */

%no-lines
%verbose
%require "2.5"

%{
#include <ctype.h>              // Для tolower
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

#include "attribute.h"
#include "parse-tree.h"
%}



// +----------------------------+
// | Определения для Bison/Yacc |
// +----------------------------+

/* Поскольку мы пытаемся построить дерево разбора,
 * нужно, чтобы с каждым элементом была связана часть дерева.
 */

%{
#define YYSTYPE Node*
%}

/* Таблица лексем. */
%token_table

/* Все лексемы, которые используются. */
%token TOKENS_START
%token _IDENTIFIER
%token _SEMI
%token _BEGIN
%token _END
%token _PLUSOP
%token _INTEGER
%token TOKENS_END

%left _PLUSOP

%{

/*  Несколько внешних программных объектов от лексера, которые нужно
предварительно объявить. */
extern char* yytext;
extern int yylineno;
```

```
extern int yylex(void);

/* Определяем функцию yyerror. */
void yyerror(char* err)
{
    fprintf(stderr, "Line %d - \"%s\"\n", yylineno, err);

    // Не очень хорошо поступаем, не освобождая память перед выходом.
    // Для завершения работы после первой ошибки нужно раскомментировать код на
следующей строке.
    // exit (1);
}

%}



// +---------------------------+
// | Информация о нетерминалах |
// +---------------------------+

%{

/* "Волшебный тип", который позволит ссылаться на нетерминалы по имени. */
#define NONTERMINAL(NAME) _ ## NAME
enum nonterms
#include "nonterminals.c"
typedef enum nonterms nonterms;
#undef NONTERMINAL



#define NONTERMINAL(NAME) #NAME
char* nonterm_names[] =
#include "nonterminals.c"

%}



// +-------------------------------+
// | Глобальные объекты Bison/Yacc |
// +-------------------------------+

%{
Node* tree;
%}



// +-----------------------------------+
// | Процедуры поддержки дерева разбора |
// +-----------------------------------+

%{
/**
 * Создает новый промежуточный узел с нужным числом непосредственных потомков.
 */
Node* CreateInteriorNode(int nonterm, AttributeSet* attributes, int arity, ...)
{
    va_list children;
    int c;
```

```
    Node* child;

    // Обработка переменного числа аргументов
    va_start(children, arity);

    // Создается новый узел дерева.
    Node* node = CreateNonterminalNode(nonterm, arity, attributes);

    // Добавляются непосредственные потомки.
    for (c = 0; c < arity; ++c)
    {
        child = va_arg(children, Node *);
        SetNodeChild(node, c, child);
    }

    // Убираемся за собой после обработки переменного числа аргументов.
    va_end (children);

    // Здесь все сделано.
    return node;
}
%}

// +--------------------+
// | Обработка списков |
// +--------------------+

%{
/**
 * Создает список из головы и хвоста.
 */
Node* ConstructList(Node* car, Node* cdr)
{
    return CreateInteriorNode(_cons, NULL, 2, car, cdr);
}

/**
 * Определяет длину списка.
 */
int GetListLength(Node* lst)
{
    int length = 0;
    while (lst->symbol != _epsilon)
    {
        lst = GetNodeChild(lst, 1);
        ++length;
    }
    return length;
}

/**
 * По заданному непустому списку строит узел дерева, который содержит
 * все элементы списка. По мере преобразования освобождает память,
 * занятую списком.
 */
Node* ConvertListToNode(int type, AttributeSet* attributes, Node* lst)
{
    int len;                        // Общая длина списка
    Node* parent;                   // Узел дерева, который строится
    Node* tmp;                      // Временный узел
```

```
    len = GetListLength(lst);
    parent = CreateNonterminalNode(type, len, attributes);
    int child = 0;

    while (lst->symbol != _epsilon)
    {
        SetNodeChild(parent, child++, GetNodeChild(lst, 0));
        tmp = lst;
        lst = GetNodeChild(lst, 1);
        if (tmp->attributes != NULL)
            FreeAttributeSet(tmp->attributes);
        free(tmp);
    }

    if (lst->attributes != NULL)
        FreeAttributeSet(lst->attributes);
    free (lst);

    return parent;
}

/**
 * Создает узел для пустой (epsilon) цепочки.
 */
Node* CreateEpsilonNode(void)
{
    return CreateNonterminalNode(_epsilon, 0, NULL);
}
%}



// +----------------------+
// | Различные полезности |
// +----------------------+

%{
/**
 * Преобразует строку к нижнему регистру.
 */
void ConvertStringToLowerCase(char* str)
{
    while (*str != '\0')
    {
      *str = tolower(*str);
      ++str;
    }
}
%}

// +----------------------+
// | Продукции грамматики |
// +----------------------+

%%
start : program      { $$ = CreateInteriorNode(_start, NULL, 1, $1); tree = $$;
}
      ;

program : statement
```

```
          { $$ = CreateInteriorNode(_program, NULL, 1, $1); }
      ;

statement
    : /* epsilon-правило */
      { $$ = CreateNonterminalNode (_empty_statement, 0, NULL); }
    | expr
      { $$ = CreateInteriorNode(_statement, NULL, 1, $1); }
    | compound_statement
      { $$ = CreateInteriorNode(_statement, NULL, 1, $1); }
    ;

expr :
      _IDENTIFIER
      {
          AttributeSet* attributes = CreateAttributeSet(1);
          char* name = strdup(yytext);
          ConvertStringToLowerCase(name);
          SetAttributeValueString(attributes, "name", name);
          Node* nodeTerminal = CreateTerminalNode(_IDENTIFIER, attributes);
          Node* node = CreateInteriorNode(_expr, NULL, 1, nodeTerminal);
          $$ = node;
      }
     |
      _INTEGER
      {
          AttributeSet* attributes = CreateAttributeSet(1);
          char* name = strdup(yytext);
          SetAttributeValueString(attributes, "text", name);
          Node* nodeTerminal = CreateTerminalNode(_INTEGER, attributes);
          Node* node = CreateInteriorNode(_expr, NULL, 1, nodeTerminal);
          $$ = node;
      }
     |
      expr _PLUSOP expr
      {
          AttributeSet* attributes = CreateAttributeSet(1);
          SetAttributeValueString(attributes, "text", "+");
          Node* nodeTerminal = CreateTerminalNode(_PLUSOP, attributes);
          Node* node = CreateInteriorNode(_expr, NULL, 3, $1, nodeTerminal, $3);
          $$ = node;
      }
    ;

compound_statement
    : _BEGIN statement_list _END
      {
          AttributeSet* attributes1 = CreateAttributeSet(1);
          SetAttributeValueString(attributes1, "name", "begin");
          Node* nodeTerminal1 = CreateTerminalNode(_BEGIN, attributes1);

          AttributeSet* attributes2 = CreateAttributeSet(1);
          SetAttributeValueString(attributes2, "name", "end");
          Node* nodeTerminal2 = CreateTerminalNode(_END, attributes2);

          $$ = CreateInteriorNode(_compound_statement, NULL, 3, nodeTerminal1,
$2, nodeTerminal2);
      }
    ;

statement_list
    : statement statement_list_tail
```

```
      {
          AttributeSet *attributes = CreateAttributeSet(0);
          $$ = ConvertListToNode(_statement_list, attributes, ConstructList($1,
$2));
      }
    ;

statement_list_tail
    : /* epsilon-правило */
      { $$ = CreateEpsilonNode(); }
    | _SEMI statement statement_list_tail
      {
          AttributeSet* attributes = CreateAttributeSet(1);
          SetAttributeValueString(attributes, "name", ";");
          Node* nodeTerminal = CreateTerminalNode(_SEMI, attributes);

          $$ = CreateInteriorNode(_statement_list_tail, NULL, 2, nodeTerminal,
ConstructList($2, $3));
      }
    ;

%%

// +--------------------+
// | Дополнительный код |
// +--------------------+

/* Подключаем лексический анализатор. */
#include "lex.yy.c"

/* Код для управления атрибутами. */
#include "attribute.c"

/* Код для построения дерева разбора. */
#include "parse-tree.c"
```

Приложение Ж. print-tree.c. Вывод на экран дерева разбора цепочки заданного языка

```
/**
 * print-tree.c
 *    Простой анализатор, который строит дерево разбора и печатает в выходной поток.
 *
 */


#include <stdlib.h>

#include "parse-tree.h"                 // Для функций PrintTree и FreeTree

// Это получаем от Bison/Yacc
extern int yyparse(void);

// Это строит наш анализатор
extern Node* tree;

int main(int argc, char* argv[])
{
    yyparse();
    PrintTree(stdout, tree);
    FreeTree(tree);
    exit (0);
}
```