Лабораторная работа №4. Синтаксический анализатор

Теоретические сведения

В иерархии грамматик Хомского выделено 4 основных группы языков (и описывающих их грамматик). При этом наибольший интерес представляют регулярные и контекстно-свободные (КС) грамматики и языки. Они используются при описании синтаксиса языков программирования. С помощью регулярных грамматик можно описать лексемы языка – идентификаторы, константы, служебные слова и прочие. На основе КС-грамматик строятся более крупные синтаксические конструкции: описания типов и переменных, арифметические и логические выражения, управляющие операторы, и, наконец, полностью вся программа на входном языке.

Входные цепочки регулярных языков распознаются с помощью конечных автоматов (КА). Они лежат в основе сканеров, выполняющих лексический анализ и выделение слов в тексте программы на входном языке. Результатом работы сканера является преобразование исходной программы в список или таблицу лексем. Дальнейшую её обработку выполняет другая часть компилятора – синтаксический анализатор. Его работа основана на использовании правил КС-грамматики, описывающих конструкции исходного языка.

> **Синтаксический анализатор** – это часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка.

Синтаксический анализатор получает строку токенов от лексического анализатора, как показано на рисунке 1, и проверяет, может ли эта строка токенов порождаться грамматикой входного языка. Ещё одной функцией синтаксического анализатора является генерация сообщений обо всех выявленных ошибках, причём достаточно внятных и полных, а кроме того, синтаксический анализатор должен уметь обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. В случае корректной программы синтаксический анализатор строит дерево разбора и передаёт его следующей части компилятора для дальнейшей обработки.

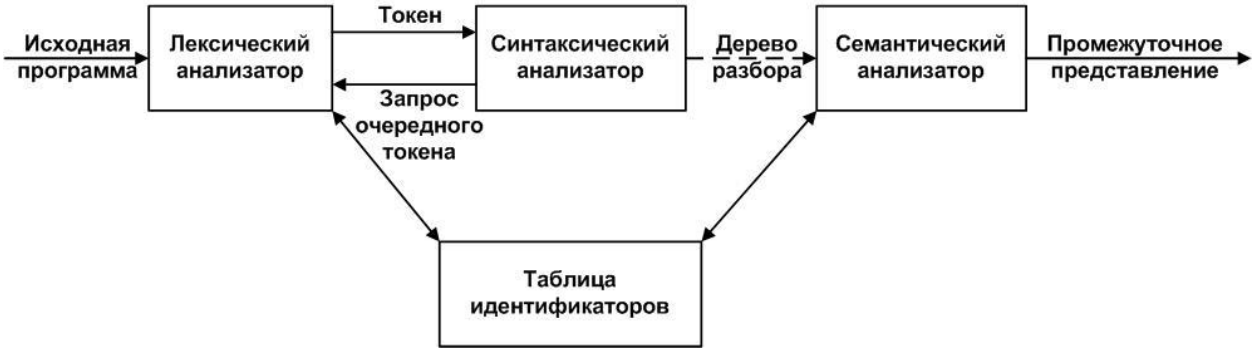

Рисунок 1. Место синтаксического анализатора в структуре компилятора

Распознавание цепочек КС-языков

Рассмотрим алгоритмы, лежащие в основе синтаксического анализа. Перед синтаксическим анализатором стоят две основные задачи: проверить правильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать её в вид, удобный для дальнейшей семантической (смысловой) обработки и генерации кода. Одним из способов такого представления является *дерево синтаксического разбора*.

Основой для построения распознавателей КС-языков являются **автоматы с магазинной памятью** – МП-автоматы – односторонние недетерминированные распознаватели с линейно-ограниченной магазинной памятью.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные «магазинные» символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от одного или нескольких верхних символов стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

При выполнении перехода МП-автомата из одной конфигурации в другую из стека удаляются верхние символы, соответствующие условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека. Допускаются переходы, при которых входной символ игнорируется (и тем самым он будет входным символом при следующем переходе). Эти переходы называются **ε-переходами**. Если при окончании цепочки автомат находится в одном из заданных конечных состояний, а стек пуст, цепочка считается принятой (после окончания цепочки могут быть сделаны ε-переходы). Иначе цепочка символов не принимается.

МП-автомат называется *недетерминированным,* если при одной и той же его конфигурации возможен более чем один переход. В противном случае (если из любой конфигурации МП-автомата по любому входному символу возможно не более одного перехода в следующую конфигурацию) МП-автомат считается *детерминированным* (ДМП-автоматом). ДМП-автоматы задают класс детерминированных КС-языков, для которых существуют однозначные КС-грамматики. Именно этот класс языков лежит в основе синтаксических конструкций всех языков программирования, так как любая синтаксическая конструкция языка программирования должна допускать только однозначную трактовку.

По произвольной КС-грамматике **G** (**VN**, **VT**, **P**, *S*), **V** = **VT** ∪ **VN** всегда можно построить недетерминированный МП-автомат, который допускает цепочки языка, заданного этой грамматикой. А на основе этого МП-автомата можно создать распознаватель для заданного языка.

Однако при алгоритмической реализации функционирования такого распознавателя могут возникнуть проблемы. Дело в том, что построенный МП-автомат будет, как правило, недетерминированным, а для МП-автоматов, в отличие от обычных КА, не существует алгоритма, который позволял бы преобразовать произвольный МП-автомат в ДМП-автомат. Поэтому программирование функционирования МП-автомата – нетривиальная задача. Если моделировать его функционирование по шагам с перебором всех возможных состояний, то может оказаться, что построенный для тривиального МП-автомата алгоритм никогда не завершится на конечной входной цепочке символов при определённых условиях.

Поэтому для построения распознавателя для языка, заданного КС-грамматикой, рекомендуется воспользоваться соответствующим математическим аппаратом и одним из существующих алгоритмов.

С точки зрения построения дерева синтаксического разбора алгоритмы синтаксического анализа можно разделить на два класса:

1) *алгоритмы нисходящего синтаксического анализа*. Эти алгоритмы строят дерево синтаксического разбора от корня к листьям. На каждом шаге к узлу, представляющему собой нетерминальный символ грамматики, добавляется поддерево, являющееся правой частью правила для данного нетерминала;
2) *алгоритмы восходящего синтаксического анализа* строят дерево синтаксического разбора от листьев к корню. На каждом шаге на текущем уровне дерева находится последовательность узлов, соответствующая правой части какого-либо правила грамматики. Нетерминальный символ, стоящий слева в этом правиле, становится новым узлом дерева.

Какой бы из этих алгоритмов ни использовался, основная трудность при реализации алгоритма синтаксического анализа состоит в том, какое правило грамматики выбрать.

Генератор синтаксических анализаторов Bison

Bison – программный инструментарий, предназначенный для генерации синтаксических анализаторов на основе LALR(1)-грамматик.

LALR(1)-грамматики (LA (англ. lookahead) – предпросмотр, L (англ. left) – порядок чтения входной цепочки символов слева направо, R (англ. right) – в результате работы алгоритма получается правосторонний вывод, 1 – количество символов предпросмотра) относятся к классу контекстно-свободных грамматик, к которым применимы восходящие алгоритмы синтаксического анализа.

Лексический анализатор, сгенерированный Flex, является сопрограммой для синтаксического анализатора, генерируемого с помощью Bison (рис. 2). Лексический анализатор возвращает синтаксическому поток токенов. Задача распознавателя выяснить взаимоотношения между этими токенами. Обычно такие взаимоотношения отображаются в виде *дерева разбора*.

Например, для арифметического выражения `1 * 2 + 3 * 4 + 5` получится дерево разбора, изображённое на рисунке 3. Операция умножения имеет более высокий приоритет, чем операция сложения, поэтому первыми будут операции `1 * 2` и `3 * 4`. Затем результаты этих двух операций складываются вместе, и эта сумма складывается с `5`. Каждая ветвь дерева показывает взаимоотношение между токенами или поддеревьями.

Любой синтаксический анализатор, сгенерированный с использованием Bison, формирует дерево разбора по мере анализа входных токенов.

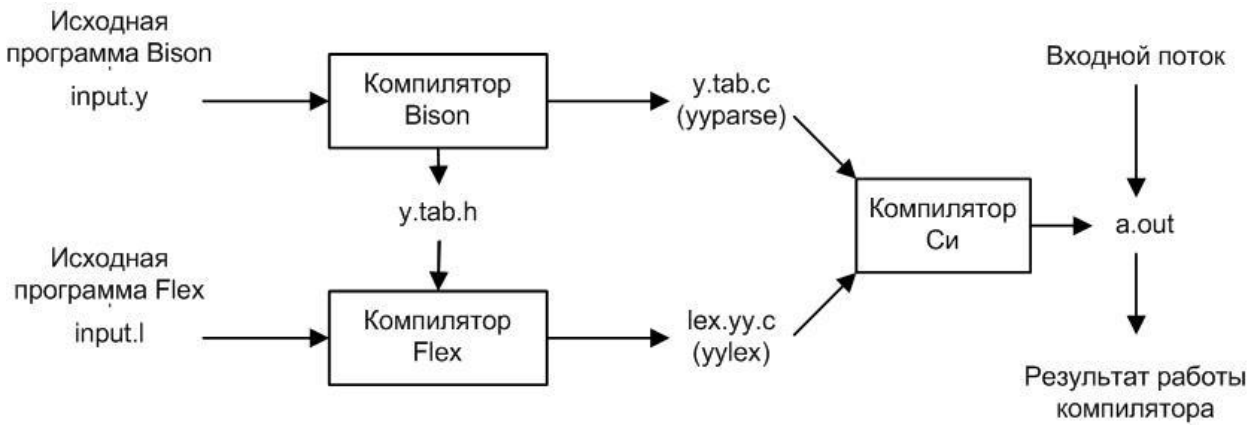

Рисунок 2. Схема совместной работы Flex и Bison

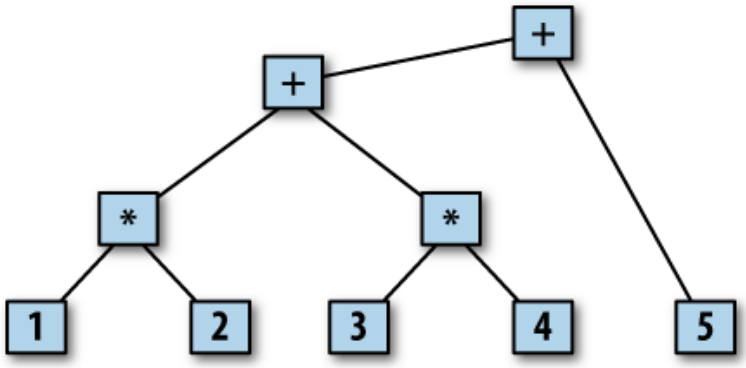

Рисунок 3. Дерево разбора для арифметического выражения 1 * 2 + 3 * 4 + 5

Как работает распознаватель Bison

Bison работает с грамматикой, которая определяется во входном файле, и создаёт синтаксический анализатор, распознающий «предложения», соответствующие этой грамматике. Для грамматики языка программирования такими предложениями будут синтаксически правильные программы на данном языке.

Синтаксически правильная программа не всегда является семантически правильной. Например, для языка Си присваивание строкового значения переменной типа `int` – семантически неверно, но удовлетворяет синтаксическим правилам языка. Bison проверяет только правильность синтаксиса.

Описание грамматики на языке Bison и его соответствие форме Бэкуса-Наура приведено в таблице 1. Вертикальная черта (|) показывает, что существует две возможности задания одного и того же нетерминального символа или что несколько правил могут иметь одинаковую левую часть. Символы в левой части правила – нетерминалы. Символы, возвращаемые лексическим анализатором, – терминалы или токены (в таблице 1 таким символом является `NAME`). Нельзя использовать одинаковые имена для терминальных и нетерминальных символов.

Таблица 1. Описание грамматики на языке Bison

Пример грамматики (Bison)	Форма Бэкуса-Наура

`statement: NAME'='expression`
`expression: NUMBER'+'NUMBER`
`| NUMBER'-'NUMBER`
	`statement → NAME'='expression`
	`expression → NUMBER'+'NUMBER`

Пример грамматики (Bison)	Форма Бэкуса-Наура
	| NUMBER'-'NUMBER

Дерево разбора для предложения `fred = 12 + 13`, соответствующего грамматике, приведенной в таблице 1, показано на рисунке 4.

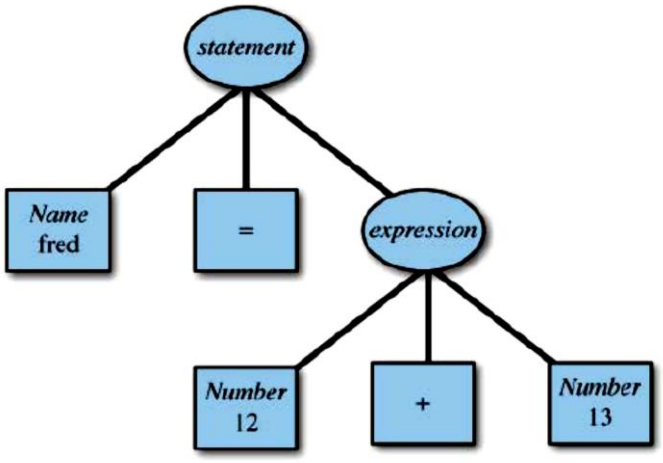

Рисунок 4. Дерево разбора для цепочки символов fred = 12 + 13

В этом примере `12 + 13` соответствует нетерминалу `expression`, а `fred = expression` формирует `statement`.

Каждая грамматика содержит начальный символ, который выступает в качестве корня дерева разбора. В данной грамматике `statement` является таким символом.

Правила могут явным или неявным образом ссылаться сами на себя. Такое свойство позволяет разбирать входные предложения любой длины. Грамматика из таблицы 1 может быть расширена для представления более длинных арифметических выражений:

```
statement: NAME'='expression
expression: expression'+'NUMBER
            | expression'-'NUMBER
            | NUMBER
```

Теперь последовательность вида `fred = 14 + 23 - 11 + 7` может быть успешно разобрана, как это показано на рисунке 5.

Распознаватель, генерируемый Bison, в процессе своей работы ищет правила, подходящие для получаемых им от лексического анализатора токенов. Он создает набор состояний, каждое из которых отражает возможную позицию в одном или нескольких частично распознанных правилах. Каждый раз при прочтении очередного токена, синтаксический анализатор помещает его во внутренний стек и переходит к новому состоянию, соответствующему этому токену. Это действие распознавателя называют *сдвигом*. Когда все символы, образующие правую часть правила, найдены, эти символы выталкиваются из стека, символ из левой части данного правила помещается в стек, а распознаватель переходит в состояние, соответствующее этому символу на верхушке стека. Такое действие называют *свёрткой*. При каждой свёртке Bison выполняет пользовательский программный код, связанный с этим правилом.

Рассмотрим пример разбора входной последовательности `fred = 12 + 13` в соответствии с правилами из таблицы 1. Синтаксический анализатор получает токены от лексического и последовательно заносит их в стек:

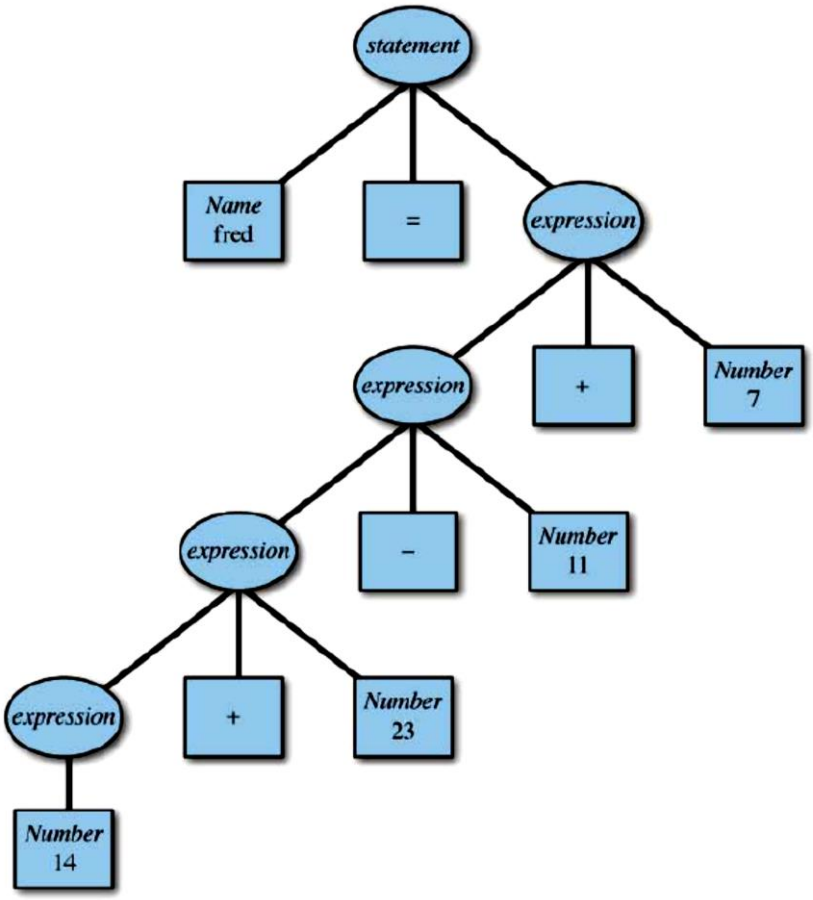

Рисунок 5. Дерево разбора для fred = 14 + 23 – 11 + 7

```
fred = 12 + 13
fred = 12 +
fred = 12
fred =
fred
```

После получения токена `13` распознаватель может применить правило `expression: NUMBER + NUMBER` для свёртки. В этом случае `12`, `+` и `13` выталкиваются из стека и заменяются нетерминалом `expression`. В стеке остается: `fred = expression`.

Теперь применяется правило `statement: NAME = expression`. `fred`, `=`, и `expression` выталкиваются из стека, а `statement` наоборот туда помещается. Достигнут конец входной последовательности, и на вершине стека находится начальный символ грамматики. Следовательно, вход успешно распознан.

Распознаватели, генерируемые Bison, могут использовать один из двух известных алгоритмов: LALR(1) (англ. Look Ahead Left to Right) и GLR (англ. Generalized Left to Right). Большинство синтаксических анализаторов используют первый алгоритм, потому что он быстрее и легче в реализации по сравнению с GLR.

Алгоритм LALR(1) не может использоваться для распознавания:

- ✓ неоднозначных грамматик;
- ✓ грамматик, требующих более одного символа предпросмотра.

Рассмотрим следующий пример. Грамматика описана правилами:

```
phrase: cart_animal AND CART
        | work_animal AND PLOW
cart_animal: HORSE | GOAT
work_animal: HORSE | OX
```

Эта грамматика однозначна, так как можно построить только одно дерево вывода для любой входной цепочки символов. Но Bison не сможет правильно разобрать эти цепочки, потому что ему потребуется два символа предпросмотра, чтобы сделать это. Например, для входа `HORSE AND CART` он не сможет определить, по какому правилу нужно сворачивать `HORSE`: `cart_animal: HORSE` или `work_animal: HORSE`, пока не встретит символ `CART`. Можно изменить первое правило этой грамматики следующим образом:

```
phrase: cart_animal CART | work_animal PLOW
```

В этом случае у Bison не возникнет проблем с разбором.

На практике языки программирования обычно описываются однозначными грамматиками, требующими не более одного символа предпросмотра.

Структура программы на языке Bison

Программа, написанная на языке Bison, состоит из трёх основных частей, как и программа на Flex:

```
Объявления
%%
Правила
%%
Вспомогательные функции
```

Первые два раздела являются обязательными, хотя и могут быть пустыми. Третий раздел и предшествующие ему символы `%%` могут отсутствовать.

Первый раздел, раздел объявлений – это управляющая информация для синтаксического анализатора, и обычно он настраивает среду исполнения для анализа. Этот раздел может включать в себя программный код на Си, который полностью копируется в начало генерируемого файла. Обычно это объявления переменных и директивы `#include`, заключенные в `%{` и `%}`. В этом разделе также могут быть объявления вида `%union`, `%start`, `%token`, `%type`, `%left`, `%right` и `%nonassoc`. Раздел объявлений может содержать комментарии в `/*` и `*/`.

Второй раздел содержит правила грамматики и действия, связанные с ними. Действие представляет собой программный код на Си, который выполняется, когда Bison применяет правило для входных данных. Например:

```
date: month'/'day'/'year { printf("date found"); };
```

В действиях можно ссылаться на нетерминальные символы из правой части правил, используя символ `$` и номер нетерминала в правиле:

```
date: month'/'day'/'year { printf("date %d-%d-%d found", $1, $3, $5); };
```

Для использования нетерминала из левой части правила используется `$$`.

Пример описания правил и действий для следующей грамматики приведён в листинге 1:

G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +, '\n'}, {*program*, *expr*, *integer*, *digit*}, **P**, *program*)
P:
program → *program expr* '\n' | ε
expr → *integer* | *expr* + *expr* | *expr* – *expr*
integer → *digit* | *digit integer*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Листинг 1. Описание правил и действий на языке Bison

```
program:
          program expr ‘\n’     { printf(“%d\n”, $2); }
          |
          ;
expr:
          INTEGER               { $$ = $1; }
          | expr ‘+’ expr       { $$ = $1 + $3; }
          | expr ‘–’ expr       { $$ = $1 – $3; }
          ;
```

С помощью левой рекурсии мы определили, что программа (`program`) состоит из 0 или более выражений (`expr`). Каждое выражение заканчивается переходом на новую строку. Как только новая строка найдена, значение выражения выводится на экран.

Правило для нетерминального символа `digit` должно быть описано в разделе правил для лексического анализатора, например, так:

```
[0–9]+     {
               yylval = atoi(yytext);
               return INTEGER;
           }
```

Таким образом, `INTEGER` – это токен, возвращаемый лексическим анализатором синтаксическому. В области объявлений программы синтаксического анализатора нужно сделать следующее определение:

```
%token INTEGER
```

Третий раздел – это программный код на Си, который полностью копируется в генерируемую программу анализатора.

Bison по умолчанию генерирует файл с именем `y.tab.c` и заголовочный файл с именем `y.tab.h`. Для нашего примера заголовочный файл будет содержать:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Программа лексического анализатора должна включать этот файл. Для получения токена синтаксический анализатор вызывает функцию `yylex()`. Значения, связанные с объявленным токеном, помещаются сканером в переменную `yylval`.

При синтаксическом разборе Bison организует хранение в памяти двух стеков: стека разбора и стека значений. Стек разбора содержит терминалы и нетерминалы, представляющие

текущее состояние разбора. Стек значений – это массив элементов типа YYSTYPE, который связывает значение с каждым элементом из стека разбора. Например, когда лексический анализатор возвращает токен INTEGER, Bison помещает этот токен в стек разбора. А соответствующее этому токену значение yylval помещается в стек значений. Таким образом, оба стека синхронизированы.

Когда применяется правило

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

происходит замещение в стеке разбора правой части правила на левую. В этом случае из стека удаляются три символа expr, '+' и expr, и сохраняется expr. Ссылка на позиции в стеке значений происходит при использовании $1 для первого символа из правой части правила, $2 – для второго и т.д. $$ обозначает верхушку стека после выполнения свёртки. Таким образом, действие в примере выполняет сложение значений, связанных с двумя символами expr, удаляет три элемента из стека значений и записывает туда полученную сумму. В результате оба стека остаются синхронизированными.

Для запуска синтаксического анализа на выполнение нужно вызвать функцию yyparse().

Полный текст лексического и синтаксического анализаторов для грамматики, приведённой выше, представлен в следующих листингах.

Листинг 2. Лексический анализатор на языке Flex – input.l

```
%option noyywrap

%{
#include <stdlib.h>
void yyerror (char *);
#include "y.tab.h"
%}

%%

[0-9]+    {
              yylval = atoi(yytext);
              return INTEGER;
          }

[+-\n]    { return *yytext; }

[  \t]    ; /* пропускаем пробелы и табуляции */

.         { yyerror("Неизвестный символ!"); }

%%
```

Листинг 3. Синтаксический анализатор на языке Bison – input.y

```
%{
    #include <stdio.h>
    int yylex(void);
```

Листинг 3. Синтаксический анализатор на языке Bison – input.y

```
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'     { printf("%d\n", $2); }
        |
        ;
expr:
        INTEGER               { $$ = $1; }
        | expr '+' expr       { $$ = $1 + $3; }
        | expr '-' expr       { $$ = $1 - $3; }
        ;

%%

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

int main(void)
{
    yyparse();
    return 0;
}
```

Для компиляции и запуска программы используются следующие команды:

```
$ bison -dy input.y
$ flex input.l
$ gcc lex.yy.c y.tab.c
$ ./a.out
1+4
5
2+3
5
```

В предыдущей лабораторной работе был создан лексический анализатор с применением Flex. Теперь необходимо реализовать синтаксический анализатор с применением Bison и связать лексический и синтаксический анализаторы в единое целое.

Входной язык, для которого создавался лексический анализатор, содержит операторы цикла **for (...; ...; ...) do ...**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).

Описанный выше входной язык был описан с помощью КС-грамматики **G** ({**for**, **do**, ':=', '<', '>', '=', '-', '+', '**(**', '**)**', ';', '.', '_', 'a', 'b', 'c', ..., 'x', 'y', 'z', 'A', 'B', 'C', ..., 'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}, {*S, A, B, C, I, L, N, Z*}, **P**, *S*) с правилами **P** (правила представлены в расширенной форме Бэкуса-Наура):

$S \rightarrow$ **for (**A; A; A;**) do** A; | **for (**A; A; A;**) do** S; | **for (**A; A; A;**) do** A; S

$A \rightarrow I := B$

$B \rightarrow$ C > C | C < C | C = C

$C \rightarrow I \mid N$

$I \rightarrow$ (_ | L) {_ | L | Z | 0}

$N \rightarrow$ [- | +] ({0 | Z} . {0 | Z} | {0 | Z}. | {0 | Z}) [(e | E) [- | +]{0 | Z}][f | l | F | L]

$L \rightarrow$ a | b | c | ... | x | y | z | A | B | C | ... | X | Y | Z

$Z \rightarrow$ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

В предыдущей лабораторной работе программа лексического анализатора выводила сообщение о типе лексемы при каждом ее распознавании. Для того чтобы связать лексический анализатор с синтаксическим, необходимо заменить вызовы `printf` на возвращение типа распознанной лексемы. В листинге 4 приведена переделанная программа лексического анализатора.

Листинг 4. Лексический анализатор на языке Flex – scanner.l

```
%option noyywrap yylineno
%{
    #include <stdio.h>
    #include "parser.tab.h"
    int ch;
    extern void yyerror (char *);
%}

digit[0-9]
letter[a-zA-Z]
delim[();,]
oper[<>=]
ws[ \t\n]

%%

for     { ch += yyleng; return FOR; }
do      { ch += yyleng; return DO;  }
("_"|{letter})("_"|{letter}|{digit})* { ch += yyleng; return ID; }
[-+]?({digit}*\.{digit}+|{digit}+\.|{digit}+)
([eE][-+]?{digit}+)?[flFL]? { ch += yyleng; return NUMBER; }
{oper}  { ch += yyleng; return CMP;     }
":="    { ch += yyleng; return ASSIGN;  }
{delim} { ch += yyleng; return *yytext; }
{ws}+   { ch += yyleng;                 }
.       { yyerror("Unknown character"); ch += yyleng; }

%%
```

Типы лексем соответствуют тем классам, которые были определены в предыдущей лабораторной работе. В случае с разделителями и знаками операций (тип лексем `DELIM`) возвращается текст, содержащий саму лексему. В этом случае в программе синтаксического анализатора на Bison в правилах будет использоваться тоже не тип этих лексем, а сами лексемы (см. листинг 5).

В переделанной программе лексического анализатора отсутствует третий блок, функция `main` больше не нужна в лексическом анализаторе, теперь она будет находиться в программе синтаксического анализатора.

В заголовочном файле `parser.tab.h` будут объявлены типы лексем, используемые лексическим анализатором. Этот файл будет сгенерирован при компиляции программы синтаксического анализатора на языке Bison.

Программа синтаксического анализатора представлена в листинге 5. В разделе объявлений представлены токены (типы лексем). Наивысший приоритет имеет токен `CMP`, так как находится ниже всех в списке объявлений. При этом он левоассоциативный (`%left`). А токен `ASSIGN` (знак присваивания) – правоассоциативный (`%right`). Ассоциативность влияет на группировку выражений, использующих операторы с одинаковым приоритетом. Выражения могут группироваться слева направо (левоассоциативные операции), справа налево (правоассоциативные операции) или вообще не группироваться. Например, выражение A – B – C может интерпретироваться как (A – B) – C при группировке слева направо или как A – (B – C) при группировке справа налево. В случае с токенами `FOR`, `DO`, `ID` и `NUMBER` ассоциативность не используется.

Листинг 5. Синтаксический анализатор на языке Bison – parser.y

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

    extern FILE *yyin;
    extern int yylineno;
    extern int ch;
    extern char *yytext;

    void yyerror(char *);
%}

%token FOR DO ID NUMBER
%right ASSIGN
%left CMP

%%

program: statement | program statement { printf("\nprogram\n"); }
statement: FOR'('expr';'expr';'expr')'DO oper';' {
                                        printf("\nstatement\n");}
statement: error';'
oper: statement | expr | statement oper { printf("\noperator\n"); }
expr: ID ASSIGN expr_cmp          { printf("\nassign\n"); }
expr_cmp: prim_expr CMP prim_expr { printf("\ncomparison\n"); }
prim_expr: ID | NUMBER            { printf("\nprimary expression\n"); }
```

Листинг 5. Синтаксический анализатор на языке Bison – parser.y

```
%%

void yyerror(char *errmsg)
{
    fprintf(stderr, "%s (%d, %d): %s\n", errmsg, yylineno, ch, yytext);
}

int main(int argc, char **argv)
{
    if(argc < 2)
    {
       printf("\nNot enough arguments. Please specify filename. \n");
       return -1;
    }
    if((yyin = fopen(argv[1], "r")) == NULL)
    {
       printf("\nCannot open file %s.\n", argv[1]);
       return -1;
    }

    ch = 1;
    yylineno = 1;

    yyparse();

    return 0;
}
```

Правила грамматики описаны в разделе правил. Правило `statement: error';'` необходимо для того, чтобы синтаксический анализатор мог выполнить восстановление после встреченной синтаксической ошибки и продолжить анализ до конца программы. Без этого правила синтаксический анализатор остановится, как только встретит первую синтаксическую ошибку. Синхронизирующим символом выступает точка с запятой (;). Это означает, что после обнаружения ошибки синтаксический анализатор будет пропускать все символы, пока не встретит ‘;’. После этого синтаксический анализ продолжится в обычном режиме.

Функция `yyerror` выводит сообщение об ошибке в стандартный поток ошибок. При этом сообщается номер строки (`yylineno`) и номер столбца (`ch`), в которых встречена ошибка, а также лексема (`yytext`), следующая за местом обнаружения ошибки.

В функции `main` открывается файл, содержащий исходный код программы для анализа, и запускается работа синтаксического анализа (`yyparse`). Вызов функции `yylex` будет происходить из функции `yyparse`.

Задание на лабораторную работу

Доработать программу лексического анализатора из лабораторной работы № 3 так, чтобы генерируемый ею поток токенов поступал на вход синтаксического анализатора. Выполнить программную реализацию синтаксического анализатора, используя генератор синтаксических анализаторов Bison. Результаты работы программы представить в виде дерева синтаксического разбора.

Контрольные вопросы

1. Какую роль выполняет синтаксический анализ в процессе компиляции?
2. Какие проблемы возникают при построении синтаксического анализатора и как они могут быть решены?
3. Какие типы грамматик существуют? Что такое КС-грамматики? Расскажите об их использовании в компиляторе.
4. Какие типы распознавателей для КС-грамматик существуют? Расскажите о недостатках и преимуществах различных типов распознавателей.
5. Поясните правила построения дерева вывода грамматики.
6. Что такое сдвиг, свёртка? Для чего необходим алгоритм «сдвиг-свёртка»?
7. Расскажите, как работает алгоритм «сдвиг-свёртка» в общем случае (с возвратами).
8. Как работает алгоритм «сдвиг-свёртка» без возвратов (объясните на своем примере)?

Список литературы

1. Bison – GNU parser generator. Режим доступа: http://www.gnu.org/software/bison/.
2. Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. – 274 p.
3. Niemann T. A Compact Guide To Lex & Yacc. Режим доступа: http://epaperpress.com/lexandyacc/download/lexyacc.pdf.