Министерство науки и высшего образования РФ

ФГАОУ ВО Пермский национальный исследовательский

политехнический университет

Кафедра «Вычислительная математика, механика и биомеханика»

Отчет по лабораторной работе № 3

тема «Разработка синтаксически управляемого транслятора с использованием flex и bison «Языки программирования и методы трансляции»

Выполнил: студент группы ИСТ-22-1Б Синьковский Г.С.

Проверил: Батин С.Е.

Пермь, 2022

Содержимое

[Упражнение 1 3](#_Toc154787823)

[1.1. Постановка задачи 3](#_Toc154787824)

[1.2. Код программы 3](#_Toc154787825)

[1.3 Пример работы программы 4](#_Toc154787826)

[Упражнение 2 5](#_Toc154787827)

[2.1. Постановка задачи 5](#_Toc154787828)

[2.2. Код программы 5](#_Toc154787829)

[2.3. Примеры работы программы 6](#_Toc154787830)

[Упражнение 3 7](#_Toc154787831)

[3.1. Постановка задачи 7](#_Toc154787832)

[3.2 Код программы 7](#_Toc154787833)

[3.3. Пример работы программы 8](#_Toc154787834)

[Упражнение 4 9](#_Toc154787835)

[4.1. Постановка задачи 9](#_Toc154787836)

[4.2 Код программы 9](#_Toc154787837)

# Упражнение 1

## 1.1. Постановка задачи

Написать программу для генератора лексических анализаторов, выводящую в консоль переданный токен.

## 1.2. Код программы

%{

#include "t1.tab.h"

%}

**%%**

[0-9]+ { return NUMBER; }

[-+\*/()\n] { return yytext[0]; }

.

**%%**

%{

#include <stdio.h>

extern int yylex();

extern int yyparse();

extern FILE \*yyin;

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

%}

%token NUMBER

**%%**

expression: NUMBER { printf("NUMBER "); }

| expression NUMBER { printf("NUMBER "); }

| expression '+' { printf("PLUS "); }

| expression '-' { printf("MINUS "); }

| expression '\*' { printf("MULTIPLY "); }

| expression '/' { printf("DIVIDE "); }

| expression '(' { printf("LPAREN "); }

| expression ')' { printf("RPAREN "); }

| expression '\n' { printf("\n"); return 0;}

;

**%%**

int yylex();

int main() {

yyparse();

return 0;

}

## 1.3 Пример работы программы

На рис. 1 продемонстрирован результат работы программы 1 упражнения.



Рисунок 1. Результат программы 1 упражнения

# Упражнение 2

## 2.1. Постановка задачи

Описать грамматику, позволяющую вычислять значения арифметических выражений, состоящих из произвольного конечного количества целых чисел и знаков «+» и «-»

## 2.2. Код программы

%{

#include "t2.tab.h"

%}

**%%**

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

[-+\n] { return yytext[0]; }

.

**%%**

%{

#include <stdio.h>

extern int yylex();

extern int yyparse();

extern FILE \*yyin;

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

%}

%union {

int number;

}

%token NUMBER

%type <number> expression

%type <number> term

**%%**

expression: expression '\n' { printf("Result: %d\n", $1); return 0;}

| expression '+' term { $$ = $1 + $3; }

| expression '-' term { $$ = $1 - $3; }

| term { $$ = $1; }

;

term: NUMBER { $$ = $1; }

;

**%%**

int yylex();

int main() {

yyparse();

return 0;

}

## 2.3. Примеры работы программы

На рис.2 показаны результаты работы программы упражнения 2.



Рисунок 2. Результат программы 2 упражнения

# Упражнение 3

## 3.1. Постановка задачи

Описать грамматику, позволяющую проводить трансляцию арифметических выражений, состоящих из вещественных чисел, знаков «+», «-», «\*», «/» и скобок из инфиксной записи в постфиксную.

## 3.2 Код программы

%{

#define YYSTYPE double

#include "t3.tab.h"

%}

**%%**

[0-9]+([.][0-9]+)? { yylval = atof(yytext); return NUMBER; }

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

.

**%%**

%{

#include <stdio.h>

extern int yylex();

extern int yyparse();

extern FILE \*yyin;

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

%}

%union {

double number;

}

%token <number> NUMBER

%type <number> factor

%type <number> term

%type <number> expr

**%%**

expr: expr '\n' { printf("\nResult: %.2f\n", $1); return 0;}

| term { $$ = $1; }

| expr '+' term { printf("+ "); $$ = $1 + $3; }

| expr '-' term { printf("- "); $$ = $1 - $3; }

;

term: factor { $$ = $1; }

| term '\*' factor { printf("\* "); $$ = $1 \* $3; }

| term '/' factor { printf("/ "); $$ = $1 / $3; }

;

factor: NUMBER { printf("%.2f ", $1); $$ = $1; }

| '(' expr ')' { $$ = $2; }

;

**%%**

int yylex();

int main() {

yyparse();

return 0;

}

## 3.3. Пример работы программы

На рис.3 представлен результат работы программы упражнения 3.

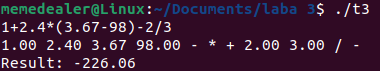


Рисунок 3. Пример работы программы упражнения 3

# Упражнение 4

## 4.1. Постановка задачи

Организовать вычисление выражений с использованием абстрактного синтаксического дерева. Для этого описать необходимые структуры и функции, осуществляющие построение абстрактного синтаксического дерева при разборе, а также его рекурсивных обход и вычисление арифметического выражения. В качестве основы грамматики использовать синтаксис арифметических выражений из лекции 3 (Слайд 9). При этом типы узлов дерева разбора примут вид:

## 4.2 Код программы

%{

#include "t4.tab.h"

%}

**%%**

"+" { return ADD; }

"-" { return SUB; }

"\*" { return MUL; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[0-9]+ { yylval.d = atof(yytext); return NUM; }

\n { return EOL; }

[ \t] ;

. { return yytext[0]; }

**%%**

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*msg) {

fprintf(stderr, "%s\n", msg);

}

struct tree {

int type;

struct tree \*left;

struct tree \*right;

};

struct value {

int type;

double val;

};

struct tree \*newTree(int type, struct tree \*left, struct tree \*right);

struct tree \*newValue(double d);

double evaluate(struct tree \*t);

void freeTree(struct tree \*t);

extern int yylex();

extern int yyparse();

extern struct tree \*root;

%}

%union {

struct tree \*t;

double d;

}

%token <d> NUM

%token ADD SUB MUL DIV LPAREN RPAREN

%token EOL

%type <t> expr term factor

%left ADD SUB

%left MUL DIV

%start program

**%%**

program:

| program expr EOL { printf("Result: %f\n", evaluate($2)); freeTree($2); return 0; }

;

expr: term

| expr ADD term { $$ = newTree('+', $1, $3); }

| expr SUB term { $$ = newTree('-', $1, $3); }

;

term: factor

| term MUL factor { $$ = newTree('\*', $1, $3); }

| term DIV factor { $$ = newTree('/', $1, $3); }

;

factor: NUM { $$ = newValue($1); }

| LPAREN expr RPAREN { $$ = $2; }

;

**%%**

struct tree \*newTree(int type, struct tree \*left, struct tree \*right) {

struct tree \*t = malloc(sizeof(struct tree));

if (!t) {

yyerror("Out of memory");

exit(1);

}

t->type = type;

t->left = left;

t->right = right;

return t;

}

struct tree \*newValue(double d) {

struct value \*v = malloc(sizeof(struct value));

if (!v) {

yyerror("Out of memory");

exit(1);

}

v->type = 'K';

v->val = d;

return (struct tree \*)v;

}

double evaluate(struct tree \*t) {

if (!t) {

yyerror("Internal yyerror: null pointer in eval");

return 0.0;

}

double v;

switch (t->type) {

case 'K': v = ((struct value \*)t)->val; break;

case '+': v = evaluate(t->left) + evaluate(t->right); break;

case '-': v = evaluate(t->left) - evaluate(t->right); break;

case '\*': v = evaluate(t->left) \* evaluate(t->right); break;

case '/': v = evaluate(t->left) / evaluate(t->right); break;

default: yyerror("Unknown operator");

return 0.0;

}

return v;

}

void freeTree(struct tree \*t) {

if (!t) return;

switch (t->type) {

case 'K':

free(t);

break;

case '+':

case '-':

case '\*':

case '/':

freeTree(t->left);

freeTree(t->right);

free(t);

break;

default: yyerror("Unknown operator");

}

}

int main() {

printf("Input expression: ");

yyparse();

}