Министерство образования и науки РФ

Новосибирский государственный технический университет

Кафедра прикладной математики

Лабораторная работа №3

и расчётно–графическая работа

по языкам программирования и методам трансляции

Факультет: ПМИ

Группа: ПМ-01

Студент: Конев А.М., Ряховский М.И.

Вариант: 6, 6

Преподаватель: Еланцева И.Л., Полетаева И.А.

Новосибирск

2013

# Цель лабораторной работы

Изучить табличные методы синтаксического анализа. Получить представление о методах диагностики и исправления синтаксических ошибок. Научится проектировать синтаксический анализатор на основе табличных методов.

# Тема расчётно-графической работы

Методы диагностики и исправления ошибок. Синтаксические ошибки.

# Входные и выходные данные

Входные данные представляют собой файл токенов. Выходные данные – файл с лексическими ошибками и дерево операторов.

# Грамматика языка

Начальный символ





Типы

  

 

Программа

- объявление переменной

 - присваивание

 - ну и вообще программы может не быть

Объявление переменной

 - тип потом что-то, заканчивается ;

 - имя переменной, потом что-то

 - либо пустая строка

 - либо операция присваивания

 - либо пустая строка

 - либо запятая и ещё одна переменная

Присваивание

 - имя переменной, знак присваивания, затем операция и ;

   



 - это может быть пустая строка

 - знак операции и затем продолжение

- продолжение может быть просто

 - а может быть в скобочках

Знаки операций

      

# Схема разбора

# Таблица разбора

1. void int float char | 1 0 0 0 1
2. void int float char | 4 0 0 0 1
3. eps | 3 0 0 1 1
4. eps | -1 0 0 0 1
5. void int float char | 11 0 1 0 1
6. main | 6 1 0 0 1
7. ( | 7 1 0 0 1
8. ) | 8 1 0 0 1
9. { | 9 1 0 0 1
10. int float char ID } | 21 0 1 0 1
11. } | -1 1 0 0 1
12. void | 13 0 0 0 0
13. int float char | 14 0 0 0 1
14. void | -1 1 0 1 1
15. int float char | 15 0 0 0 1
16. int | 18 0 0 0 0
17. float | 19 0 0 0 0
18. char | 20 0 0 0 1
19. int | -1 1 0 1 1
20. float | -1 1 0 1 1
21. char | -1 1 0 1 1
22. int float char | 24 0 0 0 0
23. ID | 26 0 0 0 0
24. } | 28 0 0 0 1
25. int float char | 42 0 1 0 0
26. int float char ID } | 21 0 0 0 0
27. ID | 29 0 1 0 1
28. int float char ID } | 21 0 0 0 1
29. } | -1 0 0 1 1
30. ID | 30 0 0 0 1
31. ID | 31 1 0 0 1
32. = += -= \*= | 34 0 1 0 1
33. ID | 60 0 1 0 1
34. ; | -1 1 0 1 1
35. = | 38 0 0 0 0
36. += | 39 0 0 0 0
37. -= | 40 0 0 0 0
38. \*= | 41 0 0 0 1
39. = | -1 1 0 1 1
40. += | -1 1 0 1 1
41. -= | -1 1 0 1 1
42. \*= | -1 1 0 1 1
43. int float char | 43 0 0 0 1
44. int float char | 15 0 1 0 1
45. ID | 46 0 1 0 1
46. ; | -1 1 0 1 1
47. ID | 47 0 0 0 1
48. ID | 48 1 0 0 1
49. = , ; | 55 0 1 0 1
50. , ; | 50 0 0 0 1
51. , | 52 0 0 0 0
52. ; | 54 0 0 1 1
53. , | 53 1 0 0 1
54. ID | 47 0 0 0 1
55. ; | -1 0 0 1 1
56. = | 57 0 0 0 0
57. , ; | 59 0 0 0 1
58. = | 58 1 0 0 1
59. ID | 60 0 0 0 1
60. , ; | -1 0 0 1 1
61. ID | 61 0 0 0 1
62. ID | 62 1 0 0 1
63. + - \* == != > < ; , ) | 63 0 0 0 1
64. + - \* == != > < | 65 0 0 0 0
65. ; , ) | 67 0 0 0 1
66. + - \* == != > < | 74 0 1 0 1
67. ID ( | 68 0 0 0 1
68. , ; ) | -1 0 0 1 1
69. ID | 70 0 0 0 0
70. ( | 71 0 0 0 1
71. ID | 60 0 0 0 1
72. ( | 72 1 0 0 1
73. ID | 60 0 1 0 1
74. ) | -1 1 0 1 1
75. + | 81 0 0 0 0
76. - | 82 0 0 0 0
77. \* | 83 0 0 0 0
78. == | 84 0 0 0 0
79. != | 85 0 0 0 0
80. < | 86 0 0 0 0
81. > | 87 0 0 0 1
82. + | -1 1 0 1 1
83. - | -1 1 0 1 1
84. \* | -1 1 0 1 1
85. == | -1 1 0 1 1
86. != | -1 1 0 1 1
87. < | -1 1 0 1 1
88. > | -1 1 0 1 1

# Тесты

|  |  |  |  |
| --- | --- | --- | --- |
| Исходный код | Таблица идентификаторов | Дерево | Файл ошибок |
| void main (){  int a, aa = 1;  char b = a + aa;  float c;  c = a + b;  int a1a;  } | identifier.our\_table[10]  [0] {name="a" type=1}  [1] {name="aa" type=1}  [2] {name="a1a" type=1}  identifier.our\_table[11]  [0] {name="b" type=2}  identifier.our\_table[12]  [0] {name="c" type=3} |  |  |
| void main (){  a = a + b  } | - | - | Ошибка в обработке }  Возможно на этом месте должно быть: + - \* == != > < ; , )  Ошибка в обработке идентификатора a не объявлен тип  Ошибка в обработке идентификатора b не объявлен тип |
| void main (){  a = a + ( d \* (b - c)) ;  } | - |  | Ошибка в обработке идентификатора a не объявлен тип  Ошибка в обработке идентификатора b не объявлен тип  Ошибка в обработке идентификатора c не объявлен тип  Ошибка в обработке идентификатора d не объявлен тип |
| void main (){  a = a + ( d \* b - c) ;  } | - |  | Ошибка в обработке идентификатора a не объявлен тип  Ошибка в обработке идентификатора b не объявлен тип  Ошибка в обработке идентификатора c не объявлен тип  Ошибка в обработке идентификатора d не объявлен тип |
| void main (){  int a = a – (b+c);} | - |  | Ошибка в обработке идентификатора b не объявлен тип  Ошибка в обработке идентификатора c не объявлен тип |
| void main (){  int a = a - (b+c;  } | - | - | Ошибка в обработке ;  Возможно на этом месте должно быть: )  Ошибка в обработке идентификатора b не объявлен тип  Ошибка в обработке идентификатора c не объявлен тип |
| void main (){  ind a;  } | - | - | Ошибка в обработке a  Возможно на этом месте должно быть: = += -= \*=  Ошибка в обработке идентификатора a не объявлен тип  Ошибка в обработке идентификатора ind не объявлен тип |
| void main (){  1 = 1;  } | - | - | Ошибка в обработке 1 константе не может быть присовенно значение |

# Код программы

bool translator::parse(string tokens\_file, string errors\_file){

ifstream parse\_token\_f(tokens\_file.c\_str()); //поток для ввода токенов

ofstream parse\_error\_f(errors\_file.c\_str()); //поток для вывода ошибок

token cur\_t, next\_t; //текущий токен и следующий за ним

bool local\_error = false; //наличие ошибки

parse\_token\_f >> cur\_t;

parse\_token\_f >> next\_t;

int cur\_row = 0; //текущая строка в таблице разбора

int prev\_row; //предыдущая строка в таблице разбора

bool have\_type = false; //определяем ли мы сейчас тип

int type\_type; //опеределяемый тип

//переменные для построения дерева разбора

bool little\_tree\_bg = false; //начато ли построение малого дерева

string token\_str\_prev; //терминал предыдущего токена

int id\_numb; //номер встречаемого индетификатора

vector<token> little\_tree\_code; // токены для создания маленького дерева

while(!parse\_token\_f.eof() && !local\_error){

string token\_str = get\_token\_text(cur\_t); // какой текст содержится в токене

if(cur\_t.table\_n == 5 || cur\_t.table\_n == 6)

token\_str = "ID";

if(token\_str == "ID")

little\_tree\_bg = true;

bool find\_terminal = false; //допустим ли данный терминал

for(int i = 0; i < parsing\_table[cur\_row].termenal.size() &!find\_terminal; i++){

if(parsing\_table[cur\_row].termenal[i] == token\_str)

find\_terminal = true;

}

if(find\_terminal){ //если получаем то, что ожидали то обрабатываем это

bool change\_row = false; //сменили ли мы строку

if(parsing\_table[cur\_row].put\_in\_stack)

parsing\_stack.push(cur\_row+1); //если надо получить в стек - ложим

if(parsing\_table[cur\_row].accept){ //принимаем терминал и если надо - расширяем дерево

if(little\_tree\_bg){

little\_tree\_code.push\_back(cur\_t);

}

if(token\_str == ";" || token\_str == ","){ //если закончили разбор цельного оператора

grow\_tree(little\_tree\_code); //добавили всё что нужно в дерево

//и перешли в исходное состояние

little\_tree\_code.clear();

little\_tree\_bg = false;

}

//все, обнуляем типа больше нет

if(token\_str == ";"){

have\_type = false;

}

//Если мы нашли тип, то мы его запоминаем

if(token\_str == "int" || token\_str == "char" || token\_str == "float"){

have\_type = true;

if(token\_str == "int")

type\_type = 1;

if(token\_str == "char")

type\_type = 2;

if(token\_str == "float")

type\_type = 3;

}

//Заносим тип в таблицу идентицикаторов

if(token\_str == "ID" && have\_type && cur\_row == 47){

identifier.set\_ind\_type(get\_token\_text(cur\_t),type\_type);

}

//Если вдруг попытались присвоить что-то константе

if(cur\_row == 30 && cur\_t.table\_n != 5){

parse\_error\_f << "Ошибка в обработке " << get\_token\_text(cur\_t) << " константе не может быть присовенно значение" <<endl;

cout << "Lex error" << endl;

local\_error = true;

}

//и пошли дальше

cur\_t = next\_t;

if(!parse\_token\_f.eof())

parse\_token\_f >> next\_t; //если принимает, то считываем новый

}

if(parsing\_table[cur\_row].should\_return){

prev\_row = cur\_row; //запоминаем предыдущий

cur\_row = parsing\_stack.top(); //если надо взять из стека - берём

parsing\_stack.pop();

change\_row = true;

}

if(!change\_row && parsing\_table[cur\_row].jump != - 1){

cur\_row = parsing\_table[cur\_row].jump; //если надо прыгнуть - прыгаем

}

}

else{ //если произошщло несоответсвие

if(parsing\_table[cur\_row].error){ //если можем судить что уже ошибка, то возвращаем её

local\_error = true;

parse\_error\_f << "Ошибка в обработке " << get\_token\_text(cur\_t) << endl;

cout << "Lex error" << endl;

//Для РГЗ 1, начало - вывод альтернатив

parse\_error\_f << "Возможно на этом месте должно быть: ";

do{

for(int i = 0 ; i < parsing\_table[cur\_row].termenal.size(); i++){

parse\_error\_f << parsing\_table[cur\_row].termenal[i] << " ";

}

cur\_row--;

}while(!parsing\_table[cur\_row].error);

parse\_error\_f << endl;

//Для РГЗ 1 - конец

}

else{ //Если нет - переходим на следующий

cur\_row++;

}

}

token\_str\_prev = token\_str;

};

//Для РГЗ 2, начало - определения проверка определения типов

for(int i = 10; i < 32; i++){

bool is\_els = true; //еслить ли элементы

int j = 0; //номер элемента в цепочке

lexeme check\_lex; //получаемая лексема

while(is\_els){

is\_els = identifier.get\_lexeme(i, j, check\_lex);

if(is\_els){

if(check\_lex.type == 0){

local\_error = true;

parse\_error\_f << "Ошибка в обработке идентификатора " << check\_lex.name << " не объявлен тип" << endl;

cout << "Type error" << endl;

}

j++;

}

};

}

//Для РГЗ 2 - конец

parse\_token\_f.close();

parse\_error\_f.close();

return !local\_error;

}

void translator::grow\_tree(vector<token> code){

tree\_el\* little\_tree\_beg = new tree\_el; //корень маленького дерева

if(code.size() > 2){

grow\_little\_tree(code, little\_tree\_beg); //веращиваем маленькое деревце

//Если это самое начало дерева

if(tree\_begin == 0){

tree\_begin = little\_tree\_beg;

tree\_cur = tree\_begin;

}

else{

tree\_cur->right = little\_tree\_beg; //и присоединяем его к большому

}

while(tree\_cur->right != 0){ //перемещаем курсор дерева

tree\_cur = tree\_cur->right;

};

}

}

void translator::grow\_little\_tree(vector<token> code, tree\_el \*&beg){

vector<token> L, R; //левое и правое поддеревья

int bracket\_num = 0; //количество скобок

int L\_num = 0;

bool flag = false;

//Если последний токен

if(code.size() == 1 && (get\_token\_text(code[0]) == ";" || get\_token\_text(code[0]) == ",")){

beg->id = ";";

beg->type = 2;

}

else{

vector<int> opers\_n; //номера "верхних операций"

//Находим все "верхние операции"

for(int i = 0; i < code.size(); i++){

if(get\_token\_text(code[i]) == "(") bracket\_num++;

if(get\_token\_text(code[i]) == ")") bracket\_num--;

if(code[i].table\_n == 2 && bracket\_num == 0) opers\_n.push\_back(i);

}

if(opers\_n.size() == 0){ //ели операций нет пропуска всё связанное с ними

flag = false;

}

else{ //если есть - ищим операцию с наименьшим приоритетом

int low\_num = -1; //номер нужной нам операции

//Сначало проверка на наличие равенства

for(int j = 0; j < opers\_n.size() && low\_num == -1; j++)

if(get\_token\_text(code[opers\_n[j]]) == "=" || get\_token\_text(code[opers\_n[j]]) == "+=" || get\_token\_text(code[opers\_n[j]]) == "-=" || get\_token\_text(code[opers\_n[j]]) == "\*=")

low\_num = opers\_n[j];

//Потом на + и -

for(int j = opers\_n.size()-1; j >=0 && low\_num == -1; j--){

if(get\_token\_text(code[opers\_n[j]]) == "+" || get\_token\_text(code[opers\_n[j]]) == "-"){

low\_num = opers\_n[j]; //нашли нашу операцию

}

}

//Если не нашли выбираем последнюю операцию

if(low\_num == -1)

low\_num = opers\_n[opers\_n.size() - 1];

//Привет интегрулятору

if(get\_token\_text(code[0]) == "(") L\_num++;

//Формируем левое поддерево

for(int j = 0; L\_num < low\_num; L\_num++, j++)

L.push\_back(code[L\_num]);

//Формируем правое поддерево

int R\_num = 0;

int l;

if(get\_token\_text(code[low\_num +1])=="(") R\_num++;

for(l=0, R\_num = low\_num +1+R\_num; R\_num<code.size(); R\_num++, l++)

R.push\_back(code[R\_num]);

beg->left = new tree\_el;

beg->right = new tree\_el;

beg->id = get\_token\_text(code[low\_num]);

beg->type = 0;

grow\_little\_tree(L, beg->left);

grow\_little\_tree(R, beg->right);

flag = true;

}

if(!flag){

int k = 0;

while(get\_token\_text(code[k]) == "(") k++;

beg->id = get\_token\_text(code[k]);

beg->type = 1;

if(code.size() > k+1){

beg->right = new tree\_el;

beg->right->id = ";";

beg->right->type = 2;

}

}

}

}

void translator::out\_tree(string f\_name){

ofstream out\_f(f\_name.c\_str());

out\_tree\_rec(out\_f, tree\_begin);

out\_f.close();

}

void translator::out\_tree\_rec(ofstream& out\_f, tree\_el \*beg){

if(beg != 0){

if(beg->left != 0) out\_tree\_rec(out\_f, beg->left);

if(beg->right != 0) out\_tree\_rec(out\_f, beg->right);

out\_f << beg->id << " ";

}

}