Белорусский государственный университет информатики и радиоэлектроники

Кафедра информатики

Лабораторная работа № 3

**Семантический анализатор**

Выполнил:

Губский М.Д.

Проверил:

Шиманский В.В.

Минск, 2016

1. ***Постановка задачи:***

В данной работе ставится задача по исследованию области семантических анализаторов, рассмотрению аналогов и написанию своего собственного анализатора семантических ошибок, заложение в программу двух типов ошибок. Правильная их обработка означает готовность работы.

1. ***Программа и комментарии:***

В результате выполнения работы использовали уже написанный лексический и синтаксический анализатор. Из выражений, уже выделенных, составлялись синтаксический конструкии по некоторым правилам. Эти правила можно увидеть в docstring каждой из функций. Код приведен приложении. После заводилась таблица для всех переменных и анализировался код со стороны правильного присваивания и(или) правильности арифметических операций (деление на ноль).

1. ***Принцип работы***

Для обнаружения ошибок, связанных присваиванием, используется лист, в который заносится имя переменной. При определении текущей конструкции как присваивание производится добавление имени переменной в лист. При вычислении арифметического выражения (присваивание, арифметические операции, операции сравнения) при наличии идентификатора в выражении производится поиск в листе переменных. Если данная переменная не объявлена или не определена, то выбрасывается исключение.

Обнаружение ошибок типа «деление на ноль» производится при разборе лексическим анализатором продукции выполнения арифметической операции, связанной с делением. Если второй параметр операции деления «0», генерируется ошибка.

Семантическая ошибка №1:

{  
 var i=10/0;  
 var f=function(n, y){  
 if (n<=1){  
 return 1;  
 }  
 else{  
 return 'f';  
 }  
 }  
 while (i>0 || i=='d') {  
 i--;  
 }  
}

В данной программе содержится ошибка «деление на 0». Данная ситуация правильно обрабатывается семантическим анализатором, выводящим соответствующее сообщение:



Семантическая ошибка №2:

Текст программы с необъявленной переменной p.

{  
 var i=10;  
 var f=function(n, y){  
 if (n<=1){  
 return p;  
 }  
 else{  
 return 'f';  
 }  
 }  
 while (i>0 || i=='d') {  
 i--;  
 }  
}

Выдается сообщение об ошибке:

****

Семантическая ошибка №3:

Текст программы с неправильным количеством параметров.

{  
 var i=10;  
 var f=function(n, i){  
 if (n<=1){  
 return 1;  
 }  
 else{  
 return 'f';  
 }  
 }  
 f(4);  
 while (i>0 || i=='d') {  
 i--;  
 }  
}

Выдается сообщение об ошибке:

****

***Вывод***

В результате работы были получены знания и практические навыки, необходимые для обнаружении семантических ошибок в коде. Было замечено, что для применения семантического анализа нужно непрерывно следить за ходом выполнения программы. В итоге работы был простроен семантический анализатор на основе уже имеющегося лексического и синтаксического, который способен выделять возможные ошибки, уведомлять о них, генерируя исключения.

**Приложение**

Для построения семантического анализатора был использован следующий код:

\_\_author\_\_ = **'Mikhail'  
  
import** token\_rules  
**from** token\_rules **import** tokens  
**import** colorama  
**import** re  
**import** ply.lex **as** lex  
**import** ply.yacc **as** yacc  
  
ident = []  
functions = {}  
  
  
**def** p\_block(p):  
 *'''  
 block :  
 | LBRACKET body RBRACKET  
 '''* **if** len(p) == 1:  
 p[0] = []  
 **else**:  
 p[0] = p[2]  
  
  
**def** p\_body(p):  
 *'''  
 body : expression  
 | function\_declaration  
 '''* p[0] = p[1]  
  
  
**def** p\_expression(p):  
 *'''  
 expression : declaration\_with\_assignment expression  
 | declaration\_with\_assignment  
 | if expression  
 | else  
 | function\_body  
 | function\_implementation SEMICOLON  
 | function\_implementation SEMICOLON expression  
 | arithmetic\_expression  
 | while expression  
 | while  
 | console\_function  
 | console\_function expression  
  
 '''* **if** p[1]:  
 p[0] = p[1]  
  
  
**def** p\_declaration\_with\_assignment(p):  
 *'''  
 declaration\_with\_assignment : VAR ID ASSIGNMENT\_OPERATOR arithmetic\_expression SEMICOLON  
 | VAR ID ASSIGNMENT\_OPERATOR function\_declaration  
 '''* **if** len(p) == 6:  
 **if** p[2] **not in** ident:  
 ident.append(p[2])  
 **else**:  
 functions[p[2]] = p[4]  
 p[0] = p[1]  
  
  
**def** p\_arithmetic\_expression(p):  
 *'''  
 arithmetic\_expression : ID MATH\_OPERATOR arithmetic\_expression  
 | ID MATH\_OPERATOR SEMICOLON  
 | NUMBER MATH\_OPERATOR arithmetic\_expression  
 | arithmetic\_expression MATH\_OPERATOR arithmetic\_expression  
 | NUMBER  
 | ID  
 | STRING  
 '''* **if** len(p) == 2:  
 p[0] = p[1]  
 **else**:  
 v1, v2 = p[1], p[3]  
 **if** p[2] == **'/' and** v2 == 0:  
 **raise** Exception(**'\n!Semantic Error!\nDivision by zero at line {0}!'**.format(lexer.lineno))  
  
 **if** type(p[1]) **not in** (float, int) **and** p[1][0] **not in** (**'\''**, **'"'**):  
 **if** p[1] **not in** ident:  
 **raise** Exception(**'\n!Semantic Error!\n{0} is not defined at line {1}!'**.format(p[1], lexer.lineno))  
  
 **if** p[3] != **';'**:  
 **if** type(p[3]) **not in** (float, int) **and** p[3][0] **not in** (**'\''**, **'"'**):  
 **if** p[3] **not in** ident:  
 **raise** Exception(**'\n!Semantic Error!\n{0} is not defined at line {1}!'**.format(p[3], lexer.lineno))  
  
 p[0] = p[1]  
  
  
**def** p\_function\_declaration(p):  
 *'''  
 function\_declaration : FUNCTION LPAREN parameters RPAREN LBRACKET function\_body RBRACKET  
 '''* p[0] = p[3]  
  
  
**def** p\_function\_body(p):  
 *'''  
 function\_body : expression  
 | RETURN expression SEMICOLON  
 '''* **if** len(p) == 2:  
 p[0] = p[1]  
 **else**:  
 **if** type(p[2]) **not in** (float, int) **and** p[2][0] **not in** (**'\''**, **'"'**):  
 **if** p[2] **not in** ident:  
 **raise** Exception(**'\n!Semantic Error!\n{0} is not defined at line {1}!'**.format(p[2], lexer.lineno - 1))  
  
 p[0] = p[2]  
  
  
**def** p\_function\_implementation(p):  
 *'''  
 function\_implementation : ID LPAREN function\_parameters RPAREN  
 | function\_implementation MATH\_OPERATOR expression  
  
 '''* **if** len(p) == 5:  
 **if** p[1] **not in** functions.keys():  
 **raise** Exception(**'\n!Semantic Error!\n{0} is not defined at line {1}!'**.format(p[1], lexer.lineno - 1))  
 **if** len(p[3]) != len(functions[p[1]]):  
 **raise** Exception(**'\n!Semantic Error!\nNot enough parameters at line {1}!'**.format(p[3], lexer.lineno - 1))  
 p[0] = p[1]  
  
**def** p\_function\_parameters(p):  
 *"""  
 function\_parameters :  
 | function\_parameters COMMA function\_parameters  
 | arithmetic\_expression  
 """* **if** len(p) == 1:  
 p[0] = []  
 **elif** len(p) == 2:  
 p[0] = [p[1]]  
 **else**:  
 p[0] = [p[1], p[3]]  
  
  
  
**def** p\_parameters(p):  
 *"""  
 parameters :  
 | parameters COMMA parameters  
 | arithmetic\_expression  
 """* **if** len(p) == 1:  
 p[0] = []  
 **elif** len(p) == 2:  
 **if** p[1] **not in** ident:  
 ident.append(p[1])  
 p[0] = [p[1]]  
 **else**:  
 **if** p[1] **not in** ident:  
 ident.append(p[1])  
 **if** p[3] **not in** ident:  
 ident.append(p[3])  
 p[0] = [p[1], p[3]]  
  
  
**def** p\_console\_function(p):  
 *'''  
 console\_function : METHOD\_OF\_CLASS\_CONSOLE LPAREN expression RPAREN SEMICOLON  
 '''* p[0] = [p[1], p[3]]  
  
  
**def** p\_predicate(p):  
 *'''  
 predicate : ID LOGICAL\_OPERATOR arithmetic\_expression  
 | ID LOGICAL\_OPERATOR ID  
 | predicate LOGICAL\_OPERATOR predicate  
 '''* **if** p[1] **not in** ident:  
 **raise** Exception(**'{0} is not defined at line {1}!'**.format(p[1], lexer.lineno))  
  
 **if** type(p[3]) **not in** (float, int) **and** p[3][0] **not in** (**'\''**, **'"'**):  
 **if** p[3] **not in** ident:  
 **raise** Exception(**'\n!Semantic Error!\n{0} is not defined at line {1}!'**.format(p[3], lexer.lineno))  
 p[0] = p[1]  
  
  
**def** p\_while(p):  
 *'''  
 while : WHILE LPAREN predicate RPAREN block  
 '''* p[0] = [p[3], p[5]]  
  
  
**def** p\_if(p):  
 *'''  
 if : IF LPAREN predicate RPAREN block  
 '''* p[0] = [p[3], p[5]]  
  
  
**def** p\_else(p):  
 *'''  
 else : ELSE block  
 '''* p[0] = [p[2]]  
  
  
**def** p\_error(p):  
 print(colorama.Fore.RED + **'\n!Syntax ERROR!\nUnexpected token {0} at line {1}\n'**.format(str(p.value),  
 p.lexer.lineno))  
  
  
parser = yacc.yacc(method=**'LALR'**, check\_recursion=**False**)  
  
**with** open(**'code.txt'**, **'r'**) **as** file:  
 data = file.read()  
  
program = data  
lexer = lex.lex(reflags=re.UNICODE | re.DOTALL, module=token\_rules)  
result = parser.parse(input=program, lexer=lexer, debug=**False**, tracking=**True**)  
**if** result:  
 print(ident)  
 print(functions)

В основе данной программы лежали уже обозначенные в прошлых лабораторных лексемы:

**import** colorama  
**import** re  
  
reserved = {  
 **'if'**: **"IF"**,  
 **'else'**: **"ELSE"**,  
 **'while'**: **"WHILE"**,  
 **'var'**: **"VAR"**,  
 **'function'**: **"FUNCTION"**,  
 **'return'**: **"RETURN"**}  
  
tokens = (  
 **'LPAREN'**, **'RPAREN'**, **'LBRACKET'**, **'RBRACKET'**,  
 **'LOGICAL\_OPERATOR'**, **'MATH\_OPERATOR'**, **'COMMA'**,  
 **'METHOD\_OF\_CLASS\_CONSOLE'**, **'ASSIGNMENT\_OPERATOR'**,  
 **'STRING'**, **'ID'**, **'NUMBER'**, **'SEMICOLON'**,  
 ) + tuple(reserved.values())  
  
t\_ignore = **' \t'**t\_ignore\_COMMENT = **r'(\/\/[^\n]\*)|(\/\\*.\*(\\*\/))'**t\_SEMICOLON = **r'\;'**t\_LPAREN = **r'\('**t\_RPAREN = **r'\)'**t\_LBRACKET = **r'\{'**t\_RBRACKET = **r'\}'**t\_MATH\_OPERATOR = **'\+\+?|\-\-?|\\*|\/|%'**t\_LOGICAL\_OPERATOR = **r'\=\=\=?|<=?|>=?|!=|&&|\|\|'**t\_ASSIGNMENT\_OPERATOR = **r'\='***# t\_BIN\_OPERATOR = r'&|\||\^|\~'*t\_STRING = **r'\"([^\"]+)\"|\'([^\']+)\''**t\_COMMA = **r'\,'***# t\_ID = r'[A-Za-z][A-Za-z0-9\_]\*'***def** t\_NUMBER(t):  
 *r'[0-9]+(\.[0-9]{0,15})?|\.[0-9]{1,15}'* **if** re.match(**'^[0-9]+$'**, t.value):  
 t.value = int(t.value)  
 **else**:  
 t.value = float(t.value)  
 **return** t  
  
  
**def** t\_error(t):  
 print(colorama.Fore.RED + **'\n!ERROR!\nIllegal character {0} at line {1}\n'**.format(t.value[0], t.lexer.lineno))  
 print(colorama.Fore.WHITE)  
 t.lexer.skip(1)  
  
  
**def** t\_newline(t):  
 *r'\n+'* t.lexer.lineno += 1  
  
  
**def** t\_METHOD\_OF\_CLASS\_CONSOLE(t):  
 *r'console.log'* t.type = reserved.get(t.value, **'METHOD\_OF\_CLASS\_CONSOLE'**)  
 **return** t  
  
  
**def** t\_ID(t):  
 *r'[A-Za-z][A-Za-z0-9\_]\*'* t.type = reserved.get(t.value, **'ID'**)  
 **return** t