Белорусский государственный университет информатики и радиоэлектроники

Кафедра информатики

Лабораторная работа № 2

**Синтаксический анализатор**

Выполнил:

Губский М.Д.

Проверил:

Шиманский В.В.

Минск, 2016

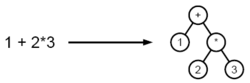
1. **Постановка задачи**

В данной работе ставится задача по исследованию области синтаксических анализаторов, рассмотрению аналогов и написанию своего собственного анализатора синтаксиса выбранного подмножества языка программирования. Требуется построить синтаксическое дерево.

1. **Теория**

**Синтакси́ческий ана́лиз**  в лингвистике и информатике — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

Синтаксический анализатор— это программа или часть программы, выполняющая синтаксический анализ.



Пример разбора выражения в дерево

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления для дальнейшей обработки.

## **Типы алгоритмов**

* Нисходящий парсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.
* Восходящий парсер — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.

1. **Программа и комментарии**

В результате выполнения работы использовали уже написанный лексический анализатор. Из выражений, уже выделенных, составлялись синтаксический конструкции по некоторым правилам. Эти правила можно увидеть в docstring каждой из функции. Код приведен приложении.

Дерево программы имеет вид:

block:

body:

expression:

declaration\_with\_assignment:

var

i

=

arithmetic\_expression:

10

declaration\_with\_assignment:

var

f

=

function\_declaration:

function

parameter:

arithmetic\_expression:

n

function\_body:

expression:

if:

logical\_expression:

n

<=

1

block:

body:

expression:

return:

expression:

arithmetic\_expression:

1

else:

block:

body:

expression:

return:

expression:

function\_implementation:

function\_implementation:

f

parameter:

arithmetic\_expression:

n

-

arithmetic\_expression:

1

function\_implementation:

f

parameter:

arithmetic\_expression:

n

-

arithmetic\_expression:

2

while:

logical\_expression:

logical\_expression:

i

>

0

||

logical\_expression:

i

==

10

body:

expression:

console.log:

expression:

function\_implementation:

f

parameter:

arithmetic\_expression:

i

arithmetic\_expression:

i

--

;

console.log:

expression:

arithmetic\_expression:

x

Рассмотрим текст программы с ошибками. При обнаружении их происходит вывод уведомления об ошибке.

Синтаксическая ошибка №1

{  
 var i=10;  
 var f=function(n){  
 if (n<=1){  
 return 1;  
 }  
 else{  
 return f(n-1)+f(n-2);  
 }  
 }  
 while (i>0 || i+10) {  
 console.log(f(i));  
 i--;  
 }  
}

Сообщение об ошибке:



Синтаксическая ошибка №2

{  
 var i=10;  
 var f=function(n){  
 if (n<=1){  
 return 1;  
 }  
 else{  
 return f(n-1)+f(n-2);  
 }  
 }  
 while (i>0 || i===10) {  
 console.log(f(i));  
 i--;  
 }  
 console.log(x==4);  
}

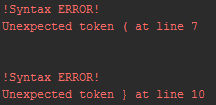
Сообщение об ошибке:



Синтаксическая ошибка №3

{  
 var i=10;  
 var f=function(n){  
 if (n<=1){  
 return 1;  
 }  
 else(){  
 return f(n-1)+f(n-2);  
 }  
 }  
 while (i>0 || i===10) {  
 console.log(f(i));  
 i--;  
 }  
 console.log(x);  
}

Сообщение об ошибке:



***Вывод***

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

**Приложение**

1. **Код разбора**

{

var i=10;

var f=function(n){

if (n<=1){

return 1;

}

else{

return f(n-1)+f(n-2);

}

}

while (i>0 || i==10) {

console.log(f(i));

i--;

}

console.log(x);

}

1. **Код программы**

Класс для представления узла с информацией о терминале для построения дерева.

**class** Node:  
 **def** parts\_str(self):  
 st = []  
 **for** part **in** self.parts:  
 st.append(part.\_\_str\_\_())  
 **return "\n"**.join(st)  
  
 **def** \_\_str\_\_(self):  
 **return** self.type + **":\n "** + self.parts\_str().replace(**"\n"**, **"\n "**)  
  
 **def** add\_child(self, parts):  
 **if** isinstance(parts, Node):  
 **if** parts.parts:  
 self.parts.extend(parts.parts)  
 **return** self  
  
 **def** \_\_init\_\_(self, type, parts):  
 self.type = type  
 self.parts = parts

Методы устанавливающие правила синтаксического анализа:

**def** p\_block(p):  
 *'''  
 block : LBRACKET body RBRACKET  
 '''* p[0] = Node(**'block'**, [p[2]])  
  
**def** p\_body(p):  
 *'''  
 body : expression  
 | function\_declaration  
 '''* p[0] = Node(**'body'**, [p[1]])  
  
**def** p\_expression(p):  
 *'''  
 expression : declaration\_with\_assignment expression  
 | declaration\_with\_assignment  
 | if expression  
 | else  
 | function\_body  
 | arithmetic\_expression  
 | function\_implementation  
 | while expression  
 | console\_function  
 | console\_function expression  
 '''* **if** len(p) == 2:  
 p[0] = Node(**'expression'**, [p[1]])  
 **else**:  
 p[0] = Node(**'expression'**, [p[1]]).add\_child(p[2])  
  
**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) == 5:  
 p[0] = Node(**'declaration\_with\_assignment'**, [p[1], p[2], p[3], p[4]])  
 **elif** len(p) == 6:  
 p[0] = Node(**'declaration\_with\_assignment'**, [p[1], p[2], p[3], p[4]])  
  
**def** p\_arithmetic\_expression(p):  
 *'''  
 arithmetic\_expression : ID MATH\_OPERATOR arithmetic\_expression  
 | ID MATH\_OPERATOR SEMICOLON  
 | NUMBER MATH\_OPERATOR arithmetic\_expression  
 | NUMBER  
 | ID  
 | STRING  
 '''* **if** len(p) == 2:  
 p[0] = Node(**'arithmetic\_expression'**, [p[1]])  
 **else**:  
 p[0] = Node(**'arithmetic\_expression'**, [p[1], p[2], p[3]])  
  
**def** p\_function\_declaration(p):  
 *'''  
 function\_declaration : FUNCTION LPAREN parameters RPAREN LBRACKET function\_body RBRACKET  
 '''* p[0] = Node(**'function\_declaration'**, [p[1], p[3], p[6]])  
  
**def** p\_function\_body(p):  
 *'''  
 function\_body : expression  
 | RETURN expression SEMICOLON  
 '''* **if** len(p) == 2:  
 p[0] = Node(**'function\_body'**, [p[1]])  
 **else**:  
 p[0] = Node(**'return'**, [p[2]])  
  
  
**def** p\_function\_implementation(p):  
 *'''  
 function\_implementation : ID LPAREN parameters RPAREN  
 | function\_implementation MATH\_OPERATOR function\_implementation  
 '''* **if** len(p) < 6:  
 p[0] = Node(**'function\_implementation'**, [p[1], p[3]])  
 **else**:  
 p[0] = Node(**'function\_implementation'**, [p[1], p[2], p[3]])  
  
**def** p\_parameters(p):  
 *"""  
 parameters : arithmetic\_expression  
 """* **if** len(p) == 1:  
 p[0] = []  
 **elif** len(p) == 2:  
 p[0] = Node(**'parameter'**, [p[1]])  
  
**def** p\_console\_function(p):  
 *'''  
 console\_function : METHOD\_OF\_CLASS\_CONSOLE LPAREN expression RPAREN SEMICOLON  
 '''* p[0] = Node(p[1], [p[3]])

**def** p\_predicate(p):  
 *'''  
 predicate : ID LOGICAL\_OPERATOR NUMBER  
 | predicate LOGICAL\_OPERATOR predicate  
 '''* p[0] = Node(**'logical\_expression'**, [p[1], p[2], p[3]])  
  
**def** p\_while(p):  
 *'''  
 while : WHILE LPAREN predicate RPAREN block  
 '''* p[0] = Node(**'while'**, [p[3]]).add\_child(p[5])  
  
**def** p\_if(p):  
 *'''  
 if : IF LPAREN predicate RPAREN block  
 '''* p[0] = Node(**'if'**, [p[3], p[5]])  
  
**def** p\_else(p):  
 *'''  
 else : ELSE block  
 '''* p[0] = Node(**'else'**, [p[2]])

**def** p\_error(p):  
 print(colorama.Fore.RED + **'\n!Syntax ERROR!\nUnexpected token {0} at line {1}\n'**.format(p.value[0], 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**(result)