Министерство образования Республики Беларусь Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе № 3

по теме «Синтаксический анализатор»

Выполнил: студент гр. 053501 Уласевич А.А.

Проверил: Ассистент кафедры

информатики Гриценко Н. Ю.

Минск 2023

**СОДЕРЖАНИЕ**

1. [Цель работы 2](#_bookmark0)
2. [Теоретические сведения 3](#_bookmark1)
3. [Результат выполнения программы 4](#_bookmark2)

[Приложение А(обязательное) Листинг программного кода 5](#_bookmark4)

# ЦЕЛЬ РАБОТЫ

Необходимо разработать синтаксический анализатор подмножества языка программирования С. В ходе синтаксического анализа исходный текст программы проверяется на соответствие синтаксическим нормам языка с построением дерева разбора (синтаксическое дерево), которое отражает синтаксическую структуру входной последовательности, а также в случае несоответствия – позволяет вывести сообщения об ошибках.

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.

Таким образом на основе анализа выражений, состоящих из литералов, операторов и круглых скобок выполняется группирование токенов исходной программы в грамматические фразы, используемые для синтеза вывода.

Представление грамматических фраз исходной программы выполнить в виде дерева. Реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR-метод, метод предшествования и пр.).

# ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Синтаксический анализ (СА) — это 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.

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

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.

Программа-обработчик представляет собой многопроходный анализатор, который может обрабатывать не только отдельные слова, но и целые предложения, используя контекст предложений и абзацев, и используя его при возникновении трудностей с омонимией или в случае неполных или непонятных предложений.

Разбиение на абзацы позволяет выделить основную мысль данного абзаца (если она, конечно, есть), создавая тем самым контекст. Формально контекстом параграфа можно считать все пары подлежащих и сказуемых, встречающихся во всех предложениях данного абзаца. Вся основная работа производится на уровне предложений. Алгоритм анализа предложения достаточно прост и может быть описан в виде состояний конечного автомата.

# РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Результат работы синтаксического анализатора представлен в виде синтаксического дерева на рисунке 1.

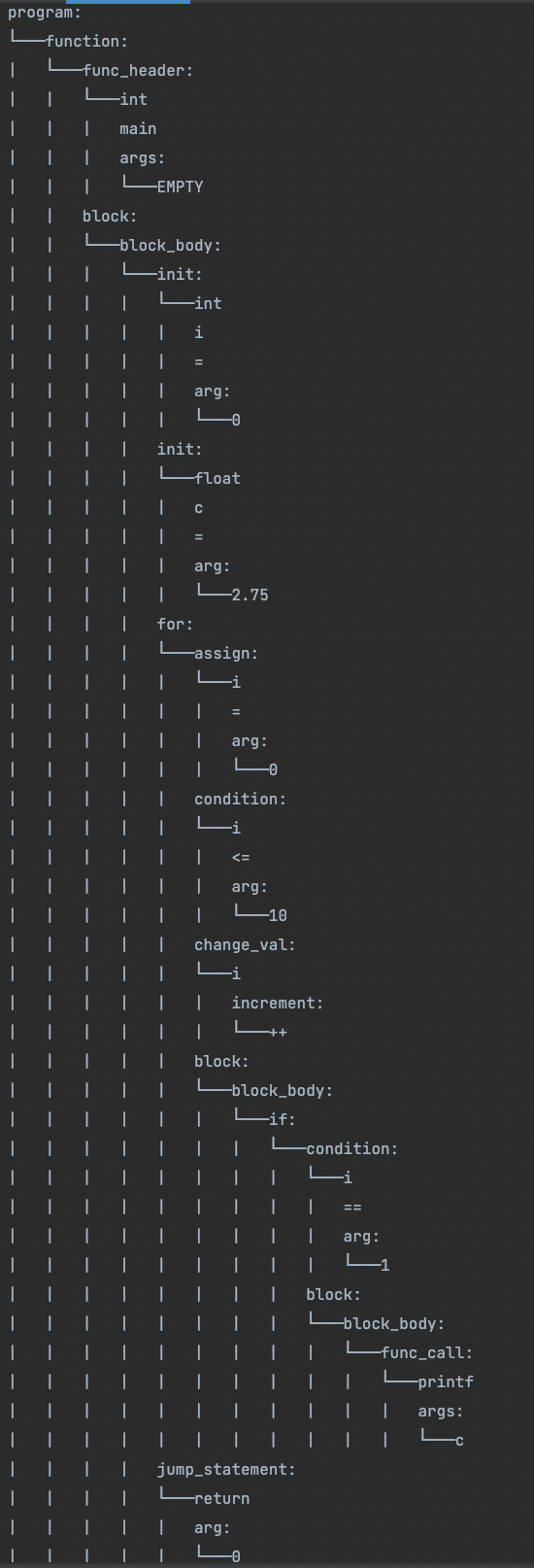


Рисунок 1 – Синтаксическое дерево

# ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг программного кода

Файл parser.py:

from ply import yacc  
from rules import tokens  
  
  
class Parser:  
 def \_\_init\_\_(self, text):  
 self.text = text  
 self.parser = yacc.yacc()  
  
 def parse(self):  
 tree = self.parser.parse(self.text)  
 print(tree)  
  
  
class Node:  
 def parts\_str(self):  
 return "\n".join(map(str, [x for x in self.children]))  
  
 def \_\_init\_\_(self, type, children):  
 self.type = type  
 self.children = children  
  
 def \_\_str\_\_(self):  
 return self.type + ":\n└───" + self.parts\_str().replace("\n", "\n|\t")  
  
 def add\_parts(self, children):  
 self.children += children  
 return self  
  
  
def p\_program(p):  
 """program : function"""  
 p[0] = Node("program", p[1:])  
  
  
def p\_function(p):  
 """function : func\_header block"""  
  
 p[0] = Node("function", p[1:])  
  
  
def p\_func\_header(p):  
 """func\_header : DATA\_TYPE FUNCDECL LPAR args RPAR"""  
 p[0] = Node("func\_header", [p[1], p[2], p[4]])  
  
  
def p\_args(p):  
 """args :  
 | expr  
 | args COMMA expr"""  
 if len(p) <= 2:  
 p[0] = Node("args", p[1:] if p[1:] else ["EMPTY"])  
 else:  
 p[0] = p[1].add\_parts([p[3]])  
  
  
def p\_block(p):  
 """block : LCURL body RCURL"""  
 p[0] = Node("block", [p[2]])  
  
  
def p\_body(p):  
 """body :  
 | body line semicolons  
 | body multiline"""  
 if len(p) > 1:  
 if p[1] is None:  
 p[1] = Node("block\_body", [])  
 p[0] = p[1].add\_parts([p[2]])  
 else:  
 p[0] = Node("block\_body", [])  
  
  
def p\_semicolons(p):  
 """semicolons : SEMICOLON  
 | semicolons SEMICOLON"""  
  
  
def p\_multiline(p):  
 """multiline : if\_statement  
 | while\_statement  
 | for\_statement"""  
 p[0] = p[1]  
  
  
def p\_line(p):  
 """line : jump\_statement  
 | init  
 | func  
 | assign"""  
 p[0] = p[1]  
  
  
def p\_jump\_statement(p):  
 """jump\_statement : RETURN arg  
 | BREAK  
 | CONTINUE"""  
 if len(p) == 3:  
 p[0] = Node("jump\_statement", p[1:])  
 else:  
 p[0] = Node("jump\_statement", [p[1]])  
  
  
def p\_indexing\_op(p):  
 """indexing\_op : ID LCUADR expr RCUADR"""  
 p[0] = Node("indexing\_op", [p[1], p[3]])  
  
  
def p\_if\_statement(p):  
 """if\_statement : IF LPAR condition RPAR block  
 | if\_statement ELSE block"""  
 if len(p) == 4:  
 p[0] = p[1].add\_parts(["else", p[3]])  
 else:  
 p[0] = Node("if", [p[3], p[5]])  
  
  
def p\_while\_statement(p):  
 """while\_statement : WHILE LPAR condition RPAR block"""  
 p[0] = Node("while", [p[3], p[5]])  
  
  
def p\_for\_statement(p):  
 """for\_statement : FOR LPAR assign SEMICOLON condition SEMICOLON change\_val RPAR block"""  
 p[0] = Node("for", [p[3], p[5], p[7], p[9]])  
  
  
def p\_change\_value(p):  
 """change\_val : ID expr"""  
 p[0] = Node("change\_val", p[1:])  
  
  
def p\_condition(p):  
 """condition : expr cond\_sign expr"""  
 p[0] = Node("condition", [p[1], p[2], p[3]])  
  
  
def p\_cond\_sign(p):  
 """cond\_sign : DEQUAL  
 | GT  
 | LT  
 | GE  
 | LE  
 | NOTEQUAL"""  
 p[0] = p[1]  
  
  
def p\_init(p):  
 """init :  
 | DATA\_TYPE ID SEMICOLON  
 | DATA\_TYPE ID EQUAL expr  
 | DATA\_TYPE ID EQUAL indexing\_op  
 | DATA\_TYPE ID LCUADR RCUADR EQUAL array\_init"""  
 if len(p) > 5:  
 p[0] = Node("init", [p[1], p[2], "[]", p[5], p[6]])  
 else:  
 p[0] = Node("init", p[1:])  
  
  
def p\_array\_init(p):  
 """array\_init : LCURL init\_block RCURL"""  
 p[0] = Node("array\_init", [p[2]])  
  
  
def p\_init\_block(p):  
 """init\_block : arg  
 | arg COMMA  
 | init\_block arg  
 | init\_block arg COMMA"""  
 if len(p) == 2:  
 p[0] = Node("init\_block", p[1:])  
 else:  
 if p[2] != ",":  
 p[0] = p[1].add\_parts(p[2:])  
 else:  
 p[0] = Node("init\_block", p[1:])  
  
  
def p\_assign(p):  
 """assign : ID EQUAL expr  
 | ID EQUAL indexing\_op  
 | indexing\_op EQUAL expr  
 | indexing\_op EQUAL indexing\_op"""  
 if len(p) == 5:  
 p[0] = Node("assign", [p[2], p[4]])  
 elif len(p) == 4 or len(p) == 3:  
 p[0] = Node("assign", p[1:])  
 else:  
 p[0] = Node("assign", [p[1], p[3]])  
  
  
def p\_func(p):  
 """func : CUSTOM\_FUNC LPAR args RPAR  
 | ID LPAR args RPAR  
 | BUILD\_IN LPAR args RPAR"""  
 if len(p) == 3:  
 p[0] = Node("func\_call", [p[1], p[2]])  
 else:  
 p[0] = Node("func\_call", [p[1], p[3]])  
  
  
def p\_expr(p):  
 """expr : fact  
 | PLUSMINUS PLUSMINUS  
 | expr PLUSMINUS fact  
 | expr MOD fact  
 | ID PLUSMINUS ID  
 | ID PLUSMINUS fact  
 | ID DIVMUL fact  
 | fact DIVMUL ID  
 | ID  
 """  
 if len(p) == 2:  
 p[0] = p[1]  
 elif len(p) == 3:  
 if p[2] == "+":  
 p[0] = Node("increment", ["++"])  
 elif p[2] == "-":  
 p[0] = Node("decrement", ["--"])  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
  
  
def p\_fact(p):  
 """fact : term  
 | fact DIVMUL term"""  
 if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
  
  
def p\_term(p):  
 """term : arg  
 | LPAR expr RPAR"""  
 if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = p[2]  
  
  
def p\_arg(p):  
 """arg : NUMBER  
 | STRING  
 | DATA\_TYPE ID  
 | DATA\_TYPE ID LCUADR RCUADR  
 | ID LCUADR RCUADR  
 | indexing\_op  
 | NUMBER ID  
 | func"""  
 if len(p) == 2:  
 p[0] = Node("arg", [p[1]])  
 else:  
 p[0] = Node("arg", p[1:])  
  
  
def p\_error(p):  
 raise SyntaxError(str(p.lineno) + ":" + str(p.lexpos) + " Error: unexpected token '" + str(p.value) + "'")

Файл rules.py:

from ply.lex import TOKEN  
  
tokens = (  
 "FUNCDECL",  
 "LPAR",  
 "RPAR",  
 "COMMA",  
 "LCURL",  
 "RCURL",  
 "LCUADR",  
 "RCUADR",  
 "CUSTOM\_FUNC",  
 "EQUAL",  
 "SEMICOLON",  
 "NUMBER",  
 "DATA\_TYPE",  
 "ID",  
 "BUILD\_IN",  
 "PLUSMINUS",  
 "DIVMUL",  
 "STRING",  
 "IF",  
 "ELSE",  
 "DEQUAL",  
 "RETURN",  
 "GT",  
 "LT",  
 "GE",  
 "LE",  
 "MOD",  
 "NOTEQUAL",  
 "WHILE",  
 "FOR",  
 "CONTINUE",  
 "BREAK",  
)  
  
identifier = r"[a-zA-Z]\w\*"  
  
types = {  
 "int": "DATA\_TYPE",  
 "float": "DATA\_TYPE",  
 "double": "DATA\_TYPE",  
 "char": "DATA\_TYPE",  
 "void": "DATA\_TYPE",  
}  
  
reserved = {  
 "if": "IF",  
 "else": "ELSE",  
 "auto": "DATA\_TYPE",  
 "while": "WHILE",  
 "for": "FOR",  
 "break": "BREAK",  
 "continue": "CONTINUE",  
 "return": "RETURN",  
 "sizeof": "BUILD\_IN",  
 "printf": "BUILD\_IN",  
 "scanf": "BUILD\_IN",  
}  
  
t\_LCUADR = r"\["  
t\_RCUADR = r"\]"  
t\_LPAR = r"\("  
t\_RPAR = r"\)"  
t\_COMMA = r","  
t\_LCURL = r"\{"  
t\_RCURL = r"\}"  
t\_DEQUAL = r"\=\="  
t\_GE = r"\>\="  
t\_LE = r"\<\="  
t\_GT = r"\>"  
t\_LT = r"\<"  
t\_MOD = r"\%"  
t\_NOTEQUAL = r"!\="  
t\_EQUAL = r"\="  
t\_SEMICOLON = r";"  
t\_PLUSMINUS = r"\+|\-"  
t\_DIVMUL = r"/|\\*"  
t\_STRING = r'("(\\.|[^"])\*")|(\'(\\.|[^\'])\*\')'  
  
  
def t\_newline(t):  
 r"\n+"  
 t.lexer.lineno += len(t.value)  
  
  
def t\_NUMBER(t):  
 r"[0-9.]+"  
 try:  
 t.value = int(t.value)  
 except BaseException:  
 try:  
 t.value = float(t.value)  
 except BaseException:  
 t.value = None  
 return t  
  
  
class TypeDefine:  
 type\_define = False  
  
  
@TOKEN(identifier)  
def t\_ID(t):  
 if TypeDefine.type\_define:  
 TypeDefine.type\_define = False  
 if t.lexer.lexdata[t.lexpos + len(t.value)] == "(":  
 reserved[t.value] = "CUSTOM\_FUNC"  
 t.type = "FUNCDECL"  
 else:  
 t.type = "ID"  
 else:  
 if t.lexer.lexdata[t.lexpos + len(t.value)] == "(":  
 if (value := reserved.get(t.value, None)) is None:  
 print("error")  
 else:  
 t.type = value  
 else:  
 if (res := types.get(t.value, "ID")) == "DATA\_TYPE":  
 TypeDefine.type\_define = True  
  
 t.type = res if t.value not in reserved else reserved[t.value]  
  
 return t  
  
  
def t\_error(t):  
 # print("Illegal character '%s' at line %d" % (t.value[0], t.lineno))  
 t.lexer.skip(1)

Файл test.c:

int main()  
{  
  
 int i[] = {1,2,3,4};  
  
 int b = i[0];  
  
 float c = 2.75;  
  
 for (i = 1; i <= 20 + 3; i++)  
  
 if (i == 1)  
 {  
 printf(c);  
 continue;  
 }  
 if (i == 2)  
 {  
 printf(c);  
 continue;  
 }  
 c = i + c;  
 c = i;  
 i = c;  
 printf(c);  
 }  
 return 0;  
}