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

Необходимо разработать синтаксический анализатор подмножества языка программирования C. 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.).

2 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.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

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

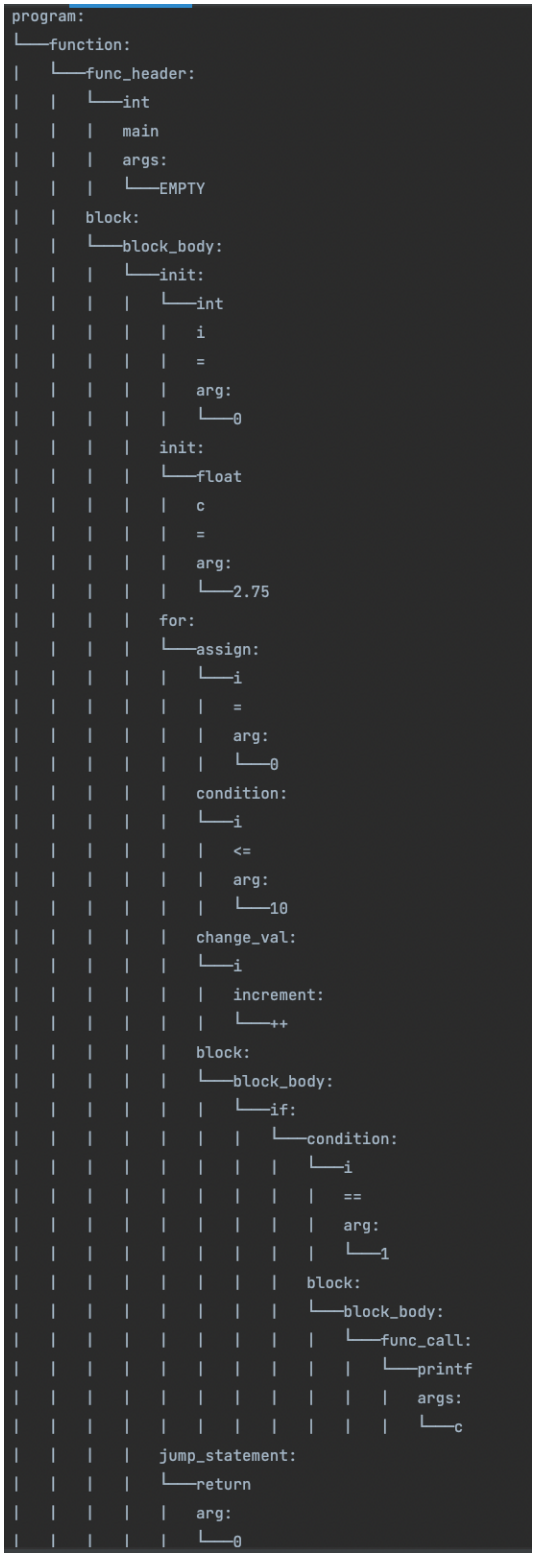

Рисунок 1 – Синтаксическое дерево

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;
}
```