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. 853504
Шевченя И.В.
Проверил:
Ст. преподаватель КИ Шиманский В. В.

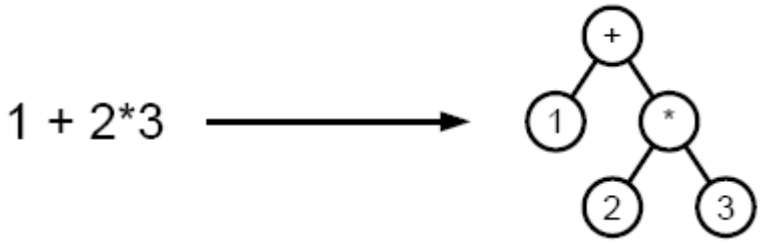
Минск 2021

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. В качестве анализируемого языка программирования был использован язык программирования C++.

2 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 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.

Типы алгоритмов:

- Нисходящий парсер (англ. top-down parser) — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.
- Метод рекурсивного спуска
- LL-анализатор
- Восходящий парсер (англ. bottom-up parser) — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.
- LR-анализатор
- GLR-парсер

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. В случае односоставных предложений используется только один состав предложения. Если его, конечно же, удается найти.

Вся основная работа производится на уровне предложений. Алгоритм анализа предложения достаточно прост и может быть описан в виде состояний конечного автомата.

Области применения синтаксических деревьев:

- Языки программирования — разбор исходного кода языков программирования, в процессе трансляции (компиляции или интерпретации);
- Структурированные данные — данные, языки их описания, оформления и т. д. Например, XML, HTML, CSS, JSON, ini-файлы, специализированные конфигурационные файлы и т. п.;
- Построение индекса в поисковой системе;
- SQL-запросы (DSL-язык);
- Математические выражения;
- Регулярные выражения (которые, в свою очередь, могут использоваться для автоматизации лексического анализа);
- Формальные грамматики;
- Лингвистика — естественные языки. Например, машинный перевод и другие генераторы текстов.
- Извлечение данных веб-страниц — веб-скрейпинг, является частным случаем парсинга.

Простейший способ реагирования на некорректную входную цепочку лексем — завершить синтаксический анализ и вывести сообщение об ошибке. Однако часто оказывается полезным найти за одну попытку синтаксического анализа как можно больше ошибок. Именно так ведут себя трансляторы большинства распространённых языков программирования.

Таким образом, перед обработчиком ошибок синтаксического анализатора стоят следующие задачи:

1. он должен ясно и точно сообщать о наличии ошибок;
2. он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжать поиск других ошибок;
3. он не должен существенно замедлять обработку корректной входной цепочки.

3. Программа и комментарии:

Код парсера:

```
from lexer_constants import *


class Parser(object):
    """
    class Parser

    """

    def __init__(self):
        self.tokens = None

    def _node(self, pos):
        """
        return new node and pos

        """
        possible_values = {
            '{': '}',
            '(': ')',
        }

        node = list()
        while self.tokens[pos].value not in
possible_values.values():
            delimiter = self.tokens[pos]
            if delimiter.value in possible_values.keys():
                new_node, pos = self._node(pos + 1)
                node.append([delimiter.value, new_node,
possible_values[delimiter.value]])
            else:
                if delimiter.value in [*ARITHMETIC_OPERATIONS,
*OVERRIDE_OPERATION, *COMPARE_SIGNS, ',']:
                    node.append([self.tokens[pos]])
                elif delimiter.tag in [WHILE, FOR, IF]:
                    node.extend([self.tokens[pos],
["condition:"]])
                else:
                    node.append(self.tokens[pos])
            pos += 1

        return node, pos

    def build(self, tokens):
        """
        return ast

        """
        ast = list()
```

```
        if tokens:
            pos = 0
            self.tokens = tokens
            ast.append("program: ")
            while pos < len(tokens):
                if tokens[pos].value == '{':
                    node, pos = self._node(pos + 1)
                    pos += 1
                    ast.append([node])
                else:
                    if tokens[pos].tag == TYPE:
                        pos_copy = pos
                        if tokens[pos + 1].tag ==
FUNC_DECLARATION:
                            node, pos = self._node(pos + 3)
                            ast.append(["function declaration:",
tokens[pos_copy].value, tokens[pos_copy + 1].value,
                                        ["args:", "(", node,
")"], "body:"])
                            pos += 1
                            continue

                    msg = 'Parser error! Expected "{" but given
"%s"' % tokens[pos].value
                    msg += ' in line {}, column
{}'.format(tokens[pos].col - 1, tokens[pos].row)
                    raise Exception(msg)
            ast.append("end program")

        return ast


def syntax_analyzer(ast, tabs):
    for i in ast:
        if isinstance(i, list):
            syntax_analyzer(i, tabs + 1)
        else:
            result, value = tabs * '  |', i.value if not
isinstance(i, str) else i
            print('{}{}'.format(result, value))


if __name__ == "__main__":
    path = "main_1.cpp"
    lexer = Lexer(path)
    parser = Parser()
    tokens = lexer.tokens()
    if len(check_if_main_exist(tokens)) == 0:
        lexer.error("Program should have function 'main'")

    if lexer.errors_list:
        lexer.errors()
```

```
    draw_result_table(tokens)
    draw_tags_groups(tokens)

    ast = parser.build(tokens)
    syntax_analyzer(ast, 2)
```

Расширение кода для более детального анализа:

```
class Node(object):
    def parts_str(self):
        return '\n'.join(map(str, [x for x in self.parts]))

    def __init__(self, type, parts):
        self.type = type
        self.parts = parts

    def __str__(self):
        return self.type + ":\n____" +
self.parts_str().replace("\n", "\n____")

    def add_parts(self, parts):
        self.parts += parts
        return self


def p_program(p):
    '''program :
               | function
               | program function
    '''
    if len(p) <= 2:
        p[0] = Node('program', p[1:])
    else:
        p[0] = p[1].add_parts([p[2]])


def p_function(p):
    '''function : func_header func_body

    '''

    p[0] = Node('function', p[1:])


def p_func_header(p):
    '''func_header : VARIABLE_TYPE FUNCDECL LPAR args RPAR'''
    p[0] = Node('func_declaration', [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_func_body(p):
    '''func_body : block'''
    p[0] = p[1]


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('body', [])
        p[0] = p[1].add_parts([p[2]])
    else:
        p[0] = Node('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 : modal_function
            | init
            | func
            | assign'''
    p[0] = p[1]


def p_modal_function(p):
    '''modal_function : RETURN arg
                      | BREAK
                      | CONTINUE'''
    if len(p) == 3:
```

```
        p[0] = Node("modal_function", p[1:])
    else:
        p[0] = Node("modal_function", [p[1]])


def p_var_call(p):
    '''var_cal : ID LCUADR expr RCUADR'''
    p[0] = Node('var_call', [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 init 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 :
            | VARIABLE_TYPE ID
```

```
            | VARIABLE_TYPE ID EQUAL ID DIVMUL NUMBER
            | VARIABLE_TYPE ID EQUAL expr
            | VARIABLE_TYPE ID EQUAL var_cal
            | VARIABLE_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 var_cal
              | var_cal EQUAL expr
              | var_cal EQUAL var_cal
              | ID expr'''
    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
            | BUILD_IN output_operator'''
    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_output_operator(p):
    '''output_operator : LT LT arg
                       | LT LT BUILD_IN
                       | LT LT ID'''

    p[0] = Node('output_operator', ["<<", p[3]])


def p_expr(p):
    '''expr : fact
            | PLUSMINUS PLUSMINUS
            | expr PLUSMINUS fact
            | expr MOD fact
            | ID'''
    if len(p) == 2:
        p[0] = p[1]
    elif len(p) == 3:
        if p[2] == "+":
            p[0] = Node('assign', ["++"])
        elif p[2] == "-":
            p[0] = Node('assign', ["--"])
    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
           | VARIABLE_TYPE ID
           | VARIABLE_TYPE ID LCUADR RCUADR
           | ID LCUADR RCUADR
           | var_cal
           | NUMBER ID
           | func'''
```

```
    if len(p) == 2:
        p[0] = Node('arg', [p[1]])
    else:
        p[0] = Node('arg', p[1:])


def p_error(p):
    print('Unexpected token in line %d: %s' % (p.lineno, p))


def build_tree(code):
    parser = yacc.yacc()
    return parser.parse(code, debug=True)


if __name__ == '__main__':
    tree = build_tree(data)
    print(tree)
```

Код программ для разбора:

1)

```
void reverseArray(int arr[], int start, int end)
{
 while (start < end)
 {
 int temp = arr[start];
 arr[start] = arr[end];
 arr[end] = temp;
 start++;
 end--;
 }
}


void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
       cout << arr[i] << " ";

   cout << endl;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);

    reverseArray(arr, 0, n - 1);
```

```
    cout << "Reversed array is";
    cout << endl;

    printArray(arr, n);

    if (a < b)
    {
        int t = 10;
    }
    else {
        int t = 15;
    }

    cout << t;

    return 0;
}
```

2)

```
void merge(int *a, int n)
{
    int mid = n / 2;
    if (n % 2 == 1)
    {
        mid++;
    }
    int h = 1;
    int *c = (int*)malloc(n * sizeof(int));
    int step;
    while (h < n)
    {
        step = h;
        int i = 0;
        int j = mid;
        int k = 0;
        while (step <= mid)
        {
            while ((i < step) && (j < n) && (j < (mid + step)))
            {
                if (a[i] < a[j])
                {
                    c[k] = a[i];
                    i++; k++;
                }
                else {
                    c[k] = a[j];
                    j++; k++;
                }
            }
            while (i < step)
            {
```

```
                c[k] = a[i];
                i++; k++;
            }
            while ((j < (mid + step)) && (j<n))
            {
                c[k] = a[j];
                j++; k++;
            }
            step = step + h;
        }
        h = h * 2;

        for (i = 0; i < n; i++)
        {
          a[i] = c[i];
        }

    }
}


int main()
{
    int a[8];
    for (int i = 0; i < 8; i++)
    {
        a[i] = rand() % 20 - 10;
    }
    for (int i = 0; i < 8; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
    merge(a, 8);
    for (int i = 0; i < 8; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");

    getchar();
    return 0;
}
```

Результат работы парсера в виде синтаксического дерева:

```
  program:
____function:
________func_declaration:
____________void
____________reverseArray
____________args:
________________arg:
```

```
_____________________int
_____________________arr
_____________________[
_____________________]
___________________arg:
_____________________int
_____________________start
___________________arg:
_____________________int
_____________________end
________block:
_____________body:
___________________while:
_____________________condition:
_________________________start
_________________________<
_________________________end
_____________________block:
_________________________body:
_____________________________init:
_____________________________int
_____________________________temp
_____________________________=
_____________________________var_call:
_______________________________arr
_______________________________start
___________________________assign:
_____________________________var_call:
_______________________________arr
_______________________________start
_____________________________=
_____________________________var_call:
_______________________________arr
_______________________________end
___________________________assign:
_____________________________var_call:
_______________________________arr
_______________________________end
_____________________________=
_____________________________temp
___________________________assign:
_____________________________start
_____________________________assign:
_______________________________++
___________________________assign:
_____________________________end
_____________________________assign:
_______________________________--
____function:
________func_declaration:
_____________void
_____________printArray
_____________args:
```

```
__________________arg:
_____________________int
_____________________arr
_____________________[
_____________________]
__________________arg:
_____________________int
_____________________size
________block:
_____________body:
__________________for:
_____________________init:
_________________________int
_________________________i
_________________________=
_________________________arg:
_____________________________0
_____________________condition:
_________________________i
_________________________<
_________________________size
_____________________change_val:
_________________________i
_________________________assign:
_____________________________++
_____________________block:
_________________________body:
_____________________________func_call:
_____________________________________cout
_____________________________________output_operator:
_________________________________________<<
_________________________________________arg:
_____________________________________________var_call:
_____________________________________________arr
_____________________________________________i
_____________________________func_call:
_____________________________________cout
_____________________________________output_operator:
_________________________________________<<
_________________________________________arg:
_____________________________________________" "
__________________func_call:
_____________________cout
_____________________output_operator:
_________________________<<
_________________________endl
____function:
________func_declaration:
_____________int
_____________main
_____________args:
__________________EMPTY
________block:
```

```
_____________body:
_________________init:
_____________________int
_____________________[]
_____________________arr
_____________________=
_____________________array_init:
_________________________init_block:
_________________________________arg:
_____________________________________1
_________________________________,
_________________________________arg:
_____________________________________2
_________________________________,
_________________________________arg:
_____________________________________3
_________________________________,
_________________________________arg:
_____________________________________4
_________________________________,
_________________________________arg:
_____________________________________5
_________________________________,
_________________________________arg:
_____________________________________6
_________________init:
_____________________int
_____________________n
_____________________=
_____________________/:
_________________________arg:
_____________________________func_call:
_____________________________________sizeof
_____________________________________args:
_________________________________________arr
_________________________arg:
_____________________________func_call:
_____________________________________sizeof
_____________________________________args:
_________________________________________arg:
_____________________________________________var_call:
_____________________________________________arr
_____________________________________________arg:
_____________________________________________0
_________________func_call:
_____________________printArray
_____________________args:
_________________________arr
_________________________n
_________________func_call:
_____________________reverseArray
_____________________args:
_________________________arr
```

```
_____________________________arg:
_____________________________0
_____________________________-:
_____________________________n
_____________________________arg:
_____________________________1
____________________func_call:
____________________cout
____________________output_operator:
____________________<<
____________________arg:
____________________"Reversed array is"
____________________func_call:
____________________cout
____________________output_operator:
____________________<<
____________________endl
____________________func_call:
____________________printArray
____________________args:
____________________arr
____________________n
____________________if:
____________________condition:
____________________a
____________________<
____________________b
____________________block:
____________________body:
_____________________________init:
_____________________________int
_____________________________t
_____________________________=
_____________________________arg:
_____________________________10
____________________else
____________________block:
____________________body:
_____________________________init:
_____________________________int
_____________________________t
_____________________________=
_____________________________arg:
_____________________________15
____________________func_call:
____________________cout
____________________output_operator:
____________________<<
____________________t
____________________modal_function:
____________________return
____________________arg:
____________________0
```

Рассмотрим текст программы с ошибками. При обнаружении их происходит вывод уведомления об ошибке (красным выделено место ошибки или отсутствующий символ/фрагмент):

1 Отсутствие открывающейся фигурной скобки:
Фрагмент кода:

```
int main()
{
 int arr[] = {1, 2, 3, 4, 5, 6};
 t n = sizeof(arr) / sizeof(arr[0]);

 printArray(arr, n);
 verseArray(arr, 0, n - 1);
 ut << "Reversed array is" << endl;
 intArray(arr, n);

 return 0;
}
```

Код ошибки:

```
Traceback (most recent call last):
  File "/home/shevchenya/PycharmProjects/mtran/lexer.py", line 562, in <module>
    ast = parser.build(tokens)
  File "/home/shevchenya/PycharmProjects/mtran/parser.py", line 72, in build
    raise Exception(msg)
Exception: Parser error! Expected "{" but given "int" in line 29
```

2 Отсутствие синтаксически необходимого блока — функции с именем main
Код ошибки:

```
Lexer errors:
    Program should have function 'main' in line 43, column 1
```

3 Появление неизвестного символа:
Код ошибки:

```
Unknown character: "#" in line 1, column 2
Unknown character: "." in line 1, column 17
Missing end of line:  in line 1, column 19
Unknown character: "#" in line 2, column 2
Missing end of line:  in line 2, column 20
```

4 Отсутствие синтаксически необходимого символа окончания строки:

Код ошибки:

```
Missing end of line:  in line 22, column 29
```

5 Отсутсвие необходимой закрывающей фигурной скобки скобки
Код ошибки:

```
Traceback (most recent call last):
  File "/home/shevchenya/PycharmProjects/mtran/lexer.py", line 562, in <module>
    ast = parser.build(tokens)
  File "/home/shevchenya/PycharmProjects/mtran/parser.py", line 57, in build
    node, pos = self._node(pos + 1)
  File "/home/shevchenya/PycharmProjects/mtran/parser.py", line 41, in _node
    raise Exception(msg)
Exception: Parser error! Missing symbol "}" in line 43, column 0
```

6 Отсутствие в определении параметров функции закрывающей круглой скобки:
Код ошибки:

```
Traceback (most recent call last):
  File "/home/shevchenya/PycharmProjects/mtran/lexer.py", line 562, in <module>
    ast = parser.build(tokens)
  File "/home/shevchenya/PycharmProjects/mtran/parser.py", line 57, in build
    node, pos = self._node(pos + 1)
  File "/home/shevchenya/PycharmProjects/mtran/parser.py", line 41, in _node
    raise Exception(msg)
Exception: Parser error! Missing symbol ")" in line 19, column 28
```

Вывод:

В результате была создана программа, строящая синтаксическое дерево по коду программы на языке программирования C++. Программа позволяет отслеживать синтаксические ошибки и выводит сообщения о них. Программа пытается выделить синтаксические конструкции из групп токенов согласно определённой грамматике. Если это не получается — значит в коде присутствует синтаксическая ошибка. Об этом сигнализируется пользователю.