Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ
по лабораторной работе
на тему

Синтаксический анализатор

Выполнил
Студент гр. 053502
Юрьев В.А.

Проверил
Ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Разработка синтаксического анализатора языка программирования, определенного в лабораторной работе 1. Построение синтаксического дерева. Показать корректность работы анализатора допущением синтаксических ошибок.

2 РЕЗУЛЬТАТЫ

2.1 Построение синтаксического дерева

В качестве тестовой программы был использован алгоритм для возведения целого числа в степень (рисунок 2.1).

```
int main()
{
    int v;
    int p;

    cin >> v >> p;

    int result = 1;
    while (p > 0)
    {
        if (p % 2 == 1)
        {
            result *= v;
        }

        v *= v;
        p = p / 2;
    }

    cout << result;
}
```

Рисунок 2.1 – Тестовая программа

Результат построения синтаксического дерева анализатором при отсутствии ошибок (рисунки 2.2, 2.3).

```
function
main
      cin
        v
        p
      result
      =
      1
```

Рисунок 2.2 – Первая часть синтаксического дерева

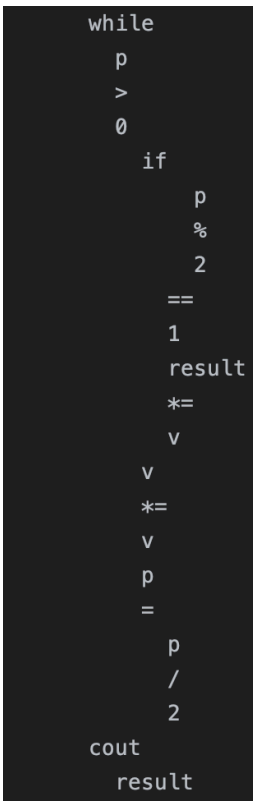

Рисунок 2.3 – Вторая часть синтаксического дерева

2.2 Обнаружение синтаксических ошибок

1) Отсутствие ; после объявления переменной (см. рисунки 2.4, 2.5).

```
int main()
{
    int v;
    int p

    cin >> v >> p;

    int result = 1;
    while (p > 0)
    {
        if (p % 2 == 1)
        {
            result *= v;
        }

        v *= v;
        p = p / 2;
    }

    cout << result;
}
```

Рисунок 2.4 – Код программы с отсутствующим символом ;

```
Exception: Expected [';'] after p
```

Рисунок 2.5 – Пример реакции анализатора на отсутствие символа ;

2) Отсутствие блока условия для цикла while (см. рисунки 2.6, 2.7).

```
int main()
{
    int v;
    int p;

    cin >> v >> p;

    int result = 1;
    while
    {
        if (p % 2 == 1)
        {
            result *= v;
        }

        v *= v;
        p = p / 2;
    }

    cout << result;
}
```

Рисунок 2.6 – Код программы с отсутствием условия для цикла while

```
Exception: Expected ['('] after while
```

Рисунок 2.7 – Пример реакции анализатора на отсутствие условия для цикла while

3) Ошибочная попытка вывода оператором cout (см. рисунки 2.8, 2.9).

```cpp
int main()
{
    int v;
    int p;

    cout << cin;
    cin >> v >> p;

    int result = 1;
    while (p > 0)
    {
        if (p % 2 == 1)
        {
            result *= v;
        }

        v *= v;
        p = p / 2;
    }

    cout << result;
}
```

Рисунок 2.8 – Код программы с ошибочной попыткой вывода

```
Exception: Expected number or variable after  <<
```

Рисунок 2.9 – Реакции анализатора на ошибочную попытку вывода

4) Попытка выполнения ошибочного действия (см. рисунки 2.10, 2.11).

```cpp
int main()
{
    int v;
    int p;

    cin >> v >> p;

    int result = 1;
    while (p > 0)
    {
        if (p % 2 == 1)
        {
            result *= continue;
        }

        v *= v;
        p = p / 2;
    }

    cout << result;
}
```

Рисунок 2.10 – Код программы с ошибочным действием

```
Exception: Expected number or variable after  *=
```

Рисунок 2.11 – Реакция анализатора на ошибочное действие

ПРИЛОЖЕНИЕ А
Код программ

```
from functions.lexer import Lexer
from nodes.nodes_module import *
from entities.constants import all_operators, ignore, libs, namespaces


class Parser:
    def __init__(self, lexer: Lexer):
        self.tokens = lexer.tokens
        self.position = 0
        self.scope = {}
        self.lexer = lexer  # instead of

    def match(self, expected: []) -> Token:
        if self.position < len(self.tokens):
            current_token = self.tokens[self.position]
            print(current_token.word, expected, current_token.word in expected)
            if current_token.word in expected:
                self.position += 1
                return current_token

        return None

    def get_prev(self):
        return self.tokens[self.position - 1].word

    def require(self, expected):
        token = self.match(expected)
        if token is None:
            raise Exception(f'Expected {expected} after {self.get_prev()}')

        return token

    def parse_variable_or_constant(self) -> Node:
        constant = self.match(self.lexer.constants_tokens.keys())
        if constant:
            return ConstantNode(constant)

        var = self.match(self.lexer.var_tokens.keys())
        if var:
            return VariableNode(var)

        constant = self.tokens[self.position]
        if constant.word == 'true' or constant.word == 'false':
            self.position += 1
            return ConstantNode(constant)
        elif constant.word == "!":
            self.position += 1
```

```
            return UnaryOperationNode(constant, self.parse_formula())

        raise Exception(f'Expected number or variable after  {self.get_prev()}')

    def parse_parentheses(self) -> Node:
        if self.match(['(']):
            node = self.parse_formula()
            self.require([')'])

            return node

        else:
            return self.parse_variable_or_constant()

    def parse_formula(self) -> Node:
        left_node = self.parse_parentheses()
        operation = self.match(all_operators)

        while operation:
            if operation.word == '<<' or operation.word == '>>':
                self.position -= 1
                break
            elif operation.word == '++' or operation.word == '--':
                left_node = UnaryOperationNode(operation, left_node)
                operation = self.match(all_operators + [':'])
            else:
                right_node = self.parse_parentheses()
                left_node = BinaryOperationNode(operation, left_node, right_node)
                operation = self.match(all_operators + [':'])

        return left_node

    def parse_variable_definition(self, variable_token):
        var = VariableNode(variable_token)

        comma = self.match([','])
        while comma:
            new_var = self.require(self.lexer.var_tokens.keys())
            if new_var:
                var = VariableNode(new_var)
            else:
                raise Exception(f'Expected variable after "," after {self.get_prev()}')

            comma = self.match([','])

        operation = self.match(['='])
        if operation:
            value = self.parse_formula()
            self.require([';'])
            return BinaryOperationNode(operation, var, value)
```

```
        self.require([';'])
        return None

    def parse_cin(self):
        operation = self.match(['>>'])
        expression = []

        while operation:
            expression.append(self.parse_formula())
            operation = self.match(['>>'])

        self.require([';'])
        return CinNode(expression)

    def parse_cout(self):
        operation = self.match(['<<'])
        expression = []

        while operation:
            endl = self.match(['endl'])
            if endl:
                expression.append(KeyWordNode(endl))
            else:
                expression.append(self.parse_formula())

            operation = self.match(['<<'])

        self.require([';'])
        return CoutNode(expression)

    def parse_while(self):
        self.require(['('])
        condition = self.parse_formula()
        self.require([')'])

        self.require(['{'])
        body = self.parse_code()
        self.require(['}'])

        return WhileNode(condition, body)

    def parse_for(self):
        self.require(['('])
        self.match(self.lexer.var_types_tokens)
        begin = self.parse_formula()
        self.require([';'])

        condition = self.parse_formula()
        self.require([';'])

        step = self.parse_formula()
```

```
        self.require([')'])

        self.require(['{'])
        body = self.parse_code()
        self.require(['}'])

        return ForNode(begin, condition, step, body)

    def parse_if_else_condition(self):
        self.require(['('])
        condition = self.parse_formula()
        self.require([')'])

        self.require(['{'])
        body = self.parse_code()
        self.require(['}'])

        if self.match(['else']):
            if self.match(['if']):
                else_condition = self.parse_if_else_condition()
            else:
                self.require(['{'])
                else_condition = self.parse_code()
                self.require(['}'])

            return IfNode(condition, body, else_condition)

        return IfNode(condition, body, None)

    def parse_function_parameters(self, types=False):
        parameters = []

        if self.match([')']):
            self.position -= 1
            return parameters

        if types:
            self.require(self.lexer.var_types_tokens)
        parameters.append(self.require(self.lexer.var_tokens.keys()))

        comma = self.match([','])
        while comma:
            if types:
                self.require(self.lexer.var_types_tokens)
            parameters.append(self.require(self.lexer.var_tokens.keys()))
            comma = self.match([','])

        return parameters

    def parse_function(self, function_token):
        self.require(['('])
```

```
        parameters = self.parse_function_parameters(True)
        self.require([')'])

        self.require(['{'])
        body = self.parse_code()
        self.require(['}'])

        return FunctionNode(function_token, parameters, body)

    def parse_function_call(self, function_token):
        self.require(['('])
        parameters = self.parse_function_parameters()
        self.require([')'])
        self.require([';'])

        return FunctionCallNode(function_token, parameters)

    def parse_switch(self):
        self.require(['('])
        variable = self.require(self.lexer.var_tokens.keys())
        self.require([')'])

        self.require(['{'])
        body = self.parse_code()
        self.require(['}'])

        return SwitchNode(variable, body)

    def parse_case(self):
        constant = self.parse_variable_or_constant()
        self.require([':'])

        return CaseNode(constant.constant)

    def parse_key_word(self, key_word):
        self.require([':'] if key_word.word == 'default' else [';'])
        return KeyWordNode(key_word)

    def parse_ignored_keywords(self, key_word):
        if key_word.word == '#include':
            self.require(libs)
        elif key_word.word == 'using':
            self.require(['namespace'])
            self.require(namespaces)
            self.require([';'])
        return None

    def parse_expression(self) -> Node:
        if self.match(self.lexer.var_tokens.keys()):
            self.position -= 1  # current position is variable
            var_node = self.parse_variable_or_constant()
```

```
            operation = self.match(all_operators)
            if operation:
                # unary and array processing
                right_formula_node = self.parse_formula()
                self.require([';'])
                return BinaryOperationNode(operation, var_node, right_formula_node)

        if self.match(self.lexer.var_types_tokens):
            variable_token = self.match(self.lexer.var_tokens.keys())
            if variable_token:
                return self.parse_variable_definition(variable_token)

            function_token = self.match(self.lexer.func_tokens.keys())
            if function_token:
                return self.parse_function(function_token)

            raise Exception(f'Expected variable or function after {self.get_prev()}')

        function_token = self.match(self.lexer.func_tokens.keys())
        if function_token:
            return self.parse_function_call(function_token)

        key_word = self.match(self.lexer.key_word_tokens)
        if key_word:
            if key_word.word in ignore:
                return self.parse_ignored_keywords(key_word)
            elif key_word.word == 'case':
                return self.parse_case()
            elif key_word.word == 'default':
                return self.parse_key_word(key_word)
            elif key_word.word == 'cin':
                return self.parse_cin()
            elif key_word.word == 'cout':
                return self.parse_cout()
            elif key_word.word == 'for':
                return self.parse_for()
            elif key_word.word == 'if':
                return self.parse_if_else_condition()
            elif key_word.word == 'switch':
                return self.parse_switch()
            elif key_word.word == 'continue':
                return self.parse_key_word(key_word)
            elif key_word.word == 'break':
                return self.parse_key_word(key_word)
            elif key_word.word == 'while':
                return self.parse_while()
    def parse_code(self) -> Node:  # block parse
        root = StatementsNode()
        while self.position < len(self.tokens):
            if self.match(['}']):
```

```
            self.position -= 1
            return root
        code_string_node = self.parse_expression()
        if code_string_node:
            root.add_node(code_string_node)

    return root
```