Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Лабораторная работа №3
Дисциплина «Основы разработки компиляторов»
Вариант 12

Выполнил:
Съестов Дмитрий Вячеславович
Группа P3317

Преподаватель:
Логинов Иван Павлович

Санкт-Петербург
2019

Задание: разработать синтаксический анализатор для второй лабораторной работы.

Синтаксический анализ осуществляется методом parse() класса Parser. Он распознаёт сначала определения переменных, а затем блок Begin…End:

```
fun parse(): AbstractSyntaxTree {
    var defs: Node? = null
    var statements: Node? = null
    getNextToken()
    while (token.type != TokenType.KEYWORD_BEGIN) {
        if (token.type == TokenType.END_OF_INPUT) error("Missing end keyword", token.line, token.pos)
        defs = Node.makeNode(NodeType.SEQUENCE, defs, parseDefinition())
    }
    expect("Begin", TokenType.KEYWORD_BEGIN)
    while (token.type != TokenType.KEYWORD_END) {
        if (token.type == TokenType.END_OF_INPUT) error("Missing end keyword", token.line, token.pos)
        statements = Node.makeNode(NodeType.SEQUENCE, statements, parseStatement())
    }
    expect("End", TokenType.KEYWORD_END)
    if (token.type != TokenType.END_OF_INPUT) error("Excpecting end of input, found: " + token.type,
token.line, token.pos)

    val root = Node.makeNode(NodeType.PROGRAM, defs, statements)
    return AbstractSyntaxTree(root)
}
```

Функция parseDefinition отвечает за анализ объявлений переменных. Каждое отдельное объявление после ключевого слова BIN или INT обрабатывается функцией parseVars:

```
private fun parseDefinition(): Node? {
    var node: Node? = null
    while (true) {
        when (token.type) {
            TokenType.KEYWORD_BEGIN -> return node
            TokenType.KEYWORD_INT, TokenType.KEYWORD_BIN -> {
                val type = if (token.type == TokenType.KEYWORD_INT) NodeType.INT_DEF else NodeType.BIN_DEF
                getNextToken()
                val def = Node.makeNode(type, parseVars())
                node = Node.makeNode(NodeType.SEQUENCE, node, def)
            }
            else -> error("Expecting variable definition, found: " + token.type, token.line, token.pos)
        }
    }
}
```

```
private fun parseVars(): Node? {
    var node: Node? = null
    while (token.type == TokenType.IDENTIFIER) {
        val ident = Node.makeLeaf(NodeType.IDENTIFIER, token.value)
        node = Node.makeNode(NodeType.SEQUENCE, node, ident)
        getNextToken()
        when (token.type) {
            TokenType.COMMA -> getNextToken()
            TokenType.SEMICOLON -> {
                getNextToken()
                return node
            }
            else -> error("Expecting comma or semicolon, found: " + token.type, token.line, token.pos)
        }
    }
```

```
    error("Expecting identifier, found: " + token.type, token.line, token.pos)
    return null
}
```

parseStatement() распознаёт отдельную операцию. В данном языке все операции в блоке Begin…End являются операциями присваивания, которые состоят из идентификатора, оператора присваивания и выражения в правой части. Последнее распознаётся функцией parseExpression():

```
private fun parseStatement(): Node? {
    var node: Node? = null
    if (token.type == TokenType.IDENTIFIER) {
        val ident = Node.makeLeaf(NodeType.IDENTIFIER, token.value)
        getNextToken()
        expect("Assign", TokenType.OP_ASSIGN)
        val expr = parseExpression(0)
        node = Node.makeNode(NodeType.OP_ASSIGN, ident, expr)
        expect("Semicolon", TokenType.SEMICOLON)
    }
    else error("Expecting start of statement, found: " + token.type, token.line, token.pos)
    return node
}
```

```
private fun parseExpression(p: Int): Node? {
    var result: Node? = null
    var node: Node?
    var op: TokenType

    when (token.type) {
        TokenType.L_BRACKET -> result = parseBrackets()
        TokenType.OP_NEGATE -> {
            getNextToken()
            node = parseExpression(TokenType.OP_NEGATE.precedence)
            result = Node.makeNode(NodeType.OP_NEGATE, node)
        }
        TokenType.IDENTIFIER, TokenType.INTEGER, TokenType.BINARY -> {
            val type = when (token.type) {
                TokenType.IDENTIFIER -> NodeType.IDENTIFIER
                TokenType.INTEGER -> NodeType.INTEGER
                else -> NodeType.BINARY
            }
            result = Node.makeLeaf(type, token.value)
            getNextToken()
        }
        else -> error("Expecting a primary, found: " + token.type, token.line, token.pos)
    }

    while (token.type.isBinaryOp && token.type.precedence >= p) {
        op = token.type
        getNextToken()
        node = parseExpression(op.precedence)
        result = Node.makeNode(op.nodeType, result, node)
    }
    return result
}
```

Примеры программ

Корректная программа

```
Int foo, bar, baz;
Bin a, b, c;

Begin  //This is a comment

      foo := 5 * 20 - 25;
      bar := -foo / 5;
      b := 1;

End
```

```
Program
      |Sequence
      |      |Sequence
      |      |      |Sequence
      |      |      |      |Definitions
      |      |      |      |      |Sequence
      |      |      |      |      |      |Sequence
      |      |      |      |      |      |      |Sequence
      |      |      |      |      |      |      |      |Identifier     foo
      |      |      |      |      |      |      |Identifier     bar
      |      |      |      |      |      |Identifier     baz
      |      |      |Statements
      |      |      |      |Sequence
      |      |      |      |      |Sequence
      |      |      |      |      |      |Sequence
      |      |      |      |      |      |      |Identifier     a
      |      |      |      |      |      |Identifier     b
      |      |      |      |      |Identifier     c
      |      |Sequence
      |      |      |Sequence
      |      |      |      |Sequence
      |      |      |      |      |Assign
      |      |      |      |      |      |Identifier     foo
      |      |      |      |      |      |Subtract
      |      |      |      |      |      |      |Multiply
      |      |      |      |      |      |      |      |Integer        5
      |      |      |      |      |      |      |      |Integer        20
      |      |      |      |      |      |      |Integer        25
      |      |      |      |Assign
      |      |      |      |      |Identifier     bar
      |      |      |      |      |Divide
      |      |      |      |      |      |Negate
      |      |      |      |      |      |      |Identifier     foo
      |      |      |      |      |      |Integer        5
      |      |      |Assign
      |      |      |      |Identifier     b
      |      |      |      |Binary
```

Программа с ошибками

```
Int foo, bar, baz;
Bin a> b> c;

Begin
      foo := 5 * 20 - 25;
      bar := {foo / 5
      b = 1;
End
```

```
C:\Users\Dmitry\Desktop\Лабы\compilers\test2.txt
-----------------------------------------------
Failed to create Parser: source contains lexical errors

identifierOrLiteral: unrecognized character: (62) > in line 2, character 6
identifierOrLiteral: unrecognized character: (62) > in line 2, character 9
identifierOrLiteral: unrecognized character: (123) { in line 7, character 8
identifierOrLiteral: unrecognized character: (61) = in line 8, character 3
```

Вывод

В ходе выполнения данной работы был реализован простой синтаксический анализатор, который выводит синтаксическое дерево для потока токенов из предыдущей работы. Синтаксический анализ программ с ошибками не осуществляется.