Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

разработка компилятора подмножества

процедурно-ориентированного языка

Пояснительная записка

RU. 643.02068048.0001-01 81 01

На 11 листах

|  |  |  |
| --- | --- | --- |
| Руководитель |  | к.т.н. доцент кафедры ИЗИ Ю.М. Монахов |
|  |
| Исполнитель |  | студент гр. ИБ-117 Д.Д. Кузин |

**Владимир 2020**

# АННОТАЦИЯ

В данном программном документе приведён текст компилятора подмножества процедурно-ориентированного языка. Компилятор реализован на языке C# с использованием генератора Antlr. Основная функция компилятора – исполнение исходного кода языка и вывод результата выполнения.

Разработка компилятора подмножества процедурного языка состоит из следующих стадий:

1) построение лексического анализатора;

2) построение синтаксического анализатора;

3) построение визитора

Оглавление

[1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА 4](#_Toc41657097)

[1.1 Основные требования 4](#_Toc41657098)

[1.2 Создание лексического анализатора 5](#_Toc41657099)

[1.3 Разработка синтаксического анализатора 6](#_Toc41657100)

[1.4 Построение визитора 7](#_Toc41657101)

# 1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

## Основные требования

Разработка будет производиться в соответствии со следующими требованиями:

* Требования к входному языку:

1. Должны присутствовать операторные скобки;
2. Должна игнорироваться индентация программы;
3. Должны поддерживаться комментарии любой длины;

* Требования к операторам:

1. Оператор присваивания;
2. Арифметические операторы;
3. Логические операторы (И, ИЛИ, НЕ);
4. Условный оператор (ЕСЛИ);
5. Оператор цикла (while, break, continue);
6. Базовый вывод (строковой литерал, переменная);
7. Типы (целочисленный, вещественный).

## 1.2. Создание лексического анализатора

Лексический анализатор является первой фазой работы компилятора. Работа лексического анализатора основана на регулярных выражениях. На вход подаётся последовательность символов, представляющая собой исходный код. На выходе образуется поток токенов или по другому лексем исходного языка.

Грамматика синтаксического и лексического анализатора основана на грамматике языка C#. Лексический и синтаксический анализатор разработаны с помощью Antlr 4 с генерацией исходных файлов под C#.

На рисунке 1 приведен список токенов для описания языка.

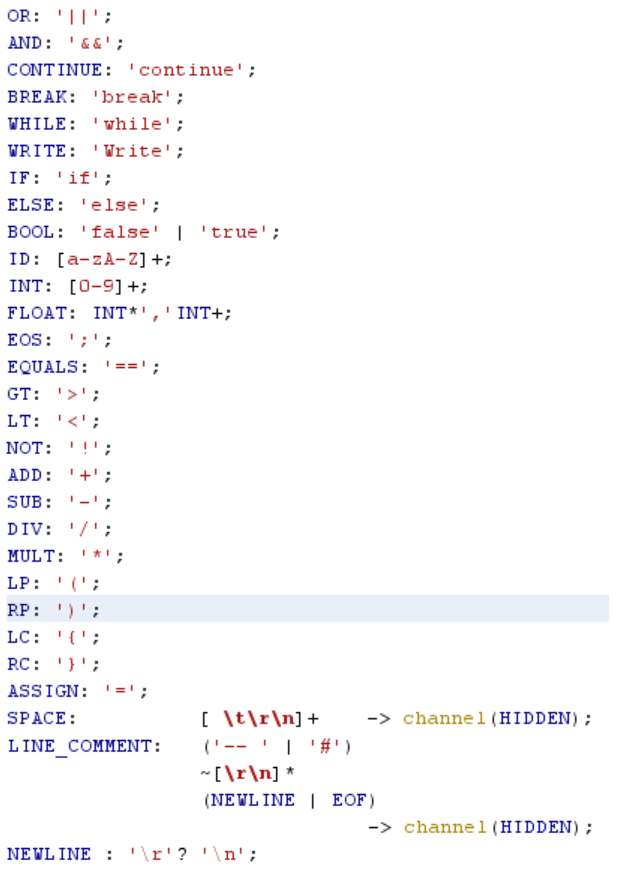


Рисунок 1 - Токены для описания языка

## 1.3. Разработка синтаксического анализатора

Второй частью компилятора является синтаксический анализ. На вход синтаксическому анализатору подаётся набор лексем из лексического анализатора. На основе правил парсера языка (рисунки 2 и 3) строится дерево разбора грамматики. Вся логика работы компилятора описана в классе MyVisitor.

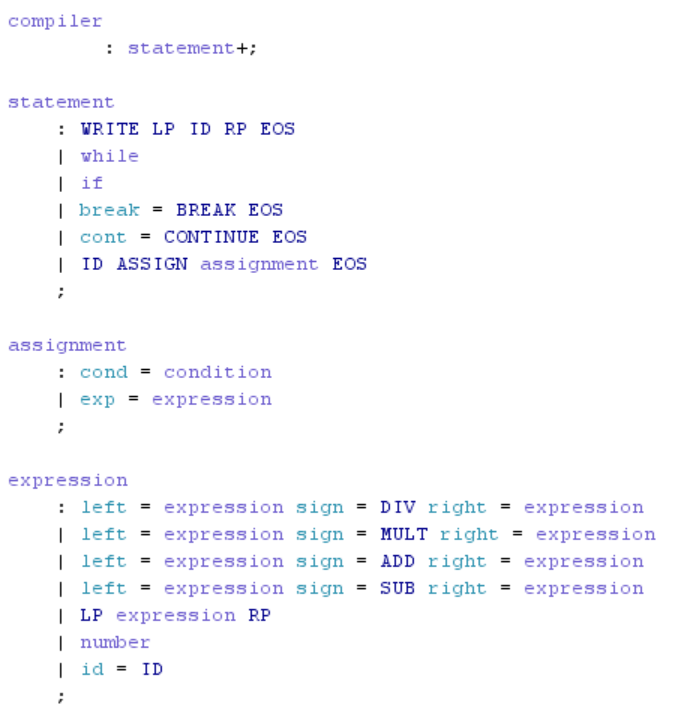


Рисунок 2 – Первая часть правил парсера

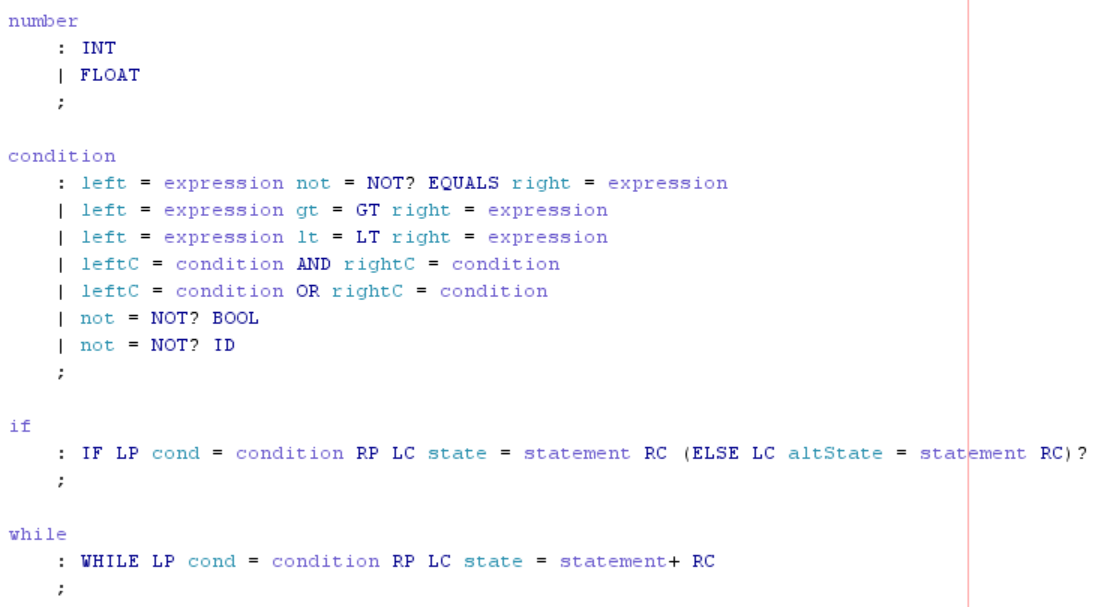


Рисунок 3 – Вторая часть правил парсера

## 1.4. Построение визитора

Корневое правило грамматики Statement вызывается первым при запуске компилятора. Если Statement является командой вывода, то соответствующее значение, переданное в качестве аргумента, выводится в консоль. В остальных случаях визитор начинает посещение дочерних узлов.

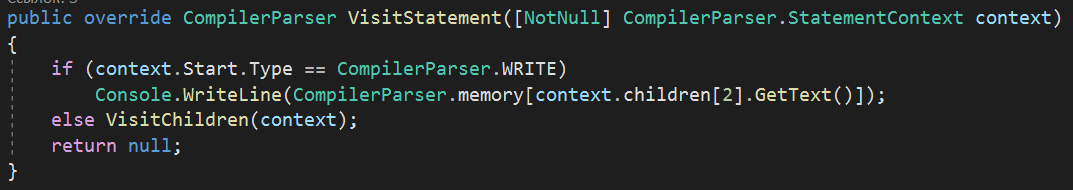


Рисунок 4 – VisitStatement

Правило Expression представляет собой работу с арифметическими выражениями, результатом всегда является число, значение которого вычисляется в методе EvaluateExpression (описано ниже) и записывается в специальное поле в переменной контекста.

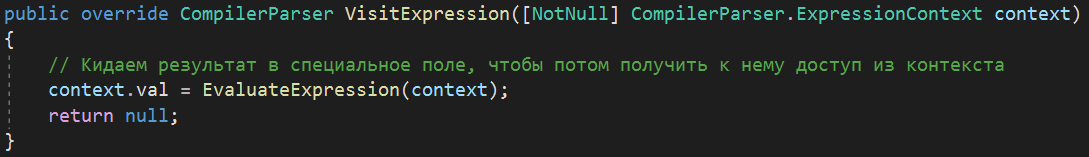


Рисунок 5 – VisitExpression

Метод EvaluateExpression рекурсивно рассчитывает значение переданного выражения на основе его вида. В метод может попасть как сложное составное выражение, так и обычное число. В случае передачи некорректного арифметического знака выбрасывается соответствующее исключение.

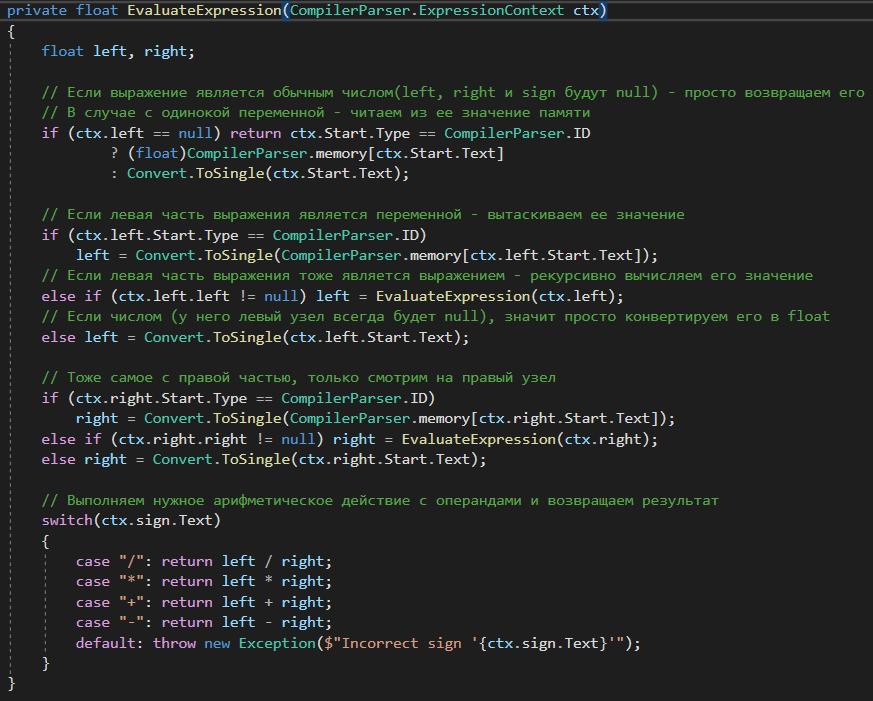


Рисунок 6 – EvaluateExpression

Правило Assignment отвечает за присваивание переменной какого либо значения. Значение может быть числом, составным выражением или другой переменной. Значение переменной сохраняется в специальной хеш таблице, ключом в данном случае является название переменной, а значением – ее значение.

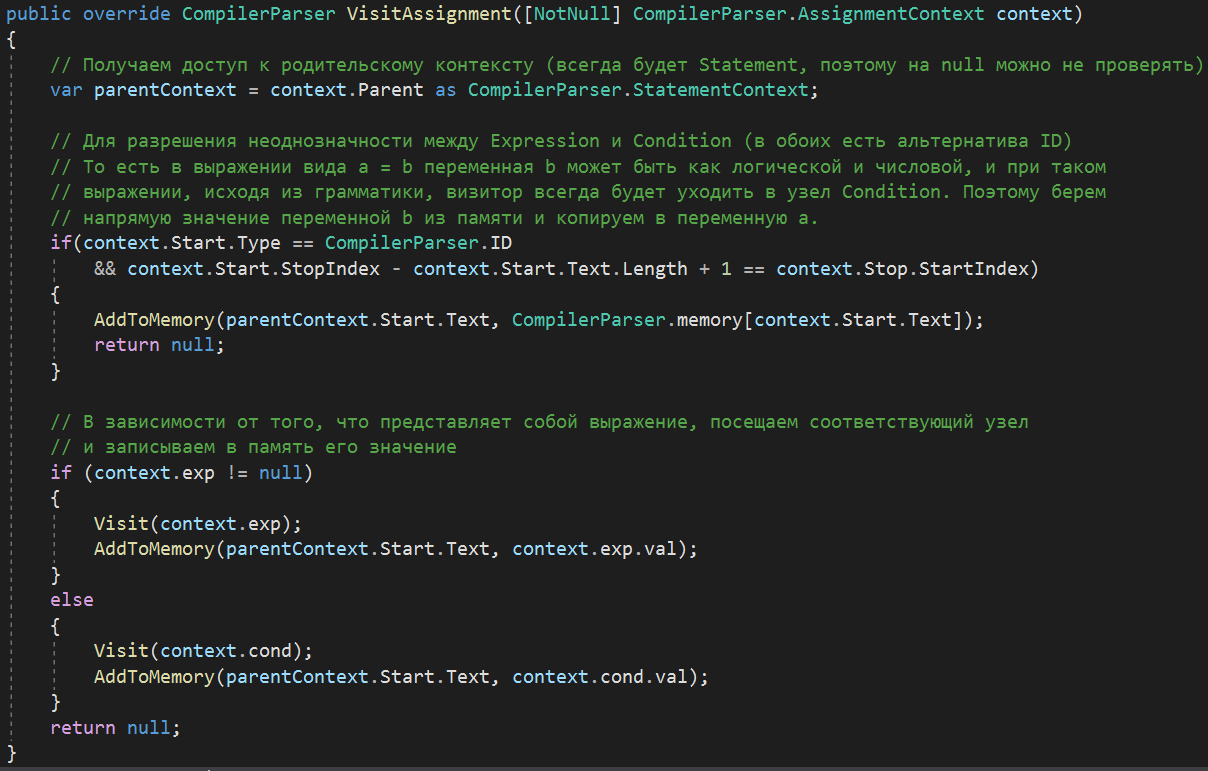


Рисунок 7 – VisitAssignment

Для добавления переменной в память используется вспомогательный метод AddToMemory, который проверяет наличие переменной с указанным названием в памяти. Если переменная уже существует, то заменяется только ее значение, если не существует – создается новая пара ключ-значение.

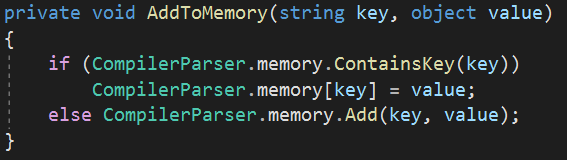


Рисунок 8 – AddToMemory

Для обработки условий и операций с логическими значениями существует правило Condition. Результатом правила всегда является логическое значение, которое записывается в специальное поле в переменную контекста. На вход правила могут быть переданы операции сравнения арифметических выражений, объединение или пересечение (И, ИЛИ) нескольких других условных конструкций, другие логические переменные, инверсия.

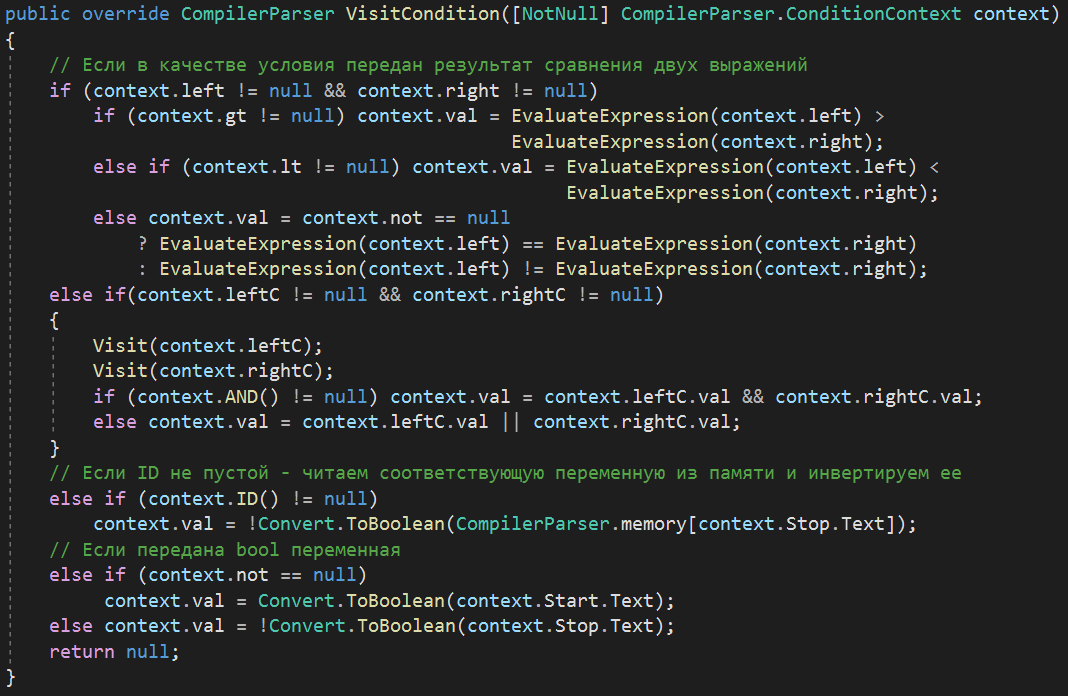


Рисунок 9 – VisitCondition

Правило If служит для ветвления выполнения программы. Если выполняется переданное в переменной контекста условие, то посещается узел с соответствующим блоком кода. В противном случае, при условии, что присутствует альтернативный блок (ветка else), посещается этот блок кода.

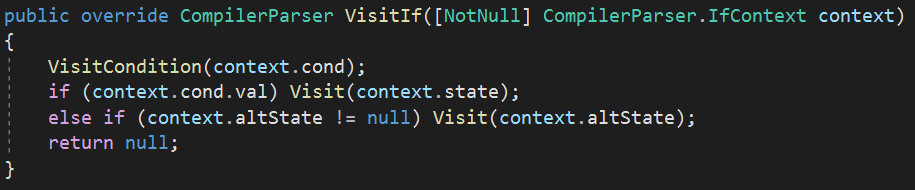


Рисунок 10 – VisitIf

Правило While отвечает за обработку циклов. Сначала проверяется условие, хранящееся в переменной контекста, при его истинности посещаются все дочерние узлы. Если один из узлов является оператором break или continue, то выполняется выход из цикла или переход к следующей итерации соответственно.

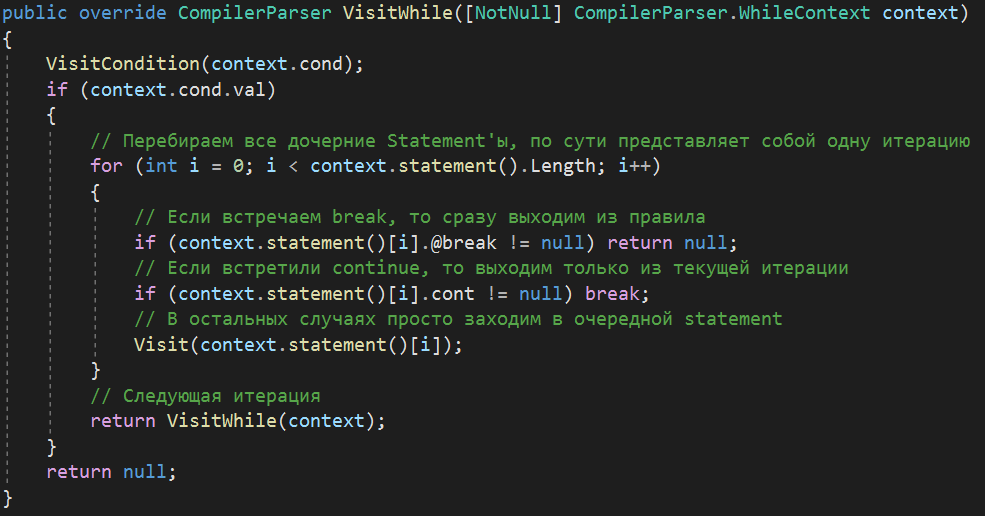


Рисунок 11 - VisitWhile