# Выбор языка программирования

Для реализации компилятора языка Паскаль был выбран язык программирования C#.

# Модуль ввода-вывода

## Проектирование

Модуль ввода-вывода должен считывать исходный текст программы и позволять лексическому анализатору получать литеры по одной, при этом сохраняя информацию о позиции литеры. Также модуль ввода-вывода должен принимать ошибки для дальнейшего составления листинга ошибок. Каждая ошибка содержит позицию ошибки и код, по которому можно получить её описание.

## Реализация

Для реализации модуля был создан класс IOModule, который хранит номер текущей строки и позицию символа в этой строке.

Основной метод класса NextChar() считывает файл построчно и двигается по текущей строке, «передвигая» текущий символ на 1 позицию. Если метод встречает конец файла, он возвращает значение null, сигнализируя, что далее символов нет.

public *char*? NextChar()

    {

        if (current\_line == null)

            return null;

        if (charnumber == current\_line.Length)

        {

            current\_line = reader.ReadLine();

            while (current\_line == "")

                current\_line = reader.ReadLine();

            linenumber++;

            charnumber = 0;

        }

        if (current\_line == null)

            return null;

        current\_char = current\_line[(*int*)charnumber];

        charnumber++;

        return current\_char;

    }

Также в экземпляре класса хранится список ошибок, который можно пополнять с помощью метода void AddError() и печатать с помощью метода PrintErrors, который принимает на вход функцию для печати.

public *void* AddError(*uint* *error\_code*, *Position* *position*)

    {

        errors.Add(new *Error*(*position*, *error\_code*));

    }

    public *void* PrintErrors(*Action*<*string*> *Print*)

    {

        foreach (var error in errors)

*Print*(error.ToString());

    }

## Тестирование

Тестирование проводилось на файлах emptyfile.txt и pascal example.txt. Ошибки добавлялись «вручную».

IOModule iomodule = new IOModule("pascal example.txt");

while (iomodule.NextChar())

{

    Console.WriteLine(iomodule.Current\_char);

    iomodule.AddError(13);

}

iomodule.PrintErrors(Console.WriteLine);

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст, экран, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, электроника

Автоматически созданное описание

# Лексический анализатор

## Проектирование

Лексический анализатор должен выделять последовательность символов (токен) определяя его тип: идентификатор, константа или ключевое слово, и при этом выдавать токены друг за другом по одному. Лексический анализатор должен пропускать комментарии и лишние пробелы, определять ошибки в целых и вещественных константах (например, слишком большое число) и при нахождении запрещённых символов. Ошибки должны подаваться в модуль ввода-вывода.

## Реализация

Лексический анализатор реализован в виде публичного класса LexicalAnalyzer, конструктор которого принимает на вход ссылку на объект класса IOModule, с помощью него производится получение следующего символа.

Основная функция анализатора NextToken() возвращает ссылку на экземпляр одного из классов-подвидов токенов CIdentToken (токен-идентификатор, хранящий имя идентификатора), CKeywordToken (токен-ключевое слово) и CConstToken (токен-константа одного из типов CVariant (int, real, string, boolean) со значением). Каждый из классов наследуется от абстрактного класса CToken с полями типа TokenType (определяет вид токена), Position (позиция символа-начала токена) и KeyWord (значение из перечисления с ключевыми словами, значением для идентификатора и значением для константы).

Значения KeyWord представлены в таблице:

|  |  |
| --- | --- |
| Элемент перечисления | Значение |
| elsesy | else |
| dosy | do |
| ifsy | if |
| programsy | program |
| varsy | var |
| beginsy | begin |
| endsy | end |
| whilesy | while |
| andsy | and |
| orsy | or |
| notsy | not |
| identsy | идентификатор |
| constsy | константа |
| star | \* |
| slash | / |
| equal | = |
| comma | , |
| semicolon | ; |
| colon | : |
| point | . |
| arrow | ^ |
| leftpar | ( |
| rightpar | ) |
| lbracket | [ |
| rbracket | ] |
| later | < |
| greater | > |
| laterequal | <= |
| greaterequal | >= |
| latergreater | <> |
| plus | + |
| minus | - |
| assign | := |
| twopoints | .. |
| integersy | int |
| stringsy | string |
| realsy | real |
| booleansy | boolean |

В методе NextToken() производит пропуск пробелов и комментариев в цикле while (для корректной обработки случая, когда подряд идёт несколько комментариев, либо много пробелов перед и после комментария и т.д.):

// пропуск пробелов и комментариев

        while (ch == ' ' || ch == '{' || ch == '(')

        {

            // пропуск комментариев в { }

            if (ch == '{')

            { … }

            // пропуск блоков комментариев в (\* \*)

            if (ch == '(')

            { … }

            ch = iomodule.NextChar();

            if (!ch.HasValue)

                return null;

        }

Проблема пропуска комментариев в (\*\*) была в том, что существует токен ‘(‘. Если после открывающей скобки нет ‘\*’, то ‘(‘ не является началом комментария, и метод возвращает токен открывающей скобки.

После пропуска метод «распознаёт» токен, ориентируясь на первый значимый (не пробел и не комментарий) символ.

Ключевые слова, идентификаторы и булевые значения начинаются с символов-букв, поэтому при нахождении буквы лексический анализатор считывает последовательность букв, цифр и нижних подчёркиваний до конца. Для определения ключевых слов используется словарь ключевых слов. Если текущий токен найден в таблице ключевых слов, то анализатор возвращает токен-ключевое слово, если текущий токен является значением ‘true’ или ‘false’, то возвращается токен с булевым значением. Во всех остальных случаях возвращается токен идентификатора.

Цифра в начале токена означает, что далее идёт целая или вещественная константа. Анализатор считывает число до конца и в случае неправильных чисел возвращает коды ошибок в модуль ввода-вывода (207 в случае вещественной константы и 203 в случае целой константы). При ошибке анализатор возвращает токен-константу с 0 значением, без ошибок значение будет равно считанному числу.

Ряд токенов имеет один символ в начале, поэтому при нахождении ряда символов производится запрос следующего символа у модуля ввода-вывода и уже точное определение токена. Пример: при нахождении символа ‘<’ анализатор запрашивает следующий символ. В случае нахождения ‘=’ анализатор возвращает токен ключевого слова «меньше или равно», при нахождении ‘>’ — «не равно», а если далее найден другой символ, возвращается токен ключевого слова «меньше». Аналогично обрабатываются остальные токены с одинаковым началом.

При обнаружении символов, с которых начинается односимвольный токен, возвращается токен соответсвенного ключевого слова.

При нахождении постороннего символа анализатор добавляет в модуль ввода-вывода ошибку с кодом 6 и «выбрасывает» FatalException, наследуемый от стандартного класса Exception.

## Тестирование

Тестирование производилось вручную с помощью файлов error1.txt и error2.txt. Лексический анализатор корректно определяет токены и добавляет ошибки в модуль ввода-вывода, а также пропускает лишние пробелы и комментарии.

Изображение выглядит как текст

Автоматически созданное описание

Вывод лексического анализатора:

programsy

Printing

semicolon

0

varsy

i

colon

integersy

semicolon

beginsy

i

assign

i

star

3

ifsy

leftpar

a

equal

b

rightpar

orsy

leftpar

a

greaterequal

c

rightpar

andsy

leftpar

a

laterequal

b

rightpar

andsy

notsy

leftpar

a

latergreater

c

rightpar

then

beginsy

c

assign

1

slash

1

semicolon

с

assign

True

semicolon

endsy

elsesy

c

assign

0

star

0

semicolon

endsy

point

Unhandled exception. FatalException: Exception of type 'FatalException' was thrown.

at LexicalAnalyzer.NextToken() in \khaldina-pmi-3-compiler\LexicalAnalyzer.cs:line 244

at Program.<Main>$(String[] args) \khaldina-pmi-3-compiler\Program.cs:line 7