# 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

# Синтаксический анализатор

## Проектирование

Синтаксический анализатор должен проверять, соответствует ли программа формальным языковым правилам, и выводить синтаксические ошибки в случае их наличия.

От лексического анализатора и модуля ввода-вывода семантический анализатор получает тип текущего токена и его положение в тексте.

Тип токена необходим для анализа правил Паскаля, которые удобно представлены в формах Бэкуса-Наура. Необходимые БНФ взяты в учебнике Залоговой Л.А.

Для каждого правила можно описать функцию, которая будет вызывать функции других правил при встрече нетерминального символа в правиле. Все терминальные символы будут проверены на совпадение с ожидаемыми и в случае ошибки будут сообщать о ней модулю ввода-вывода.

## Реализация

Синтаксический анализатор представляет собой публичный класс Analyzer. Для корректной работы ему необходим модуль ввода-вывода и лексический анализатор.

От лексического анализатора синтаксический получает следующий токен.

Так как файл программы на паскале может быть некорректным или пустым, для обработки случая непредвиденного конца файла класс имеет функцию bool EOF(), которая проверяет, является ли следующий токен пустым, и если это так, сообщает об ошибке в модуль ввода-вывода, после чего возвращает true. С помощью этой функции производится проверка на конец файла, чтобы избежать обращения к пустому указателю на следующий токен.

Функция void Accept() проверяет текущий символ на совпадение с ожидаемым и в случае несовпадения сообщает модулю ввода-вывода об ошибке. Так как в вызывающей функции может быть несколько ожидаемых символов, Accept принимает на вход некоторое количество значений типа KeyWord. Для добавления ошибок в модуль ввода-вывода необходимы коды ошибок, которые хранятся в значениях перечисления KeyWord, например, ключевое слово beginsy (begin) имеет значение 17, что соответствует ошибке «должно идти слово BEGIN». Список ошибок взят из учебника Залоговой Л.А.

Для каждой необходимой БНФ статический анализатор имеет функцию, которая проверяет эту форму. Все функции в сопоставлении с БНФ представлены в таблице:

|  |  |
| --- | --- |
| Функция | БНФ |
| Program | <программа>::=program <имя>(<имя файла>{,<имя файла>}); <блок>. |
| Block | <блок>: :=<раздел меток><раздел констант><раздел типов><раздел переменных><раздел процедур и функций><раздел операторов> |
| VariableDeclarationPart | <раздел переменных> : = var <описание однотипных переменных>; {<описание однотипных переменных>;} | <пусто> |
| VariableDeclaration | <описание однотипных переменных>::=<имя>{,<имя>} : <тип> |
| Type | <тип>::=integer | real | boolean | string (необходимо было реализовать только эти типы) |
| StatementPart | <составной оператор>: := begin <оператор>{; <оператор>} end |
| Statement | <оператор>::= <простой оператор>|<сложный оператор>  <простой оператор>::=<оператор присваивания>|<оператор процедуры>|<оператор перехода>|<пустой оператор>  <сложный оператор>::=<составной оператор>|<выбирающий оператор>|<оператор цикла>|<оператор присоединения> |
| Assignment | <оператор присваивания>::=<переменная>:=<выражение>|<имя функции>:=<выражение> |
| Expression | <выражение>::=<простое выражение>|<простое выражение><операция отношения><простое выражение> |
| SimpleExpression | <простое выражение> ::=<знак><слагаемое>{<аддитивная операция><слагаемое>} |
| RelationalOperator | <операция отношения>::==|<>|<|<=|>=|>|in |
| AddingOperator | <аддитивная операция>::= + | - | or |
| MultiplyingOperator | <мультипликативная операция>::=\*|/|div|mod|and |
| Term | <слагаемое>::=<множитель>{<мультипликативная операция><множитель>} |
| Factor | <множитель>::=<переменная>|<константа без знака>|(<выражение>)|<обозначение функции>|<множество>|not <множитель>  <константа без знака>::=<число без знака>|<строка>|<имя константы>|nil |
| IfStatement | <условный оператор>::= if <выражение> then <оператор>|if <выражение> then <оператор> else <оператор> |
| WhileStatement | <цикл с предусловием>::= while <выражение> do <оператор> |

Функция Program идёт по БНФ <программа>: делает Accept токенов program, идентификатора и точки с запятой, затем вызывает функцию, которая реализует БНФ <блок> и делает Accept точки.

Функция Block пропускает раздел меток, констант и процедур и функций, и вызывает функции, которые реализуют БНФ раздела переменных и раздела операторов.

В функции раздела переменных VariableDeclarationPart происходит Accept ключевого слова var, вызывается функция БНФ описания однотипных переменных и делается Accept точки с запятой. Если описание однотипных переменных не одно, функция в цикле повторяет действия до тех пор, пока ей встречается идентификатор после Accept точки с запятой. Если ключевого слова var в начале нет, ничего не происходит и функция считает, что раздел переменных пуст.

Функция однотипных переменных VariableDeclaration производит Accept идентификаторов и запятых, пока они не кончатся, затем делает Accept двоеточия и вызывает функцию БНФ типа.

Тип в функции Type определяется как integersy, stringsy, realsy и booleansy. По текущему токену с помощью оператора выбора находится нужный код и производится Accept.

StatementPart реализует БНФ составного оператора: Accept begin, затем вызов функции оператора, пока за ним есть точка с запятой, и Accept end.

Statement по первому токену определяет тип оператора и вызывает соответствующую функцию: в случае наличия идентификатора вызывает функцию оператора присваивания, в случае слова begin вызывает функцию поставного оператора, при if функцию, обрабатывающую БНФ условного оператора и в случае while функцию с циклом с предусловием.

Оператор присваивания Assignment делает Accept идентификатора и токена присваивания, а затем вызывает БНФ выражения.

Выражение Expression вызывает функцию простого выражения, а затем проверяет, является ли текущий токен одним из токенов операции отношения, которые перечислены в статическом поле List<KeyWord> relationalOperators. Если текущий токен является одним из них, вызывается функция операций отношения и функция простого выражения.

Функция RelationalOperator делает Accept relationalOperators.

При реализации функции простого выражения SimpleExpression возникла проблема, связанная с БНФ в учебнике Залоговой Л.А., судя по которым, ЛЮБОЕ выражение должно начинаться с унарного знака (что неверно). Было принято решение сделать унарный знак опциональным. При его наличии производится Accept знака. После этого функция вызывает БНФ слагаемого Term, а затем при наличии следующего токена в множестве аддитивных операций addingOperators вызывает функцию AddingOperator, которая производит Accept этого токена, и снова обращается к функции слагаемого.

Функция слагаемого работает аналогично функции простого выражения: она вызывает функцию множителя, а потом проверяет, является ли следующий токен одной из мультипликативных операций multiplyingOperators, и до тех пор, пока это так, она вызывает сначала функцию БНФ мультипликативной операции MultiplyingOperator (производящую Accept токена), а потом функцию множителя.

Множитель может начинаться с идентификатора или константы без знака (то есть целой, вещественной, строковой и булевой константы), поэтому при встрече такого токена функция делает Accept текущего токена. Если текущий токен является открывающей скобкой, функция делает её Accept, вызывает выражение (так как в скобках может быть выражение) и делает Accept закрывающей скобки. В случае встречи токена not она делает Accept и вызывает саму себя, так как получается «не множиитель».

Функция условного оператора производит Accept ключевого слова if, затем вызывает функцию выражения, делает Accept ключевого слова then и вызывает разбор оператора. Если далее встречается ветка else, она делает Accept этого токена, а затем вызывает функцию оператора.

Цикл с предусловием работает аналогично: производит Accept while, вызывает разбор выражения, делает Accept слова do и разбирает оператор.

## Тестирование

На текущем этапе анализатор, проверяющий только синтаксис без нейтрализации ошибок, должен корректно «проходиться» по правильной с точки зрения синтаксиса программе и выдавать ошибку в случае встречи первого некорректного токена.

Тестирование проводилось на файлах error1.txt, error2.txt и emptyfile.txt. error2.txt, несмотря на своё название (хихи), содержит корректную с точки зрения синтаксиса программу, и анализатор корректно проходит по ней, разбирая синтаксис (это было проверено в режиме отладки).

# Нейтрализация синтаксических ошибок

## Проектирование

## Реализация

## Тестирование