Министерство образования Республики Беларусь

Учреждение образования «Белорусский Государственный Университет Информатики и Радиоэлектроники»

Кафедра «Информатика»

Лабораторная работа № 1-2

«Лексический и синтаксический анализаторы»

по курсу «Теория трансляции»

**Выполнил:**

Волчецкий А.М. гр. 253504

**Проверила:**

Шиманский В.В.

Минск 2015

**Задание**

Написать лексический и синтаксический анализаторы для ограниченного множества операторов языка C#.

Теоретические сведения

Поскольку лексический анализатор является частью компилятора, которая читает исходный текст, он может заодно выполнять и пробельных символов (пробел, символы табуляции и новой строки, а также, возможно, некоторые другие символы, использующиеся для отделения токенов друг от друга во входном потоке). Еще одной задачей является синхронизация сообщений об ошибках, генерируемых компилятором, с исходной программой. Например, лексический анализатор может отслеживать количество символов новой строки, чтобы каждое сообщение об ошибке сопровождалось номером строки, в которой она обнаружена. В некоторых компиляторах лексический анализатор создает копию исходной программы с сообщениями об ошибках, вставленными в соответствующие места исходного текста. Если исходная программа использует макропрепроцессор, то раскрытие макросов также может выполняться лексическим анализатором.

Поскольку лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Обычно при работе лексический анализатор взаимодействует также с таблицей символов. Когда лексический анализатор встречается с лексемой, составляющей идентификатор, эту лексему требуется внести в таблицу символов. В некоторых случаях лексический анализатор может получать из таблицы символов некоторую информацию об идентификаторах, которая может помочь ему верно определить передаваемый синтаксическому анализатору токен.

Иногда лексические анализаторы разделяются на соединенные каскадом два процесса

* Сканирование состоит из простых процессов, не требующих токенизации входного потока, таких как удаление комментариев и уплотнение последовательностей пробельных символов в один.
* Собственно лексический анализ — более сложная часть, которая генерирует токены из выходного потока сканера.

Имеется ряд причин, по которым фаза анализа компиляции разделяется на лексический и синтаксический анализ.

1. Наиболее важной причиной является упрощение разработки. Отделение лексического анализатора от синтаксического часто позволяет упростить как минимум одну из фаз анализа. Например, включить в синтаксический анализатор работу с комментариями и пробельными символами существенно сложнее, чем удалить их лексическим анализатором. При создании нового языка разделение лексических и синтаксических правил может привести к более четкому и ясному построению языка.
2. Увеличивается эффективность компилятора. Отдельный лексический анализатор позволяет применять более специализированные методики, предназначенные исключительно для решения лексических задач. Кроме того, применение методов буферизации для чтения входного потока может существенно увеличить производительность компилятора.
3. Увеличивается переносимость компилятора. Особенности входных устройств могут ограничивать возможности лексического анализатора.

При рассмотрении лексического анализа используются три связанных, но различных термина.

• Токен представляет собой пару, состоящую из имени токена и необязательного атрибута. Имя токена — это абстрактный символ, представляющий тип лексической единицы, например конкретное ключевое слово или последовательность входных символов, составляющую идентификатор. Имена токенов являются входными символами, обрабатываемыми синтаксическим анализатором.

• Шаблон (pattern) — это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой просто последовательность символов, образующих это ключевое слово. Для идентификаторов и некоторых других токенов шаблон представляет собой более сложную структуру, которой соответствуют (matched) многие строки.

• Лексема представляет собой последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Без помощи других компонентов компилятора лексическому анализатору сложно обнаружить ошибки в исходном тексте программы. Например, если в программе на языке С# строка fi впервые встретится в контексте

fi ( а == f(x)) ...

лексический анализатор не сможет определить, что именно представляет собой fi — неверно записанное слово if или необъявленный идентификатор функции.

Поскольку fi является корректной лексемой для токена id, лексический анализатор должен вернуть этот токен синтаксическому анализатору и позволить другой фазе компилятора — в данном случае, по всей видимости, синтаксическому анализатору — обработать ошибку перестановки местами двух букв.

Однако представим ситуацию, когда лексический анализатор не способен продолжать работу, поскольку ни один из шаблонов не соответствует префиксу оставшегося входного потока. Простейшей стратегией в этой ситуации будет восстановление в "режиме паники". Мы просто удаляем входные символы до тех пор, пока лексический анализатор не встретит распознаваемый токен в начале оставшейся входной строки. Этот метод восстановления может запутать синтаксический анализатор, но для интерактивной среды данная методика может быть вполне адекватной.

Существуют и другие возможные действия по восстановлению после ошибки.

1. Удаление одного символа из оставшегося входного потока.
2. Вставка пропущенного символа в оставшийся входной поток.
3. Замена символа другим.
4. Перестановка двух соседних символов.

В нашей модели компилятора синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка имен токенов порождаться грамматикой исходного языка. Мы также ожидаем от синтаксического анализатора сообщений обо всех выявленных ошибках, причем достаточно внятных и полных, а кроме того, умения обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. Концептуально в случае корректной программы синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки. В действительности явное построение дерева разбора не требуется, поскольку проверки и действия трансляции могут выполняться в процессе синтаксического анализа. Таким образом, синтаксический анализатор и прочие части начальной стадии компилятора могут быть реализованы в виде единого модуля.

Имеется три основных типа синтаксических анализаторов грамматик: универсальные, восходящие и нисходящие. Универсальные методы разбора, такие как алгоритмы Кока-Янгера-Касами (Cocke-Younger-Kasami) и Эрли (Earley), могут работать с любой грамматикой. Однако эти обобщенные методы слишком неэффективны для использования в промышленных компиляторах. Методы, обычно применяемые в компиляторах, можно классифицировать как нисходящие (сверху вниз — top-down) или восходящие (снизу вверх — bottom- up). Как явствует из названий, нисходящие синтаксические анализаторы строят дерево разбора сверху (от корня) вниз (к листьям), тогда как восходящие начинают с листьев и идут к корню. В обоих случаях входной поток синтаксического анализатора сканируется посимвольно слева направо. Наиболее эффективные нисходящие и восходящие методы работают только с подклассами грамматик, однако некоторые из этих классов, такие как LL- и LR- грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования. Реализованные вручную синтаксические анализаторы чаще работают с LL-грамматикамию Синтаксические анализаторы для большего класса LR-грамматик обычно создаются с помощью автоматизированных инструментов.

Если компилятор будет иметь дело исключительно с корректными программами, его разработка и реализация существенно упрощаются. Однако от компилятора ожидается, что он будет помогать программисту обнаруживать и устранять ошибки, которые неизбежно содержатся в программах несмотря на огромные усилия даже самых квалифицированных программистов. Примечательно, что хотя ошибки — явление чрезвычайно распространенное, лишь в нескольких языках вопрос обработки ошибок рассматривался еще на фазе проектирования языка. Наша цивилизация была бы совсем другой, если бы в естественных языках были такие же требования к синтаксической точности, как и в языках программирования. Большинство спецификаций языков программирования, тем не менее, не определяет реакции компилятора на ошибки — этот вопрос отдается на откуп разработчикам компилятора. Однако планирование системы обработки ошибок с самого начала работы над компилятором может как упростить его структуру, так и улучшить его реакцию на ошибки. Ошибки в программе могут быть на самых разных уровнях.

* + Лексические ошибки включают неверно записанные идентификаторы, ключевые слова или операторы, например использование идентификатора elipseSize вместо ellipseSize или отсутствие кавычек вокруг текста, являющегося строкой.
  + Синтаксические ошибки включают неверно поставленные точки с запятой или лишние или недостающие фигурные скобки { или }. В качестве еще одного примера в С или Java синтаксической ошибкой является конструкция case без охватывающего switch (однако эта ситуация часто пропускается синтаксическим анализатором и перехватывается позже, на уровне генерации кода).
  + Семантические ошибки включают несоответствие типов операторов и их операндов. В качестве примера можно привести оператор return со значением в методе Java, возвращающем тип void.
  + Логические ошибки могут быть любыми — от неверных решений программиста до использования в программе на языке С оператора присваивания = вместо оператора сравнения ==. Программа, содержащая оператор =, может быть корректной, но делать совсем не то, чего хотел от нее программист.

Точность современных методов разбора позволяет очень эффективно выявлять синтаксические ошибки в программе. Некоторые методы синтаксического анализа, такие как LL и LR, обнаруживают ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен. Говоря более точно, они обладают свойством орректного префикса (viable-prefix property), т.е. они обнаруживают ошибку, как только встречают префикс, который не может быть префиксом ни одной корректной строки данного языка. Еще одна причина, по которой делается упор на восстановление после ошибки в процессе синтаксического анализа, — многие ошибки, чем бы они ни были вызваны, оказываются синтаксическими и выявляются при дальнейшей невозможности синтаксического анализа. Эффективно обнаруживаются и некоторые семантические ошибки, такие как несоответствие типов. Однако в общем случае точное определение семантических и логических ошибок во время компиляции является крайне трудной задачей.

Обработчик ошибок синтаксического анализатора имеет очень просто формулируемые, но очень сложно реализуемые цели:

* + он должен ясно и точно сообщать о наличии ошибок;
  + он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжить поиск других ошибок;
  + он не должен существенно замедлять обработку корректной программы.

К счастью, обычные ошибки достаточно просты, и для их обработки часто достаточно относительно простых механизмов обработки ошибок.

Каким образом обработчик ошибок должен сообщить об обнаруженных неприятностях? Самое меньшее, что он должен сделать, — указать место в исходной программе, где выявлена ошибка, поскольку, скорее всего, реальная ошибка находится несколькими токенами ранее. Распространенная стратегия заключается в том, чтобы вывести проблемную строку с указателем позиции, в которой обнаружена ошибка.

**Код программы**

using System;

using System.Collections.Generic;

using System.Globalization;

using System.Linq.Expressions;

using System.Text;

using Irony.Ast;

namespace Irony.Parsing {

public class Grammar {

#region properties

/// <summary>

/// Gets case sensitivity of the grammar. Read-only, true by default.

/// Can be set to false only through a parameter to grammar constructor.

/// </summary>

public readonly bool CaseSensitive;

//List of chars that unambigously identify the start of new token.

//used in scanner error recovery, and in quick parse path in NumberLiterals, Identifiers

[Obsolete("Use IsWhitespaceOrDelimiter() method instead.")]

public string Delimiters = null;

[Obsolete("Override Grammar.SkipWhitespace method instead.")]

// Not used anymore

public string WhitespaceChars = " \t\r\n\v";

public LanguageFlags LanguageFlags = LanguageFlags.Default;

public TermReportGroupList TermReportGroups = new TermReportGroupList();

//Terminals not present in grammar expressions and not reachable from the Root

// (Comment terminal is usually one of them)

// Tokens produced by these terminals will be ignored by parser input.

public readonly TerminalSet NonGrammarTerminals = new TerminalSet();

/// <summary>

/// The main root entry for the grammar.

/// </summary>

public NonTerminal Root;

/// <summary>

/// Alternative roots for parsing code snippets.

/// </summary>

public NonTerminalSet SnippetRoots = new NonTerminalSet();

public string GrammarComments; //shown in Grammar info tab

public CultureInfo DefaultCulture = CultureInfo.InvariantCulture;

//Console-related properties, initialized in grammar constructor

public string ConsoleTitle;

public string ConsoleGreeting;

public string ConsolePrompt; //default prompt

public string ConsolePromptMoreInput; //prompt to show when more input is expected

#endregion

#region constructors

public Grammar() : this(true) { } //case sensitive by default

public Grammar(bool caseSensitive) {

\_currentGrammar = this;

this.CaseSensitive = caseSensitive;

bool ignoreCase = !this.CaseSensitive;

var stringComparer = StringComparer.Create(System.Globalization.CultureInfo.InvariantCulture, ignoreCase);

KeyTerms = new KeyTermTable(stringComparer);

//Initialize console attributes

ConsoleTitle = Resources.MsgDefaultConsoleTitle;

ConsoleGreeting = string.Format(Resources.MsgDefaultConsoleGreeting, this.GetType().Name);

ConsolePrompt = ">";

ConsolePromptMoreInput = ".";

}

#endregion

#region Reserved words handling

//Reserved words handling

public void MarkReservedWords(params string[] reservedWords) {

foreach (var word in reservedWords) {

var wdTerm = ToTerm(word);

wdTerm.SetFlag(TermFlags.IsReservedWord);

}

}

#endregion

#region Register/Mark methods

public void RegisterOperators(int precedence, params string[] opSymbols) {

RegisterOperators(precedence, Associativity.Left, opSymbols);

}

public void RegisterOperators(int precedence, Associativity associativity, params string[] opSymbols) {

foreach (string op in opSymbols) {

KeyTerm opSymbol = ToTerm(op);

opSymbol.SetFlag(TermFlags.IsOperator);

opSymbol.Precedence = precedence;

opSymbol.Associativity = associativity;

}

}//method

public void RegisterOperators(int precedence, params BnfTerm[] opTerms) {

RegisterOperators(precedence, Associativity.Left, opTerms);

}

public void RegisterOperators(int precedence, Associativity associativity, params BnfTerm[] opTerms) {

foreach (var term in opTerms) {

term.SetFlag(TermFlags.IsOperator);

term.Precedence = precedence;

term.Associativity = associativity;

}

}

public void RegisterBracePair(string openBrace, string closeBrace) {

KeyTerm openS = ToTerm(openBrace);

KeyTerm closeS = ToTerm(closeBrace);

openS.SetFlag(TermFlags.IsOpenBrace);

openS.IsPairFor = closeS;

closeS.SetFlag(TermFlags.IsCloseBrace);

closeS.IsPairFor = openS;

}

public void MarkPunctuation(params string[] symbols) {

foreach (string symbol in symbols) {

KeyTerm term = ToTerm(symbol);

term.SetFlag(TermFlags.IsPunctuation|TermFlags.NoAstNode);

}

}

public void MarkPunctuation(params BnfTerm[] terms) {

foreach (BnfTerm term in terms)

term.SetFlag(TermFlags.IsPunctuation|TermFlags.NoAstNode);

}

public void MarkTransient(params NonTerminal[] nonTerminals) {

foreach (NonTerminal nt in nonTerminals)

nt.Flags |= TermFlags.IsTransient | TermFlags.NoAstNode;

}

//MemberSelect are symbols invoking member list dropdowns in editor; for ex: . (dot), ::

public void MarkMemberSelect(params string[] symbols) {

foreach (var symbol in symbols)

ToTerm(symbol).SetFlag(TermFlags.IsMemberSelect);

}

//Sets IsNotReported flag on terminals. As a result the terminal wouldn't appear in expected terminal list

// in syntax error messages

public void MarkNotReported(params BnfTerm[] terms) {

foreach (var term in terms)

term.SetFlag(TermFlags.IsNotReported);

}

public void MarkNotReported(params string[] symbols) {

foreach (var symbol in symbols)

ToTerm(symbol).SetFlag(TermFlags.IsNotReported);

}

#endregion

#region virtual methods: CreateTokenFilters, TryMatch

public virtual void CreateTokenFilters(LanguageData language, TokenFilterList filters) {

}

//This method is called if Scanner fails to produce a token; it offers custom method a chance to produce the token

public virtual Token TryMatch(ParsingContext context, ISourceStream source) {

return null;

}

//Gives a way to customize parse tree nodes captions in the tree view.

public virtual string GetParseNodeCaption(ParseTreeNode node) {

if (node.IsError)

return node.Term.Name + " (Syntax error)";

if (node.Token != null)

return node.Token.ToString();

if(node.Term == null) //special case for initial node pushed into the stack at parser start

return (node.State != null ? string.Empty : "(State " + node.State.Name + ")"); // Resources.LabelInitialState;

var ntTerm = node.Term as NonTerminal;

if(ntTerm != null && !string.IsNullOrEmpty(ntTerm.NodeCaptionTemplate))

return ntTerm.GetNodeCaption(node);

return node.Term.Name;

}

/// <summary>

/// Override this method to help scanner select a terminal to create token when there are more than one candidates

/// for an input char. context.CurrentTerminals contains candidate terminals; leave a single terminal in this list

/// as the one to use.

/// </summary>

public virtual void OnScannerSelectTerminal(ParsingContext context) { }

/// <summary>Skips whitespace characters in the input stream. </summary>

/// <remarks>Override this method if your language has non-standard whitespace characters.</remarks>

/// <param name="source">Source stream.</param>

public virtual void SkipWhitespace(ISourceStream source) {

while (!source.EOF()) {

switch (source.PreviewChar) {

case ' ': case '\t':

break;

case '\r': case '\n': case '\v':

if (UsesNewLine) return; //do not treat as whitespace if language is line-based

break;

default:

return;

}//switch

source.PreviewPosition++;

}//while

}//method

/// <summary>Returns true if a character is whitespace or delimiter. Used in quick-scanning versions of some terminals. </summary>

/// <param name="ch">The character to check.</param>

/// <returns>True if a character is whitespace or delimiter; otherwise, false.</returns>

/// <remarks>Does not have to be completely accurate, should recognize most common characters that are special chars by themselves

/// and may never be part of other multi-character tokens. </remarks>

public virtual bool IsWhitespaceOrDelimiter(char ch) {

switch (ch) {

case ' ': case '\t': case '\r': case '\n': case '\v': //whitespaces

case '(': case ')': case ',': case ';': case '[': case ']': case '{': case '}':

case (char)0: //EOF

return true;

default:

return false;

}//switch

}//method

//The method is called after GrammarData is constructed

public virtual void OnGrammarDataConstructed(LanguageData language) {

}

public virtual void OnLanguageDataConstructed(LanguageData language) {

}

//Constructs the error message in situation when parser has no available action for current input.

// override this method if you want to change this message

public virtual string ConstructParserErrorMessage(ParsingContext context, StringSet expectedTerms) {

if(expectedTerms.Count > 0)

return string.Format(Resources.ErrSyntaxErrorExpected, expectedTerms.ToString(", "));

else

return Resources.ErrParserUnexpectedInput;

}

// Override this method to perform custom error processing

public virtual void ReportParseError(ParsingContext context) {

string error = null;

if (context.CurrentParserInput.Term == this.SyntaxError)

error = context.CurrentParserInput.Token.Value as string; //scanner error

else if (context.CurrentParserInput.Term == this.Indent)

error = Resources.ErrUnexpIndent;

else if (context.CurrentParserInput.Term == this.Eof && context.OpenBraces.Count > 0) {

if (context.OpenBraces.Count > 0) {

//report unclosed braces/parenthesis

var openBrace = context.OpenBraces.Peek();

error = string.Format(Resources.ErrNoClosingBrace, openBrace.Text);

} else

error = Resources.ErrUnexpEof;

}else {

var expectedTerms = context.GetExpectedTermSet();

error = ConstructParserErrorMessage(context, expectedTerms);

}

context.AddParserError(error);

}//method

#endregion

#region MakePlusRule, MakeStarRule methods

public BnfExpression MakePlusRule(NonTerminal listNonTerminal, BnfTerm listMember) {

return MakeListRule(listNonTerminal, null, listMember);

}

public BnfExpression MakePlusRule(NonTerminal listNonTerminal, BnfTerm delimiter, BnfTerm listMember) {

return MakeListRule(listNonTerminal, delimiter, listMember);

}

public BnfExpression MakeStarRule(NonTerminal listNonTerminal, BnfTerm listMember) {

return MakeListRule(listNonTerminal, null, listMember, TermListOptions.StarList);

}

public BnfExpression MakeStarRule(NonTerminal listNonTerminal, BnfTerm delimiter, BnfTerm listMember) {

return MakeListRule(listNonTerminal, delimiter, listMember, TermListOptions.StarList);

}

protected BnfExpression MakeListRule(NonTerminal list, BnfTerm delimiter, BnfTerm listMember, TermListOptions options = TermListOptions.PlusList) {

//If it is a star-list (allows empty), then we first build plus-list

var isPlusList = !options.IsSet(TermListOptions.AllowEmpty);

var allowTrailingDelim = options.IsSet(TermListOptions.AllowTrailingDelimiter) && delimiter != null;

//"plusList" is the list for which we will construct expression - it is either extra plus-list or original list.

// In the former case (extra plus-list) we will use it later to construct expression for list

NonTerminal plusList = isPlusList ? list : new NonTerminal(listMember.Name + "+");

plusList.SetFlag(TermFlags.IsList);

plusList.Rule = plusList; // rule => list

if (delimiter != null)

plusList.Rule += delimiter; // rule => list + delim

if (options.IsSet(TermListOptions.AddPreferShiftHint))

plusList.Rule += PreferShiftHere(); // rule => list + delim + PreferShiftHere()

plusList.Rule += listMember; // rule => list + delim + PreferShiftHere() + elem

plusList.Rule |= listMember; // rule => list + delim + PreferShiftHere() + elem | elem

if (isPlusList) {

// if we build plus list - we're almost done; plusList == list

// add trailing delimiter if necessary; for star list we'll add it to final expression

if (allowTrailingDelim)

plusList.Rule |= list + delimiter; // rule => list + delim + PreferShiftHere() + elem | elem | list + delim

} else {

// Setup list.Rule using plus-list we just created

list.Rule = Empty | plusList;

if (allowTrailingDelim)

list.Rule |= plusList + delimiter | delimiter;

plusList.SetFlag(TermFlags.NoAstNode);

list.SetFlag(TermFlags.IsListContainer); //indicates that real list is one level lower

}

return list.Rule;

}//method

#endregion

#region Hint utilities

protected GrammarHint PreferShiftHere() {

return new PreferredActionHint(PreferredActionType.Shift);

}

protected GrammarHint ReduceHere() {

return new PreferredActionHint(PreferredActionType.Reduce);

}

protected TokenPreviewHint ReduceIf(string thisSymbol, params string[] comesBefore) {

return new TokenPreviewHint(PreferredActionType.Reduce, thisSymbol, comesBefore);

}

protected TokenPreviewHint ReduceIf(Terminal thisSymbol, params Terminal[] comesBefore) {

return new TokenPreviewHint(PreferredActionType.Reduce, thisSymbol, comesBefore);

}

protected TokenPreviewHint ShiftIf(string thisSymbol, params string[] comesBefore) {

return new TokenPreviewHint(PreferredActionType.Shift, thisSymbol, comesBefore);

}

protected TokenPreviewHint ShiftIf(Terminal thisSymbol, params Terminal[] comesBefore) {

return new TokenPreviewHint(PreferredActionType.Shift, thisSymbol, comesBefore);

}

protected GrammarHint ImplyPrecedenceHere(int precedence) {

return ImplyPrecedenceHere(precedence, Associativity.Left);

}

protected GrammarHint ImplyPrecedenceHere(int precedence, Associativity associativity) {

return new ImpliedPrecedenceHint(precedence, associativity);

}

protected CustomActionHint CustomActionHere(ExecuteActionMethod executeMethod, PreviewActionMethod previewMethod = null) {

return new CustomActionHint(executeMethod, previewMethod);

}

#endregion

#region Term report group methods

/// <summary>

/// Creates a terminal reporting group, so all terminals in the group will be reported as a single "alias" in syntex error messages like

/// "Syntax error, expected: [list of terms]"

/// </summary>

/// <param name="alias">An alias for all terminals in the group.</param>

/// <param name="symbols">Symbols to be included into the group.</param>

protected void AddTermsReportGroup(string alias, params string[] symbols) {

TermReportGroups.Add(new TermReportGroup(alias, TermReportGroupType.Normal, SymbolsToTerms(symbols)));

}

/// <summary>

/// Creates a terminal reporting group, so all terminals in the group will be reported as a single "alias" in syntex error messages like

/// "Syntax error, expected: [list of terms]"

/// </summary>

/// <param name="alias">An alias for all terminals in the group.</param>

/// <param name="terminals">Terminals to be included into the group.</param>

protected void AddTermsReportGroup(string alias, params Terminal[] terminals) {

TermReportGroups.Add(new TermReportGroup(alias, TermReportGroupType.Normal, terminals));

}

/// <summary>

/// Adds symbols to a group with no-report type, so symbols will not be shown in expected lists in syntax error messages.

/// </summary>

/// <param name="symbols">Symbols to exclude.</param>

protected void AddToNoReportGroup(params string[] symbols) {

TermReportGroups.Add(new TermReportGroup(string.Empty, TermReportGroupType.DoNotReport, SymbolsToTerms(symbols)));

}

/// <summary>

/// Adds symbols to a group with no-report type, so symbols will not be shown in expected lists in syntax error messages.

/// </summary>

/// <param name="symbols">Symbols to exclude.</param>

protected void AddToNoReportGroup(params Terminal[] terminals) {

TermReportGroups.Add(new TermReportGroup(string.Empty, TermReportGroupType.DoNotReport, terminals));

}

/// <summary>

/// Adds a group and an alias for all operator symbols used in the grammar.

/// </summary>

/// <param name="alias">An alias for operator symbols.</param>

protected void AddOperatorReportGroup(string alias) {

TermReportGroups.Add(new TermReportGroup(alias, TermReportGroupType.Operator, null)); //operators will be filled later

}

private IEnumerable<Terminal> SymbolsToTerms(IEnumerable<string> symbols) {

var termList = new TerminalList();

foreach(var symbol in symbols)

termList.Add(ToTerm(symbol));

return termList;

}

#endregion

#region Standard terminals: EOF, Empty, NewLine, Indent, Dedent

// Empty object is used to identify optional element:

// term.Rule = term1 | Empty;

public readonly Terminal Empty = new Terminal("EMPTY");

public readonly NewLineTerminal NewLine = new NewLineTerminal("LF");

//set to true automatically by NewLine terminal; prevents treating new-line characters as whitespaces

public bool UsesNewLine;

// The following terminals are used in indent-sensitive languages like Python;

// they are not produced by scanner but are produced by CodeOutlineFilter after scanning

public readonly Terminal Indent = new Terminal("INDENT", TokenCategory.Outline, TermFlags.IsNonScanner);

public readonly Terminal Dedent = new Terminal("DEDENT", TokenCategory.Outline, TermFlags.IsNonScanner);

//End-of-Statement terminal - used in indentation-sensitive language to signal end-of-statement;

// it is not always synced with CRLF chars, and CodeOutlineFilter carefully produces Eos tokens

// (as well as Indent and Dedent) based on line/col information in incoming content tokens.

public readonly Terminal Eos = new Terminal("EOS", Resources.LabelEosLabel, TokenCategory.Outline, TermFlags.IsNonScanner);

// Identifies end of file

// Note: using Eof in grammar rules is optional. Parser automatically adds this symbol

// as a lookahead to Root non-terminal

public readonly Terminal Eof = new Terminal("EOF", TokenCategory.Outline);

//Artificial terminal to use for injected/replaced tokens that must be ignored by parser.

public readonly Terminal Skip = new Terminal("(SKIP)", TokenCategory.Outline, TermFlags.IsNonGrammar);

//Used as a "line-start" indicator

public readonly Terminal LineStartTerminal = new Terminal("LINE\_START", TokenCategory.Outline);

//Used for error tokens

public readonly Terminal SyntaxError = new Terminal("SYNTAX\_ERROR", TokenCategory.Error, TermFlags.IsNonScanner);

public NonTerminal NewLinePlus {

get {

if(\_newLinePlus == null) {

\_newLinePlus = new NonTerminal("LF+");

//We do no use MakePlusRule method; we specify the rule explicitly to add PrefereShiftHere call - this solves some unintended shift-reduce conflicts

// when using NewLinePlus

\_newLinePlus.Rule = NewLine | \_newLinePlus + PreferShiftHere() + NewLine;

MarkPunctuation(\_newLinePlus);

\_newLinePlus.SetFlag(TermFlags.IsList);

}

return \_newLinePlus;

}

} NonTerminal \_newLinePlus;

public NonTerminal NewLineStar {

get {

if(\_newLineStar == null) {

\_newLineStar = new NonTerminal("LF\*");

MarkPunctuation(\_newLineStar);

\_newLineStar.Rule = MakeStarRule(\_newLineStar, NewLine);

}

return \_newLineStar;

}

} NonTerminal \_newLineStar;

#endregion

#region KeyTerms (keywords + special symbols)

public KeyTermTable KeyTerms;

public KeyTerm ToTerm(string text) {

return ToTerm(text, text);

}

public KeyTerm ToTerm(string text, string name) {

KeyTerm term;

if (KeyTerms.TryGetValue(text, out term)) {

//update name if it was specified now and not before

if (string.IsNullOrEmpty(term.Name) && !string.IsNullOrEmpty(name))

term.Name = name;

return term;

}

//create new term

if (!CaseSensitive)

text = text.ToLower(CultureInfo.InvariantCulture);

string.Intern(text);

term = new KeyTerm(text, name);

KeyTerms[text] = term;

return term;

}

#endregion

#region CurrentGrammar static field

//Static per-thread instance; Grammar constructor sets it to self (this).

// This field/property is used by operator overloads (which are static) to access Grammar's predefined terminals like Empty,

// and SymbolTerms dictionary to convert string literals to symbol terminals and add them to the SymbolTerms dictionary

[ThreadStatic]

private static Grammar \_currentGrammar;

public static Grammar CurrentGrammar {

get { return \_currentGrammar; }

}

internal static void ClearCurrentGrammar() {

\_currentGrammar = null;

}

#endregion

#region AST construction

public virtual void BuildAst(LanguageData language, ParseTree parseTree) {

if (!LanguageFlags.IsSet(LanguageFlags.CreateAst))

return;

var astContext = new AstContext(language);

var astBuilder = new AstBuilder(astContext);

astBuilder.BuildAst(parseTree);

}

#endregion

}//class

}//namespace

Содержание анализируемого кода:

using System;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 1;

string s = Convert.ToString(i);

Console.WriteLine(s);

int j = int.Parse(s);

bool result = j.Equals(i);

Console.Write(result);

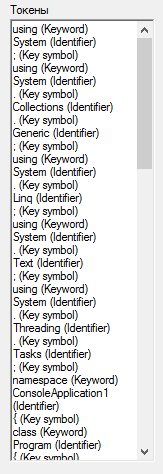
Console.ReadKey();

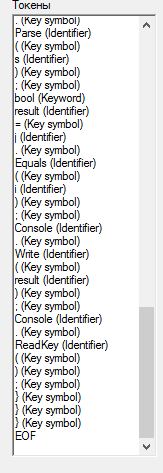
}

}

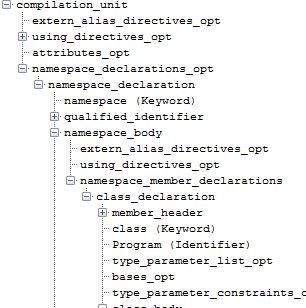
}

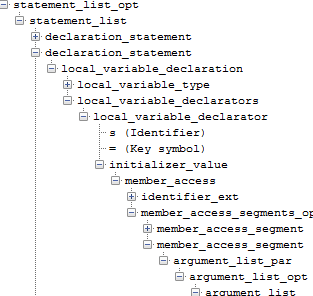
Результат выполнения лексического анализа





Дерево разбора:





Вывод сообщений о наличии ошибок:

