Parser Theory

Drikus Kleefsman
mail:project@.drikus.net

Januari 4, 2014

A short introduction to the use of TinyPG1.3. Version 1.0

1 Introduction

TinyPG is a parser generator in C#, written by Herre Kuijpers. The project can be found on Codeplex (http://www.codeproject.com/Articles/28294/a-Tiny-Parser-Generator-v1-2). This article is a short introduction on the product, on how it it works and on how to use it. For the first sections of this article only the executable files are needed. Until you arrive at section 4 may be it is best to download only the executables and not the sourcecode.

The parser comes with an article by Herre in which he lists the main strengths of TinyPG. He says it is a powerful tiny utility in which to define the grammar for a new compiler, that

- provides syntax and semantics checking of the grammar
- generates a tiny set of sources for the parser/scanner/parsetree (just three .cs or .vb files without any external dependencies)
- allows each generated file to remain clearly human readable, and debuggable with Visual Studio(!)
- includes an expression evaluation tool that generates a traversable parse tree
- has the option to include C# code blocks inside the grammar, adding immediate functionality with just a few lines of code
- includes a tiny regular expression tool
- tries to keep things as simple and tiny as possible

In section 2 the syntax of (E)BNF is introduced. In section 3 we look at the simple example of an Expression evaluator that shows how to use TinyPG and in section 4 we look at the source code of TinyPG itself.

A very nice introduction on the theory of parsing can be found in "Grammars and parsing with C# 2.0", by Peter Sestoft and Ken Friis Larsen (SL-06) (http://www.itu.dk/people/kfl/parsernotes.pdf). Also Chapter 5 from the classic "Algorithms + Data structures = Programs" by Niklaus Wirth (W-76) (however, not on the internet) is a good introduction.

2 Syntax

2.1 Production rules

A text in a languague is a string of terminal symbols. But not every sequence of terminal symbols is correct in a language. The text has to follow certain syntax rules that together form the grammar of the language. To give the syntax rules we introduce the concept of non-terminals. As can be read in SL-06 the grammar of a language is given by G = (T,N,R,S), where T is a set of terminals, N a set of nonterminals, R a set of rules, and S a starting symbol S. Starting from S all possible sequences of terminals are given by the rules of the grammar. To state the rules we use the language E(NBF). We will assume that the rules are context free: every rule will have the following form $A = f_1|...|f_n$, where $A \in N$ is a nonterminal and each alternative f_i is a sequence $e_{i1}e_{i2}...e_{im}$ where each e_{ij} is a terminal or nonterminal symbol. And as a special sequence we have the empty sequence written as Λ.

2.2 Examples of rules

2.2.1 Example 1

The first two examples are from (W-76). In this first example the scentences of a natural language are restricted by the following three production rules. I am using a BNF convention to write non-terminals within < ... > brackets.

$$\begin{aligned} &< Scentence > ::= < Subject >< Predicate > \\ &< Subject > ::= cats|dogs \\ &< Predicate > ::= sleep|eat \end{aligned}$$

The terminals are cats, dogs, sleep and eat. An example of a scentence that can be formed with these rules is "cats eat".

2.2.2 Example 2

In this example we have a recursive production rule and this gives rise to an infinity of possible scentences.

$$\begin{aligned} &< S > ::= x < A > \\ &< A > ::= z|y < A > \end{aligned}$$

An example of a scentence that can be formed with these rules is "xyyyyyyz", but there is of course no limit on the number of y's between the opening 'x' and closing 'z'.

2.2.3 Example 3

In this example we see that optional elements can be expressed using the empty sequence Λ.

$$\begin{aligned} &< A > ::= < Body > \mid \Lambda \\ &< Body > ::= ... \end{aligned}$$

When the non-terminal $< A >$ is evaluated one has the choice between the non-terminal $< Body >$ or skipping this non-terminal.

2.2.4 Example 4

In this example we see that repetitive elements can be expressed using the empty sequence Λ.

$$< A > ::=< Body >< A > \mid \Lambda$$
$$< Body > ::= ...$$

When the non-terminal $< A >$ is evaluated one has the choice between once more choosing the non-terminal $< Body >$ or skipping this non-terminal.

2.3 EBNF and Syntax diagrams

BNF is a very lowlevel syntax. Extensions have been introduced, one of them being EBNF (Extended BNF). Two important extensions in EBNF are the use of brackets [...] to enclose an optional element and the use of {...} to enclose a repetitive (0, 1 ... ∞ times) element.

TinyPG uses a form of EBNF. There is no special symbol for the empty sequence and Start is a reserved symbol for the Start non-terminal. It uses the following predefined symbols in the production rules for non-terminals:

- * - the symbol or sub-rule can occur 0 or more times.
- + - the symbol or sub-rule can occur 1 or more times.
- ? - the symbol or sub-rule can occur 0 or 1 time.
- | - this defines a choice between two sub rules.
- whitespace - the symbol or sub-rules must occur after each other.
- (...) - allows definition of a sub-rule.

EBNF is a language and as such has also its own syntax rules. The development of TinyPG was started with the statement of the syntax of EBNF, using EBNF itself as language. It gives a precise description of the allowable content for the input of TinyPG. This description can be found in the file "BNFGrammar 1.3.tpg".

2.3.1 Example 5

This example is found at the end of file "BNFGrammar 1.3.tpg" translated to the syntax used in this text. The terminals for the parser are in capital letters. A side note: TinyPG uses "->" instead of "::=", does not use < ... > brackets and a rule always ends in a ";".

$$Rule ::= STRING \mid < Subrule >$$
$$Subrule ::=< ConcatRule > (PIPE < ConcatRule >)*$$
$$ConcatRule ::=< Symbol > +$$
$$Symbol ::= (IDENTIFIER \mid (BRACKETOPEN < Subrule > BRACKETCLOSE))$$
$$UNARYOPER?$$

As usual in this software we have a scanner (also called a lexer) and a parser. The scanner (lexer) reads the terminal symbols and passes them to the parser. In the same file the definition of the terminal symbols can be found. The definition of the terminals used in the example are:

$$
\begin{aligned}
BRACKETOPEN &::= @"\(" \\
BRACKETCLOSE &::= @"\)" \\
PIPE &::= @"\|" \\
UNARYOPER &::= @"(\ * |\ + |\?)" \\
IDENTIFIER &::= @"[a - zA - Z\_][a - zA - Z0 - 9\_] * " \\
STRING &::= @"@?\""(\\""\""|[\""]) * \""""
\end{aligned}
$$

In the right hand sides of the terminal definitions there are of C# strings that are used in Microsofts implementation of regular expressions. In the C# language the @ at the beginning of the string tells the compiler that everyting between the openig and closing quotation marks must stay uninterpreted. The reserved symbols "*", "+", "?", "(" and ")" have a similar meaning as in EBNF. The expressions between "[" and "]" are expressions that tell what letter is accepteble. The regular expression "[a-zA-Z_]" represents one char that can be a lowercase letter, an uppercase letter or an underscore. And the regular expression IDENTIFIER := [a-zA-Z_][a-zA-Z0-9_]* means that an identifier can start with an alphabetic symbol followed by an arbitrary number of alphabetic or decimal symbols. The scanner of TinyPG makes use of the Microsoft regular expression classes to find Terminals in the input text. For more on Microsoft regular expressions see (MS-14-RE).

The complete content of the file "BNFGrammar 1.3.tpg" is as follows:

```
// @TinyPG - a Tiny Parser Generator v1.3
// Copyright  Herre Kuijpers 2008-2012
// this grammar describes the BNF notation

// Incorporated revisions by William A. McKee Aug. 14, 2008

<% @TinyPG Namespace="TinyPG" %>

// Terminals:
BRACKETOPEN       -> @"\(";
BRACKETCLOSE      -> @"\)";
CODEBLOCK            -> @"\{[^\}]*\}([^};][^}]*\}+)*;";
COMMA                   -> @",";
SQUAREOPEN         -> @"\[";
SQUARECLOSE       -> @"\]";
ASSIGN                   -> @"=";
PIPE                        -> @"\|";
SEMICOLON            -> @";";
UNARYOPER           -> @"(\*|\+|\?)";
IDENTIFIER              -> @"[a-zA-Z_][a-zA-Z0-9_]*";
INTEGER                  -> @"[0-9]+";
DOUBLE                   -> @"[0-9]*\.[0-9]+";
```

```
HEX                          -> @"(0x[0-9a-fA-F]{6})";
ARROW                    -> @"->";
DIRECTIVEOPEN     -> @"<%\s*@";
DIRECTIVECLOSE   -> @"%>";
EOF                          -> @"^$";
STRING                    -> @"@?\""(\""\""|[^\""])*\""";

[Skip]
WHITESPACE          -> @"\s+";

[Skip]
COMMENTLINE        -> @"//[^\n]*\n?";

[Skip]
COMMENTBLOCK     -> @"/\*[^*]*\*+(?:[^/*][^*]*\*+)*/";

// Production lines LL(1):
Start              -> Directive* ExtProduction* EOF;
Directive  -> DIRECTIVEOPEN IDENTIFIER NameValue* DIRECTIVECLOSE;
NameValue -> IDENTIFIER ASSIGN STRING;
ExtProduction      -> Attribute* Production;
Attribute        -> SQUAREOPEN IDENTIFIER ( BRACKETOPEN Params? BRACKETCLOSE )? SQUARECLOSE
Params -> Param (COMMA Param)*;
Param -> INTEGER | DOUBLE | STRING | HEX;
Production         -> IDENTIFIER ARROW Rule (CODEBLOCK | SEMICOLON);
Rule               -> STRING | Subrule;
Subrule            -> ConcatRule (PIPE ConcatRule)* ;
ConcatRule       -> Symbol+;
Symbol             -> (IDENTIFIER | (BRACKETOPEN Subrule BRACKETCLOSE) ) UNARYOPER?;
```

A nice graphical representation for ENBF syntax rules is with syntax diagrams. In (W-76) examples of syntax diagrams are given e.g. for the language PL/0. Latex has the rail package for writing these syntax diagrams.

3 Example of its use

The article of Herre already has an example of its use. In it a numeric expression evaluator is defined. The grammar of this expression evaluator is defined in the file "Examples/Simple expression1.tpg" as

```
//Terminals:
NUMBER  -> @"[0-9]+";
PLUSMINUS  -> @"(\+|-)";
MULTDIV  -> @"\*|/";
BROPEN  ->  @"\(";
BRCLOSE  ->  @"\)";
EOF         -> @"^$";
```

```
[Skip] WHITESPACE  -> @"\s+";

Start  -> (AddExpr)?  EOF;
AddExpr  -> MultExpr (PLUSMINUS MultExpr)*
MultExpr  -> Atom (MULTDIV Atom)*;
Atom  -> NUMBER | BROPEN AddExpr BRCLOSE;
```

In the file "Examples/Simple expression2.tpg" this syntax is augmented with code blocks to do the evaluation of in input expression into a number. For instance the AddExpr has the following codeblock

```
AddExpr -> MultExpr (PLUSMINUS MultExpr)*
{
int Value = Convert.ToInt32($MultExpr);
int i = 1;
while ($MultExpr[i] != null)
{
string sign = $PLUSMINUS[i-1].ToString();
if (sign == "+")
Value += Convert.ToInt32($MultExpr[i++]);
else
Value -= Convert.ToInt32($MultExpr[i++]);
}

return Value;
};
```

This code tells us how to evaluate the Value of an AddExpr. We will see the C# code of this later on.

When running the TinyPG you can import the example file (use the version "Examples/Simple expression2.tpg") and then press F6 (or select menu option Build — Generate). The result of the run are three important files with names Scanner.cs, Parser.cs and ParseTree.cs. When using "Examples/Simple expression2.tpg" one can press F5 (or select menu option Build — Generate and Run) to evaluate values of expressions (use the Evaluation evaluator window for the input of expresions) that follow the given syntax rules like 5+7*(80-4). Results (and errors) are reported in the Output window (537 in this example).

3.1 Contents of generated files

The generated files from "Examples/Simple expression2.tpg" can be used to evaluate expressions like 5+7*(80-4). Create a project that contains the generated file Scanner.cs, Parser.cs and ParseTree.cs and add to it the following Program file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using TinyPG;

namespace TestParser
{
    class Program
    {
        static void Main(string[] args)
        {
            Parser parser = new Parser(new Scanner());
            ParseTree parseTree = parser.Parse("5+7*(80-4)");
            System.Console.WriteLine("5+7*(80-4)= " + parseTree.Eval());
            System.Console.ReadLine();
        }
    }
}
```

When executing the program the result in the output window will be 5+7*(80-4)= 537. The way the parser does its work is simpel. The parser always knows what the possible choices of terminals are after the current one. It asks the lexer to read the next terminal. The parser starts with the method ParseStart for the Start production line of the syntax for an expression. In the method it creates a Start token with

```
ParseNode node = parent.CreateNode(scanner.GetToken(TokenType.Start), "Start");
```

because this token will be the first to be added to the syntaxtree. After that the parser asks the lexer to read ahead the next terminal.

```
tok = scanner.LookAhead(TokenType.NUMBER, TokenType.BROPEN);
```

The scanner knows that after the start symbol the next terminal has to be one of three, a NUMBER, a BROPEN or a EOF. This follows from the production lines in the sytax of expressions. After the start terminal there have to follow zero or more AddExpr non-terminals followed by an EOF terminal. And an AddExpr has to start with an Atom, which starts with a NUMBER or a BROPEN. If it starts with a NUMBER or a BROPEN the parsing continues in the method ParseAddExpr(node) and else we have to have the ending EOF terminal of the production line for Start. Of course the explanation of the ParseAddExpr(node) method can be found looking at the production line for the AddExpr. In the process of parsing the found terminals are added to the syntaxtree.

The codeblocks added to the syntax productionlines can be found back in the methods EvalStart, EvalAddExpr, EvalMultExpr and EvalAtom of parsetree. For example the code of EvalAddExpr is as follows

```
protected virtual object EvalAddExpr(ParseTree tree, params object[] paramlist)
{
    int Value = Convert.ToInt32(this.GetValue(tree, TokenType.MultExpr, 0));
    int i = 1;
```

```
    while (this.GetValue(tree, TokenType.MultExpr, i) != null)
    {
        string sign = this.GetValue(tree, TokenType.PLUSMINUS, i-1).ToString();
        if (sign == "+")
            Value += Convert.ToInt32(this.GetValue(tree, TokenType.MultExpr, i++));
        else
            Value -= Convert.ToInt32(this.GetValue(tree, TokenType.MultExpr, i++));
    }
    return Value;
}
```

Together with the methods GetValue and Eval of the parsetree the Eval of the parsetree is done. In the produced code for the codeblock "this" is an AddExpr non terminal. The syntax for an AddExpr is: AddExpr ->MultExpr (PLUSMINUS MultExpr)*, so to evaluate a AddExpr one has to evaluate the first MultExpr and after that a number of (PPLUSMINUS MultExpr), which is done in the while-loop. Look at how the first $MultExpr$ in the codeblock is translated to $MultExpr[0]$ (index=0) in the C# code (tree, TokenType.MultExpr, 0).

Of course we can use a parsetree without codeblocks. Then we traverse the parsetree and at moments of our choosing we produce the things we want. As an example, we might use as input to TinyPG the syntax for a C header file and generate the parser files for this syntax. Then we parse a C header file with the generated parser to produce the parsetree. Traversing this tree we can for instance translate a C function prototype to a C# method whenever the node is of type FunctionNode.

4 The source code (or what Herre did)

Now let us look at the sourcecode of TinyPG. The sourcecode can be downloaded from the Codeplex site. Our main interest is in the code found in the subdirectory "Compiler".

To be able to understand what is going on read the whole article (SL-06). In summary, TinyPG is a LL(1) parser generator. It reads a syntax and produces a top-down LL(1) parser for that syntax. The generated parser tries to find a derivation of the Start symbol that matches the input string that is parsed. It does this by reading symbols from the input string from left to right (the meaning of the first 'L' in LL(1)) and in the process of finding a match the Leftmost nonterminal in the derivation tree is replaced (the meaning of the second 'L' in LL(1)) trying to find a match with 1 symbol from the input string (the '1' in LL(1)). In chapter 3 of (SL-06) it is shown that not every syntax is acceptable for LL(1) parsing. The syntax should be written so that it is always possible to choose between alternatives. If we have a nonterminal A written as $A \rightarrow f_1|f_2|...f_n$ then the choice between the alternatives must be made based on only one non-terminal t that is at the head of the input stream. If the choice cannot be made we do not have a good syntax. We will choose option f_i if this option f_i can start with symbol t or if this option can be empty and the symbol A can be followed by s. For the latter we have to look in all the other derivation rules to find the symbol A on the righthandside and the symbols following the symbol A. The set of terminals an option f_i can start with is called $First(f_i)$. The set of a terminals a non-terminal A can follow is called $Follow(A)$. How to construct these sets, see (SL-06).

To write a parser generator for a random LL(1) syntax like TinyPG a number of steps are needed. The first step is to read the syntax and analyze it. Reading the syntax actually

means parsing what you read and store it in a syntaxtree to analyze it there. This text will be parsed and put in a tree structure. Let us call this stucture, like Herre did, a Grammartree. It contains the grammar for which to produce a parser. After translating the grammartree to a more suitable structure (a kind of abstract grammartree and let us call this structure, like Herre did, the Grammar) that is the basis for generating the parser for the grammar. That generated parser also has to read in a file and translate it to a syntaxtree. Let us call this structure Parsetree. So, the whole process is $Input \rightarrow GrammarTree \rightarrow Grammar \rightarrow Parsercode \rightarrow compiledParser \rightarrow Parsetree$

4.1 What happens, an example

Let us follow this process in our example of the expression evaluator where the production rules were as follows:

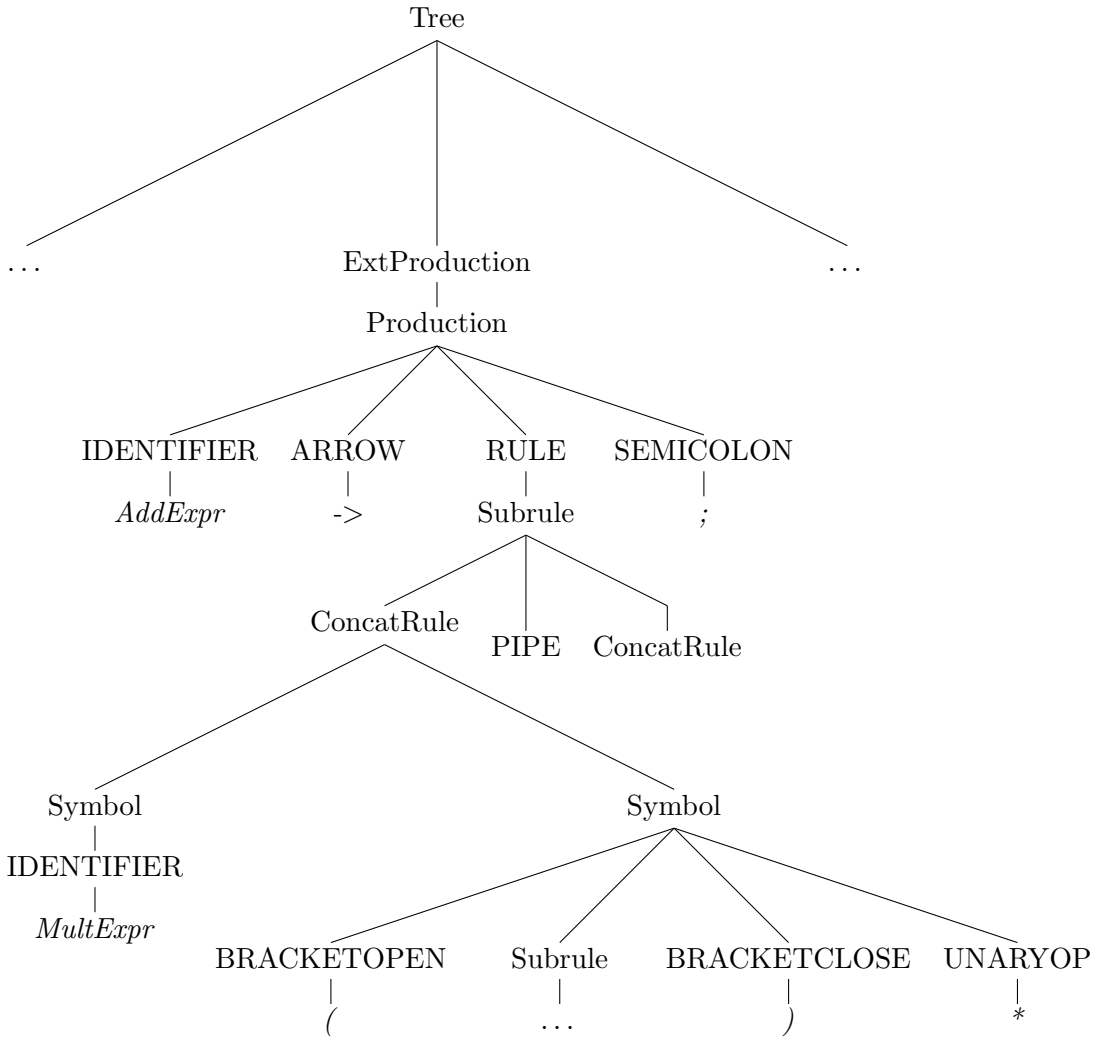
```
//Terminals:
NUMBER  -> @"[0-9]+";
PLUSMINUS  -> @"(\+|-)";
MULTDIV  -> @"\*|/";
BROPEN  ->  @"\(";
BRCLOSE  ->  @"\)";
EOF         -> @"^$";

[Skip] WHITESPACE  -> @"\s+";

Start  -> (AddExpr)?  EOF;
AddExpr  -> MultExpr (PLUSMINUS MultExpr)* | (MultExpr)*;
MultExpr  -> Atom (MULTDIV Atom)*;
Atom  -> NUMBER | BROPEN AddExpr BRCLOSE;
```

But wait, the production line for AddExpr has been changed! We added "— (MultExpr)*;" to the line!. It is production line for AddExpr that will have our attention in what follows and we made it a little more complex and even totally unacceptable because because now we have two options in the line and both start with the symbol "MultExpr". Let us see what happens.

The parser that reads this file knows that he has to expect that the contents of the production lines follow the EBNF syntax given in the file "BNFGrammar 1.3.tpg" (forgetting here the terminal symbols and directives):

```
Start            ->  ExtProduction* EOF;
ExtProduction  -> Attribute* Production;
Attribute    -> SQUAREOPEN IDENTIFIER ( BRACKETOPEN Params? BRACKETCLOSE )? SQUARECLOSE;
Params           -> Param (COMMA Param)*;
Param             -> INTEGER | DOUBLE | STRING | HEX;
Production     -> IDENTIFIER ARROW Rule (CODEBLOCK | SEMICOLON);
Rule             -> STRING | Subrule;
Subrule        -> ConcatRule (PIPE ConcatRule)* ;
ConcatRule      -> Symbol+;
Symbol        -> (IDENTIFIER | (BRACKETOPEN Subrule BRACKETCLOSE) ) UNARYOPER?;
```

The parser (source can be found in the subdirecory Compiler) produces a GrammarTree. A small part of the tree is represented in the following tree (part of the production rule AddExpr ->MultExpr (PLUSMINUS MultExpr)* — (MultExpr)*;.



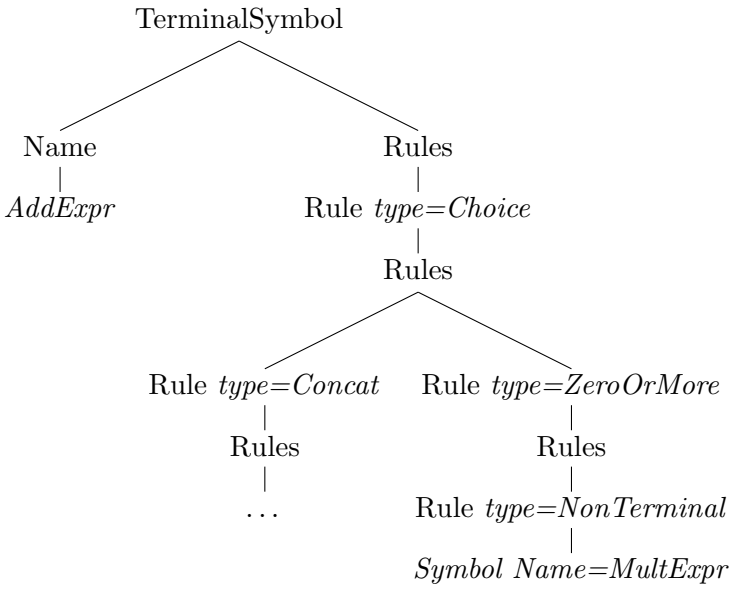
The topnode of the tree is the tree itself and to this node are added nodes that contain Tokens of type TokenType.ExtProduction and a node with a Token of TokenType.EOF. There are many nodes directly under the start node, one for each production line but also for each terminal line. All those lines in the file with the expression-syntax are seen by the parser as examples of an ExtProduction. This parser simply thinks let me find as many ExtProductions and finally a EOF and my job is done. In the same way in finding an ExProduction it tries to find a number of Attributes followed by a Production. Because in the example-file there are no attributes only Production nodes (a GrammarNode that contain a Token of type TokenType.Production) are added to the tree. In the figure we see how our AddExpr is stored in the ENBF tree and that

the source text is found back as leaves leaves of this tree.

After the GrammarTree is built the tree gets translated to a Grammar. This is done by calling the Eval() method of the tree and this will call the Eval on the nodes resursivly. Exept for the code for the Eval function itself which can be found in the file ParseTree all the code can be found in GrammarTree.cs. ParseTree.Eval(treeNode,) uses the token in tree node to call one of the EvalXXX functions in GrammarTree.cs. Beginning with EvalStart(treeNode, objecLlist). This method creates the Grammar object and this object is passed to other functions using the objectlist. After that it adds information about its nodes to the Grammar and finally it calls the Eval function on all its nodes for recursion, which calls one of the other EvalXXX functions in GrammarTree. The reason for creating the (so called abstract tree) Grammar is that the Grammartree, although it has all the information needed, is not the most elegant structure to produce the code for the parser that will be generated. In Grammar we will create a better structure. A good example for this is in finding the Nonterminal rules with the regular expressions. In the GramarTree these are just ExtProductions and it is not easy to see why some are ExtProductions for Terminals and some for NonTerminals. In the method EvalStart the solution is given with the following code for a node of type ExtProduction:

```
if (n.Nodes[n.Nodes.Count - 1].Nodes[2].Nodes[0].Token.Type == TokenType.STRING)
```

It means look in the last of the nodes of an ExtProduction (ExtProduction ->Attribute* Production;) and then it its third node (Production ->IDENTIFIER ARROW Rule (CODE-BLOCK — SEMICOLON);) which is the Rule node, and then in its first node (Rule ->STRING — Subrule;), if that has a token of TokenType.STRING) ... then we have a Terminal production. In the Grammar object it is added to the list of Symbol. Also if attributes are present in the ExtProduction they are now added as children to the TerminalSymbol. After EvalStart has done its work all symbols are added to the Grammar (the lefthand sides of the ExtProductions). In the other methods to the symbols the Rules are added (the righthandsides). After all the evaluations have been done the Grammar object for our Expression evaluator has 1 directive, 1 skipsymbol (whitespace) and 11 symbols (7 TerminalSYmbols and 4 NonTerminalSymbols). A Terminal has a Name (for instance NUMBER) and an Expression (for NUMBER this is @"[0-9]*". A NonTerminalSymbol contains a Rule structure that is a translation of the treestructure in a GrammarTree (a Rule can contain zero or more Rules) and here things are simplified. A Rule has a type and it is immediatly clear if a rule is an optional element. A part of the rule structure for our example production rule AddExpr ->MultExpr (PLUSMINUS MultExpr)* — (MultExpr)* ; is now represented by



After calling the Eval() on the GrammarTree one can call Grammar.Preprocess(). In it a call is made to DetermineFirsts(). Of course DetermineFirsts() uses the Grammar structure to find out what the First symbols are for each rule. NonTerminalSymbols have a field FirstTerminals that is filled with the possible alternatives. For instance for the NonTerminal AddExpr in our example the field FirstTerminals is filled with the TerminalSymbol values NUMBER and BROPEN and for the NonTerminal Start the field is filled with the TerminalSymbol values NUMBER, BROPEN and EOF. The FirstTerminals are however not used as a check that the syntax of the given input is correct. The syntax that was used in our example $AddExpr{->}MultExpr(PLUSMINUSMultExpr)* \mid (MultExpr)*$; gave no compiler warnings although it is obvious that a syntax that starts with the same symbol in the options (here MultExpr) can give problems. However the produced Parser code does give rise to errors the moment it gets compiled. The code that is generated looks like this:

```
private void ParseAddExpr(ParseNode parent)
{
    Token tok;
    ParseNode n;
    ParseNode node = parent.CreateNode(scanner.GetToken(TokenType.AddExpr), "AddExpr")
    parent.Nodes.Add(node);

    tok = scanner.LookAhead(TokenType.NUMBER, TokenType.BROPEN);
    switch (tok.Type)
    {
        case TokenType.NUMBER:
        case TokenType.BROPEN:

            ParseMultExpr(node);

            tok = scanner.LookAhead(TokenType.PLUSMINUS);
```

```
            while (tok.Type == TokenType.PLUSMINUS)
            {
              ...
            }
            break;

        case TokenType.NUMBER:
        case TokenType.BROPEN:
            tok = scanner.LookAhead(TokenType.NUMBER, TokenType.BROPEN);
            while (tok.Type == TokenType.NUMBER
                || tok.Type == TokenType.BROPEN)
            {
                ParseMultExpr(node);
                tok = scanner.LookAhead(TokenType.NUMBER, TokenType.BROPEN);
            }
            break;
        default:
            tree.Errors.Add(...);
            break;
     }
    parent.Token.UpdateRange(node.Token);
}
```

As can be seen, in TinyPG the code follows the syntax (AddExpr ->MultExpr (PLUSMINUS MultExpr)* | (MultExpr)*;) and the FirstTerminals are used each time a NonTerminal is on the righthandside of the productionline for producing case statements. But because the MultExpr is in the head of the first option as well in the head of the next option of the productionline we have twice the case statements "case TokenType.NUMBER: case TokenType.BROPEN:" in the loop giving an obvious compile error.

The production of the code can get started after that the Grammar knows its FirstTerminals. The parser is generated with a call to Compiler.Compile(Grammar) which calls BuildCode that will use the right generator (the default is the C# generator) to build the parser code.

4.2 How to create a Parser generator

One of the nice things of the EBNF language is that it is possible to express it's syntax in EBNF itself. This is done in the file "BNFGrammar 1.3.tpg". So, if you have a tool like TinyPG then it can be used to generate a parser for EBNF itself. If you feed TinyPG with the file "BNFGrammar 1.3.tpg" you will see this happening and the generated Parser.cs is identical to the Parser.cs used by TinyPG. It seems a good reason not to write that code by hand. Write the code that cannot be generated. Write the code for GrammarTree, Grammar and the code for the generation of the parser. Create a Grammar object for the syntax in "BNFGrammar 1.3.tpg" and generate your parser and scanner. It will take quit some time I guess to write all the code for TinyPG.

5 The rest of the code

Apart from creating all the code for a generator there is more to be done when programming TinyPG. What has been done as well is

- A user interface is made. Use is made of Floaty objects for windows.
- A special window is made for evaluating Microsoft Regular expressions.
- Code for text-highlighting is made.
- Evaluting Codeblocks is made possible. With a Microsoft CodeProvider object the generated treeview code can be compiled to executable.

6 License

Herre's article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL). To give an impression of the CPOL licence, here is its Pramble

Preamble

This License governs Your use of the Work. This License is intended to allow developers to use the Source Code and Executable Files provided as part of the Work in any application in any form.

The main points subject to the terms of the License are:

- Source Code and Executable Files can be used in commercial applications;
- Source Code and Executable Files can be redistributed; and
- Source Code can be modified to create derivative works.
- No claim of suitability, guarantee, or any warranty whatsoever is provided. The software is provided "as-is".
- The Article(s) accompanying the Work may not be distributed or republished without the Author's consent

This License is entered between You, the individual or other entity reading or otherwise making use of the Work licensed pursuant to this License and the individual or other entity which offers the Work under the terms of this License ("Author").

7 More

7.1 More on the internet

- (SL-06) "Grammars and parsing with C# 2.0", by Peter Sestoft and Ken Friis Larsen; http://www.itu.dk/people/kfl/parsernotes.pdf
- (MS-14-RE) "Regular Expression Language - Quick Reference", MSDN Library, http://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx

7.2 More in books

- (W-76) "Algorithms + Data structures = Programs", by Niklaus Wirth; ISBN 0-13-022418-9