9,814,526 members (46,056 online)

CODE PROJECT For those who code

SpreadsheetGear Performance Spreadsheet Components Download Free Trial

home articles quick answers discussions features community help

Search for articles, questions, tips

Articles » General Programming » Algorithms & Recipes » Parsers and Interpreters Next →

Article
Browse Code
Stats
Revisions (5)
Alternatives
Comments & Discussions (253)

Add your own alternative version

a Tiny Parser Generator v1.2

By **Herre Kuijpers**, 22 Sep 2010

★ ★ ★ ★ ★ 4.95 (171 votes)

Is your email address OK? You are signed up for our newsletters but your email address is either unconfirmed, or has not been reconfirmed in a long time. Please click **here** to have a confirmation email sent so we can confirm your email address and start sending you newsletters again. Alternatively, you can **update your subscriptions**.

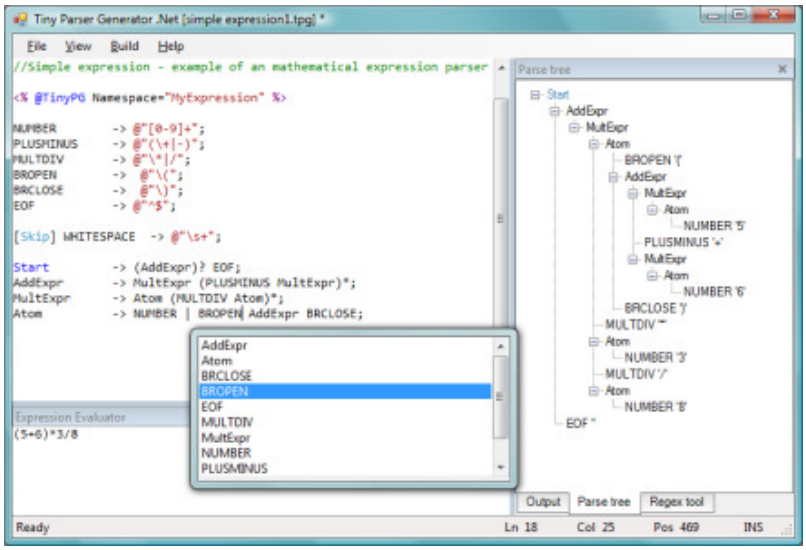
Download TinyPG v1.3 binaries - 88.9 KB

Download TinyPG v1.3 sources - 0.98 MB

Download TinyPG v1.2 binaries - 77.07 KB

Download TinyPG v1.2 sources - 225.32 KB

Download TinyPG v1.0 sources - 103.91 KB



Introduction

@TinyPG stands for "a Tiny Parser Generator". This particular generator is an LL(1) recursive descent parser generator. This means that instead

About Article

@TinyPG is a utility that makes it easier to write and try out your own parser/compiler.

Type	**Article**
Licence	**CPOL**
First Posted	**1 Aug 2008**
Views	**192,970**
Downloads	**7,368**
Bookmarked	**395 times**

.NET2.0 Win2K WinXP Win2003 Vista VS2005 , +

Bookmark Print Email

of generating a state machine out of a grammar like most compiler compilers, it will generate source code directly; basically generating a method for each non-terminal in the grammar. Terminals are expressed using .NET's powerful Regular Expressions. To help the programmer create .NET Regular Expressions, a Regular Expression (Regex) tool is embedded in TinyPG. Grammars can be written using the extended BNF notation.

TinyGP v1.2 now allows you to generate a scanner, parser, and parsetree file in either C# or VB code(!). These can be compiled directly into your own projects. Additionally, now it is possible to generate code for your own text highlighter which you can use directly in your own text editor project. A simple example is added at the end of this article.

In this article, I will not go into depth about compiler theory, and a basic understanding of compilers and grammars is required. For your reference, I have included a list of terms used in this article explained on Wikipedia:

- grammar,
- BNF,
- terminal,
- non-terminal,
- LL(1),
- lookahead,
- recursive decent,
- concrete syntax tree (CST),
- parse tree.

Nowadays, with the availability of numerous compiler compilers, it becomes hard to think of a new name for a compiler compiler. 'Yet Another Compiler Compiler' was taken, so I decided to name this tiny tool after its many strengths:

- 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*

Background

Since there are already a number of compiler compilers out there, you might wonder why write another one? The reasons for writing this utility are:

Top News

WebKit Group Strikes Back: Let's Remove Chrome

Get the Insider News free each morning.

Related Videos

C# .NET Programming - 1107 Generating Random Numbers

C# .NET Programming - 0704 Exercise - Generating A Form Letter

- The fun factor. Seriously, who doesn't want to write his/her own compiler compiler!?
- The desire for a compiler compiler that generates C# code.
- The utility should be free, allowing developers to use the generated code for whatever purpose with no restrictive licenses.
- The generated code should be readable and relatively easy to debug if needed.
- The generated code should be completely independent and not require any external libraries to run (e.g., libraries from the compiler compiler itself).
- The source code should be made available to allow developers to modify the utility as they like.
- The tool should be free.
- It should be possible to separate the semantics from the syntax (using subclassing), or combine them (using inline code blocks).

Using the tool

I will explain the usage of the tool by means of two tiny tutorials where we will start writing a tiny expression calculator. In the first tutorial, we will define the grammar for the expression evaluator; this will allow us to parse expressions. In the second tutorial, we will add some code blocks to implement the semantics of the expression evaluator; after this step, we will also be able to evaluate or calculate simple expressions.

Before starting, I want to explain a little about naming conventions I prefer to use, since there are no clear conventions for naming, and this utility does not require you to use any specific convention. For readability in the generated code, however, I propose the following:

- For terminals, use only upper case names, perhaps with underscores; e.g., NUMBER, IDENTIFIER, WHITESPACE, or BRACKET_OPEN.
- For non-terminals, use Pascal casing. This corresponds to .NET standards, and since from each non-terminal a method is generated, the method will then also be Pascal cased. E.g., Start, Expression, MultiplyExpression.

To write valid grammars, the following rules should be followed:

- Each production rule must be terminated with a `;` character.
- `Start` is a reserved word which indicates the starting symbol for the grammar.
- Comments can be included using either `//` to comment a line, or `/* ... */` to comment a section.
- When including code blocks, the code block must be written between `{ ... };`. Note that the closing bracket must be immediately followed by the semicolon with no whitespace, or it will result in errors.

Let's start the tutorial.

Simple expression calculator syntax

Related Articles

Simple CSS Parser

Fast Mathematical Expressions Parser

A Tiny Expression Evaluator

The expression evaluator revisited (Eval function in 100% managed .NET)

Spart, a parser generator framework 100% C#

muParserSSE

A Vector Type for C#

An extensible math expression parser with plug-ins

Crafting an interpreter Part 3 - Parse Trees and Syntax Trees

Achieving PostScript and Wmf outputs for OpenGL

The MiniWalker

Writing Your Own RTF Converter

Evaluate Expressions from C# using JavaScript's Eval() Function

Introduction to GOLD Parser

Irony - .NET Compiler Construction Kit

MinosseCC: a lexer/parser generator for C#

C# CodeDOM parser

Using Reflection.Emit to Precompile Expressions to MSIL

Opening a door towards Spirit: a parser framework

Building an SQL Logic Engine

The goal of the expression evaluator will be to parse and evaluate simple numeric expressions consisting of (integer) numbers, +, -, *, and / symbols. To make it a bit more fun, we will also allow the use of sub-expressions with the ( ) symbols. For example, 4*(24/2-5)+14 would be a valid expression that should evaluate to 42. This example is included in TinyPG, by opening the *simple expression1.tpg* file for C#, or the *simple expression1 vb.tpg* file for the VB variant.

Terminals and Regular Expressions

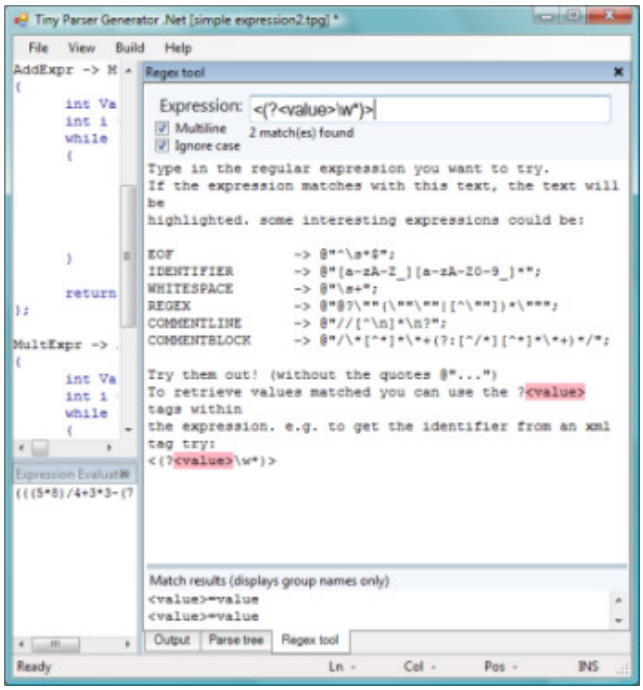
Since we have decided what symbols to allow in the calculator, we can start by defining the terminal symbols of our grammar using terminal production rules and Regular Expressions:

Collapse | Copy Code

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

Terminals can be defined using .NET's Regex syntax, including the @ sign if required. Using .NET's Regex saves a lot of coding, and keeps the *scanner.cs* source file very small and easily readable. As you may have already guessed, the terminal definitions will be directly used within the Regex by the generated parser.

This may be a good opportunity to brush up on your Regular Expression skills. In order to play around with Regular Expressions, I have included the Regex tool within the utility. Just click on the **Regex tool** tab and enter your Regular Expression. Any matches will be highlighted in the text immediately. This way, you can test if your Regular Expression is matching the right symbols. At the end of this article, I will include a number of often used Regular Expressions that may be very useful in your own language.



Note that TinyPG does not have any reserved or predefined terminal symbols. For example, some parser generators reserve the EOF terminal because it may be difficult to express. Regex can almost cope with any

kind of terminal symbol, including an `EOF` symbol. An `EOF` symbol can be defined as follows:

Collapse | Copy Code

```
EOF -> @"^$";
```

The `^` character indicates the regex will scan from the beginning of the text string/file, the `$` indicates that the regex should scan until the end of the string/file. Because no characters were specified in between, this must be the end of the file (or text/string).

In order to be less restrictive about the formatting of the expression, we would like to allow whitespaces. However, we do not want to check in the grammar for whitespaces. In fact, we would like to have the scanner simply ignore whitespaces and continue scanning to the next symbol/token. This can be done by defining a terminal production rule for whitespace and prefixing it with the `[Skip]` attribute, like so:

Collapse | Copy Code

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

Non-terminals and production rules

Once the terminals have been defined, we can define production rules for the non-terminals. TinyPG supports the extended BNF notation, allowing the following symbols to be used in the production rules: `*`, `+`, `?`, `(`, `)`, `|`, and whitespace. These have the following meaning:

- `*` - 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.

The grammar must start with a `Start` non-terminal, so `Start` is a reserved word. Let's define the grammar as follows:

Collapse | Copy Code

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

The `Start` production rule will check if there is an `AddExpr` (optional), and then expects an end of file (no other tokens are expected).

`AddExpr` will only add or subtract `MultExpr` expressions. `MultExpr` will multiply or divide `Atom`s. By writing the grammar this way, the precedence of the `*` and `/` symbols over the `+` and `-` symbols is defined explicitly. The `Atom` for now can be either a number (integer) or another expression. So with only four simple production rules, you can already parse complicated expressions like `(((5*8)/4+3*3-(7+2)) / 5)`.

Running the parser

To compile the grammar, press F6. The output pane should become

visible, displaying again the grammar as it is internally represented, but also displaying the "First" symbols as non-terminal. The "First" symbols are those symbols that the parser will use in making the decision as to which non-terminal or production rule should be parsed next. The generated C# code will also be compiled internally and can be run. If all goes well, "**Compilation successful**" should be displayed.

Press Ctrl+E to open the **Expression Evaluator** pane, then type an expression for the grammar to evaluate, for example, `(((5*8)/4+3*3-(7+2)) / 5)`, and press F5 to run the parser. The expression will be evaluated and "e;Parse was successful" should be displayed. If the parse was successful, the evaluator will continue to evaluate the resulting parse tree. Because we have not implemented any logic behind the production rules, you will get the following warning: "Result: Could not interpret input; no semantics implemented."

Simple expression calculator semantics

Adding code blocks

Now that we have a working grammar, we can start adding semantics to the production rules in terms of code blocks. Code blocks are snippets of C# (or VB) code that are almost directly inserted into the generated parser after replacing some variables. Let's start with the first production rule:

Collapse | Copy Code

```
Start -> (AddExpr)? EOF { return $AddExpr; };
```

Notice the variable `$AddExpr`. `$AddExpr` corresponds to the value of the non-terminal `AddExpr`. A `$`-variable is defined for each terminal and non-terminal in the production rule, and during code generation, the variable will be replaced by an expression. In this case, `$AddExpr` is replaced by a .NET expression that will evaluate the `AddExpr` non-terminal. In this example, `$EOF` would also be a valid variable. Note that terminal and non-terminals always return values of type `object`. You will need to do your own explicit casting if you want to do calculations as in the following snippet:

Collapse | Copy Code

```
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;
};
```

Notice that in this production rule, the term `MultExpr` is defined twice. So to which value instance of `MultExpr` is `$MultExpr` referring? Even more so, the latter `MultExpr` can be repeated endlessly. To refer to a specific instance of the `MultExpr` value, you can use zero-based indexers on the `$`-variable, so that `$MultExpr[1]` will refer to the

second defined instance of `MultExpr` in the input expression. Thus, if we have the expression `3+4*2-6`, we will have three `MultExpr` non-terminals: `3`, `4*2` (= `8`), and `6`. So, `$MultExpr[1]` will evaluate to the value `8`. `$MultExpr[3]`, however, is not available, and will therefore evaluate to the .NET `null` value.

So this code evaluates the first `MultExpr` (in other words, `$MultExpr` is short for `$MultExpr[0]`) and assigns it to `Value`. We know that this always exists according to the grammar. Then, we loop through all of the `$MultExpr[i]` and add to or subtract from `Value` until `$MultExpr[i]` evaluates to `null`. In order to decide if a `MultExpr` should be added or subtracted, we evaluate the `$PLUSMINUS` token. In this manner, we can now actually calculate additions and subtractions.

The same approach can be used for the `MultExpr` production rule. The code is not presented here, but can be found in the *simple expression2.tpg* file.

Last but not least, there is the `Atom` production rule which can be defined as follows:

Collapse | Copy Code

```
Atom -> NUMBER | BROPEN AddExpr BRCLOSE
{ if ($NUMBER != null)
        return $NUMBER;
    else
        return $AddExpr;
};
```

Because the `Atom` rule contains a choice between a `NUMBER` and a sub expression, the code checks if either of the sub rules is null. If not, we return this value.

Running the generated parser/compiler

Compile the grammar again by pressing F6, which should not result in any errors. Then type in the expression `(((5*8)/4+3*3-(7+2)) / 5)` and press F5. This time, the expression should parse successfully, the outcome is calculated, and should return "Result: 2".

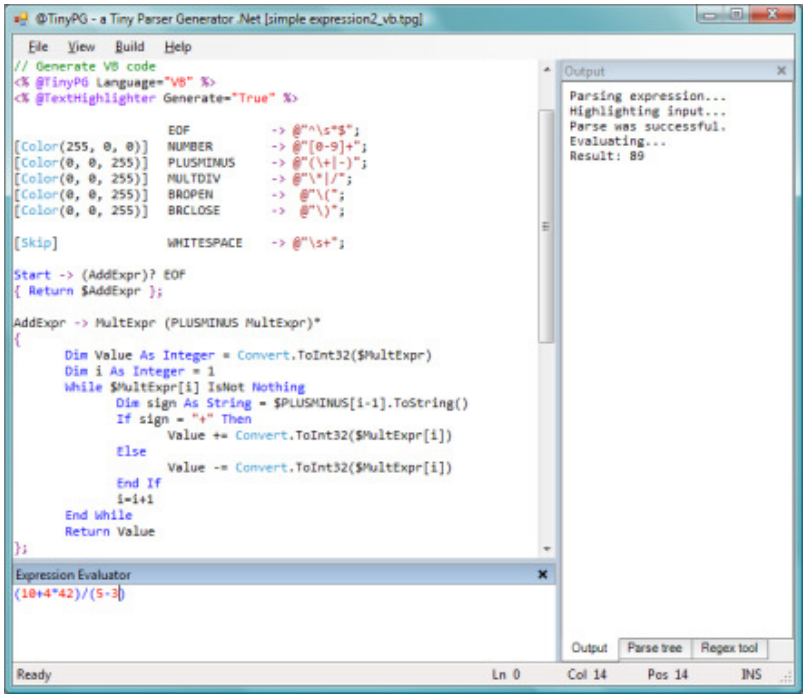
Congratulations, you have written your first TinyPG compiler!

Highlighting your expressions

To make things more interesting, Version 1.2 allows you to add text highlighting to your grammar. Adding text highlighting is done in two steps:

1. Add the `TextHighlighter` directive to the grammar
2. Add the `Color` attribute to terminals you want to have highlighted

When the `TextHighlighter` directive is added, @TinyPG will generate a *TextHighlighter.cs* (or *.vb*) file. The generated TextHighlighter makes use of the generated parser/scanner to parse the input text and apply the color from the `Color` attribute to any terminal it recognizes. For example, the directives and terminals of the simple expression evaluator could look like this:



Collapse | Copy Code

```
// By default the TextHighlighter is not generated.
// Set its Generate attribute to true
<% @TextHighlighter Generate="true" %>

// highlight numbers in red and symbols in blue
[Color(255, 0, 0)] NUMBER -> @"[0-9]+";
[Color(0, 0, 255)] PLUSMINUS -> @"(\+|-)";
[Color(0, 0, 255)] MULTDIV -> @"\*|/";
[Color(0, 0, 255)] BROPEN -> @"\(";
[Color(0, 0, 255)] BRCLOSE -> @"\)";
```

When running expressions in the input pane, the expressions would be calculated and additionally highlighted in the input pane! This makes it very simple to write your own text highlighters.

Additional remarks

Note 1: The use of terminal characters immediately in the production rules is not allowed. This is allowed for some compiler compilers; however, I feel it does not add to the readability of the generated code. Therefore, each terminal must be defined explicitly as a regex expression. Non-terminal production rules may only refer to terminals or other non-terminals in a LL(1) manner. This means that the parser will only be able to look ahead one token. When scanning and parsing, an input token always corresponds to a terminal symbol or is a syntax error. For that reason, the parser only needs to look ahead one terminal to decide which production rule to choose.

E.g., the following grammar will result in errors because it is not LL(1):

Collapse | Copy Code

```
// this is an LL(2) rule, you need to look
// ahead 2 symbols to determine which rule to choose
Start -> PLUSMINUS NUMBER | PLUSMINUS Expression;
```

The parser must choose the production (sub)rule based on one look ahead. Because both subrules are starting with the `PLUSMINUS` terminal, the parser cannot decide. TinyPG does not check that grammars are LL(1), but will simply generate the code. On compilation

of the code, however, you will run into compilation errors. Luckily, an LL(k) (k > 1) rule can always be rewritten to LL(1) form as follows:

Collapse | Copy Code

```
// the rule has been rewritten to LL(1), now only 1 symbol at a
time is
// required to be looked at to make the decision
Start -> PLUSMINUS ( NUMBER | Expression);
```

By rewriting the rule as shown, the production rule is now LL(1) and can be successfully generated in an LL(1) parser ... or can it? The LL(k) problem is perhaps slightly more complicated. What if `Expression` is defined as:

Collapse | Copy Code

```
Expression -> NUMBER | IDENTIFIER;
```

Again, the parser will have the same problem; when it encounters a `NUMBER` token, should it choose the NUMBER rule of `Start` or should it continue parsing an `Expression`? This time, it is more difficult to solve the problem, and in this case, there is no easy solution. It will be necessary to rethink (partly rewrite) your grammar.

Note 2: TinyPG will not detect errors inside code blocks, but of course, the .NET compiler will. This can lead to .NET compilation errors that are hard to track and map back onto the grammar code blocks. It may be difficult to see what the issue is. The best way to debug this problem is to open the source code in Visual Studio and trying to compile it.

Note 3: It is also possible to separate the semantics from the syntax by not inserting code blocks directly into the grammar. TinyPG will generate three source code files: the scanner, the parser, and the parsetree. When parsing an input string successfully, the parser will return a filled parsetree. Normally, TinyPG will insert code blocks directly into the parsetree. The parsetree can then be evaluated separately. In this case, we create a subclass of the parsetree and insert our own code there (the methods to implement can be overridden by the subclass). Then, when calling the parser, you supply it with a new instance of your own parsetree. The parser will then fill this parsetree and return it again.

Of course, an alternate manner is to simply evaluate the parsetree directly in the code, by traversing the tree nodes; however, somehow I feel this option is less "clean."

Partial Context Sensitive/Ambiguous Grammars

@TinyPG V1.2 now supports partial ambiguous grammars. Given the simple expression grammar, assume we would like to make a distinction between `FACTOR`s and `TERM`s. The problem is, both `FACTOR`s and `TERM`s are numbers and can be defined as:

Collapse | Copy Code

```
[Color(255, 0, 0)] FACTOR_NUMBER -> @"[0-9]+";    // mark
factors in red
[Color(0, 255, 0)] TERM_NUMBER -> @"[0-9]+";      // mark terms
in green
```

This is typically an ambiguous grammar because a `NUMBER` as input can match both symbols unless, for example, you define your grammar

to only expect a TERM. For example,

Collapse | Copy Code

```
Start -> TERM_NUMBER (MULTDIV) FACTOR_NUMBER;
```

The first input number is expected to be a TERM while the second number is expected to be a FACTOR. Depending on the context (the rule the parser is parsing), the scanner will interpret a number as a TERM or as a FACTOR, respectively. In the example, the first number will be marked green, and the second in red.

Using the code

Once you have generated the Scanner, Parser, ParseTree, and optionally the TextHighlighter classes, and tested them with TinyPG, you obviously now want to use the code in your own project. This can be done by creating a new C# project with Visual Studio. Add the generated files to the project and compile, just to make sure there are no errors.

To call the parser, use the following code:

Collapse | Copy Code

```
#using TinyPG; // add the TinyPG namespace

...

// create the scanner to use
Scanner scanner = new Scanner();

// create the parser, and supply the scanner it should use
Parser parser = new Parser(scanner);

//create a texthighligher (if one was generated)
//and attach the RichTextbox and parser and scanner.
TextHighlighter highlighter =
    new TextHighlighter(richTextbox, scanner, parser);

// define the input for the parser
string input = "... your expression here ...";

// parse the input. the result is a parse tree.
ParseTree tree = parser.Parse(input);
```

Notice that a ParseTree object is returned. The parse tree contains the structure of the input. If the syntax of the input is not correct, the ParseTree will contain errors. You can check for errors by investigating the ParseTree.Errors property.

Notice also that the TextHighlighter accepts a RichTextBox control. TextHighlighter will automatically start capturing its events, analyzing its content, and updating the content of the RichTextBox control.

If all is well, you can go ahead and evaluate the parse tree:

Collapse | Copy Code

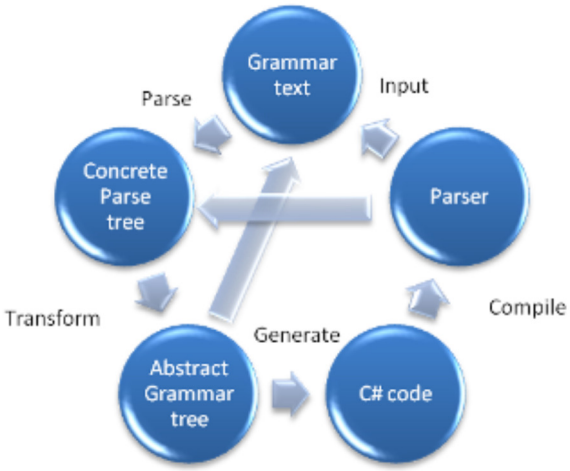
```
// evaluate the parse tree; do not pass any additional
parameters
object result = ParseTree.Eval(null);

// write the result of the evaluation to the console:
Console.WriteLine("result: " + result.ToString());
```

Notice that the `Eval(...)` function returns a result of type `object`. During evaluation, you are free to decide on what the return type will be. This gives you more freedom; however, it also means you will have to cast types explicitly. To display the result, cast it to a string using the `result.ToString()` method.

To conclude, this is all that is needed to build and implement your own grammar and parser. The generated code does not have any external dependencies, nor are any additional libraries required in order to work with the code.

The Tiny Parser Generator evolution cycle

What I have always found intriguing about compiler compilers is that they are able to compile their own grammar and hence generate a compiler from that, which, in this case, is a compiler compiler. Initially, of course, there is no compiler compiler available yet. So how to build one? In this section, I will explain the evolution cycle as I have applied it for TinyPG, as shown in the following graph:



- **Step 1:** Define the grammar as input text to be parsed.
- **Step 2:** Derive a concrete parse tree by parsing the input grammar.
- **Step 3:** Transform the parse tree to an abstract parse tree: the grammar tree.
- **Step 4:** The grammar tree contains all the information about the grammar stored as a tree.
    - **Step 4a:** Generate the grammar text from the grammar tree. This is a check to see if the input grammar corresponds to the grammar in the grammar tree. If they do not correspond, then most likely, the transformation has gone wrong.
    - **Step4b:** Generate the parser C# source code
- **Step 5:** Compile the sources into the actual parser.
- **Next steps:** Take the generated grammar from Step 4a and use it as input for the parser, continue the cycle in Step 2.

To bootstrap the whole process, start with creating the abstract grammar tree manually (in code).

The trickiest parts are the Transformation and Generation processes. Once you have that going, the rest is relatively easy.

Using TinyPG directives

Sometimes you want to be able to set some additional parameters for the tool so it knows how and where to generate the code; for instance, specifying you want to generate C# or VB code. For this purpose, I included the option to insert meta information directly into the grammar by means of directives. I was inspired by the way this is handled by ASPX pages using the `<% ... %>` tags. I decided this would be a handy and compact format that would be strict enough to allow some parameters to be specified, and would be easy to extend at a later stage to add more directives.

Notice that the syntax highlighting for code blocks is now also implemented in V1.2. Code blocks will be highlighted according to the respective `Language` setting of the `@TinyPG` directive.

The directives must be defined before the grammar implementation. Currently, the following directives are supported: `@TinyPG`, `@Parser`, `@Scanner`, and `@ParseTree`, and can be used as follows:

Collapse | Copy Code

```
// Namespace allows you to define a different namespace for the
Scanner,
//     Parser and ParseTree generated classes. By default this
is set to "TinyPG"
// Language allows you to define the language to generate.
// Only C# (default) or VB are supported for now.
// OutputPath allows you to define the absolute or relative
outputpath for the
//     files. The relative path is relative to the grammar
file.
//     By default this is set to "./"
// Template path is relative to the TinyPG.exe file. By default
this is set to
//     "Templates"
<% @TinyPG Namespace="MyNamespace" Language="C#"
           OutputPath="MyGrammarPath"
TemplatePath="MyParserTemplates" %>

// the Generate parameter specifies wether the file
// should be generated. By default this is set to True
<% @Parser Generate="True" %>
<% @ParserTree Generate="False" %>  // turn off generation of
the the ParseTree.cs file
<% @Scanner Generate="True" %>
<% @TextHighlighter Generate="True" %>
// indicates code for the TextHighlighter should be generated
also
```

Some handy Regular Expressions

Writing your own Regular Expressions is not always easy, especially if you want to match tokens that are often used also in programming languages. I have summarized a few Regular Expressions here that can be very helpful.

Collapse | Copy Code

```
// codeblock will match any text between { ... };
Regex codeblock = new Regex(@"\s*\{[^\}]*\}([^};][^}]*\})*;");
```

```
//eof will match the end of an input string only
Regex eof = new Regex(@"^$");

//whitespace will match any whitespace including tabs.
//This one is trivial, but often required.
Regex whitespace = new Regex(@"\s+");

//regex_string will match any text that is a .Net string.
// It also takes the ", "", @ and \ into account
Regex regex_string = new Regex(@"@?\""(\""\""|[^\""])*\""");

//commentline will match with a single line of text that starts
with
// and scan it until the end of the line
//very handy if you want to support commenting by your parser
Regex commentline = new Regex(@"//[^\n]*\n?");

//commentblock will match any text between /* ... */ .
//Very handy if you want to support commenting by your parser
Regex commentblock = new Regex(@"/\*[^*]*\*+(?:[^/*]
[^*]*\*+)*/", RegexOptions.Compiled);
```

That's it for the interesting Regular Expressions I found on the web or wrote myself. If you have any interesting/complicated ones, please drop me a line.

Points of interest

Apart from the parser generator functionality, the TinyPG utility also contains a number of additional components that may be interesting. The controls/features that are new in version 1.2 are made bold:

- **C# or VB code generation.** I have separated the generation code and templates inside the project, and added support for VB.NET as well as C#. It is now possible to create grammars to directly generate either C# or VB code. Inline C# or VB code blocks are supported.
- **Partial support for context sensitive/ambiguous grammars.** Instead of evaluating all possible terminal symbols as possible input of the grammar, I have adapted the `LookAhead` function to only look for the *expected* terminal symbols. So, if, for example, two terminals are defined to match the same input, resulting in an ambiguous grammar, the parser still knows which terminal to look for, based on the rule it is parsing.
- **Code highlighting.** Once you have a parsetree, it is relatively easy to highlight both terminals and non-terminals. Because TinyPG V1.1 code highlighting became rather slow if the text was too long, I decided to make the code highlighting asynchronous in V1.2. This seems to be working rather well; however, when the text is too large (say over 500 lines of code), highlighting may take a little time to finish. Additionally, I have included code highlighting of C# and VB inline code blocks, but unfortunately, no code completion yet.
- **Syntax highlighting.** Because EBNF notation is a language in itself, why not have the syntax and semantics of the grammar checked as you type? Any syntactic or semantic errors found will be highlighted directly in the code by underlining the text in red. Hovering over the error with the mouse will even reveal a tooltip showing what the error is, just as in Visual Studio. The `TextMarker` class is responsible for underlining the erroneous

words. That class can be easily reused in your own projects as it has no dependencies. Add it to your project, wire it up with the `RichtTextbox` control, and assign erroneous words to it. Those will then be automatically marked.

- **Context sensitive code completion.** Depending on the section in which you are typing, code completion will appear with the relevant verbs to complete your typed word. Obviously, this feature was also inspired by Visual Studio. This feature is implemented in the `AutoComplete` form which has no dependencies, and can therefore also be reused. Just add it to your own project, wire it up with a `RichTextbox` control, add the keywords, and you are set to go. Turning auto-completion on or off can be managed through the `Enabled` property of the `AutoCompletion` control.
- `TabControlEx`. I included an extension on the standard .NET `TabControl` called `TabControlEx`. This one does render the tabs correctly when the control is turned upside down, unlike its super-class. The only problem this one has right now is that it does not render the tabs correctly when they are positioned on the left or right side.
- The `RegexControl` is a fully functional drop in to test your Regular Expressions. This is not too exciting though, since there are numerous of these tools out there.
- DockExtender. I reused this code from another project I posted here on CodeProject. It will allow you to drag, drop, and dock the panels on the main window.
- `HeaderLabel` control. This is an extension of the `Label` control. The label is given a gradient background color based on the currently selected Windows theme. It is also possible to activate or deactivate the label, giving it an active or inactive background color. To top it off, I even added the little close 'x' on the label that will activate if you hover over it, essentially creating a caption header much like that used on a form.
- Parse tree. Once you are able to compile your grammar and parse your own expressions, the parse tree will be available. When clicking on nodes in the parse tree, its corresponding tokens will be highlighted in the Expression Evaluator pane, providing a way to browse through the tree and see which tokens are parsed at what point in the parse tree. It may help in debugging potential mistakes in the grammar.

All in all, even though I named this the Tiny Parser Generator, this has become more than a tiny project. I have spent quite some effort in making this all work together nicely, and now I feel this is worth sharing with the community. In a future release, I would like to add the following functionality:

- Option to specify the namespace for the generated classes.
- Support for LL(k) (multiple look aheads); this makes it a bit easier to write grammars; even though LL(1) will produce nicer code.
- Better code highlighting (will probably require partly rewriting the `RichtTextBox` control). This could be a separate project in itself. Any help from the community on this would be greatly appreciated!
- Highlighting of code blocks! This will of course require parsing C#, but hey, we have a parser generator available now! I just need to find the grammar for C#. Anyone? This is still high on

my priority list, but I am getting closer to implementing it.
- Generate for different languages. Because this will most likely require some major reworking on some parts of the code, I will stick with just C# and VB.NET for the time being.
- Perhaps a nicer graphical display of the parse tree (e.g., something as is used in Antlr).
- Display of a state machine like is done in Antlr. That's a nice feature, and may also make the production rules more clear.
- Better error handling and displaying. For example, make it clickable to jump to the position where the error occurred.
- Run the evaluator on a separate thread. If you insert faulty code blocks (for example, endless loops), the tool will currently hang. By running it in a separate thread, this can be controlled.

If you have any ideas for new features, comments, or remarks, please drop a note!

History

@TinyPG v1.0

Tiny Parser Generator Version 1.0 was released on 1st of August 2008. This version includes the basic implementation for generating simple top down parsers.

@TinyPG v1.1

Tiny Parser Generator Version 1.1 was released on 17th of August. Version 1.1 contains additional features, making the editor easier to use:

- Text and syntax highlighting.
- Auto-completion.
- Improved /revised grammar for the EBNF language.
- Support for directives.
- Improved FIRST algorithm.

@TinyPG v1.2

Tiny Parser Generator Version 1.2 was released on 1st of September. Version 1.2 contains additional features, making the editor easier to use:

- Generation of C# and/or VB code (!)
- Allows for context partial sensitive/ambiguous grammars.
- Code block highlighting for C# or VB code.
- Attributes now allow parameters.
- Color(R,G,B) attribute added.
- Generation of texthighlighter code.
- Asynchronous text and syntax highlighting.

@TinyPG v1.3

Tiny Parser Generator Version 1.3 was released on 19th of September

2010. Version 1.3 is a minor upgrade, and mainly fixes some issues:

- Syntax highlighting issues with skipped tokens have been addressed. Any skipped tokens are now appended to the next actual token (`Token.Skipped`). Any trailing skipped tokens are kept within the scanner (`Scanner.Skipped`).
- Added support for Unicode in the `TextHighlighter`.
- Added serialization features to the ParseTree output. So it is now possible to save it to XML and query it using, e.g., XPath. This may come in handy in scenarios where you want to divide the parsing process from the interpretation process.
- Fixed bug with highlighting in the TinyPG editor.

Special thanks go out to William A. McKee for being actively involved with @TinyPG, inspiring me to improve the tool further, and helping me revise the EBNF grammar. I have implemented some of his ideas in @TinyPG v1.2, including support for (partly) ambiguous grammars. Also, I want to thank Vicor Morgante for his dedication on using the tool and helping me further to improve.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

Herre Kuijpers
Architect Capgemini

Netherlands
Member

Currently Herre Kuijpers is employed at Capgemini Netherlands for over 10 years, where he developed skills with all kinds of technologies, methodologies and programming languages such as c#, ASP.Net, Silverlight, VC++, Javascript, SQL, UML, RUP, WCF. Currently he fulfills the role of software architect in various projects.

Herre Kuijpers is a very experienced software architect with deep knowledge of software design and development on the Microsoft .Net platform. He has a broad knowledge of Microsoft products and knows how these, in combination with custom software, can be optimally implemented in the often complex environment of the customer.

Article Top Like 0 1 Tweet 7
Rate this: *Poor* *Excellent* Vote

Comments and Discussions

Add a Comment or Question

Search this forum Go

Profile popups Spacing Relaxed Noise Medium Layout Normal

Per page 25 Update

First Prev Next

My vote of 5	**Y.F. Chuang**	**3 Apr '13 - 21:24**
how to commit new features	**Jissai**	**7 Mar '13 - 0:27**
Re: how to commit new features	**pingz**	7 Mar '13 - 3:46
Is there a way to sort on mutiple columns	**HOASTONGE**	**5 Jan '13 - 2:48**
My vote of 5	**HawVie**	**30 Dec '12 - 6:35**
Great Project ... but im stuck on an ugly thing ..	**Cedric Decoster**	**30 Nov '12 - 1:14**
Re: Great Project ... but im stuck on an ugly thing ..	**Herre Kuijpers**	30 Nov '12 - 17:26
TinyPG on GitHub	**pingz**	**18 Nov '12 - 8:14**
Re: TinyPG on GitHub	**pingz**	27 Dec '12 - 2:41
Excelent! But how to skip whitespaces except when only thing between quotes?	**lootNinja**	**9 Aug '12 - 0:07**
Re: Excelent! But how to skip whitespaces except when only thing between quotes?	**Herre Kuijpers**	9 Aug '12 - 20:31

Re: Excelent! But how to skip whitespaces except when only thing between quotes?	lootNinja	9 Aug '12 - 21:17
Re: Excelent! But how to skip whitespaces except when only thing between quotes?	lootNinja	9 Aug '12 - 21:11
My vote of 5	**Ghazi Sarhan**	**24 Jun '12 - 4:27**
Great work. 5! but...	**gtrotta**	**8 Jun '12 - 20:02**
Re: Great work. 5! but...	**Herre Kuijpers**	9 Jun '12 - 19:28
Re: Great work. 5! but...	**gtrotta**	10 Jun '12 - 0:30
Re: Great work. 5! but...	**Herre Kuijpers**	10 Jun '12 - 1:16
Re: Great work. 5! but...	**gtrotta**	10 Jun '12 - 20:31
Add 'var' Support	**pingz**	**20 May '12 - 8:21**
Re: Add 'var' Support	**pingz**	18 Nov '12 - 9:36
Removing Leading Whitespace?	**pingz**	**19 May '12 - 23:04**
Awesome work... and one improvement... and one question... and one suggestion	**Jorge Varas**	**7 Apr '12 - 10:00**
Re: Awesome work... and one improvement... and one question... and	**androidandy**	11 May '12 - 10:06

one suggestion

Re: Awesome work... and one improvement... and one question... and one suggestion **Jorge Varas** 12 May '12 - 7:51

Last Visit: 18 Apr '13 - 15:56
Last Update: 18 Apr '13 - 12:39 Refresh **1** 2 3 4 5 6 7 8 9 10 11 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink | Advertise | Privacy | Mobile
Web04 | 2.6.130416.1 | Last Updated 21 Sep 2010

Layout: fixed | fluid

Article Copyright 2008 by Herre Kuijpers
Everything else Copyright © CodeProject, 1999-2013
Terms of Use