Toy Parser Generator
or
How to easily write parsers in Python

Christophe Delord
christophe.delord@free.fr
http://christophe.delord.free.fr/en/tpg/

January 26, 2003

Contents

I Introduction and tutorial 7

1 Introduction 8
- 1.1 Introduction . . . 8
- 1.2 License . . . 8
- 1.3 Structure of the document . . . 8

2 Installation 10
- 2.1 Getting TPG . . . 10
- 2.2 Requirements . . . 10
- 2.3 TPG for Linux and other Unix like . . . 10
- 2.4 TPG for M$ Windows . . . 10
- 2.5 TPG for other operating systems . . . 10

3 Tutorial 11
- 3.1 Introduction . . . 11
- 3.2 Defining the grammar . . . 11
- 3.3 Reading the input and returning values . . . 12
- 3.4 Embeding the parser in a script . . . 13
- 3.5 Conclusion . . . 16

II TPG reference 17

4 Usage 18
- 4.1 Package content . . . 18
- 4.2 Command line usage . . . 19

5 Grammar structure 20
- 5.1 TPG grammar structure . . . 20
- 5.2 Comments . . . 21
- 5.3 Options . . . 21
  - 5.3.1 Magic option . . . 21
  - 5.3.2 CSL options . . . 21
- 5.4 Python code . . . 21
  - 5.4.1 Syntax . . . 21
  - 5.4.2 Indentation . . . 21
- 5.5 TPG parsers . . . 22
  - 5.5.1 Initialisation . . . 22
  - 5.5.2 Rules . . . 22
  - 5.5.3 Python code . . . 22

6 Lexer **23**
6.1 Regular expression syntax . . . 23
6.2 Token definition . . . 23
6.2.1 Predefined tokens . . . 23
6.2.2 Inline tokens . . . 24
6.3 Token matching . . . 24
6.3.1 Splitting the input string . . . 24
6.3.2 Matching tokens in grammar rules . . . 24

7 Parser **26**
7.1 Declaration . . . 26
7.2 Base classes of TPG parsers . . . 26
7.2.1 Default base class . . . 26
7.2.2 User defined base classes . . . 26
7.3 Grammar rules . . . 26
7.4 Parsing terminal symbols . . . 27
7.5 Parsing non terminal symbols . . . 27
7.5.1 Starting the parser . . . 27
7.5.2 In a rule . . . 27
7.6 Sequences . . . 27
7.7 Cut . . . 28
7.8 Alternatives . . . 28
7.9 Repetitions . . . 28
7.10 Precedence and grouping . . . 28
7.11 Actions . . . 28
7.11.1 Abstract syntax trees . . . 29
7.11.2 Text extraction . . . 30
7.11.3 Object . . . 30
7.11.4 Actions in Python code . . . 31

8 Context sensitive lexer **34**
8.1 Introduction . . . 34
8.2 Grammar structure . . . 34
8.3 CSL lexers . . . 34
8.3.1 Regular expression syntax . . . 34
8.3.2 Token definition . . . 34
8.3.3 Token matching . . . 35
8.4 CSL parsers . . . 35

III Some examples to illustrate TPG **37**

9 Complete interactive calculator **38**
9.1 Introduction . . . 38
9.2 New functions . . . 38
9.2.1 Trigonometric and other functions . . . 38
9.2.2 Memories . . . 38
9.3 Source code . . . 38
9.3.1 TPG grammar . . . 38
9.3.2 Python script . . . 40

10 Infix/Prefix/Postfix notation converter **43**
10.1 Introduction . . . 43
10.2 Abstract syntax trees . . . 43
10.3 Grammar . . . 43
10.3.1 Infix expressions . . . 43
10.3.2 Prefix expressions . . . 44
10.3.3 Postfix expressions . . . 44
10.4 Source code . . . 44

IV Internal structure of TPG for the curious **47**

11 Structure of the package **48**
11.1 General structure of the package . . . 48

12 Lexer **49**
12.1 Token matching . . . 49

13 Parser **50**
13.1 Interface with the lexer . . . 50
13.2 Sequences of subexpressions . . . 50
13.3 Alternatives between subexpressions . . . 50
13.4 Repetitions . . . 50

14 Code generation **51**
14.1 Inheritance . . . 51
14.2 Lexer . . . 52
14.3 Parser . . . 53
14.3.1 Grammar rules . . . 53
14.3.2 Symbols . . . 54
14.3.3 Sequences . . . 55
14.3.4 Cut . . . 56
14.3.5 Alternatives . . . 57
14.3.6 Repetitions . . . 58
14.3.7 Abstract syntax trees . . . 60
14.3.8 Text extraction . . . 61
14.3.9 Python objects . . . 61

List of Figures

3.1 Grammar for expressions . . . 11
3.2 Terminal symbol definition for expressions . . . 12
3.3 Grammar of the expression recognizer . . . 12
3.4 *make_op* function . . . 13
3.5 Token definitions with functions . . . 13
3.6 Return values for (non) terminal symbols . . . 13
3.7 Expression recognizer and evaluator . . . 14
3.8 Python code generation from a grammar . . . 14
3.9 Complete Python script with expression parser . . . 15

4.1 Grammar embeding example . . . 18
4.2 Parser compilation example . . . 19
4.3 Parser usage example . . . 19

5.1 TPG grammar structure . . . 20
5.2 Code indentation examples . . . 22

6.1 Token definition examples . . . 23
6.2 Inline token definition examples . . . 24
6.3 Token usage examples . . . 24
6.4 Token usage examples . . . 25

7.1 User defined base classes for TPG parsers . . . 26
7.2 Rule declaration . . . 27
7.3 Precedence in TPG expressions . . . 28
7.4 AST example . . . 29
7.5 AST update example . . . 29
7.6 Backtracking with *WrongMatch* example . . . 32
7.7 Backtracking with the *check* method example . . . 32
7.8 Backtracking with the *check* keyword example . . . 33
7.9 Error reporting the *error* method example . . . 33
7.10 Error reporting the *error* keyword example . . . 33

8.1 Token definition in CSL parsers example . . . 34
8.2 Separator definition in CSL parsers examples . . . 35
8.3 Token usage in CSL parsers examples . . . 35

14.1 Inheritance example . . . 51
14.2 Lexer example . . . 52
14.3 Rule declaration example . . . 53
14.4 Terminal symbol matching example . . . 54
14.5 Non terminal symbol matching example . . . 54
14.6 Sequence of expressions example . . . 55

14.7 Cut example 56
14.8 Alternative in expressions example 57
14.9 Repetition examples: builtin ?, * and + ... 58
14.10 Repetition examples: user defined {m,n} 59
14.11 AST instanciation example 60
14.12 AST update example 60
14.13 Text extraction 61
14.14 Python object in TPG 61

Part I

Introduction and tutorial

Chapter 1

Introduction

1.1 Introduction

TPG (Toy Parser Generator) is a Python[1] parser generator. It is aimed at easy usage rather than performance. My inspiration was drawn from two different sources. The first was GEN6. GEN6 is a parser generator created at ENSEEIHT[2] where I studied. The second was PROLOG[3], especially DCG[4] parsers. I wanted a generator with a simple and expressive syntax and the generated parser should work as the user expects. So I decided that TPG should be a recursive descendant parser (a rule is a procedure that calls other procedures) and the grammars are attributed (attributes are the parameters of the procedures). This way TPG can be considered as a programming language or more modestly as Python extension.

1.2 License

TPG is available under the GNU Lesser General Public.

> Toy Parser Generator: A Python parser generator
>
> Copyright (C) 2002 Christophe Delord
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

1.3 Structure of the document

Part I starts smoothly with a gentle tutorial as an introduction. I think this tutorial may be sufficent to start with TPG.

[1]Python is a wonderful object oriented programming language available at http://www.python.org

[2]ENSEEIHT is a french engineer school (http://www.enseeiht.fr).

[3]PROLOG is a programming language using logic. My favorite PROLOG compiler is SWI-PROLOG (http://www.swi-prolog.org).

[4]Definite Clause Grammars.

Part II is a reference documentation. It will detail TPG as much as possible.

Part III gives the reader some examples to illustrate TPG.

Part IV is an explanation of how TPG works internally. It details the predictive algorithm and shows the generated code. It is not needed to read this part but it can help to understand how TPG works or why some grammars fail.

Chapter 2

Installation

2.1 Getting TPG

TPG is freely available on its web page (http://christophe.delord.free.fr/en/tpg). It is distributed as a package using *distutils*[1].

2.2 Requirements

TPG is a *pure Python* package. It may run on *any platform* supported by Python. The only requirement of TPG is *Python 2.2* or newer. Python can be downloaded at http://www.python.org.

2.3 TPG for Linux and other Unix like

Download *TPG-X.Y.Z.tar.gz*, unpack and run the installation program:

```
tar xzf TPG-X.Y.Z.tar.gz
cd TPG-X.Y.Z
python setup.py install
```

You may need to be logged as root to install TPG.

2.4 TPG for M$ Windows

Download *TPG-X.Y.Z.win32.exe* and run it.

2.5 TPG for other operating systems

TPG should run on any system provided that Python is installed. You should be able to install it by running the *setup.py* script (see 2.3).

[1]distutils is a Python package used to distribute Python softwares

Chapter 3

Tutorial

3.1 Introduction

This short tutorial presents how to make a simple calculator. The calculator will compute basic mathematical expressions (`+, -, *, /`) possibly nested in parenthesis. We assume the reader is familiar with regular expressions.

3.2 Defining the grammar

Expressions are defined with a grammar. For example an expression is a term, a term is a sum of factors and a factor is a product of atomic expressions. An atomic expression is either a number or a complete expression in parenthesis.

We describe such grammars with rules. A rule describe the composition of an item of the language. In our grammar we have 3 items (Term, Factor, Atom). We will call these items 'symbols' or 'non terminal symbols'. The decomposition of a symbol is symbolized with $\rightarrow$. The grammar of this tutorial is given in figure 3.1.

Figure 3.1: Grammar for expressions

Grammar rule	Description
$Term \rightarrow Factor\ ((\prime+\prime\vert\prime-\prime)\ Factor)*$	A term is a factor eventually followed with a plus ($\prime+\prime$) or a minus ($\prime-\prime$) sign and an other factor any number of times ($*$ is a repetition of an expression 0 or more times).
$Factor \rightarrow Atom\ ((\prime*\prime\vert\prime/\prime)\ Atom)*$	A factor is an atom eventually followed with a $\prime*\prime$ or $\prime/\prime$ sign and an other atom any number of times.
$Atom \rightarrow number \mid \prime(\prime\ Term\ \prime)\prime$	An atomic expression is either a number or a term in parenthesis.

We have defined here the grammar rules (i.e. the sentences of the language). We now need to describe the lexical items (i.e. the words of the language). These words - also called *terminal symbols* - are described using regular expressions. In the rules we have written some of these terminal symbols $(+, -, *, /, (,))$. We have to define *number*. For sake of simplicity numbers are integers composed of digits (the corresponding regular expression can be $[0-9]+$). To simplify the grammar and then the Python script we define two terminal symbols to group the operators (additive and multiplicative operators). We can also define a special symbol that is ignored by

TPG. This symbol is used as a separator. This is generaly usefull for white spaces and comments. The terminal symbols are given in figure 3.2

Figure 3.2: Terminal symbol definition for expressions

Terminal symbol	Regular expression	Comment
number	$[0-9]+$ or $\backslash d+$	One or more digits
add	$[+-]$	a $+$ or a $-$
mul	$[*/]$	a $*$ or a $/$
spaces	$\backslash s+$	One or more spaces

This is sufficient to define our parser with TPG. The grammar of the expressions in TPG can be found in figure 3.3.

Figure 3.3: Grammar of the expression recognizer

```
parser Calc:

    separator spaces: '\s+' ;
    token number: '\d+' ;
    token add: '[+-]' ;
    token mul: '[*/]' ;

    START -> Term ;

    Term -> Fact ( add Fact )* ;

    Fact -> Atom ( mul Atom )* ;

    Atom -> number | '\(' Term '\)' ;
```

Calc is the name of the Python class generated by TPG. *START* is a special non terminal symbol treated as the *axiom*[1] of the grammar.

With this small grammar we can only recognize a correct expression. We will see in the next sections how to read the actual expression and to compute its value.

3.3 Reading the input and returning values

The input of the grammar is a string. To do something useful we need to read this string in order to transform it into an expected result.

This string can be read by catching the return value of terminal symbols. By default any terminal symbol returns a string containing the current token. So the token '(' always returns the string '('. For some tokens it may be useful to compute a Python object from the token. For example *number* should return an integer instead of a string, *add* and *mul* should return a function corresponding to the operator. That why we will add a function to the token definitions. So we associate *int* to *number* and *make_op* to *add* and *mul*.

int is a Python function converting objects to integers and *make_op* is a user defined function (figure 3.4).

To associate a function to a token it must be added after the token definition as in figure 3.5

[1]The axiom is the symbol from which the parsing starts

Figure 3.4: *make_op* function

```
def make_op(s):
    return {
        '+': lambda x,y: x+y,
        '-': lambda x,y: x-y,
        '*': lambda x,y: x*y,
        '/': lambda x,y: x/y,
    }[s]
```

Figure 3.5: Token definitions with functions

```
    separator spaces: '\s+' ;
    token number: '\d+' int ;
    token add: '[+-]' make_op;
    token mul: '[*/]' make_op;
```

We have specified the value returned by the token. To read this value after a terminal symbol is recognized we will store it in a Python variable. For example to save a *number* in a variable *n* we write *number/n*. In fact terminal and non terminal symbols can return a value. The syntax is the same for both sort of symbols. In non terminal symbol definitions the return value defined at the left hand side is the expression return by the symbol. The return values defined in the right hand side are just variables to which values are saved. A small example may be easier to understand (figure 3.6).

Figure 3.6: Return values for (non) terminal symbols

Rule	Comment
`X/x ->`	Defines a symbol X. When X is called, x is returned.
`Y/y`	X starts with a Y. The return value of Y is saved in y.
`Z/z`	The return value of Z is saved in z.
`{{ x = y+z }}`	Computes x.
`;`	Returns x.

In the example described in this tutorial the computation of a *Term* is made by applying the operator to the factors, this value is then returned :

```
Term/t -> Fact/t ( add/op Fact/f {{ t = op(t,f) }} )* ;
```

This example shows how to include Python code in a rule. Here `{{...}}` is copied verbatim in the generated parser.

Finally the complete parser is given in figure 3.7.

3.4 Embeding the parser in a script

To embed a TPG parser in a Python program, you only need the *tpg.compile* function. This function takes a grammar (in a string[2]) and returns the Python object code of the parser. If you

[2] It may be a good pratice to use only raw strings. This will ease the pain of writing regular expressions.

Figure 3.7: Expression recognizer and evaluator

```
parser Calc:

    separator spaces: '\s+' ;

    token number: '\d+' int ;
    token add: '[+-]' make_op ;
    token mul: '[*/]' make_op ;

    START -> Term ;

    Term/t -> Fact/t ( add/op Fact/f {{ t = op(t,f) }} )* ;

    Fact/f -> Atom/f ( mul Atom/a {{ f = op(f,a) }} )* ;

    Atom/a -> number/a | '\(' Term/a '\)' ;
```

need the Python source code of the parser you can call the *tpg.translate* function. One way to use this parser is to *exec* its definition. A practical way to build parsers is to *exec* the result of *tpg.compile* (figure 3.8).

Figure 3.8: Python code generation from a grammar

```
import tpg

exec(tpg.compile(r""" # Your grammar here """))

# You can instanciate your parser here
```

To use this parser you now just need to import tpg, compile the grammar and instanciate an object of the class *Calc* as in figure 3.9.

Figure 3.9: Complete Python script with expression parser

```
import tpg

def make_op(s):
    return {
        '+': lambda x,y: x+y,
        '-': lambda x,y: x-y,
        '*': lambda x,y: x*y,
        '/': lambda x,y: x/y,
    }[s]

exec(tpg.compile(r"""

parser Calc:

    separator spaces: '\s+' ;

    token number: '\d+' int ;
    token add: '[+-]' make_op ;
    token mul: '[*/]' make_op ;

    START/e -> Term/e ;
    Term/t -> Fact/t ( add/op Fact/f {{ t = op(t,f) }} )* ;
    Fact/f -> Atom/f ( mul/op Atom/a {{ f = op(f,a) }} )* ;
    Atom/a -> number/a | '\(' Term/a '\)' ;

"""))

calc = Calc()
expr = raw_input('Enter an expression: ')
print expr, '=', calc(expr)
```

3.5 Conclusion

This tutorial shows some of the possibilities of TPG. If you have read it carefully you may be able to start with TPG. The next chapters present TPG more precisely. They contain more examples to illustrate all the features of TPG.

Happy TPG'ing!

Part II

TPG reference

Chapter 4

Usage

4.1 Package content

TPG is a package which main function is to take a grammar and return a parser[1]. You only need to import TPG and use these four objects:

tpg.compile(grammar): This function takes a grammar in a string and produces the Python object code of the parser. You can call exec to actually build it.

tpg.translate(grammar): This function takes a grammar in a string and produces the Python source code of the parser. You can call exec to actually build it.

tpg.LexerError: This exception is raised when the lexer fails.

tpg.ParserError: This exception is raised when the parser fails.

tpg.SemanticError: This exception is raised by the grammar itself when some semantic properties fail.

The grammar must be in a string (see figure 4.1).

Figure 4.1: Grammar embeding example

```
my_grammar = r"""

parser Foo:

    START/x -> Bar/x .

    Bar/x -> 'bar'/x .

"""
```

The *tpg.compile* function produces Python code from the grammar (see figure 4.2).

Then you can use the new generated parser. The parser is now simply a Python class (see figure 4.3).

[1] More precisely it returns the Python source code of the parser

Figure 4.2: Parser compilation example

```
exec(tpg.compile(my_grammar))    # Compiles my_grammar
```

Figure 4.3: Parser usage example

```
test = "bar"
my_parser = Foo()
x = my_parser(test)               # Uses the START symbol
print x
x = my_parser.parse('Bar', test)  # Uses the Bar symbol
print x
```

4.2 Command line usage

The *tpg* script is just a wrapper for the package. It reads a grammar in a file and write the generated code in a Python script. To produce a Python script from a grammar you can use *tpg* as follow:

```
tpg [-v|-vv] grammar.g [-o parser.py]
```

tpg accepts some options on the command line:

-v turns *tpg* into a verbose mode (it displays parser names).

-vv turns *tpg* into a more verbose mode (it displays parser names and simplified rules).

-o file.py tells *tpg* to generate the parser in *file.py*. The default output file is *grammar.py* if -o option is not provided and *grammar.g* is the name of the grammar.

Chapter 5

Grammar structure

5.1 TPG grammar structure

TPG grammars may contain three parts:

Options are defined at the beginning of the grammar (see 5.3).

Parsers are described in sections starting with the *parser* keyword (see 5.5).

Python codes can appear in sections starting with the *main* keyword or before the first parser (see 5.4).

See figure 5.1 for a generic TPG grammar.

Figure 5.1: TPG grammar structure

```
# Options
set magic = "/usr/bin/env python"

# Python code
{{
    class MyClass:
        pass
}}

# Parser Foo
parser Foo:

    START -> X Y Z ;

# More Python code
main:
{{
    def myfunction:
        pass
}}
```

5.2 Comments

Comments in TPG start with # and run until the end of the line.

```
# This is a comment
```

5.3 Options

Some options can be set at the beginning of TPG grammars. The syntax for options is:

set *name* sets the boolean *name* option to *true*.

set *name* = "*value*" sets the *name* option to *value*.

set no*name* disables the *name* option.

5.3.1 Magic option

The magic option tells TPG which interpreter is called when the script is run. The first line of the generated code will start with `#!` and contains the command line to execute the appropriate interpreter (`/usr/bin/env python` for example). This has no effect on M$ Windows.

set magic = "/usr/bin/env python" adds `#!/usr/bin/env python` to the first line.

set nomagic generates no magic line. This is the default behaviour.

5.3.2 CSL options

By default TPG lexers are context free. The *CSL* option tells TPG to generate a context sensitive lexer (see 8).

set CSL generates context sensitive lexers.

set noCSL generates context free lexers. This is the default behaviour.

5.4 Python code

Python code section are not handled by TPG. TPG won't complain about syntax errors in Python code sections, it is Python's job. They are copied verbatim to the generated Python parser.

5.4.1 Syntax

Python code is enclosed in double curly brackets. That means that Python code must not contain to consecutive close brackets. You can avoid this by writting } } (with a space) instead of }} (without space).

5.4.2 Indentation

Python code can appear in several parts of a grammar. Since indentation has a special meaning in Python it is important to know how TPG handles spaces and tabulations at the beginning of the lines. In TPG indentation is important only in Python code sections (in *main* parts, in *parser* parts and in *rules*).

When TPG encounters some Python code it removes in all non blank lines the spaces and tabulations that are common to every lines. TPG considers spaces and tabulations as the same character so it is important to always use the same indentation style. Thus it is advised not to mix spaces and tabulations in indentation. Then this code will be reindented when generated according to its location (in a class, in a method or in global space).

The figure 5.2 shows how TPG handles indentation.

Figure 5.2: Code indentation examples

Code in grammars	Generated code	Comment

`{{`
`␣␣␣␣if␣1==2:`
`␣␣␣␣␣␣␣␣print␣"???"`
`␣␣␣␣else:`
`␣␣␣␣␣␣␣␣print␣"OK"`
`}}`
	`if␣1==2:`
	`␣␣␣␣print␣"???"`
	`else:`
	`␣␣␣␣print␣"OK"`
		Correct: these lines have four spaces in common. These spaces are removed.

`{{␣␣if␣1==2:`
`␣␣␣␣␣␣␣␣print␣"???"`
`␣␣␣␣else:`
`␣␣␣␣␣␣␣␣print␣"OK"`
`}}`
	`if␣1==2:`
	`␣␣␣␣␣␣print␣"???"`
	`␣␣else:`
	`␣␣␣␣␣␣print␣"OK"`
		WRONG: it's a bad idea to start a multiline code section on the first line since the common indentation may be different from what you expect. No error will be raised by TPG but Python won't compile this code.

`{{␣␣␣␣␣␣␣print␣"OK"␣}}`	`print␣"OK"`	*Correct*: indentation does not matter in a one line Python code.

5.5 TPG parsers

A grammar can contain as many parsers as needed. A parser declaration starts with the *parser* keyword and contains rules and Python code sections (local to the parser).

5.5.1 Initialisation

The initialisation of Python objects is made by the *__init__* method. This method is generated by TPG and cannot be overriden. To resolve this problem an *init* method (i.e. without the double underscores) is called at initialization time with the arguments given to *__init__*. See 5.5.3 to add methods to a parser.

5.5.2 Rules

Each rule will be translated into a method of the parser.

5.5.3 Python code

Python code that is local to a parser will be copied in the generated class. This is usually used to add methods or attributes to the parser.

Chapter 6

Lexer

6.1 Regular expression syntax

The lexer is based on the *re*[1] module. TPG profits from the power of Python regular expressions. This document assumes the reader is familiar with regular expressions.

You can use the syntax of regular expressions as expected by the *re* module except from the grouping syntax since it is used by TPG to decide which token is recognized.

6.2 Token definition

6.2.1 Predefined tokens

Tokens can be explicitely defined by the *token* and *separator* keywords.

A token is defined by:

a name which identifies the token. This name is used by the parser.

a regular expression which describes what to match to recognize the token.

an action which can translate the matched text into a Python object. It can be a function of one argument or a non callable object. It it is not callable, it will be returned for each token otherwise it will be applied to the text of the token and the result will be returned. This action is optional. By default the token text is returned.

Token definitions end with a *;* .

See figure 6.1 for examples.

Figure 6.1: Token definition examples

```
#     name      reg. exp          action
token integer: '\d+'              int;
token ident  : '[a-zA-Z]\w*'      ;

separator spaces   : '\s+';       # white spaces
separator comments: '#.*';        # comments
```

[1] *re* is a standard Python module. It handles regular expressions. For further information about *re* you can read http://python.org/doc/2.2/lib/module-re.html

The order of the declaration of the tokens is important. The first token that is matched is returned. The regular expression has a special treatment. If it describes a keyword, TPG also looks for a word boundary after the keyword. If you try to match the keywords *if* and *ifxyz* TPG will internally search `if\b` and `ifxyz\b`. This way, *if* won't match *ifxyz* and won't interfere with general identifiers (`\w+` for example).

There are two kinds of tokens. Tokens defined by the *token* keyword are parsed by the parser and tokens defined by the *separator* keyword are considered as separators (white spaces or comments for example) and are wiped out by the lexer.

6.2.2 Inline tokens

Tokens can also be defined on the fly. Their definition are then inlined in the grammar rules. This feature may be useful for keywords or punctuation signs. Inline tokens can not be transformed by an action as predefined tokens. They always return the token in a string.

See figure 6.2 for examples.

Figure 6.2: Inline token definition examples

```
IfThenElse ->
    'if' Cond
    'then' Statement
    'else' Statement
    ;
```

Inline tokens have a higher precedence than predefined tokens to avoid conflicts (an inlined *if* won't be matched as a predefined *identifier*).

6.3 Token matching

TPG works in two stages. The lexer first splits the input string into a list of tokens and then the parser parses this list.

6.3.1 Splitting the input string

The lexer split the input string according to the token definitions (see 6.2). When the input string can not be matched a *tpg.LexerError* exception is raised.

The lexer may loop indefinitely if a token can match an empty string since empty strings are everywhere.

6.3.2 Matching tokens in grammar rules

Tokens are matched as symbols are recognized. Predefined tokens have the same syntax than non terminal symbols. The token text (or the result of the function associated to the token) can be saved by the infix / operator (see figure 6.3).

Figure 6.3: Token usage examples

```
S -> ident/i;
```

Inline tokens have a similar syntax. You just write the regular expression (in a string). Its text can also be save (see figure 6.4).

Figure 6.4: Token usage examples

```
S -> '(' '\w+'/i ')';
```

Chapter 7

Parser

7.1 Declaration

A parser is declared with the *parser* keyword. The declaration may have a list of base classes from which the parser will inherit. Then follows grammar rules and code sections.

7.2 Base classes of TPG parsers

TPG parsers can inherit from other Python classes.

7.2.1 Default base class

TPG parsers always inherits from the *tpg.base.ToyParser* class which defines the common behaviour of every parsers.

7.2.2 User defined base classes

The user can add more base classes to TPG parsers by adding a class list to the parser definition as in figure 7.1.

Figure 7.1: User defined base classes for TPG parsers

```
parser MyParser(BaseClass1, BaseClass2):
    ...
```

7.3 Grammar rules

Rule declarations have two parts. The left side declares the symbol associated to the rule, its attributes and its return value. The right side describes the decomposition of the rule. Both parts of the declaration are separated with an arrow ($\rightarrow$) and the declaration ends with a *;*.

The symbol defined by the rule as well as the symbols that appear in the rule can have attributes and return values. The attribute list - if any - is given as an object list enclosed in left and right angles. The return value - if any - is extracted by the infix */* operator. See figure 7.2 for example.

Figure 7.2: Rule declaration

```
SYMBOL <att1, att2, att3> / return_expression_of_SYMBOL ->

    A <x, y> / ret_value_of_A

    B <y, z> / ret_value_of_B

    ;
```

7.4 Parsing terminal symbols

Each time a terminal symbol is encountered in a rule, the parser compares it to the current token in the token list. If it is different the parser backtracks.

7.5 Parsing non terminal symbols

7.5.1 Starting the parser

You can start the parser from the axiom or from any other non terminal symbol. When the parser can not parse the whole token list a *tpg.ParserError* is raised. The value returned by the parser is the return value of the parsed symbol.

From the axiom

The axiom is a special non terminal symbol named *START*. Parsers are callable objects. When an instance of a parser is called, the *START* rule is parsed. The first argument of the call is the string to parse. The other arguments of the call are given to the *START* symbol.

This allows to simply write `x=calc("1+1")` to parse and compute an expression if *calc* is an instance of an expression parser.

From another non terminal symbol

It's also possible to start parsing from any other non terminal symbol. TPG parsers have a method named *parse*. The first argument is the name of the symbol to start from. The second argument is the string to parse. The other arguments are given to the specified symbol.

For example to start parsing a *Factor* you can write:

```
f=calc.parse('Factor', "2*3")
```

7.5.2 In a rule

To parse a non terminal symbol in a rule, TPG call the rule corresponding to the symbol.

7.6 Sequences

Sequences in grammar rules describe in which order symbols should appear in the input string. For example the sequence *A B* recognizes an *A* followed by a *B*. Sequences can be empty.

For example to say that a *sum* is a *term plus* another *term* you can write:

```
Sum -> Term '+' Term ;
```

7.7 Cut

The cut idiom is drawn from the Prolog cut (*!*). When the *!* operator is encountered it is ignored. When TPG backtracks on a cut, a syntax error is raised so as to *cut* other possible alternatives.

For example, the rule `R -> a ! b c | d ;` will raise a *ParserError* exception if it recognizes an a not followed by a b and a c, without trying to parse a d.

The cut also helps TPG to report errors. In the previous example, TPG will report an error after a instead of backtracking to the topmost rule.

7.8 Alternatives

Alternatives in grammar rules describe several possible decompositions of a symbol. The infix pipe operator (|) is used to separate alternatives. $A \mid B$ recognizes either an A or a B. If both A and B can be matched only the first match is considered. So the order of alternatives is very important. If an alternative has an empty choice, it must be the last.

For example to say that an *atom* is an *integer* or an *expression in paranthesis* you can write:

```
Atom -> integer | '\(' Expr '\)' ;
```

7.9 Repetitions

Repetitions in grammar rules describe how many times an expression should be matched.

A? recognizes zero or one A.

A* recognizes zero or more A.

A+ recognizes one or more A.

A{m,n} recognizes at least m and at most n A.

Repetitions are greedy. Repetitions are translated into Python loops. Thus whatever the length of the repetitions, Python stack will not overflow.

7.10 Precedence and grouping

The figure 7.3 lists the different structures in increasing precedence order. To override the default precedence you can group expressions with parenthesis.

Figure 7.3: Precedence in TPG expressions

Structure	Example
Alternative	$A \mid B$
Cut	$A\ !\ B$
Sequence	$A\ B$
Repetitions	$A?$, $A*$, $A+$
Symbol and grouping	A and (...)

7.11 Actions

Grammar rules can contain actions and Python code. Actions are handled by TPG and Python code is copied verbatim into the generated code.

7.11.1 Abstract syntax trees

An abstract syntax tree (AST) is an abstract representation of the structure of the input. A node of an AST is a Python object (there is no constraint about its class). AST nodes are completely defined by the user.

The figure 7.4 shows a node symbolizing a couple.

Figure 7.4: AST example

```
{{
    class Couple:
        def __init__(self, a, b):
            self.a = a
            self.b = b
}}

parser Foo:

    COUPLE -> '(' ITEM/a ',' ITEM/b ')' c = Couple<a,b> ;

    COUPLE/Couple<a,b> -> '(' ITEM/a ',' ITEM/b ')' ;
```

Creating an AST

AST can be created by the infix = operator (see figure 7.11.1).

Updating an AST

When parsing lists for example it is useful to save all the items of the list. The infix - operator call the *add* method of an AST (see figure 7.5). This method is defined by the user. TPG won't check that the class actually has an *add* method.

Figure 7.5: AST update example

```
{{
    class List(list):
        add = list.append
}}

parser ListParser:

    LIST/l ->
        '('
            l = List<>
            ITEM/a l-a
            ( ',' ITEM/a l-a )*
        ')'
        ;
```

7.11.2 Text extraction

TPG can extract a portion of the input string. The idea is to put marks while parsing and then extract the text between the two marks. This extracts the whole text between the marks, including the tokens defined as separators.

7.11.3 Object

TPG knows some basics about Python objects. An object in TPG is a Python object using a special syntax. The use of parenthesis has been rejected because it would have introduced ambiguities in the TPG grammar. *Parenthesis have been replaced with left and right angles (< and >).* Appart from this particularity, TPG object syntax is a subset of the Python syntax.

An object can be:

- an identifier
- a string
- a tuple
- a code object (in double curly brackets)
- a text extraction (infix .. operator)
- an acces to an attribute (infix . operator)
- a call to a method or a function
- a slice operation

Identifier

No mystery about identifiers except that TPG identifier definition includes true identifiers and integers.

```
I_m_an_Identifier_13
1975
```

String

A TPG string is a subset of Python strings. TPG doesn't accept triple quoted strings. If you absolutely need triple quoted strings you can encapsulate them in Python code objects.

```
"I'm a string"
'I\'m a string too"
```

Argument lists and tuples

Argument list is a comma separated list of objects. *Remember that arguments are enclosed in left and right angles.*

```
<object1, object2, object3>
```

Argument lists and tuples have the same syntax except from the possibility to have default arguments, argument lists and argument dictionnaries as arguments as in Python.

```
RULE<arg1, arg2=18, arg3=None, *other_args, **keywords> -> ;
```

Python code object

A Python code object is a piece of Python code in double curly brackets. Python code used in an object expression must have only one line.

```
{{ dict([ (x,x**2) for x in range(100) ]) # Python embeded in TPG }}
```

Text extraction

Text extraction is done by the infix .. operator. Marks can be put in the input string by the prefix @ operator.

```
@beginning
...
@end
...
my_string = beginning .. end
```

Acces to an attribute

Exactly as in Python.

```
my_object.my_attribute
```

Call to a method or a function

Exactly as in Python except from the use of left and right angle instead of parenthesis.

```
my_object.my_method<arg1, arg2>
my_function<arg1, arg2>
my_function_without_arg<>
```

Slice extraction

As in Python.

```
my_list[object]
my_list[object1:object2]
my_list[:object2]
my_list[object1:]
my_list[:]
```

7.11.4 Actions in Python code

TPG parsers also have some interesting methods that can be used in Python code.

Getting the line number of a token

The *lineno* method returns the line number of the current token. If the first parameter is a mark (see 7.11.2) the method returns the line number of the token following the mark.

Backtracking

The user can force the parser to backtrack in rule actions. The parser classes have a *WrongMatch* method for that purpose (see figure 7.6).

Parsers have another useful method named *check* (see figure 7.7). This method checks a condition. If this condition is false then *WrongMatch* if called in order to backtrack.

A shortcut for the *check* method is the *check* keyword followed by the condition to check (see figure 7.8).

Figure 7.6: Backtracking with *WrongMatch* example

```
# NATURAL matches integers greater than 0
NATURAL/n ->
    number/n
    {{ if n<1: self.WrongMatch() }}
    ;
```

Figure 7.7: Backtracking with the *check* method example

```
# NATURAL matches integers greater than 0
NATURAL/n ->
    number/n
    {{ self.check(n>=1) }}
    ;
```

Error reporting

The user can force the parser to stop and raise an exception. The parser classes have a *error* method for that purpose (see figure 7.9). This method raises a *SemanticError*.

A shortcut for the *error* method is the *error* keyword followed by the object to give to the *SemanticError* exception (see figure 7.10).

Figure 7.8: Backtracking with the *check* keyword example

```
# NATURAL matches integers greater than 0
NATURAL/n ->
    number/n
    check {{ n>=1 }}
    ;
```

Figure 7.9: Error reporting the *error* method example

```
# FRACT parses fractions
FRACT/<n,d> ->
    number/n '/' number/d
    {{ if d==0: self.error("Division by zero") }}
    ;
```

Figure 7.10: Error reporting the *error* keyword example

```
# FRACT parses fractions
FRACT/<n,d> ->
    number/n '/' number/d
    ( check d | error "Division by zero" )
    ;
```

Chapter 8

Context sensitive lexer

8.1 Introduction

Before the version 2 of TPG, lexers were context sensitive. That means that the parser commands the lexer to match some tokens, i.e. different tokens can be matched in a same input string according to the grammar rules being used. These lexers were very flexible but slower than context free lexers because TPG backtracking caused tokens to be matched several times.

In TPG 2, the lexer is called before the parser and produces a list of tokens from the input string. This list is then given to the parser. In this case when TPG backtracks the token list remains unchanged.

Since TPG 2.1.2, context sensitive lexers have been reintroduced in TPG. By default lexers are context free but the *CSL* option (see 5.3.2) turns TPG into a context sensitive lexer.

8.2 Grammar structure

CSL grammar have the same structure than non CSL grammars (see 5.1) except from the *CSL* option (see 5.3.2).

8.3 CSL lexers

8.3.1 Regular expression syntax

The CSL lexer is based on the *re* module. The difference with non CSL lexers is that the given regular expression is compiled as this, without any encapsulation. Grouping is then possible and usable.

8.3.2 Token definition

In CSL lexers there is no predefined tokens. Tokens are always inlined and there is no precedance issue since tokens are matched while parsing, when encountered in a grammar rule.

A token definition can be simulated by defining a rule to match a particular token (see figure 8.1).

Figure 8.1: Token definition in CSL parsers example

```
number/int<n> -> '\d+'/n ;
```

In non CSL parsers there are two kinds of tokens: true tokens and token separators. To declare separators in CSL parsers you must use the special *separator* rule. This rule is implicitly used before matching a token. It is thus necessary to distinguish lexical rules from grammar rules. Lexical rule declarations start with the *lex* keyword. In such rules, the *separator* rule is not called to avoid infinite recursion (*separator* calling *separator* calling *separator* ...). The figure 8.2 shows a separator declaration with nested C++ like comments.

Figure 8.2: Separator definition in CSL parsers examples

```
lex separator -> spaces | comment ;

lex spaces -> '\s+' ;

lex comment -> '/\*' in_comment* '\*/' ;        # C++ nested comments
lex in_comment -> comment | '\*[^/]|[^\*]' ;
```

8.3.3 Token matching

In CSL parsers, tokens are matched as in non CSL parsers (see 6.3). There is a special feature in CSL parsers. The user can benefit from the grouping possibilities of CSL parsers. The text of the token can be saved with the infix / operator. The groups of the token can also be saved with the infix // operator. This operator (available only in CSL parsers) returns all the groups in a tuple. For example, the figure 8.3 shows how to read entire tokens and to split tokens.

Figure 8.3: Token usage in CSL parsers examples

```
lex identifier/i -> '\w+'/s ;           # a single identifier

lex string/s -> "'([^\']*)'"//<s> ;     # a string without the quotes

lex item/<key,val> -> "(\w+)=(.*)"//<key,val> ; # a tuple (key, value)
```

8.4 CSL parsers

There is no difference between CSL and non CSL parsers except from lexical rules which look like grammar rules[1].

[1] In fact lexical rules and grammar rule are translated into Python in a very similar way

Part III

Some examples to illustrate TPG

Chapter 9

Complete interactive calculator

9.1 Introduction

This chapter presents an extention of the calculator described in the tutorial (see 3). This calculator has more functions and a memory.

9.2 New functions

9.2.1 Trigonometric and other functions

This calculator can compute some numerical functions (*sin*, *cos*, *sqrt*, ...). The *make_op* function (see figure 3.4) has been extended to return these functions. Tokens must also be defined to scan function names. *funct1* defines the name of unaries functions and *funct2* defines the name of binaries functions. Finally the grammar rule of the atoms has been added a branch to parse functions. The *Function* non terminal symbol parser unaries and binaries functions.

9.2.2 Memories

The calculator has memories. A memory cell is identified by a name. For example, if the user types $pi = 4 * atan(1)$, the memory cell named pi will contain the value of π and $cos(pi)$ will return -1.

To display the content of the whole memory, the user can type *vars*.

The variables are saved in a dictionnary. In fact the parser itself is a dictionnary (the parser inherits from the *dict* class).

The *START* symbol parses a variable creation or a single expression and the *Atom* parses variable names (the *Var* symbol parses a variable name and returns its value).

9.3 Source code

9.3.1 TPG grammar

The calculator source code can be a grammar for TPG. I.e. the *calc.g* file is translated into a *calc.py* script by TPG. Just type in:

```
tpg calc.g
```

Here is the complete source code (*calc.g*):

```
set magic = "/usr/bin/env python"
```

```
{{
import math
import operator
import string
}}

parser Calc(dict):

{{
    def mem(self):
        vars = self.items()
        vars.sort()
        memory = [ "%s = %s"%(var, val) for (var, val) in vars ]
        return "\n\t" + "\n\t".join(memory)

    def make_op(self, op):
        return {
            '+'   : operator.add,
            '-'   : operator.sub,
            '*'   : operator.mul,
            '/'   : operator.div,
            '%'   : operator.mod,
            '^'   : lambda x,y:x**y,
            '**'  : lambda x,y:x**y,
            'cos' : math.cos,
            'sin' : math.sin,
            'tan' : math.tan,
            'acos': math.acos,
            'asin': math.asin,
            'atan': math.atan,
            'sqr' : lambda x:x*x,
            'sqrt': math.sqrt,
            'abs' : abs,
            'norm': lambda x,y:math.sqrt(x*x+y*y),
        }[op]
}}

separator space: '\s+' ;

token pow_op: '\^|\*\*' self.make_op ;
token add_op: '[+-]' self.make_op ;
token mul_op: '[*/%]' self.make_op ;
token funct1: '(cos|sin|tan|acos|asin|atan|sqr|sqrt|abs)\b' self.make_op ;
token funct2: '(norm)\b' self.make_op ;
token real: '(\d+\.\d*|\d*\.\d+)([eE][-+]?\d+)?|\d+[eE][-+]?\d+' string.atof ;
token integer: '\d+' string.atol ;
token VarId: '[a-zA-Z_]\w*' ;

START/e ->
        'vars' e=self.mem<>
    |   VarId/v '=' Expr/e self[v]=e
    |   Expr/e
    ;
```

```
Var/self.get<v,0> -> VarId/v ;

Expr/e -> Term/e ( add_op/op Term/t e=op<e,t> )* ;

Term/t -> Fact/t ( mul_op/op Fact/f t=op<t,f> )* ;

Fact/f ->
        add_op/op Fact/f f=op<0,f>
    |   Pow/f
    ;

Pow/f -> Atom/f ( pow_op/op Fact/e f=op<f,e> )? ;

Atom/a ->
        real/a
    |   integer/a
    |   Function/a
    |   Var/a
    |   '\(' Expr/a '\)'
    ;

Function/y ->
        funct1/f '\(' Expr/x '\)' y = f<x>
    |   funct2/f '\(' Expr/x1 ',' Expr/x2 '\)' y = f<x1,x2>
    ;

main:

{{
    print "Calc (TPG example)"
    calc = Calc()
    while 1:
        l = raw_input("\n:")
        if l:
            try:
                print calc(l)
            except Exception, e:
                print e
        else:
            break
}}
```

9.3.2 Python script

The calculator can be directly embeded in a Python script. The grammar is in a string and compiled using the *tpg* module.

Here is the complete source code (*calc2.py*):

```
#!/usr/bin/env python

import math
import operator
import string
import tpg
```

```
def make_op(op):
    return {
        '+'   : operator.add,
        '-'   : operator.sub,
        '*'   : operator.mul,
        '/'   : operator.div,
        '%'   : operator.mod,
        '^'   : lambda x,y:x**y,
        '**'  : lambda x,y:x**y,
        'cos' : math.cos,
        'sin' : math.sin,
        'tan' : math.tan,
        'acos': math.acos,
        'asin': math.asin,
        'atan': math.atan,
        'sqr' : lambda x:x*x,
        'sqrt': math.sqrt,
        'abs' : abs,
        'norm': lambda x,y:math.sqrt(x*x+y*y),
    }[op]

exec(tpg.compile(r"""

parser Calc(dict):

{{
    def mem(self):
        vars = self.items()
        vars.sort()
        memory = [ "%s = %s"%(var, val) for (var, val) in vars ]
        return "\n\t" + "\n\t".join(memory)
}}

separator space: '\s+' ;

token pow_op: '\^|\*\*' make_op ;
token add_op: '[+-]' make_op ;
token mul_op: '[*/%]' make_op ;
token funct1: '(cos|sin|tan|acos|asin|atan|sqr|sqrt|abs)\b' make_op ;
token funct2: '(norm)\b' make_op ;
token real: '(\d+\.\d*|\d*\.\d+)([eE][-+]?\d+)?|\d+[eE][-+]?\d+' string.atof ;
token integer: '\d+' string.atol ;
token VarId: '[a-zA-Z_]\w*' ;

START/e ->
        'vars' e=self.mem<>
    |   VarId/v '=' Expr/e self[v]=e
    |   Expr/e
    ;

Var/self.get<v,0> -> VarId/v ;

Expr/e -> Term/e ( add_op/op Term/t e=op<e,t> )* ;
```

```
Term/t -> Fact/t ( mul_op/op Fact/f t=op<t,f> )* ;

Fact/f ->
        add_op/op Fact/f f=op<0,f>
    |   Pow/f
    ;

Pow/f -> Atom/f ( pow_op/op Fact/e f=op<f,e> )? ;

Atom/a ->
        real/a
    |   integer/a
    |   Function/a
    |   Var/a
    |   '\(' Expr/a '\)'
    ;

Function/y ->
        funct1/f '\(' Expr/x '\)' y = f<x>
    |   funct2/f '\(' Expr/x1 ',' Expr/x2 '\)' y = f<x1,x2>
    ;

"""))

print "Calc (TPG example)"
calc = Calc()
while 1:
    l = raw_input("\n:")
    if l:
        try:
            print calc(l)
        except Exception, e:
            print e
    else:
        break
```

Chapter 10

Infix/Prefix/Postfix notation converter

10.1 Introduction

In the previous example, the parser computes the value of the expression on the fly, while parsing. It is also possible to build an abstract syntax tree to store an abstract representation of the input. This may be usefull when several passes are necessary.

This example shows how to parse an expression (infix, prefix or postfix) and convert it in infix, prefix and postfix notation. The expression is saved in a tree. Each node of the tree correspond to an operator in the expression. Each leave is a number. Then to write the expression in infix, prefix or postfix notation, we just need to walk throught the tree in a particular order.

10.2 Abstract syntax trees

The AST of this converter has two types of node:

class Op is used to store operators (`+`, `-`, `*`, `/`, `^`). It has two sons associated to the sub expressions.

class Atom is an atomic expression (a number or a symbolic name).

Both classes are instanciated by the *__init__* method. The *infix*, *prefix* and *postfix* methods return strings containing the representation of the node in *infix*, *prefix* and *postfix* notation.

10.3 Grammar

10.3.1 Infix expressions

The grammar for infix expressions is similar to the grammar used in the previous example.

```
EXPR/e -> TERM/e ( '[+-]'/op TERM/t e=Op<op,e,t,1> )* ;
TERM/t -> FACT/t ( '[*/]'/op FACT/f t=Op<op,t,f,2> )* ;
FACT/f -> ATOM/f ( '\^'/op FACT/e f=Op<op,f,e,3> )? ;

ATOM/a -> ident/s a=Atom<s> | '\(' EXPR/a '\)' ;
```

10.3.2 Prefix expressions

The grammar for prefix expressions is very simple. A compound prefix expression is an operator followed by two subexpressions.

```
EXPR_PRE/e ->
    ident/s e=Atom<s>
|   '\(' EXPR_PRE/e '\)'
|   OP/<op,prec> EXPR_PRE/a EXPR_PRE/b e=Op<op,a,b,prec>
;

OP/<op,prec> ->
    '[+-]'/op prec=1
|   '[*/]'/op prec=2
|   '\^'/op   prec=3
;
```

10.3.3 Postfix expressions

At first sight postfix and infix grammars may be very similar. Only the position of the operators changes. So a compound postfix expression is a first expression followed by a second and an operator. This rule is left recursive. As TPG is a descendant recursive parser, such rules are forbidden to avoid infinite recursion. To remove the left recursion a classical solution is to rewrite the grammar like this:

```
EXPR_POST/e -> ATOM_POST/a SEXPR_POST<a>/e ;

ATOM_POST/a ->
    ident/s a=Atom<s>
|   '\(' EXPR_POST/a '\)'
;

SEXPR_POST<e>/e ->
    EXPR_POST/e2 OP/<op,prec> SEXPR_POST<Op<op,e,e2,prec>>/e
|   ;
```

The parser first searches for an atomic expression and then builds the AST by passing partial expressions by the attributes of the *SEXPR_POST* symbol.

10.4 Source code

Here is the complete source code (*notation.py*):

```
#!/usr/bin/env python

# Infix/prefix/postfix expression conversion

import tpg

class Op:
    """ Binary operator """
    def __init__(self, op, a, b, prec):
        self.op = op            # operator ("+", "-", "*", "/", "^")
        self.prec = prec        # precedence of the operator
        self.a, self.b = a, b   # operands
```

```
    def infix(self):
        a = self.a.infix()
        if self.a.prec < self.prec: a = "(%s)"%a
        b = self.b.infix()
        if self.b.prec <= self.prec: b = "(%s)"%b
        return "%s %s %s"%(a, self.op, b)
    def prefix(self):
        a = self.a.prefix()
        b = self.b.prefix()
        return "%s %s %s"%(self.op, a, b)
    def postfix(self):
        a = self.a.postfix()
        b = self.b.postfix()
        return "%s %s %s"%(a, b, self.op)

class Atom:
    """ Atomic expression """
    def __init__(self, s):
        self.a = s
        self.prec = 99
    def infix(self): return self.a
    def prefix(self): return self.a
    def postfix(self): return self.a

exec(tpg.compile(r"""

# Grammar for arithmetic expressions

parser ExpressionParser:

separator space: '\s+';
token ident: '\w+';

START/<e,t> ->
    EXPR/e      t='infix'       '\n'
|   EXPR_PRE/e  t='prefix'      '\n'
|   EXPR_POST/e t='postfix'     '\n'
;

# Infix expressions

EXPR/e -> TERM/e ( '[+-]'/op TERM/t e=Op<op,e,t,1> )* ;
TERM/t -> FACT/t ( '[*/]'/op FACT/f t=Op<op,t,f,2> )* ;
FACT/f -> ATOM/f ( '\^'/op FACT/e f=Op<op,f,e,3> )? ;

ATOM/a -> ident/s a=Atom<s> | '\(' EXPR/a '\)' ;

# Prefix expressions

EXPR_PRE/e ->
    ident/s e=Atom<s>
|   '\(' EXPR_PRE/e '\)'
|   OP/<op,prec> EXPR_PRE/a EXPR_PRE/b e=Op<op,a,b,prec>
;
```

```
# Postfix expressions

EXPR_POST/e -> ATOM_POST/a SEXPR_POST<a>/e ;

ATOM_POST/a ->
    ident/s a=Atom<s>
|   '\(' EXPR_POST/a '\)'
;

SEXPR_POST<e>/e ->
    EXPR_POST/e2 OP/<op,prec> SEXPR_POST<Op<op,e,e2,prec>>/e
|   ;

OP/<op,prec> ->
    '[+-]'/op prec=1
|   '[*/]'/op prec=2
|   '\^'/op   prec=3
;

"""))

parser = ExpressionParser()
while 1:
    e = raw_input(":")
    if e == "": break
    try:
        expr, t = parser(e+"\n")
    except (tpg.LexicalError, tpg.SyntaxError), e:
        print e
    else:
        print e, "is a", t, "expression"
        print "\tinfix   :", expr.infix()
        print "\tprefix  :", expr.prefix()
        print "\tpostfix :", expr.postfix()
```

Part IV

Internal structure of TPG for the curious

Chapter 11

Structure of the package

11.1 General structure of the package

TPG is delivered in a Python package named *tpg*. It is composed of:

__init__.py turns *tpg* directory into a package. It defines some data about the current release (version, author, ...) and it imports in its local namespace the five useful objects *compile*, *translate*, *LexerError*, *ParserError* and *SemanticError*.

base.py defines the base class of the generated parsers and other classes used by these parsers. It's a kind of runtime for the parsers.

codegen.py contains the classes used by the parser to represent the AST corresponding to the parsed grammar. Theses classes have the necessary methods for code generation.

parser.g contains the grammar that recognizes TPG grammars. It defines the syntax of TPG grammars and builds the AST of the grammar.

parser.py is automatically generated by TPG itself from *parser.g*.

Release.py contains release data (version, author, ...).

tpg is a wrapper script for TPG. It reads a grammar and produces a Python script.

Chapter 12

Lexer

12.1 Token matching

Tokens are defined by their regular expressions (see 6.2). TPG builds a regular expression by assembling each regular expression in a *or* structure. For example to recognize *int* ($[0-9]+$) and *word* ($[a-zA-Z]+$), TPG builds this composite expression: $(?P<int>[0-9]+) \mid (?P<word>[a-zA-Z]+)$ This expression is then compiled using the *re* module.

For each token we save its name, its text, its value (i.e. the result of its action applied to its text), the line number and the position of the start and the end of the token in the input string.

There is a special token named *EOF* used as the erroneous token when a lexical error appears near the end of the input.

Chapter 13

Parser

13.1 Interface with the lexer

The lexer produces a list of tokens (see 6.3). The parser save the number of the current token. Each time a token is matched (*_eat* method), the current token number is incremented. This counter does not appear in the generated code. It is handled by the *_eat* method.

13.2 Sequences of subexpressions

There is nothing particular about sequences (see 7.6). A sequence of expressions is translated into a sequence of Python statements (see 14.3.3).

13.3 Alternatives between subexpressions

Alternatives (see 7.8) are tried in the order of their declaration. The first match will stop the search. When a branch fails (i.e. a call to the *_eat* method raises a *TPGWrongMatch* exception) the alternative control structure catches the exception and tries the next branch. On the last branch the exception is not catched in order to be handled by an outer choice point (see 14.3.5).

13.4 Repetitions

Repetitions (see 7.9) use the same scheme as alternatives. The *TPGWrongMatch* exception stops the loop when raised (see 14.3.6).

Chapter 14

Code generation

This chapter shows the code generated by TPG. It is not necessary to read it to understand how TPG works. This chapter has been written mostly the curious readers.

14.1 Inheritance

TPG parsers can inherit from other Python classes (see 7.2). See figure 14.1 for the generated code.

Figure 14.1: Inheritance example

Grammar	Generated code
`parser MyParser(Base1, Base2):`	`class MyParser(tpg.base.ToyParser,Base1,Base2):`

14.2 Lexer

The figure 14.2 shows token precedence (see 6.2). Tokens are declared in the order of appearance except from inline tokens that are declared before predefined tokens.

Figure 14.2: Lexer example

Grammar	Generated code

<pre>parser Foo:
 	
    token integer: '\d+' int ;
    token arrow: '->' ;
    separator spaces: '\s+' ;
 	
    S ->
 	
        '\('
        integer
        arrow
        '\)'
        ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def _init_scanner(self):
	        self._lexer = tpg.base._Scanner(
	            tpg.base._TokenDef(r"_tok_1", r"\("),
	            tpg.base._TokenDef(r"_tok_2", r"\)"),
	            tpg.base._TokenDef(r"integer", r"\d+", int, 0),
	            tpg.base._TokenDef(r"arrow", r"->", None, 0),
	            tpg.base._TokenDef(r"spaces", r"\s+", None, 1),
	        )
	
	    def S(self,):
	        """ S -> '\(' integer arrow '\)' """
	        self._eat('_tok_1') # \(
	        self._eat('integer')
	        self._eat('arrow')
	        self._eat('_tok_2') # \)</pre>

14.3 Parser

14.3.1 Grammar rules

Grammar rules (see 7.3) are used to define what a symbol is composed of. A rule is translated into a method of the parser class (see figure 14.3). The attributes of the symbol are the parameters of the methods. The docstring of the method is the grammar rule.

Figure 14.3: Rule declaration example

Grammar	Generated code
`parser Foo:`	`class Foo(tpg.base.ToyParser,):`
`Symbol1 -> ;`	`def Symbol1(self,):` `""" Symbol1 -> """`
`Symbol2<arg1, arg2, arg3> -> ;`	`def Symbol2(self,arg1,arg2,arg3):` `""" Symbol2 -> """`

`Symbol3/ret_val -> ;`	`def Symbol3(self,):`
	`""" Symbol3 -> """`
	`return ret_val`

`Symbol4<arg1, arg2, arg3>/ret_val -> ;`	`def Symbol4(self,arg1,arg2,arg3):`
	`""" Symbol4 -> """`
	`return ret_val`

14.3.2 Symbols

Terminal symbols

Terminal symbols (see 6.2) are recognized by calling the *_eat* method with the name of the token to match (see figure 14.4). Terminal symbols can return the token text in a string. If the current token is not the expected token, *_eat* raises a *TPGWrongMatch* exception. This exception will be catched either by an outer choice point to try another choice or by TPG to turn this exception into a *ParserError* exception.

Figure 14.4: Terminal symbol matching example

Grammar	Generated code

<pre>parser Foo:
 	
    token predef: 'bar' ;
 	
    S ->
 	
        'inline'
        predef
        'inline'/s1
        predef/s2
        ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def _init_scanner(self):
	        self._lexer = tpg.base._Scanner(
	            tpg.base._TokenDef(r"inline", r"inline"),
	            tpg.base._TokenDef(r"predef", r"bar", None, 0),
	        )
	
	    def S(self,):
	        """ S -> 'inline' predef 'inline' predef """
	        self._eat('inline')
	        self._eat('predef')
	        s1 = self._eat('inline')
	        s2 = self._eat('predef')</pre>

Non terminal symbols

Non terminal symbols (see 7.5) are recognized by calling their rules (see figure 14.5). Non terminal symbols can have attributes, a return value or both.

Figure 14.5: Non terminal symbol matching example

Grammar	Generated code

<pre>parser Foo:
 	
    S ->
 	
        NTerm1
        NTerm2<arg1, arg2>
        NTerm3/ret_val
        NTerm4<arg1, arg2>/ret_val
        ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def S(self,):
	        """ S -> NTerm1 NTerm2 NTerm3 NTerm4 """
	        self.NTerm1()
	        self.NTerm2(arg1,arg2)
	        ret_val = self.NTerm3()
	        ret_val = self.NTerm4(arg1,arg2)</pre>

14.3.3 Sequences

The token number is updated by the *_eat* method when called so a sequence (see 7.6) in a rule is translated into a sequence of statements in Python (see figure 14.6).

Figure 14.6: Sequence of expressions example

Grammar	Generated code

<pre>parser Foo:
 	
    S -> A B C ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def S(self,):
	        """ S -> A B C """
	        self.A()
	        self.B()
	        self.C()</pre>

14.3.4 Cut

The cut mechanism (see 7.7) is implemented as a shortcut to the *TPGWrongMatch* exception. When the sequence following a cut fails, i.e. when it raises a *TPGWrongMatch* exception, TPG turns this exception into a *ParserError* exception to immediately abort parsing (see figure 14.7).

Figure 14.7: Cut example

Grammar	Generated code

<pre>parser Foo:
 	
    S ->
 	
        A1 !
 	
            B1
            C1
 	
    |
 	
        A2 !
 	
            B2
            C2
    ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def S(self,):
	        """ S -> A1 B1 C1 | A2 B2 C2 """
	        __p1 = self._cur_token
	        try:
	            self.A1()
	            try:
	                self.B1()
	                self.C1()
	            except self.TPGWrongMatch, e:
	                self.ParserError(e.last)
	        except self.TPGWrongMatch:
	            self._cur_token = __p1
	            self.A2()
	            try:
	                self.B2()
	                self.C2()
	            except self.TPGWrongMatch, e:
	                self.ParserError(e.last)</pre>

14.3.5 Alternatives

Alternatives (see 7.8) are tried in the order they are declared. Before trying the first branch, TPG saves the current token number. If the first choice fails, the token number is restored before trying the second branch. When a branch fails, the *_eat* method raises a *TPGWrongMatch* exception which is catched by the alternative structure. This algorithm is very simple to implement but isn't very efficient. This is how the computation of any prediction table is avoided.

Figure 14.8: Alternative in expressions example

Grammar	Generated code

<pre>parser Foo:
 	
    S -> A | B | C | D ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def S(self,):
	        """ S -> A | B | C | D """
	        __p1 = self._cur_token
	        try:
	            try:
	                self.A()
	            except self.TPGWrongMatch:
	                self._cur_token = __p1
	                self.B()
	        except self.TPGWrongMatch:
	            self._cur_token = __p1
	            try:
	                self.C()
	            except self.TPGWrongMatch:
	                self._cur_token = __p1
	                self.D()</pre>

14.3.6 Repetitions

Repetitions (see 7.9) are implemented in a similar way to alternatives. The *TPGWrongMatch* tells TPG when to go out of the loop. See figures 14.9 and 14.10 for repetition examples.

Figure 14.9: Repetition examples: builtin ?, * and +

<table>
<tr><th>Grammar</th><th>Generated code</th></tr>
<tr><td><pre>
parser Repetitions:

    ZERO_or_ONE ->

        A ?
        ;

    ZERO_or_MORE ->

        A *
        ;

    ONE_or_MORE ->

        A +
        ;
</pre></td><td><pre>
class Repetitions(tpg.base.ToyParser,):

    def ZERO_or_ONE(self,):
        """ ZERO_or_ONE -> A? """
        __p1 = self._cur_token
        try:
            self.A()
        except self.TPGWrongMatch:
            self._cur_token = __p1

    def ZERO_or_MORE(self,):
        """ ZERO_or_MORE -> A* """
        __p1 = self._cur_token
        while 1:
            try:
                self.A()
                __p1 = self._cur_token
            except self.TPGWrongMatch:
                self._cur_token = __p1
                break

    def ONE_or_MORE(self,):
        """ ONE_or_MORE -> A+ """
        __p1 = self._cur_token
        __n1 = 0
        while 1:
            try:
                self.A()
                __n1 += 1
                __p1 = self._cur_token
            except self.TPGWrongMatch:
                if __n1 >= 1:
                    self._cur_token = __p1
                    break
                else:
                    self.WrongMatch()
</pre></td></tr>
</table>

Figure 14.10: Repetition examples: user defined {m,n}

Grammar	Generated code

<pre>parser Repetitions:
 	
    USER_DEFINED ->
 	
        A{2,5}
        ;</pre>
	<pre>class Repetitions(tpg.base.ToyParser,):
	
	    def USER_DEFINED(self,):
	        """ USER_DEFINED -> A{2,5} """
	        __p1 = self._cur_token
	        __n1 = 0
	        while __n1<5:
	            try:
	                self.A()
	                __n1 += 1
	                __p1 = self._cur_token
	            except self.TPGWrongMatch:
	                if __n1 >= 2:
	                    self._cur_token = __p1
	                    break
	                else:
	                    self.WrongMatch()</pre>

14.3.7 Abstract syntax trees

Abstract syntax trees (see 7.11.1) are simply Python objects. The figure 14.11 shows the instanciation of a node. The figure 14.12 shows the update with the *add* method.

Figure 14.11: AST instanciation example

Grammar

```
{{
    class Couple:
        def __init__(self, a, b):
            self.a = a
            self.b = b
}}
parser Foo:
    COUPLE1/c ->

        c=Couple<a,b>
        ;

    COUPLE2/Couple<a,b> ->

        ;
```

Generated code

```
class Couple:
    def __init__(self, a, b):
        self.a = a
        self.b = b
class Foo(tpg.base.ToyParser,):

    def COUPLE1(self,):
        """ COUPLE1 ->  """
        c = Couple(a,b)
        return c

    def COUPLE2(self,):
        """ COUPLE2 ->  """
        return Couple(a,b)
```

Figure 14.12: AST update example

Grammar

```
{{
    class List(list):
        add = list.append
}}
parser Foo:
    LIST/l ->

        l = List<>
        ITEM/a
        l-a
        ;
```

Generated code

```
class List(list):
    add = list.append
class Foo(tpg.base.ToyParser,):

    def LIST(self,):
        """ LIST -> ITEM """
        l = List()
        a = self.ITEM()
        l.add(a)
        return l
```

14.3.8 Text extraction

Text can be extracted (see 7.11.2) from the input string (including separators). The prefix @ operator puts a mark on the current token. The infix .. operator extracts the text between two marks.

The figure 14.13 shows how this extraction works.

Figure 14.13: Text extraction

Grammar	Generated code

<pre>parser Foo:
 	
    S ->
 	
        A
        @x   # put a mark 'x'
        B
        C
        @y   # put a mark 'y'
        t = x..y   # extract from 'x' to 'y'
        ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def S(self,):
	        """ S -> A B C """
	        self.A()
	        x = self._mark()
	        self.B()
	        self.C()
	        y = self._mark()
	        t = self._extract(x,y)</pre>

14.3.9 Python objects

TPG has an adapted syntax for some Python expressions (see 7.11.3).

The figure 14.14 shows this implementation.

Figure 14.14: Python object in TPG

Grammar	Generated code

<pre>parser Foo:
 	
    Bar ->
 	
        x = y
        x = "string"
        x = <y>
        x = <y, z>
        x = {{ x + y }}
        x = y.z
        x = y<a,b>
        x = z<>
        x = lst[1]
        x = lst[2:3]
        x = lst[:3]
        x = lst[2:]
        x = lst[:]
        ;</pre>
	<pre>class Foo(tpg.base.ToyParser,):
	
	    def Bar(self,):
	        """ Bar -> """
	        x = y
	        x = r"string"
	        x = (y,)
	        x = (y, z,)
	        x = x + y
	        x = y.z
	        x = y(a,b)
	        x = z()
	        x = lst[1]
	        x = lst[2:3]
	        x = lst[:3]
	        x = lst[2:]
	        x = lst[:]</pre>