Toy Parser Generator
or
How to easily write parsers in Python

Christophe Delord

December 29, 2012

Contents

List of Figures

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) 2001-2011 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 FIT- NESS 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 the 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.

Chapter 2

Installation

2.1 Getting TPG

TPG is freely available on its web page (http://cdsoft.fr/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 3.2* is also supported). 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 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 shows 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 sum of terms and a term is a product of factors. A factor 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 (Expr, Term, Factor). 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
$Expr \rightarrow Term \; (('+'|'-') \; Term)*$	An expression is a term eventually followed with a plus ($'+'$) or a minus ($'-'$) sign and an other term any number of times ($*$ is a repetition of an expression 0 or more times).
$Term \rightarrow Fact \; (('*'|'/') \; Fact)*$	A term is a factor eventually followed with a $'*'$ or $'/'$ sign and an other factor any number of times.
$Fact \rightarrow number \; | \; '(' \; Expr \; ')'$	A factor is either a number or an expression 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

```
class Calc(tpg.Parser):
    r"""

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

    START -> Expr ;

    Expr -> Term ( add Term )* ;

    Term -> Fact ( mul Fact )* ;

    Fact -> number | '\(' Expr '\)' ;

    """
```

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).

---

[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]
```

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

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:

```
Expr/t -> Term/t ( add/op Term/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.

Figure 3.7: Expression recognizer and evaluator

```
class Calc(tpg.Parser):
    r"""

    separator spaces: '\s+' ;

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

    START -> Expr ;

    Expr/t -> Term/t ( add/op Term/f $t=op(t,f)$ )* ;

    Term/f -> Fact/f ( mul Fact/a $f=op(f,a)$ )* ;

    Fact/a -> number/a | '\(' Expr/a '\)' ;

    """
```

3.4   Embeding the parser in a script

Since TPG 3 embeding parsers in a script is very easy since the grammar is the doc string[2] of a class (see figure 3.8).

Figure 3.8: Writting TPG grammars in Python

```
import tpg

class MyParser(tpg.Parser):
    r""" # Your grammar here """

# You can instanciate your parser here
my_parser = MyParser()
```

To use this parser you now just need to instanciate an object of the class *Calc* as in figure 3.9.

---

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

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]

class Calc(tpg.Parser):
    r"""

    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 define a class which particular metaclass converts a doc string into a parser. You only need to import TPG and use these five objects:

tpg.Parser: This is the base class of the parsers you want to define.

tpg.Error: This exception is the base of all TPG exceptions.

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

tpg.SyntacticError: 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 the doc string of the class (see figure 4.1).

Figure 4.1: Grammar embeding example

```
class Foo(tpg.Parser):
    r"""

    START/x -> Bar/x ;

    Bar/x -> 'bar'/x ;

    """
```

Then you can use the new generated parser. The parser is simply a Python class (see figure 4.2).

4.2 Command line usage

The *tpg* script reads a Python script and replaces TPG grammars (in doc string) by Python code. To produce a Python script (*.py) from a script containing grammars (*.pyg) you can use *tpg* as follow:

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

Figure 4.2: 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
```

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.pyg* is the name of the grammar.

Notice that .pyg files are valid Python scripts. So you can choose the run .pyg file (slower startup but easier for debugging purpose) or turn them into a .py file (faster startup but needs a "precompilation" stage). You can also write .py scripts containing grammars to be used as Python scripts.

In fact I only use the *tpg* script to convert *tpg.pyg* into *tpg.py* because TPG needs obviously to be a pure Python script (it can not translate itself at runtime). Then in most cases it is very convenient to directly write grammars in Python scripts.

Chapter 5

Grammar structure

5.1 TPG grammar structure

TPG grammars are contained in the doc string[1] of the parser class. TPG grammars may contain three parts:

ReStructuredText docstring is a part of the grammar that is ignored by TPG. It shall end with a '::' at the end of a line.

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

Tokens are introduced by the *token* or *separator* keyword (see 6.2).

Rules are described after tokens (see 5.5).

See figure 5.1 for a generic TPG grammar.

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

5.3.1 Lexer option

The *lexer* option tells TPG which lexer to use.

set lexer = NamedGroupLexer is the default lexer. It is context free and uses named groups of the *re* package (and its limitation of 100 named groups, ie 100 tokens).

set lexer = Lexer is similar to *NamedGroupLexer* but doesn't use named groups. It is slower than *NamedGroupLexer*.

---

[1]If the grammar (i.e. the doc string) is a unicode string then the generated parser can parse unicode strings

Figure 5.1: TPG grammar structure

```
class Foo(tpg.Parser):
    r"""
    Here is some ReStructuredText documentation
    (for Sphinx for instance).

    And now, the grammar::

        # Options
        set lexer = CSL

        # Tokens
        separator spaces    '\s+'        ;
        token int           '\d+'   int ;

        # Rules
        START -> X Y Z ;

    """

foo = Foo()
result = foo("input string")
```

set lexer = CacheNamedGroupLexer is similar to *NamedGroupLexer* except that tokens are first stored in a list. It is faster for heavy backtracking grammars.

set lexer = CacheLexer is similar to *Lexer* except that tokens are first stored in a list. It is faster for heavy backtracking grammars.

set lexer = ContextSensitiveLexer is the context sensitive lexer (see 8).

5.3.2 Word bondary option

The *word_boundary* option tells the lexer to search for word boundaries after identifiers.

set word_boundary = True enables the word boundary search. This is the default.

set word_boundary = False disables the word boundary search.

5.3.3 Regular expression options

The *re* module accepts some options to define the behaviour of the compiled regular expressions. These options can be changed for each parser.

set lexer_ignorecase = True enables the *re.IGNORECASE* option.

set lexer_locale = True enables the *re.LOCALE* option.

set lexer_multiline = True enables the *re.MULTILINE* option.

set lexer_dotall = True enables the *re.DOTALL* option.

set lexer_verbose = True enables the *re.VERBOSE* option.

set lexer_unicode = True enables the *re.UNICODE* option.

5.4   Python code

Python code sections 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

Before TPG 3, Python code was enclosed in double curly brackets. That means that Python code must not contain two consecutive close brackets. You can avoid this by writting } } (with a space) instead of }} (without space). This syntaxe is still available but the new syntax may be more readable. The new syntax uses $ to delimit code sections. When several $ sections are consecutive they are seen as a single section.

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.

When TPG encounters some Python code it removes from 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.

5.5   TPG parsers

TPG parsers are *tpg.Parser* classes. The grammar is the doc string of the class.

5.5.1   Methods

As TPG parsers are just Python classes, you can use them as normal classes. If you redefine the *__init__* method, do not forget to call *tpg.Parser.__init__*.

5.5.2   Rules

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

Figure 5.2: Code indentation examples

Code in grammars (old syntax)	Code in grammars (new syntax)	Generated code	Comment
| ```
{{
␣␣␣␣if␣1==2:
␣␣␣␣␣␣␣␣print␣"???"
␣␣␣␣else:
␣␣␣␣␣␣␣␣print␣"OK"
}}
␣␣␣␣
``` | ```
$␣␣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"
}}
␣␣␣␣
``` | The new syntax has no trouble in that case. | ```
if␣1==2:
␣␣␣␣␣␣print␣"???"
␣␣else:
␣␣␣␣␣␣print␣"OK"
␣␣␣␣
``` | *WRONG*: it is 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 will not compile this code. |
| ```
{{␣␣␣␣print␣"OK"␣}}
␣␣␣␣
``` | ```
$␣␣␣␣␣␣␣print␣"OK"
␣␣␣␣

or

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

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 by name syntax since it is used by TPG to decide which token is recognized.

Here is a summary[2] of the regular expression syntax:

"." (Dot.) In the default mode, this matches any character except a newline. If the *DOTALL* flag has been specified, this matches any character including a newline.

"ˆ" (Caret.) Matches the start of the string, and in *MULTILINE* mode also matches immediately after each newline.

"\$" Matches the end of the string or just before the newline at the end of the string, and in *MULTILINE* mode also matches before a newline. foo matches both 'foo' and 'foobar', while the regular expression $foo\$$ matches only 'foo'. More interestingly, searching for $foo.\$$ in 'foo1\nfoo2\n' matches 'foo2' normally, but 'foo1' in *MULTILINE* mode.

"*" Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. $ab*$ will match 'a', 'ab', or 'a' followed by any number of 'b's.

"+" Causes the resulting RE to match 1 or more repetitions of the preceding RE. $ab+$ will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

"?" Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. $ab?$ will match either 'a' or 'ab'.

?, +?, ?? The "", "+", and "?" qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE $<.*>$ is matched against '<H1>title</H1>', it will match the entire string, and not just '<H1>'. Adding "?" after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using $.*?$ in the previous expression will match only '<H1>'.

{m} Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, $a\{6\}$ will match exactly six "a" characters, but not five.

---

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

[2] From the Python documentation: http://docs.python.org/lib/re-syntax.html

{m,n} Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, $a\{3,5\}$ will match from 3 to 5 "a" characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example, $a\{4,\}b$ will match aaaab or a thousand "a" characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}? Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', $a\{3,5\}$ will match 5 "a" characters, while $a\{3,5\}?$ will only match 3 characters.

"\\" Either escapes special characters (permitting you to match characters like "*", "?", and so forth), or signals a special sequence; special sequences are discussed below.

[] Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a "-". Special characters are not active inside sets. For example, $[akm\$]$ will match any of the characters "a", "k", "m", or "\$"; $[a-z]$ will match any lowercase letter, and $[a-zA-Z0-9]$ matches any letter or digit. Character classes such as \\w or \\S (defined below) are also acceptable inside a range. If you want to include a "]" or a "-" inside a set, precede it with a backslash, or place it as the first character. The pattern [] will match ']', for example.

You can match the characters not within a range by complementing the set. This is indicated by including a "^" as the first character of the set; "^" elsewhere will simply match the "^" character. For example, [^5] will match any character except "5", and [^^] will match any character except "^".

"|" $A|B$, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the "|" in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by "|" are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the "|" operator is never greedy. To match a literal "|", use \\|, or enclose it inside a character class, as in [|].

(...) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the $\backslash number$ special sequence, described below. To match the literals "(" or ")", use \\(or \\), or enclose them inside a character class: [(] [)].

(?=...) Matches if ... matches next, but doesn't consume any of the string. This is called a lookahead assertion. For example, $Isaac(? = Asimov)$ will match 'Isaac ' only if it's followed by 'Asimov'.

(?!...) Matches if ... doesn't match next. This is a negative lookahead assertion. For example, $Isaac(?!Asimov)$ will match 'Isaac ' only if it's not followed by 'Asimov'.

(?<=...) Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion. $(? <= abc)def$ will find a match in "abcdef", since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or $a|b$ are allowed, but $a*$ and $a\{3,4\}$ are not.

(?<!...) Matches if the current position in the string is not preceded by a match for .... This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

$\backslash A$ Matches only at the start of the string.

$\backslash b$ Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that $\backslash b$ is defined as the boundary between $\backslash w$ and $\backslash W$, so the precise set of characters deemed to be alphanumeric depends on the values of the UNICODE and LOCALE flags. Inside a character range, $\backslash b$ represents the backspace character, for compatibility with Python's string literals.

$\backslash B$ Matches the empty string, but only when it is not at the beginning or end of a word. This is just the opposite of $\backslash b$, so is also subject to the settings of LOCALE and UNICODE.

$\backslash d$ Matches any decimal digit; this is equivalent to the set $[0-9]$.

$\backslash D$ Matches any non-digit character; this is equivalent to the set $[\hat{\ }0-9]$.

$\backslash s$ Matches any whitespace character; this is equivalent to the set $[\ \backslash t\backslash n\backslash r\backslash f\backslash v]$.

$\backslash S$ Matches any non-whitespace character; this is equivalent to the set $[\hat{\ }\ \backslash t\backslash n\backslash r\backslash f\backslash v]$.

$\backslash w$ When the LOCALE and UNICODE flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set $[a-zA-Z0-9\_]$. With LOCALE, it will match the set $[0-9\_]$ plus whatever characters are defined as alphanumeric for the current locale. If UNICODE is set, this will match the characters $[0-9\_]$ plus whatever is classified as alphanumeric in the Unicode character properties database.

$\backslash W$ When the LOCALE and UNICODE flags are not specified, matches any non-alphanumeric character; this is equivalent to the set $[\hat{\ }a-zA-Z0-9\_]$. With LOCALE, it will match any character not in the set $[0-9\_]$, and not defined as alphanumeric for the current locale. If UNICODE is set, this will match anything other than $[0-9\_]$ and characters marked as alphanumeric in the Unicode character properties database.

$\backslash Z$ Matches only at the end of the string.

$\backslash a\ \backslash f\ \backslash n\ \backslash r\ \backslash t\ \backslash v\ \backslash x\ \backslash\backslash$ Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser.

$\backslash 0xyz$, $\backslash xyz$ Octal escapes are included in a limited form: If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. As for string literals, octal escapes are always at most three digits in length.

6.2   Token definition

6.2.1   Predefined tokens

Tokens can be explicitly 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. If 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.

Figure 6.1: Token definition examples

```
#      name     reg. exp        action
token integer: '\d+'           int;
token ident  : '[a-zA-Z]\w*'   ;
token bool   : 'True|False'     $ lambda b: b=='True'

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

Token definitions must end with a *;* when no action is specified. The dots after the token name are optional.

See figure 6.1 for examples.

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). This behaviour can be disabled since the version 3 of TPG (see 5.3.2).

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 definitions 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* will not 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. The default lexer is lazy in TPG 3. Tokens are generated while parsing. This way TPG 3 need less memory when parsing huge files.

6.3.1 Splitting the input string

The lexer splits the input string according to the token definitions (see 6.2). When the input string can not be matched a *tpg.LexicalError* 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 saved (see figure 6.4).

Figure 6.4: Token usage examples

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

Chapter 7

Parser

7.1 Declaration

A parser is declared as a *tpg.Parser* class. The doc string of the class contains the definition of the tokens and rules.

7.2 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. When no return value is specified, TPG creates a variable named as the symbol. See figure 7.1 for example.

Figure 7.1: Rule declaration

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

    A <x, y> / ret_value_of_A

    B <y, z> / ret_value_of_B

    ;

S1 / $f(A,B)$ ->
    A         # return value of A stored in variable A
    B         # return value of B stored in variable B
;

S2 ->         # we can use S2 to compute the return value
    A         $ S2 = A
    B         $ S2 = f(S2, B)
    ;
```

7.3   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.

7.4   Parsing non terminal symbols

7.4.1   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.SyntacticError* 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 *Term* you can write:

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

7.4.2   In a rule

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

7.5   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 ;
```

7.6   Alternatives

Alternatives in grammar rules describe several possible decompositions of a symbol. The infix pipe operator (|) is used to separate alternatives. *A | 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. Empty choices in other positions will be reported as syntax errors.

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

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

7.7   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, the Python stack will not overflow.

7.8   Precedence and grouping

The figure 7.2 lists the different structures in increasing precedence order. To override the default precedence you can group expressions with parenthesis.

Figure 7.2: Precedence in TPG expressions

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

7.9   Actions

Grammar rules can contain actions as Python code. Python code is copied verbatim into the generated code and is delimited by `$...$`, `$...EOL`[1] or `{{...}}`.

Please be aware that indentation should obey Python indentation rules. See the grammar description for further information (see figure 5.2).

7.9.1   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 7.3 shows a node symbolizing a couple.

Creating an AST

AST are created in Python code (see section 7.9.1).

Updating an AST

When parsing lists for example it is useful to save all the items of the list. In that case one can use a list and its append method (see figure 7.4).

---

[1]EOL means End Of Line

Figure 7.3: AST example

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

class Foo(tpg.Parser):
    r"""
    COUPLE/$Couple(a,b)$ -> '(' ITEM/a ',' ITEM/b ')' ;

    # which is equivalent to
    # COUPLE/c -> '(' ITEM/a ',' ITEM/b ')' $ c = Couple(a,b) $ ;
    """
```

Figure 7.4: AST update example

```
class ListParser(tpg.Parser):
    r"""
    LIST/l ->
        '('                 $ l = []
            ITEM/a          $ l.append(a)
            ( ',' ITEM/a    $ l.append(a)
            )*
        ')'
        ;
    """
```

7.9.2 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.

7.9.3 Object

TPG 3 doesn't handle Python object as TPG 2. Only identifiers, integers and strings are known. Other objects must be written in Python code delimited either by $...$ or by {{...}}.

Argument lists and tuples

An 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 or in dollar signs. 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 *extract* method. Marks can be put in the input string by the *mark* method or the prefix operator.

```
    @beginning        # equivalent to $ beginning = self.mark()
    ...
    @end              # equivalent to $ end = self.mark()
    ...
    $ my_string = self.extract(beginning, end)
```

Getting the line and column number of a token

The *line* and *column* methods return the line and column number of the current token. If the first parameter is a mark (see 7.9.2) 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 module has a *WrongToken* exception for that purpose (see figure 7.5).

Figure 7.5: Backtracking with *WrongToken* example

```
    # NATURAL matches integers greater than 0
    NATURAL/n ->
        number/n
        $ if n<1: raise tpg.WrongToken $
        ;
```

Parsers have another useful method named *check* (see figure 7.6). This method checks a condition. If this condition is false then *WrongToken* is called in order to backtrack.

Figure 7.6: Backtracking with the *check* method example

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

A shortcut for the *check* method is the *check* keyword followed by the condition to check (see figure 7.7).

Figure 7.7: Backtracking with the *check* keyword example

```
# NATURAL matches integers greater than 0
NATURAL/n ->
    number/n
    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 7.8). This method raises a *SemanticError*.

Figure 7.8: Error reporting the *error* method example

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

A shortcut for the *error* method is the *error* keyword followed by the object to give to the *SemanticError* exception (see figure 7.9).

Figure 7.9: 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 5.3.1) turns TPG into a context sensitive lexer.

8.2 Grammar structure

CSL grammars have the same structure than non CSL grammars (see 5.1) except from the *lexer = CSL* option (see 5.3.1).

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.

8.3.2 Token matching

In CSL parsers, tokens are matched as in non CSL parsers (see 6.3).

8.4 CSL parsers

There is no difference between CSL and non CSL parsers.

Chapter 9

Debugging

9.1 Introduction

When I need to debug a grammar I often add print statments to visualize the parser activity. Now with TPG 3 it is possible to print such information automatically.

9.2 Verbose parsers

Normal parsers inherit from *tpg.Parser*. If you need a more verbose parser you can use *tpg.VerboseParser* instead. This parser prints information about the current token each time the lexer is called. The debugging information has currently two levels of details.

Level 0 displays no information.

Level 1 displays tokens only when the current token matches the expected token.

Level 2 displays tokens if the current token matches or not the expected token.

The level is defined by the attribute *verbose*. Its default value is 1.

Figure 9.1: Verbose parser example

```
class Test(tpg.VerboseParser):
    r"""

    START -> 'x' 'y' 'z' ;

    """

    verbose = 2
```

The information displayed by verbose parsers has the following format:

```
[eat counter][stack depth]callers: (line,column) <current token> == <expected token>
```

eatcounter is the number of calls of the lexer.

stackdepth is the depth of the Python stack since the axiom.

callers is the list of non terminal symbols called before the current symbol.

$(line, column)$ is the position of the current token in the input string.

$==$ means the current token matches the expected token (level 1 or 2).

$! =$ means the current token doesn't match the expected token (level 2).

Part III

Some examples to illustrate TPG

Chapter 10

Complete interactive calculator

10.1 Introduction

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

10.2 New functions

10.2.1 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 unary functions and *funct2* defines the name of binary functions. Finally the grammar rule of the atoms has been added a branch to parse functions. The *Function* non terminal symbol parser unary and binary functions.

10.2.2 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* contains the value of π and $cos(pi)$ returns -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).

10.3 Source code

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

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

import math
import operator
import string
import tpg

if tpg.__python__ == 3:
    operator.div = operator.truediv
```

```
    raw_input = input

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]

class Calc(tpg.Parser, dict):
    r"""
        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+' $ float
        token integer   '\d+'                                        $ int
        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)
    ;

    """

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

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

Chapter 11

Infix/Prefix/Postfix notation converter

11.1 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 notations. The expression is saved in a tree. Each node of the tree corresponds 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.

11.2 Abstract syntax trees

The AST of this converter has three 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).

class Func is used to store functions.

These 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.

11.3 Grammar

11.3.1 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 $a=Atom(s)$ | '\(' EXPR/a '\)'
       |  func1/f '\(' EXPR/x              '\)' $a=Func(f,x)
       |  func2/f '\(' EXPR/x ',' EXPR/y '\)' $a=Func(f,x,y)
;
```

11.3.2   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)
|   func1/f EXPR/x                      $ e=Func(f,x)
|   func2/f EXPR/x EXPR/y               $ e=Func(f,x,y)

;


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

11.3.3   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
    |   func2/f SEXPR_POST<$Func(f, e, e2)$>/e
    )
|   func1/f SEXPR_POST<$Func(f, e)$>/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.

11.4   Source code

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

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

# Infix/prefix/postfix expression conversion

import tpg
```

```python
if tpg.__python__ == 3:
    raw_input = input

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

class Func:
    """ Function expression """
    def __init__(self, name, *args):
        self.name = name
        self.args = args
        self.prec = 99
    def infix(self):
        args = [a.infix() for a in self.args]
        return "%s(%s)"%(self.name, ",".join(args))
    def prefix(self):
        args = [a.prefix() for a in self.args]
        return "%s %s"%(self.name, " ".join(args))
    def postfix(self):
        args = [a.postfix() for a in self.args]
        return "%s %s"%(" ".join(args), self.name)

# Grammar for arithmetic expressions

class ExpressionParser(tpg.Parser):
    r"""
```

```
separator space '\s+';
token func1 '\b(sin|cos|tan)\b' ;
token func2 '\b(min|max)\b' ;
token ident '\w+';

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

# 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)
    |   '\(' EXPR/a '\)'
    |   func1/f '\(' EXPR/x '\)'         $ a=Func(f,x)
    |   func2/f '\(' EXPR/x ',' EXPR/y '\)' $ a=Func(f,x,y)
;

# 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)
|   func1/f EXPR/x                       $ e=Func(f,x)
|   func2/f EXPR/x EXPR/y                $ e=Func(f,x,y)
;

# 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
    |   func2/f SEXPR_POST<$Func(f, e, e2)$>/e
```

```
        )
    |   func1/f SEXPR_POST<$Func(f, e)$>/e
    |   ;

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

    """

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