Rparse Manual

Romeo Dellarocco

Table of Contents

Rparse Manual

Introduction

“Hence, computing science is —and will always be— concerned with the interplay between mechanized and human symbol manipulation…”

— E. W. Dijkstra

The rparse parser generator can help simplify the tedious task of writing computer programs that can perform machine translation and analysis for users of the python programming language. Rparse can generate the language recognizer portion (the parser) of such programs.

Rparse generates LL(1) parsers in the python language. Users of the UNIX tools LEX and YACC will find some comforting similarities in the syntax of rparse grammar specification files.

Features

- Parsers automatically generate syntax trees.
- Parsers do not store state information between parses.
- Top-down parsing is used because it ensures that semantic actions are executed as soon as it is possible to do so.

Designing a parser with rparse

Rparse generates top-down parsers (LL(1) parsers). To generate a parser, rparse needs a grammar. Read on.

Specifying a grammar

Rparse generates parsers from the grammar specifications that you provide to it. As the input is processed by a rparse generated parser, it is accepted or rejected by attempting to match the input to the structure specified in the grammar spec.

Rparse parsers parse grammars that have a maximum complexity of LL(1). This type of parser has the feature that parsing decisions are made immediately during parsing. This allows semantic actions to control parsing as it occurs, based on context.

Rparse input files

A rparse input file is separated into three sections. Each section is separated by a line containing nothing but a '%%' (a double percent).

Input file sections

1. header
2. user code (copied verbatim to the output file)
3. grammar

Section 1 (the header section) is where you must specify the names of the terminal symbols of the language that your parser will recognize. The names must be all uppercase.

Ex:

```
[CODE]
%symbol FOO
%symbol BAR
```

Rparse will assign a numeric code to each name in the generated file 'symbols.py'.

Section 2 (the user code section) is copied word-for-word to the output file. Use this section to define variables, define functions, and import any python modules that your parser's semantic actions will use.

Section 3 (the grammar section) is where the productions, rules, and semantic actions of the language that is to be parsed are specified. Parsing starts with the first production that is specified.

If an action block appears before any productions in the grammar section, the code contained therein will be executed in the context of the the parser's **init** function. The **init** function will accept keyword arguments. They will be stored in the class instance variable named 'keywords'. Use this functionality to pass data to the class instances.

Specifying Productions and Rules

Productions are specified as:

```
production_name: definition ;
```

Production definitions are defined by rules. Alternative rules are separated by vbar '|' characters. Rules are defined by expressions followed by optional actions.

Rules are composed of terminal symbol names, production names, or expressions.

Terminal symbols are referred to by the names given to them in the %symbol directives in section 1 of the rparse input file.

Rparse expressions

Expressions are expressed in a form of Extended Backus-Naur Format (EBNF) notation as follows:

Names

The names of terminal symbols are composed of uppercase letters, such as: FOO

The names of productions may be mixed case, or lowercase. They may contain numbers after the first character, as in: `bar, bar2`

Special Names

Rparse reserves this name. You must not redefine it:

`EOF` ‘end of input’ symbol

Rparse Operators

All rparse operators are prefix operators. The operator symbol precedes its operand.

prefix operators affect parsing:

```
(exp)   grouping operator ( the '|' vbar is allowed in these.)
?exp    optional
*exp    closure
+exp    positive closure
```

The grouping operator is an easy way to keep a grammar spec concise. It allows alternative expressions as its operand.

Ex: This expression will match the terminal symbols FOO or BAR or BAZ:

```
(FOO | BAR | BAZ)
```

The ‘optional’ operator allows an expression operand to be matched or not matched.

The closure operator matches zero or more occurrences of its expression operand.

The positive closure operator matches one or more occurrences of its expression operator.

The simplest expressions are production names, and terminal names. Production names are recursively expanded to the corresponding production during parsing. Terminal symbol names match tokens that are returned by the parser’s scanner.

Semantic actions

Each production, rule, and EBNF expression may be followed by an optional action. The code that is contained therein will be executed when the parser recognizes the preceding expression.

An action is specified by a pair of braces, within which python code may be.

Ex:

```
[CODE]
{ print "I am a semantic action" }
```

Semantic actions may span more than one line. Indentation within semantic actions is consistent with itself.

Ex:

```
[CODE]
header_section: *header_option
    {
    generator_string = "nothing"
    for i in $1:
       if i[0] == "%generate":
        generator_string = dequote(unescape(i[1][0]))
    $$ = header_section_class(generator_string, $1)
    }
    ;
```

The indentation of the python code is referenced from the first line of actual python code. The absolute amount of indentation does not matter. Rparse will make sure that your code has the proper amount of indentation when rparse copies it to the output file.

It is strongly recommended that you use a python editor (such as IDLE) when you edit rparse input files in order to ensure that there are no nasty surprises with your indentation.

Expressions and Semantic Values

Each EBNF expression has an associated semantic value. Semantic values may be accessed in semantic actions by using the variables $$, $1, $2, $3 .. $N.

The variable $$ is the return value of the current production. $1 .. $N are the return values of each top-level positional subexpression in the corresponding rule of the corresponding production.

Default Values of Semantic Actions

Each expression type has a default semantic value. These values are passed up and up the parse tree as parsing occurs, generating a syntax tree.

Expression type, default value:

```
( ) , List
?   , the value of its subexpression or None
*   , List or None
+   , List
TERMINAL, capitalized names represent rparse_tokens (see rparselib/token.py)
nonterminal (production name), the value of the corresponding production.
```

A production will only have a default value when there is no user provided semantic action. If you provide a semantic action, you are responsible for setting the production's semantic value.

Ex: Explicitly setting the default value.

```
[CODE]
foo: BAR BAZ
{
    # By default, a list is made:
    $$ = [$1, $2] # [BAR, BAZ]
}
;
```

The grouping operator may contain an arbitrary number of subexpressions. It serves as a quick and easy way to reduce the number of productions in a grammar, but remember that it wraps its subexpression in a list. This can be confusing if the subexpressions also evaluate to lists.

Ex:

```
[CODE]
foo: (BAR BAZ) BOO
{
    # $1 is a list of BAR, BAZ
    # $2 is BOO

    # Flatten the list:
    $1.append($3)
    $$ = $1
}
;
```

Be warned! If there is an action, the automatic ast generation won't happen.

Ex:

```
[CODE]
foo: BAR BAZ { };
# Has the python null value: None
```

Rparse Invocation

If you have saved your parser specification in the file foo.gram, type the following at the shell prompt:

```
rparse foo.gram
```

This will generate:

- parser.py
- symbols.py

Rparselib Data Types

Rparse relies on a small run-time library, rparselib, which defines data types that are used by rparse parsers and scanners, namely the type of an rparse token, and rparse exception types.

To use rparselib data types, import them into your python program like this:

```
    import rparse3.rparselib.token as token
    import rparse3.rparselib.exceptions as exceptions
```

- rparse_token object:
    - symbol: integer
    - text: string
    - lineno: integer
    - colno: integer
- rparse exception types:
    - rparse_error: A generic rparse error

The Parser Interface

```
[CODE]
Parser:
    output interface:
        parse(scanner, debug=False, **keywords) --> ast
```

The result of parsing, the ‘ast’ (abstract syntax tree) is first defined by the structure of the grammar, and then by any ast modifications in semantic actions.

As you can see from the parser interface, a parser is required to have a scanner. How you make the scanner is up to you. You can use the supplemental scanner module that is included with rparse in the rparse/scanner directory, or you can write your own scanner as long as it conforms to the rparse scanner interface.

The Scanner Interface

Rparse parsers need to have a scanner available to them. You can use any scanner that supports this interface.

```
[CODE]
Scanner(file):
    output interface:
        get_token() --> instance of
        rparselib.token.rparse_token
        advance()   -- delayed advance. Delay calling
                    'file.read(1)' until 'get_token()'
                    is called so
                    that actions are executed asap.
        _advance()  -- consume input, call 'file.read()'
                    (it might block while waiting for I/O)

    operational interface:
        postfilter(token_types)
                -- set the scanner's built-in filter to
                   filter out token types in the list
                   or tuple
                   'token_types'. 'token_types' may
                   be empty.
```

Using an Rparse Parser in Your Programs

The simplest example of using a rparse parser:

```
[CODE]
import scanner
import parser

filename = "input.txt"
s = scanner.Scanner( open(filename) )
p = parser.Parser()

p.parse(s)
```

Here it is one more time with proper error checking:

```
[CODE]
import sys
import scanner
import parser
import rparselib

try:
    f = file("input.txt")
except IOError as e:
    sys.stdout.write(str(e))
    sys.exit(1)

s = scanner.Scanner(f)
```

```
p = parser.Parser()

try:
    p.parse(s)
except rparselib.exceptions.rparse_error, e:
    message = "Rparse error: file='%s': %s\n" \
          % (filename, e.message)
    sys.stderr.write(message)
```