Yappy
Yet another LR(1) parser generator for Python

Rogério Reis, Nelma Moreira
DCC-FC & LIACC, Universidade do Porto, Portugal

2000-2003

1 What is Yappy?

Yappy provides a lexical analyser and a LR parser generator for Python applications. Currently it builds SLR, LR(1) and LALR(1) parsing tables. Tables are kept in Python shelves for use in parsing. Some ambiguous grammars can be handled if priority and associativity information is provided.
Yappy can also be useful for teaching and learning LR parsers techniques, although for the moment no special functionalities are built-in, apart from some information in the Yappy API documentation.

2 How to use

To build a parser you must provide:

a tokenizer the rules for the lexical analyser

a grammar the grammar productions and the associated semantic actions

After a parser is generated it can be used for parsing strings of the language.
We assume familiarity with some basic notions of formal languages as regular expressions, context-free grammars, LR grammars and LR parsing [ASU86, HMU00, GJ90]. Some knowledge of the Python language [Lut96] is also required.

2.1 Lexer

The class Lexer implements a lexical analyser based on Python regular expressions. An instance must be initialised with a **tokenizer**, which is, a list of tuples:
`(re,funct,op?)`
where:

`re` is an uncompiled Python regular expression

`funct` the name of a function that returns the pair `(TOKEN, SPECIAL_VALUE)`, where `TOKEN` is the token to be used by the parser and `SPECIAL_VALUE` an eventual associated semantic value. If `funct` equals `""` the token is ignored. This can be used for delimiters. The argument is the string matched by `re`.

`op` if present, is a tuple with operator information: `(TOKEN,PRECEDENCE,ASSOC)` where `PRECEDENCE` is an integer and `ASSOC` the string `'left'` or `'right'`.

Restriction: if a keyword is substring of another its rule must appear after the larger keyword for the obvious reasons...

The following list presents a tokenizer for regular expressions:

```
l = [("\s+",""),
     ("@epsilon",lambda x: (x,x)),
     ("@empty_set",lambda x: (x,x)),
     ("[A-Za-z0-9]",lambda x: ("id",x)),
     ("[+]",lambda x: ("+",x),("+",100,'left')),
     ("[*]",lambda x: (x,x),("*",300,'left')),
     ("\(|\)",lambda x: (x,x)) ]
```

A lexical analyser is created by instantiating a `Lexer` class:

```
>>> from yappy.parser import *
>>> a=Lexer(l)
```

Information about operators may be retrieved from

2.1.1 Scanning

`Lexer` has two methods for scanning:

`scan()`: from a string

`readscan()`: from `stdin`

```
>>> from yappy.parser import *
>>> a=Lexer(l)
>>> a.scan("(a + b)* a a b*")
[('(', '('), ('id', 'a'), ('+', '+'), ('id', 'b'), (')', ')'),
('*', '*'), ('id', 'a'), ('id', 'a'), ('id', 'b'), ('*', '*')]
>>>
```

See `Yappy` documentation for more details.

2.2 LRparser

The class `LRParser` implements a `LR` parser generator. An instance must be initialised with:

`grammar`

`table_shelve` a file where the parser is saved

`no_table` if `0`, `table_shelve` is created even if already exists; *default* is `1`.

`tabletype` type of `LR` table: `SLR`, `LR1`, `LALR`; *default* is `LALR`

`operators` provided by `Lexer`

2.2.1 Grammars

A **grammar** is a list of tuples

`(LeftHandSide,RightHandSide,SemFunc,Prec)`

with

`LeftHandSide` nonterminal (currently a `string`)

`RightHandSide` a list of symbols (terminals and nonterminals)

`SemFunc` a semantic action

`Prec` if present, a pair `(PRECEDENCE,ASSOC)` for conflict disambiguation.

Restriction: The first production is for the `start` symbol.

Here is an unambiguous grammar for regular expressions:

```
grammar = [("r",["r","|","c"],self.OrSemRule),
                        ("r",["c"],self.DefaultSemRule),
                    ("c",["c","s"],self.ConcatSemRule),
                    ("c",["s"],self.DefaultSemRule),
                    ("s",["s","*"],self.StarSemRule),
                    ("s",["f"],self.DefaultSemRule),
                    ("f",["b"],self.DefaultSemRule),
                    ("f",["(","r",")"],self.ParSemRule),
                    ("b",["id"],self.BaseSemRule),
                    ("b",["@empty_set"],self.BaseSemRule),
                    ("b",["@epsilon''],self.BaseSemRule)]
```

The previous description can be easily rephrased in a more user-friendly manner. We provide two ways:

`grules()` function Allows the grammar productions being described as `Word -> Word1 ... Word2` or `Word -> []` (for empty rules), where the rule symbol and the separator of the `RHS` words can be specified.

The previous grammar can be rewritten as:

```
 grammar = grules([("r -> r | c",self.OrSemRule),
                   ("r -> c",self.DefaultSemRule),
                   ("c -> c s",self.ConcatSemRule),
                   ("c -> s",self.DefaultSemRule),
                   ("s -> s *",self.StarSemRule),
                   ("s -> f",self.DefaultSemRule),
                   ("f -> b",self.DefaultSemRule),
                   ("f -> ( r )",self.ParSemRule),
                   ("b -> id",self.BaseSemRule),
                   ("b -> @empty_set",self.BaseSemRule),
                   ("b -> @epsilon",self.BaseSemRule)])
```

We can also write an ambiguous grammar if we provided precedence information, that allows to solve conflicts (`shift-reduce`).

```
grammar = grules([("reg -> reg | reg",self.OrSemRule),
                 ("reg -> reg reg",self.ConcatSemRule,(200,'left')),
                 ("reg -> reg *",self.StarSemRule),
                 ("reg -> ( reg )",self.ParSemRule),
                 ("reg -> id",self.BaseSemRule),
                 ("reg ->  @empty_set",self.BaseSemRule),
                 ("reg ->  @epsilon",self.BaseSemRule)
                  ])
```

As a string that allows multiple productions for a left hand side:

```
  grammar ="""  reg -> reg + reg {{ self.OrSemRule }} |
              reg reg {{ self.ConcatSemRule // 200 left }} |
              reg * {{ self.StarSemRule }} |
              ( reg ) {{self.ParSemRule }} |
              id {{ self.BaseSemRule }};
           """
```

where:

`rulesym="->"` production symbol

`rhssep=''` RHS symbols separator

`opsym='//'` operator definition separator

`semsym='{{'` semantic rule start marker

`csemsym='}}'` semantic rule end marker

`rulesep='|'` separator for multiple rules for a LHS

`ruleend=';'` end marker for one LHS rule

The separators can be redefined in the `tokenizer` of `Yappy_grammar` class. An empty rule can be `reg ->;` . If no semantic rule is given, `DefaultSemRule` is assumed.

See `Yappy` documentation for more details.

2.2.2 Semantic Actions

As usual the semantic value of an expression will be a function of the semantic values of its parts. The semantics of a `token` is defined by the tokenizer 2.1. The semantic actions for grammar rules are specified by `Python`functions that can be evaluated in a given `context`. Our approach is essentially borrowed from the `kjParsing` package [rs00]: a semantic function takes as arguments a list with the semantic values of the `RightHandSide` of a rule and a `context` and returns a value that represents the meaning of the `LeftHandSide` and performs any side effects to the `context`. For instance, by *default* the semantic value of a rule can be the semantic value of the first element of the `RightHandSide`:

```
def DefaultSemRule(list,context=None):
   """Default  semantic rule"""
   return list[0]
```

Assuming the definition of some objects for regular expressions, t Trivial semantic rules for printing regular expressions can be:

```
def OrSemRule(self,list,context):
    return "%s+%s" %(list[0],list[2])

def ConcatSemRule(self,list,context):
    return list[0]+list[2]

def ParSemRule(self,list,context):
    return "(%s)" %list[1]

def BaseSemRule(self,list,context):
    return list[0]

def StarSemRule(self,list,context):
    return list[0]+'*'
```

2.2.3 Error handling

No error recovery is currently implemented. Errors are reported with rudimentary information, see the exception error classes in `Yappy` documentation.

2.2.4 Parser generation

Given the above information, a parser is generated by instantiating a `LRparser` class:

```
>>>from yappy.parser import *
>>>parse = LRparser(grammar,table,no_table,tabletype,operators)
```

Some information about LR table generated can be retrieved, by printing some attributes:

- The grammar rules can listed by `print parse.cfgr`
- The LR table (`ACTION` and `GOTO` functions) can be listed by `print parse`

```
>>> print parse.cfgr
0 | ('reg', ['reg', '+', 'reg'], RegExp2.OrSemRule, ('200', 'left'))
1 | ('reg', ['reg', 'reg'],RegExp2.ConcatSemRule, ('200', 'left'))
2 | ('reg', ['reg', '*'],RegExp2.StarSemRule)
3 | ('reg', ['(', 'reg', ')'], RegExp2.ParSemRule)
4 | ('reg', ['id'], RegExp2.BaseSemRule )
5 | ('@S', ['reg'], DefaultSemRule)
>>> print parse
Action table:

State
        +       *       (       )       id      $       #
0                       s1              s2
1                       s1              s2
2       r4      r4      r4      r4      r4      r4
3       s5      s6      s1              s2      r[]
4       s5      s6      s1      s8      s2
5                       s1              s2
6       r2      r2      r2      r2      r2      r2
7       r1      s6      r1      r1      s2      r1
8       r3      r3      r3      r3      r3      r3
9       r0      s6      r0      r0      s2      r0
 Goto table:
State
        reg     @S
0       3
1       4
3       7
4       7
5       9
7       7
9       7
```

If `_DEBUG` is set, several comments are printed during the table construction, in particular the collection of LR items.

Conflict resolution If the grammar is ambiguous, parsing action conflicts will be generated. If the `noconflicts` attribute is 0, only the precedence and associativity information will be used for **shift/reduce** conflict resolution. But if `noconflicts` is 1, conflicts will be resolved in the standard manner (for `yacc` like-parsers):

`shit/reduce` if precedence/associativity information is available try to use it; otherwise conflict is resolved in favor of **shift**. No messages will be given if the number of this type of conflicts is **exactly** the value of the `expect` attribute. The `expect` attribute can be set when some conflicts is legitimate.

`reduce/reduce` the rule listed first will be choosed

If any of these conflicts occurs, a list of the resolved conflicts are listed and more information can be found in the `Log` attribute. The `Log` has the following attributes:

`items` the set of LR items (`self.Log.items`) (not current available)

`conflicts` the `shit/reduce` (`sr`) and the `reduce/reduce` (`rr`) conflicts(`self.Log.conflicts`)

Currently no prettyprinting is available for these values.

2.2.5 Parsing

The method `parsing` accepts a list of `tokens` and a context and returns a parsed result:

```
>>>parse.parsing(a.scan("(a+b)*aab*"))
```

The attribute `output` records the grammar rules that were applied for parsing the string:

```
>>>parse.output
[4, 4, 0, 3, 2, 4, 1, 4, 1, 4, 1, 2]
```

If `_DEBUG` is set, it is possible to see each application of a table action and the values in the stack.

2.3 Yappy

The `Yappy` class is a wrapper for defining a parser and for parsing. Basically it creates the lexical analyser and the parser. This class is a subclass of `LRparser`.
It defines the following I/O functions:

`input` for inputing a string to be parsed: or as argument, or if not given, from `stdin`.

`inputfile` accepts input from a file

Here is a complete parser for regular expressions:

```
from yappy.parser import *

class ParseReg(Yappy):
     def __init__(self,no_table=0, table='tablereg'):
        grammar ="""
        reg -> reg + reg {{ self.OrSemRule }} |
               reg reg {{ self.ConcatSemRule // (200,'left') }} |
               reg * {{ self.StarSemRule }} |
               ( reg ) {{self.ParSemRule}} |
               id {{ self.BaseSemRule}} |
               @empty_set {{ self.BaseSemRule}} |
               @epsilon {{ self.BaseSemRule}} | ;
        """
        tokenize = [
        ("\s+",""),
        ("@epsilon",lambda x: ("id'',x),("id",400,'left')),
        ("@empty_set",lambda x: ("id",x),("id",400,'left')),
        ("[A-Za-z0-9]",lambda x: ("id",x),("id",400,'left')),
        ("[+]",lambda x: ("+",x),("+",100,'left')),
        ("[*]",lambda x: (x,x),("*",300,'left')),
        ("\(|\)",lambda x: (x,x)) ]
        Yappy.__init__(self,grammar,tokenize,table,no_table)

    ##Semantic rules build a parse tree...
    def OrSemRule(self,list,context):
        return "(%s+%s)" %(list[0],list[2])

    def ConcatSemRule(self,list,context):
        return "(%s%s)" %(list[0],list[2])

    def ParSemRule(self,list,context):
        return "(%s)" %list[1]

    def BaseSemRule(self,list,context):
        return list[0]

    def StarSemRule(self,list,context):
        return "(%s*)" %list[0]
```

An instance is used as:

```
>>> d = ParseReg()
>>> d = input("(a+b)*aab*")
>>> (a+b)*aab*
```

See `Yappy` documentation for more details.

3 Download

`Yappy` system requires Python (≥ 2.2). It was only tested in GNU Linux Systems.
It is available for download at http://www.ncc.up.pt/fado/Yappy.

4 History, current and future work

This project began in early 2000 as part of the `FAdo` project. Although several parser generator systems are now available for `Python` it seems that the ones implementing `LR` parsers are not very dynamic...
Some current/future work includes:

- extensions for GLR parsers
- semantic actions in the middle of rules
- EBNF grammar rules
- error recovery

And anyone who would like to collaborate/give suggestions is obviously welcome...

5 What's new

2004-03-25

- Corrected bug in `Yappy_grammar` semantic rules, concerning precedence values.
- If `_DEBUG` is set, the collection of items used in the generation of a LR table is printed.

6 Documentation

The `Yappy` api documentation can be found at http://www.ncc.up.pt/fado/Yappy/api/index.html Some simple demo parsers can be found in the file `demo.py` in the examples directory (for instance `/usr/share/doc/python2.2/examples`):

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[GJ90] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide.* Prentice Hall, 1990. Available for downloading in PostScript format.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 2nd edition, 2000.

[Lut96] M. Lutz. *Programming Python.* O'Reilly & Associates, 1996.

[rs00] Aaron Watte rs. *Parse Generation in Python.* New Jersey Institute of Technology, 2000.