A python grammar checker

Brett G. Giles

February 15, 2003

Contents

1 Introduction

This document is an example of parsing a grammar using pylex and pyyacc. In addition it gives step by step printouts of computing nullable, first and follow sets.

1.1 The Grammar for Grammar

The grammar for general grammars is defined by the following rules:

1 ⟨grpy.grm 1⟩≡

```
cfg -> rulelist .

rulelist -> rule rulelist
          | .

rule  -> NONTERM ARROW productionlist DOT .

productionlist -> production BAR productionlist
            | production.

production -> elementlist
             |  .

elementlist -> element
           | elementlist element .

element -> NONTERM | TERM .
```

This code is written to file `grpy.grm`.
Uses `element` 13, `production` 12, and `rule` 15.

and by the tokens as:

2a ⟨tokenlist 2a⟩≡
```
ARROW   ``->''

DOT      ``.''

BAR   ``|''

TERM  [A-Z]+

NONTERM [a-z_]+
```
This code is written to file `tokenlist`.

2 Lexical Scan and tokens

We only need a very simple lexer, with a few tokens. We start with importing the `lex` and `sys` modules.

2b ⟨grlex.py 2b⟩≡
```
#!/usr/bin/python
#
import lex,sys
⟨tokens 2c⟩
⟨patterns 3⟩
⟨otherLexCode 4⟩
```
This code is written to file `grlex.py`.

The set of tokens is straightforward, with names given as above.

2c ⟨tokens 2c⟩≡
```

tokens = (
     'NONTERM', 'TERM', 'DOT', 'ARROW', 'BAR'
     )

```
This code is used in chunk 2b.

The patterns section defines how we recognize each of the tokens. Each variable or method in this section is prefixed by `t_`. The portion following that must match the name of a token to be returned, or if it is not to return a token, can be a name of your choice. The special name `t_ignore` is implemented specially by pylex and should be used for dropping whitespace.

In the patterns themselves, we first ignore whitespace, including newlines. Following that are the recognition patterns for the tokens.

3 ⟨patterns 3⟩≡

```
# Completely ignored characters
t_ignore           = ' \t'

def t_mlcomment(t):
    r' /\*(.|\n)*?\*/'
    t.lineno += t.value.count('\n')



# Newlines
def t_newline(t):
    r'\n+'
    t.lineno += t.value.count("\n")

# Operators
t_DOT              = r'\.'
t_BAR             = r'\|'
t_ARROW        = r'->'
t_TERM        = r'[A-Z]+'
t_NONTERM =   r'[a-z_]+'

def t_error(t):
    print "Illegal character %s" % repr(t.value[0])
    t.skip(1)
```

This code is used in chunk 2b.

At this point, we initialize the lexing and create a function that returns a list of the tokens. The `tokenize` function accepts a string as input and returns the list of tokens found. Note that in a parser using python yacc, the function `tokenize` is not used. It is left here simply for use in the program when the lexer is run alone.

4 ⟨otherLexCode 4⟩≡

```
lex.lex()

def tokenize(data):
    lex.input(data)
    retval=[]
    while 1:
        tok = lex.token()
        if not tok: break      # No more input
        retval.append(tok)

    return retval

if __name__ == "__main__":
    # Test it out
    data = sys.stdin.read()
    # Give the lexer some input
    tkns = tokenize(data)

    # Tokenize
    for tok in tkns:
        print tok
        print
```

This code is used in chunk 2b.

Defines:

`tokenize`, never used.

3 Parsing.

We use py-yacc for parsing. See the documentation on the website for details of how the file is set up.

3.1 Python YACC file for parsing

The first file, `gryacc.py` is composed of production rules. The second file `grobjects.py` is a set of classes for the rules.

In the yacc file, we must:

- import the yacc module.
- bring in the tokens from the lexer we wrote.

Typically, this is done by importing the whole tokens file and then assigning `tokens` to have the same value as the `tokens` in the lexer file.

5 ⟨gryacc.py 5⟩≡

```
#!/usr/bin/python
import yacc
import grobj
import grlex
import sys

tokens = grlex.tokens

⟨rules 6⟩
⟨yaccmain 7⟩
```

This code is written to file `gryacc.py`.

3.2 Yacc rules

Rules for yacc consist of:

1. A method definition. The name is significant. Start with `p_` followed by the name on the left hand side of the rule you are looking at. You may then follow with further identification, normally only when considering rules with multiple right hand sides. In those cases, I split each right hand side into its own method and add `_n` for $n = 1, 2, \ldots$.
2. A doc string. The actual rule is specified in the doc string of the method, similar to what you already did for the lexing. The parameter `t` passed to the method is a list with at least as many members as there are terminals or nonterminals specified in the rule. For example in `p_cfg` below, `t` has 2 members, while in `p_rule` it has 4.
3. A body. Each method then assigns the result of the parse to `t[0]` using the other items of `t` as needed. In our example below, we either create a class instance (in `p_cfg, p_rule, p_production` and `p_element`) or we create /append to a list of these (in `p_rulelist_n`, `p_productionlist_n` and `p_elementlist_n`).

The special rule `p_empty` is used for recognition of empty productions.

6 ⟨rules 6⟩≡
```
def p_cfg(t):
    'cfg  : rulelist'
    t[0] = grobj.grammar(t[1])

def p_rulelist_1(t):
    'rulelist  : rule rulelist '
    t[0] = t[2]
    t[0].insert(0,t[1])

def p_rulelist_2(t):
    'rulelist  : empty '
    t[0] = []

def p_rule(t):
    'rule  : NONTERM ARROW productionlist DOT '
    t[0] = grobj.rule(t[1], t[3])

def p_productionlist_1(t):
    'productionlist : production BAR productionlist'
    if t[3] :
        t[0] = t[3]
        t[0].insert(0,t[1])
    else:
        t[0] = [t[1]]


def p_productionlist_2(t):
    'productionlist : production'
    t[0] = [t[1]]

def p_production_1(t):
    'production : elementlist'
    t[0] = grobj.production(t[1])

def p_production_2(t):
    'production : empty'
    t[0] = grobj.production([] )
```

```
def p_elementlist_1(t):
    'elementlist : elementlist element'
    if t[1] :
        t[0] = t[1]
        t[0].append(t[2])
    else:
        t[0] = [t[2]]

def p_elementlist_2(t):
    'elementlist : element'
    t[0] = [t[1]]

def p_element(t):
    '''element : NONTERM
               | TERM'''
    t[0] = grobj.element(t[1])

def p_empty(t):
    'empty :'
    pass

def p_error(t):
    print "Whoa. You are seriously hosed."
    # Read ahead looking for a closing '.'
    while 1:
        tok = yacc.token()             # Get the next token
        print tok
        if not tok or tok.type == 'DOT': break
    yacc.restart()
```

This code is used in chunk 5.
Uses `element` 13, `grammar` 8b, `production` 12, and `rule` 15.

Here, we assign the parser to `grparse` and write some code that can be run if the yaccer is called directly.

7 ⟨yaccmain 7⟩≡

```
grparse = yacc.yacc()


if __name__ == "__main__":
    s = sys.stdin.read()
    print s
    result = yacc.parse(s)
    print result
```

This code is used in chunk 5.
Defines:
`grparse`, used in chunk 17a.

3.3 Classes defined for grammar parsing.

This is the python file containing the basic classes used for parsing. Note that grammar constructions that are simply lists of other construction do not have a class specified. Rather, we simply use the native list type in Python.

8a ⟨grobj.py 8a⟩≡

```
#!/usr/bin/python

from types import *
import string

def isEltNull(elment):
    return elment.isNull()
def allnull(elmntlist):
    return reduce((lambda x, y : x and y ), (map(isEltNull,elmntlist)), 1)
⟨grammarClass 8b⟩
⟨ruleClass 15⟩
⟨productionClass 12⟩
⟨elementClass 13⟩
⟨helperclasses 16⟩
```

This code is written to file `grobj.py`.
Uses `isNull` 13.

The `grammar` class is our top level class, corresponding to the `p_cfg` rule in the parser. It contains the methods for determining the first and follow sets of the grammar.

8b ⟨grammarClass 8b⟩≡

```
class grammar:
⟨grammarinit 8c⟩
⟨grammarmisc 9a⟩
⟨computenull 9b⟩
⟨computeFirst 10⟩
⟨computeFollow 11⟩
```

This code is used in chunk 8a.
Defines:
`grammar`, used in chunk 6.

The initialization of the class includes saving the rules and creating a set of the terminals and non-terminals of the grammar. Flags stating whether nullable etc. calculations have been done are set to false.

8c ⟨grammarinit 8c⟩≡

```
    def __init__(self,rules):
        self.rules = rules
        self.nullDone = 0
        self.firstDone = 0
        self.followDone = 0
```

This code is used in chunk 8b.
Defines:
`firstDone`, used in chunks 10 and 11.
`followDone`, used in chunk 11.
`nullDone`, used in chunks 9b and 10.
`rules`, used in chunks 9–11.

We have one miscellaneous function in this class, the function that prints the computed table.

9a ⟨grammarmisc 9a⟩≡
```
def printtable(self):
    print  '%20s|%20s|%20s|%20s|'%('nonterm','nullable',
                                   'First','Follow')
    printed={}
    for rule in self.rules:
        if printed.has_key(rule.lhs):
            pass
        else:
            elmnt = rule.lhs
            printed[elmnt]=1
            if elmnt.elType() == 'N':
                print '%20s|%20s|%20s|%20s|'%(20*'-',20*'-',20*'-',20*'-')
                elmnt.fullprint('%20s|%20s|%20s|%20s|')
    print '%20s|%20s|%20s|%20s|'%(20*'-',20*'-',20*'-',20*'-')
    print
```

This code is used in chunk 8b.
Defines:
printtable, used in chunks 9–11.
Uses lhs 15, rule 15, and rules 8c.

computeNullable is our first worker function in grammar. The algorithm is:

repeat
  for each production $X \rightarrow Y_1, \ldots, Y_k$
    if all Y_i are nullable **or** $k = 0$
      $X.nullable \leftarrow$ **true**
until no changes.

9b ⟨computenull 9b⟩≡
```
def computeNullable(self):
    changed = 1
    self.printtable()
    while changed :
        changed = 0
        for rul in self.rules:
            x = rul.lhs
            if not x.isNull():
                for prodn in rul.productions():
                    pelmnts = prodn.pdnelements()
                    if allnull(pelmnts):
                        changed =  x.setNullable() or changed

        self.printtable()
    print
    print
    self.nullDone = 1
```

This code is used in chunk 8b.
Defines:
computeNullable, used in chunk 17a.
Uses isNull 13, lhs 15, nullDone 8c, pdnelements 12, printtable 9a, productions 15, rules 8c, and setNullable 13.

`computeFirst` is the second worker function in `grammar`. The algorithm is:

repeat
  for each production $X \rightarrow Y_1, \ldots, Y_k$
    for $i \leftarrow 1$ **to** k
      if all $Y_1, \ldots, Y_{i-1}$ are nullable **or** $i = 1$
        $X.first \leftarrow X.first \bigcup Y_i.first$
until no changes.

10 ⟨computeFirst 10⟩≡

```
    def computeFirst(self):
        if not self.nullDone:
            print 'Must compute nullable first.'
            return
        changed = 1
        self.printtable()
        while changed :
            changed = 0
            for rul in self.rules:
                x = rul.lhs
                for prodn in rul.productions():
                    pelmnts = prodn.pdnelements()
                    k = len(pelmnts)
                    for i in range(0,k):
                        if allnull(pelmnts[0:i]) :
                            changed = x.addToFirst(pelmnts[i].firstSet()) or changed
            self.printtable()
        self.firstDone = 1
```

This code is used in chunk 8b.
Defines:
`computeFirst`, used in chunk 17a.
Uses `addToFirst` 13, `firstDone` 8c, `lhs` 15, `nullDone` 8c, `pdnelements` 12, `printtable` 9a, `productions` 15, and `rules` 8c.

`computeFollow` is the final worker function in `grammar`. The algorithm is:

> **repeat**
>   **for** each production $X \rightarrow Y_1, \ldots, Y_k$
>   **for** $i \leftarrow 1$ **to** k
>   **if** all $Y_i, \ldots, Y_k$ are nullable **or** $i = k$
> $Y_i.follow \leftarrow X.follow \bigcup Y_i.follow$
>   **for** $j \leftarrow i + 1$ **to** k
>   **if** all $Y_{i+1}, \ldots, Y_{j-1}$ are nullable **or** $i + 1 = j$
> $Y_i.follow \leftarrow Y_i.follow \bigcup Y_j.first$
> **until** no changes.

11 ⟨computeFollow 11⟩≡
```
    def computeFollow(self):
        if not self.firstDone:
            print 'Must compute first sets before follow.'
            return
        changed = 1
        self.printtable()
        while changed :
            changed = 0
            for rul in self.rules:
                x = rul.lhs
                for prodn in rul.productions():
                    pelmnts = prodn.pdnelements()
                    k = len(pelmnts)
                    for i in range(0,k):
                        if allnull(pelmnts[i+1:]) :
                            changed = pelmnts[i].addToFollow(x.followSet()) or changed
                        for j in range(i+1,k):
                            if allnull(pelmnts[i+1:j]) :
                                changed =  pelmnts[i].addToFollow(pelmnts[j].firstSet()) or changed
            self.printtable()
        self.followDone = 1
```

This code is used in chunk 8b.
Defines:
`computeFollow`, used in chunk 17a.
Uses `addToFollow` 13, `firstDone` 8c, `followDone` 8c, `lhs` 15, `pdnelements` 12, `printtable` 9a, `productions` 15, and `rules` 8c.

This class is simply a element container.

12 ⟨productionClass 12⟩≡

```
class production:
    def __init__(self,elist):
        self.elts = elist

    def __repr__(self):
        retval = 'prodn:' + self.elts.__repr__() +"\n"
        return retval
    def pdnelements(self):
        return self.elts
```

This code is used in chunk 8a.
Defines:
`pdnelements`, used in chunks 9–11.
`production`, used in chunks 1 and 6.

This class has an interesting implementation similar to that of how a singleton class can be done in Python. Rather than a singleton, we want to be able to add new terminals and nonterminals as elements as we come across them. However, we want to identify all instances that are the same element. For example, if we created 5 instances, three with passing the string *expression* and two with the string *term*, there should be only two *distinct* instances of element.

To accomplish this, we use a class level dictionary that uses the elt string as its key. The value of this for any element is again a dictionary which contains what would normally be thought of as instance variable. In the actual instance, we keep the value of the key (`self.elt`) and a reference to the sub-dictionary that it points to (`self.me`).

The choice of a dictionary to hold the *instance variables* is not the only one. It could have been an instance of a subclass, a list or any other structure that would hold the required data.

Note that the use of *getter* and *setter* methods is highly encouraged as this makes the actual implementation of the class transparent to the rest of the program.

13 ⟨elementClass 13⟩≡

```
class element:
    '''The class is primarily a global dictionary. Whenver a new
    element is added, we add it to the dictionary, unless it is
    already there.'''
    elements={}
    def __init__(self,elt):
        if element.elements.has_key(elt):
            self.elt = elt
            self.me = element.elements[elt]
        else:
            self.elt = elt
            element.elements[elt]={}
            self.me=element.elements[elt]
            if elt[0] in string.uppercase:
                self.me['elementType'] = 'T'
                self.me['first'] = set(elt)
            else:
                self.me['elementType'] = 'N'
                self.me['first'] = set()
            self.me['nullable'] = 0
            self.me['follow'] = set()

    def addToFirst(self,terminalSet):
        '''Check if we have the members of terminalSet in
        our first set already. If so, return 0(False) as we
        have not changed anything. Otherwise, union the two sets
        and return true(1)'''
        if self.me['first'].contains(terminalSet):
            return 0
        else:
            self.me['first'].union(terminalSet)
            return 1

    def addToFollow(self,terminalSet):
        '''Check if we have the members of terminalSet in
        our follow set already. If so, return 0(False) as we
        have not changed anything. Otherwise, union the two sets
        and return true(1)'''
        if self.me['follow'].contains(terminalSet):
            return 0
        else:
            self.me['follow'].union(terminalSet)
            return 1
```

```
    def fullprint(self,fmtstring):
        print fmtstring%(self.elt,self.me['nullable'],self.me['first'],self.me['follow'])

    def setNullable(self):
        '''Check if we are already nullable. If so, return 0(False) as we
        have not changed anything. Otherwise, set to 1
        and return true(1)'''
        if self.me['nullable']:
            return 0
        else:
            self.me['nullable'] = 1
            return 1
    def isNull(self):
        return self.me['nullable']

    def firstSet(self):
        return self.me['first']

    def followSet(self):
        return self.me['follow']

    def elType(self):
        return self.me['elementType']

    def __str__(self):
        return self.elt
    def __repr__(self):
        return self.elt

    def __eq__(self,other):
        return self.elt == other.elt

    def __hash__(self):
        return self.elt.__hash__()
```

This code is used in chunk 8a.
Defines:
- `addToFirst`, used in chunk 10.
- `addToFollow`, used in chunk 11.
- `element`, used in chunks 1, 6, and 15.
- `elements`, never used.
- `elementType`, never used.
- `elt`, never used.
- `isNull`, used in chunks 8a and 9b.
- `me`, never used.
- `setNullable`, used in chunk 9b.

Uses `contains` 16, `members` 16, `set` 16, and `union` 16.

This class just holds the element on the left hand side and all the productions on the right hand side of a rule definition.

15 ⟨ruleClass 15⟩≡

```
class rule:
    def __init__(self,lhs,rhs):
        self.lhs = element (lhs)
        self.rhs = rhs

    def __repr__(self):
        retval = self.lhs.__repr__() + "->" + self.rhs[0].__repr__() +"\n"
        for x in range (1, len(self.rhs)):
            retval = retval + "        |" + self.rhs[x].__repr__() + "\n"

        return retval

    def productions(self):
        return self.rhs
```

This code is used in chunk 8a.
Defines:
  lhs, used in chunks 9–11.
  productions, used in chunks 9–11.
  rhs, never used.
  rule, used in chunks 1, 6, and 9a.
Uses element 13.

Unfortunately, Python does not have a built in "set" type. The distinguishing feature of a set is that it is a container class that allows only one copy of a particular item in it. This is most easily accomplished using a dictionary. For example, when adding new items to the set, we simply assign a value of 0 to the __store keyed by the item.

16 ⟨helperclasses 16⟩≡

```
class set:
    def __init__(self, item=None):
        if item:
            self.__store = {item:0}
        else:
            self.__store = {}

    def size(self):
        return len(self.__store)

    def members(self):
        return self.__store.keys()

    def contains(self,otherset):
        '''Does self already contain otherset?'''
        for x in otherset.members():
            if not  self.__store.has_key(x): return 0

        return 1

    def union(self,listSetElt):
        if type(listSetElt) == ListType:
            for i in listSetElt[:]:
                self.__store[i]=0
        elif  type(listSetElt) == InstanceType:
            if ("%s"%listSetElt.__class__) == 'grobj.set':
                for i in listSetElt.members():
                    self.__store[i]=0
            else:
                self.__store[listSetElt]=0

    def __repr__(self):
        '''Return representation of list of keys.'''
        return self.__store.keys().__repr__()

    def __contains__(self,item):
        return (self.__store.has_key(item))

    def __getitem__(self,key):
        '''Convert to list of keys and then index.'''
        return self.__store.keys()[key]
```

This code is used in chunk 8a.
Defines:
- `contains`, used in chunk 13.
- `members`, used in chunk 13.
- `set`, used in chunk 13.
- `size`, never used.
- `union`, used in chunk 13.

3.4 Driver file for running grammar

Import, parse, compute.

17a ⟨grammar.py 17a⟩≡

```
#!/usr/bin/python
#
from grobj import *
from gryacc import grparse
# get sys so we can access stdin
import sys

# alias the stdin file desciptor
s = sys.stdin.read()
cfgrammar = grparse.parse (s)

cfgrammar.computeNullable()
cfgrammar.computeFirst()
cfgrammar.computeFollow()
```

This code is written to file grammar.py.
Uses computeFirst 10, computeFollow 11, computeNullable 9b, and grparse 7.

4 Test data

A variety of data to ensure this thing works correctly.

17b ⟨sexp.grm 17b⟩≡

```
s_expression  -> atomic_symbol
              |  LPAR s_expression end_s_expression RPAR .

end_s_expression -> DOT s_expression
                 |  s_expressions .

s_expressions -> s_expressions DOT s_expression
              |     .

atomic_symbol  -> LETTER atom_part .

atom_part  -> LETTER atom_part
           |  NUMBER atom_part
           | .
```

This code is written to file sexp.grm.

5 Appendices

Chunk list

Index