Writing a Language Interpreter with Python Lex-Yacc (PLY)

Dr. Ralf Schlatterbeck
Open Source Consulting

Email: office@runtux.com
Web: http://www.runtux.com
Tel. +43/650/621 40 17

WTF: Why Basic?

- I've written an Antenna Simulator pymininec
- It uses an algorithm from the 1980s
- . . . originally implemented in Basic on the UNIVAC and later the Apple][and even later on the IBM-PC (the first machines came with a Basic ROM)
- The algorithm is still relevant today
- e.g. it can simulate tapered diameters in antenna elements
- . . . which the well-known NEC version 2 (numeric electromagnetics code) gets wrong
- pymininec is highly optimized (vectorized using numpy)

WTF: Why Basic?

- I needed something to test my code against
- Rob Hagemans PC-Basic emulates single-precision float and faithfully emulates the memory limits of the machines at the time
- Yabasi (Yet Another BASic Interpreter)
  - uses double precision for all numeric variables
  - has no memory limit (in the code)
- . . . and it turned out to be *a lot* faster (Factor 90)

On Grammars

- Different types of parsers for different grammars
- Left-to-right, leftmost derivation (LL) is the typical recursive-descend parser
- Left-to-right, rightmost derivation (LR) is a bottom-up parser
- A number in parentheses after LL or LR indicates the length of lookahead
- Any LR(k) grammar can be converted to LR(1)
- LR is slightly more expressive than LL
- PLY uses a LALR(1) parser (LA = look-ahead)
- Dragon book Aho, Sethi, Ullman: "Compilers"

Lexer vs. Parser

- Lexer reads input and feeds tokens to the parser
- Lexer uses regular expressions to define tokens
- Tokens have names, e.g. `MINUS` for '-'
- We use this for reserved words
- ... for function names in Basic (yech)
- ... for tokens like parentheses, comma, ...
- PLY's lexer either uses regular expressions
- ... or token functions (with regex in docstring)
- Tokens have a type and a value
- The type is the name in the parser
- The value is used in the parser (e.g. a constant)

Lexer: Token examples

```
t_NE = r'(<>)|(><)'
t_EQ = r'='
t_GE = r'>='

def t_HEXNUMBER(self, t):
    r'&[hH][0-9A-Fa-f]+'
    v = int(t.value[2:], 16)
    t.value = (v, int)
    return t
```

The prefix `t_` defines a token and is hard-coded in PLY's lexer

Parser: Tokens in Grammar

- Parser uses special grammar functions
- These have grammar production as function docstring
- Grammar function start with prefix `p_`
- Grammar functions get parameter `p`
- For tokens (from lexer) p[i] is t.value in lexer

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    # p[0]        p[1]       p[2] p[3]
```


Parser: Use a class

- We can put all the grammar function into a class
- Grammar function becomes class method
- So we can access our interpreter object

```
class Interpreter:
    def __init__ ...:
        self.lexer = lexer.Tokenizer()
        self.parser = yacc.yacc(
            module = self, debug = True)
        ...
    def p_expression_plus(self, p):
        'expression : expression PLUS term'
```

Passing Information Around

- `p[0]` is the return value of a grammar rule
- `p[i]` are the values coming from the right side
- This way we can, e.g., create an abstract syntax tree (AST)
- ... or other data structures during parsing
- Grammars can have different right sides
- ... in a single rule
- AST construction is detailed in PLY docs
- I'm proposing to create something like a "Concrete syntax tree": Executable code

Passing Information Around

```
def p_assignment_statement(self, p):
    """
        assignment-statement : lhs EQ expr
    """
    p[0] = ('assign', p[1], p[3])
def cmd_assign(self, lhs, expr):
    if callable(expr):
        result = expr()
    else:
        result = expr
    lhs().set(result)
```


Literals from Lexer

```
def p_literal(self, p):
    """
        literal : NUMBER
                | HEXNUMBER
                | STRING_DQ
                | STRING_SQ
    """
    p[0] = p[1][0]
```


Parsing Expressions

We create a function to be called during execution

```
def p_expression_literal(self, p):
    """
        expr : literal
    """
    p1 = p[1]
    def x():
        return p1
    p[0] = x
```

Parsing Expressions

For every expression dynamically create a function

```
def p_expression_twoop(self, p):
    """
        expr : expr PLUS   expr
             | expr MINUS  expr
             | expr TIMES  expr
             | expr EQ     expr
             | expr GT     expr
             | expr GE     expr
             ...
    """
```


Parsing Expressions

```
f1 = p[1]
f3 = p[3]
if p[2] == '+':
    def x():
        return f1() + f3()
elif p[2] == '-':
    def x():
        return f1() - f3()
...
p[0] = x
```


Operator Precedence

- As it stands now we would get a `shift-reduce conflict`
- And expression would be parsed right-to-left
- But we want $3 * 5 + 15 = 30$ (not 60)
- So we need to specify operator precedence

$\rightarrow$ Define the rules when we have no parentheses

- Associativity defines rules for the same operator
- Specify if an operator is right- or left associative
- $+$ and $*$ can be both $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- $-$ and $/$ are left-associative $a - b - c = (a - b) - c$
- An assignment operator is typically right-associative

Operator Precedence

First entries have lowest precedence

```
precedence = \
    ( ('left', 'AND', 'OR')
    , ('left',
       'LT',  'GT', 'LE', 'GE', 'NE', 'EQ')
    , ('left', 'PLUS',  'MINUS')
    , ('left',
       'TIMES', 'DIVIDE', 'MOD', 'INTDIV')
    , ('left', 'EXPO')
    , ('right', 'UMINUS')
    )
```

Parser Conflicts

- We can get two types of parse conflicts
- shift/reduce – is resolved in favor of shift
- reduce/reduce – typically an error in your grammar
- . . . the grammar is ambiguous
- Don't worry too much about shift/reduce: the default typically does what you want
- e.g. for nested `if/then/else` the `else` belongs to the innermost `if`
- Track down reduce/reduce conflicts, though

Putting it together

- Statements and expressions are parsed during "compile time"
- . . . and executed during run
- Each line is parsed separately (yukk, Basic)
- Expressions become nested python functions
- Compile each line into one or multiple statements
- There is a dict to look up next line number
- During execution we have executable python code
- This probably accounts for much of the speed gain compared to PC-Basic

Software

- github.com/schlatterbeck/yabasi
  Yet Another BASic Interpreter
- github.com/schlatterbeck/pymininec
  Antenna Simulation
- github.com/Kees-PA3KJ/MiniNec
  Ancient Mininec Basic Code
- github.com/schlatterbeck/mininec-3-doc
  Re-typeset version of original Mininec version-3 report (1986)