Parser Generators

Mark Boady

August 14, 2013

Interpreters

- We have seen many interpreters for different programming languages
- Every programming language has a grammar
- Math using integers, addition, and multiplication is a simple grammar
- We will use a parser generator to automatically generate a parser based on a grammar
- In PA3, we will see a full language implemented with a parser generator

Context Free Grammar

- What is a grammar for math with integers, addition, and multiplication

```
expr -> term + expr | term
term -> factor * term | factor
factor -> NUMBER | ( expr )
NUMBER -> [0-9]+
```

- How do we parse an expression?
- Example: 5+7*9
  - expr -> term + expr (term=5, expr=7*9)
  - Path 1: term=5
    - term -> factor (factor=5)
    - factor -> NUMBER (number=5)

Context Free Grammar

- Grammar

```
expr -> term + expr | term
term -> factor * term | factor
factor -> NUMBER | ( expr )
NUMBER -> [0-9]+
```

- Example: 5+7*9
  - expr -> term + expr (term=5, expr=7*9)
  - Path 2: expr=7*9
    - expr -> term (term=7*9)
    - term -> factor * term (factor=7, term=9)
    - factor -> NUMBER (number=7)
    - term -> factor (factor 9)
    - factor -> NUMBER (number=9)

Parser Generator

- Parser Generators for many languages exist
  - Python: PLY
  - C++: YACC/LEXX
  - Java: JFLEX
- We will be using PLY in this lab
- This gives us a way to quickly make a parser for our CFG
- In Lab 5, we will use math_example.py written using PLY
- The python file needs to be defined in a very specifc way.

PLY File

- Initialize Libraries
- At the top of the file we need to import the PLY libraries

```
import ply.lex as lex
import ply.yacc as yacc
import sys
```

Tokens

- The first thing to define are the tokens
- Tokens: Single characters or base symbols.
- Initially we just list the names we give out tokens.

```
tokens =(
    'NUMBER',
    'PLUS',
    'TIMES',
    'LPAREN',
    'RPAREN'
    )
```

Tokens

- After listing the tokens, we need to define what each token means.
- In PLY, we can use regular expressions to define tokens.

```
t_PLUS= r'\+'
t_TIMES= r'\*'
t_LPAREN= r'\('
t_RPAREN= r'\)'
t_ignore= ' \t' #Ignore whitespace
def t_NUMBER( t) :
    r'[0-9]+'
    t.value = int(t.value)
    return t;
```

Special Tokens

- There are two additional special tokens we need to define
- Error tells PLY what to do if it finds an illegal character

```
def t_error(t):

    print "Illegal character '%s' on line %d"
    % (t.value[0],t.lexer.lineno)
    return t;
```

- Newline tells PLY what you want your newline character to be.
- We can also could line numbers here for errors

```
def t_newline( t ):

    r'\n+'
    t.lexer.lineno += len(t.value)
```

Tokens

- After all tokens have been defined, we need to tell PLY to build part of the parser.
- The lexer uses a parse table to break the input string up into tokens (a tokenizer)

```
lex.lex()
```

Grammar

- After Defining the tokens, we can give the grammar
- We need to define each rule in our grammar

```
expr -> term + expr | term
```

- This requires 2 function definitions in PLY

```
def p_expr_a( p ):

    'expr : term PLUS expr'
    p[0] =p[1] + p[3]

p_expr_b( p ):

    'expr : term'
    p[0] = p[1]
```

Naming

- The grammar definition follows strict rules

```
def p_expr_a( p ):

    'expr : term PLUS expr'
    p[0] =p[1] + p[3]
```

- The function name must start with a p_
- The name you give the function must match the name used in the CFG string
- If you need multiple definitions, you can append _a, _b, etc

Definitions

- The grammar definition follows strict rules

```
def p_expr_a( p ):

    'expr : term PLUS expr'
    p[0] =p[1] + p[3]
```

- The first line of the definition gives the CFG to match
- The second line gives the action to perform on a match
- In this example
  - p[0] = the return value expr
  - p[1] = the value from term
  - p[2] = the string for +
  - p[3] = the value from expr

Multiplication

```
term -> factor * term | factor
```

- The rules for multiplication in PLY

```
def p_term_a( p ):
    'term : factor TIMES term'
    p[0] = p[1] * p[3]

def p_term_b( p ):
    'term : factor'
    p[0] = p[1]
```

Factors

- The factor symbol is used to make expressions in parenthesis act like numbers

```
factor -> NUMBER | ( expr )
```

- The rules for factor in PLY

```
def p_factor_a( p ):

    'factor : NUMBER'
    p[0] = p[1]

def p_factor_b( p ):

    'factor : LPAREN expr RPAREN'
    p[0] = p[2]
```

- Reminder: We know NUMBER returns an int from the token definition

Parser

- We need two more commands to finish the parser
- First we define how to handle errors

```
def p_error( p ):

    print "Syntax error in input!", str(p)
    sys.exit( 2 )
```

- Lastly, we build the actual parser

```
yacc.yacc()
```

Using the Parser

- The final part of the file defines the main function
- It reads from stdin and then calls the parser we generated

```
if __name__ == '__main__' :
    print "....''
    print "....";
    expression = sys.stdin.readline()
    while expression != "":

        result = yacc.parse(expression)
        print "The Answer is "+str(result)
        expression = sys.stdin.readline()
```

Lab 5

- Test the parser to see how it works for simple expressions
- Extend the Parser to handle exponents
  - 2^5
- Change the grammar
- Implement the new grammar you came up with
- If you don't finish you can submit by email