Recursive Descent Parsers

Mark Boady

August 20, 2013
Recursive Descent Parsers

- Last week, we experimented with parser generators
- This week, we will build a recursive descent parser
- PA3 will include build a recursive descent parser
  - Same Grammar as PA2
  - You may be able to reuse some of your logic for building diagrams
Grammar

expression -> term {+ term}
term -> factor {* factor}
factor -> number | ( expr )

- This grammar is an EBNF
- The brackets {} mean zero or more
The Class RDParse contains all the code for the recursive descent parser

We call the constructor and then parse an input string

```python
parser=RDParser()
res = parser.parse(somestring)
```
We can treat the input as a character array.

**getToken()**

- Finds the next non-whitespace character and stores in as self.token

```python
def getToken(self):
    #Skip Whitespace
    self.position+=1
    while self.input[self.position]==' ':
        self.position+=1
    self.token=self.input[self.position]
```
parse(expression)

- The parse command takes a string and returns the result of parsing and evaluating the string
- The call to getToken() loads the first character into self.token
- The command function starts the recursive descent parser
  - each line is a command

def parse(self, expression):

    # Initialize the input as a list
    self.input = list(expression)
    self.position = -1
    # Get the first character to start up the parser
    self.getToken()
    return self.command()
Command

- We are allowing one expression to be input per line
- command -> expression \n
- After parsing an expression, we need to check that there are no remaining characters

```python
def command(self):
    #command -> expr \n    result = self.expr()
    if self.token==="\n":
        return result
    else:
        print "Error: tokens after end of input"
```
The `expr` command parses an expression by parsing multiple terms.

As long as we keep seeing `+` symbols, the loop continues.

```python
def expr(self):
    #expr -> term { + term}
    result=self.term()
    while self.token == '+':
        self.match('+','+ expected')
        result +=self.term()
    return result
```
Match

- Match is an error checking command
- We expect to see a token, for example `self.match('+','+ expected')`
- If the token is found, we just call `getToken()` to skip it and move on
- If the token is not found, then there is a parse error

```python
def match(self, c, e):
    if self.token == c:
        self.getToken()
    else:
        print "Error: " + e
```
Term

- Term was called by expr
- It uses factor to handle its input
- This follows the grammar
- We keep looping as long as factors are being multiplied

```python
def term(self):
    #term -> factor { * factor}
    result = self.factor()
    while self.token == '*':
        self.match('**',"* Expected")
        result *= self.factor()
    return result
```
A factor is either a number or an expression with parenthesis.
We can check for parenthesis and assume number otherwise
It is important to remove the left parenthesis before recursively calling expr

def factor(self):
    #factor -> number | (expr)
    if self.token=="(":
        self.match("(( Expected")
        result = self.expr()
        self.match("\)\) Expected")
    else:
        result = self.number()
    return result
We don’t get regular expression matching in a recursive descent parser

We need to read all the digits and then convert the number to an integer

def number(self):
    #number -> [0-9]*
mystr=""
    while self.token.isdigit():
        mystr+=self.token
    self.getToken()
    return int(mystr)
The main function reads a line of input and calls our parser.

It continues until it reaches the EOF (Ctrl-D)

```python
if __name__ == '__main__' :
    print "...
    expression = sys.stdin.readline()
    while expression != "":
        parser=RDPParser()
        res = parser.parse(expression)
        print("The answer is "+str(res))
        expression = sys.stdin.readline()
```
You will download the recursive descent parser and test it
Extend the parser to allow exponents
PA3 - (Part 1) Implement a Recursive Descent Parser for Regular Expression
PA3 - (Part 2) Add conditionals to the Mini Language
Mini Language Overview Next Week
  Slides are already on my website if you want to start early