Calculadora “bc”

Bruno Kieling
Hariel Giacomuzzi
Lucas Teixeira
Faculdade de Informática - PUCRS
90619-900 - Porto Alegre - Brasil

31 de outubro de 2016

Resumo

Este artigo descreve a implementação dos analisadores léxico e sintático referentes a primeira entrega do trabalho da cadeira de Compiladores. Serão demonstradas como foram implementados ambos analisadores utilizando a ferramenta PLY[1], uma implementação do lex e yacc para Python.

Introdução

Este artigo retrata o desenvolvimento do trabalho da cadeira de Compiladores, este mesmo será entregue em duas datas, uma de forma parcial e outra de forma íntegra, portanto abordaremos somente a primeira parte do projeto.

Como desafio devemos implementar analisadores léxico e sintático para uma calculadora, definição de um sistema de erros para entradas inválidas, que devem emitir mensagens adequadas e retomar o processamento. Além disso deve ser feita uma especificação de estrutura da Tabela de Símbolos e da Árvore Sintática Abstrata (ASA).

Para a construção do léxico e do sintático foram atribuídas algumas especificações da calculadora, conhecida por “bc”, para esta implementação algumas regras serão simplificadas pelo professor.

Segue as regras:

1. Identificadores, usados para nomes de funções, argumentos e variáveis seguem a regra de formação da linguagem Java;
2. A calculadora trabalha apenas com os tipos numérico (double) e lógico (boolean), o tipo é inferido durante o processamento. Os valores numéricos são sempre apresentados com três casas decimais;
3. Embora possam ser declaradas variáveis globais (a partir de um comando de atribuição), os argumentos e variáveis utilizadas em uma função são sempre locais a esta. Funções recursivas são permitidas o que exigirá um certo cuidado para utilização destes elementos;
4. A calculadora contém três modos básicos de operação, a sintaxe específica de cada um pode ser encontrada na documentação, alterações são permitidas a critério do grupo, desde que explicadas no relatório entregue:
    a. Imediato: A expressão é compilada (gera uma ASA) e executada imediatamente. Exemplo: “2^3+5”. A expressão é avaliada e seu resultado mostrado na tela.

1.A implementação de PLY pode ser encontrada neste endereço: http://www.dabeaz.com/ply/

b. Atribuição: a expressão é compilada, executada, e armazenada associada ao identificador. Exemplo: f = 2^b+5
c. Declaração de função. A expressão é compilada e armazenada em uma tabela de funções para uso posterior. Exemplo: define d (n) { return (2*n); }
d. Controle:
    i. #help – mostra um pequeno auxílio ao uso da calculadora (conteúdo a critério do grupo)
    ii. #load “nome_arquivo”, permite carregar em memória um conjunto de declarações (declarações de funções)
    iii. declarações (declarações de funções) #save “nome_arquivo”, grava o conteúdo atual da tabela de funções
    iv. #show “ident” – apresenta os dados armazenados na tabela de valores/funções associadas ao identificador “ident”
    v. #show_all – apresenta todos os dados armazenados nas tabelas de valores/funções

5. Operadores válidos, valendo as regras de precedência e associatividade da linguagem Java:
    a. Aritméticos: +, -, *, /, ^ (potência, maior precedência e associativo à direita). Operandos devem ser numéricos e o resultado é numérico
    b. Relacionais: >, >=, <, <=, !=. Operandos numéricos e o resultado lógico.
    c. Lógicos: &&, ||, !. Operandos lógicos, resultado lógico
    d. Atribuição: =, +=, *=. Identificador do mesmo tipo do resultado da expressão.
    e. Condicional: “?:”. Expressão de teste lógica, outras duas do mesmo tipo.
    f. Sequência: “,” . Expressões são avaliadas sequencialmente e o resultado da lista é o resultado da última expressão avaliada.
6. Comandos
    a. Seleção (if)
    b. Repetição (while e for)
    c. Impressão (print)
7. Definição de funções.

Por uma questão de preferência entre o grupo que desenvolveu a solução estaremos fazendo a implementação em PLY.

Léxico

Antes de definirmos a estrutura do analisador léxico da calculadora “bc” devemos entender como este analisador funciona, e quais são suas principais funções dentro do escopo de Compiladores.

O analisador léxico é responsável pelo processo de analisar a entrada de linhas de caracteres e produzir seqüência de símbolos léxicos. Através do analisador léxico podemos verificar determinado alfabeto e quando analisamos uma palavra podemos definir se existe ou não caracteres que não fazem parte do alfabeto em questão.

Para que o analisador funcione, devemos implementá-lo. A partir deste ponto demonstraremos como foi implementado o nosso analisador léxico para a calculadora “bc”. Como já comentado a implementação que fizemos foi em Python utilizando a implementação de lex e yacc PLY.

Para a implementação de nosso interpretador definimos palavras reservadas e tokens que serão utilizados.

Palavras reservadas

```
# lista de palavras reservadas
reserved = {
   'if' : 'IF',
   'else' : 'ELSE',
   'while' : 'WHILE',
   'for' : 'FOR',
   'define' : 'DEFINE'
}
```

Tokens

```
# lista de nomes de tokens
tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
   'ATTR',
   'LESSTHAN',
   'GREATERTHAN',
   'UMINUS',
   'LCURLYBRACKETS',
   'RCURLYBRACKETS',
   'SEMICOLON',
   'ID',
   'COMMA'
]+list(reserved.values())
```

Após definidas as palavras reservadas e os tokens que serão utilizados em toda implementação do interpretador devemos identificar o que cada token deve reconhecer ao receber a entrada. Para isso usamos expressões regulares, que nos provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de especificação.

Expressões Simples

```
# expressões regulares para expressões simples
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_LESSTHAN = r'<'
t_GREATERTHAN = r'>'
t_UMINUS = r'-'
t_LCURLYBRACKETS = r'{'
t_RCURLYBRACKETS = r'}'
t_SEMICOLON = r';'
```

```
t_ATTR = r'='
t_ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_COMMA = r','
```

Também é necessário ignorarmos espaços e ‘tabs’:

```
# uma string para guardar caracteres descartados como espaços e
tabs...
t_ignore  = ' \t'
```

E para finalizarmos a identificação de alguns tokens separamos expressões que futuramente terão mais impacto na implementação da calculadora “bc”.

Expressões Complexas

```
# regra para identificar numeros inteiros
def t_NUMBER(t):
      r'\d+'
      return t
#  regra para mostrar erros...
def t_error(t):
      print("Illegal character '%s' on position: %s" % (t.value[0],
t.lineno))
# para manter o track do numedor de linhas
def t_newline(t):
      r'\n+'
```

Bom, agora que conseguimos identificar vários caracteres exigidos pelo exercício, espaços em branco, números, nova linha e caracteres ilegais, podemos começar a implementação de nosso sintático.

Sintático

Como fizemos com o analisador léxico, devemos também entender o analisador sintático. Este analisador tem como processo analisar uma sequência de entrada para determinar sua estrutura gramatical segundo uma gramática formal.

O analisador sintático é responsável por transformar um texto de entrada, por leitura ou teclado, em uma estrutura de dados, normalmente uma árvore. Este analisador está fortemente ligado ao analisador léxico que como já vimos é responsável por gerar um grupo de tokens, para que com eles o analisador sintático através de um conjunto de regras possa construir uma árvore sintática da estrutura.

Enquanto o analisador léxico nos proporcionou um desafio simples para a identificação apenas de caracteres, o analisador sintático foi um pouco mais complicado, pois não somente tínhamos mais que reconhecer caracter por caracter, mas agora temos de reconhecer sequências inteiras de tokens, palavras reservadas e produções e garantir que estejam de acordo com a gramática estabelecida para a calculadora “bc”. Que segue a seguir.

Gramática

```
################################################
```

```
GRAMÁTICA QUE ESTÁ IMPLEMENTADA
################################################

'''EXPR : NUMBER'''
            | ID ATTR NUMBER
        | ID ATTR BOOL
            | ID
            | LPAREN EXPR RPAREN
            | EXPR ATTR EXPR
            | EXPR PLUS EXPR
            | EXPR MINUS EXPR
            | EXPR DIVIDE EXPR
            | EXPR TIMES EXPR
            | EXPR PLUSEQUAL EXPR
            | EXPR MINUSEQUAL EXPR
            | EXPR TIMESEQUAL EXPR'''

'''LID : ID, LID
         | ID '''

'''BLOCK : LCURLYBRACKETS CMD RCURLYBRACKETS'''

'''OPER   : PLUS
          | MINUS
          | DIVIDE
          | TIMES
          | LESSTHAN
          | GREATERTHAN
          | ATTR'''

'''CMD   : EXPR SEMICOLON
         | EXPR SEMICOLON CMD
         | empty'''

'''DEFINES : DEFINE ID LPAREN LID RPAREN BLOCK'''

'''FORS : FOR LPAREN EXPR SEMICOLON EXPR SEMICOLON EXPR SEMICOLON
RPAREN BLOCK'''

'''empty :'''
```

Após a definição da gramática precisamos implementar o analisador sintático do interpretador, como estamos trabalhando com uma calculadora precisamos seguir as instruções do desafio, entre elas sermos capaz de fazer contas de adição, subtração, multiplicação e divisão, além de ser possível identificar formações como “+=”, “-=” entre outras funções, como função de atribuição.

Então como deve ficar nosso analisador sintático? Usamos expressões para identificar todas as possibilidades requeridas no desafio, de expressões básicas à complexas, então chegamos a tal fórmula:

Expressões

```
def p_EXPR_NUMBER(p):
  '''EXPR : NUMBER'''
```

```
def p_EXPR_ID_ATTR_EXPR(t):
  '''EXPR : ID ATTR NUMBER
            | ID ATTR BOOL'''

def p_EXPR_ID(p):
  '''EXPR : ID'''

def p_EXPR_LPAREN_EXPR_RPAREN(p):
  '''EXPR : LPAREN EXPR RPAREN'''

def p_EXPR_ATTR_EXPR(p):
  '''EXPR : EXPR ATTR EXPR'''

def p_EXPR_PLUS_EXPR(p):
  '''EXPR : EXPR PLUS EXPR'''

def p_EXPR_MINUS_EXPR(p):
  '''EXPR : EXPR MINUS EXPR'''

def p_EXPR_DIVIDE_EXPR(p):
  '''EXPR : EXPR DIVIDE EXPR'''

def p_EXPR_TIMES_EXPR(p):
  '''EXPR : EXPR TIMES EXPR'''

def p_EXPR_PLUSEQUAL_EXPR(p):
  '''EXPR : EXPR PLUSEQUAL EXPR'''

def p_EXPR_MINUSEQUAL_EXPR(p):
  '''EXPR : EXPR MINUSEQUAL EXPR'''

def p_EXPR_TIMESEQUAL_EXPR(p):
  '''EXPR : EXPR TIMESEQUAL EXPR'''

def p_LID(p):
  '''LID : ID'''

def p_BLOCK(p):
  '''BLOCK : LCURLYBRACKETS CMD RCURLYBRACKETS'''
  pass

def p_OPER(p):
  '''OPER : PLUS
          | MINUS
          | DIVIDE
          | TIMES
          | LESSTHAN
          | GREATERTHAN
          | ATTR'''

def p_CMD(p):
  '''CMD : EXPR SEMICOLON
         | EXPR SEMICOLON CMD
         | empty'''
  pass

def p_DEFINES(p):
```

```
  '''DEFINES : DEFINE ID LPAREN LID RPAREN BLOCK'''
  pass

def p_FORS(p):
  '''FORS : FOR LPAREN EXPR SEMICOLON EXPR SEMICOLON EXPR SEMICOLON
RPAREN BLOCK'''
  pass
```

Além de expressões para o funcionamento das regras também precisamos identificar erros e garantir que quando o analisador sintático realmente encontre vazio quando é necessário.

Identificando erros

```
# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Empty rule for the sake of needing
def p_empty(p):
    'empty :'
    pass
```

Após a implementar o analisador sintático nós já começamos a implementação do analisador semântico.

Conclusão

A escolha da implementação lex e yacc, PLY foi de grande ajuda ao grupo, mesmo sendo necessário estudar sua documentação, tivemos uma melhor compreensão da matéria em questão para o desenvolvimento deste trabalho. Muitos desafios foram encontrados enquanto implementamos a calculadora “bc”, entre eles definir algumas produções para que funcionassem de forma correta.

Para o restante do projeto já estamos no caminho para a implementação do analisador semântico do mesmo.