INSTITUTO TECNOLÓGICO DE AERONÁUTICA - ITA

2º PROJETO DE CTC - 34

Projeto 4: Analisador sintático bottom-up de GLC na FNG

INTEGRANTES

André Simão - andregsimao@gmail.com

Felipe Tuyama - ftuyama@gmail.com

Matheus Leão - leaomatheus11@gmail.com

Marco Aurélio - marco.pdsv@gmail.com

PROFESSORES

Carlos Henrique Ribeiro carlos@ita.br

Carlos Henrique Q. Forster forster@ita.br

São José dos Campos, 19 de outubro de 2015

1 INTRODUÇÃO

O projeto consiste no desenvolvimento de um software *Parser Bottom-Up* na linguagem de programação Python (versão 2.7). A partir da leitura da especificação de uma gramática GLC (Gramática Livre de Contexto) na FNG (Forma Normal de Greibach) e de uma cadeia, o programa indica a pertinência ou não desta cadeia à linguagem produzida pela GLC. Em caso positivo, informa também a derivação correspondente.

2 METODOLOGIA

(To carteando uma metodologia, se quiserem mudar depois...)

O desenvolvimento do projeto consistiu nas seguintes etapas:

1. Estudo do algoritmo
2. Definição da entrada e saída do programa
3. Implementação do algoritmo na linguagem Python 2.7

2.1 Estudo do algoritmo

O algoritmo consiste em metodologia inversa à análise top-down. Parte-se de uma cadeia e, através de reduções, busca-se atingir o símbolo inicial da gramática. Se não é possível atingir o símbolo inicial, então a cadeia não faz parte da linguagem gerada pela gramática.

A busca é em largura, percorrendo a árvore de reduções em níveis. Em geral, há menos passos que a metodologia top-down, uma vez que há menos possibilidades de trocas. Desconsiderado as produções que resultam em cadeia vazia, a árvore possui no máximo n níveis, sendo n o tamanho da cadeia lida.

Para considerar as produções em cadeia vazia, adaptou-se a gramática para algumas produções de gramática irrestrita, uma vez que, com exceção de quando se parte da cadeia inicial, existe um símbolo terminal à esquerda. Por exemplo, seja a gramatica com produções A -> vazio e S -> aA. É possível considerar as produções S -> aA e aA -> a, ou seja, sendo Produção inversa[a] = aA. A exceção ocorre quando a produção é S -> vazio, e nesse caso, trata-se como caso a parte.

A fim de reconstruir as derivações, para cada forma sentencial atingida. é salvada a forma anterior que a originou e basta percorrer o caminho inverso a partir de S caso exista.

No caso geral, a pilha possui O(n+m) elementos, mas o algoritmo pode ter complexidade exponencial em função do número de níveis da árvore n e de m, sendo n o tamanho da cadeia lida e m o número de produções.

2.2 Definição da entrada e saída do programa

2.2.1 Entrada

A entrada do programa desenvolvido é composta por dois arquivos txt, GLC.txt e string.txt, que especificam a Gramática Livre de Contexto na forma Normal de Greibach e a cadeia a ser testada, respectivamente. É suposto que os dois arquivos de entrada estejam no mesmo diretório do programa desenvolvido.

2.2.1.1 Arquivo GLC.txt

As duas primeiras linhas do arquivo GLC.txt contém, separados por espaço, os símbolos não terminais (1º linha) e os símbolos terminais (2º linha), a terceira linha contém o símbolo inicial, cada uma das linhas seguintes contém uma produção da gramática.

Exemplo de arquivo GLC.txt:

```
>S P
>a b + -
>S
>S a
>S b
>S aPS
>S bPS
>P +
>P -
>
```

Pode-se observar no exemplo acima, 6º linha, a produção “S aPS”. Isso significa que S produz aPS. Para representar a produção vazia *(A → ε)*, apenas o símbolo não terminal A é escrito, “A”.

É suposto que cada símbolo (terminal ou não) é representado por um único caractere, diferente de ‘\n’ e ‘ ’, da tabela ASC II.

2.2.1.2 Arquivo string.txt

O arquivo string.txt possui uma única linha contendo a cadeia a ser testada.

Exemplo de arquivo string.txt:

>a+b-a

2.2.2 Saída

A primeira linha da saída do programa é composta por um único caractere:

‘0’: A cadeia fornecida não pertence à linguagem da Gramática
‘1’: A cadeia fornecida pertence à linguagem da Gramática

Caso a cadeia fornecida pertença à linguagem da Gramática, a segunda linha indica a derivação correspondente.

1º Exemplo de saída:

>0
>

2º Exemplo de saída:

>1
>S aPS a+S a+bPS a+b-S a+b-a

2.3 Implementação do algoritmo na linguagem Python 2.7

O código desenvolvido está comentado e encontra-se na seção 6, Apêndice.

3 CONCLUSÃO

A implementação realizada nesse trabalho de um algoritmo para verificação se uma determinada cadeia pertence a uma GLC e escrita da possível derivação correspondente a essa cadeia tem diversas aplicações práticas. Tais linguagens são importantes para definir linguagens de programação. Por exemplo, as linguagens que requerem o balanceamento de parênteses e a maioria das expressões aritméticas é gerada por gramáticas livres de contexto. Além disso gramáticas livre-de-contexto são úteis a modelar a segunda estrutura do RNA . A estrutura secundária do RNA envolve Nucleotídeos dentro de uma molécula de RNA de filamento único, que são complementares entre si e, portanto , pares de bases. Este emparelhamento de bases é

biologicamente importante para o bom funcionamento da molécula de RNA. Grande parte desse emparelhamento de bases pode ser representado em uma gramática livre-de-contexto.

4 REFERÊNCIAS BIBLIOGRÁFICAS

[1] RIBEIRO, C. H; FORSTER, C. H. Q. Slide 6: CTC-34 Automata e Linguagens Formais, 1-15 de outubro de 2015. Notas de Aula.

5 TESTES

teste1.txt

S P

a b + -

S

S a

S b

S aPS

S bPS

P +

P -

cadeia1.txt

a+b-a

resultado esperado:

Cadeia está de acordo com a GLC

S -> aPS -> a+S -> a+bPS -> a+b-S -> a+b-a

teste2.txt

S P

a b + -

S

S a

S b

S aPS

S bPS

P +

P -

cadeia2.txt

b++b

resultado esperado:

Cadeia não está de acordo com a GLC

teste3.txt

S A P T E F

a b + - * ( ) ^

S

S aE

S bE

S aPS

S bPS

S (SF

F)

P +

P -

P *

E ^S

E

cadeia3.txt

(a*b+a^(a-b))

resultado esperado:

Cadeia está de acordo com a GLC

S -> (SF -> (aPSF -> (a*SF -> (a*bPSF -> (a*b+SF -> (a*b+aEF -> (a*b+a^SF -> (a*b+a^(SFF -> (a*b+a^(aPSFF -> (a*b+a^(a-SFF -> (a*b+a^(a-bEFF -> (a*b+a^(a-bFF -> (a*b+a^(a-b)F -> (a*b+a^(a-b))

teste4.txt

S

a

S

S

cadeia4.txt (vazia)

resultado esperado:

Cadeia está de acordo com a GLC

S -> (produz vazio)

6 APÊNDICE

O código a seguir foi desenvolvido em Python 2.7.

```
#
#     Analisador sintático bottom-up de GLC na FNG
#
#  Autores: André Simão
#           Felipe Tuyama
#           Matheus Leão
#           Marco Aurélio
import sys

# Implementação da Fila


class Fila(object):
```

```
    def __init__(self):
        self.dados = []

    def insere(self, elemento):
        self.dados.append(elemento)

    def remove(self):
        return self.dados.pop(0)

    def vazia(self):
        return len(self.dados) == 0

    def length(self):
        return len(self.dados)

    def first(self):
        return self.dados[0]

    def log(self):
        print "Imprimindo a fila:"
        for i in range(0, len(self.dados)):
            self.dados[i].log()

# Armazena a arvore de derivaçoes para imprimir a derivação que produz
# a cadeia fornecida na entrada


class Noh(object):

    def __init__(self, w, father):
```

```
        self.w = w
        self.father = father

    def log(self):
        # print "self.w = " + str(self.w)
        if self.father:
            print str("father[ " + str(self.w) + " ] = " + str(self.father.w))
        else:
            print str("father[ " + str(self.w) + " ] = " + "None")

# Leitura da Gramática na FNG:


def readGrammar():
    global reader
    global V
    global Sig
    global S
    grammarFile = open('GLC.txt', 'r')
    # Cada linha do arquivo é posta em uma lista:
    reader = grammarFile.readlines()

    # Construindo V
    V = str(reader[0]).split(' ')
    V[-1] = V[-1].strip()

    # Construindo Sig
    Sig = str(reader[1]).split(' ')
    Sig[-1] = Sig[-1].strip()

    # Construindo S
```

```
S = reader[2]
S = S.strip()

# Construindo P
for i in range(3, len(reader)):
    prod = str(reader[i]).split(' ')
    prod[-1] = prod[-1].strip()
    if(len(prod) == 1):
        prod = prod + [""]
    if not prod[0] in P:
        P[prod[0]] = [prod[1]]
    else:
        P[prod[0]] = P[prod[0]] + [prod[1]]

# Construindo Pinv
for i in range(3, len(reader)):
    prod = str(reader[i]).split(' ')
    prod[-1] = prod[-1].strip()
    if len(prod) == 1:
        prod = prod + [""]
    if not prod[1] in Pinv:
        Pinv[prod[1]] = [prod[0]]
    else:
        Pinv[prod[1]] = Pinv[prod[1]] + [prod[0]]

# Adicionando as produções indiretas obtidas através das produção vazia
if "" in Pinv:
    for i in range(3, len(reader)):
        prod = str(reader[i]).split(' ')
        prod[-1] = prod[-1].strip()
        if len(prod) == 2:
```

```
            s = prod[1]
            nDeriv = 0
            for j in range(0, len(prod[1])):
                if s[j-nDeriv] in Pinv[""]:
                    lastS = s
                    s = s[:j-nDeriv] + s[j+1-nDeriv:]
                    if not s in Pinv:
                        Pinv[s] = [lastS]
                    else:
                        Pinv[s] = Pinv[s] + [lastS]
                    nDeriv += 1

    grammarFile.close()


def testReadGrammar():
    print "V:"
    print V
    print "Sig:"
    print Sig
    print "S:"
    print S
    print "P:"
    print P
    print "Pinv:"
    print Pinv
    print "PIndDeriv:"
    print PIndDeriv
    print "PInvIndDeriv:"
    print PInvIndDeriv
```

```
# Leitura da Cadeia a ser analisada:


def readCadeia():
    global reader
    stringFile = open('string.txt', 'r')
    # Generalizar para várias cadeias?
    string = stringFile.readline()
    stringFile.close()
    return string

# Verifica se existe uma redução válida


def addReduction(w, father):
    F.insere(Noh(w, father))


def stringReduction(n):
    w = n.w
    newReduction = ""
    Pointer = len(w) - 1
    if Pointer < 0:  # Caso w seja a cadeia vazia ("")
        return

    Pointer = len(w) - 1
    while not w[Pointer] in Sig:
        Pointer -= 1
        if Pointer < 0:
            return
```

```
    # Para cada regra de Produção:
    for j in range(Pointer+1, len(w) + 1):
        # Procuro uma derivação direta que produza parte de w:
        if w[Pointer:j] in Pinv:
            # Expando o nó, colocando os filhos na Fila:
            # Esse novo for (em k) é para considerar dois simbolos distintos
            # produzindo a mesma coisa
            # Ex.: A -> aB && B -> aB
            for k in range(0, len(Pinv[w[Pointer:j]])):
                addReduction(w[:Pointer] + Pinv[w[Pointer:j]][k] + w[j:], n)

# Parser bottom-up da cadeia w:


def BFSparser():
    F.insere(Noh(string, None))
    # F.log()
    while not F.vazia():
        #F.log()
        if (F.first().w == S):
            return F.first()
        # print F.dados
        stringReduction(F.first())
        F.remove()
    return None

# Imprime a sequencia de derivações para se chegar na cadeia fornecida


def printPath(n):
    s = ""
```

```
    while n.father != None:
        s += n.w
        s += " "
        n = n.father
    s += n.w
    print s


# Dados da nossa gramática (Rascunho, pode mudar):
V = []      # Lista de Símbolos não Terminais
Sig = []    # Lista de Símbolos Terminais
S = ""      # Símbolo inicial S pertencente a V

P = {}    # Matriz de Produções:
# Agora P é um dicionário
# P["S"] é a lista de produções do Símbolo não Terminal S (número 0)
# P["A"] é a lista de produções do Símbolo não Terminal A (número 1)
# ...  (assim por diante)

# Por exemplo:
# Se S-> a  &&  S-> aB && A-> b
# Teríamos
# P["S"] = [ "a" , "aB" ]
# P["A"] = [ "b" ]
# P =   {
#           "S": [ "a" , "aB" ],
#           "A": [ "b" ]
#       }

Pinv = {}  # Semelhante à matriz de Produções P, só que na ordem inversa:
# Agora as keys são a parte direita de cada produção e os values
```

```
# são a parte esquerda da produção
# Por exemplo:
# Se S-> a  &&  S-> aB && A-> b
# Teríamos
# Pinv["a"] = "S"
# Pinv["aB"] = "S"
# Pinv["b"] = "A"
# Pinv =    {
#             "a": "S",
#             "aB": "S",
#             "A":
#         }
# Essa Pinv vai ser útil para o BFSparser
# As produções vazias estão em Pinv = {"": ["A", "B"], ...}
# Portanto Pinv[""] representa a lista de simbolos que produzem
# a cadeia vazia

F = Fila()  # Fila usada para o BFS

# Código principal - Rotina main()
# print "*************************"
# print "*   Parser bottom-up     *"
# print "*************************"
readGrammar()
#testReadGrammar()
string = readCadeia()      # Cadeia a ser analisada
if not string:                                  # Caso especial, cadeia vazia
   if ("" in P[S]):
      print 1
      print S
   else:
```

```
        print 0


else:
    result = BFSparser()
    if result:
        print 1
        #print "[Cadeia pertence a linguagem da Gramatica]"
        printPath(result)
        # Informar a derivação
else:
    print 0
    #print "[Cadeia nao pertence a linguagem da Gramatica]"
# print "**************************"
```