Universidade do Minho
Licenciatura em Engenharia Informática

Grupo 11

Maio 2022

Processamento de Linguagens
Trabalho Prático 2: RecDesc

Duarte Moreira (a93321) Lucas Carvalho (a93176) Ricardo Gama (a93237)

Índice

1 Introdução

Na realização do segundo trabalho prático da unidade curricular de Processamento de Linguagens tivemos como principal objectivo aumentar a nossa experiência em linguagens e em programação gramatical, reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT).

Desenvolver processadores de linguagens, a partir de uma gramática tradutora, utilizar geradores de compiladores baseados em gramáticas tradutoras completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python foram alguns dos outros objetivos que este trabalho visava.

Após a escolha do enunciado, **RecDesc**, o foco principal seria escrever um parser recursivo descendente capaz de reconhecer uma linguagem LL(1). Assim, neste projeto começamos por criar uma linguagem para descrição de gramáticas independentes de contexto e um reconhecedor para essa linguagem que fosse capaz de verificar se a linguagem é LL(1) ou não. De seguida, começamos a gerar o código Python que implementa o parser recursivo descendente.

2 Desenho e Implementação da Solução

O primeiro passo para o desenvolvimento deste trabalho prático passava por criar uma linguagem para descrição de gramáticas independentes de contexto. O grupo optou por abordar o tema do futebol e criou uma gramática que caraterizasse um plantel de futebol:

```
G = {T,N,P,S}
T = {'[', ']', ',', 'Nome:', 'Posicoes:', 'id', 'GK', 'LAT',
     'CEN', 'MED', 'EXT', 'PL'}
N = {Plantel, Jogadores, Jogador, Cont1, Nome,
     Posicoes, Posicao, Cont2}
S = Plantel
P ={
    Z          -> Plantel '$'
    Plantel    -> '[' Jogadores ']'
    Jogadores  -> Jogador Cont1
    Cont1      -> ',' Jogador Cont1
                |
    Jogador    -> "Nome:" Nome "Posicoes:" '[' Posicoes ']'
    Nome       -> id
    Posicoes   -> Posicao Cont2
    Cont2      ->  ',' Posicao Cont2
                |
    Posicao    ->  GK | LAT | CEN | MED | EXT | PL
    }
```

Após a criação da gramática, o próximo passo foi verificar se a linguagem seria LL(1). Para esta verificação foi importante reconhecer aquelas que são as regras principais que fazem uma gramática LL(1):

Definition 1 (LL(1) Condition) *A grammar* G = (T,N,S,P) *satisfies LL(1) condition if and only if:*

$$\forall_{A\to\alpha_1, A\to\alpha_2} : lookahead(A \to \alpha_1) \bigcap lookahead(A \to \alpha_2) = \emptyset$$

Figura 1: Definition 1

Definition 2 (Lookahead(1)) *The Lookahead(1) set of Terminal symbols of a production $p \in P$ is defined in the following way:*

$$lookahead(A \to \alpha) = First(\alpha) \bigcup \begin{cases} \emptyset & , \alpha \not\Rightarrow^* \epsilon \\ \boldsymbol{Follow(A)} & , \alpha \Rightarrow^* \epsilon \end{cases}$$

Figura 2: Definition 2

Definition 3 (First(1)) *The First(1) set of Terminal symbols of a Terminal , Non-terminal symbol or a String (a sequence of symbols) is defined follows:*

1. $First(\epsilon) = \emptyset$
2. $First(t) = \{t\},\ t \in T$
3. $First(A) = \bigcup_{A \to \beta_i} First(\beta_i),\ A \in N$
4. $First(\alpha) = First(X) \bigcup \begin{cases} \emptyset & , X \not\Rightarrow^* \epsilon \\ First(\alpha') & , X \Rightarrow^* \epsilon \end{cases} , \alpha = X\alpha'$

Figura 3: Definition 3

Definition 4 (Follow(1)) *The Follow(1) set of Terminal symbols of a Non-Terminal symbol is defined by:*

$$Follow(A) = \bigcup_{Y \to \alpha A \beta} (First(\beta) \bigcup \begin{cases} \emptyset & , \beta \not\Rightarrow^* \epsilon \\ Follow(Y) & , \beta \Rightarrow^* \epsilon \end{cases}$$

Figura 4: Definition 4

Para isto, o grupo gerou o código capaz de calcular o *First* e o *Follow* de cada produção, tornando assim mais fácil o cálculo do *Lookahead* de cada produção. Para além disto, geramos o código capaz de reconhecer se existe recursividade à esquerda na gramática e, se for o caso, corrigi-la.

- ***Recursividade* à esquerda**

Antes de começar com o cálculo dos *Firsts*, *Follows* e *Lookaheads* o grupo começa por verificar se a gramática em questão contém recursividade à esquerda, um dos conflitos das gramáticas LL(1). Para tal, basta verificar para todas as produções, se o seu lado esquerdo é semelhante ao primeiro símbolo que aparece do lado direito.

```
def verify_LeftRecursion():
    res = True
    for key,value in producoes.items():
        for p in value:
            if len(p) > 0:
                if key == p[0]:
                    res = False
    return res
```

Caso a gramática contenha recursividade à esquerda, esta é corrigida retirando a produção com a recursividade e adicionando duas novas. O código utilizado encontra-se disponível na Figura 8 presente nos Anexos. A técnica descrita é a seguinte:

```
# com recursividade a esquerda
A -> A alpha | beta
# sem recursividade a esquerda
A -> beta A'
A'-> alpha A' |
```

- ***First***

Para o cálculo do conjunto *First* o problema foi dividido em dois. Quando o primeiro símbolo que nos aparece do lado direito é terminal ou nulo.

```
# simbolos terminais e nulos
    if len(prod) != 0:
        if prod[0] in T:
            return [prod[0]]
    else:
        return []
```

E por sua vez quando o símbolo é não terminal. Para este caso procuramos todas as ocurrências desse símbolo do lado esquerdo e calculamos os seus *Firsts* que acabarão por ser unidos.

```
# simbolos nao terminais
    if prod[0] in NT:
        res = []
        occur = producoes[prod[0]]

        for o in occur:
            for i in first(o):
                res.append(i)

        if [] not in res:
            return res
        else:
            pass
```

- ***Follow***

O cálculo dos conjuntos *Follow* é mais extenso e por essa razão não é apresentado aqui mas sim nos anexos (Figura 9), no entanto explicaremos o algoritmo utilizado.

Numa primeira fase, é criado um *set* que será responsável por guardar todos os símbolos do conjunto já calculados. Por sua vez são percorridos todos os lados esquerdos das produções e para estes, todos os seus lados direitos, ou seja, dois ciclos aninhados. Nesta fase iremos verificar se o símbolo não terminal que foi passado como argumento se encontra no lado direito atual, se não se encontrar a iteração termina. Caso contrário, ou seja, encontramos o símbolo, vamos verificar se nos encontramos no fim da produção ou não. Se não for o fim da produção respetiva, iremos calcular o conjunto *First* do símbolo imediatamente à frente e caso verifiquemos que a produção é anulável, ou seja, contém o símbolo '#' no seu conjunto *First* teremos que unir com o *Follow* do símbolo não terminal do lado esquerdo respetivo. Caso não seja anulável seguimos em frente. Por outro lado, se for o fim da produção, o que é feito é calcular o conjunto *Follow* do símbolo do lado esquerdo, no entanto, garantindo que estes não são iguais para evitar ciclos infinitos. Por fim, adicionamos os conjuntos calculados e adicionamos ao *set* mencionado inicialmente e passamos à próxima iteração.

- ***Lookahead***

O cálculo do *Lookahead* é muito simples. São calculados os conjuntos *First* e *Follow* e de seguida é verificado se a produção fornecida é anulável ou não. Se não for anulável, o *Lookahead* será igual ao *First*, caso contrário, será a união dos dois conjuntos.

```
def lookAhead(le,ld):
    fi = []
    for p in ld:
        fi.append(first(p))
    fo = follow(le)

    if isNullable(le):
        fi.remove([])
        return fi + [fo]
    else:
        return fi
```

Para agora verificarmos que a gramática é de facto LL(1) seguimos a regra apresentada na Figura 1. Ou seja, para duas produções com o mesmo lado esquerdo, a interseção dos seus *Lookaheads* terá que ser vazia. Para isso, o que fazemos é unir o *Lookahead* das produções com o mesmo lado esquerdo, transformar num *set*, uma vez que este não permite valores repetidos e verificamos se o tamanho do conjunto se mantém. Se se mantiver, significa que não tinhamos elementos repetidos e a regra é cumprida, caso contrário concluimos que a gramática não é LL(1).

```
def verify_lookAheads():
    res = True
    for value in lookAheads.values():
        if len(value) > 1:
            aux = []
            for i in value:
                aux = aux + i
            if len(aux) != len(set(aux)):
                res = False
    return res
```

De seguida, passamos à implementação do parser recursivo descendente capaz de reconhecer esta gramática. Inicialmente, identificamos a lista de tokens que iriam ser úteis à implementação do parser, assim como as suas expressões regulares:

```
tokens = ['PA', 'PF', 'VIRG', 'ID', 'GK', 'LAT', 'CEN', 'MED',
          'EXT', 'PL', 'NOME', 'POSICOES']
t_PA    = r'\['
t_PF    = r'\]'
t_VIRG  = r','
t_ID    = r'[A-Za-z]+'

def t_NOME(t):
    r'Nome\:'
    return t

def t_POSICOES(t):
    r'Posicoes\:'
    return t

def t_GK(t):
    r'GK'
    return t

def t_LAT(t):
    r'LAT'
    return t

def t_CEN(t):
    r'CEN'
    return t

def t_MED(t):
    r'MED'
    return t

def t_EXT(t):
    r'EXT'
    return t

def t_PL(t):
    r'PL'
    return t
```

Numa primeira fase, o grupo encontrou vários conflitos e colisões no reconhecimento de palavras. Logo, foi necessário que vários tokens fossem definidos em funções, de forma a que estes tivessem prioridade e não existisse qualquer conflito ou colisão indesejado.

Seguidamente, o foco foi a criação do parser em si. Tal como é referido no enunciado, a tarefa de escrever um parser recursivo descendente é bastante repetitiva e segue um padrão algorítmico. Assim, para além da definição do **parserError**, **rec_Parser** e **rec_term** destacaremos algumas definições mais importantes da nossa implementação:

A definição do **Cont1** verifica se o próximo símbolo é **'VIRG'**. Se assim for, reconhece esse símbolo, chama a função que reconhece o **Jogador** e volta a chamar **Cont1**. Se o próximo símbolo for **'PF'**, nada é feito. Se nenhum destes casos acontecer, existirá algum erro com o parser.

```
def rec_Cont1():
    global prox_simb
    if prox_simb.type == 'VIRG':
        rec_term('VIRG')
        rec_Jogador()
        rec_Cont1()
    elif prox_simb.type == 'PF':
        pass
    else:
        parserError(prox_simb)
```

O destaque para o **rec_Jogador** deve-se à funcionalidade adicionada pelo grupo. Nesta definição, existe uma variável global **posicoesCadaJogador** que é capaz de contar em quantas posições diferentes um jogador consegue jogar.

```
def rec_Jogador():
    global posicoesCadaJogador
    rec_term('NOME')
    rec_Nome()
    rec_term('POSICOES')
    rec_term('PA')
    rec_Posicoes()
    rec_term('PF')
    print("Este jogador e capaz de jogar em",
          posicoesCadaJogador,"posicoes diferentes")
    posicoesCadaJogador = 0
```

No **rec_Nome**, novamente, o destaque é a funcionalidade de contar o número de jogadores que compõem um plantel adicionada pelo grupo.

```
def rec_Nome():
    global nrJogadores
    rec_term('ID')
    nrJogadores += 1
```

O **rec_Posicao** foi a definição que originou um maior trabalho. É nesta definição que são reconhecidas todas as posições possíveis. Para além disto, com as várias variáveis globais, somos capazes de contar quantos jogadores são capazes de atuar em cada uma das posições. Temos ainda a capacidade de contar em quantas posições diferentes um jogador consegue jogar, valor esse que é utilizado no **rec_Jogador**, tal como foi referido anteriormente. Por último, criamos uma lista de posições(iniciada a vazio) onde colocamos as posições que ainda não estão lá representadas, podendo assim ser calculado (**posicoesPreenchidas**) para quantas posições diferentes existem jogadores disponíveis no plantel.

```
posicoes = []
def rec_Posicao():
    global prox_simb
    global posicoesCadaJogador
    global gks
    global lats
    global cens
    global meds
    global exts
    global pls
    global posicoesPreenchidas
    if prox_simb.type == 'GK':
        rec_term('GK')
        if "GK" not in posicoes:
            posicoes.append("GK")
            posicoesPreenchidas += 1
        gks += 1
        posicoesCadaJogador += 1
    elif prox_simb.type == 'LAT':
        rec_term('LAT')
        if "LAT" not in posicoes:
            posicoes.append("LAT")
            posicoesPreenchidas += 1
        lats += 1
        posicoesCadaJogador += 1
    elif prox_simb.type == 'CEN':
        rec_term('CEN')
        if "CEN" not in posicoes:
            posicoes.append("CEN")
            posicoesPreenchidas += 1
        cens += 1
        posicoesCadaJogador += 1
    elif prox_simb.type == 'MED':
        rec_term('MED')
        if "MED" not in posicoes:
            posicoes.append("MED")
```

```
            posicoesPreenchidas += 1
        meds += 1
        posicoesCadaJogador += 1
    elif prox_simb.type == 'EXT':
        rec_term('EXT')
        if "EXT" not in posicoes:
            posicoes.append("EXT")
            posicoesPreenchidas += 1
        exts += 1
        posicoesCadaJogador += 1
    elif prox_simb.type == 'PL':
        rec_term('PL')
        if "PL" not in posicoes:
            posicoes.append("PL")
            posicoesPreenchidas += 1
        pls += 1
        posicoesCadaJogador += 1
    else: parserError(prox_simb)
```

3 Exemplos de utilização

Nesta secção do relatório apresentamos os resultados da aplicação do nosso trabalho a diferentes Plantéis criados pelo grupo.

Antes da aplicação do parser aos vários Plantéis, começamos por mostrar a aplicação do código que verifica se a gramática é LL(1) ou não. Não o sendo, o código tem a capacidade de a corrigir.

```
# COM RECURSIVIDADE A ESQUERDA
producoes = {
#Plantel    : '[' Jogadores ']'
#Jogadores  : Jogadores ',' Jogador | Jogador
#Jogador    : "Nome:" Nome "Posicoes:" '[' Posicoes ']'
#Nome       : id
#Posicoes   : Posicoes ',' Posicao | Posicao
#Posicao    : GK | LAT | CEN | MED | EXT | EXT | PL
"Plantel"   : [['PA', 'Jogadores', 'PF']],
"Jogadores": [['Jogadores', 'VIRG', 'Jogador'],
               ['Jogador']],
"Jogador"   : [['Nome:', 'Nome',
                'Posicoes:', 'PA', 'Posicoes', 'PF']],
"Nome"      : [['id']],
"Posicoes"  : [['Posicoes', 'VIRG', 'Posicao'],
               ['Posicao']],
"Posicao"   : [['GK'],['LAT'],
               ['CEN'],['MED'],
               ['EXT'],['PL']]
}

# Simbolos nao terminais
NT = ['Plantel', 'Jogadores', 'Jogador',
      'Nome', 'Posicoes', 'Posicao']

# Simbolos terminais
T = ['PA', 'PF', 'VIRG', 'Nome:', 'Posicoes:', 'id',
     'GK', 'LAT','CEN', 'MED', 'EXT', 'PL']
```

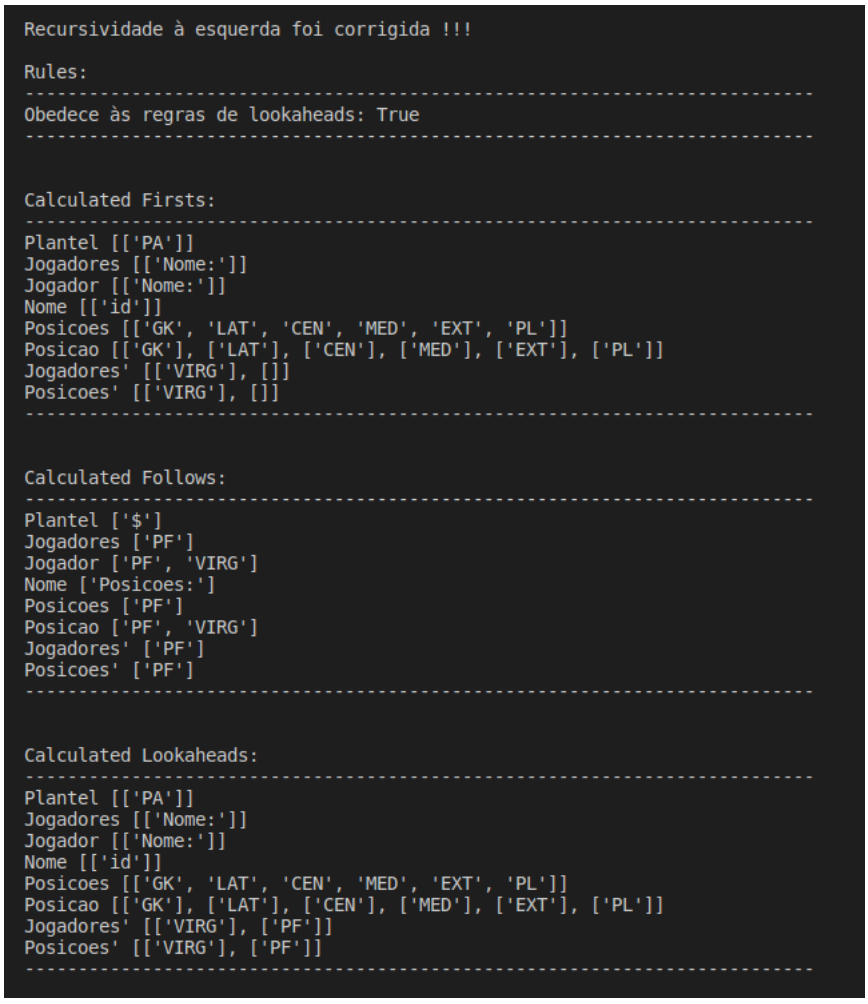

Figura 5: Verificação da Recursividade

Inicialmente, mostramos a aplicação do nosso parser a um Plantel mais pequeno, de apenas 3 jogadores:

```
[Nome: Ricardo Posicoes: [MED,EXT],
 Nome: Miguel Posicoes: [EXT,PL],
 Nome: Filipe Posicoes: [PL]]
```

```
term: [
term: Nome:
term: Ricardo
term: Posicoes:
term: [
term: MED
term: ,
term: EXT
term: ]
Este jogador é capaz de jogar em 2 posicoes diferentes
term: ,
term: Nome:
term: Miguel
term: Posicoes:
term: [
term: EXT
term: ,
term: PL
term: ]
Este jogador é capaz de jogar em 2 posicoes diferentes
term: ,
term: Nome:
term: Filipe
term: Posicoes:
term: [
term: PL
term: ]
Este jogador é capaz de jogar em 1 posicoes diferentes
term: ]
That's the end...


|-----------------------------------------PLANTEL INFO-----------------------------------------|

Este plantel é constituído por 3 jogadores
Este plantel tem jogadores que podem atuar em 3 posicoes diferentes
Existem 0 guarda-redes neste plantel
Existem 0 laterais neste plantel
Existem 0 centrais neste plantel
Existem 1 medios neste plantel
Existem 2 extremos neste plantel
Existem 2 pontas-de-lança neste plantel
```

Figura 6: Plantel Simples

De seguida, mostramos o resultado da aplicação do parser a um plantel de 24 jogadores.

```
[Nome: Ricardo Posicoes: [MED,EXT],
 Nome: Miguel Posicoes: [EXT,PL],
 Nome: Filipe Posicoes: [PL],
 Nome: Duarte Posicoes: [PL,EXT],
 Nome: Lucas Posicoes: [MED,CEN,PL],
 Nome: Manuel Posicoes: [GK],
 Nome: Agostinho Posicoes: [EXT,PL],
 Nome: Alves Posicoes: [CEN],
 Nome: Costa Posicoes: [MED,EXT,CEN],
 Nome: Marques Posicoes: [GK],
 Nome: Paulinho Posicoes: [MED,LAT],
 Nome: Pires Posicoes: [LAT,EXT],
 Nome: Abreu Posicoes: [MED,EXT],
 Nome: Zeus Posicoes: [EXT,PL],
 Nome: Armando Posicoes: [PL],
 Nome: Pedro Posicoes: [PL,EXT],
 Nome: Diogo Posicoes: [MED,CEN,PL],
 Nome: Serginho Posicoes: [GK],
 Nome: Jesus Posicoes: [EXT,PL],
 Nome: Moreira Posicoes: [CEN],
 Nome: Carmo Posicoes: [MED,EXT,CEN],
 Nome: Lipo Posicoes: [GK],
 Nome: Zeca Posicoes: [MED,LAT],
 Nome: Bruno Posicoes: [LAT,EXT]]
```

```
term: ]
That's the end...


|-----------------------------------------PLANTEL INFO-----------------------------------------|

Este plantel é constituído por 24 jogadores
Este plantel tem jogadores que podem atuar em 6 posicoes diferentes
Existem 4 guarda-redes neste plantel
Existem 4 laterais neste plantel
Existem 6 centrais neste plantel
Existem 8 medios neste plantel
Existem 12 extremos neste plantel
Existem 10 pontas-de-lança neste plantel
```

Figura 7: Plantel de 24 Jogadores

4 Conclusões

Após a realização do trabalho prático número 2, consideramos que foi bastante benéfico para o aprofundamento da matéria lecionada na unidade curricular de Processamento de Linguagens, com destaque para a utilização do módulo **lex** do Python, que foi essencial para a conceção do parser recursivo descendente.

Acreditamos que conseguimos atingir os objetivos principais do trabalho. Conseguimos criar uma gramática que cumprisse todos os requisitos que fizessem dela uma gramática LL(1) e, de seguida, geramos o parser recursivo descendente capaz de a reconhecer. Para isto, foi fundamental a boa conceção da gramática, pois tornou a tarefa de gerar o parser bastante mais simples.

O grupo apresentou algumas dificuldades em gerar o código que verificasse se a gramática era LL(1) ou não, mas no final conseguiu ultrapassar este problema e cumpriu também com este obejtivos do trabalho. Adicionalemnte, fomos capazes de implementar algumas funcionalidades no parser, tais como a indicação de quantos jogadores compõem o plantel, para quantas posições diferentes existem jogadores no plantel, quantos jogadores existem para cada posição e, por fim, quantas posições cada jogador pode fazer.

5 Anexos

```python
# Resolve o conflito de recursividade à esquerda
def solve_LeftRecursion():
    store = {}

    for le in producoes:
        alphaRules = [] # regras com recursividade
        betaRules = [] # regras sem recursividade

        # todos os lados direitos de um lado esquerdo com o mesmo nome
        alllld = producoes[le]
        for ld in alllld:
            if ld[0] == le: # existe recursividade
                alphaRules.append(ld[1:])
            else:
                betaRules.append(ld)

        # criação das novas regras (por em evidência)
        if len(alphaRules) != 0:

            # geração de um novo simbolo nao terminal
            le_aux = le + "'"
            while (le_aux in producoes.keys()) or (le_aux in store.keys()):
                le_aux += "'"
            NT.append(le_aux)

            # beta rule, sem recursividade
            for b in range(0, len(betaRules)):
                betaRules[b].append(le_aux)
            producoes[le] = betaRules

            # alpha rule, com recursividade
            for a in range(0, len(alphaRules)):
                alphaRules[a].append(le_aux)
            alphaRules.append([])
            store[le_aux] = alphaRules

    # substitui novas regras
    for left in store:
        producoes[left] = store[left]
```

Figura 8: *Solve_LeftRecursion*

```python
# Cálculo do Follow de um simb não terminal
def follow(nt):

    # follows calculados ate ao momento
    current = set()
    if nt == 'Plantel':
        current.add('$')

    # lado esquerdo das producoes
    for le in producoes:
        # todos os lados direitos do respetivo lado esquerdo, lista de listas
        lds = producoes[le]
        # todos os lados direitos do respetivo lado esquerdo, por lista
        for ld in lds:
            if nt in ld:
                while nt in ld:
                    index_nt = ld.index(nt) # indice do simb na producao
                    ld = ld[index_nt + 1:] # o que está para a frente do simb

                    # verificar se estamos no fim da producao ou nao
                    if len(ld) != 0:
                        # verificamos o que é que está à frente e calculamos o first
                        res = []
                        if ld[0] in NT:
                            prod = producoes[ld[0]]
                            for p in prod:
                                if len(first(p)) != 0:
                                    res.append(first(p))
                                else:
                                    res.append(['#'])   # para representar null
                        else:
                            res.append(first(ld))

                        aux = []
                        for i in res:
                            aux = aux + i
                        res = aux

                        # se a producao for anulavel vamos ter que unir com o follow do lado esquerdo
                        if '#' in res:
                            newList = []
                            res.remove('#')
                            ansNew = follow(le)

                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = res + ansNew
                                else:
                                    newList = res + [ansNew]
                            else:
                                newList = res
                            res = newList
                    else:
                        # se tivermos no fim da producao
                        # calculamos o follow do lado esquerdo
                        # garantindo que não são iguais
                        if nt != le:
                            res = follow(le)

                    # atualizamos os follows ja calculados
                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                current.add(g)
                        else:
                            current.add(res)
    return list(current)
```

Figura 9: *Follow*