Aula Teórica 6 (guião)

Semana de 20 a 24 de Outubro de 2025

José Carlos Ramalho

Sinopsis:

- Parsers Top-Down: o Recursivo Descendente;
- A condição LL(1).

- Imagens e algum conteúdo retirados da sebenta "Processamento de Linguagens: Reconhecedores Sintáticos" de José João Almeida e José Bernardo Barros, editada em Abril de 2022.

Gramáticas

- Gramáticas Regulares;
- Gramáticas Independentes de Contexto;
- Gramáticas Dependentes de Contexto;
- Gramáticas Livres.

Gramáticas: especificação

Uma gramática G é um tuplo (N, T, S, P), onde:

- N - conjunto de símbolos não terminais;
- T - conjunto de símbolos terminais;
- S - símbolo inicial;
- P - as produções.

Exemplificar: uma turma de alunos

```
Turma PL2025 .
Alunos{
    ("A23876", "Ana Maria")
    ("A78654", "Paulo Azevedo")
    ...
}
```

Qual seria a gramática?

```
T = {}
S =
```

```
N =
P =
```

(Dar exemplos)

LISP + MarkDown

```
(doc
    (tit "Primeiro exemplo")
    (subtit "Aula 6: 2025-03-14")
    "Este é um primeiro exemplo."
)
```

Expressões S

```
(+ 5 4)
(+ (- 7 2) (+ 12 19))
```

Revisitando os parentesis

```
()
(()())()
```

Expressões aritméticas básicas

```
4 + 15
7 - 12 * 3
7 + 8 * 2 / 3
```

Voltando às listas

Exercício: a linguagem das listas

Exemplos:

```
[]
[2]
```

```
[ 2, 4, 5]
[ 2, 4, [ 5, 7, 9], 6]
```

Símbolos terminais: `T = {'[', ']', num}`

Produções:

```
Lista --> '[' ']'
        | '[' Conteudo ']'

Conteudo --> num
           | num ',' Conteudo
```

Analisador Léxico

```
# listas_analex.py
# 2023-03-21 by jcr
# ----------------------
import ply.lex as lex

tokens = ('NUM', 'PA', 'PF', 'VIRG')

t_NUM = r'[+\-]?\d+'
t_PA = r'\['
t_PF = r'\]'
t_VIRG = r','

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore = '\t '

def t_error(t):
    print('Carácter desconhecido: ', t.value[0], 'Linha: ',
t.lexer.lineno)
    t.lexer.skip(1)

lexer = lex.lex()
```

Programa exemplo

```
# listas_program.py
# 2023-03-21 by jcr
# ----------------------
```

```
linha = input("Introduza uma lista: ")
rec_Parser(linha)
```

Reconhcedores (Parsers) Top-Down: o recursivo descendente

- Vamos escrever uma função para cada símbolo terminal (T) ou não terminal (N).
- Para os terminais basta verificar se o símboloque é preciso reconhecer é o que está a seguir na string de entrada:

$$\textbf{função } recT\ (\ t : T) : tipoValT$$
$$\begin{cases} \textbf{se} \quad t = simboloseg \longrightarrow \begin{cases} val \leftarrow simboloseg.valor \\ ABORT \end{cases} \\ \quad\ \textbf{senão} \longrightarrow \\ \text{return } val \end{cases}$$

Para os não terminais a situação é um pouco mais complexa.

Para produções da forma: `A -> x1 x2 ...xn | ... | y1 y2 ...yn` é preciso decidir qual das regras se vai seguir.

No exemplo das listas, quais as produções com que devemos ter cuidado?

```
p1: Lista --> '[' ']'
p2:         | '[' Conteudo ']'

p3: Conteudo --> num
p4:            | num ',' Conteudo
```

No decurso da análise do parser, este terá de decidir entre `p1` e `p2` quando estiver a tentar derivar o símbolo `Lista` e entre `p3` e `p4` quando estiver a tentar derivar o símbolo `Conteúdo`.

Para se poder escolher a produção a seguir é preciso calcular o conjunto de símbolos terminais que poderão dar início à derivação especificada nessa produção. A estes conjuntos chamamos `lookaheads`.

$$
\begin{array}{l}
\textbf{função } recA\ (\ ) : tipoValA \\
\left\{\begin{array}{l}
\textbf{se } simboloseg \in lookahead(A \to x1x2...xn) \longrightarrow \\
\quad \left\{\begin{array}{l} val1 \leftarrow recx1; \\ val2 \leftarrow recx2; \\ ... \\ valn \leftarrow recxn; \\ valA \leftarrow f(val1, val2, ...valn) \end{array}\right. \\
simboloseg \in lookahead(A \to ...) \longrightarrow \\
\quad \left\{\begin{array}{l} ... \end{array}\right. \\
simboloseg \in lookahead(A \to y1y2...ym) \longrightarrow \\
\quad \left\{\begin{array}{l} val1 \leftarrow recy1; \\ val2 \leftarrow recy2; \\ ... \\ valm \leftarrow recym; \\ valA \leftarrow g(val1, val2, ...valm) \end{array}\right. \\
\textbf{senão} \longrightarrow \\
\quad \left\{\begin{array}{l} [erro] \end{array}\right. \\
\text{return } valA
\end{array}\right.
\end{array}
$$

Cálculo de Lookaheads

O Cálculo do `lookahead` faz-se com a ajuda de duas funções auxiliares: `First` e `Follow`.

First

Esta função calcula o conjunto de símbolos terminais que são inícios válidos de uma determinada sequência de símbolos. Esta sequência pode iniciar-se por um símbolo terminal ou por um não terminal.

Para um sequência iniciada por um não terminal calcula-se o `FirstN` correspondente a esse símbolo:

$$\textbf{função } first\_nt\ (\ G : gramatica, A : NT) : T - set$$
$$\left\{ \begin{array}{l} \textbf{para } \ (A \to rhs) \in Producoes(G) \textbf{ fazer} \\ \quad \left\{ \ fir \leftarrow fir \cup first\_str(rhs) \right. \\ \\ \text{return } fir \end{array} \right.$$

Para os restantes casos aplica-se a função seguinte:

$$\textbf{função } first\_str\ (\ G : gramatica : s : (T|NT) - seq) : T - set$$
$$\left\{ \begin{array}{ll} \textbf{se} & s = <> \longrightarrow & \left\{ fir \leftarrow \{\} \right. \\ & s = <h.t> \wedge h \in T \longrightarrow & \left\{ fir \leftarrow h \right. \\ & s = <h.t> \wedge h \in NTanulaveis \longrightarrow & \left\{ fir \leftarrow first\_nt(h) \cup first\_str(t) \right. \\ & s = <h.t> \wedge h \notin NTanulaveis \longrightarrow & \left\{ fir \leftarrow first\_nt(h) \right. \\ \text{return } fir \end{array} \right.$$

Follow

Follow ou terminal seguinte de um símbolo não terminal `A`, corresponde a determinar o conjunto de símbolos terminais que se podem seguir àquilo em que A derive. O cálculo do `Follow(A)` vai depender do contexto em que `A` possa estar inserido. Esses contextos vão ser lidos nas produções em que `A` aparece no lado direito.

Nomeadamente se existe uma produção: `B -> ... A cont`

então o `Follow(A)` vai incluir o início de `cont` ou seja `first_str(cont)`; Quando `cont` for anulável, então aquilo que se puder seguir a `B` também pertencerá a `Follow(A)`.

$$\textbf{função } follow\ (\ G : gramatica, A : NT) : T - set$$
$$\left\{ \begin{array}{l} \textbf{para } \ (B \to \ ...\ A\ cont) \in Producoes(G) \textbf{ fazer} \\ \quad \left\{ \begin{array}{l} \textbf{se } [\text{cont é anulável}] \longrightarrow \\ \quad \left\{ \ fol \leftarrow fol \cup first\_str(G, cont) \cup follow(G, B) \right. \\ \textbf{senão} \longrightarrow \\ \quad \left\{ \ fol \leftarrow fol \cup first\_str(G, cont) \right. \end{array} \right. \\ \\ \text{return } fol \end{array} \right.$$

Condição LL(1)

Para quaisquer duas derivações com o mesmo lado esquerdo, a **interseção dos seus lookaheads tem que ser vazia**.

Exemplo: cálculo dos lookaheads para o caso das listas

Gramática:

```
p1: Lista --> '[' ']'
p2:         | '[' Conteudo ']'

p3: Conteudo --> num
p4:            | num ',' Conteudo
```

Lookaheads:

```
la(p1) = First('[' ']') = {'['}
la(p2) = First('[' Conteudo ']') = {'['}

la(p1) ∩ la(p2) = {'['} ==> Conflito LL(1)

la(p3) = First(num) = {num}
la(p4) = First(num ',' Conteudo) = {num}

la(p3) ∩ la(p4) = {num} ==> Conflito LL(1)
```

Quando existem conflitos não é possível reconhecer a linguagem com um parser TopDown que esteja a ver apenas um símbolo à frente. A solução passa por transformar a gramática.

Exercício (da sebenta):

Considere a seguinte gramática para um subconjunto do LISP:

```
p1: S   -> Exp '.'
p2: Exp -> int
p3:      | '(' Funcao ')'
p4: Funcao -> '+' Lista
p5:         | '*' Lista
p6: Lista  -> Exp Lista
p7:         | ε
```

Identifique os conjuntos de produções potencialmente problemáticos e verifique a condição LL(1) para esses casos.

Analisador Sintático: recursivo descendente

```
# listas_anasin.py
# 2023-03-21 by jcr
# ----------------------
import listas_analex

prox_simb = ('Erro', '', 0, 0)

# Lista --> '[' ']'
#         | '[' Conteudo ']'
def rec_Lista():
    global prox_simb
    ... Temos um problema!!!

def rec_Parser(data):
    global prox_simb
    lexer.input(data)
    prox_simb = lexer.token()
    rec_Lista()
    print("That's all folks!")
```

Com a alteração das produções de Lista

```
def parserError(simb):
    print("Erro sintático, token inesperado: ", simb)

def rec_term(simb):
    global prox_simb
    if prox_simb.type == simb:
        prox_simb = lexer.token()
    else:
        parserError(prox_simb)

# P4: Conteudo --> num
# P5:            | num ',' Conteudo
# É preciso alterar para:
# P4: Conteudo --> num Cont2
# P5: Cont2    -->
# P6: Cont2    --> ',' Conteudo

def rec_Cont2():
    global prox_simb
    if prox_simb.type == 'VIRG':
        rec_term('VIRG')
        rec_Conteudo()
        print("Reconheci P6: Cont2    --> ',' Conteudo")
    elif prox_simb.type == '???':
        print("Reconheci P5: Cont2 -->")
    else:
        parserError(prox_simb)
```

```
def rec_Conteudo():
    rec_term('NUM')
    rec_Cont2()
    print("Reconheci P4: Conteudo --> num Cont2")

def rec_LCont():
    global prox_simb
    if prox_simb.type == 'PF':
        rec_term('PF')
        print("Reconheci P2: LCont --> ']'")
    elif prox_simb.type == 'NUM':
        rec_Conteudo()
        rec_term('PF')
        print("Reconheci P3: LCont --> Conteudo ']'")
    else:
        parserError(prox_simb)

# P1: Lista --> '[' LCont
# P2: LCont --> ']'
# P3:         | Conteudo ']'
def rec_Lista():
    global prox_simb
    rec_term('PA')
    rec_LCont()
    print("Reconheci P1: Lista --> '[' LCont")
```

Gramática Concreta Final

```
P1: Lista --> '[' LCont
P2: LCont --> ']'
P3:         | Conteudo ']'
P4: Conteudo --> num Cont2
P5: Cont2    -->
P6: Cont2    --> ',' Conteudo
```

Gramática Abstrata

```
P1: Lista -->
P2:         | Conteudo
P3: Conteudo --> num Conteudo
P4:            |
```

Modelo em Python

```
class Lista
```