3 – Análise Sintática

Linguagens e Programação

Ana Madureira
Engenharia Informática
Ano Lectivo: 2021/2022

Fontes:

1. Compiladores Princípios e práticas, Keneth C.Louden, Thomson, 2004.
   Cap. 3 Gramáticas Livres de Contexto e Análise Sintática

2. Compiladores - Princípios , Técnicas e Ferramentas, Alfred V. Aho, Monica S. Lam e Ravi Sethi, Pearson, 2ª edição, 2007.
   Cap. 4 Análise Sintáctica

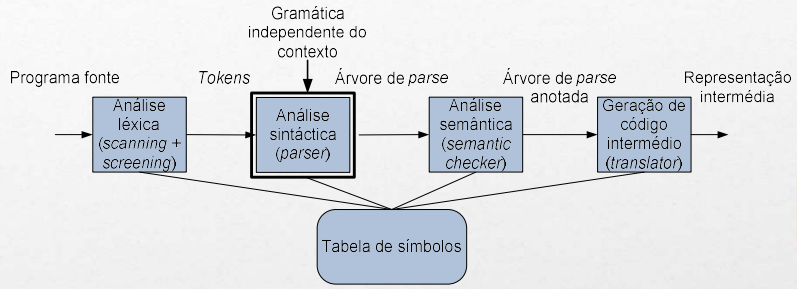
1

Análise Sintática

- Verificar se uma dada sequência de *tokens* constitui um programa válido
- Caso a sequência esteja correcta, extrair a sua estrutura (construir a árvore de *parse*) de acordo com as regras gramaticais que especificam a linguagem
- Compete ao analisador sintático encontrar erros na sequência de *tokens* e reportá-los o mais cedo possível, sem cessar a análise do resto da sequência
- Outras tarefas que podem ser realizadas conjuntamente com a análise sintática:
  - extrair, calcular e armazenar informação (atributos) relativamente aos símbolos gramaticais (terminais e não-terminais)
  - fazer verificações relativamente a esses atributos (p. ex. consistência de tipos)
  - ir gerando as "instruções" do código intermédio
    - As primeiras duas tarefas enunciadas neste ponto constituem a análise semântica

2

Análise sintáctica



Analisador sintáctico ou *parser*:

- **entrada**: sequência de *tokens* provenientes do analisador léxico
- **saída**: árvore de *parse* da sequência, se estiver de acordo com a gramática (muitas vezes esta árvore não é construída explicitamente e as fases seguintes executam-se em simultâneo com a análise sintáctica)

3

Construção de um analisador sintáctico

- É necessário:
  - Um formalismo para especificar a linguagem de programação
    - **gramática independente do contexto**
  - Um método eficiente para determinar se uma dada sequência de tokens está ou não contida na linguagem.
    - **Tentar uma derivação da sequência de *tokens* por aplicação das regras gramaticais (produções) a partir do símbolo inicial da gramática**
- Abordagens:
  - **Análise *top-down***
    - Constrói-se a árvore de *parse* partindo da raiz (símbolo inicial) até se atingirem todos os terminais (*tokens*)
    - É o método preferido caso se implemente o analisador sintáctico "à mão"
  - **Análise *bottom-up***
    - A construção da árvore de *parse* inicia-se nas folhas (*tokens*) e prossegue até se atingir o símbolo inicial, que constitui a raiz da árvore
    - Constitui um método bastante poderoso, mas difícil de implementar, pelo que se recorre geralmente a ferramentas automáticas para gerar este tipo de *parsers*

4

Definição de gramática

G = (V, Σ, P, S)

- Uma gramática é composta por 4 componentes:
  - Um conjunto Σ de **símbolos terminais** (as "palavras" ou tokens)
  - Um conjunto **V** de **símbolos não-terminais** (classes sintáticas ou gramaticais)
    - $\Sigma \cap V = \emptyset$
  - Um **conjunto P de produções** (também chamadas **regras gramaticais** ou regras de reescrita) da forma $\alpha \rightarrow \beta$, onde α e β são sequências de elementos de Σ e V (ou seja pertencem a $(\Sigma \cap V)^*$)
  - Um elemento especial de V, o símbolo $S \in V$, que se designa por **símbolo inicial** da gramática ou **símbolo de partida**
- Uma gramática G gera sequências de elementos de Σ (as "frases"); o conjunto de todas as sequências de elementos de Σ que é possível gerar, de acordo com a gramática G, chama-se a linguagem definida por G e denota-se como L(G)
- Outras definições:
  - Costuma chamar-se a $\Sigma \cup V$ o **vocabulário da gramática**
  - Genericamente a um $\omega \in (\Sigma \cup V)^*$ chama-se uma **forma sentencial**
  - Genericamente a um $w \in \Sigma^*$ uma **frase** (sentence)
  - A sequência vazia designa-se por **ε**

5

Símbolos Não terminais e Produções

- O **conjunto V dos símbolos não-terminais** de uma gramática é sempre um conjunto finito, contendo um **símbolo especial**, S (símbolo inicial), que representa todas as frases geradas pela gramática
- Cada **não-terminal** pode ser entendido como representando uma classe de frases, também conhecida por categoria sintáctica
- As produções de uma gramática formam um conjunto de regras de reescrita ou de substituição:
  - A regra **α → β** pode ser lida como "**a sequência α pode ser substituída (reescrita) pela sequência β**"
- Existem várias classes de gramáticas, consoante as restrições que se impõem à forma das sequências α e β que tomam parte do conjunto de produções
- Convenções para a notação:
  - Variáveis são iniciadas com letra maiúscula;
  - Símbolos do alfabeto com letra minúscula
  - $\alpha \rightarrow \beta$ para representar produções
  - $\alpha \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$ (produções alternativas)
  - $\alpha \rightarrow \varepsilon$

6

Classes de gramáticas - a hierarquia de Chomsky

Noam Chomsky definiu 4 classes de gramáticas:

- Nível 0 - **α e β sem restrições** - podem gerar linguagens muito complexas, difíceis de reconhecer
- Nível 1 - **Gramáticas dependentes do contexto** - onde $\alpha = \sigma A \beta$ e $\beta = \sigma \omega$ β, sendo $A \in V$ e $\omega \neq \varepsilon$ - geram linguagens menos complexas que as anteriores mas ainda difíceis de reconhecer
- Nível 2 - **Gramáticas independentes de contexto** - aqui todas as regras têm a forma $A \rightarrow \omega$ onde $A \in V$ - geram linguagens mais restritas que os níveis anteriores, mas suficientemente poderosas para serem usadas na definição das linguagens de programação; existem algoritmos bem conhecidos e eficientes para as reconhecer
- Nível 3 - **Gramáticas regulares** - todas as regras têm a forma $A \rightarrow wB$ ou $A \rightarrow w$, onde A e B $\in V$ e $w \in \Sigma^*$ - são equivalentes às expressões regulares utilizadas na especificação dos tokens (é sempre possível construir uma gramática regular a partir de uma expressão regular, gerando exatamente a mesma linguagem, e vice-versa)

7

Linguagens de programação: Conceitos

- Um programa numa dada linguagem pode ser visto como uma "frase" (*sentence*), ou sequências de "**frases**"
- Cada frase é uma sequência de componentes mais pequenos que são as "**palavras**" ou ***tokens*** da linguagem
- Cada "palavra", por sua vez, é uma sequência de "**caracteres**" individuais pertencentes a um conjunto - o **alfabeto**
- O conjunto de "**palavras**" ou ***tokens*** de uma linguagem de programação constituem o seu léxico:
  - Como já vimos, o léxico é definido, a partir do alfabeto, por expressões regulares
  - Compete ao analisador léxico (*scanner*), num compilador, construir os *tokens* a partir dos caracteres individuais do alfabeto.
- As regras para formar as "**frases**" de uma linguagem de programação constituem a sua **sintaxe**
- A sintaxe de uma linguagem de programação é definida por uma **gramática** (**independente do contexto ou de contexto livre**)

8

Derivações

- Uma **derivação** é uma sequência de reescritas, partindo do símbolo inicial S e terminando numa frase da linguagem (contendo apenas terminais)
- Mais formalmente define-se:
  - **Passo de derivação**: reescrita de uma forma sentencial da forma $\alpha A \gamma \Rightarrow \alpha\beta\gamma$, onde $A \rightarrow \beta$ é uma produção pertencente à gramática
  - **Derivação**: É uma sucessão de passos de derivação começando no símbolo inicial S e terminando numa frase $w \in \sum^*$ (escreve-se $S \Rightarrow^* w$)
- A expressão **$\alpha \Rightarrow \beta$** pode ler-se "**α deriva β**" ou "**α pode reescrever-se como β**"
- A expressão **$\alpha \Rightarrow^* \beta$** pode ler-se como "**α deriva em zero ou mais passos β**"
- A expressão **$\alpha \Rightarrow^+ \beta$** pode ler-se como "**α deriva num ou mais passos β**"
- As formas intermédias de uma derivação são formas sentenciais

9

Exemplo

Seja a gramática definida como:

- $\Sigma = \{ a, b, c\}$
- $V = \{ A, B\}$
  $P = \{ \quad A \rightarrow aA ,$
  $A \rightarrow bB ,$
  $A \rightarrow cB ,$
  $B \rightarrow a ,$
  $B \rightarrow b ,$
  $B \rightarrow c \quad \}$
- $S = A$

As frases da linguagem gerada por esta gramática constroem-se seguindo as regras:

1. Começar com o símbolo inicial S
2. Aplicar regras gramaticais de modo a reescrever as formas sentenciais que se vão gerando
3. Continuar a aplicação das regras até se atingir uma forma sentencial que apenas contenha terminais.

$A \Rightarrow bB \Rightarrow bc$

$A \Rightarrow aA \Rightarrow aaA \Rightarrow aacB \Rightarrow aacb$

$A \Rightarrow aA \Rightarrow aaA \Rightarrow aaaA \Rightarrow aaabB \Rightarrow aaaba$

10

Derivações canónicas

- Quando se executa uma derivação é legítimo escolher para reescrita (aplicando uma produção) qualquer não-terminal da forma sentencial presente
- No entanto, podemos optar por um método sistemático:
  - Escolher sempre para reescrita o não-terminal **mais à esquerda**
  - Escolher sempre para reescrita o não-terminal **mais à direita**
- Qualquer dos métodos sistemáticos anteriores constitui uma **derivação canónica**:
  - A derivação mais à esquerda, ou
  - A derivação mais à direita.
- A linguagem definida por uma gramática - o conjunto de todas as frases deriváveis do símbolo inicial - é então:

$$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$$

11

Exemplo

Considere-se a seguinte gramática independente do contexto para expressões simples:

Exp → Exp Op Exp | (Exp) | num
Op → + | - | * | /

Esta é uma forma abreviada de especificar uma gramática (também chamada BNF - Backus-Naur Form).

Nesta forma os terminais são nomes escritos com minúsculas ou símbolos individuais; aqui teríamos:

T = { (,), +, -, *, /, num}

Os não terminais são identificados pelas partes esquerdas das regras gramaticais (produções) e geralmente escrevem-se começando por uma maiúscula; temos então:

N = { Exp, Op}

O símbolo inicial é o não-terminal definido na primeira regra; aqui

S = Exp

As regras que têm a mesma parte esquerda (o mesmo não-terminal numa gramática livre de contexto) podem ser apresentadas como uma série de alternativas usando o meta-símbolo |; neste exemplo temos 7 regras ou produções

Pretende-se provar, através de uma derivação, que a sequência **(34-3)*42** pertence à linguagem gerada pela gramática (ou seja é uma expressão sintacticamente correcta).

As *strings* 34, 3 e 42 são instâncias do *token* num.

Assim, aquela sequência, em termos de *tokens* da gramática, será:

(num - num) * num

Uma derivação (mais à direita):

Exp ⇒ Exp Op Exp
⇒ Exp Op num
⇒ Exp * num
⇒ (Exp) * num
⇒ (Exp Op Exp) * num
⇒ (Exp Op num) * num
⇒ (Exp - num) * num
⇒ (num - num) * num

12

Exemplos

Gramática que gera todas as *strings* que contêm um ***a*** rodeado de um nº arbitrário de pares de parêntesis, seja { (^{n} a)^n, $\mathbb{Z}_{n\geq 0}$ }

S →(S) | a

A seguinte gramática gera a linguagem vazia, ou seja **L(G) =ϕ**

S→(S)

Gramática que gera todas as sequências de parêntesis emparelhados:

S →(S) S | ε

Algumas sequências geradas:

(( )) ( )
(((( ))))
(( ) (( ))) ( )

Gramática para instruções (muito simplificada):

Statement→ If-stmt | other
If-stmt → if (Exp) Statement |
if (Exp) Statement else Statement
Exp → 0 | 1

que gera frases como:

other
if (1) other
if (0) if (1) other
if (0) if (1) other else other
if (1) other else if (0) other else other
...

Uma gramática que gera exactamente a mesma linguagem da anterior pode ser:

Statement → If-stmt | other
If-stmt → if (Exp) Statement Else-part
Else-part → else Statement | ε
Exp → 0 | 1

13

Árvores de *parse*

- São **representações gráficas** (em forma de árvore) para as **derivações**, em que os nós dessa árvore têm associados **símbolos terminais** ou **não-terminais** da gramática
- A **raiz** da árvore está sempre associada ao **símbolo inicial**
- Os **nós internos da árvores** têm sempre associados **símbolos não-terminais**
- As **folhas** representam sempre **terminais**
- Os **filhos** de um determinado nó interno representam a reescrita do não-terminal associado a esse nó, através de uma **regra gramatical**, num dos **passos da derivação**

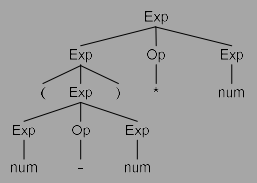
14

Exemplos

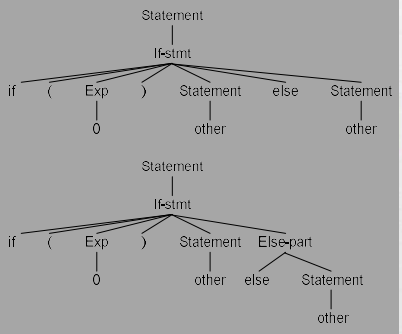
Construir a árvore de *parse* para a derivação da frase:
(num - num) * num
Usando a gramática da página 12, a derivação feita foi:

Exp ⇒ Exp Op Exp
⇒ Exp Op num
⇒ Exp * num
⇒ (Exp) * num
⇒ (Exp Op Exp) * num
⇒ (Exp Op num) * num
⇒(Exp - num) * num
⇒ (num - num) * num

à qual corresponde a árvore:



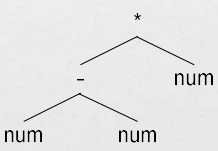
Árvores de *parse* para a derivação da frase " **if (0) other else other** ", pelas duas últimas gramáticas da página 14:

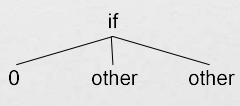


15

Árvores sintáticas abstractas

- Estas árvores são versões simplificadas das árvores de *parse* (concretas)
  - Geralmente todos os nós com não-terminais são eliminados
  - Refletem a dependência estrutural entre terminais
  - Os nós internos contêm "operadores" (em sentido lato)
  - As folhas contêm operandos ou informação
- Para os exemplos anteriores teríamos simplesmente:





16

Construção de um analisador sintáctico

- **Abordagens:**
  - **Análise *top-down***
    - Constrói-se a árvore de *parse* partindo da raiz (símbolo inicial) até se atingirem todos os terminais (*tokens*)
    - É o método preferido caso se implemente o analisador sintáctico "à mão"
  - **Análise *bottom-up***
    - A construção da árvore de *parse* inicia-se nas folhas (*tokens*) e prossegue até se atingir o símbolo inicial, que constitui a raiz da árvore
    - Constitui um método bastante poderoso, mas difícil de implementar, pelo que se recorre geralmente a ferramentas automáticas para gerar este tipo de *parsers*

17

Análise *Top-Down*

- Neste tipo de análise sintáctica, a árvore de parse é construída partindo da raiz, até se chegar à sequência de *tokens* do texto da entrada:
  - Gera-se sempre uma derivação mais à esquerda
  - Os nós da árvore de parse construída são visitados em pré-ordem (descida recursiva) ou próximo;
- São usados essencialmente dois métodos para implementar a análise *top-down*:
  - O método da descida recursiva, em que se associa uma rotina a cada não-terminal da gramática que é chamada quando se pretende reescrever esse símbolo;
  - O método da análise preditiva não-recursiva, que necessita de um stack auxiliar e é guiada por uma tabela de parse.
- Estes tipos de análise sintáctica requerem geralmente uma classe de gramáticas livres de contexto, denominadas LL(k) (na prática LL(1))
  - O método da descida recursiva, sendo geralmente escrito "à mão", pode incorporar técnicas ad-hoc e portanto relaxar um pouco as exigências das gramáticas LL(k).

18

Análise Top-Down - Exemplo

Considere-se a gramática:

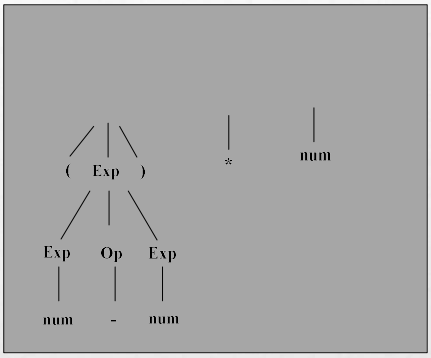
Exp → Exp Op Exp | (Exp) | num
Op → + | - | * | /

E a sequência de tokens:

(num - num) * num

Numa análise **top-down** partimos do símbolo inicial (Exp) e tentamos chegar à sequência de tokens através de uma derivação geralmente mais à esquerda:

Exp ⇒ Exp Op Exp
⇒ (Exp) Op Exp
⇒ (Exp Op Exp) Op Exp
⇒ (num Op Exp) Op Exp
⇒ (num - Exp) Op Exp
⇒ (num - num) Op Exp
⇒ (num - num) * Exp
⇒ (num - num) * num



19

Análise Bottom-Up

- Os parsers ***top-down*** têm de decidir qual a regra gramatical a aplicar tendo visto, na prática, apenas um **símbolo** do seu lado direito.
- Nos analisadores sintácticos ***bottom-up*** as regras gramaticais só são aplicadas depois de se ter visto e reconhecido toda a sua parte direita e possivelmente mais o(s) símbolo(s) seguinte(s).
- Os parsers ***bottom-up*** aplicam as regras gramaticais substituindo a sua parte direita pelo não-terminal que constitui a sua parte esquerda.
- **Constroem uma árvore de parse partindo das folhas** (terminais - ***tokens***) até se chegar ao símbolo inicial da gramática.
- A construção feita desta forma é sempre uma derivação mais à direita (***rightmost***) feita por ordem inversa.

  Considere-se a gramática **S → (S) S | ε** e a entrada **( )**.

  Uma derivação feita desta forma pode ser:

  ( ) ⇐ (S) ⇐ (S) S ⇐ S

20

Análise *BottomUp - Exemplo*

Considere-se a gramática:

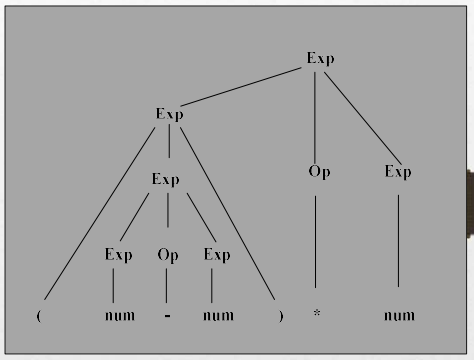
Exp → Exp Op Exp | (Exp) | num
Op → + | - | * | /

E a sequência de tokens:

(num - num) * num

Na análise ***bottom-up*** partimos da sequência de ***tokens*** que pretendemos analisar e vamos aplicando regras gramaticais (ao contrário), até se chegar ao símbolo inicial. Trata-se de uma **derivação mais à direita**:

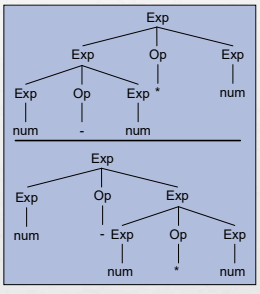
⇐ (num - num) * num
⇐ (Exp - num) * num
⇐ (Exp Op num) * num
⇐ (Exp Op Exp) * num
⇐ (Exp) * num
⇐ Exp * num
⇐ Exp Op num
⇐ Exp Op Exp
Exp

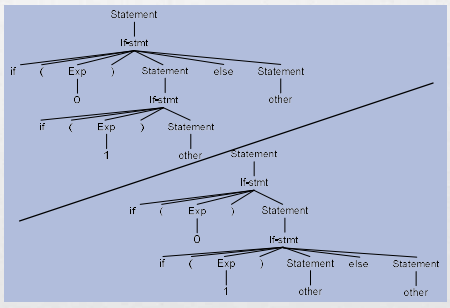


21

Ambiguidade

- Diz-se que uma gramática é **ambígua** se existir pelo menos uma frase da linguagem, gerada pela gramática, com mais do que uma árvore de *parse*
  - A ambiguidade poderá indicar que uma mesma frase pode ter mais do que um significado
  - As gramáticas dos dois exemplos anteriores são ambíguas:





22

Eliminação da Ambiguidade: o caso das expressões

- No caso das expressões a ambiguidade elimina-se definindo para cada operador uma precedência e uma associatividade
- A gramática deve ser modificada para levar em conta a precedência e a associatividade dos operadores

Podemos definir dois níveis de prioridade:
•a mais baixa para os operadores aditivos: + e -
•a mais alta para os operadores multiplicativos: * e /
Para cada um destes níveis de prioridade associa-se um não-terminal da gramática: neste caso denominando-os **termos** (para os operadores aditivos) e **factores** (para os multiplicativos)
A nova gramática, levando em conta estes níveis de prioridade, poderia ser:

Exp → Exp AddOp Exp | Term
AddOp → + | -
Term → Term MulOp Term | Factor
MulOp → * | /
Factor → (Exp) | num

Esta gramática resolve o problema da ambiguidade do exemplo anterior, mas ainda mantém a ambiguidade, p. ex. na expressão **num - num + num**

Para resolver este último problema falta definir uma associatividade dos vários níveis de precedência, que pode ser:
- **à esquerda**, as operações do mesmo nível são efectuadas da esquerda para a direita
- **à direita**, as operações do mesmo nível são efectuadas da direita para a esquerda

Modificar a gramática para levar em conta a associatividade dos operadores implica misturar os não-terminais dos níveis de precedência nas regras gramaticais:

Exp → Exp AddOp Term | Term
AddOp → + | -
Term → Term MulOp Factor | Factor
MulOp → * | /
Factor → (Exp) | num

Aqui define-se, para ambos os níveis, a **associatividade à esquerda**.

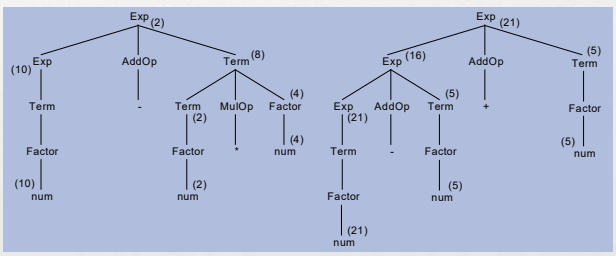
23

Exemplo

Usando a última gramática para expressões apresentada na página anterior, calcular:

10 - 2 * 4 e **21 - 5 + 5**

Com a gramática da página anterior qualquer derivação das expressões apresentadas só tem uma árvore de *parse*:



24

Eliminação da Ambiguidade: O caso if-then-else

- Para resolver este caso clássico de ambiguidade é também necessário modificar a gramática de modo a aceitar só uma das possíveis árvores de *parse*, ou modificar a linguagem ...

Para associar cada **else** com o último **if** podemos ter a gramática:

Statement → Stmt1 | Stmt2
Stmt1 → if (Exp) Stmt1 else Stmt1 | other
Stmt2 → if (Exp) Statement |
if (Exp) Stmt1 else Stmt2
Exp → 0 | 1

Outra solução poderá ser modificar a linguagem de modo a que a instrução **if** tenha sempre um terminador **fi**:

Statement→ If-stmt | other
If-stmt → if (Exp) Statement fi |
if (Exp) Statement else Statement fi
Exp → 0 | 1

Assim não há dúvidas na diferença entre:

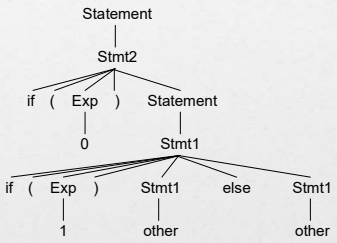
if (0) if (1) other fi else other fi e
if (0) if (1) other else other fi fi

25

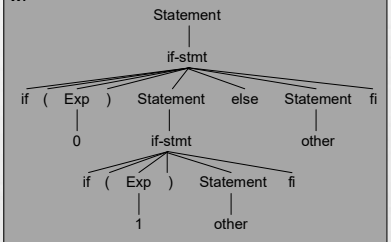
Árvores de *parse* para as gramáticas anteriores

A frase:

if (0) if (1) other else other que tinha 2 árvores de *parse* na gramática anterior, com a gramática anterior passa a ter uma única interpretação:



Para a frase:
if (0) if (1) other fi else other fi não há qualquer dúvida se usarmos a gramática que introduz o terminador **fi** nas instruções **if**:



26

Outra notação para especificar regras gramaticais (EBNF)

- Considerem-se os seguintes pares de regras: **A →A α | β ou A → α A | β** (designadas por regras recursivas à esquerda ou à direita)
  - Por aplicação sucessiva da 1ª regra gera-se uma sequência de α's, que pode ser finalizada pela aplicação da 2ª regra:
    - Teríamos então gerado as sequências: β α α …α α ou α α …α α β
  - Este tipo de sequências pode exprimir-se em EBNF como: **A → β {α} ou A → {α} β, em que { …} significa 0 ou mais …**
  - Outro exemplo: Exp →Exp AddOp Term | Term expresso como:
    Exp →Term { AddOp Term }
- Considere-se agora: Stmt → If-stmt | other e
  If-stmt → if (Exp) Stmt | if (Exp) Stmt else Stmt
  - Este último par de regras para If-Stmt pode ser expresso em EBNF como:
    If-stmt → if (Exp) Stmt [else Stmt], onde **[…] significa que …é opcional (pode ocorrer ou não)**
  - Esta construção também pode substituir construções recursivas como:
    Exp →Term AddOp Exp | Term que pode ser substituída por:
    Exp →Term [AddOp Exp]
- Sempre que uma **gramática seja especificada na forma EBNF pode sempre redefinir-se para a forma BNF** (definindo a mesma linguagem)

27