Aula 02 – Análise Sintática (*Parsing*)

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

- Por ser um sistema bastante complexo, um compilador é dividido em módulos.
- O primeiro módulo do compilador realiza a análise léxica.
- O segundo módulo realiza a análise sintática (*parsing*).
- **Estes *slides***: discussão sobre os principais conceitos de análise sintática.
- **Objetivos**: apresentar a teoria fundamental de análise sintática aplicada em compiladores.

Referências

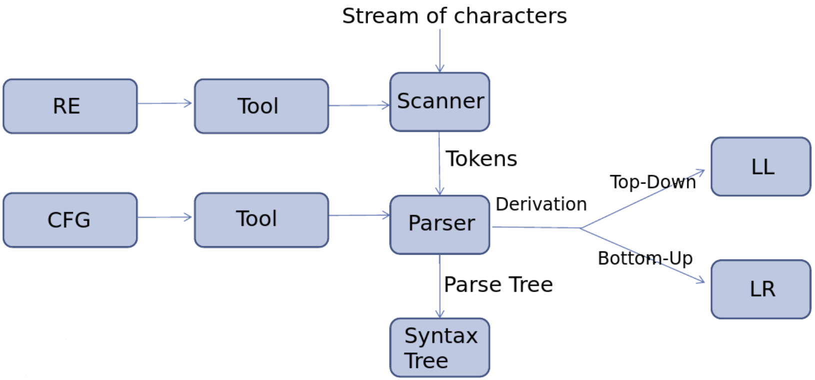
Chapters 3, 4 & 5
K. C. Louden
Chapter 3 – Parsers
K. D. Cooper
Chapters 4 & 5
D. Thain

Parte I

Gramáticas Livre de Contexto e Análise Sintática

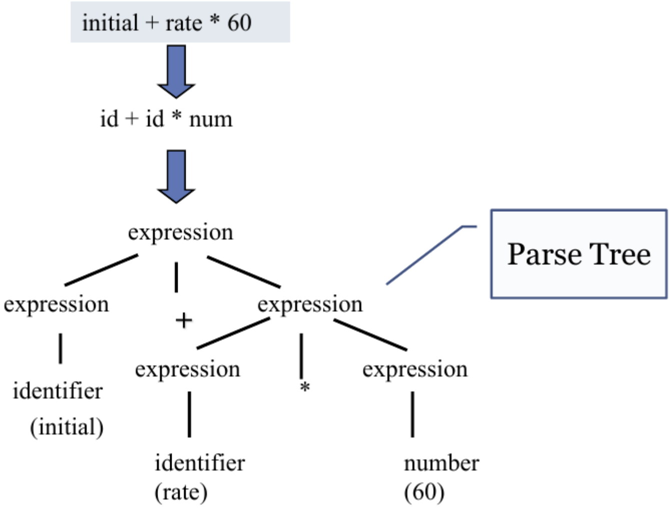
Introdução

- **Análise sintática (*parsing*)**: determinar a estrutura (sintaxe) de um programa.
- Figura abaixo mostra uma visão geral do processo de *scanning* e *parsing*. (CFG – *contex-free grammar*, gramática livre de contexto. LL e LR $\Rightarrow$ Partes II e III.)



Função de um *Parser*

A função fundamental de um *parser* é transformar uma sequência de *tokens* em uma árvore de derivação (*parse tree*).



Funcionamento de um *Parser*

- A sequência de *tokens* não é um parâmetro explícito de entrada do *parser*.
- O *parser* chama a função `getToken` (`yylex`) do *scanner* para obter o próximo *token*.
- O funcionamento do *parser* depende do tipo de compilador.
- Compilador *single-pass*:
  - Não é necessária uma construção explícita da *parse tree*.
  - Processo de *parsing* é realizado em conjunto com as demais operações do compilador.

Funcionamento de um *Parser*

Compilador *multi-pass*:

- O *parser* produz como resultado uma *parse tree* ou alguma outra representação intermediária do código.
- Passadas seguintes do compilador usam a árvore como entrada.
- A estrutura da árvore depende da estrutura sintática da linguagem de entrada (em particular, da gramática).
- A árvore é construída como uma estrutura de dados dinâmica (ponteiros e alocação no *heap*).
- Os nós da árvore contém campos para armazenar informações (atributos) usados em todo o processo de compilação.

Tratamento de Erros

- Tratamento de erro do *parser* de um compilador real:
  - Exibe uma mensagem de erro.
  - Tenta se recuperar do erro e continuar o processo de análise sintática.
  - Importante para tentar encontrar a maior quantidade de erros possíveis durante a compilação.
  - Maior desafio é a geração de mensagens de erro significativas e fazer o reinício do *parser* o mais próximo possível do ponto aonde o erro ocorreu.
- Tratar erros adequadamente é um dos maiores problemas no desenvolvimento de um *parser*.
- Nos laboratórios e trabalho, vamos simplificar o processo e parar a compilação assim que o primeiro erro for detectado.

Gramáticas Livres de Contexto

- Linguagens Regulares são classificadas como tipo 3 na Hierarquia de Chomsky (HC) e podem ser descritas por expressões regulares (*regular expressions* – REs).
- REs não são poderosas o suficiente para descrever muitas das estruturas comuns que aparecem em linguagens de programação (LPs).
- Por exemplo, REs não conseguem expressar recursão e linguagens aninhadas (e.g., colchetes casados `[[...]]`).
- $\Rightarrow$ É necessário um tipo de linguagem mais expressiva para descrever a sintaxe de LPs.
- Usa-se as linguagens livres de contexto (tipo 2 na HC).
- Linguagens livres de contexto são descritas (geradas) por gramáticas livres de contexto (*context-free grammars* – CFGs).

Gramáticas Livres de Contexto

Definição – CFG

Uma CFG é uma tupla $G = \langle T, N, S, P \rangle$ onde:

- T é um conjunto de símbolos terminais;
- N é um conjunto de símbolos não-terminais;
- $S \in N$ é o símbolo inicial; e
- P é um conjunto de produções (regras de reescrita) da forma $A \rightarrow \alpha$, onde $A \in N$ e $\alpha \in (T \cup N)^*$.

No cenário de compiladores:

- T: os tipos de *tokens* reconhecidos pelo *scanner*.
- N: todos os símbolos da gramática que não são *tokens*.
- S: símbolo que representa toda a estrutura de um programa.
- P: o conjunto de regras recursivas que permite a geração de qualquer programa da LP com sintaxe válida.

Comparação entre CFGs e REs

- Uma CFG especifica a estrutura sintática de uma LP.
- Uma CFG é formada por regras recursivas.
- *Exemplo*: as regras abaixo compõem uma gramática de expressões aritméticas.

```
exp -> exp op exp | (exp) | NUM
op  -> + | - | * | /
```

- Nas regras acima, o símbolo | indica uma escolha.
- Na verdade, | é apenas uma simplificação de notação para a escrita de várias regras em uma mesma linha.
- Portanto, temos 7 regras no exemplo acima.

Comparação entre CFGs e REs

- A definição formal de uma CFG pode ser extraída facilmente das regras.
- Do exemplo, temos a CFG $G = \langle T, N, S, P \rangle$ onde:
  - $T = \{\,\texttt{(}, \texttt{)}, \texttt{NUM}, +, -, \star, /\}$, símbolos que não aparecem na cabeça de nenhuma regra.
  - $N = \{\texttt{exp}, \texttt{op}\}$, símbolos que aparecem na cabeça das regras (e portanto podem ser reescritos).
  - $S = \texttt{exp}$, a cabeça da primeira regra é o símbolo inicial.
  - P é o conjunto das 7 regras apresentadas anteriormente.

- Ao contrário das CFGs, as REs não permitem recursões.
- *Exemplo*: as REs abaixo indicam a representação léxica de um número.

```
digit  = 0|1|2|3|4|5|6|7|8|9
number = digit digit*
```

Comparação entre CFGs e REs

Em uma RE:

- Três operações: escolha (|), concatenação (sem meta-símbolo), repetição (*).
- = representa a definição de um padrão.

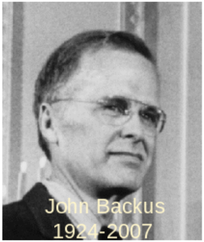
Em uma CFG:

- Três operações: escolha (|), concatenação (sem meta-símbolo), repetição (expressa por recursão).
- -> representa a reescrita de um símbolo não-terminal.

- Os meta-símbolos e convenções de CFGs usados aqui são bastante utilizados mas não existe um padrão universal a ser seguido.
- Alternativas comuns para -> incluem =, : (usado na ferramenta Bison) e ::=.

Comparação entre CFGs e REs

- As convenções apresentadas para as regras de CFGs são chamadas de Backus-Naur Form (BNF).
- BNF foi usada por John Backus e Peter Naur para descrever a sintaxe da LP Algol 60.



- A notação original utiliza o meta-símbolo ::= e cerca os não-terminais por < e >.
- Nessa representação, o exemplo anterior fica:

```
<exp> ::= <exp> <op> <exp> | (<exp>) | NUM
<op>  ::= + | - | * | /
```

Comparação entre CFGs e REs

- Regras de uma CFG determinam as sequências de *tokens* que são sintaticamente válidas.
- *Exemplo*: para a gramática de expressões aritméticas:
  - `(34-3)*42` é uma expressão válida,
  - enquanto `(34-3*42` não é.
- As sequências válidas de *tokens* são obtidas através de derivações na CFG.
- Uma derivação é uma sequência de aplicações das regras da CFG.
- Uma derivação começa com o não-terminal *S* e termina com uma sequência de (terminais) *tokens*.
- A cada passo da derivação um único não-terminal é reescrito através da aplicação de uma das regras da CFG.

Derivações

Exemplo: derivação da expressão `(34-3)*42` pelas regras da gramática.

- Passos da derivação são indicados por $\Rightarrow$.
- Cada passo tem a regra aplicada indicada do lado direito.

(1)	$exp \Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
(2)	$\Rightarrow exp\ op$ **number**	$[exp \rightarrow$ **number**$]$
(3)	$\Rightarrow exp$ ***** **number**	$[op \rightarrow$ ***** $]$
(4)	$\Rightarrow$ **(** exp **)** ***** **number**	$[exp \rightarrow$ **(** exp **)**$]$
(5)	$\Rightarrow$ **(** $exp\ op\ exp$ **)** ***** **number**	$[exp \rightarrow exp\ op\ exp]$
(6)	$\Rightarrow$ **(**$exp\ op$ **number)** ***** **number**	$[exp \rightarrow$ **number**$]$
(7)	$\Rightarrow$ **(**exp **-** **number)** ***** **number**	$[op \rightarrow$ **-** $]$
(8)	$\Rightarrow$ **(number - number)*number**	$[exp \rightarrow$ **number**$]$

Linguagem de uma Gramática

Linguagem de uma CFG

Dado uma CFG $G = \langle T, N, S, P \rangle$, o conjunto de todas as sequências de *tokens* obtidas por derivações a partir do símbolo inicial S é a linguagem da gramática G, definida por

$$L(G) = \{s \mid S \Rightarrow^* s\} \quad .$$

- Na definição acima, s é uma sentença que representa uma sequência qualquer de *tokens*.
- $\Rightarrow^*$ indica uma sequência de zero ou mais passos da derivação.

Linguagem de uma Gramática – Exemplos

Exemplo 3.1 (do livro do Louden)

Considere a gramática G formada pelas regras

$$E \to (E)\,|a \quad .$$

Essa gramática gera a linguagem

$$L(G) = \{a, (a), ((a)), \ldots\} = \{(^n a)^n \mid n \geq 0\} \quad .$$

A derivação para a *string* $((a))$ é dada por

$$E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow ((a)) \quad .$$

Obs.: Se G não tivesse a segunda regra, a sua linguagem seria vazia, pois não haveria uma forma de derivar uma *string* de terminais (recursão infinita).

Linguagem de uma Gramática – Exemplos

Exemplo 3.4 (do livro do Louden)

Considere a seguinte gramática de declarações:

$$
\begin{aligned}
&\textit{statement} \rightarrow \textit{if-stmt} \mid \textbf{other} \\
&\textit{if-stmt} \rightarrow \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{statement} \\
&\qquad\qquad \mid \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{statement}\ \textbf{else}\ \textit{statement} \\
&\textit{exp} \rightarrow \textbf{0} \mid \textbf{1}
\end{aligned}
$$

A linguagem dessa gramática consiste de comandos `if` aninhados como em C. Exemplos de *strings* válidas:

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

Recursão

- As regras gramaticais

$$A \rightarrow Aa \,|\, a \qquad \text{ou} \qquad A \rightarrow aA \,|\, a$$

  geram a linguagem $\{a^n \mid n \geq 1\}$ (o conjunto de todas as *strings* com um ou mais a's).
- A *string* $aaaa$ pode ser gerada pelas duas primeiras regras acima através da derivação:

$$A \Rightarrow Aa \Rightarrow Aaaa \Rightarrow Aaaa \Rightarrow aaaa \quad .$$

- Regras são classificadas quanto à recursão pela posição aonde a cabeça da regra aparece no corpo.
  - Regra recursiva à esquerda: $A \rightarrow Aa$, a cabeça A aparece como o primeiro símbolo do corpo.
  - Regra recursiva à direita: $A \rightarrow aA$, a cabeça A aparece como o último símbolo do corpo.

String Vazia e Recursão

- O meta-símbolo ϵ (epsilon) representa a *string* vazia.
- Regras da forma *empty* $\rightarrow \epsilon$ são chamadas de ϵ-*productions*.
- Uma gramática que gera uma linguagem contendo a *string* vazia deve ter pelo menos uma ϵ-*production*.
- *Exemplo*: As regras gramaticais

$$A \rightarrow Aa \,|\, \epsilon \qquad \text{ou} \qquad A \rightarrow aA \,|\, \epsilon$$

geram a linguagem $\{a^n \mid n \geq 0\}$ (o conjunto de todas as *strings* com zero ou mais a's).

Exemplo 3.5 (do livro do Louden)

Considere a gramática

$$A \rightarrow (A)A \mid \epsilon$$

que gera *strings* de parênteses balanceados.

A *string* (()(()))() é gerada pela derivação a seguir.

$$\begin{aligned} A &\Rightarrow (A)A \Rightarrow (A)(A)A \Rightarrow (A)(A) \Rightarrow (A)() \Rightarrow ((A)A)() \\ &\Rightarrow (()A)() \Rightarrow (()(A)A)() \Rightarrow (()(A))() \\ &\Rightarrow (()((A)A))() \Rightarrow (()(()A))() \Rightarrow (()(()))() \end{aligned}$$

A ϵ-*production* é usada para fazer o símbolo A desaparecer quando necessário.

String Vazia e Recursão – Exemplos

Exemplo 3.6 (do livro do Louden)

A gramática de declarações do Exemplo 3.4 pode ser reescrita usando uma ϵ-*production*.

$$\begin{aligned}
&\textit{statement} \rightarrow \textit{if-stmt} \mid \texttt{other} \\
&\textit{if-stmt} \rightarrow \texttt{if (} \textit{exp} \texttt{) } \textit{statement else-part} \\
&\textit{else-part} \rightarrow \texttt{else } \textit{statement} \mid \varepsilon \\
&\textit{exp} \rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}$$

A ϵ-*production* indica que a construção *else-part* é opcional.

Derivação vs. Estrutura

- Derivações não representam a estrutura das *strings* de forma única.
- Podem existir várias derivações para a mesma *string*.

Exemplo: derivação da expressão `(34-3)*42` pelas regras da gramática de expressões aritméticas (como já visto).

$$
\begin{array}{lll}
(1) & exp \Rightarrow exp\ op\ exp & [exp \rightarrow exp\ op\ exp] \\
(2) & \Rightarrow exp\ op\ \mathbf{number} & [exp \rightarrow \mathbf{number}] \\
(3) & \Rightarrow exp\ \mathbf{*}\ \mathbf{number} & [op \rightarrow \mathbf{*}\ ] \\
(4) & \Rightarrow \mathbf{(}\ exp\ \mathbf{)}\ \mathbf{*}\ \mathbf{number} & [exp \rightarrow \mathbf{(}\ exp\ \mathbf{)}] \\
(5) & \Rightarrow \mathbf{(}\ exp\ op\ exp\ \mathbf{)}\ \mathbf{*}\ \mathbf{number} & [exp \rightarrow exp\ op\ exp] \\
(6) & \Rightarrow \mathbf{(}exp\ op\ \mathbf{number)}\ \mathbf{*}\ \mathbf{number} & [exp \rightarrow \mathbf{number}] \\
(7) & \Rightarrow \mathbf{(}exp\ \mathbf{-}\ \mathbf{number)}\ \mathbf{*}\ \mathbf{number} & [op \rightarrow\ \mathbf{-}\ ] \\
(8) & \Rightarrow \mathbf{(number\ -\ number)\ *number} & [exp \rightarrow \mathbf{number}]
\end{array}
$$

Derivação vs. Estrutura

A mesma expressão admite a derivação alternativa a seguir.

(1)	$exp \Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
(2)	$\Rightarrow \mathtt{(}exp\mathtt{)}\ op\ exp$	$[exp \rightarrow \mathtt{(}\ exp\ \mathtt{)}]$
(3)	$\Rightarrow \mathtt{(}exp\ op\ exp\mathtt{)}\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
(4)	$\Rightarrow \mathtt{(number}\ op\ exp\mathtt{)}\ op\ exp$	$[exp \rightarrow \mathtt{number}]$
(5)	$\Rightarrow \mathtt{(number} - exp\mathtt{)}\ op\ exp$	$[op \rightarrow \mathtt{-}]$
(6)	$\Rightarrow \mathtt{(number - number)}\ op\ exp$	$[exp \rightarrow \mathtt{number}]$
(7)	$\Rightarrow \mathtt{(number - number)\ *}\ exp$	$[op \rightarrow \mathtt{*}]$
(8)	$\Rightarrow \mathtt{(number - number)\ *\ number}$	$[exp \rightarrow \mathtt{number}]$

Derivação vs. *Parse Tree*

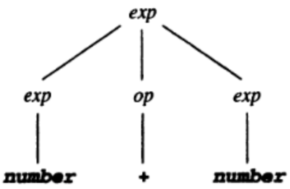
- *Parse Tree* – também conhecida como: árvore de derivação, árvore (de análise) sintática, árvore de sintaxe concreta, etc.
- Tanto uma derivação como uma *parse tree* servem para representar a construção (estrutura) de uma dada *string*.
- Derivações são estruturas lineares: ordem de reescrita as distinguem.
- Por outro lado, *parse trees* são estruturas ramificadas: destaque para a organização e não para a ordem.
- Uma *parse tree* pode representar mais de uma derivação.

Derivação vs. *Parse Tree*

- Uma *parse tree* associada a uma derivação é uma árvore rotulada.
- Nós interiores são rotulados por não-terminais.
- Nós folha são rotulados por terminais.
- Filhos de cada nó indicam a reescrita realizada em um passo da derivação.

Exemplo: a derivação mostrada na figura da esquerda corresponde à *parse tree* da figura da direita.

$$\begin{aligned} exp &\Rightarrow exp\ op\ exp \\ &\Rightarrow \mathtt{number}\ op\ exp \\ &\Rightarrow \mathtt{number} + exp \\ &\Rightarrow \mathtt{number} + \mathtt{number} \end{aligned}$$



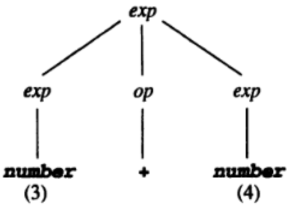
Derivação vs. *Parse Tree*

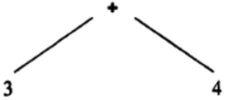
- A *parse tree* anterior corresponde a inúmeras derivações.
- Algumas derivações seguem uma regra de escolha do não-terminal a ser reescrito.
- A derivação do slide anterior é dita derivação mais à esquerda (*leftmost derivation*).
- *Exemplo* – Derivação mais à direita (*rightmost derivation*):

$$
\begin{aligned}
\textit{exp} &\Rightarrow \textit{exp op exp} \\
&\Rightarrow \textit{exp op}\ \texttt{number} \\
&\Rightarrow \textit{exp} + \texttt{number} \\
&\Rightarrow \texttt{number} + \texttt{number}
\end{aligned}
$$

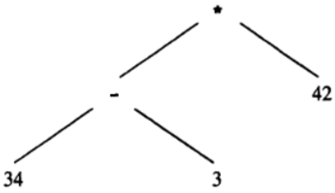
- **Importante**: não confunda derivação à esquerda ou direita com recursividade à esquerda ou direita.
  - Derivação mais à esquerda ou à direita: característica da sequência de aplicação das regras.
  - Recursividade à esquerda ou à direita: característica da estrutura das regras.

- A *parse tree* contém mais informação do que é necessário para as demais fases do compilador.
- Nesse caso, a *parse tree* pode ser simplificada, gerando uma árvore de sintaxe abstrata (*abstract syntax tree* – AST).
- *Exemplo*: Para a expressão `3+4`, temos a seguinte *parse tree* e AST, respectivamente.





- *Exemplo*: A expressão `(34-3)*42` pode ser representada pela AST abaixo.
- Note que os *tokens* de parênteses foram descartados.
- Estrutura da árvore já indica a semântica usual de parênteses na aritmética.



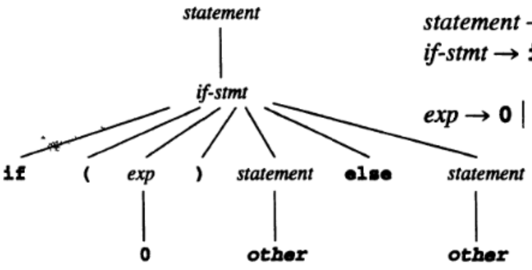
Parse Tree vs. AST – Exemplos

$$
\begin{aligned}
\textit{statement} &\rightarrow \textit{if-stmt} \mid \texttt{other} \\
\textit{if-stmt} &\rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement} \\
&\quad\mid \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement}\ \texttt{else}\ \textit{statement} \\
\textit{exp} &\rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

INPUT

`if (0) other else other`

Parse Tree vs. AST – Exemplos

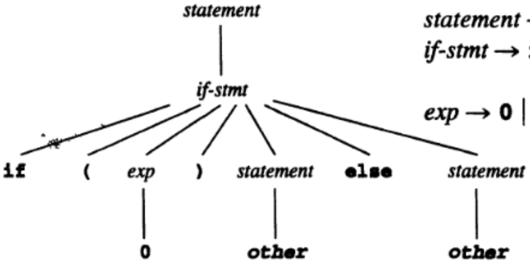


statement → *if-stmt* | **other**
if-stmt → **if** **(** *exp* **)** *statement*
  | **if** **(** *exp* **)** *statement* **else** *statement*
exp → **0** | **1**

INPUT

if (0) other else other

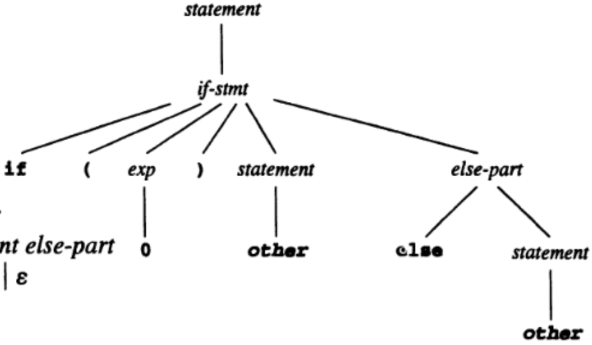
Parse Tree vs. AST – Exemplos



statement → *if-stmt* | **other**
if-stmt → **if (** *exp* **)** *statement*
| **if (** *exp* **)** *statement* **else** *statement*
exp → **0** | **1**

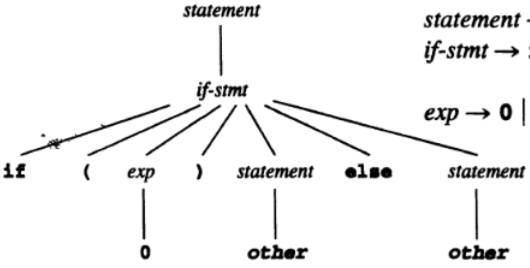
INPUT

`if (0) other else other`



statement → *if-stmt* | **other**
if-stmt → **if (** *exp* **)** *statement else-part*
else-part → **else** *statement* | ε
exp → **0** | **1**

Parse Tree vs. AST – Exemplos



statement → *if-stmt* | **other**

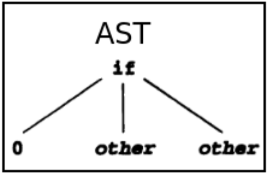
if-stmt → **if** **(** *exp* **)** *statement*

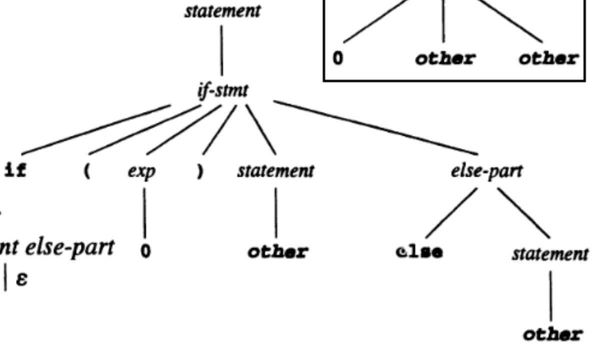
| **if** **(** *exp* **)** *statement* **else** *statement*

exp → **0** | **1**

INPUT

if (0) other else other





statement → *if-stmt* | **other**

if-stmt → **if** **(** *exp* **)** *statement else-part*

else-part → **else** *statement* | ε

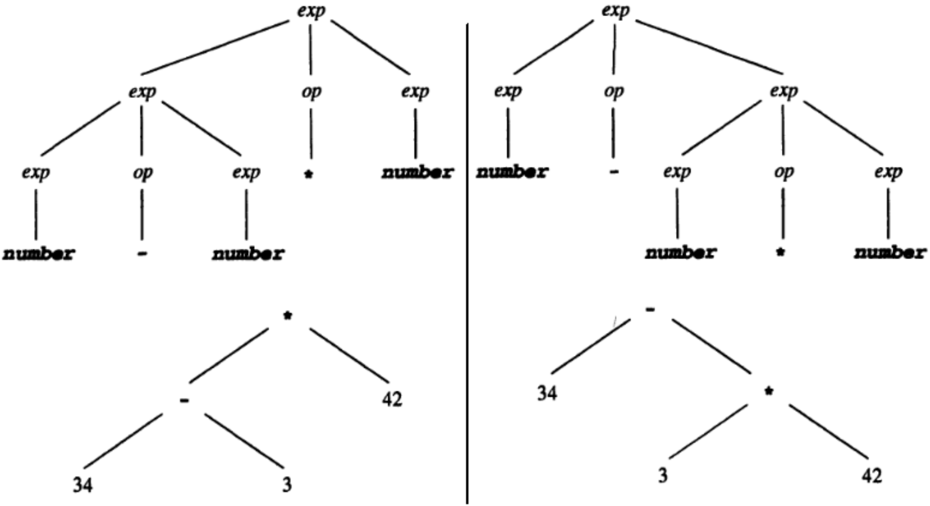
exp → **0** | **1**

Ambiguidade

- *Parse trees* representam a estrutura sintática do programa de entrada segundo as regras da CFG.
- Como já visto, uma mesma *parse tree* pode representar diferentes derivações na CFG.
- Nesse sentido, podemos dizer que a *parse tree* (e AST associada) serve como representação única da entrada.
- No entanto, essa afirmação só é válida quando a CFG for livre de ambiguidade (não-ambígua).
- *Exemplo*: considere novamente a gramática de expressões aritméticas abaixo, e a entrada `34-3*42`.

```
exp -> exp op exp | (exp) | NUM
op  -> + | - | * | /
```

É possível construir duas *parse trees*, correspondentes a duas derivações mais à esquerda distintas.



Ambiguidade

- Gramática ambígua: gera uma *string* que admite mais de uma *parse tree* distinta.
- Note que ambiguidade é uma propriedade (das regras) da gramática.
- Uma CFG ambígua causa um grande problema para o *parser*: CFG não especifica de forma precisa a estrutura sintática da LP.
- Infelizmente não existe uma forma automática para remoção de ambiguidade.
- Determinar se uma dada CFG é ambígua é um problema indecidível [Hopcroft & Ullman, 1979].
- Felizmente, existem subclasses de CFGs para as quais o teste de ambiguidade é decidível.
  Ex.: gramáticas LR(k) [Knuth, 1965].

Ambiguidade

- Mesmo quando é possível usar um algoritmo para testar ambiguidade, não existe uma construção equivalente para remover ambiguidade de forma automática.
- Situação levou ao desenvolvimento de técnicas padrão para lidar com ambiguidade no cenário de compiladores.
- Existem dois métodos básicos para tal.
  1. Regras de desambiguação: especificam, em cada caso de ambiguidade, qual das árvores é a correta.
  2. Modificar a gramática: requer a elaboração de uma forma equivalente da CFG que força a construção da árvore correta (elimina a ambiguidade).
- Todos os construtores de *parsers* permitem a especificação de regras de desambiguação.
- Método 1 é preferível pois não requer mudanças na gramática.

Removendo Ambiguidade – Modificando a CFG

- Para remover a ambiguidade na gramática de expressões aritméticas, é necessário indicar a precedência das operações.
- Caso seja usado o método 2 (modificar a gramática), precisamos agrupar os operadores em níveis de precedência.
- Para cada nível escrevemos uma regra distinta.
- *Exemplo*: A precedência de multiplicação e divisão sobre adição e subtração pode ser caracterizada pela gramática modificada exibida a seguir.

Removendo Ambiguidade – Modificando a CFG

$$
\begin{aligned}
&\textit{exp} \rightarrow \textit{exp addop exp} \mid \textit{term}\\
&\textit{addop} \rightarrow \texttt{+} \mid \texttt{-}\\
&\textit{term} \rightarrow \textit{term mulop term} \mid \textit{factor}\\
&\textit{mulop} \rightarrow \texttt{*} \mid \texttt{/}\\
&\textit{factor} \rightarrow \texttt{(}\ \textit{exp}\ \texttt{)} \mid \texttt{number}
\end{aligned}
$$

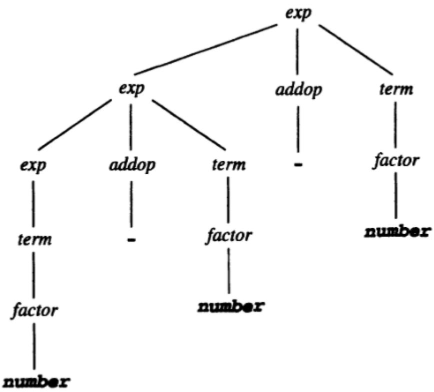
- Multiplicação e divisão são agrupadas sob *term*, e soma e subtração sob *exp*.
- Como o caso base para *exp* é *term* (segunda escolha), soma e subtração aparecem mais perto da raiz na árvore.
- Isso caracteriza uma menor precedência.
- Cascata de precedências: agrupamento de operadores em níveis distintos de precedência na notação BNF.
- Cascata de precedências torna as *parse trees* mais complexas mas não afetam as ASTs.

Removendo Ambiguidade – Modificando a CFG

- A gramática modificada ainda é ambígua: não especifica associatividade dos operadores.
- *Exemplo*: a expressão `34-3-42` admite duas *parse trees*, correspondentes às interpretações `(34-3)-42` ou `34-(3-42)`.
- A maior parte das operações aritméticas é associativa à esquerda. Exceção: exponenciação.
- Associatividade é definida alterando-se a regra que define o nível precedência da operação:
  - Regra recursiva à esquerda: operador associativo à esquerda.
  - Regra recursiva à direita: operador associativo à direita.

Versão final da gramática com operadores associativos à esquerda e *parse tree* para expressão `34-3-42`.

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → ***** | **/**
factor → **(** *exp* **)** | **number**



⇒ Removendo ambiguidade sem alterar a gramática: comandos especiais do `bison` (Laboratório 02).

Problema do *Else* Pendente

A gramática abaixo é ambígua devido ao *else* opcional.

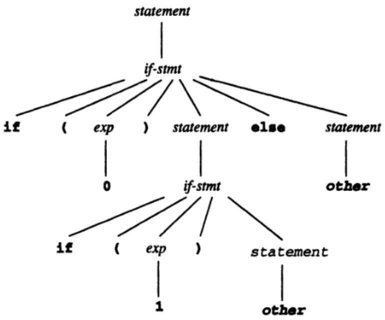
$$
\begin{aligned}
&\textit{statement} \rightarrow \textit{if-stmt} \mid \mathbf{other} \\
&\textit{if-stmt} \rightarrow \mathbf{if}\ (\ \textit{exp}\ )\ \textit{statement} \\
&\qquad\qquad \mid \mathbf{if}\ (\ \textit{exp}\ )\ \textit{statement}\ \mathbf{else}\ \textit{statement} \\
&\textit{exp} \rightarrow \mathbf{0} \mid \mathbf{1}
\end{aligned}
$$

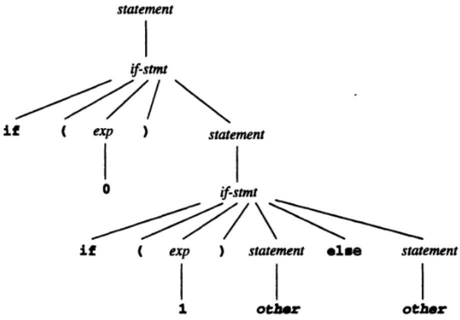
- Esse é o chamado problema do *else* pendente (*dangling else problem*).
- Originário da sintaxe da LP Algol 60.
- É possível alterar as regras da gramática para eliminar a ambiguidade.
- Mais fácil é usar uma palavra reservada para fechar o *if*: caso do Algol 68, Ada, etc.

Para a entrada

```
if (0) if (1) other else other
```

a gramática admite as duas *parse trees* abaixo.





Problema do *Else* Pendente

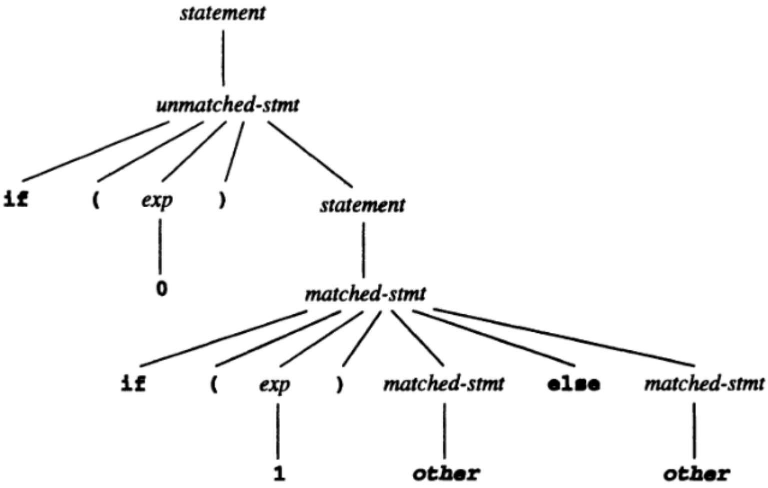
- Qual das duas árvores é a correta?
- $\Rightarrow$ A segunda: convencionou-se que o *else* é sempre associado ao *if* mais próximo.
- Gramática pode ser reescrita como a seguir.

$$
\begin{aligned}
&\textit{statement} \rightarrow \textit{matched-stmt} \mid \textit{unmatched-stmt} \\
&\textit{matched-stmt} \rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{matched-stmt}\ \texttt{else}\ \textit{matched-stmt} \mid \texttt{other} \\
&\textit{unmatched-stmt} \rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement} \\
&\qquad\qquad\qquad \mid \texttt{if}\ (\ \textit{exp}\ )\ \textit{matched-stmt}\ \texttt{else}\ \textit{unmatched-stmt} \\
&\textit{exp} \rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

- Regras permitem somente um *matched-stmt* antes de um *else*.
- Força todos os blocos de *else* a serem casados o mais cedo possível.

Problema do *Else* Pendente

Parse tree gerada pela gramática modificada.



Parte II

Top-Down Parsing

Top-Down Parsing – Introdução

- A tarefa de encontrar a *parse tree* requer:
  - Símbolo inicial.
  - Entrada a ser analisada.
- Existem duas formas fundamentais para detectar a relação (estrutura) entre os dois:
  - Análise sintática descendente (*top-down parsing*).
  - Análise sintática ascendente (*bottom-up parsing*).
- Um *parser top-down* processa os *tokens* realizando uma derivação mais à esquerda.
- *Parsers top-down* podem ser de duas formas:
  - *Backtracking*: evitado pois é muito ineficiente – complexidade exponencial.
  - Preditivo: usado na prática – em contraste com *parsing* oracular, que não é implementável.

Top-Down Parsing – Introdução

- ***Parsing* preditivo**: deseja-se a habilidade de "adivinhar" corretamente qual regra aplicar.
- Obtida olhando-se um *token* à frente na sequência: *look-ahead*.
- Existem dois métodos para *parsing top-down* preditivo:
  - *Parsing* descendente recursivo (*recursive-descent*): versátil, adequado para *parsers* escritos à mão.
  - *Parsing* LL(1): algoritmo baseado em pilha. Usado no ANTLR.
- LL(k): *Left-to-right, Leftmost, look-ahead k*.
- *Left-to-right*: processa a entrada da esquerda para a direita.
- *Leftmost*: constrói a derivação mais à esquerda.
- *Look-ahead k*: usa k símbolos de *look-ahead*. Para a grande maioria das LPs, é suficiente usar $k = 1$.

Método Básico de Descida Recursiva

- Ideia básica do método: um regra $A \rightarrow \beta$ é uma definição de um procedimento para reconhecer A.
- O lado direito da regra β descreve a estrutura do código.
- *Exemplo*: regra `factor -> (exp) | NUMBER` pode ser reconhecida pelo procedimento abaixo.

```
procedure factor ;
begin
  case token of
  ( :   match( ( ) ;
        exp ;
        match( ) ) ;
  number :
        match(number) ;
  else error ;
  end case ;
end factor ;
```

- Variável *token*: próximo símbolo da entrada (*look-ahead*).
- Procedimento *match*: casa o próximo símbolo e avança entrada.
- Procedimento *exp*: múltiplas chamadas dos não-terminais.

Método Básico de Descida Recursiva

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

Procedimento *match*:

- Casa o *look-ahead* (variável *token*) com o parâmetro *expectedToken*.
- Em caso de sucesso: avança a entrada.
- Caso contrário: declara um erro e aborta.
- Chamadas de *match*(`(`) e *match*(`number`): certeza de sucesso.
- Não é o caso para *match*(`)`): programa de entrada pode estar incorreto.

EBNF: *Extended Backus-Naur Form*

- Formato das regras usando BNF:
  - **Lado esquerdo**: não-terminal.
  - **Lado direito**: sequência de terminais e não-terminais (forma sentencial).
  - Múltiplas regras para um mesmo não-terminal separadas por $|$.
- Formato das regras usando EBNF:
  - **Lado direito**: admite o uso de expressões regulares.
  - O restante continua igual ao BNF.
- Sempre é possível converter uma gramática em EBNF para uma outra equivalente em BNF.
- Uso de EBNF é mais conveniente, especialmente para o desenvolvimento de *recursive-descent parsers*.
- ANTLR aceita gramáticas em EBNF, Bison não.

EBNF: *Extended Backus-Naur Form*

- Colchetes indicam que uma parte do corpo é opcional.
- *Exemplo*: a regra em EBNF

$$\textit{if-stmt} \rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement}\ [\ \texttt{else}\ \textit{statement}\ ]$$

é equivalente às regras em BNF

$$\begin{aligned}\textit{if-stmt} \rightarrow\ & \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement} \\ |\ & \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement}\ \texttt{else}\ \textit{statement}\end{aligned}$$

- Chaves indicam repetição.
- *Exemplo*: a regra em EBNF

$$\textit{exp} \rightarrow \textit{term}\ \{\ \textit{addop term}\ \}$$

é equivalente às regras em BNF

$$\textit{exp} \rightarrow \textit{exp addop term} \mid \textit{term}$$

Exemplo – Reconhecendo um *if*

A regra em EBNF

$$\textit{if-stmt} \rightarrow \texttt{\textbf{if}}\ \texttt{\textbf{(}}\ \textit{exp}\ \texttt{\textbf{)}}\ \textit{statement}\ [\ \texttt{\textbf{else}}\ \textit{statement}\ ]$$

pode ser reconhecida pelo procedimento abaixo.

```
procedure ifStmt ;
begin
   match (if) ;
   match (() ;
   exp ;
   match ()) ;
   statement ;
   if token = else then
      match (else) ;
      statement ;
   end if ;
end ifStmt ;
```

- Colchetes em EBNF são traduzidos para um teste.
- Notação EBNF fica muito próxima do código.
- Se fossem usadas as regras BNF equivalentes, seria necessário escolher uma das duas regras no início da aplicação.
- Escolha não é possível porque o corpo do *if* tem tamanho arbitrário.

Exemplo – Reconhecendo uma *exp*

A regra em EBNF $exp \rightarrow term \ \{\ addop\ term\ \}$ pode ser reconhecida pelo procedimento abaixo.

```
procedure exp ;
begin
   term ;
   while token = + or token = - do
      match (token) ;
      term ;
   end while ;
end exp ;
```

- Chaves em EBNF são traduzidas para um *loop*.
- Se fossem usadas as regras BNF equivalentes teríamos uma recursão infinita: $exp \rightarrow exp\ addop\ term \mid term$.
- *Parsers* de descida recursiva não conseguem lidar com gramáticas BNF recursivas à esquerda.
- Notação EBNF resolve esse problema.

Exemplo – Calculando uma *exp*

- Recursividade à esquerda das regras originais em BNF do slide anterior indicam que os operadores $+$ e $-$ são associativos à esquerda.

```
function exp : integer ;
var temp : integer ;
begin
   temp := term ;
   while token = + or token = - do
      case token of
       + : match (+) ;
            temp := temp + term ;
       - : match (-) ;
            temp := temp - term ;
      end case ;
   end while ;
   return temp ;
end exp ;
```

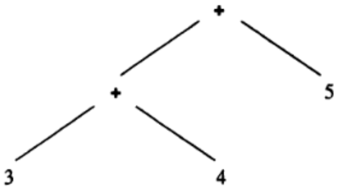
- Essa associatividade é preservada pelo código de *exp* gerado a partir da regra em EBNF.
- Código ao lado ilustra como é possível introduzir ações semânticas nos procedimentos do *parser* de descida recursiva.
- Nesse caso, os procedimentos na verdade são funções que retornam um resultado inteiro.

Construindo uma AST com Ações Semânticas

- É possível introduzir ações semânticas que constroem a AST do programa de entrada.
- É interessante notar que na função abaixo, a construção da AST é *bottom-up* apesar do *parser* ser *top-down*.

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
   temp := term ;
   while token = + or token = - do
      newtemp := makeOpNode(token) ;
      match (token) ;
      leftChild(newtemp) := temp ;
      rightChild(newtemp) := term ;
      temp := newtemp ;
   end while ;
   return temp ;
end exp ;
```

AST para a entrada `3+4+5`:



Construindo uma AST com Ações Semânticas

Para um comando `if` a construção da AST é *top-down*.

```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
   match (if) ;
   match (( ) ;
   temp := makeStmtNode(if) ;
   testChild(temp) := exp ;
   match ( )) ;
   thenChild(temp) := statement ;
   if token = else then
      match (else) ;
      elseChild(temp) := statement ;
   else
      elseChild(temp) := nil ;
   end if ;
end ifStatement ;
```

Flexibilidade do método permite ao programador ajustar facilmente a sequência de ações. ⇒ *Hand-coded parsers*.

Avaliação do *Parser* Descendente Recursivo

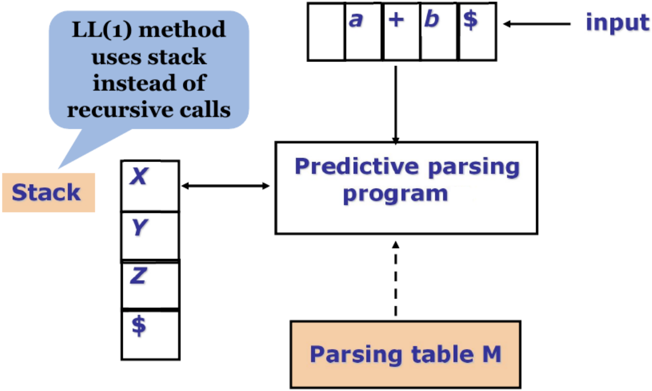
O método descendente recursivo simplesmente traduz gramáticas em procedimentos. É simples de escrever e entender, mas é *ad-hoc* e tem as seguintes inconveniências.

1. Pode ser difícil se converter uma gramática em BNF para uma equivalente em EBNF.
2. É difícil de decidir entre as regras $A \to \alpha$ e $A \to \beta$, se ambos α e β começam com não-terminais. Requer a computação dos conjuntos FIRST.
3. Pode ser necessário saber quais *tokens* podem suceder legalmente o não-terminal A. Requer a computação dos conjuntos FOLLOW.

Por essas razões, foram investigados métodos mais formais e gerais para a construção de *parsers top-down*.

Parsing LL(1)

- Linguagens livre de contexto são reconhecidas por autômatos de pilha (*pushdown automata*).
- Um *parser* LL(1) é uma implementação (simplificada) desse tipo de autômato.



- *Parsing* LL(1) usa uma pilha para computação ao invés de chamadas recursivas de procedimentos.
- Estado atual do *parser* é bem mais simples de visualizar.
- Considere uma gramática de parênteses balanceados

$$S \rightarrow (S)S \mid \epsilon \quad .$$

A tabela a seguir mostra as ações do *parser* para a entrada (). Símbolo $ representa EOF.

	Parsing stack	Input	Action
1	$\$ S$	$()\$$	$S \rightarrow (S)S$
2	$\$ S) S ($	$()\$$	match
3	$\$ S) S$	$)\$$	$S \rightarrow \varepsilon$
4	$\$ S)$	$)\$$	match
5	$\$ S$	$\$$	$S \rightarrow \varepsilon$
6	$\$$	$\$$	accept

Ações são determinadas por uma tabela de *parsing*.

Parsing LL(1)

- Um *parser* LL(1) começa empilhando o símbolo inicial.
- A entrada é aceita se, após uma série de ações, a pilha e a sequência de *tokens* de entrada ficam vazias.
- Algoritmo usa duas ações:
  - *Generate*: usando uma regra $A \rightarrow \alpha$, substitui o não-terminal A no topo da pilha por α (invertido).
  - *Match*: casa um *token* no topo da pilha com o próximo *token* da entrada.
- Ações de *generate* correspondem exatamente aos passos de uma derivação mais à esquerda da entrada.
- Característica fundamental do *parsing top-down*.

Tabela de *Parsing* LL(1)

- Tabela de *Parsing* LL(1): expressa as possíveis escolhas de regras para um não-terminal A que esteja no topo da pilha, considerando o *look-ahead*.
- Essa tabela é uma matriz $M[N, T]$, aonde N é o conjunto de não-terminais da gramática e T é o conjunto de terminais (incluindo \$).
- *Exemplo*: para a gramática

$$S \rightarrow (S)S \mid \epsilon$$

temos a tabela abaixo

$M[N, T]$	(	)	\$
S	$S \rightarrow (S) S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Tabela de *Parsing* LL(1)

Regras para o preenchimento da tabela $M[N, T]$:

1. Dado uma regra $A \rightarrow \alpha$ tal que existe uma derivação $\alpha \Rightarrow^* a\beta$, aonde a é um *token*, adicione $A \rightarrow \alpha$ em $M[A, a]$.
2. Dado uma regra $A \rightarrow \alpha$ tal que existem derivações $\alpha \Rightarrow^* \epsilon$ e $S\$ \Rightarrow^* \beta A a \gamma$, adicione $A \rightarrow \alpha$ em $M[A, a]$.

- Infelizmente as regras acima são difíceis de implementar diretamente: algoritmos usam os conjuntos FIRST e FOLLOW.
- Processo de construção da tabela anterior:
  - Regra 1: produção $S \rightarrow (S)S$ adicionada em $M[S, (]$.
  - Regra 2: produção $S \rightarrow \epsilon$ adicionada em $M[S,)]$ pois $S\$ \Rightarrow (S)S\$$.
  - Regra 2: produção $S \rightarrow \epsilon$ adicionada em $M[S, \$]$ pois $S\$ \Rightarrow^* S\$$.

Gramática LL(1)

Gramática LL(1) – Definição 1

Uma gramática é uma gramática LL(1) se a tabela de *parsing* LL(1) associada possuir no máximo uma produção em cada célula.

- Consequência imediata da definição: uma gramática LL(1) não pode ser ambígua.
- Construção da tabela provê um método algorítmico para a detecção de ambiguidade em uma gramática.
- Isso não contradiz o resultado de indecidibilidade do caso geral: algoritmo só funciona para gramáticas LL(1).

Algoritmo de *Parsing* LL(1)

```
(* assumes $ marks the bottom of the stack and the end of the input *)
push the start symbol onto the top of the parsing stack ;
while the top of the parsing stack ≠ $ and the next input token ≠ $ do
  if the top of the parsing stack is terminal a
       and the next input token = a
  then (* match *)
    pop the parsing stack ;
    advance the input ;
  else if the top of the parsing is nonterminal A
       and the next input token is terminal a
       and parsing table entry M[A, a] contains
            production A → X1X2 . . . Xn
  then (* generate *)
    pop the parsing stack ;
    for i := n downto 1 do
      push Xi onto the parsing stack ;
  else error ;
if the top of the parsing stack = $
       and the next input token = $
then accept
else error ;
```

Exemplo – *If-Statements*

$$
\begin{aligned}
&\textit{statement} \rightarrow \textit{if-stmt} \mid \texttt{other} \\
&\textit{if-stmt} \rightarrow \texttt{if}\ (\ \textit{exp}\ )\ \textit{statement}\ \textit{else-part} \\
&\textit{else-part} \rightarrow \texttt{else}\ \textit{statement} \mid \varepsilon \\
&\textit{exp} \rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

M[N, T]	**if**	**other**	**else**	**0**	1	**$**
statement	*statement* → *if-stmt*	*statement* → **other**				
if-stmt	*if-stmt* → **if** (*exp*) *statement* *else-part*					
else-part			*else-part* → **else** *statement* *else-part* → ε			*else-part* → ε
exp				*exp* → **0**	*exp* → **1**	

Exemplo – *If-Statements*

- A célula *M*(*else-part*, `else`) contém duas regras: ambiguidade do *else* pendente.
- Gramática não é LL(1) mas pode-se usar algum critério de remoção de ambiguidade.
- Por exemplo, priorizar a primeira regra sobre a segunda.
- Feito isso, a gramática pode ser usada para *parsing* como se fosse uma gramática LL(1).

Dificuldades do *Parser* LL(1)

- Como no caso do *parser* de descida recursiva, um *parser* LL(1) não consegue lidar com recursão à esquerda.
- No primeiro caso: reescrever a gramática usando EBNF.
- No segundo caso a solução é similar: é necessário reescrever a gramática (usando somente BNF) para uma forma que o algoritmo consiga tratar.
- Duas técnicas comuns:
  - Remoção de recursão à esquerda (*left recursion removal*).
  - Fatoração à esquerda (*left factoring*).
- Ambas as técnicas podem ser automatizadas mas não garantem que uma gramática LL(1) sempre vai ser obtida.

Remoção de Recursão à Esquerda

- Visto anteriormente: recursão à esquerda é comumente utilizada para tornar as operadores associativos à esquerda.
- *Exemplo*: regras abaixo indicam que soma e subtração são associativos à esquerda.

$$exp \rightarrow exp + term \mid exp - term \mid term$$

- Exemplo acima é um caso de recursão imediata à esquerda: a recursão ocorre somente dentro de uma regra.
- Recursão à esquerda indireta: caso mais complexo – recursão envolvendo duas ou mais regras, como no exemplo abaixo.

$$\begin{aligned} A &\rightarrow B\,b \mid \ldots \\ B &\rightarrow A\,a \mid \ldots \end{aligned}$$

- Existe um algoritmo geral que lida com ambos os casos (detalhes no livro do Louden).

Remoção de Recursão à Esquerda

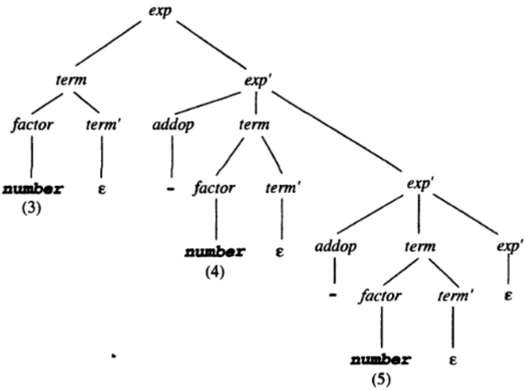
- Remoção de recursão à esquerda não modifica a linguagem, mas altera a gramática, e por consequência a *parse tree*.
- Isso causa uma complicação para o *parser* pois os operadores ainda precisam ser associativos à esquerda.
- Isso é ilustrado pelo exemplo a seguir, com a gramática original do lado esquerdo e a modificada do lado direito.

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → *****
factor → **(** *exp* **)** | **number**

exp → *term exp'*
exp' → *addop term exp'* | ε
addop → **+** | **-**
term → *factor term'*
term' → *mulop factor term'* | ε
mulop → *****
factor → **(** *exp* **)** | **number**

Remoção de Recursão à Esquerda

Parse tree para a expressão `3-4-5`:



- Árvore não exibe mais a associatividade à esquerda.
- No entanto, o *parser* ainda deve construir uma AST correta (i.e., com a associatividade à esquerda preservada).

Fatoração à Esquerda

- Fatoração à esquerda é necessária quando duas ou mais regras possuem um mesmo prefixo, como por exemplo:

$$A \to \alpha\beta \mid \alpha\gamma \quad .$$

- Um *parser* LL(1) não consegue distinguir tais regras.
- A solução é fatorar o prefixo comum α e reescrever as regras como

$$A \to \alpha A' \qquad A' \to \beta \mid \gamma \quad .$$

- Assim como no caso de remoção de recursão à esquerda, existe um algoritmo para fatoração à esquerda (detalhes no livro do Louden).
- Processo também modifica a gramática e complica a implementação do *parser*.

Parsing LL(1) e Construção da AST

- É mais difícil incluir a construção da AST no *parser* LL(1) do que no *parser* de descida recursiva.
- **Motivo**: a estrutura da AST pode ser obscurecida pela remoção de recursão à esquerda e pela fatoração à esquerda.
- A pilha de *parsing* armazena somente estruturas previstas pela gramática, não estruturas de fato vistas.
- **Solução**: atrasar a construção dos nós da AST até a hora em que algo é desempilhado da pilha de *parsing*.
- Uma pilha extra é usada para armazenar nós da AST, e marcadores de ação são colocados na pilha de *parsing* para indicar quando e quais ações devem ser realizadas na pilha da árvore.

Parsing LL(1) e Construção da AST

- Suponha uma gramática simplificada somente com adição:

$$E \rightarrow E \ + \ \mathtt{n} \mid \mathtt{n} \quad .$$

- Gramática LL(1) equivalente após remoção da recursão à esquerda:

$$E \rightarrow \mathtt{n}E' \qquad E' \rightarrow \ + \ \mathtt{n}E' \mid \epsilon \quad .$$

- Para computar o valor de uma expressão usamos uma pilha extra de valores para armazenar os resultados intermediários da computação.
- Duas operações podem ser realizadas na pilha de valores
  - **push**: realizada pelo procedimento de *match*.
  - **addstack**: escalonada na pilha de *parsing* através do símbolo especial $\#$.
- Operação **addstack** requer uma modificação na gramática:

$$E' \rightarrow \ + \ \mathtt{n}\#E' \mid \epsilon \quad .$$

Exemplo para a expressão 3+4+5.

Parsing stack	Input	Action	Value stack
$ E	3 + 4 + 5 $	$E \to \mathbf{n}\, E'$	$
$ E' **n**	3 + 4 + 5 $	match/push	$
$ E'	+ 4 + 5 $	$E' \to + \mathbf{n}\, \#\, E'$	3 $
$ E' # **n** +	+ 4 + 5 $	match	3 $
$ E' # **n**	4 + 5 $	match/push	3 $
$ E' #	+ 5 $	addstack	4 3 $
$ E'	+ 5 $	$E' \to + \mathbf{n}\, \#\, E'$	7 $
$ E' # **n** +	+ 5 $	match	7 $
$ E' # **n**	5 $	match/push	7 $
$ E' #	$	addstack	5 7 $
$ E'	$	$E' \to \varepsilon$	12 $
$	$	accept	12 $

É possível utilizar um método semelhante para a construção da AST da expressão.

Conjuntos FIRST e FOLLOW

- Para completar o algoritmo de *parsing* LL(1) é necessário construir a tabela $M[N, T]$. Essa construção depende da computação dos conjuntos FIRST, FOLLOW e FIRST+.
- FIRST : $(N \cup T) \rightarrow 2^T$ – função que mapeia formas sentenciais em conjuntos de terminais.
  - FIRST(β) indica os terminais iniciais com os quais uma derivação de β pode começar.
  - Aplicada principalmente sobre não-terminais para decidir qual regra utilizar.
- FOLLOW : $N \rightarrow 2^T$ – função que mapeia não-terminais em conjuntos de terminais.
  - FOLLOW(A) indica os terminais que podem seguir A em uma forma sentencial derivável.
- FIRST+ : Regra $\rightarrow 2^T$ – função que mapeia regras da gramática em conjuntos de terminais.
  - FIRST+($A \rightarrow \beta$) indica os terminais iniciais de derivações de A que começam com a essa regra.

Computando o Conjunto FIRST

- Conjunto FIRST pode ser computado por um algoritmo de ponto-fixo.
- Complexidade do algoritmo é dependente somente do tamanho dos conjuntos N e T. Na prática o algoritmo converge após poucas iterações.
- Ideia geral do algoritmo:
  1. Inicialmente mapear todos os elementos de N para conjuntos vazios.
  2. Para cada regra $A \rightarrow a\beta$, inclua a no conjunto FIRST(A). Se o corpo da regra é somente ϵ, inclua-o também.
  3. Para cada regra $A \rightarrow B\gamma$, inclua em FIRST(A) o conjunto de FIRST(B) computado na iteração anterior.
  4. Repita os passos 2 e 3 até que todos os conjuntos FIRST fiquem estáveis.

Considere a gramática abaixo.

$$A \to BC \mid a$$
$$B \to Cb \mid \epsilon$$
$$C \to c \mid \epsilon$$

A computação do conjunto FIRST para os não-terminais da gramática é dada pela seguinte tabela.

	Iteração				
N	0	1	2	3	4
A	$\emptyset$	a	$a\ c\ \epsilon$	$a\ c\ \epsilon\ b$	$a\ c\ \epsilon\ b$
B	$\emptyset$	ϵ	$\epsilon\ c\ b$	$\epsilon\ c\ b$	$\epsilon\ c\ b$
C	$\emptyset$	$c\ \epsilon$	$c\ \epsilon$	$c\ \epsilon$	$c\ \epsilon$

Obs.: para todo $a \in T$, FIRST(a) = $\{a\}$.

Computando o Conjunto FOLLOW

- Conjunto FOLLOW também é computado por um algoritmo de ponto-fixo.
- Valem as mesmas considerações de complexidade e eficiência do conjunto FIRST.
- FOLLOW é uma função de não-terminais $A \in N$ para conjuntos de terminais que podem seguir A.
- Para o cálculo desses conjuntos, busca-se as ocorrências de um não-terminal no corpo das outras regras.
- Ideia geral do algoritmo:
  1. Inicialize FOLLOW com $\{\$\}$ para o símbolo inicial e $\emptyset$ para os demais não-terminais. (\$ $=$ EOF).
  2. Para cada regra $A \rightarrow \alpha B$, copie FOLLOW(A) para FOLLOW(B).
  3. Para cada regra $A \rightarrow \alpha B C \gamma$, inclua FIRST($C$) em FOLLOW($B$).
  4. Repita os passos 2 e 3 até que todos os conjuntos FOLLOW fiquem estáveis.

Computando o Conjunto FOLLOW – Exemplo

Considere a mesma gramática anterior.

$$A \rightarrow BC \mid a$$
$$B \rightarrow Cb \mid \epsilon$$
$$C \rightarrow c \mid \epsilon$$

A computação do conjunto FOLLOW é dada pela seguinte tabela.

	Iteração		
N	0	1	2
A	\$	\$	\$
B	$\emptyset$	\$ c	\$ c
C	$\emptyset$	\$ b	\$ b

Obs.: ϵ nunca é um elemento de FOLLOW.

Conjuntos FIRST e FOLLOW e FIRST+

- Dada uma regra $A \to B\alpha$, FIRST(A) contém FIRST(B), com a possível exceção de ϵ.
- Para a regra $A \to \alpha B$, FOLLOW(B) contém FOLLOW(A).
- Conjuntos FIRST trabalham "sobre a esquerda" do corpo de uma regra, conjuntos FOLLOW "sobre a direita".

- FIRST+ mapeia regras para conjuntos de terminais.
- Para toda regra $A \to \beta$ na gramática:
  1. Se $\epsilon \notin$ FIRST(β), então FIRST+($A \to \beta$) = FIRST(β).
  2. Caso contrário, FIRST+($A \to \beta$) = FIRST(β) $\cup$ FOLLOW(A).

Gramática LL(1) – Definição 2

Uma gramática é LL(1) se, para qualquer não-terminal A com múltiplas regras $A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$,

$$\text{FIRST+}(A \to \beta_i) \cap \text{FIRST+}(A \to \beta_j) = \emptyset,\ \forall 1 \leq i, j \leq n, i \neq j.$$

Construindo a Tabela de *Parsing* LL(1)

Relembrando as regras para o preenchimento da tabela:

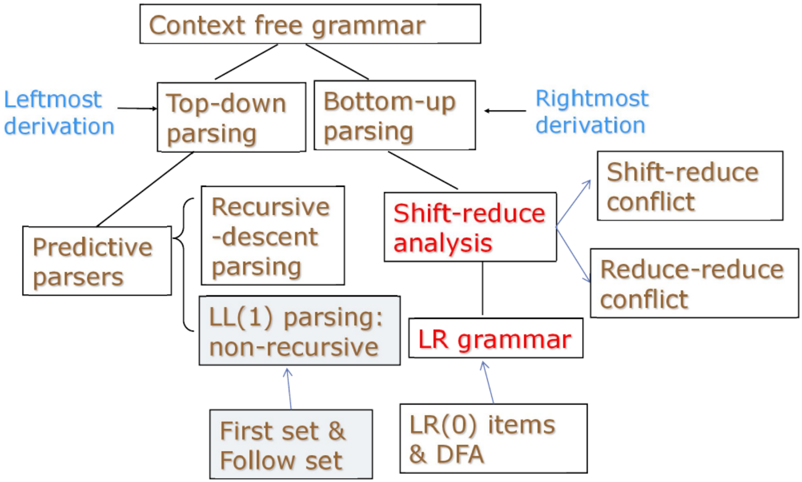
1. Dado uma regra $A \to \alpha$ tal que existe uma derivação $\alpha \Rightarrow^* a\beta$, aonde a é um *token*, adicione $A \to \alpha$ em $M[A, a]$.
2. Dado uma regra $A \to \alpha$ tal que existem derivações $\alpha \Rightarrow^* \epsilon$ e $S\$ \Rightarrow^* \beta A a \gamma$, adicione $A \to \alpha$ em $M[A, a]$.

- Claramente, o *token* a da regra 1 está em FIRST(α) e o a da regra 2 está em FOLLOW(A).
- Assim, obtemos o seguinte algoritmo para a construção da tabela LL(1): para cada regra $A \to \alpha$ e para cada terminal a em FIRST+($A \to \alpha$), adicione $A \to \alpha$ em $M[A, a]$.
- Note que, se a gramática é LL(1), cada célula da tabela possui no máximo uma regra. (Consequência imediata da segunda definição de gramática LL(1).)

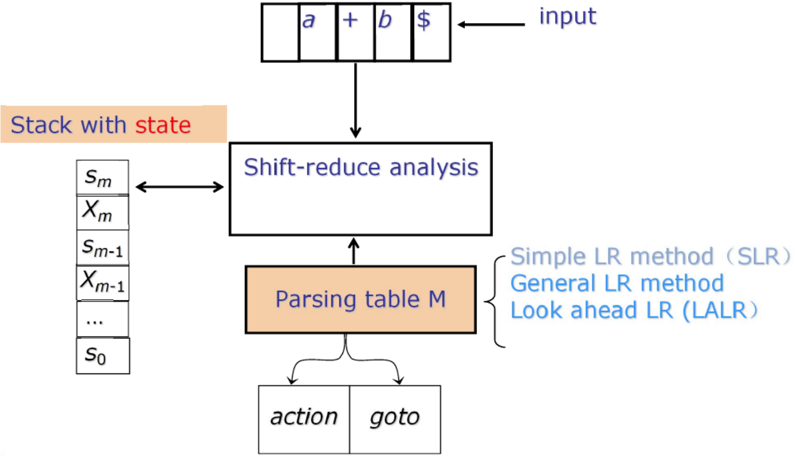
Parte III

Bottom-Up Parsing

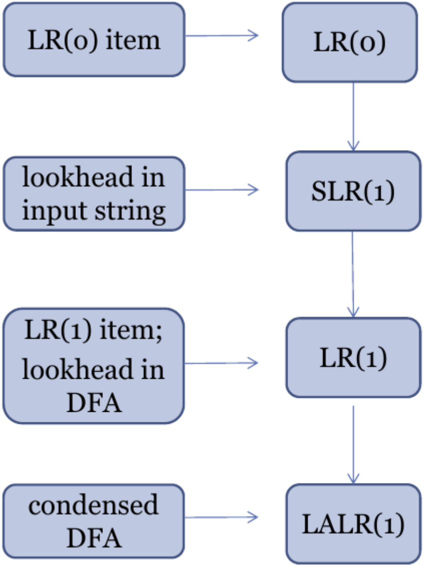
Visão Geral do Processo de *Parsing*



Visão Geral do *Parsing Bottom-Up*

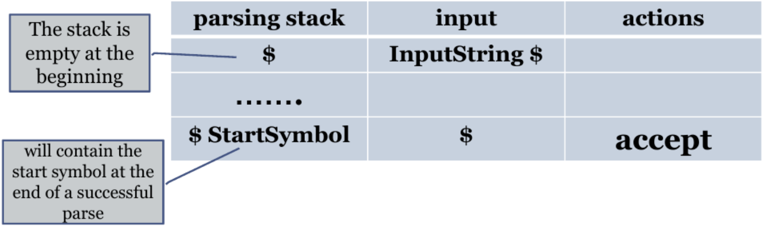


Diferentes Algoritmos de *Parsing Bottom-Up*



Visão Geral do *Parsing Bottom-Up*

- Um *parser bottom-up* usa uma pilha para realizar a análise sintática.
- A pilha de *parsing* pode conter *tokens*, não-terminais e também alguma informação extra sobre o estado do *parser*.
- A figura abaixo mostra um visão esquemática da organização do *parser*.



parsing stack	input	actions
$	InputString $	
…….		
$ StartSymbol	$	accept

Visão Geral do *Parsing Bottom-Up*

- Um *parser bottom-up* pode realizar duas outras ações além de *accept*: *shift* e *reduce*.
- ***Shift***: desloca um terminal da entrada para o topo da pilha.
- ***Reduce***: reduz uma forma sentencial α no topo da pilha para um não-terminal A, segundo uma regra $A \rightarrow \alpha$ da gramática.
- Por conta das ações que ele realiza, um *parser bottom-up* também é chamado de *parser shift-reduce*.
- Para poder realizar o *parsing bottom-up*, a gramática deve ser aumentada com um novo símbolo inicial S' e uma regra $S' \rightarrow S$. (Justificativa será dada adiante.)

Visão Geral do *Parsing Bottom-Up* – Exemplo 1

Considere a gramática aumentada de parênteses balanceados.

$$S' \rightarrow S$$
$$S \rightarrow (\ S\ )\ \ S \mid \varepsilon$$

O *parse bottom-up* da entrada `()` é dado a seguir.

	Parsing stack	Input	Action
1	\$	( ) \$	shift
2	\$ (	) \$	reduce $S \rightarrow \varepsilon$
3	\$ (S	) \$	shift
4	\$ (S)	\$	reduce $S \rightarrow \varepsilon$
5	\$ (S) S	\$	reduce $S \rightarrow (\ S\ )\ S$
6	\$ S	\$	reduce $S' \rightarrow S$
7	\$ S'	\$	accept

Visão Geral do *Parsing Bottom-Up* – Exemplo 2

Considere a gramática aumentada para expressões de adição.

$$E' \rightarrow E$$
$$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$$

O *parse bottom-up* da entrada `n+n` é dado a seguir.

	Parsing stack	Input	Action
1	$\$$	$\mathbf{n} + \mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+ \mathbf{n}\,\$$	reduce $E \rightarrow \mathbf{n}$
3	$\$\,E$	$+ \mathbf{n}\,\$$	shift
4	$\$\,E +$	$\mathbf{n}\,\$$	shift
5	$\$\,E + \mathbf{n}$	$\$$	reduce $E \rightarrow E + \mathbf{n}$
6	$\$\,E$	$\$$	reduce $E' \rightarrow E$
7	$\$\,E'$	$\$$	accept

Visão Geral do *Parsing Bottom-Up* – Observações

- Um *parser shift-reduce* acompanha uma derivação mais à direita da entrada, mas os passos da derivação ocorrem na ordem inversa.
- No Exemplo 1 há quatro reduções, correspondendo a seguinte derivação mais à direita:

$$S' \Rightarrow S \Rightarrow (\ S\ )\ S \Rightarrow (\ S\ ) \Rightarrow (\ ) \quad .$$

- No Exemplo 2 a derivação correspondente é

$$E' \Rightarrow E \Rightarrow E\ \mathtt{+}\ \mathtt{n} \Rightarrow \mathtt{n}\ \mathtt{+}\ \mathtt{n} \quad .$$

- A cada passo da derivação, a forma sentencial está dividida entre a pilha e a entrada.

Visão Geral do *Parsing Bottom-Up* – Observações

- *Parser* realiza *shifts* até que seja possível realizar um *reduce*.
- Isso ocorre quando a sequência de símbolos a partir do topo da pilha corresponde ao corpo de uma regra da gramática.
- Um *handle* da forma sentencial é formado por:
  - Uma sequência de símbolos (*string*) na pilha.
  - A posição na forma sentencial aonde a sequência ocorre.
  - A regra que deve ser usada na redução.
- Se uma gramática é livre de ambiguidades, então existe somente uma única derivação mais à direita, e portanto os *handles* são únicos.
- Determinar o próximo *handle* é a tarefa principal de um *parser shift-reduce*.

Visão Geral do *Parsing Bottom-Up* – Observações

- A *string* de um *handle* corresponde ao corpo de uma regra e o último símbolo deste corpo está no topo da pilha.
- No entanto, para ser de fato um *handle*, não basta que a *string* no topo da pilha case com o corpo de uma regra.
- De fato, se uma produção-ϵ faz parte da gramática (como no Exemplo 1), então o seu corpo (a *string* vazia) sempre está no topo da pilha.
- Assim, temos uma restrição sobre as reduções: elas só podem ocorrem quando a *string* resultante de fato corresponde a uma forma sentencial válida na derivação.
- *Exemplo*: no passo 3 da tabela do Exemplo 1, uma redução por $S \to \epsilon$ poderia ser feita, mas a *string* resultante (S S) não é uma forma sentencial válida.
- $\Rightarrow$ Os algoritmos de *parsing bottom-up* devem computar os *handles* de forma a garantir que eles só serão reduzidos nos momentos adequados.

Itens LR(0)

- Um item LR(0) de uma CFG é uma regra com uma posição no corpo marcada por um ponto (.).
- Recebem esse nome porque não fazem referência ao *look-ahead* (em contraste com itens LR(1), que o fazem).
- Para uma regra qualquer $A \rightarrow \beta_1 \dots \beta_n$, o marcador pode aparecer antes ou depois de qualquer β_i.
- *Exemplo*: se $A \rightarrow \alpha\beta$ é uma regra, então $A \rightarrow \alpha \,.\, \beta$ é um dos itens LR(0) da gramática.
- Itens indicam um passo intermediário no reconhecimento do corpo de uma regra.
- *Exemplo*: o item $A \rightarrow \alpha \,.\, \beta$ indica que α já foi visto (está no topo da pilha) e que é possível derivar os próximos *tokens* de entrada a partir de β.

A gramática

$$S' \to S$$
$$S \to (\,S\,)\;S \mid \varepsilon$$

possui os seguintes oito itens LR(0)

$$S' \to .S$$
$$S' \to S.$$
$$S \to .(S)S$$
$$S \to (.S)S$$
$$S \to (S.)S$$
$$S \to (S).S$$
$$S \to (S)S.$$
$$S \to .$$

A gramática

$$E' \to E$$
$$E \to E + \mathbf{n} \mid \mathbf{n}$$

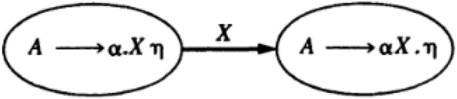
possui os seguintes oito itens LR(0)

$$E' \to .E$$
$$E' \to E.$$
$$E \to .E + \mathbf{n}$$
$$E \to E. + \mathbf{n}$$
$$E \to E + .\mathbf{n}$$
$$E \to E + \mathbf{n}.$$
$$E \to .\mathbf{n}$$
$$E \to \mathbf{n}.$$

Autômato Finito de Itens

- Itens LR(0) podem ser usados como estados de um autômato finito (FA).
- Esse FA mantém informação sobre a pilha de *parsing* e o progresso das operações de *shift-reduce*.
- Inicialmente vamos usar um autômato finito não-determinístico (NFA).
- O autômato determinístico (DFA) equivalente pode ser obtido a partir do NFA usando-se o algoritmo clássico de construção de subconjuntos.
- Transições no FA são rotuladas por um símbolo $X \in (N \cup T)$.

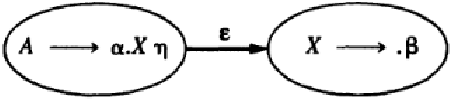
- *Exemplo*: dado um item (estado) $A \to \alpha\, .\, X\eta$, temos uma transição sobre o símbolo X para o estado $A \to \alpha X\, .\, \eta$. Graficamente:



- Se X é um *token*, a transição corresponde a um *shift* de X da entrada para o topo da pilha.
- Se X é um não-terminal, a interpretação da transição é um pouco mais complexa, já que X não aparece na entrada.

Autômato Finito de Itens

- Essa transição ainda corresponde ao empilhamento de X durante o processo de *parsing*.
- Tal empilhamento só pode acontecer com uma redução por uma regra da forma $X \to \beta$.
- Uma redução por $X \to \beta$ deve ser precedida pelo reconhecimento de β.
- O item $X \to .\beta$ representa o início desse processo.
- Assim, devemos adicionar uma transição como



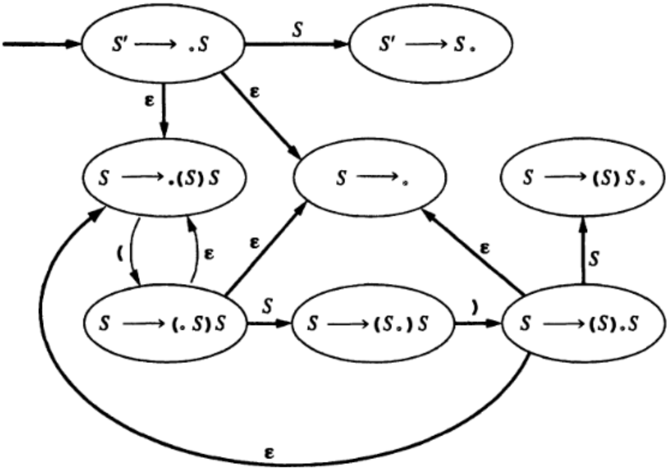
para todas as regras aonde X é a cabeça.

Autômato Finito de Itens

- O estado inicial do NFA deve corresponder ao estado inicial do *parser*: pilha vazia e pronto para reconhecer S (símbolo inicial).
- Qualquer item $S \rightarrow .\alpha$ poderia servir como estado inicial, mas a gramática pode ter várias regras aonde S é a cabeça.
- Isso justifica o uso de gramáticas aumentadas: item $S' \rightarrow .S$ garante a unicidade do estado inicial.
- O NFA não tem estados finais pois o seu propósito é acompanhar o estado do *parser* e não reconhecer *strings*.
- O próprio *parser* é que deve decidir quando aceitar a entrada.

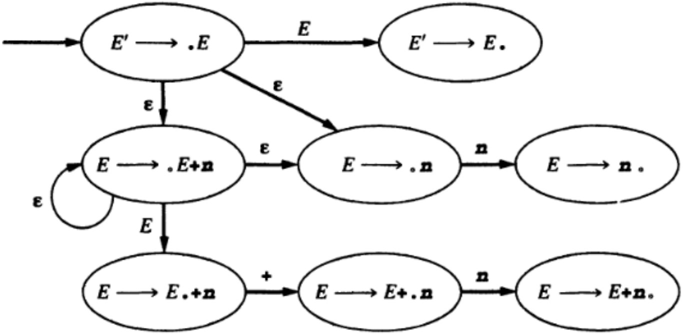
Autômato Finito de Itens – Exemplos

A gramática $\begin{array}{l} S' \rightarrow S \\ S \rightarrow (\ S\ )\ \ S \mid \varepsilon \end{array}$ possui o seguinte NFA:



Autômato Finito de Itens – Exemplos

A gramática $\begin{array}{l} E' \to E \\ E \to E + \mathbf{n} \mid \mathbf{n} \end{array}$ possui o seguinte NFA:



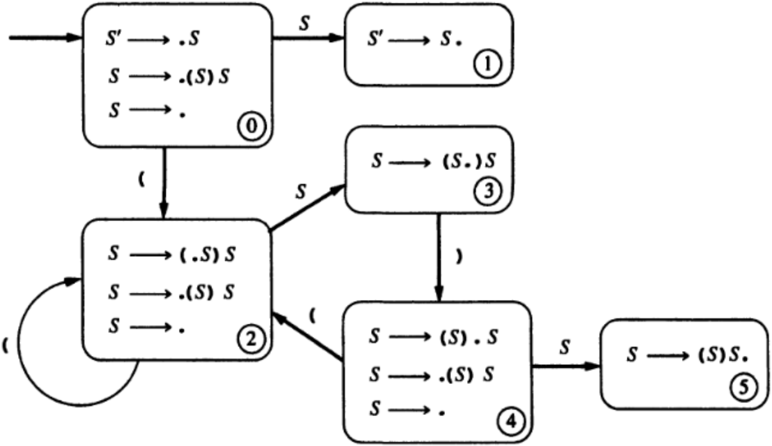
O item $E \to .E \;\; + \;\; \mathtt{n}$ tem uma transição-ϵ para ele próprio. Isso ocorre em todas as CFGs com recursão à esquerda.

Autômato Finito de Itens

- O NFA de itens LR(0) pode ser facilmente construído de forma algorítmica a partir das regras da gramática.
- No entanto, um NFA não é adequado para acompanhar os estados de *parsing* pois o *parser* deve ser um programa determinístico.
- Assim, usa-se o algoritmo clássico de construção de subconjuntos para se obter um DFA equivalente ao NFA original.
- Por questões de eficiência, nas ferramentas geradoras de *parsers bottom-up* esses dois passos são unificados em um só.
- *Exemplo*: o Bison computa diretamente o DFA a partir das regras da gramática.

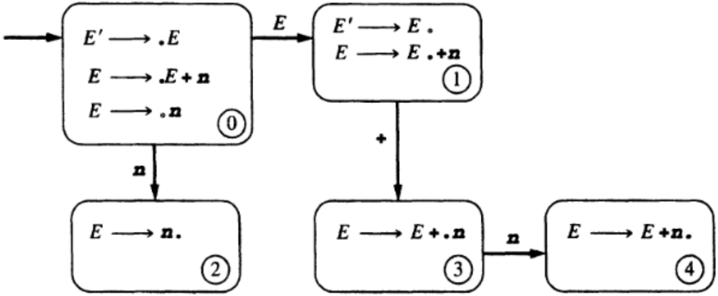
Autômato Finito de Itens – Exemplos

A gramática $\begin{array}{l} S' \to S \\ S \to (\ S\ )\ \ S \mid \varepsilon \end{array}$ possui o seguinte DFA:



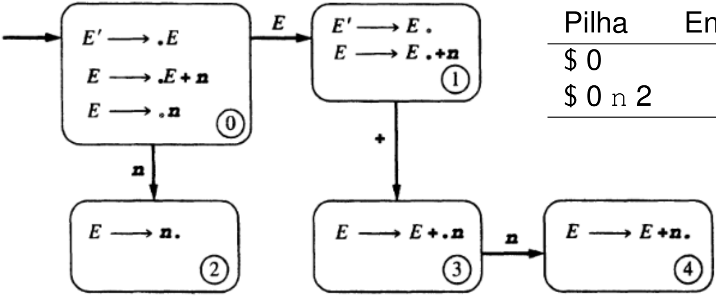
Autômato Finito de Itens – Exemplos

A gramática $\begin{array}{l} E' \to E \\ E \to E + \mathbf{n} \mid \mathbf{n} \end{array}$ possui o seguinte DFA:



Algoritmo de *Parsing* LR(0)

- O algoritmo precisa rastrear o estado atual no DFA de itens.
- Modificamos a pilha de *parsing* para armazenar os números dos estados, além dos demais símbolos.
- Empilhamos o número do novo estado após empilhar cada símbolo.



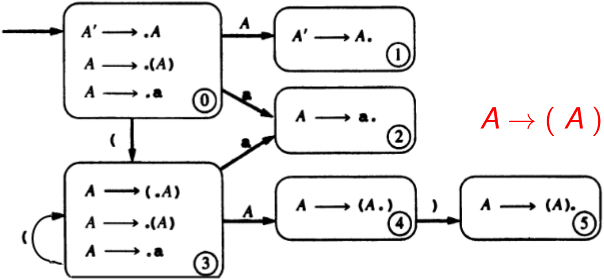
Pilha	Entrada
$ 0	n α $
$ 0 n 2	α $

Algoritmo de *Parsing* LR(0)

Seja s o estado atual no topo da pilha de *parsing*:

1. Se o estado s contém um item da forma $A \to \alpha\ .\ X\beta$, onde X é um terminal:
   1. Se o próximo *token* da entrada é X: faça *shift* de X e empilhe o novo estado que contém o item $A \to \alpha\ X\ .\ \beta$.
   2. Se o próximo *token* da entrada não é X: erro de sintaxe.
2. Se o estado s contém um item completo (da forma $A \to \gamma\ .$), faça *reduce* por essa regra e:
   1. Se S' (o símbolo inicial) foi reduzido e a entrada acabou: *accept*.
   2. Se S' foi reduzido e a entrada não acabou: erro de sintaxe.
   3. Caso contrário:
      1. Desempilhe γ, revelando o estado antigo s_γ onde a construção de γ começou.
      2. Tome a transição $s_\gamma \xrightarrow{A} s'$, empilhando A e s'.

Algoritmo de *Parsing* LR(0) – Exemplo



$A \rightarrow (\ A\ )\ |\ \mathtt{a}$

	Parsing stack	Input	Action
1	$ 0	((a)) $	shift
2	$ 0 (3	(a)) $	shift
3	$ 0 (3 (3	a)) $	shift
4	$ 0 (3 (3 a 2	)) $	reduce $A \rightarrow$ a
5	$ 0 (3 (3 *A* 4	)) $	shift
6	$ 0 (3 (3 *A* 4) 5	) $	reduce $A \rightarrow (\ A\ )$
7	$ 0 (3 *A* 4	) $	shift
8	$ 0 (3 *A* 4) 5	$	reduce $A \rightarrow (\ A\ )$
9	$ 0 *A* 1	$	accept

Algoritmo de *Parsing* LR(0)

- O DFA de itens LR(0) e as ações especificadas pelo algoritmo de *parsing* LR(0) podem ser combinadas em uma tabela de *parsing*.
- Isso torna o método controlado por tabela, como outros já vistos anteriormente.
- *Exemplo*:

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	$A' \rightarrow A$				
2	reduce	$A \rightarrow \mathbf{a}$				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \rightarrow (A)$				

Próximo estado:
terminais – colunas *Input*, não-terminais – coluna *Goto*.

- Suponha um estado do DFA que contém os seguintes itens, onde X é um terminal.

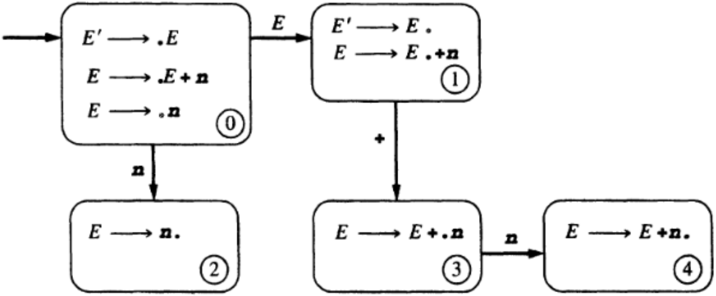
$$(1)\ A \rightarrow \alpha\ . \qquad\qquad (2)\ A \rightarrow \alpha\ .\ X\beta$$

- Seguindo o algoritmo de *parsing* LR(0), o item (2) nos diz para realizar uma ação de *shift*.
- Por outro lado, o item (1) indica uma ação de *reduce*.
- Essa indefinição sobre qual ação realizar caracteriza um conflito de *shift-reduce*.
- De forma similar, os dois itens abaixo em um mesmo estado do DFA, ilustram um conflito de *reduce-reduce*, pois não é possível decidir sobre qual regra usar.

$$(3)\ A \rightarrow \alpha\ . \qquad\qquad (4)\ B \rightarrow \alpha\ .$$

Gramática LR(0)

- Gramática LR(0): gramática livre de ambiguidades que causam conflitos de *shift-reduce* ou *reduce-reduce*.
- Na prática: um estado com um item completo não pode conter outros itens.
- A gramática de somas não é LR(0) pois o estado 1 contém um conflito de *shift-reduce*.



- Praticamente nenhuma gramática “real” é LR(0).

Parsing SLR(1)

- O *parsing* LR(1) simples, chamado SLR(1), usa o DFA de itens LR(0) como visto anteriormente.
- Além disto o algoritmo SLR(1) é mais poderoso que o LR(0) por usar um *look-ahead*.
- O *look-ahead* é combinado com o conjunto FOLLOW de um não-terminal para se decidir qual redução realizar.
- Essa modificação bastante simples no algoritmo de *parsing* permite o reconhecimento de uma quantidade maior de linguagens.
- O algoritmo modificado é apresentado a seguir, com as alterações indicadas em **negrito**.

Algoritmo de *Parsing* SLR(1)

Seja s o estado atual no topo da pilha de *parsing*:

1. Se o estado s contém um item da forma $A \rightarrow \alpha\ .\ X\beta$, onde X é um terminal:
    1. Se o próximo *token* da entrada é X: faça *shift* de X e empilhe o novo estado que contém o item $A \rightarrow \alpha\ X\ .\ \beta$.
    2. Se o próximo *token* da entrada não é X: erro de sintaxe.
2. Se o estado s contém um item completo $A \rightarrow \gamma\ .$ **e o próximo *token* da entrada está em FOLLOW(A)**, faça *reduce* e:
    1. Se S' foi reduzido e a entrada acabou: *accept*.
    2. Se S' foi reduzido e a entrada não acabou: erro de sintaxe.
    3. Caso contrário:
        1. Desempilhe γ, revelando o estado antigo s_γ onde a construção de γ começou.
        2. Tome a transição $s_\gamma \xrightarrow{A} s'$, empilhando A e s'.
3. **Se o próximo *token* da entrada não estiver em FOLLOW(A): erro de sintaxe.**

Gramática SLR(1)

- Uma gramática é SLR(1) se, e somente se, para qualquer estado *s*, as seguintes condições forem satisfeitas:
  1. Para qualquer item $A \to \alpha \,.\, X\beta$ em *s*, onde *X* é um terminal, não existe em *s* um item completo $B \to \alpha\,.$ com *X* em FOLLOW(*B*).
  2. Para dois itens completos $A \to \alpha\,.$ e $B \to \alpha\,.$ em *s*, FOLLOW(*A*) $\cap$ FOLLOW(*B*) $= \emptyset$.
- Violação da condição 1 é um conflito de *shift-reduce*.
- Gramática pode ser “reparada” em alguns casos estabelecendo que *shift* tem prioridade sobre *reduce*. Isso resolve o problema do *else* pendente.
- Violação da condição 2 é um conflito de *reduce-reduce*.
- Conflitos desse tipo em geral indicam um erro na gramática.
- A gramática de somas apresentada anteriormente não é LR(0) mas é SLR(1).

Limitações do *Parsing* SLR(1)

- Considere a gramática de comandos em Pascal abaixo e a sua versão simplificada à direita.

$$
\begin{aligned}
&stmt \rightarrow call\text{-}stmt \mid assign\text{-}stmt \\
&call\text{-}stmt \rightarrow \textbf{identifier} \\
&assign\text{-}stmt \rightarrow var \; \textbf{:=} \; exp \\
&var \rightarrow var \; \textbf{[} \; exp \; \textbf{]} \mid \textbf{identifier} \\
&exp \rightarrow var \mid \textbf{number}
\end{aligned}
\qquad\qquad
\begin{aligned}
&S \rightarrow \textbf{id} \mid V \; \textbf{:=} \; E \\
&V \rightarrow \textbf{id} \\
&E \rightarrow V \mid \textbf{n}
\end{aligned}
$$

- Não é possível reconhecer a linguagem dessa gramática usando *parsing* SLR(1). (A gramática não é SLR(1).)
- O DFA de itens possui um estado contendo os itens $S \rightarrow \texttt{id}\,.$ e $V \rightarrow \texttt{id}\,.$ e temos que FOLLOW(S) $= \{\$\}$ e FOLLOW(V) $= \{:=, \$\}$. $\Rightarrow$ Conflito de *reduce-reduce*.
- Isso é um problema pois esse tipo de construção é comum em linguagens de programação.
- $\Rightarrow$ Precisamos de um método de *parsing* LR mais poderoso.

O algoritmo de *parse* SLR(1):

- Constrói um DFA de itens LR(0).
- O *look-ahead* só é utilizado durante a execução do *parser*: operações de *reduce* testam se o *look-ahead* está no conjunto FOLLOW.
- Método não pode ser aplicado para dois itens LR(0) $A \to \alpha\,.$ e $B \to \alpha\,.$ com $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) \neq \emptyset$.

Parsing LR(1) Canônico

O algoritmo de *parse* LR(1) geral (canônico):

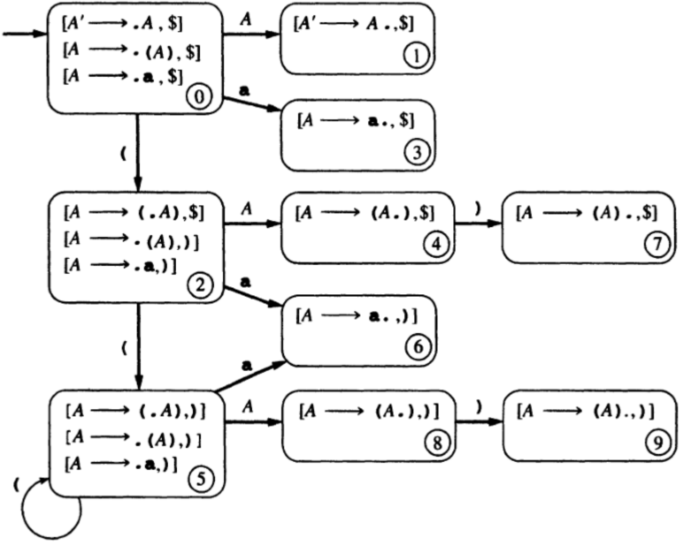
- Constrói um DFA de itens LR(1).
- Um item LR(1) tem a forma $[A \rightarrow \alpha\ .\ \beta, a]$, onde $A \rightarrow \alpha\ .\ \beta$ é um item LR(0) e a é um *token* de *look-ahead*.
- Para um item completo $[A \rightarrow \alpha\beta.\ , a]$: operações de *reduce* testam se o *look-ahead* é a.
- É essencial destacar que a é o *look-ahead* esperado quando o item estiver completo.
- Os *tokens* dos itens LR(1) só servem para distinguir entre as regras que estão "habilitadas" para redução. Estes *tokens* não influenciam as operações de *shift*.
- Embutir o *look-ahead* na construção do DFA torna o método mais poderoso pois permite relaxar a restrição do algoritmo SLR(1).

Parsing LR(1) Canônico

Construindo o NFA de itens LR(1):

- Similar à construção do NFA de itens LR(0).
- Requer uma gramática aumentada como anteriormente.
- Estados do NFA são conjuntos de itens LR(1). Estado inicial contém $[S' \to S, \$]$.
- Definição das transições:
  1. Temos que $[A \to \alpha \,.\, X\beta, a] \xrightarrow{X} [A \to \alpha X \,.\, \beta, a]$, para qualquer símbolo $X \in (N \cup T)$.
  2. Dado um item $[A \to \alpha\ .B\beta, a]$ onde B é um não-terminal, há transições-ϵ para itens $[B \to .\gamma, b]$, para toda regra $B \to \gamma$ e todo *token* b em $\text{FIRST}(\beta a)$.
- No caso 1, se X é um terminal, temos uma operação de *shift*. É essencial notar que não é necessário que $X = a$, pois a indica qual deve ser o *look-ahead* no momento da futura operação de redução.

DFA de itens LR(1) para a gramática $A \to (A) \mid \texttt{a}$.



Algoritmo de *Parsing* LR(1)

Seja s o estado atual no topo da pilha de *parsing*:

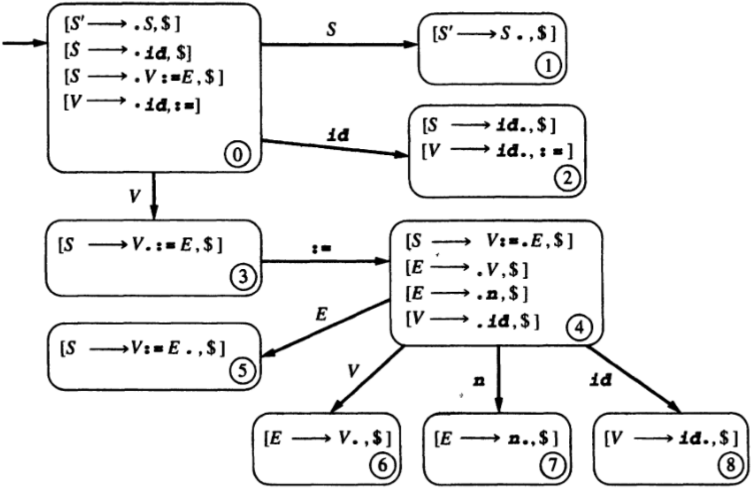
1. Se o estado s contém um item da forma $[A \to \alpha\ .\ X\beta, a]$, onde X é um terminal:
   1. Se o próximo *token* da entrada é X: faça *shift* de X e empilhe o novo estado que contém o item $[A \to \alpha\ X\ .\ \beta, a]$.
   2. Se o próximo *token* da entrada não é X: erro de sintaxe.
2. Se o estado s contém um item completo $[A \to \gamma.\ , a]$ **e o próximo *token* da entrada é** a, faça *reduce* e:
   1. Se S' foi reduzido e a entrada acabou: *accept*.
   2. Se S' foi reduzido e a entrada não acabou: erro de sintaxe.
   3. Caso contrário:
      1. Desempilhe γ, revelando o estado antigo s_γ onde a construção de γ começou.
      2. Tome a transição $s_\gamma \xrightarrow{A} s'$, empilhando A e s'.
3. **Se o próximo *token* da entrada não for a: erro de sintaxe.**

Gramática LR(1)

- Uma gramática é LR(1) se, e somente se, para qualquer estado *s*, as seguintes condições forem satisfeitas:
  1. Para qualquer item $[A \rightarrow \alpha\ .\ X\beta, a]$ em *s*, onde *X* é um terminal, não existe em *s* um item completo $[B \rightarrow \alpha\ .\ , X]$. (Caso contrário é um conflito de *shift-reduce*.)
  2. Não existem dois itens em *s* da forma $[A \rightarrow \alpha\ .\ , a]$ e $[B \rightarrow \alpha\ .\ , a]$. (Caso contrário é um conflito de *reduce-reduce*.)
- Próximo slide apresenta o DFA de itens LR(1) para a gramática que não é tratável pelo método SLR(1).
- Em geral, o DFA de itens LR(1) é bem maior (fator de 10) que o DFA de itens LR(0).
- O tamanho do DFA afeta diretamente o tamanho da tabela de *parsing*.
- Preocupações com eficiência levaram ao desenvolvimento do método LALR(1).

DFA de itens LR(1) para a gramática

$S \rightarrow \texttt{id} \mid V \texttt{:=} E$
$V \rightarrow \texttt{id}$
$E \rightarrow V \mid \texttt{n}$

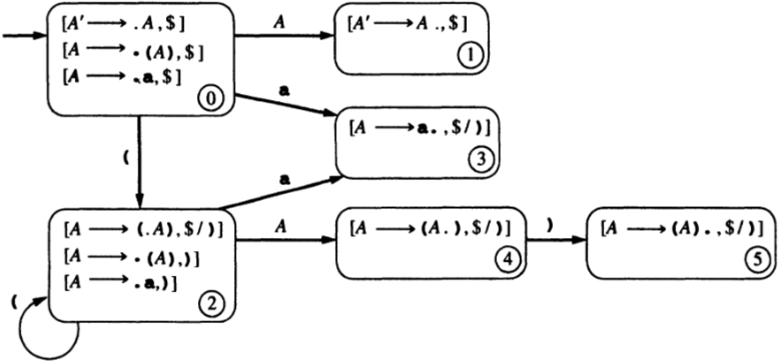


Parsing LALR(1)

- No DFA de itens LR(1), muitos estados distintos são diferenciados apenas pelo *token* de *look-ahead*, i.e., esses estados possuem o mesmo item LR(0).
- O algoritmo de *parse* LALR(1) identifica estes estados e combina os *look-aheads*.
- Assim, o DFA contém itens da forma $[A \rightarrow \alpha\ .\ \beta, a/b/c]$.
- *Parsing* LALR(1) possui praticamente o mesmo poder de reconhecimento que o LR(1) canônico e ao mesmo tempo utiliza um DFA menor.
- Por conta disso, LALR(1) é o método preferido para a implementação de *parsers bottom-up*.
- O Bison gera *parsers* LALR(1).

Parsing LALR(1)

DFA de itens LALR(1) para a gramática $A \rightarrow (A) \mid \texttt{a}$.



DFA de itens LR(1) tinha 10 estados, já o DFA acima tem o mesmo número de estados que o DFA de itens LR(0).

Parsing LALR(1)

- O algoritmo de *parsing* LALR(1) é o mesmo que o algoritmo de *parsing* LR(1) geral.
- A construção LALR(1) pode introduzir conflitos que não existem no método LR(1) geral, mas isso raramente acontece na prática.
- Se uma gramática é LR(1), o método LALR(1) não introduz conflitos de *shift-reduce* mas pode introduzir conflitos de *reduce-reduce*.
- Gramáticas LR(1) que não são LALR(1) geralmente não ocorrem em LPs. (Veja exemplo no exercício 5.2 do livro.)
- Vale destacar que não é necessário construir o DFA de itens LR(1) para depois condensá-lo em itens LALR(1).
- O DFA de itens LALR(1) pode ser construído diretamente a partir do DFA de itens LR(0) através de um processo chamado propagação de *look-aheads*.

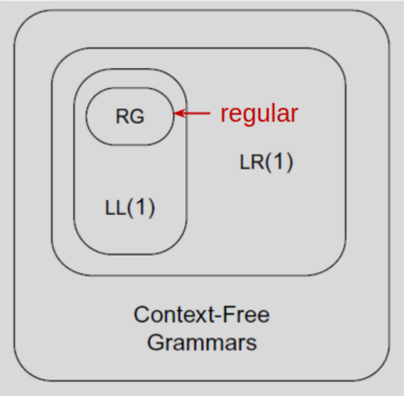
Sumário

Comparativo entre os métodos *bottom-up* e *top-down*:

	Método LR(1)	**Método LL(1)**
Gramática	sem restrições	sem recursão à esquerda
Tabela	estados $\times$ símbolos (grande)	terminais $\times$ não-terminais (pequena)
Pilha	símbolos e estados	somente símbolos

Sumário

- Poder de reconhecimento dos algoritmos de *parsing* LR:
$$LR(0) < SLR(1) < LALR(1) < LR(1) \quad .$$
- Poder de expressão das gramáticas:



Aula 02 – Análise Sintática (*Parsing*)

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**