

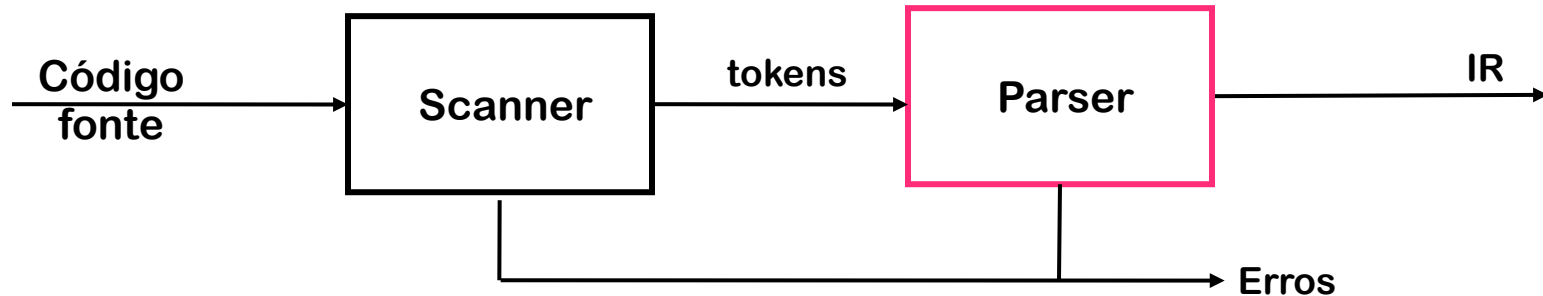
MAB 471
2011.2

Análise Sintática

<http://www.dcc.ufrj.br/~fabiom/comp>



O Front End



Parser

- Verifica a corretude gramatical da sequência de palavras e categorias sintáticas produzida pelo scanner
- Determina se a entrada está sintaticamente bem formada
- Guia a verificação em níveis mais profundos que a sintaxe
- Constrói uma representação IR do código



O Estudo de Análise Sintática

Processo de descobrir a derivação de uma sentença

- Modelo matemático da sintaxe — uma gramática G
- Algoritmo para testar pertinência em $L(G)$
- O objetivo é produzir parsers, não estudar a matemática de linguagens arbitrárias

Roteiro de Estudo

- 1 Gramáticas livres de contexto e derivações
- 2 Parsing top-down
 - Parsers LL(1) gerados e parsers recursivos escritos à mão
- 3 Parsing bottom-up
 - Parsers LR(1) gerados



Especificando sintaxe com uma gramática

Sintaxe livre de contexto é especificada com uma gramática livre de contexto (CFG)

Ovelha \rightarrow Ovelha bée
| bée

CFG que define o conjunto de sons de uma ovelha

Escrita em uma variante de BNF (forma de Backus-Naur)

Formalmente, uma gramática é uma quádrupla, $G = (S, N, T, P)$

- S é o símbolo inicial (gera as strings em $L(G)$)
- N é um conjunto de símbolos não-terminais (variáveis sintáticas)
- T é um conjunto de símbolos terminais (palavras)
- P é um conjunto de produções ou regras de reescrita ($P: N \rightarrow (N \cup T)^+$)



Por que não linguagens regulares e DFAs?

Nem toda linguagem é regular

(RLs \subset CFLs \subset CSLs)

Não dá para construir um DFA para essas linguagens:

- $L = \{ p^k q^k \}$

(linguagens de parênteses)

- $L = \{ w c w^r \mid w \in \Sigma^* \}$

Nenhuma dessas linguagens é regular

Se uma linguagem é regular ou não é um tanto sutil. As linguagens abaixo são regulares

- Os e 1s alternados

$$(\varepsilon \mid 1)(01)^*(\varepsilon \mid 0)$$

- Número par de 0s e 1s



Limites das Linguagens Regulares

Vantagens de Expressões Regulares

- Notação simples e poderosa para especificar padrões
- Construção automática de reconhecedores eficientes
- Vários tipos de sintaxe podem ser especificadas com REs

Exemplo — uma expressão regular para expressões aritméticas

Termo $\rightarrow [a-zA-Z] ([a-zA-Z] | [0-9])^*$

Op $\rightarrow + | - | * | /$

Expr $\rightarrow (\text{Termo Op})^* \text{Termo}$

$[a-zA-Z] ([a-zA-Z] | [0-9])^* (+ | - | * | /)^* [a-zA-Z] ([a-zA-Z] | [0-9])^*$

Naturalmente isso gera um DFA...

Se REs são tão úteis, por que não usá-las para tudo?

\Rightarrow Não pode acrescentar parênteses, chaves, pares begin-end...



Gramáticas Livres de Contexto

O que torna uma gramática "livre de contexto"?

A gramática Ovelha tem uma forma específica:

Ovelha \rightarrow Ovelha bée

| bée

Produções têm um único não-terminal do lado esquerdo, o que torna impossível codificar contexto à esquerda ou direita.

\Rightarrow A gramática é livre de contexto

Uma gramática sensível ao contexto pode ter mais de um não-terminal do lado esquerdo.

Note que $L(\text{Ovelha})$ é na verdade uma linguagem regular: bée⁺



Uma gramática mais útil

Para explorar os usos de CFGs vamos usar Expr

0	Expr	→	Expr Op Expr
1			<u>num</u>
2			<u>id</u>
3	Op	→	+
4			-
5			*
6			/

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
2	<id, <u>x</u> > Op Expr
4	<id, <u>x</u> > - Expr
0	<id, <u>x</u> > - Expr Op Expr
1	<id, <u>x</u> > - <num, <u>2</u> > Op Expr
5	<id, <u>x</u> > - <num, <u>2</u> > * Expr
2	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >

- Uma sequência de reescritas é uma derivação
- O processo de achar uma derivação é o parsing

A derivação acima é: $\text{Expr} \Rightarrow^* \text{id} - \text{num} * \text{id}$



Derivações

O objetivo do parsing é construir uma derivação

- A cada passo, escolhemos um não-terminal para reescrever
- Escolhas distintas levam a derivações distintas

Duas derivações de interesse

- **Mais à esquerda** — reescreva NT mais à esquerda em cada passo
- **Mais à direita** — reescreva NT mais à direita em cada passo

Essas são duas derivações sistemáticas

(Não ligamos para derivações aleatórias)

O exemplo no slide anterior foi uma derivação mais à esquerda

- Naturalmente existe também uma mais à direita
- Vamos ver que nessa gramática ela é bem diferente



Derivações

O objetivo do parsing é construir uma derivação

Uma derivação é uma série de passos de reescrita

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentença}$$

- Cada γ_i é uma forma sentencial
 - Se γ contém apenas terminais, γ é uma **sentença** em $L(G)$
 - Se γ contém 1 ou mais não-terminais, γ é uma **forma sentencial**
- Para obter γ_i de γ_{i-1} , reescreva um NT $A \in \gamma_{i-1}$ usando $A \rightarrow \beta$
 - Troque a ocorrência de $A \in \gamma_{i-1}$ por β para obter γ_i
 - Em uma derivação mais à esquerda, esse seria o primeiro NT $A \in \gamma_{i-1}$

Uma **forma sentencial à esquerda** ocorre em uma derivação à esquerda

Uma **forma sentencial à direita** ocorre em uma derivação à direita



As duas derivações de $\underline{x} - \underline{2} * \underline{y}$

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
2	$\langle \text{id}, \underline{x} \rangle$ Op Expr
4	$\langle \text{id}, \underline{x} \rangle -$ Expr
0	$\langle \text{id}, \underline{x} \rangle -$ Expr Op Expr
1	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle$ Op Expr
5	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle * \text{Expr}$
2	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{y} \rangle$

Mais à esquerda

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
2	Expr Op $\langle \text{id}, \underline{y} \rangle$
5	Expr * $\langle \text{id}, \underline{y} \rangle$
0	Expr Op Expr * $\langle \text{id}, \underline{y} \rangle$
1	Expr Op $\langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{y} \rangle$
4	Expr - $\langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{y} \rangle$
2	$\langle \text{id}, \underline{x} \rangle - \langle \text{num}, \underline{2} \rangle * \langle \text{id}, \underline{y} \rangle$

Mais à direita

Nos dois casos, $\text{Expr} \Rightarrow^* \underline{\text{id}} - \underline{\text{num}} * \underline{\text{id}}$

- As duas derivações produzem árvores diferentes
- As árvores implicam em ordens de avaliação diferentes!

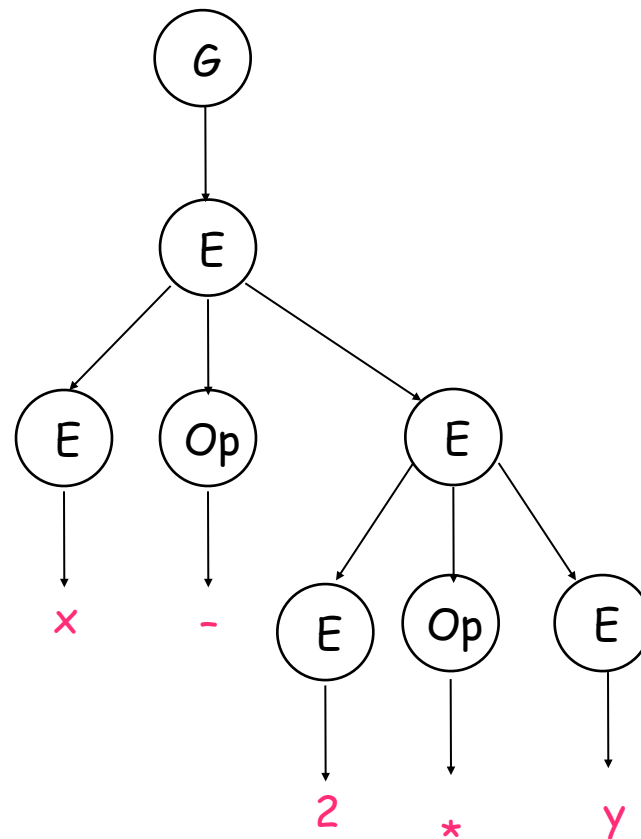


Derivações e Árvores

Mais à esquerda

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
2	$\langle \text{id}, \underline{x} \rangle$ Op Expr
4	$\langle \text{id}, \underline{x} \rangle$ - Expr
0	$\langle \text{id}, \underline{x} \rangle$ - Expr Op Expr
1	$\langle \text{id}, \underline{x} \rangle$ - $\langle \text{num}, \underline{2} \rangle$ Op Expr
5	$\langle \text{id}, \underline{x} \rangle$ - $\langle \text{num}, \underline{2} \rangle$ * Expr
2	$\langle \text{id}, \underline{x} \rangle$ - $\langle \text{num}, \underline{2} \rangle$ * $\langle \text{id}, \underline{y} \rangle$

Isso avalia como $\underline{x} - (\underline{2} * \underline{y})$



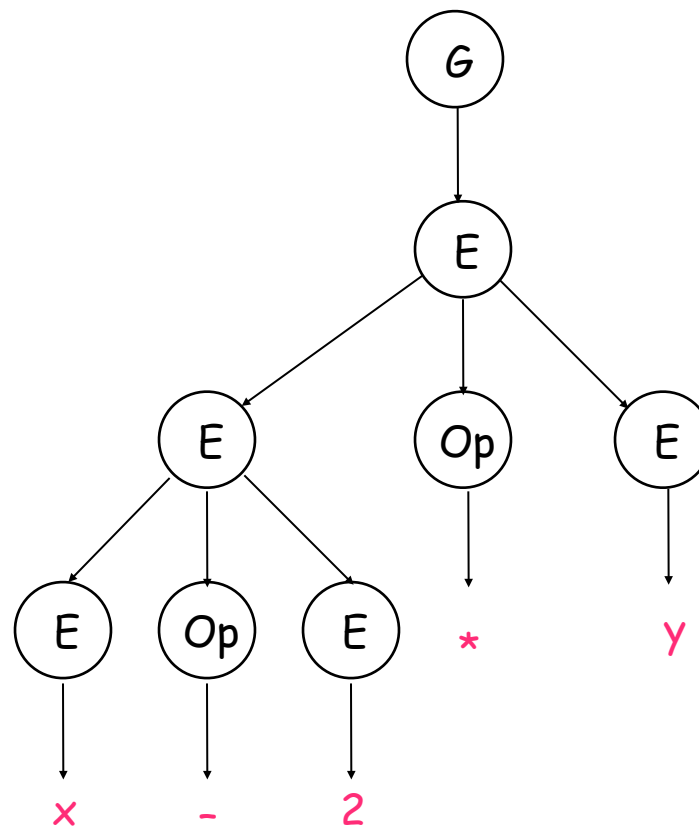


Derivações e Árvores

Mais à direita

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
2	Expr Op <id, <u>y</u> >
5	Expr * <id, <u>y</u> >
0	Expr Op Expr * <id, <u>y</u> >
1	Expr Op <num, <u>2</u> > * <id, <u>y</u> >
4	Expr - <num, <u>2</u> > * <id, <u>y</u> >
2	<id, <u>x</u> > - <num, <u>2</u> > * <id, <u>y</u> >

Isso avalia como $(x - 2) * y$





Derivações e Precedência

Essas duas derivações apontam um problema com a gramática:

Não tem nenhuma noção de precedência, ou ordem de avaliação

Para adicionar precedência

- Crie um não-terminal para cada nível de precedência
- Isole a parte correspondente da gramática
- Force o parser a reconhecer subexpressões de maior precedência primeiro

Para expressões algébricas

- Parênteses primeiro (nível 1)
- Multiplicação e divisão, depois (nível 2)
- Adição e subtração, por último (nível 3)



Derivações e Precedência

Adicionando precedência algébrica tradicional produz:

	0	G	\rightarrow	Expr
nível 3	{	1	Expr	\rightarrow Expr + Termo
		2		Expr - Termo
		3		Termo
nível 2	{	4	Termo	\rightarrow Termo * Fator
		5		Termo / Fator
		6		Fator
nível 1	{	7	Fator	\rightarrow (Expr)
		8		<u>num</u>
		9		<u>id</u>

Ligeiramente maior

- Precisa de mais reescrita para chegar nos mesmos terminais
- Codifica precedência esperada
- Produz a mesma árvore sob derivação mais à esquerda ou direita
- Corretude ganha velocidade do parser

Vamos ver como lida com $x-2*y$

Uma forma da "gramática de expressões clássica"



Derivações e Precedência

Adicionando precedência algébrica tradicional produz:

	0	G	\rightarrow	Expr
nível 3	{	1	Expr	\rightarrow Expr + Termo
		2		Expr - Termo
		3		Termo
nível 2	{	4	Termo	\rightarrow Termo * Fator
		5		Termo / Fator
		6		Fator
nível 1	{	7	Fator	\rightarrow (Expr)
		8		<u>num</u>
		9		<u>id</u>

Ligeiramente maior

- Precisa de mais reescrita para chegar nos mesmos terminais
- Codifica precedência esperada
- Produz a mesma árvore sob derivação mais à esquerda ou direita
- Corretude ganha velocidade do parser

Vamos ver como lida com $x-2*y$

Uma RE para expressões não pode lidar com precedência

Introduzimos parênteses, também (além do poder de RE)

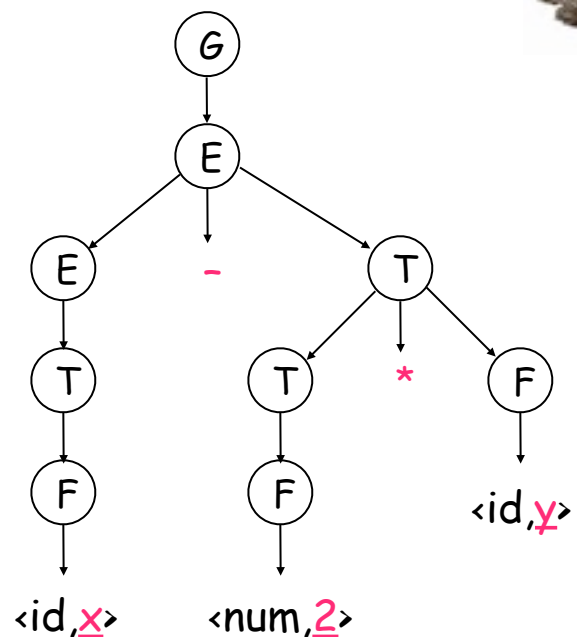
Uma forma da "gramática de expressões clássica"



Derivations and Precedence

Regra	Forma Sentencial
—	G
0	Expr
2	Expr - Termo
4	Expr - Termo * Fator
9	Expr - Termo * $\langle id, y \rangle$
6	Expr - Fator * $\langle id, y \rangle$
8	Expr - $\langle num, z \rangle$ * $\langle id, y \rangle$
3	Termo - $\langle num, z \rangle$ * $\langle id, y \rangle$
6	Fator - $\langle num, z \rangle$ * $\langle id, y \rangle$
9	$\langle id, x \rangle$ - $\langle num, z \rangle$ * $\langle id, y \rangle$

Derivação mais à direita



Árvore de Parsing

Deriva $x - (z * y)$, junto com uma árvore apropriada.

Tanto a derivação mais à esquerda quanto a mais à direita dão a mesma expressão, pois a gramática tem a precedência correta explicitamente.



Gramáticas Ambíguas

Vamos voltar para a gramática de expressões original.

Ela tem outros problemas.

0	Expr	→	Expr Op Expr
1			<u>num</u>
2			<u>id</u>
3	Op	→	+
4			-
5			*
6			/

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
②	<id, <u>x</u> > Op Expr
4	<id, <u>x</u> > - Expr
0	<id, <u>x</u> > - Expr Op Expr
1	<id, <u>x</u> > - <num, <u>z</u> > Op Expr
5	<id, <u>x</u> > - <num, <u>z</u> > * Expr
2	<id, <u>x</u> > - <num, <u>z</u> > * <id, <u>y</u> >

- Essa gramática permite múltiplas derivações mais à esquerda de $\underline{x} - \underline{z} * \underline{y}$
- Difícil de automatizar derivação se tem mais de uma escolha
- A gramática é **ambígua**

Escolha diferente



Duas derivações à esquerda para $x - 2 * y$

A diferença:

- Produções diferentes escolhidas no segundo passo

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
②	$\langle \text{id}, x \rangle$ Op Expr
4	$\langle \text{id}, x \rangle$ - Expr
0	$\langle \text{id}, x \rangle$ - Expr Op Expr
1	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$ Op Expr
5	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$ * Expr
1	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$ * $\langle \text{id}, y \rangle$

Escolha original

Regra	Forma Sentencial
—	Expr
0	Expr Op Expr
①	Expr Op Expr Op Expr
2	$\langle \text{id}, x \rangle$ Op Expr Op Expr
4	$\langle \text{id}, x \rangle$ - Expr Op Expr
1	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$ Op Expr
5	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$ * Expr
2	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$ * $\langle \text{id}, y \rangle$

Nova escolha

- Ambas produzem $x - 2 * y$



Gramáticas Ambíguas

Definições

- Se uma gramática tem mais de uma derivação mais à esquerda para uma mesma forma sentencial, a gramática é **ambígua**
- Se uma gramática tem mais de uma derivação mais à direita para uma mesma forma sentencial, a gramática é **ambígua**
- As derivações mais à esquerda e mais à direita podem diferir, mesmo em uma gramática não ambígua
 - Mas a árvore de parsing tem que ser a mesma!

Exemplo clássico — o problema do if-then-else

```
Cmd → if Expr then Cmd  
      | if Expr then Cmd else Cmd  
      | ... outros cmds ...
```

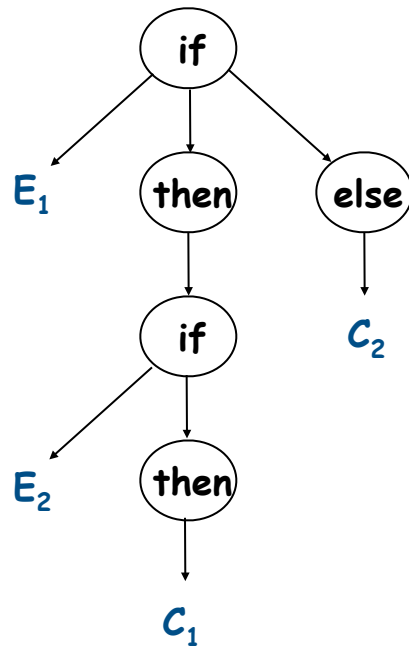
Ambiguidade inerente na gramática



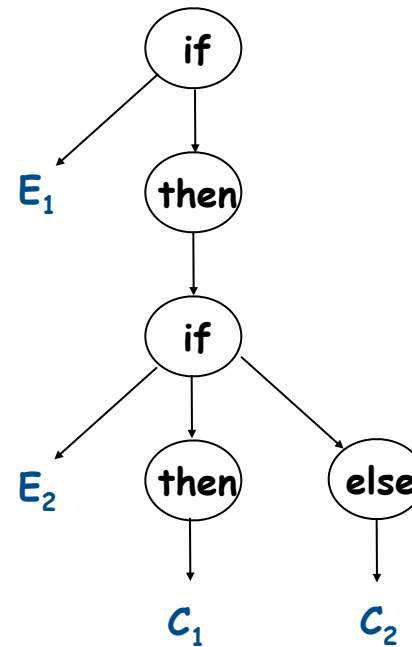
Ambiguidade

Essa forma sentencial tem duas derivações

if $Expr_1$ then if $Expr_2$ then Cmd_1 else Cmd_2



regra 2, então
regra 1



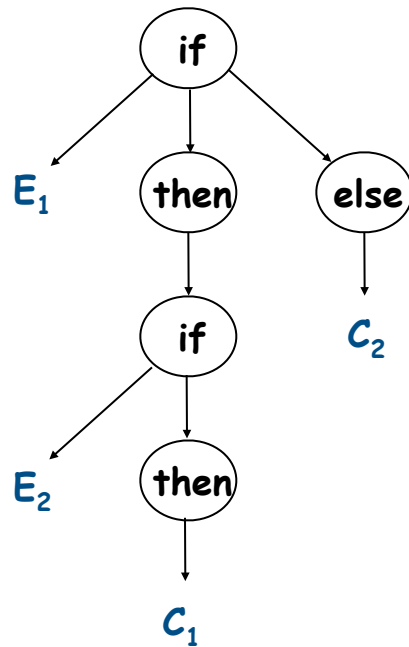
regra 1, então
regra 2



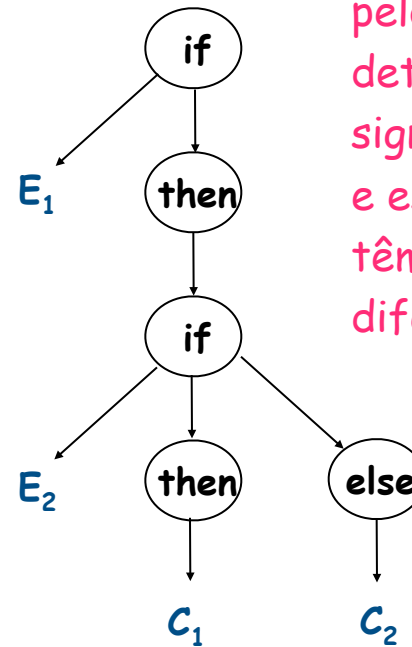
Ambiguidade

Essa forma sentencial tem duas derivações

if $Expr_1$ then if $Expr_2$ then Cmd_1 else Cmd_2



regra 2, então
regra 1



regra 1, então
regra 2

O problema é que a estrutura construída pelo parser vai determinar o significado do código, e essas duas formas têm significados diferentes!



Ambiguidade

Removendo a ambiguidade

- Reescrever a gramática para evitar o problema
- Casa cada else com o if mais interno (*senso comum*)

0	Cmd	→	<u>if</u> Expr <u>then</u> Cmd
1			<u>if</u> Expr <u>then</u> CmdElse <u>else</u> Cmd
2			outros comandos
3	CmdElse	→	<u>if</u> Expr <u>then</u> CmdElse <u>else</u> CmdElse
4			outros comandos

Intuição: uma vez em CmdElse, não podemos gerar um else não casado
... um if sem um else só pode vir da regra regra 1 ...

Apenas uma derivação mais à direita para o exemplo



Ambiguidade

if $Expr_1$ then if $Expr_2$ then Cmd_1 else Cmd_2

Regra	Forma Sentencial
—	Cmd
0	<u>if</u> $Expr$ <u>then</u> Cmd
1	<u>if</u> $Expr$ <u>then</u> <u>if</u> $Expr$ <u>then</u> $CmdElse$ <u>else</u> Cmd
2	<u>if</u> $Expr$ <u>then</u> <u>if</u> $Expr$ <u>then</u> $CmdElse$ <u>else</u> C_2
4	<u>if</u> $Expr$ <u>then</u> <u>if</u> $Expr$ <u>then</u> C_1 <u>else</u> C_2
?	<u>if</u> $Expr$ <u>then</u> <u>if</u> E_2 <u>then</u> C_1 <u>else</u> C_2
?	<u>if</u> E_1 <u>then</u> <u>if</u> E_2 <u>then</u> C_1 <u>else</u> C_2

Outras produções para derivar E_s

Apenas uma derivação mais à direita para o exemplo



Ambiguidade mais profunda

Ambiguidade normalmente é uma confusão na CFG

Sobrecarga pode causar ambiguidade mais profunda

$$a = f(17)$$

Em várias linguagens f pode ser uma função ou um array

Remover essa ambiguidade requer contexto

- Como f foi declarado
- Uma questão de tipo e não de sintaxe livre de contexto
- Requer solução extra-gramatical (fora da CFG)
- Deve ser lidada com um mecanismo diferente
 - Sair da gramática ao invés de usar uma gramática mais complexa



Ambiguidade - palavra final

Ambiguidade surge de duas fontes distintas

- Confusão na sintaxe livre de contexto (if-then-else)
- Confusão que requer contexto para resolver (sobrecarga)

Resolvendo ambiguidade

- Para remover ambiguidade livre de contexto, reescreva a gramática
- Para lidar com ambiguidade sensível ao contexto saia da gramática
 - Conhecimento de declarações, tipos...
 - Aceita um superconjunto de $L(G)$ e verifique por outros meios
 - Problema de projeto de linguagem

Às vezes, se aceita uma gramática ambígua

- Técnicas de análise que "fazem a coisa certa"
- por ex., sempre escolhem a mesma derivação



Técnicas de Análise Sintática

Parsers top-down (LL(1), recursivos)

- Comece da raiz da árvore e vá em direção às folhas
- Escolha uma produção e tente casar com a entrada
- Má escolha \Rightarrow pode precisar voltar
- Algumas gramáticas não precisam voltar (parsing preditivo)

Parsers bottom-up (LR(1), parser de precedência)

- Comece nas folhas e vá em direção à raiz
- À medida que a entrada é consumida, codifique as escolhas em um estado interno
- Comece em um estado válido para primeiros tokens
- Parsers bottom-up lidam com uma grande classe de gramáticas



Análise top-down

Um parser top-down começa na raiz da árvore de parse

O nó raiz é rotulado com o símbolo inicial da gramática

Algoritmo de análise top-down:

Construa o nó raiz da árvore

Repita até franja da árvore casar com a entrada

- 1 Em um nó com rótulo A , escolha uma produção de A e, para cada símbolo no lado direito, construa o filho apropriado
- 2 Quando um terminal é adicionado à franja mas não casa com a entrada, volte e tente outra escolha
- 3 Ache o próximo nó para expandir (rótulo \in NT)

A chave é escolher a produção certa no passo 1

- Escolha guiada pela entrada



Lembra da gramática de expressões?

Chamamos essa versão de "gramática de expressões clássica"

0	G	\rightarrow	Expr
1	Expr	\rightarrow	Expr + Termo
2			Expr - Termo
3			Termo
4	Termo	\rightarrow	Termo * Fator
5			Termo / Fator
6			Fator
7	Fator	\rightarrow	(Expr)
8			<u>num</u>
9			<u>id</u>

A entrada é $\underline{x} - \underline{2} * \underline{y}$



Exemplo

Vamos tentar $\underline{x} - \underline{2} * \underline{y}$:

G

Regra	Forma Sentencial	Entrada
-	G	$\uparrow x - 2 * y$

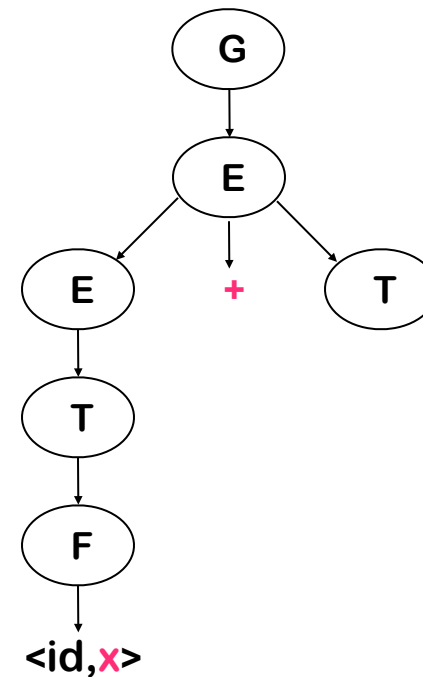
\uparrow é a posição na entrada



Exemplo

Vamos tentar $x - 2 * y$:

Regra	Forma Sentencial	Entrada
—	G	$\uparrow x - 2 * y$
0	Expr	$\uparrow x - 2 * y$
1	Expr + Termo	$\uparrow x - 2 * y$
3	Termo + Termo	$\uparrow x - 2 * y$
6	Fator + Termo	$\uparrow x - 2 * y$
9	$\langle id, x \rangle$ + Termo	$\uparrow x - 2 * y$
→	$\langle id, x \rangle$ + Termo	$x \uparrow - 2 * y$



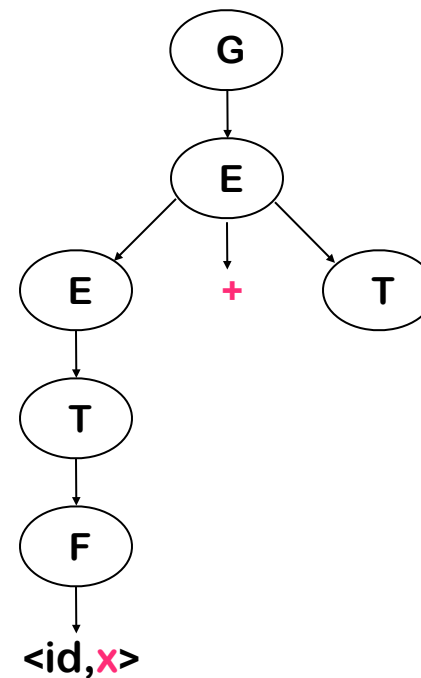
↑ é a posição na entrada



Exemplo

Vamos tentar $x - z * y$:

Regra	Forma Sentencial	Entrada
—	G	$\uparrow x - z * y$
0	Expr	$\uparrow x - z * y$
1	Expr + Termo	$\uparrow x - z * y$
3	Termo + Termo	$\uparrow x - z * y$
6	Fator + Termo	$\uparrow x - z * y$
9	$\langle id, x \rangle$ + Termo	$\uparrow x - z * y$
→	$\langle id, x \rangle$ + Termo	$x \uparrow - z * y$



Funcionou bem, exceto que "-" não casa "+"

O parser tem que voltar pra cá

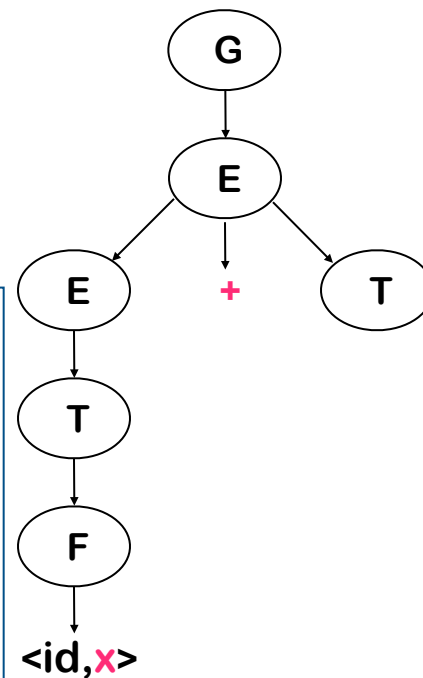
↑ é a posição na entrada



Exemplo

Vamos tentar $x - z * y$:

Regra	Forma Sentencial	Entrada
—	G	$\uparrow x - z * y$
0	Expr	$\uparrow x - z * y$
1	Expr + Termo	$\uparrow x - z * y$
3	Termo + Termo	$\uparrow x - z * y$
6	Fator + Termo	$\uparrow x - z * y$
9	$\langle id, x \rangle$ + Termo	$\uparrow x - z * y$
→	$\langle id, x \rangle$ + Termo	$x \uparrow - z * y$



Funcionou bem, exceto que "-" não casa "+"

O parser tem que voltar pra cá

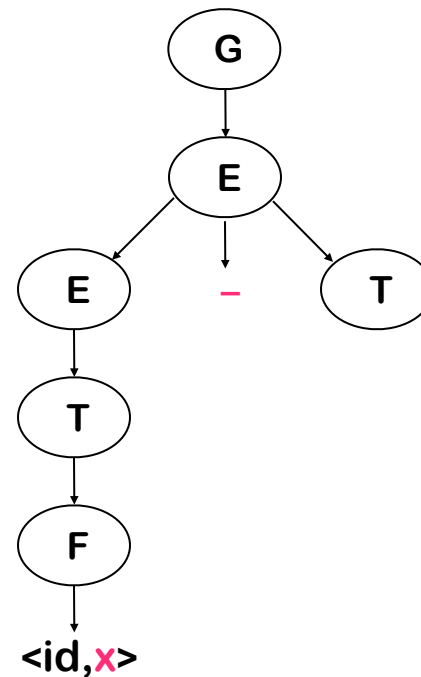
↑ é a posição na entrada



Exemplo

Continuando com $x - 2 * y$:

Regra	Forma Sentencial	Entrada
—	G	$\uparrow x - 2 * y$
0	Expr	$\uparrow x - 2 * y$
2	Expr - Termo	$\uparrow x - 2 * y$
3	Termo - Termo	$\uparrow x - 2 * y$
6	Fator - Termo	$\uparrow x - 2 * y$
9	$\langle id, x \rangle$ - Termo	$\uparrow x - 2 * y$
→	$\langle id, x \rangle$ - Termo	$x \uparrow - 2 * y$
→	$\langle id, x \rangle$ - Termo	$x - \uparrow 2 * y$

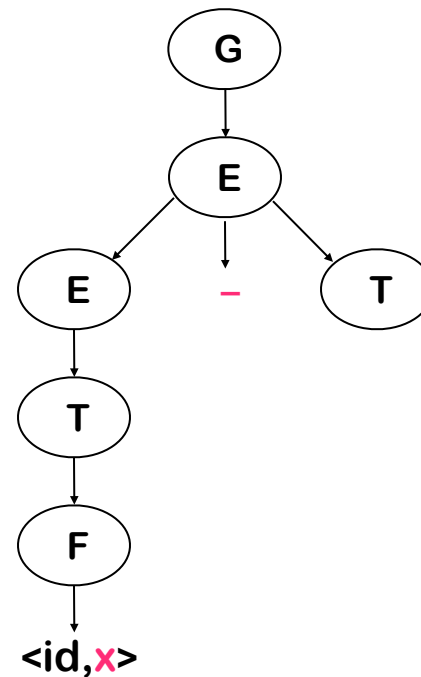




Exemplo

Continuando com $x - 2 * y$:

Regra	Forma Sentencial	Entrada
—	G	$\uparrow x - 2 * y$
0	Expr	$\uparrow x - 2 * y$
2	Expr - Termo	$\uparrow x - 2 * y$
3	Termo - Termo	$\uparrow x - 2 * y$
6	Fator - Termo	$\uparrow x - 2 * y$
9	$\langle id, x \rangle$ - Termo	$\uparrow x - 2 * y$
→	$\langle id, x \rangle$ - Termo	$x \uparrow - 2 * y$
→	$\langle id, x \rangle$ - Termo	$x - \uparrow 2 * y$



⇒ Agora precisamos expandir Termo - o último NT na franja



Exemplo

Tentar casar o "2" em $x - 2 * y$:

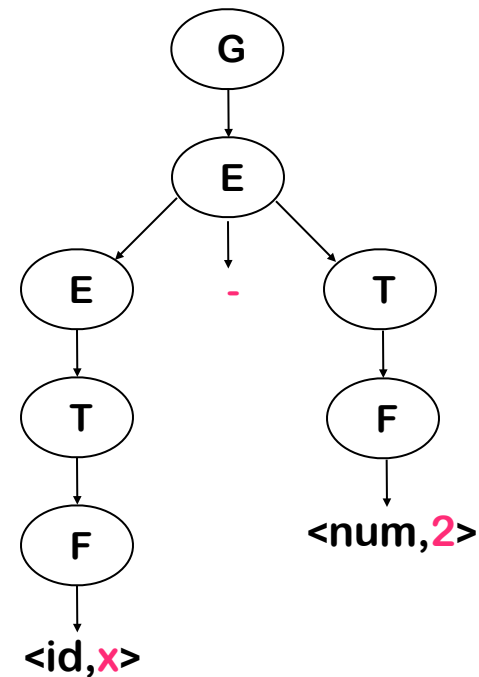
Regra	Forma Sentencial	Entrada
→	$\langle \text{id}, x \rangle$ - Termo	$x - \uparrow 2 * y$
6	$\langle \text{id}, x \rangle$ - Fator	$x - \uparrow 2 * y$
8	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$	$x - \uparrow 2 * y$
→	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$	$x - 2 \uparrow * y$



Exemplo

Tentar casar o "2" em $x - 2 * y$:

Regra	Forma Sentencial	Entrada
→	$\langle id, x \rangle$ - Termo	$x - \uparrow 2 * y$
6	$\langle id, x \rangle$ - Fator	$x - \uparrow 2 * y$
8	$\langle id, x \rangle$ - $\langle num, 2 \rangle$	$x - \uparrow 2 * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$	$x - 2 \uparrow * y$

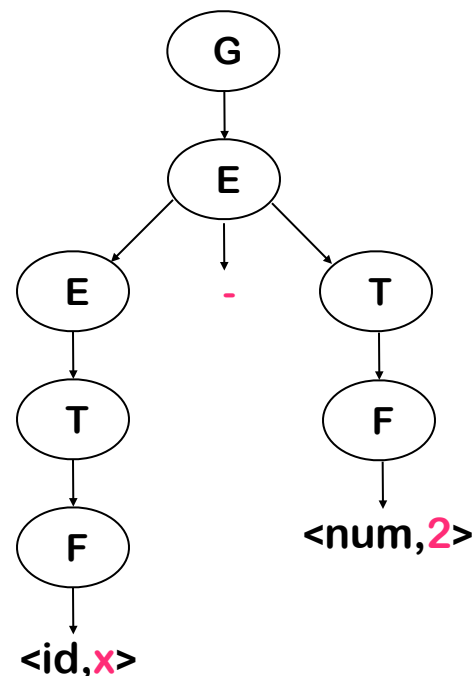




Exemplo

Tentar casar o "2" em $x - 2 * y$:

Regra	Forma Sentencial	Entrada
→	$\langle \text{id}, x \rangle$ - Termo	$x - \uparrow 2 * y$
6	$\langle \text{id}, x \rangle$ - Fator	$x - \uparrow 2 * y$
8	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$	$x - \uparrow 2 * y$
→	$\langle \text{id}, x \rangle$ - $\langle \text{num}, 2 \rangle$	$x - 2 \uparrow * y$



Onde estamos?

- "2" casa com "2"
 - Temos mais entrada, mas acabaram os NTs para expandir
 - Terminamos a expansão muito cedo
- ⇒ Voltar novamente...



Exemplo

Tentando de novo com "2" em $x - \underline{2} * y$:

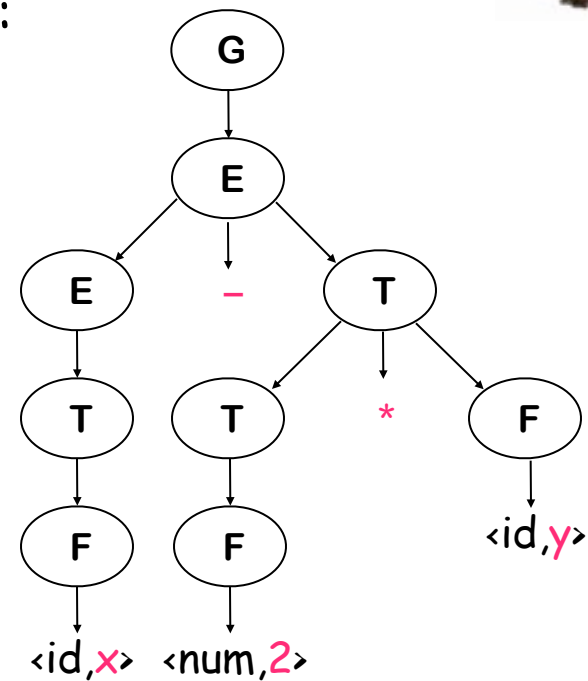
Regra	Forma Sentencial	Entrada
→	$\langle id, \underline{x} \rangle$ - Termo	$x - \uparrow \underline{2} * y$
4	$\langle id, \underline{x} \rangle$ - Termo * Fator	$x - \uparrow \underline{2} * y$
6	$\langle id, \underline{x} \rangle$ - Fator * Fator	$x - \uparrow \underline{2} * y$
8	$\langle id, \underline{x} \rangle$ - $\langle num, \underline{2} \rangle$ * Fator	$x - \uparrow \underline{2} * y$
→	$\langle id, \underline{x} \rangle$ - $\langle num, \underline{2} \rangle$ * Fator	$x - \underline{2} \uparrow * y$
→	$\langle id, \underline{x} \rangle$ - $\langle num, \underline{2} \rangle$ * Fator	$x - \underline{2} * \uparrow y$
9	$\langle id, \underline{x} \rangle$ - $\langle num, \underline{2} \rangle$ * $\langle id, \underline{y} \rangle$	$x - \underline{2} * \uparrow y$
→	$\langle id, \underline{x} \rangle$ - $\langle num, \underline{2} \rangle$ * $\langle id, \underline{y} \rangle$	$x - \underline{2} * y \uparrow$



Exemplo

Tentando de novo com "2" em $x - 2 * y$:

Regra	Forma Sentencial	Entrada
→	$\langle id, x \rangle$ - Termo	$x - \uparrow 2 * y$
4	$\langle id, x \rangle$ - Termo * Fator	$x - \uparrow 2 * y$
6	$\langle id, x \rangle$ - Fator * Fator	$x - \uparrow 2 * y$
8	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - \uparrow 2 * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - 2 \uparrow * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - 2 * \uparrow y$
9	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$	$x - 2 * \uparrow y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$	$x - 2 * y \uparrow$

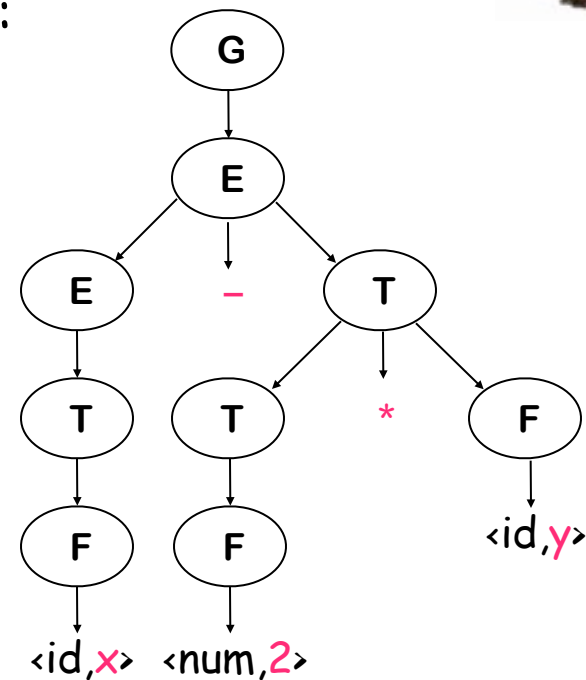




Exemplo

Tentando de novo com "2" em $x - 2 * y$:

Regra	Forma Sentencial	Entrada
→	$\langle id, x \rangle$ - Termo	$x - \uparrow 2 * y$
4	$\langle id, x \rangle$ - Termo * Fator	$x - \uparrow 2 * y$
6	$\langle id, x \rangle$ - Fator * Fator	$x - \uparrow 2 * y$
8	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - \uparrow 2 * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - 2 \uparrow * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - 2 * \uparrow y$
9	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$	$x - 2 * \uparrow y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$	$x - 2 * y \uparrow$



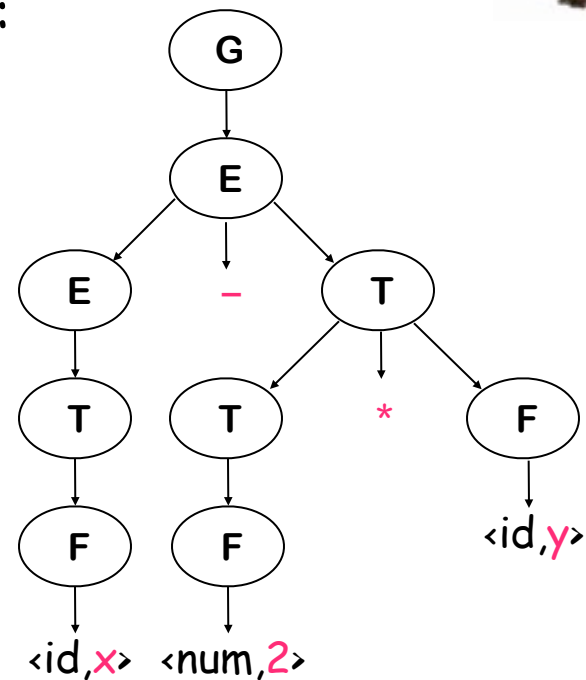
Dessa vez, casamos e consumimos toda a entrada
⇒ Sucesso!



Exemplo

Tentando de novo com "2" em $x - 2 * y$:

Regra	Forma Sentencial	Entrada
→	$\langle id, x \rangle$ - Termo	$x - \uparrow 2 * y$
4	$\langle id, x \rangle$ - Termo * Fator	$x - \uparrow 2 * y$
6	$\langle id, x \rangle$ - Fator * Fator	$x - \uparrow 2 * y$
8	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - \uparrow 2 * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - 2 \uparrow * y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * Fator	$x - 2 * \uparrow y$
9	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$	$x - 2 * \uparrow y$
→	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$	$x - 2 * y \uparrow$



Conclusão:

O parser tem que fazer a escolha correta quando expande um NT. Escolhas erradas são esforço desperdiçado.



Outra análise possível

Outras escolhas para expansão são possíveis

Regra	Forma Sentencial	Entrada
—	G	$\uparrow x - \underline{2} * y$
0	Expr	$\uparrow x - \underline{2} * y$
1	Expr + Termo	$\uparrow x - \underline{2} * y$
1	Expr + Termo + Termo	$\uparrow x - \underline{2} * y$
1	Expr + Termo + Termo + Termo	$\uparrow x - \underline{2} * y$
1	e por aí vai...	$\uparrow x - \underline{2} * y$

Não consome a entrada!

Essa expansão não termina

- Escolha da expansão errada leva a não terminação
- O parser não pode entrar em um loop infinito!
- Parser deve fazer a escolha correta



Recursão à Esquerda

Parsers top-down não lidam com recursão à esquerda

Formalmente,

Uma gramática é recursiva à esquerda se $\exists A \in NT$ tal que
 \exists uma derivação $A \Rightarrow^+ A\alpha$, para alguma string $\alpha \in (NT \cup T)^+$

Nossa gramática de expressões é recursiva à esquerda

- Isso leva a não-terminação em um parser top-down
- Em um parser top-down, qualquer recursão tem que ser à direita
- Queremos converter recursão à esquerda em recursão à direita

Um compilador nunca pode entrar em loop infinito



Eliminando Recursão à Esquerda

Para remover recursão à esquerda, podemos transformar a gramática

Considere um fragmento de gramática com a forma

$$\begin{array}{l} \text{Foo} \rightarrow \text{Foo } \alpha \\ \quad \quad | \beta \end{array}$$

onde nem α nem β começam com Foo

Podemos reescrevê-lo como

$$\begin{array}{l} \text{Foo} \rightarrow \beta \text{ Bar} \\ \text{Bar} \rightarrow \alpha \text{ Bar} \\ \quad \quad | \varepsilon \end{array}$$

onde Bar é um novo não-terminal

A nova gramática define a mesma linguagem, usando apenas recursão à direita

Vazio



Eliminando Recursão à Esquerda

A gramática de expressões tem dois casos de rec. à esq.

$$\begin{array}{l} \text{Expr} \rightarrow \text{Expr} + \text{Termo} \\ \quad | \text{Expr} - \text{Termo} \\ \quad | \text{Termo} \end{array} \qquad \begin{array}{l} \text{Termo} \rightarrow \text{Termo} * \text{Fator} \\ \quad | \text{Termo} / \text{Fator} \\ \quad | \text{Fator} \end{array}$$

Aplicando a transformação leva a

$$\begin{array}{l} \text{Expr} \rightarrow \text{Termo Expr}' \\ \text{Expr}' \rightarrow + \text{Termo Expr}' \\ \quad | - \text{Termo Expr}' \\ \quad | \varepsilon \end{array} \qquad \begin{array}{l} \text{Termo} \rightarrow \text{Fator Termo}' \\ \text{Termo}' \rightarrow * \text{Fator Termo}' \\ \quad | / \text{Fator Termo}' \\ \quad | \varepsilon \end{array}$$

Esses fragmentos usam apenas recursão à direita

Recursão à direita geralmente implica associatividade à direita, mas nesse caso a associatividade à esquerda é preservada.



Eliminando Recursão à Esquerda

Substituindo de volta na gramática

0	G	\rightarrow	Expr
1	Expr	\rightarrow	Termo Expr'
2	Expr'	\rightarrow	+ Termo Expr'
3			- Termo Expr'
4			ϵ
5	Termo	\rightarrow	Fator Termo'
6	Termo'	\rightarrow	* Fator Termo'
7			/ Fator Termo'
8			ϵ
9	Fator	\rightarrow	(Expr)
10			<u>num</u>
11			<u>id</u>

- Não muito intuitiva, mas correta
- É associativa à esquerda, como a original
 - \Rightarrow Transformação ingênua gera uma gramática que associa à direita



Escolhendo a Produção "Certa"

Se escolher a produção errada um parser top-down vai precisar voltar

Alternativa é espiar a entrada e usar isso para escolher a produção (lookahead)

Quanto lookahead é preciso?

- No caso geral, uma quantidade arbitrária
- Algoritmos não tão eficientes (cúbico em função do tamanho da entrada)

Por sorte,

- Muitas CFGs podem ser analisadas com lookahead limitado (e pequeno)
- A maior parte das linguagens de programação está nesse caso

As classes mais interessantes de gramática são LL(1) e LR(1)

Vamos começar com LL(1) e parsers preditivos (top-down)



Parsers Preditivos

Ideia básica

Dado $A \rightarrow \alpha \mid \beta$, o parser deve escolher poder escolher entre α e β

Conjuntos FIRST

Para algum lado direito $\alpha \in G$, $\text{FIRST}(\alpha)$ é o conjunto de tokens que aparece como primeiro símbolo nas strings que α deriva

Isso é, $\underline{x} \in \text{FIRST}(\alpha)$ se só se $\alpha \Rightarrow^* \underline{x} \gamma$, para algum γ

Por enquanto vamos deixar o problema de calcular conjuntos FIRST pra lá.



Parsers Preditivos

Ideia básica

Dado $A \rightarrow \alpha \mid \beta$, o parser deve escolher poder escolher entre α e β

Conjuntos FIRST

Para algum lado direito $\alpha \in G$, $\text{FIRST}(\alpha)$ é o conjunto de tokens que aparece como primeiro símbolo nas strings que α deriva

Isso é, $\underline{x} \in \text{FIRST}(\alpha)$ se só se $\alpha \Rightarrow^* \underline{x} \gamma$, para algum γ

A Propriedade LL(1)

Se $A \rightarrow \alpha$ e $A \rightarrow \beta$ são produções da gramática, gostaríamos que

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

Isso permitiria ao parser escolher qual produção seguir com um símbolo de lookahead!



Parsers Preditivos

E quanto a produções ϵ ?

⇒ Elas complicam a definição de LL(1)

Se $A \rightarrow \alpha$ e $A \rightarrow \beta$ e $\epsilon \in \text{FIRST}(\alpha)$, então precisamos garantir que $\text{FIRST}(\beta)$ é disjunto de $\text{FOLLOW}(A)$, também, onde

FOLLOW(A) = o conjunto de símbolos terminais seguem imediatamente A em uma forma sentencial

Isto é, $\underline{x} \in \text{FOLLOW}(A)$ se $S \Rightarrow^* \alpha A \underline{x} \gamma$ para algum α e γ

Definindo **FIRST⁺(A → α)** como

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, se $\epsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$, caso contrário

Então uma gramática é LL(1) se só se $A \rightarrow \alpha$ e $A \rightarrow \beta$ implica

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$$



Parsers Preditivos

Dada uma gramática com a propriedade LL(1)

- Podemos escrever uma rotina simples para reconhecer cada lado esquerdo
- Código é simples e rápido

Considere $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, com

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset \text{ se } i \neq j$$



Parsers Preditivos

Dada uma gramática com a propriedade LL(1)

- Podemos escrever uma rotina simples para reconhecer cada lado esquerdo
- Código é simples e rápido

Considere $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, com

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset \text{ se } i \neq j$$

```
/* reconheça A */  
if (lookahead  $\in$  FIRST $^+(A \rightarrow \beta_1)$ )  
    reconheça  $\beta_1$  e retorne  
else if (lookahead  $\in$  FIRST $^+(A \rightarrow \beta_2)$ )  
    reconheça  $\beta_2$  e retorne  
else if (lookahead  $\in$  FIRST $^+(A \rightarrow \beta_3)$ )  
    reconheça  $\beta_3$  e retorne  
else  
    erro de sintaxe
```



Parsers Preditivos

Dada uma gramática com a propriedade LL(1)

- Podemos escrever uma rotina simples para reconhecer cada lado esquerdo
- Código é simples e rápido

Considere $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, com

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset \text{ se } i \neq j$$

```
/* reconheça A */  
if (lookahead  $\in$  FIRST+(A  $\rightarrow$   $\beta_1$ ))  
    reconheça  $\beta_1$  e retorne  
else if (lookahead  $\in$  FIRST+(A  $\rightarrow$   $\beta_2$ ))  
    reconheça  $\beta_2$  e retorne  
else if (lookahead  $\in$  FIRST+(A  $\rightarrow$   $\beta_3$ ))  
    reconheça  $\beta_3$  e retorne  
else  
    erro de sintaxe
```

Gramáticas com a propriedade LL(1) são chamadas gramáticas preditivas pois o parser pode “prever” a expansão correta em cada ponto da análise.

Parsers que aproveitam a propriedade LL(1), ou sua generalização para lookaheads maiores, são chamados parsers preditivos.

Um tipo de parser preditivo é o parser recursivo.



Parsers Recursivos

Lembre da gramática de expressões, depois de transformada

0	G	\rightarrow	Expr
1	Expr	\rightarrow	Termo Expr'
2	Expr'	\rightarrow	+ Termo Expr'
3			- Termo Expr'
4			ϵ
5	Termo	\rightarrow	Fator Termo'
6	Termo'	\rightarrow	* Fator Termo'
7			/ Fator Termo'
8			ϵ
9	Fator	\rightarrow	(Expr)
10			<u>num</u>
11			<u>id</u>

Isso produz um parser com seis rotinas mutuamente recursivas:

- G
- Expr
- ExprL
- Termo
- TermoL
- Fator

Cada uma reconhece um NT.

O parser prossegue de modo top-down.



Parsers Recursivos

Algumas rotinas do parser de expressões

G()

```
token ← next_token();  
if (Expr() = true & token = EOF)  
  then próxima fase;  
  else  
    erro de sintaxe;  
    return false;
```

Expr()

```
if (Termo() = false)  
  then return false;  
  else return ExprL();
```




Parsers Recursivos

Algumas rotinas do parser de expressões

G()

```
token ← next_token();  
if (Expr() = true & token = EOF)  
  then próxima fase;  
  else  
    erro de sintaxe;  
    return false;
```

Expr()

```
if (Termo() = false)  
  then return false;  
  else return ExprL();
```

Fator()

```
if (token = NUM) then  
  token ← next_token();  
  return true;  
else if (token = ID) then  
  token ← next_token();  
  return true;  
else if (token = '(')  
  token ← next_token();  
  if (Expr() = true & token = ')') then  
    token ← next_token();  
    return true;  
  // saindo do if  
  erro de sintaxe;  
  return false;
```



Parsers Recursivos

Algumas rotinas do parser de expressões

G()

```
token ← next_token();  
if (Expr() = true & token = EOF)  
  then próxima fase;  
  else  
    erro de sintaxe;  
    return false;
```

Expr()

```
if (Termo() = false)  
  then return false;  
  else return ExprL();
```

Pode ficar separado nos ifs internos para melhores mensagens de erro!

Fator()

```
if (token = NUM) then  
  token ← next_token();  
  return true;  
else if (token = ID) then  
  token ← next_token();  
  return true;  
else if (token = '(')  
  token ← next_token();  
  if (Expr() = true & token = ')') then  
    token ← next_token();  
    return true;  
  // saindo do if  
  erro de sintaxe;  
  return false;
```



E se a gramática não for LL(1) ?

Podemos transformar uma gramática não-LL(1) em LL(1)?

- No caso geral não
- Em muitos casos, sim

Seja uma gramática G com produções $A \rightarrow \alpha \beta_1$ e $A \rightarrow \alpha \beta_2$

- Se α não deriva ε , então

$$\text{FIRST}^+(A \rightarrow \alpha \beta_1) \cap \text{FIRST}^+(A \rightarrow \alpha \beta_2) \neq \emptyset$$

- E a gramática não é LL(1)

Se passarmos o prefixo comum α para uma produção separada então talvez a gramática se torne LL(1).

$$A \rightarrow \alpha A', A' \rightarrow \beta_1 \text{ e } A' \rightarrow \beta_2$$

Agora, se $\text{FIRST}^+(A' \rightarrow \beta_1) \cap \text{FIRST}^+(A' \rightarrow \beta_2) = \emptyset$, G pode ser LL(1)



E se a gramática não for LL(1) ?

Fatoração à esquerda

Para cada não-terminal A

ache o maior prefixo α comum a 2 ou mais alternativas para A

se $\alpha \neq \epsilon$ então

troque todas as produções

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \alpha \beta_3 \mid \dots \mid \alpha \beta_n \mid \gamma$$

por

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$$

Repita até que nenhum terminal tenha produções com prefixos comuns

Essa transformação torna algumas gramáticas LL(1)

Mas existem linguagens que não têm gramáticas LL(1)



Fatoração à Esquerda

Seja uma gramática de expressões recursiva à direita simples

0		G	\rightarrow	Expr
1		Expr	\rightarrow	Termo + Expr
2				Termo - Expr
3				Termo
4		Termo	\rightarrow	Fator * Termo
5				Fator / Termo
6				Fator
7		Fator	\rightarrow	<u>num</u>
8				<u>id</u>

First+(1) = First+(2) = First+(3)

First+(4) = First+(5) = First+(6)

Vamos fatorar essa gramática.



Fatoração à Esquerda

Após fatorar nós temos

0	G	\rightarrow	Expr
1	Expr	\rightarrow	Termo Expr'
2	Expr'	\rightarrow	+ Expr
3			- Expr
4			ϵ
5	Termo	\rightarrow	Fator Termo'
6	Termo'	\rightarrow	* Termo
7			/ Termo
8			ϵ
9	Fator	\rightarrow	<u>num</u>
10			<u>id</u>

A gramática agora é LL(1)



Parsers Preditivos, novamente

Dada uma gramática LL(1), e seus conjuntos FIRST e FOLLOW

- Emita uma rotina para cada não-terminal
 - Cascata de if-then-else para escolher alternativas
 - Retorna true em sucesso (ou executa ação) ou reporta erro
 - Código simples e funcional, mas pode ser feio
- Construção automática de um parser recursivo!

Melhorando a situação

- Cascata de if-then-else pode ser lenta
 - Um bom switch pode ser melhor
- Que tal usar uma tabela?
 - Interpretar a tabela com um parser genérico, como no scanner



Construção de Parsers LL(1)

Estratégia

- Codificar gramática em tabela
- Usar parser genérico para interpretar

Exemplo

- O não-terminal Fator tem 3 alternativas
 - (Expr) ou id ou num
- Tabela pode ser:

0	G	→	Expr
1	Expr	→	Termo Expr'
2	Expr'	→	+ Termo Expr'
3			- Termo Expr'
4			ε
5	Termo	→	Fator Termo'
6	Termo'	→	* Fator Termo'
7			/ Fator Termo'
8			ε
9	Fator	→	<u>num</u>
10			<u>id</u>
11			(Expr)

		Terminais								
		+	-	*	/	Id.	Num.	()	EOF
Não-terminais	<u>Fator</u>	—	—	—	—	10	9	11	—	—



Construção de Parsers LL(1)

Estratégia

- Codificar gramática em tabela
- Usar parser genérico para interpretar

Exemplo

- O não-terminal Fator tem 3 alternativas
 - (Expr) ou id ou num
- Tabela pode ser:

0	G	→	Expr
1	Expr	→	Termo Expr'
2	Expr'	→	+ Termo Expr'
3			- Termo Expr'
4			ε
5	Termo	→	Fator Termo'
6	Termo'	→	* Fator Termo'
7			/ Fator Termo'
8			ε
9	Fator	→	<u>num</u>
10			<u>id</u>
11			(Expr)

		Terminais								
		+	-	*	/	Id.	Num.	()	EOF
Não-terminais	Fator	—	—	—	—	10	9	11	—	—

Parser deve acusar um erro



Construção de Parsers LL(1)

Estratégia

- Codificar gramática em tabela
- Usar parser genérico para interpretar

Exemplo

- O não-terminal Fator tem 3 alternativas
 - (Expr) ou id ou num
- Tabela pode ser:

0	G	→	Expr
1	Expr	→	Termo Expr'
2	Expr'	→	+ Termo Expr'
3			- Termo Expr'
4			ε
5	Termo	→	Fator Termo'
6	Termo'	→	* Fator Termo'
7			/ Fator Termo'
8			ε
9	Fator	→	<u>num</u>
10			<u>id</u>
11			(Expr)

		Terminais								
		+	-	*	/	Id.	Num.	()	EOF
Não-terminais	Fator	—	—	—	—	10	9	11	—	—

Parser deve acusar um erro

Se a entrada é um número use produção 9



Tabela de Parsing LL(1) para Expressões

	+	-	*	/	Id	Num	()	EOF
<i>G</i>	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Termo	—	—	—	—	5	5	5	—	—
Termo'	8	8	6	7	—	—	—	8	8
Fator	—	—	—	—	10	9	11	—	—

Linha do slide anterior



Parser Genérico

```
token ← GetToken()           // Inicialização com lookahead
push EOF                       // e pilha de parsing
push S
TOP ← topo da pilha

loop
  if TOP = EOF and token = EOF then
    break com sucesso // sai
  else if TOP é terminal then
    if TOP = token then
      pop                       // reconheceu TOP
      token ← GetToken()
    else break com erro procurando TOP
  else                          // TOP é não-terminal
    if TABELA[TOP,token] é  $A \rightarrow B_1 B_2 \dots B_k$  then
      pop                       // se livra de A
      push  $B_k, B_{k-1}, \dots, B_1$  // nessa ordem!
    else break com erro expandindo TOP
  TOP ← topo da pilha
```



Construção de Parsers LL(1)

Construindo a tabela completa

- Uma linha para cada NT e uma coluna pra cada T
- Um interpretador para a tabela
- Um algoritmo para construir a tabela

Preenchendo TABELA[X,y], $X \in NT$, $y \in T$

1. entrada é regra $X \rightarrow \beta$, se $y \in \text{FIRST}^+(X \rightarrow \beta)$
2. entrada é **error** caso contrário

Se uma entrada tem mais de uma regra então G não é LL(1)

Esse é o algoritmo de construção de tabelas LL(1)