MAB 471
2012.1

Análise Sintática Bottom-up

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

Recapitulando parsers top-down

- Constróem árvore sintática da raiz até as folhas
- Recursão à esquerda faz parsers entrarem em loop
  - Transformações eliminam recursão à esquerda em alguns casos
- Conjuntos FIRST, FIRST$^+$, FOLLOW e condição LL(1)
  - Condição LL(1) implica que a gramática funciona com um parser preditivo
  - Fatoração à esquerda transforma algumas gramáticas não-LL(1) em LL(1)
- Dada uma gramática LL(1), podemos
  - Escrever um parser recursivo
  - Construir uma tabela para um parser LL(1) genérico
- Parser LL(1) não precisa construir uma árvore
  - Podemos gerar código final diretamente

Análise bottom-up (definições)

O objetivo da análise é construir uma derivação

Uma derivação é uma sequência de passos de reescrita

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

- Cada γ_i é uma forma sentencial
  - Se γ só tem terminais então γ é uma sentença de $L(G)$
  - Se γ tem 1 ou mais não-terminais então γ é uma forma sentencial
- Para obter γ_i de γ_{i-1} reescreva algum NT $A \in \gamma_{i-1}$ usando $A \rightarrow \beta \in P$
  - Troque uma ocorrência de A em γ_{i-1} por β para obter γ_i
  - Em uma derivação mais à esquerda use a primeira ocorrência, em uma mais à direita a última

Parsers bottom-up constróem uma derivação mais à direita

Análise bottom-up (definições)

Um parser bottom-up constrói uma derivação começando da sentença de entrada de volta para o símbolo inicial S

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

bottom-up

Para **reduzir** γ_i para γ_{i-1} case algum lado direito β com γ_i então substitua β com seu lado esquerdo correspondente, **A**. (assumindo produção $A \rightarrow \beta$)

Vendo em termos de árvore sintática estamos indo das folhas para a raiz

- Nós sem um pai na árvore parcial são a **franja superior**
- Cada substituição de β por A encolhe a franja superior, por isso é uma redução.
- A derivação é de trás pra frente, mas a entrada ainda é processada da esquerda para a direita

Encontrando Reduções

Seja a gramática

0	G	→	a A B e
1	A	→	A b c
2		|	b
3	B	→	d

E a entrada abbcde

Forma Sentencial	Próx. redução Prod	Pos
abbcde	2	2
a A bcde	1	4
a A de	3	3
a A B e	0	4
G	—	—

O truque é ler a entrada para achar a próxima redução

O mecanismo tem que ser eficiente

Encontrando Reduções

Seja a gramática

0	G	→	a A B e
1	A	→	A b c
2		|	b
3	B	→	d

E a entrada abbcde

Forma Sentencial	Próx. redução Prod	Pos
abbcde	2	2
a A bcde	1	4
a A de	3	3
a A B e	0	4
G	—	—

O truque é ler a entrada para achar a próxima redução

O mecanismo tem que ser eficiente

"Pos" é onde o canto direito de β ocorre na forma sentencial

Encontrando Reduções

Seja a gramática

0	G	→	a A B e
1	A	→	A b c
2		|	b
3	B	→	d

E a entrada abbcde

Forma Sentencial	Próx. redução Prod	Pos
abbcde	2	2
a A bcde	1	4
a A de	3	3
a A B e	0	4
G	—	—

O truque é ler a entrada para achar a próxima redução

O mecanismo tem que ser eficiente

"Pos" é onde o canto direito de β ocorre na forma sentencial

O processo de achar a próxima redução parece advinhação, mas pode ser automatizado de um jeito eficiente para muitas gramáticas

Encontrando Reduções (Handles)

O parser deve encontrar uma substring β da franja superior que casa alguma produção $A \rightarrow \beta$ usada em um passo da derivação mais à direita

A substring β é um handle

Formalmente,

Um handle de uma forma sentencial à dreita γ é um par $\langle A \rightarrow \beta, k \rangle$ onde $A \rightarrow \beta \in P$ e k é a posição em γ do canto direito de β.

Se $\langle A \rightarrow \beta, k \rangle$ é um handle então substituir β em γ por A produz a forma sentencial à direita da qual γ é derivada.

Como γ é uma forma sentencial à direita, a substring à direita do handle só pode ter terminais

$\Rightarrow$ o parser não vai precisar ler (muito) após o handle

Exemplo

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

Gramática de expressões recursiva à esquerda

Parsers bottom-up funcionam com gramáticas recursivas à esquerda (ou à direita).

Damos preferência para recursão à esquerda pela facilidade para construir árvores com associatividade à esquerda.

Exemplo

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

Gramática de expressões recursiva à esquerda

derivação

Prod	Forma sentencial
—	G
0	Expr
2	Expr - Termo
4	Expr - Termo * Fator
8	Expr - Termo * <id,y>
6	Expr - Fator * <id,y>
7	Expr - <num,2> * <id,y>
3	Termo - <num,2> * <id,y>
6	Fator - <num,2> * <id,y>
8	<id,x> - <num,2> * <id,y>

Derivação à direita de x - 2 * y

Exemplo

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

Gramática de expressões recursiva à esquerda

Prod	Forma sentencial	Handle
—	G	—
0	Expr	0,1
2	Expr - Termo	2,3
4	Expr - Termo * Fator	4,5
8	Expr - Termo * <id,y>	8,5
6	Expr - Fator * <id,y>	6,3
7	Expr - <num,2> * <id,y>	7,3
3	Termo - <num,2> * <id,y>	3,1
6	Fator - <num,2> * <id,y>	6,1
8	<id,x> - <num,2> * <id,y>	8,1

parse

Handles para derivação à direita de x - 2 * y

Mais Handles

Parsers bottom-up acham a deriv. mais à direita de trás pra frente

- Processa entrada da esquerda pra direita
  - Franja superior da árvore parcial está em (NT |T)* T*
  - O handle sempre aparece com o canto direito na junção entre (NT | T)* e T* (o foco da análise LR)
  - Podemos manter o prefixo que precisamos da franja superior em uma pilha
  - A pilha substitui a informação da posição
- Handles aparecem no topo da pilha
- Toda a informação pra decidir se reduz o handle está no foco
  - Próxima palavra da entrada
  - O símbolo mais à direita na franja e seus vizinhos imediatos
  - Em um parser LR, informação adicional na forma de um "estado"

Handles são Únicos

Teorema:

> Se G não é ambígua então cada forma sentencial à direita tem um handle **único**.

Esboço de prova:

1. G não é ambígua $\Rightarrow$ derivação mais à direita é única
2. $\Rightarrow$ uma única produção $A \rightarrow \beta$ usada para derivar γ_i de γ_{i-1}
3. $\Rightarrow$ uma única posição ***k*** onde $A \rightarrow \beta$ é usada
4. $\Rightarrow$ um único handle $\langle A \rightarrow \beta, k \rangle$

Segue direto das definições

Se acharmos os handles podemos construir uma derivação

Handles são Únicos

Teorema:

Se G não é ambígua então cada forma sentencial à direita tem um handle **único**.

Esboço de prova:

1. G não é ambígua $\Rightarrow$ derivação mais à direita é única
2. $\Rightarrow$ uma única produção $A \rightarrow \beta$ usada para derivar γ_i de γ_{i-1}
3. $\Rightarrow$ uma única posição ***k*** onde $A \rightarrow \beta$ é usada
4. $\Rightarrow$ um único handle $\langle A \rightarrow \beta, k \rangle$

Segue direto das definições

Se acharmos os handles podemos construir uma derivação

O handle sempre aparece com o canto direito no topo da pilha.

→ Quantas produções o parser deve considerar?

Parser Shift-Reduce

Para implementar um parser bottom-up usamos a técnica **shift-reduce**

Um parser shift-reduce é um autômato de pilha com quatro ações

- Shift — próxima palavra é empilhada
- Reduce — canto direito de handle está no topo da pilha
  - Ache canto esquerdo na pilha
  - Desempilhe handle e empilhe NT apropriado
- Accept — pare e reporte sucesso
- Error — chame uma rotina de aviso/recuperação de erros

Accept e Error são simples

Shift é só um push e uma chamada ao scanner

Reduce faz |rhs| pops e um push

Parser Shift-Reduce

Para implementar um parser bottom-up usamos a técnica **shift-reduce**

Um parser shift-reduce é um autômato de pilha com quatro ações

- Shift — próxima palavra é empilhada
- Reduce — canto direito de handle está no topo da pilha
  - Ache canto esquerdo na pilha
  - Desempilhe handle e empilhe NT apropriado
- Accept — pare e reporte sucesso
- Error — chame uma rotina de aviso/recuperação de erros

Accept e Error são simples

Shift é só um push e uma chamada ao scanner

Reduce faz |rhs| pops e um push

Mas como o parser sabe quando fazer shift e quando fazer reduce?
Faz shift até ter um handle no topo da pilha.

Parser Shift-Reduce

Um parser shift-reduce simples:

```
push ERRO
token ← next_token( )
repeat until (topo = G and token = EOF)
    if topo é handle A→β
        then      // reduce β a A
            pop |β| símbolos
            push A
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
         else     // acabou a entrada!
            reporta erro
```

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id		

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id	8,1	reduce 8
$ Fator	- num * id	6,1	reduce 6
$ Termo	- num * id	3,1	reduce 3
$ Expr	- num * id		

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id	8,1	reduce 8
$ Fator	- num * id	6,1	reduce 6
$ Termo	- num * id	3,1	reduce 4
$ Expr	- num * id		

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

Expr não é um handle nesse ponto pois não aparece nesse ponto na derivação.

Isso parece advinhação, mas na verdade essa decisão pode ser automatizada de maneira eficiente.

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id	8,1	reduce 8
$ Fator	- num * id	6,1	reduce 6
$ Termo	- num * id	3,1	reduce 3
$ Expr	- num * id	nenhum	shift
$ Expr -	num * id	nenhum	shift
$ Expr - num	* id		

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id	8,1	reduce 8
$ Fator	- num * id	6,1	reduce 6
$ Termo	- num * id	3,1	reduce 3
$ Expr	- num * id	nenhum	shift
$ Expr -	num * id	nenhum	shift
$ Expr - num	* id	7,3	reduce 7
$ Expr - Fator	* id	6,3	reduce 6
$ Expr - Termo	* id		

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id	8,1	reduce 8
$ Fator	- num * id	6,1	reduce 6
$ Termo	- num * id	3,1	reduce 3
$ Expr	- num * id	nenhum	shift
$ Expr -	num * id	nenhum	shift
$ Expr - num	* id	7,3	reduce 7
$ Expr - Fator	* id	6,3	reduce 6
$ Expr - Termo	* id	nenhum	shift
$ Expr - Termo *	id	nenhum	shift
$ Expr - Termo * id			

0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Handle	Ação
$	id - num * id	nenhum	shift
$ id	- num * id	8,1	reduce 8
$ Fator	- num * id	6,1	reduce 6
$ Termo	- num * id	3,1	reduce 3
$ Expr	- num * id	nenhum	shift
$ Expr -	num * id	nenhum	shift
$ Expr - num	* id	7,3	reduce 7
$ Expr - Fator	* id	6,3	reduce 6
$ Expr - Termo	* id	nenhum	shift
$ Expr - Termo *	id	nenhum	shift
$ Expr - Termo * id		8,5	reduce 8
$ Expr - Termo * Fator		4,5	reduce 4
$ Expr - Termo		2,3	reduce 2
$ Expr		0,1	reduce 0
$ G		nenhum	accept

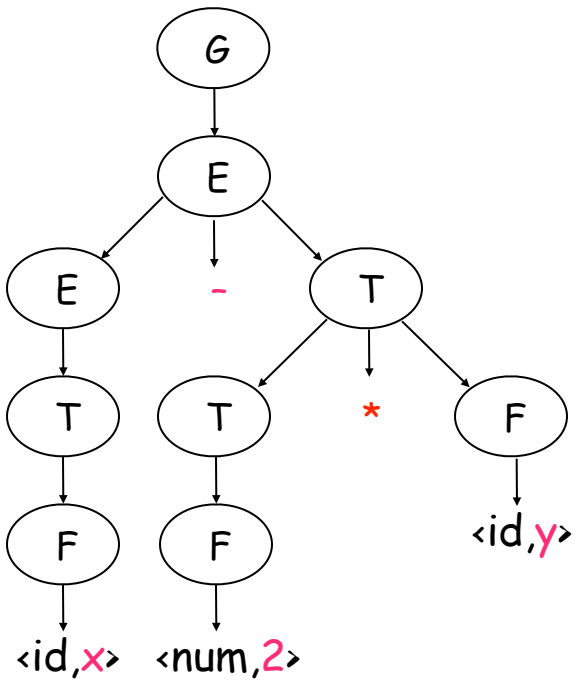
0	G	→	Expr
1	Expr	→	Expr + Termo
2		|	Expr - Termo
3		|	Termo
4	Termo	→	Termo * Fator
5		|	Termo / Fator
6		|	Fator
7	Fator	→	num
8		|	id
9		|	(Expr)

5 shifts +
9 reduces +
1 accept

1. Shift até o topo da pilha ser canto direito de handle
2. Achar o canto esquerdo do handle e reduzir

De volta a x - 2 * y

Pilha	Entrada	Ação
$	id - num * id	shift
$ id	- num * id	reduce 8
$ Fator	- num * id	reduce 6
$ Termo	- num * id	reduce 3
$ Expr	- num * id	shift
$ Expr -	num * id	shift
$ Expr - num	* id	reduce 7
$ Expr - Fator	* id	reduce 6
$ Expr - Termo	* id	shift
$ Expr - Termo *	id	shift
$ Expr - Termo * id		reduce 8
$ Expr - Termo * Fator		reduce 4
$ Expr - Termo		reduce 2
$ Expr		reduce 0
$ G		accept



Árvore correspondente

Uma Lição sobre Handles

Um handle deve ser uma substring de uma forma sentencial γ tal que:

- Case com o lado direito β de alguma regra $A \rightarrow \beta$; e
- Exista alguma derivação mais à direita a partir do símbolo inicial que produz a forma sentencial γ com $A \rightarrow \beta$ como a última produção usada

- Simplesmente procurar por regras com lados direitos que casem com o topo da pilha não é o bastante
- Ponto Fundamental: Como sabemos que achamos um handle sem primeiro gerar um monte de derivações?
  - Resposta: usamos a informação do que já vimos na entrada (contexto esquerdo), codificada na forma sentencial e em um "estado" do parser, e a próxima palavra da entrada, o "lookahead". (1 palavra além do handle.)
  - Estados derivados por análise da gramática
  - Codificamos tudo isso em um DFA para reconhecer handles

Uma Lição sobre Handles

Um handle deve ser uma substring de uma forma sentencial γ tal que:

- Case com o lado direito β de alguma regra $A \rightarrow \beta$; e
- Exista alguma derivação mais à direita a partir do símbolo inicial que produz a forma sentencial γ com $A \rightarrow \beta$ como a última produção usada

- Simplesmente procurar por regras com lados direitos que casem com o topo da pilha não é o bastante
- Ponto Fundamental: Como sabemos que achamos um handle sem primeiro gerar um monte de derivações?
  - Resposta: usamos a informação do que já vimos na entrada (contexto esquerdo), codificada na forma sentencial e em um "estado" do parser, e a próxima palavra da entrada, o "lookahead". (1 palavra além do handle.)
  - Estados derivados por análise da gramática
  - Codificamos tudo isso em um DFA para reconhecer handles

O uso do contexto esquerdo é precisamente a razão pela qual gramáticas LR(1) são mais poderosas que gramáticas LL(1)

Parsers LR(1)

- Parsers LR(1) são parsers shift-reduce de tabela que usam um token de lookahead
- A classe de gramáticas que esses parsers reconhecem é chamada de gramáticas LR(1)

Definição informal:

Uma gramática é LR(1) se, dada uma derivação mais à direita

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

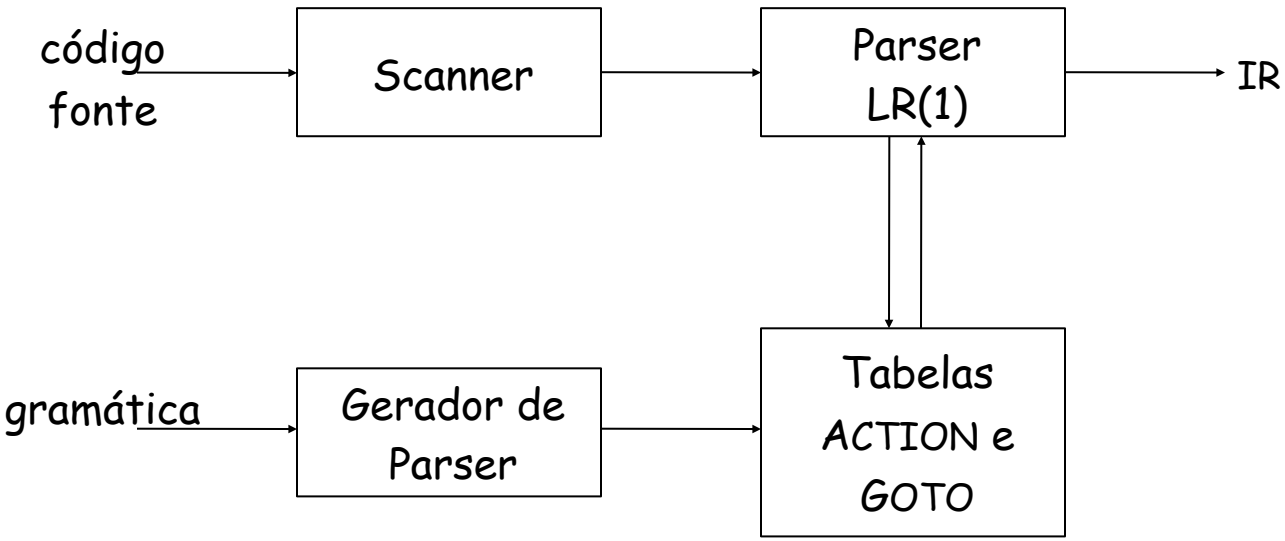
Podemos

1. isolar o handle de cada forma sentencial γ_i, e
2. determinar a produção pela qual reduzir,

varrendo γ_i da esquerda para a direita, indo no máximo 1 símbolo além do final do handle de γ_i

Parsers LR(1)

Estrutura de um parser LR(1):



Tabelas podem ser feitas à mão, mas o processo é tedioso

Facilmente automatizável, muitas implementações disponíveis

Parser LR(1)

```
push ERRO
push s0                          // estado inicial
token ← scanner.next_token();
loop {
        s ← topo
        if ( ACTION[s,token] == "reduce A→β" ) then {
            pop 2*|β|       // desempilha 2*|β| símbolos
            s ← topo
            push A
            push GOTO[s,A]  // empilha próx. estado
        }
        else if ( ACTION[s,token] == "shift si" ) then {
                push token; push si
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "accept" & token == EOF )
                then break;
        else throw "erro de sintaxe";
}
reporta sucesso
```

O parser genérico

- usa uma pilha e um scanner
- usa duas tabelas, chamadas ACTION e GOTO

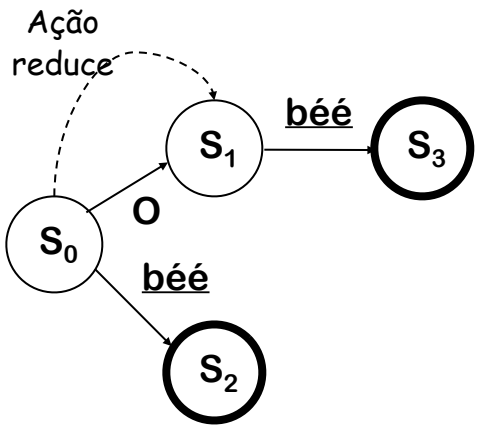
  ACTION: estado x token → estado

  GOTO: estado x NT → estado
- shift |entrada| vezes
- reduz |derivação| vezes
- aceita no máximo uma vez
- detecta erros por falha dos outros três casos
- segue esquema básico do parser shift-reduce

Parsers LR(1)

Como o LR(1) funciona?

- Gramática não ambígua ⇒ derivação mais à direita única
- Manter franja superior na pilha
  - Todos os handles ativos incluem o topo da pilha
  - Empilhar entradas até topo ser canto direito de handle
- Linguagem dos handles é regular (finita)
  - DFA reconhecedor de handles
  - Tabelas ACTION & GOTO codificam DFA
- Para casar subtermo, faça a transição
  e deixe estados antigos na pilha
- Estado final no DFA ⇒ ação reduce
  - Novo estado é GOTO[estado no topo, NT]



Linguagem de Parênteses

Parênteses balanceados

- Além do poder de REs
- Mostra papel do contexto

0	G	→	Lista
1	Lista	→	Lista Par
2		|	Par
3	Par	→	(Par)
4		|	( )

Linguagem de Parênteses

ACTION			
Estado	eof	(	)
0		S 3	
1	acc	S 3	
2	R 2	R 2	
3		S 6	S 7
4	R 1	R 1	
5			S 8
6		S 6	S 10
7	R 4	R 4	
8	R 3	R 3	
9			S 11
10			R 4
11			R 3

GOTO		
Estado	Lista	Par
0	1	2
1		4
2		
3		5
4		
5		
6		9
7		
8		
9		
10		
11		

0	G	→	Lista
1	Lista	→	Lista Par
2		|	Par
3	Par	→	(Par)
4		|	( )

Linguagem de Parênteses

Estado	Lookahead	Pilha	Handle	Ação
—	(	$ 0	—nenhum—	—
0	(	$ 0	—nenhum—	shift 3
3	)	$ 0 (3	—nenhum—	shift 7
7	eof	$ 0 (3) 7	( )	reduce 4
2	eof	$ 0 Par 2	Par	reduce 2
1	eof	$ 0 Lista 1	Lista	accept

Analisando "( )"

0	G	→	Lista
1	Lista	→	Lista Par
2		|	Par
3	Par	→	(Par)
4		|	( )

Linguagem de Parênteses

0	G	→	Lista	
1	Lista	→	Lista Par	
2		|	Par	
3	Par	→	(Par)	
4		|	( )	

Analisando "(( )) ( )"

Est.	LA	Pilha	Handle	Ação
—	(	$ 0	—nada—	—
0	(	$ 0	—nada—	shift 3
3	(	$ 0 (3	—nada—	shift 6
6	)	$ 0 (3 (6	—nada—	shift 10
10	)	$ 0 (3 (6) 10	( )	reduce 4
5	)	$ 0 (3 Par 5	—nada—	shift 8
8	(	$ 0 (3 Par 5) 8	(Par)	reduce 3
2	(	$ 0 Par 2	Par	reduce 2
1	(	$ 0 Lista 1	—nada—	shift 3
3	)	$ 0 Lista 1 (3	—nada—	shift 7
7	eof	$ 0 Lista 1 (3) 7	( )	reduce 4
4	eof	$ 0 Lista 1 Par 4	Lista Par	reduce 1
1	eof	$ 0 Lista 1	Lista	accept

Linguagem de Parênteses

0	G	→	Lista	
1	Lista	→	Lista Par	
2		|	Par	
3	Par	→	(Par)	
4		|	( )	

Est.	LA	Pilha	Handle	Ação
—	(	$ 0	—nada—	—
0	(	$ 0	—nada—	shift 3
3	(	$ 0 (3	—nada—	shift 6
6	)	$ 0 (3 (6	—nada—	shift 10
10	)	$ 0 (3 (6) 10	( )	reduce 4
5	)	$ 0 (3 Par 5	—nada—	shift 8
8	(	$ 0 (3 Par 5) 8	(Par)	reduce 3
2	(	$ 0 Par 2	Par	reduce 2
1	(	$ 0 Lista 1	—nada—	shift 3
3	)	$ 0 Lista 1 (3	—nada—	shift 7
7	eof	$ 0 Lista 1 (3) 7	( )	reduce 4
4	eof	$ 0 Lista 1 Par 4	Lista Par	reduce 1
1	eof	$ 0 Lista 1	Lista	accept

Analisando "(( )) ( )"

Vamos ver como o parser reduz "( )"

Linguagem de Parênteses

Est.	Lookahead	Pilha	Handle	Ação
—	(	$ 0	—nada—	—
0	(	$ 0	—nada—	shift 3
3	)	$ 0 (3	—nada—	shift 7
7	eof	$ 0 (3) 7	( )	reduce4
2	eof	$ 0 Par 2	Par	reduce 2
1	eof	$ 0 Lista 1	Lista	accept

Analisando "( )"

Aqui, retirar ( ) da pilha revela s_0.

Goto(s_0,Par) é s_2.

0	G	→	Lista
1	Lista	→	Lista Par
2		|	Par
3	Par	→	(Par)
4		|	( )

Linguagem de Parênteses

0	G	→	Lista	
1	Lista	→	Lista Par	
2		|	Par	
3	Par	→	(Par)	
4		|	( )	

Analisando "(( )) ( )"

Est.	LA	Pilha	Handle	Ação
—	(	$ 0	—nada—	—
0	(	$ 0	—nada—	shift 3
3	(	$ 0 (3	—nada—	shift 6
6	)	$ 0 (3 (6	—nada—	shift 10
10	)	$ 0 (3 (6) 10	( )	reduce 4
5	)	$ 0 (3 Par 5	—nada—	shift 8
8	(	$ 0 (3 Par 5) 8	(Par)	reduce 3
2	(			
1	(			
3	)			
7	eof			
4	eof			
1	eof			

Aqui, retirando ( ) da pilha revela s_3, que representa o contexto esquerdo de um (não casdo.

Goto(s_3,Par) é s_5, um estado em que esperamos um). Esse caminho leva a uma redução pela produção 3, Par → (Par) .

Linguagem de Parênteses

0	G	→	Lista	
1	Lista	→	Lista Par	
2		|	Par	
3	Par	→	(Par)	
4		|	( )	

Est.	LA	Pilha	Handle	Ação
—	(	$ 0	—nada—	—
0	(	$ 0	—nada—	shift 3
3				ift 6
6				ft 10
10				uce 4
5				ift 8
8				uce 3
2				uce 2
1	(	$ 0 Lista 1	—nada—	shift 3
3	)	$ 0 Lista 1 (3	—nada—	shift 7
7	eof	$ 0 Lista 1 (3) 7	( )	reduce 4
4	eof	$ 0 Lista 1 Par 4	Lista Par	reduce 0
1	eof	$ 0 Lista 1	Lista	accept

Aqui, tirando ( ) da pilha revela s_1, que representa o contexto esquerdo de uma Lista previamente reconhecida.

Goto(s_1,Par) é s_4, um estado em que podemos reduzir Lista Par para Lista (em um lookahead de (ou eof).

Analisando "(( )) ( )"

Três cópias de "reduce 4" com contextos diferentes — produzem três comportamentos distintos