Disciplina: Compiladores
Análise Sintática - Continuação

Prof. Flávio Márcio de Moraes e Silva

Parsing Bottom-up

- Construir a árvore de parse a partir das folhas em direção a raiz
- Método Shift-Reduce (deslocar-reduzir):
  - “reduzir” o lado direito de uma produção pelo lado esquerdo até que se chegue na raiz.
  - se **o string for escolhido corretamente** teremos traçado uma derivação mais a direita ao reverso.
- Operam sobre gramáticas LR(k): (Left to right) análise da sentença da esquerda para a direita, (Rigthmost derivation) produzindo uma derivação mais a direita, (k) levando em conta “k” símbolos.

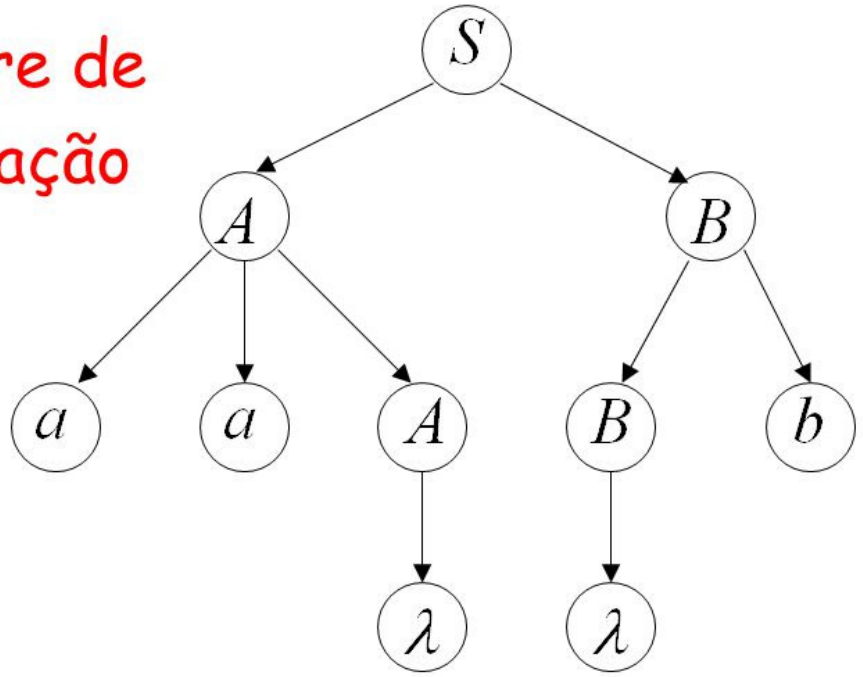
Árvore de derivação

- Notação que exibe uma derivação no formato de árvore.

$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Árvore de derivação



24

Parsing Bottom-up

- Exemplo
- S -> aABe
- A -> Abc | b
- B -> d

- podemos reduzir a sentença **abbcde** para S:
- abbcde A => b
- aAbcde A => Abc
- aAde B => d
- aABe S => aABe
- S

Handle

- Informalmente, um handle de um string:
  - é um substring que casa com o lado direito de uma produção e sua redução representa um passo na derivação mais a direita ao reverso (ddr).
- Formalmente, um handle de uma forma sentencial mais a direita **y** é uma produção *A* -> w e uma posição de **y** onde "w" pode ser substituido por *A* produzindo um passo na ddr.
- Note:
- abbcde A => b é um handle
- aAbcde A => Abc é um handle
- mas se usássemos:
- aAAcde A => b não é um handle

Handle

- Note que: uma gramática ambígua pode ter mais de um handle para uma forma sentencial e uma gramática não ambígua apenas um.
- Dado:
- (1) E -> E + E
- (2) E -> E * E
- (3) E -> (E)
- (4) E -> id
- e o string **id + id * id** temos:

id + id * id	=>	E + id * id	E + id * id
	=>	E + E * id	E + E * id
	=>	E + E * E	E * id
	=>	E + E	E * E
	=>	E	E

Handle Pruning

- Sendo z, y e w formas sentencias da gramática em questão.
- Um handle pruning é um processo de redução de “y” para *A* em “zyw” dado o handle *A* -> y. Pode ser visto como uma poda na árvore de parse, onde se remove os filhos de *A* da árvore.

Implementação usando pilha

- Estruturas:
  - pilha: armazena os símbolos da gramática,
  - buffer de entrada: armazena o string **w** de entrada,
  - $: um símbolo usado como marca especial.
- Exemplo: Pilha Entrada
- $ w$
- Operações:
  - deslocar zero ou mais elementos para a pilha até que se encontre um handle “b” no topo da mesma;
  - reduzir “b” para o lado esquerdo da produção;
  - repetir o ciclo até que: pilha contenha o símbolo inicial e a entrada esteja vazia ou um erro tenha sido detectado.

Exemplo

PILHA	ENTRADA	AÇÃO
$	id + id * id$	shift
$id	+ id * id$	reduce: E -> id
$E	+ id * id$	shift
$E +	id * id$	shift
$E + id	* id$	reduce: E -> id
$E + E	* id$	shift
$E + E *	id$	shift
$E + E * id	$	reduce: E -> id
$E + E * E	$	reduce: E -> E * E
$E + E	$	reduce: E -> E + E
$E	$	accept

Conflitos (shift/reduce)

- Existem gramáticas que não podem ser analisadas por um parser shift-reduce:
  - conflitos shift/reduce
  - conflitos reduce/reduce
- Obs: estas gramáticas não fazem parte da classe LR(k)
- Exemplo:
- *stmt* -> **if** *expr* **then** *stmt*
- | **if** *expr* **then** *stmt* **else** *stmt*
- | **other**
- Pilha Entrada
- .... if *expr* then *stmt* else ... $
- conflito shift/reduce
- **O que fazer?**
- Note que: a gramática é ambígua. Podemos resolver o conflito em favor do shift (deslocar/empilhar). Casar “else” com o “if” mais próximo.

Conflitos (reduce/reduce)

- Suponha que o analisador léxico retorne a token **id** para identificadores como: variáveis, procedimentos, vetores etc. Considere ainda a seguinte gramática:
- *stmt* -> **id** (*parameter_list*) | *expr* := *expr*
- *parameter_list* -> *parameter_list* , *parameter* | *parameter*
- *parameter* -> id
- *expr* -> id (*expr_list*) | id
-
- **A(i, j) será visto pelo parser como: id (id, id)**
- Note: Pilha Entrada
- ... id (id ... , id) ...
- Conflito reduce/reduce
-
- **É evidente a operação de redução em id.**
- **Mas qual produção usar?** Parameter -> id ou expr -> id

Solução (reduce/reduce)

- Alterar a gramática:
- *stmt* -> proc**id** (*parameter_list*)
- *parameter* -> id
- *expr* -> id (*expr_list*) | id
- Alteração implica em um analisador léxico mais sofisticado: sempre consultar a tabela de símbolos antes de retornar uma token.
- **Se A(i, j) é visto pelo parser como: id (id, id)**
- teremos: Pilha Entrada
- ... id (**id** ... , id) ...
- reduzindo **id** para *expr*.
-
- **Se A(i, j) é visto pelo parser como: procid (id, id)**
- teremos: Pilha Entrada
- ... procid (**id** ... , id) ...
- reduzindo **id** para *parameter*.

Parsing de precedência operadora

- Operam sobre a classe das gramáticas de operadores:
  - Nenhuma produção gera "lambida".
  - Em nenhuma produção encontramos duas variáveis não terminais adjacentes.

- Exemplo
- E -> EAE | (E) | -E | id
- A -> + | - | * | / | ^
-
- não é uma gramática operadora mas a gramática abaixo é:
-
- E -> E+E | E-E | E*E | E/E | E^E | (E) | -E | id

Parsing de precedência operadora

- Para uma gramática operadora podemos facilmente implementar ("a mão") um parser shift-reduce eficiente:
- Definimos 3 operações de precedência **<** , = , **>** entre pares de terminal onde:
- a **<** b : **a** tem menor precedência que **b**
- a = b : **a** tem a mesma precedência que **b**
- a **>** b : **a** tem maior precedência que **b**

- A seguir um exemplo para a gramática abaixo:
- E -> E + E
- E -> E * E
- E -> id

Usando as relações de precedência

- O objetivo das relações de precedência é delimitar o handle de uma forma sentencial a direita.
- Considere **$ < a** e **a > $** para todo terminal **a** e a seguinte tabela de precedência:

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

- Note que:
- se * tem maior precedência que + então: *** > +** e **+ < ***
- a entrada para **id** e **id** é vazia (nunca dois ids ocorrerão em seqüência)

Usando as relações de precedência

- Um handle pode ser encontrado da seguinte forma:
- Percorra o string da esquerda para direita até que um **>** seja encontrado.
- Volte até que um **<** seja encontrado.
- Tudo que estiver entre **<** e **>** é parte do handle
- Considerando o string **id + id * id,** o string contendo a precedência operadora é:
- **$ < id > + < id > * < id > $**
- O 1º handle encontrado é **id** chegando a E+id*id.
- Isto nos leva a: **$ < + < id > * < id > $.**
- O 2º handle encontrado é **id** chegando a E+E*id.
- Isto nos leva a: **$ < + < * < id > $.**
- O 3º handle encontrado é **id** chegando a E+E*E.
- Isto nos leva a: **$ < + < * > $.**

Usando as relações de precedência

- **$ < + < * > $**
- **$ < + > $**
- **$ $**
- Observações:
  - Símbolos não terminais não influenciam o parser logo não precisamos mante-los na pilha;
  - Toda vez que uma relação de **<** ou = for encontrada entre o terminal no topo da pilha e o terminal na entrada uma operação de shift será efetuada;
  - Se uma relação de **>** for encontrada uma operação de redução será efetuada;
  - Se não existir uma relação entre os mesmos então um erro foi detectado.

Algoritmo

- Entrada: String w e a tabela de precedência
- *ip* é um ponteiro para o primeiro símbolo de w$;
- repita
- if $ esta no no topo da pilha e *ip* aponta para $
- then return “sentença reconhecida!!!”
- else begin
- sendo “*a*” o topo da pilha e “*b*” é o símbolo apontado por *ip*
- if *a* **<** *b* or *a* **=** *b* then begin
- empilhe *b*;
- e avance *ip* para o próximo símbolo;
- end;
- else if *a* **>** *b* then
- repeat
- desempilhe
- até que o último desempilhado **>** terminal do topo pilha
- else erro( )
- end

Exercício

- Mostre os movimentos do parser abaixo para reconhecer a seguinte entrada: $ id * id + id * id $

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

Resolução

- $ < id > * < id > + < id > * < id > $
- $ < * < id > + < id > * < id > $
- $ < * > + < id > * < id > $
- $ < + < id > * < id > $
- $ < + < * < id > $
- $ < + < * > $
- $ + $

Método mecânico

- Obter as relações de precedência diretamente a partir da gramática analisada, mas esta precisa ser não ambígua.

- Para cada dois terminais “a” e “b” quaisquer e um não terminal “X” qualquer, presentes no lado direito das regras. Teremos:

- (1) a = b: para regras do lado direito, onde os terminais “a” e “b” aparecem adjacentes ou separados por um único não terminal “X”. Exemplo: S -> ab | aXb
- (2) a < primeiros(X): para regras do lado direito, onde um terminal “a” aparece seguido de um não terminal “X”. Exemplo: S -> aX
- (3) últimos(X) > b: para regras do lado direito, onde um não terminal “X” aparece seguido de um terminal “b”. Exemplo: S -> Xb

Exemplo – Método mecânico

- E -> E + T | T
- T -> T * F | F
- F -> P ^ F | P
- P -> id | (E)

Exemplos de derivações:
E => E + T
E => T => T * F
E => T => F => P ^ F
E => T => F => P => id
E => T => F => P => (E)

- Seguir os passos:
- 1) Para cada não terminal determinar os terminais que podem ocorrer como primeiro e como último em alguma forma sentencial derivada a partir dele. Olhar para a gramática.

	Primeiros	Últimos
E	+, *, ^, (, id	+, *, ^,), id
T	*, ^, (, id	*, ^,), id
F	^, (, id	^,), id
P	(, id	), id

Outro Exemplo (Primeiros / Últimos)

S -> A a B b C
A -> x A | x
B -> lambida
C -> y

S => A a B b C => x A a B b C => x x a B b C => x x a b C => x x a b y

	Primeiros	Últimos
S	a, x	b, y
A	x	x
B		
C	y	y

Exemplo – Método mecânico

- 2) para computar <, procurar pares aX (terminal, não terminal) nos lados direitos de produção. Tem-se que “a” tem menor precedência que qualquer “primeiro terminal” derivado a partir de X.
- Pares: +T, *F, ^F, (E
- Relações:
  - \+ < { *, ^, (, id } Primeiros de T
  - * < { ^, (, id } Primeiros de F
  - ^ < { ^, (, id } Primeiros de F
  - (< { +, *, ^, (, id } Primeiros de E

Exemplo – Método mecânico

- 3) para computar >, procurar pares Xb (não terminal, terminal) nos lados direitos de produção. Qualquer “último terminal” derivado de X tem precedência maior do que “b”.
- Pares: E+, T*, P^, E)
- Relações:
  - Últimos de E { +, *, ^,), id } > +
  - Últimos de T { *, ^,), id } > *
  - Últimos de P {), id } > ^
  - Últimos de E { +, *, ^,), id } >)

Exemplo – Método mecânico

- 4) Para computar =, examinar os lados direitos de produção procurando por formas aZb onde “a” e “b” são terminais e Z é lambida ou não terminal.
  - A única ocorrência desta forma de produção é P -> (E)
  - Portando (=)

- 5) $ tem precedência menor do que todos os “primeiros terminais” derivados a partir do símbolo inicial:
  - $ < { +, *, ^, (, id }

- 6) Todos os “últimos terminais” derivados a partir do símbolo inicial são > do que $:
  - { +, *, ^,), id } > $

Tabela de precedência resultante

	id	+	*	^	(	)	$
id		>	>	>		>	>
+	<	>	<	<	<	>	>
*	<	>	>	<	<	>	>
^	<	>	>	<	<	>	>
(	<	<	<	<	<	=	
)		>	>	>		>	>
$	<	<	<	<	<		

Exercício 1 (Tabela de precedência)

- Gramática ambígua:
- E -> E+E | E-E | E*E | E/E | E^E | (E) | ~E | id
- (gramática não ambígua equivalente, já com a precedência de operadores (~ e ^) > (* e /) > (+ e -)
- E -> E + T | E – T | T
- T -> T * F | T / F | F
- F -> P ^ F | ~ P | P
- P -> id | (E)

- (1) Usando o método mecânico construa a sua tabela de precedência para a gramática acima.
- (2) Mostre o reconhecimento do string ~ id + (id * id)

Resolução – Exercício 1

Exemplos de derivações:
E => E + T
E => E - T
E => T => T * F
E => T => T / F
E => T => F => P ^ F
E => T => F => ~ P
E => T => F => P => id
E => T => F => P => (E)

ab ou aXb: a = b
(E)

aX: a < Primeiros (X)
+T, -T, *F, /F, ^F, ~P, (E

Xb: Últimos(X) > b
E+, E-, T*, T/, P^, E)

$ < Primeiros(E)
Ultimos(E) > $

	Primeiros	Últimos
E	+, -, *, /, ^, ~, (, id	+, -, *, /, ^, ~,), id
T	*, /, ^, ~, (, id	*, /, ^, ~,), id
F	^, ~, (, id	^, ~,), id
P	(, id	), id

Tabela de precedência resultante

	+	-	*	/	^	~	(	)	id	$
+	>	>	<	<	<	<	<	>	<	>
-	>	>	<	<	<	<	<	>	<	>
*****	>	>	>	>	<	<	<	>	<	>
/	>	>	>	>	<	<	<	>	<	>
^	>	>	>	>	<	<	<	>	<	>
~	>	>	>	>			<	>	<	>
(	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>			>		>
id	>	>	>	>	>			>		>
$	<	<	<	<	<	<	<		<	

Ações do parser para: ~ id + (id * id)

- $ < ~ < id > + < (< id > * < id >) > $
- $ < ~ > + < (< id > * < id >) > $
- $ < + < (< id > * < id >) > $
- $ < + < (< * < id >) > $
- $ < + < (< * >) > $
- $ < + < (=) > $
- $ < + > $
- $ $

Funções de precedência

- Em geral, não é necessário armazenar a tabela de precedência para um parser: codificar a tabela através de 2 funções ***f*** e ***g*** que mapeiam os símbolos terminais em inteiros.
-
- As funções ***f*** e ***g*** são determinadas de tal forma que:
  - f (a) < g(b) sempre que a **<** b,
  - f (a) = g(b) sempre que a = b,
  - f (a) > g(b) sempre que a **>** b.
- Obs: Nem toda tabela de relação tem funções de precedência que a codifique.
- Entrada: Matriz de precedência operadora.
- Saída: Funções de precedência, ou uma indicação de sua inexistência.

Funções de precedência

- Vantagem: sendo “n” o número de terminais da gramática, enquanto a tabela de precedência ocupa espaço O(n*n) a tabela com as funções de precedência ocupa O(2*n).

- Na comparação de dois símbolos “a” e “b” usa-se a função “f” para o símbolo da pilha e “g” para o da entrada.
- Assim, a relação de precedência entre “a” e “b” é equivalente à relação numérica entre f(a) e g(b). Sempre existirá uma relação <, > ou = entre f(a) e g(b).
- As entradas de erro da matriz de precedência não terão representação.
- Os erros serão detectados quando, em reduções, os handles não forem encontrados na pilha.

Funções de precedência

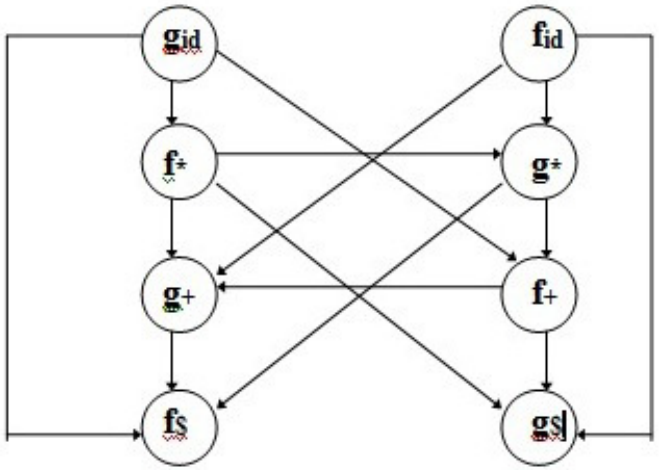
- Algoritmo:
- 1. Crie os símbolos f_i e g_i para cada i terminal ou $.
- 2. Particione os símbolos em grupos de tal forma que:
  - se a = b então f_a e g_b estão no mesmo grupo, note que se a = b e c = b então f_a e f_c estão no mesmo grupo (já que ambas estão no grupo de g_b).
- 3. Crie um grafo direcionado cujos nós são os grupos encontrados em (2). Se:
  - a <* b trace uma aresta do grupo g_b para f_a.
  - a *> b trace uma aresta do grupo f_a para g_b.
- 4. Se o grafo contiver ciclos, então as funções não existem. Caso contrário a função é calculada pela maior distância de f/g para as folhas.

Funções de precedência

Considere a seguinte matriz:

	id	+	*	$
id		*>	*>	*>
+	<*	*>	<*	*>
*	<*	*>	*>	*>
$	<*	<*	<*	

Como não existe a relação $\overset{\circledcirc}{=}$ cada símbolo gera um grupo próprio, logo:



	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Exercício 2 (Tabela + Funções)

- 1) Crie a tabela de precedência para a gramática abaixo:
- S -> ~S | (S) | S=>S | p | q

- Eliminando a ambiguidade:
- S -> S => T | T
- T -> ~P | P
- P -> p | q | (S)

- Obs: o símbolo “=>” é um único terminal

- 2) Depois crie a tabela com as funções de precedência.