Compiladores

Análise sintática (4)
Análise ascendente
Autômatos Empilhar/Reduzir

Lembrando: construir a tabela de análise LL(1)

- Como fazer ?
  - Re-escrever gramática para satisfazer condições de LL(1)
  - Calcular conjuntos First e Follow
  - Para cada produção A → α
    1. Para cada a ∈ First(α)
       - incluir A → α em M[A,a]
    2. Se ε ∈ First(α)
       - incluir A → α em M[A,b] para cada b em Follow(A)
    3. Se ε ∈ First(α) e $ ∈ Follow(A)
       - incluir A → α to M[A,$]
  - Todas entradas não definidas são erros

Plano da aula

- Transformação de Gramática
- Análise bottom-up (ascendente): princípios gerais
  - Vocabulário
  - Exemplos
- Analisador com pilha (reduz/empilha)
- Gramática LR
  - Tabelas LR.

Top-Down x Bottom Up

Gramática: S → A B
A → c | ε
B → cbB | ca

Entrada:
ccbca

Top-Down/Esquerda		Bottom-Up/Direita	
S ⇒ AB	S→AB	ccbca ⇐ Acbca	A→c
⇒ cB	A→c	⇐ AcbB	B→ca
⇒ ccbB	B→cbB	⇐ AB	B→cbB
⇒ ccbca	B→ca	⇐ S	S→AB

Redução – exemplo 1

S → aABe abbcde
A → Abc | b
B → d

Redução = substituição do lado direito de uma produção pelo não terminal correspondente (lado esquerdo)

Redução – exemplo 1

S → aABe abbcde
A → Abc | b aAbcde
B → d

Redução – exemplo 1

S → aABe
A → Abc | b
B → d

abbcde

handle = seqüência de símbolos do lado direito da produção, tais que suas reduções levam, no final, ao símbolo inicial da gramática

Redução – exemplo 1

S → aABe
A → Abc | b
B → d

abbcde
aAbcde

Redução – exemplo 1

S → aABe
A → Abc | b
B → d

abbcde
aAbcde
aAde

Redução – exemplo 1

S → aABe
A → Abc | b
B → d

abbcde
aAbcde
aAde
aABe

Redução – exemplo 1

S → aABe
A → Abc | b
B → d

abbcde
aAbcde
aAde
aABe
S

Redução – exemplo 2

E → E + E
E → E * E
E → (E) | id

id + id * id

Redução – exemplo 2

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E) \mid id$

id + id * id
E + id * id

Redução – exemplo 2

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E) \mid id$

id + id * id
E + id * id
E + E * id

Redução – exemplo 2

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E) \mid id$

id + id * id
E + id * id
E + E * id
E * id

Redução – exemplo 2

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E) \mid id$

id + id * id
E + id * id
E + E * id
E * id
E * E

Redução – exemplo 2

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E) \mid id$

id + id * id
E + id * id
E + E * id
E * id
E * E
E

Ações bottom-up (empilha-reduz)

- A análise Bottom-Up vai necessitar:
  - De uma pilha para guardar os símbolos
  - De um buffer de entrada para a sentença w a ser reconhecida.
- Operações lícitas:
  - empilha (shift):
    - coloca no topo da pilha o símbolo que está sendo lido e avança o cabeçote de leitura na string
  - reduz (reduce):
    - substitui o handle no topo da pilha pelo não terminal correspondente
  - aceita:
    - reconhece que a sentença foi gerada pela gramática
  - erro:
    - ocorrendo erro de sintaxe, chama uma subrotina de atendimento a erros

Ações bottom-up: 3 problemas

- Problema 1: Como decidir qual lado direito de produção trocar pelo lado esquerdo (redução) ?

Ações bottom-up: 3 problemas

- Problema 1: Como decidir qual lado direito de produção trocar pelo lado esquerdo (redução) ?
  - Ler entrada da esquerda para direita
  - Em algumas situações uma escolha aparece:
    1. ler mais um caractere da entrada

    ou
    2. aplicar redução

$S \rightarrow abc \mid a$

Análise	Entrada	Ação
$	abc$	Ler
a$	bc$	Ler
ab$	c$	Ler
abc$		Redução
S$	$	Aceitar

Ações bottom-up: 3 problemas

- Problema 1: Como decidir qual lado direito de produção trocar pelo lado esquerdo (redução)?
  - conflito reduce/reduce
- Problema 2: Como escolher entre ler ou aplicar a redução?
  - Conflito Shift / Reduce
- Problema 3: Como guardar o substring que está sendo analisando?
  - Pode ser uma combinação de 1 ou mais símbolos (terminais ou não-terminais)
  - Pilha

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

Pilha	Entrada	Ação
$	id + id * id	shift
$id	+ id * id	Reduce $E \rightarrow id$
$E	+ id * id	shift
$E+	id * id	shift
$E+id	* id	Reduce $E \rightarrow id$
$E+E	* id	Reduce $E \rightarrow E + E$
$E	* id	shift
$E*	id	shift
$E*id	$	Reduce $E \rightarrow id$
$E*E	$	Reduce $E \rightarrow E * E$
$E	$	aceita!

Ações em Parsing Empilha-Reduz

- Reduz
  - Se:
    - $\gamma\alpha$ está na pilha
    - $A \rightarrow \alpha$
    - existe um $\beta \in T^*$ tal que $S \Rightarrow^* \gamma A \beta \Rightarrow \gamma \alpha \beta$
  - Então:
    - Pode-se remover o "handle" α;
    - **Reduz**-se $\gamma \beta$ para γA na pilha
  - $\gamma\alpha$ é um prefixo viável
    - Ele pode derivar numa seqüência de terminais
- Shift (empilha)
  - Empilhar o terminal, avançar entrada
- Erro
- Aceitar

Várias soluções para o parsing Bottom-Up

- O parsing bottom-up é mais poderoso do que o parsing Top-Down
  - Regras aplicadas em reverso
  - Pode "adiar decisões" de reduções
    - Pode usar mais de um símbolo na entrada para tomar a decisão.
- Vários algoritmos para o parsing Shift-Reduce (empilha-reduz):
  - LR(0)
  - SLR(1)
  - LR(1)
  - LALR(1)

Parsing LR

- Parser baseado em tabelas
  - Lê a entrada de esquerda para direita (L)
  - Cria derivações mais a direita (R)
- Usa um autômato finito com pilha
  - Cada estado representa produções da gramática
  - Transições são feitas a cada terminal/não terminal

Estruturas de dados:
Pilha de estados {s} e símbolos
Tabela de ações: Action[s,t]; t ∈ **T**
Tabela de transições: Goto[s,X]; X ∈ **N**

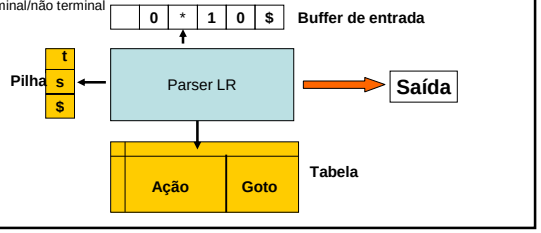


Tabela Ação/Transição (LR)

- Ação: A partir de um estado s e de um terminal t, Ação[s, t] indica:
  - Se se faz um shift S (empilha) ou um Reduce R (reduz) ao ler o símbolo 'a' na entrada.
  - Caso S: indica também o estado a empilhar;
    - Fazer:
      - empilhar o estado e avançar na leitura.
  - Caso R: indica também a regra a reduzir e, **a seguir**, aplica uma Transição.
    - Fazer:
      - depilhar tantos estados como símboloS reduzidoS;
      - empilhar o estado alvo do Goto(topo Pilha, Símbolo reduzido).
- Transição (goto): a partir de um estado s e de um não-terminal X, Goto[s, X] indica o próximo estado a empilhar.

Exemplo de uso de tabela Ação/Transição

- Gramática usada:

$$S \rightarrow T$$
$$T \rightarrow F \mid T * F$$
$$F \rightarrow id \mid (T)$$

Tabela Ações/Transições

Ações | Goto

	*	(	)	id	$	E	T	F
0		S5		S8			2	1
1	R1	R1	R1	R1	R1			
2	S3				Ok!			
3		S5		S8				4
4	R2	R2	R2	R2	R2			
5		S5		S8			6	1
6	S3		S7					
7	R4	R4	R4	R4	R4			
8	R3	R3	R3	R3	R3			

Tabela Ações/Transições

Ações | Goto

Terminais | Não-Terminais

Estados

	*	(	)	id	$	E	T	F
0		S5		S8			2	1
1	R1	R1	R1	R1	R1			
2	S3				Ok!			
3		S5		S8				4
4	R2	R2	R2	R2	R2			
5		S5		S8			6	1
6	S3		S7					
7	R4	R4	R4	R4	R4			
8	R3	R3	R3	R3	R3			

Tabela Ações/Transições

Terminais | Não-Terminais

Estados

Shift e empilha 8

	*	(	)	id	$	E	T	F
0		S5		S8			2	1
1	R1	R1	R1	R1	R1			
2	S3				Ok!			
3		S5		S8				4
4	R2	R2	R2	R2	R2			
5		S5		S8			6	1
6	S3		S7					
7	R4	R4	R4	R4	R4			
8	R3	R3	R3	R3	R3			

Tabela Ações/Transições

Terminais

Não-Terminais

Estados

	*	(	)	id	$	E	T	F
0		S5		S8			2	1
1	R1	R1	R1	R1	R1			
2	S3				Ok!			
3		S5		S8				4
4	R2	R2	R2	R2	R2			
5		S5		S8			6	1
6	S3		S7					
7	R4	R4	R4	R4	R4			
8	R3	R3	R3	R3	R3			

Shift e empilha 8

Reduce a 4a regra

Análise de "(id)*id" (1/2)

Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (T)

Stack	Input	Action
0	(id) * id $	Shift S5
0 5	id) * id $	Shift S8
0 5 8	) * id $	Reduz 3 F→id, pop 8, goto [5,F]=1
0 5 1	) * id $	Reduz 1 T→ F, pop 1, goto [5,T]=6
0 5 6	) * id $	Shift S7
0 5 6 7	* id $	Reduz 4 F→ (T), pop 7 6 5, goto [0,F]=1
0 1	* id $	Reduz 1 T → F pop 1, goto [0,T]=2

Análise de "(id)*id" (2/2)

Stack	Input	Action
0 1	* id $	Reduz 1 T→F, pop 1, goto [0,T]=2
0 2	* id $	Shift S3
0 2 3	id $	Shift S8
0 2 3 8	$	Reduz 3 F→id, pop 8, goto [3,F]=4
0 2 3 4	$	Reduz 2 T→T * F pop 4 3 2, goto [0,T]=2
0 2	$	Aceitar

Construindo tabelas LR

- Para montar as tabelas, precisa-se:
  - Definir os estados,
  - Definir os Gotos/Ações
- Os estados devem capturar o andamento na análise de quais símbolos podem ser reduzidos.
  - Vai-se dever calcular quais conjuntos de símbolos podem ser considerados como handles.
  - Noção de ponto / fechamento.
- Os gotos e as açoes vão ser definidos graças ao cálculo das transições entre os estados.
  - Tendo analisado que tenho um handle, quais símbolos podem aparecer depois?
  - Sucessor(.)

Cálculo dos conjuntos de itens canônicos LR

- A tabela pre-calcula todas as derivações possíveis a partir de um dado ponto de análise Left -> Right da entrada.
- Parte de noção de configuração ou ítem LR(0) associado a uma regra (ou conjunto de regras)
  = uma regra da gramática com um 'ponto' em algum lugar à direita.
- Exemplo: a regra **T → T * F** possui quatro ítens:
  T → •T * F
  T → T • * F
  T → T * • F
  T → T * F •
  - O ponto • representa até onde foi feita a análise
- Quer-se calcular os conjuntos canônicos de ítens
  = todos os ítens alcançáveis a partir de um conjunto de regras da gramática.
- Cada conjunto será um estado do parser LR
- Similar à conversão de AFND para AFD

Cálculo dos conjuntos

- Propriedade de Fechamento:
  - Se $T \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_n$ está em um conjunto, e X_{i+1} é um não-terminal que deriva em **α (i.e.: existe a regra $X_{i+1} \rightarrow \alpha$)**, então também entra no conjunto:
    $X_{i+1} \rightarrow \bullet \alpha$
  - Itera-se essa operação
    - Calcula-se o conjunto como um ponto fixo
- Ponto inicial:
  - Acrecenta-se um não-terminal S' à gramática
  - Adiciona-se uma produção S' → S
  - Conjunto de ítens Inicial é:
    fechamento(S' → • S)

Exemplo: fechamento(S' → • T)

S' → T
T → F | T * F
F → id | (T)

S' → • T
T → • F
T → • T * F

F → • id
F → • (T)

Estado 0

Sucessor(C,X)

- É o segundo procedimento útil para montar a tabela LR
  - Pega em argumento um conjunto de ítens C e um símbolo X
  - Retorna um conjunto de ítens
  - Informalmente: "mover o ponto pelo símbolo X"
- Cálculo do sucessor a partir do estado e0 ao ler X:
  1. mover ponto para direita em todos os ítens de e0 onde o ponto precede X
     - Para todas as regras A → **α • X β** em C, retorna A → **α X • β**
  2. Calcular o fechamento deste conjunto de ítens.
  3. Esse novo estado será sucessor(e0,X)

Exemplo de Sucessor

e0 = Estado 0 =

{S' → • T, T → • F, T → • T * F, F → • id,
F → • (T) }

Sucessor(e0, "(") = ?

Fechamento(F → (• T))

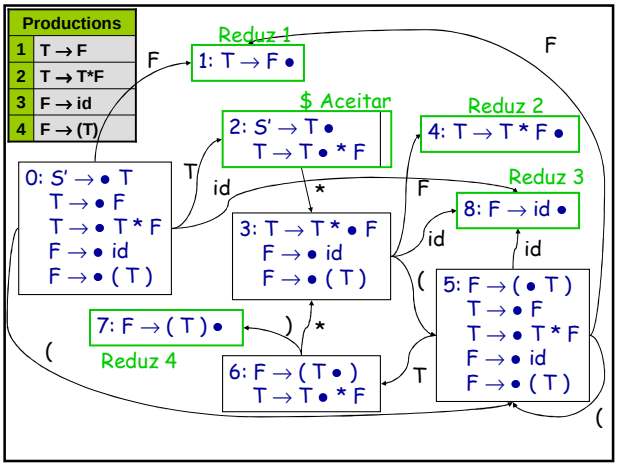
= { F → (• T)
T → • F,
T → • T * F,
F → • id,
F → • (T) }

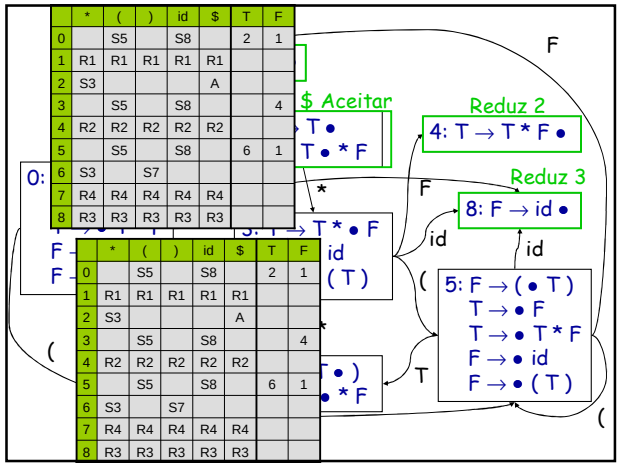
S' → T
T → F | T * F
F → id | (T)

Construção dos Conjuntos de Ítens

- Família de conjuntos de ítens
  - Cada conjunto será um estado do parser

proc items(G')
C = fechamento({S' → • S});
do foreach I ∈ C **do**
foreach X ∈ **(N ∪ T) do**
C = C ∪ Sucessor(I, X);
while C é modificado;





	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Construção Tabela LR(0)

1. Construir F = {I_0, I_1, …I_n}
   - I_0 é o estado inicial

2. a) Se {S' → S•} ∈ I_i
   então ação[i,$] = aceitar
   b) Se {A → α•} ∈ I_i e A != S'
   então ação[i,_] = reduzir A → α
   c) Se {A → α•aβ} ∈ I_i e Sucessor(I_i,a)=I_j
   então ação[i,a] = shift j
3. Se Sucessor(I_i,A) == I_j então goto[i,A] = j
4. Todas as entradas não definidas são erros

Observações

- LR(0) sempre reduz se {A → α•} ∈ I_i, sem lookahead
- Ítens Shift e Reduce podem estar no mesmo conjunto de configurações
- Pode haver mais de um ítem reduce por conjunto
- Problema com A → Є

LR(0) é fraco, considere:

S' → T
T → F | T * F
F → id | (T)
F → id = T
T → id

5: F → id •
F → id • = T
…
Conflito Shift/reduce!

2: F → id •
T → id •…
Conflito Reduce/Reduce!

Sumário

- A análise bottom-up usa:
  - Handles, para determinar qual cadeia reduzir;
  - Tabelas LR (Goto/Transições) para determinar quais reduções e shifts efetuar.
- Usa-se um autômato de estados finitos, com uma pilha, para reconhecer uma sentença.
  - Construído a partir dos itens canônicos
    - Fechamento + propagação
- Vimos o LR(0) que é limitado.
  - Como melhorar o algoritmo?