5 – Análise Sintática Ascendente

Linguagens e Programação

Ana Madureira

Engenharia Informática
Ano Letivo: 2021/2022

Fontes:

1. Compiladores Princípios e práticas, Keneth C.Louden, Thomson, 2004.
   Cap. 5 Análise Sintáctica Ascendente
2. Processadores de Linguagens – da concepção à implementação, Rui Gustavo Crespo. IST Press.1998.
   Cap. 5 Análise Sintáctica Ascendente
3. Compiladores - Princípios , Técnicas e Ferramentas, Alfred V. Aho, Monica S. Lam e Ravi Sethi, Pearson, 2ª edição, 2007.
   Cap. 4 secção 4.5 Análise Sintáctica Bottom-Up

1

Classe de Analisadores Sintáticos

- Verificar se uma dada sequência de *tokens* constitui um programa válido
- **Descendentes (top-down)**
  - Com retrocesso
  - Sem retrocesso
    - Recursivo
    - Preditivo (LL)
- **Ascendentes (bottom-up)**
  - Shift-reduce
  - Shift-reduce com análise de precedência
  - **LR**
    - LR(0)
    - SLR(1)
    - LALR(1)
    - LR(1)

2

Análise *Bottom-Up*

- Os parsers ***top-down*** têm de decidir qual a regra gramatical a aplicar tendo visto, na prática, apenas um **símbolo** do seu lado direito.
- Nos analisadores sintáticos ***bottom-up*** as regras gramaticais só são aplicadas depois de se ter visto e reconhecido toda a sua parte direita e possivelmente mais o(s) símbolo(s) seguinte(s).
- Os parsers ***bottom-up*** aplicam as regras gramaticais substituindo a sua parte direita pelo não-terminal que constitui a sua parte esquerda.
- **Constroem uma árvore de parse partindo das folhas** (terminais - ***tokens***) até se chegar ao símbolo inicial da gramática. Fazem uma visita em pós-ordem da árvore construída.
- A construção feita desta forma é sempre uma derivação mais à direita (***rightmost***) feita por ordem inversa.
- Considere-se a gramática $S \rightarrow (\,S\,)\,S \mid \varepsilon$ e a entrada $(\,)$.

  Uma derivação feita desta forma pode ser:

  $(\,) \Leftarrow (\,S\,) \Leftarrow (\,S\,)\,S \Leftarrow S$

3

Análise *BottomUp - Exemplo*

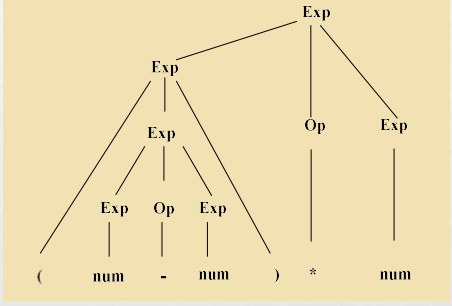
Considere-se a gramática:
Exp → Exp Op Exp | (Exp) | num
Op → + | - | * | /

E a sequência de tokens:

(num - num) * num

Na análise ***bottom-up*** partimos da sequência de ***tokens*** que pretendemos analisar e vamos aplicando regras gramaticais (ao contrário), até se chegar ao símbolo inicial.
Trata-se de uma **derivação mais à direita**:

⇐ (num - num) * num
⇐ (Exp - num) * num
⇐ (Exp Op num) * num
⇐ (Exp Op Exp) * num
⇐ (Exp) * num
⇐ Exp * num
⇐ Exp Op num
⇐ Exp Op Exp
Exp



4

Análisadores Sintáticos *Ascendentes*

- Foram propostos e adotados na implementação de compiladores diversos algoritmos, dos quais se destacam:
  - CKY
  - Precedência de operadores
  - Precedência simples
  - Precedência fraca
  - Contexto limitado
  - **LR (Knuth)**

5

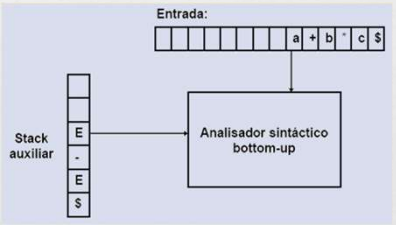
Implementação dos *parsers* Ascendentes

- A sequência de símbolos da entrada termina com $.
- A stack de reconhecimento pode conter sequências alternadas de variáveis e terminais e de estados.
- Baseiam-se nos seguintes tipos de ações efetuadas sobre os tokens da entrada e sobre uma stack auxiliar (de terminais e não-terminais):
  - **Deslocar** (shift) - retira da entrada o token corrente e coloca-o na stack (push)
  - **Reduzir** (reduce) - Substitui símbolos do topo da stack por um não-terminal, por aplicação de uma regra gramatical
  - **Aceitação** – termina a análise sintática reconhecendo a entrada. O texto da entrada é aceite se após o seu consumo estiver no topo da stack apenas o símbolo inicial da gramática
  - **Erro** – termina a análise sintática com erro de reconhecimento

6

Implementação dos *parsers Ascendentes*

- A sequência de símbolos da entrada termina com $.
- A stack de reconhecimento pode conter sequências alternadas de variáveis e terminais e de estados.
- Baseiam-se nos seguintes tipos de ações efetuadas sobre os *tokens* da entrada e sobre uma *stack* auxiliar (de terminais e não-terminais):
  - **Deslocar (shift)** - retira da entrada o token corrente e coloca-o na stack (push)
  - **Reduzir (reduce)** - Substitui símbolos do topo da stack por um não-terminal, por aplicação de uma regra gramatical
  - **Aceitação** – termina a análise sintática reconhecendo a entrada. O texto da entrada é aceite se após o seu consumo estiver no topo da stack apenas o símbolo inicial da gramática
  - **Erro** – termina a análise sintática com erro de reconhecimento.



7

Exemplo

- Considere-se a seguinte gramática para expressões:
  E → F + E | F
  F → id * F | id | (E)
  Pretende-se fazer a análise sintática da entrada **id + id * id** utilizando o método *bottom-up* e as acções de **deslocar-reduzir**. Uma possibilidade seria:

	Stack	Entrada	Acções
1	$	id + id * id $	deslocar
2	$ id	+ id * id $	reduzir F→id
3	$ F	+ id * id $	deslocar
4	$ F +	id * id $	deslocar
5	$ F + id	* id $	deslocar
6	$ F + id *	id $	deslocar
7	$ F + id * id	$	reduzir F→id
8	$ F + id * F	$	reduzir F→id*F
9	$ F + F	$	reduzir E→F
10	$ F + E	$	reduzir E→F+E
11	$ E	$	aceitar

Há vários passos neste processo de derivação onde poderia haver dúvidas quanto à próxima acção.

Na configuração 3, por exemplo, poderia ser possível realizar uma redução pela regra E → F, em vez do deslocamento efectuado;

Na configuração 5 também era possível fazer uma redução em vez do deslocamento.

No entanto se nessas configurações críticas tivéssemos feito reduções em vez de deslocamentos não seria mais possível atingir o símbolo inicial da gramática E.

8

Conflitos durante a análise sintática de deslocar e reduzir

- Existem gramáticas independentes do contexto para as quais este tipo de análise não pode ser usada
- Um analisador deste tipo para uma tal gramática pode atingir uma configuração na qual, mesmo conhecendo o conteúdo da stack e o próximo símbolo de entrada (lookahead), não pode decidir:
  - entre deslocar ou reduzir (**conflito shift/reduce**)
  - qual das diversas reduções alternativas realizar (**conflito reduce/reduce**)

Stmt → if Exp then Stmt |
if Exp then Stmt else Stmt
Stack: … if Exp then Stmt
Entrada: else … $
Vamos optar por reduzir if Exp then Stmt através de Stmt
ou vamos empilhar(deslocar) o else da entrada?

Par → id
Exp → id
***Stack*: … id (id**
Entrada: , id) … $
Vamos optar por reduzir id que se encontra no topo da stack por Par ou por Exp?

9

Definições

- A concatenação dos símbolos da stack com o que resta da entrada constitui sempre uma forma sentencial da derivação que o parser bottom-up está a fazer do texto da entrada:
  - Essa derivação é sempre uma derivação mais à direita
  - A configuração de um parser bottom-up (símbolos da stack + o que resta da entrada) constitui o que se chama uma forma sentencial direita
- Os símbolos que se encontram em cada configuração de um parser bottom-up na sua stack constituem o que chama um prefixo viável da forma sentencial direita que constitui essa configuração
  - Assim, ε e 'id' são prefixos viáveis de 'id + id * id' na gramática do exemplo anterior; no entanto 'id +' já não o é
- Durante a análise sintáctica bottom-up nem todas as reduções possíveis se devem efectuar.
  - Só se deve efetuar uma redução se o símbolo inicial da gramática puder vir a ser atingido
  - Só se deve efetuar uma redução quando o topo da stack contiver o que se chama um handle
  - Um handle é uma subcadeia de uma forma sentencial direita, correspondente ao lado direito de uma regra gramatical, cuja redução ao respectivo não-terminal constitui um passo na derivação mais à direita do texto inicial
  - Formalmente: β é um handle da forma sentencial direita α β w, onde α é qualquer sequência de símbolos e w qualquer sequência de terminais, se existir uma regra X → β e se S ⇒* α X w ⇒ α β w
- Os parsers bottom-up devem reconhecer *handles* para efetuarem uma redução

10

Análise sintáctica LR(k)

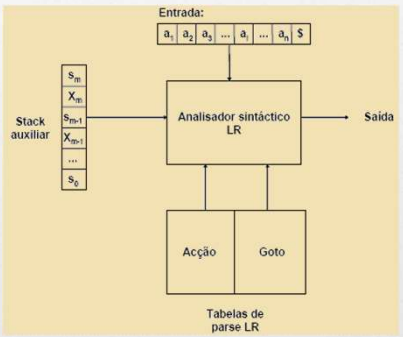
- Método de análise sintática *bottom-up* onde, além da *stack* auxiliar se examinam os próximos k *tokens* (terminais) da entrada
- Só é prática a situação **k=1**
- É mais poderosa que a análise sintáctica *top-down* LL(k)
- Praticamente todas as construções das linguagens de programação descritas por gramáticas livres de contexto podem ser analisadas por ***parsers* LR**
- É difícil de implementar "à mão"
- Significado de **LR(k)**:
  - **L** - A análise faz-se percorrendo a entrada da esquerda para a direita (Left to right parse)
  - **R** - A análise produz uma derivação mais à direita por ordem inversa (Rightmost derivation in reverse)
  - **k** - número de terminais de antecipação (lookahead) da entrada que são examinados, para determinar a próxima ação (deslocar ou reduzir)

11

Funcionamento do analisador sintático LR

Problemas a resolver durante a análise bottom-up:

- escolher uma das operações: reduzir ou deslocar
- no caso de redução, localizar na stack a subcadeia ? a reduzir
- no caso de existir mais do que uma produção com o lado direito igual a ?, que produção escolher?



Stack	Fila entrada	Configuração
$	w$	inicial
$S	$	final

- s_j - indicação de um estado
- X_j - um símbolo gramatical (terminal ou não-terminal)
- s_0 - estado inicial

Existem diferentes métodos para a construção de tabelas LR:
- **SLR (simple LR)**
- **LR (canónico)**
- **LALR (lookahead LR)**

12

Algoritmo do analisador sintático

Configuração inicial:

- **Entrada total presente**
- **Estado inicial** (S_0) no topo da *stack*

Tabelas de *Parse*:

- **Tabela ação: A[s,t]** - matriz indexada pelos estados e pelos *tokens* da entrada mais o símbolo $. Cada entrada da tabela contém 4 tipos de ações:
  - **si** - deslocar (shift) um token da entrada para a stack e cobri-lo com o estado i
  - **rj** - reduzir o topo da stack pela regra gramatical j
  - **acc** - aceitar o texto da entrada
  - **posições em branco** - erros sintáticos
- **Tabela goto: G[s, N]** - matriz indexada pelos estados e pelos não-terminais da gramática (usada para determinar o próximo estado após uma operação *reduce*)- As entradas que não estão em branco contêm um estado

Seja tok o próximo *token*, inicializado com o 1º token da entrada

```
repeat forever
    s = top(); a = tok;
    if (A[s, a] == si)
        push(a); push(i); advance();
    else if (A[s, a] == rj)
        R = regra j (p. ex. X → β)
        pop 2 * |β| símbolos da stack;
        k = top();
        push(X); push(G[k,X]);
        output("X → β")
    else if (A[s, a] == acc)
        accept();
        return;
    else
        error();
```

13

Exemplo

Seja a seguinte gramática para expressões:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

As tabelas de *parse* LR para esta gramática podem ser:

estado	Acção						Goto		
	Id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

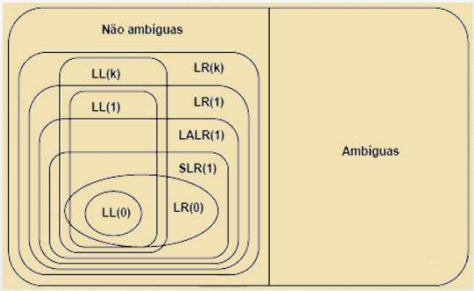
Verificar, usando a tabela, se a entrada "id * id + id" pertence à gramática:

Stack	Entrada	Acção
0	id * id + id $	shift 5
0id5	* id + id $	reduce by 6
0F3	* id + id $	reduce by 4
0T2	* id + id $	shift 7
0T2*7	id + id $	shift 5
0T2*7id5	+ id $	reduce by 6
0T2*7F10	+ id $	reduce by 3
0T2	+ id $	reduce by 2
0E1	+ id $	shift 6
0E1+6	id $	shift 5
0E1+6Id5	$	reduce by 6
0E1+6F3	$	reduce by 4
0E1+6T9	$	reduce by 1
0E1	$	accept

14

Gramáticas LR(k)

Hierarquia das gramáticas independentes de contexto



Uma gramática diz-se LR(k) se for possível construir um *parser* LR(k) para essa gramática, sem conflitos. Uma condição necessária é ser não ambígua.

15

Passos na Construção de Tabelas LR

1. Estender a gramática inicial, introduzindo um novo símbolo inicial S', um novo terminal $ e uma nova produção S'→S
2. Construir um autómato finito determinístico (AFD) a partir da extensão de G, onde os estados do autómato têm a informação necessária para o analisador sintático decidir a ação a executar.
3. Representar o AFD por uma tabela de transição de estados.
4. Definir a tabela de ações

16

Construção de tabelas SLR
Itens LR(0)

- Método mais simples (do que os métodos LR canónico e LALR) – baseia-se no conceito de itens LR(0)
- Um item LR(0) é qualquer regra gramatical associada a uma posição no lado direito da regra
- Essa posição é representada por um . colocado no lado direito da produção (este . é um metacaracter e não um terminal)
- Um item de uma produção indica o quanto de uma já examinamos a uma dada altura do processo de análise sintática.
- Por exemplo A →Aa produz os quatro itens seguintes:
  - **A → .Aa (indica que desejamos em seguida examinar uma cadeia na entrada)**
  - **A → A.a (indica que acabamos de examinar uma cadeia na entrada, derivável a partir de A e que esperamos em seguida ver uma cadeia derivável de a)**
  - **A → Aa.(indica que se pode efectuar a redução)**
- Produções vazias do tipo A → ε só possuem um item LR(0) que é A → .

17

SLR – Simple LR

- Construir C={I_0, I_1, ..., I_n}
- Se {$A \rightarrow \alpha$. a β} pertence a I_i e goto(I_i,a) = I_j , então acção(i,a) = *shift* j
- Se {$A \rightarrow \alpha$. } pertence a I_i, então para todo o a em FOLLOW(A), então acção(i,a) = *reduce* $A \rightarrow \alpha$; A não pode ser S'
- Se {S' → S} pertence a I_i, então acção(i,$) = aceita.
- Se {$A \rightarrow \alpha$. X β} pertence a I_i, X variável (não terminal), então salto(i,X) = j

18

Construção de tabelas SLR
Itens LR(0)

- Para a gramática S → (S) S | ε a coleção de itens LR(0) que é possível definir é: **S → . (S) S S → (S .) S S → (S) S .**
  S → (. S) S S → (S) . S S → .
- A ideia central do SLR é
  - construir primeiro a partir da gramática um AFD que reconheça prefixos viáveis.
  - Agrupar esses itens em conjuntos , os quais dão origem aos estados do analisador sintático SLR.
- Os itens podem ser vistos como os estados do AFD que reconhece os prefixos viáveis
- Para construir a coleção canónica LR(0) para uma gramática, define-se uma gramática aumentada e duas funções: fecho e desvio.
- É possível construir um AFN entre os itens LR(0) de uma gramática:
  - Se A → α . ω η for um item e ω um símbolo gramatical cria-se uma transição para o item A → α ω . η, etiquetada por ω
  - Se A → α . X η for um item e X um não-terminal criam-se transições-ε para todos os itens do tipo X → . β
- Para existir sempre um estado inicial a gramática deverá conter uma nova regra S'→ S, sendo o estado inicial o que contém o item S' → . S

19

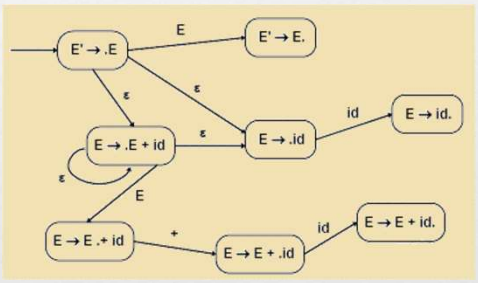
Exemplo – construção do AFN entre os itens

- **Construir o AFN de itens LR(0) da gramática:**

E' → E
E → E + id | id

- **Os itens LR(0) desta gramática aumentada são:**

E' → .E
E' → E.
E → .E + id
E → E .+ id
E → E + .id
E → E + id.
E → .id
E → id.

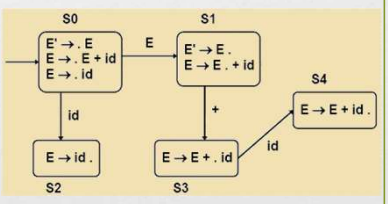


20

AFD's e colecções de itens LR(0)

- Após a construção do AFN entre itens LR(0) é possível construir um AFD equivalente mínimo, seguindo o algoritmo referido no capítulo da análise léxica (algoritmo da construção dos subconjuntos)
- Os itens agrupados num mesmo estado deste AFD mínimo constituem o que se chama um conjunto de itens LR(0). A coleção de todos os conjuntos de itens constitui a coleção canónica de itens da gramática
- Para o AFN do exemplo anterior chegaríamos à coleção canónica e ao AFD seguintes:

S0: E' → . E E → . E + id E → . id	**S1:** E' → E . E → E . + id	**S2:** E → id .
S3: E → E + . id	**S4:** E → E + id .	



21

Parsers LR(0)

- Por vezes o AFD e os estados a que se chega usando os itens LR(0) é suficiente para a construção de um *parser* sem *lookahead*.
- Algoritmo para o *parser* LR(0):
  1. Se o estado s do topo da stack contém algum item da forma A → α . ω η com ω um terminal então a ação é **shift**. Se o terminal que for para a stack for x então o estado s deve conter algum item da forma A → α . x η (caso contrário temos um erro sintático); o estado seguinte, a colocar no topo da stack é o estado que contiver o item A → α x . η
  2. Se o estado s contiver um item completo (um item da forma A → α .) então a ação é reduzir pela regra A → α; uma redução pela regra S' → S é a aceitação se a entrada estiver vazia; a redução pela regra A → α retira da stack os símbolos de α e os correspondentes estados; o estado que fica no topo terá de conter então um item B → β . Aγ ; seguidamente o símbolo A é colocado na stack juntamente com o estado que contiver o item B → β A . γ
- Para que uma gramática seja LR(0) é necessário que a sua coleção canónica de itens satisfaça certas condições, por forma a que o algoritmo anterior seja aplicável e livre de conflitos.
- Se um estado contiver um item completo (A → α .) não pode conter mais nenhum item. Caso contrário existirá um conflito ***shift-reduce*** (com outros itens não completos) ou um conflito ***reduce-reduce*** (com outro item completo)
- Logo a gramática anterior não é LR(0) (ver o estado S1).

22

Exemplo LR(0)

Construir os estados e as ações LR(0) para a gramática: A → (A) | a

Deve-se acrescentar a regra A' → A para definir a aceitação.

A coleção canónica de itens contém os conjuntos:

S0:
A' → . A
A → . (A) (shift)
A → . a

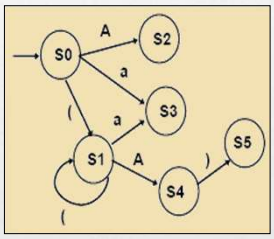
S1:
A → (. A)
A → . (A) (shift)
A → . a

S2:
A' → A . (accept)

S3:
A → a . (reduce)

S4:
A → (A .) (shift)

S5:
A → (A) . (reduce)



Stack	Entrada	Acção
0	((a)) $	shift
0(1	(a)) $	shift
0(1(1	a)) $	shift
0(1(1a3	)) $	reduce
0(1(1A4	)) $	shift
0(1(1A4)5	) $	reduce
0(1A4	) $	shift
0(1A4)5	$	reduce
0A2	$	accept

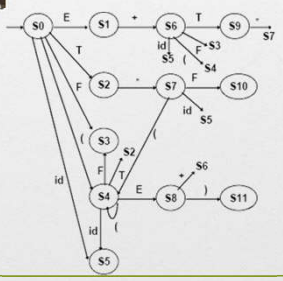
23

Exemplo – construção da coleção canónica de itens LR(0) e respetivo AFD

Para a gramática aumentada:

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

Obtemos o seguinte conjunto de itens e o respetivo AFD:



É possível construir directamente o AFD de itens sem passar primeiro pelo AFN [Aho, secção 4.7]

S0:
E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

S1:
E' → E .
E → E . + T

S2:
E → T .
T → T . * F

S3:
T → F.

S4:
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

S5:
F → id .

S6:
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

S7:
T → T * . F
F → . (E)
F → . id

S8:
F → (E .)
E → E . + T

S9:
E → E + T .
T → T . * F

S10:
T → T * F.

S11:
F → (E).

24

Tabelas SLR(1)

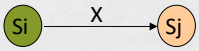
- A gramática anterior não é LR(0) (ver estados S1, S2 e S9)
- Sem lookahead surgem demasiados conflitos
- No entanto, se considerarmos o próximo token, muitos desses conflitos podem ser resolvidos
- Por exemplo, se considerarmos o estado 1 da página anterior que pede uma redução pela regra **E' → E** e um deslocamento relativo ao outro item, podemos ter estas duas acções dependentes do próximo token. Considerando que se fizermos a redução isso equivale a dizer que se acabou de reconhecer o não-terminal que é a parte esquerda da regra (E' neste caso), isso significa que legalmente o próximo token só pode ser um elemento do conjunto followers desse não-terminal. Assim, no caso de S1 do acetato anterior, a redução só se deve fazer se o próximo token pertencer a followers(E') = { $ } e o deslocamento só se deve fazer se o próximo token for + (o terminal a seguir ao . num item).
- Temos então acções diferentes, mas perfeitamente distinguidas pelo próximo token.

25

Construção de tabelas SLR(1)

- Considerando um token de lookahead podemos construir uma tabela de parse LR(1) a partir do AFD de itens LR(0), a que se dá o nome de SLR(1):
  1. Construir a colecção canónica de itens LR(0): C = { S0, S1, ..., Sn }
  2. O estado inicial é o que contém S' → .S (geralmente numerado como estado 0)
  3. Construir a matriz A[s, t] da forma:
     a. Se X → α.aβ ∈ Si (a - terminal) e goto(Si, a) = Sj então A[i, a] = sj (shift j)
     b. Se X → α. ∈ Si (X → α - regra k) então A[i, a] = rk (reduce by k) para todos os a's ∈ followers(X)
     c. Se S' → S. ∈ Si então A[i, $] = accept
  4. Se goto(Si, X) = Sj (X - não-terminal) então G[i, X] = j

goto(Si,X) = Sj ⇒ Se o estado i for coberto com o símbolo X então X será coberto pelo estado j, ou seja, ou próximo estado será j



26

Exemplo de construção da tabela SLR(1)

Construir a tabela SLR(1) para a gramática do slide 13

- Considere-se o estado S0 (é o estado inicial porque contém o item E' → . E)
- Temos deslocamentos para os estados S4 e S5 com os *tokens* (e id (correspondentes aos itens F → . (E) e F → . id):
  A[0, ( ] = s4 e A[0, id] = s5
- Temos também algumas entradas da tabela Goto:
  G[0, E] = 1; G[0, T] = 2; G[0, F] = 3
- Considere-se agora o estado S2:
- O item E → T. indica uma acção de redução para os *tokens* pertencentes a followers(E) = { $, +,) }
- O item T → T . * F indica uma acção de deslocamento para o *token* *, que não entra em conflito com as reduções anteriores
  A[2, +] = A[2,)] = A[2, $] = reduce by E → T (r2)
  A[2, *] = s7

Considerando a seguinte numeração para as regras gramaticais:

0. E' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Seria definida a tabela seguinte:

27

Exemplo de construção da tabela SLR(1)

estado	Acção						Goto		
	Id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

28

Gramáticas SLR(1)

- Uma gramática diz-se SLR(1) se a construção da tabela de *parse* pelo método SLR(1) não tiver qualquer conflito (ações distintas na mesma célula da tabela)
- Embora um grande número de gramáticas de linguagens de programação seja SLR(1) ainda há algumas que não o são.
- Exemplo:

Consideremos um estado composto pelos itens:
Sn: A → B. = D
C → B.

Consideremos ainda que followers(C) contém o *token* =
Neste estado podemos ter:
- um deslocamento no *token* =
- uma redução pela regra C → B nos *tokens* que são os followers(C) – este conjunto contém o *token* =

Neste caso, 1 *token* de *lookahead* não é suficiente para decidir entre o *shift* e o *reduce*.

29

Itens LR(1)

- Por vezes, na construção SLR(1) aparecem ainda reduções inválidas, para cadeias que não são *handles*, gerando conflitos
- É o que acontece no exemplo anterior (nem todas as cadeias que coincidem com a parte direita de uma regra e são seguidas por um elemento do conjunto *followers* da parte esquerda são *handles*)
- Para melhorar o reconhecimento dos *handles* é necessário incorporar os *tokens* de *lookahead* nos itens, obtendo-se o que chama itens LR(1) [Aho, secção 7.4]
- A construção da tabela LR(1) canónica é feita com base na construção dos estados LR(1) e respectivo AFD, a partir dos itens LR(1) [Aho, secção 7.4]
- Se a tabela gerada não contiver conflitos, então diz-se que a gramática correspondente é LR(1) – praticamente todas as linguagens de programação são LR(1)

30

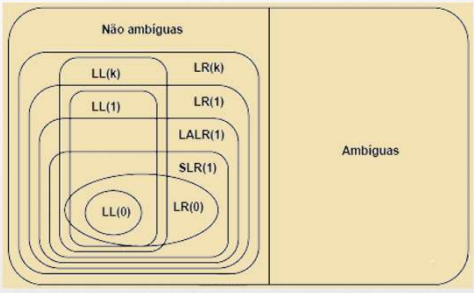
Método LR canónico

- Os estados do AFD têm informação adicional para além da incorporada nos estados do AFD para o analisador SLR.
- Uso de antevisão (lookahead)
- A vantagem do LR canónico consiste no reconhecimento de maior nº de linguagens de programação
- Todas as linguagens SLR e todas as linguagens LL(1) são linguagens LR(1) mas o contrário já não se verifica.
- As tabelas geradas pelo método LR canónico são normalmente de dimensão superior às tabelas geradas pelo método SLR
- Uma gramática G diz-se LR(1) se for possível gerar as tabelas do Analisador sintático LR canónico para a gramática estendida de G.

31

Gramáticas LR(k)

Hierarquia das gramáticas independentes de contexto



Uma gramática diz-se LR(k) se for possível construir um *parser* LR(k) para essa gramática, sem conflitos. Uma condição necessária é ser não ambígua.

32

Parsers LALR(1)

- Os *parsers* LR(1) são bastante poderosos, mas o número de estados tende a ser muito grande, relativamente aos estados das tabelas SLR(1)
  - Para uma implementação da gramática de uma linguagem média é frequente obter alguns milhares de estados LR(1), contra apenas poucas centenas para a tabela SLR(1)
- Na prática usa-se um outro método de construção de tabelas, dito LALR, que contém o mesmo número de estados dos *parsers* SLR, mas mantém, praticamente intacto, o poder dos *parsers* LR canónicos [Aho, secção 7.4]
- A ideia é agrupar os estados que diferem apenas no símbolo de antevisão. Referem-se duas abordagens para a construção das tabelas do analisador LALR
  - Os estados são deduzidos a partir dos estados do analisador LR canónico por um procedimento de agrupamento.
  - Os estados são identificados de raiz calculando os símbolos de antevisão quando necessário.
- As tabelas LALR(1) são, em geral, de dimensão superior quando comparadas com as tabelas SLR.

33

Gramáticas LALR(1)

- Uma gramática G diz-se LALR(1), ou simplesmente LALR se for possível gerar as tabelas do analisador sintático ascendente LALR para essa gramática, sem conflitos.
- As tabelas LAR são em geral de dimensão superior às tabelas SLR.

34

Recuperação de erros nos *parsers bottom-up*

- Quando na tabela de *parse* se encontra uma entrada de erro é necessário recuperá-lo e tentar continuar a análise sintática. Para isso consideram-se as seguintes possíveis ações:
  - **Descartar um estado (**pop) da stack auxiliar do parser
  - **Descartar símbolos (**terminais) da entrada até aparecer algum símbolo válido para o estado actual
  - Colocar um novo estado na stack do parser (push)
- Destas é necessário escolher a ação mais adequada. Um método possível poderá ser:
  - Fazer o pop da stack até aparecer um estado que tenha uma entrada na tabela Goto não vazia;
  - Se um dos estados constantes da tabela Goto do estado do topo tiver uma ação válida no próximo token da entrada, fazer o push desse estado. Preferir as ações válidas de deslocamento (shift) sobre as de redução (reduce);
  - Se não existir nenhuma ação válida no próximo token da entrada para um dos estados de Goto do estado do topo descartar símbolos da entrada até que isso aconteça.

35