Bottom-Up Parsing

Maria Rita Di Berardini

Dipartimento di Matematica e Informatica
Universitá di Camerino
mariarita.diberardini@unicam.it

Bottom-up Parsing

- Costruisce l'albero di derivazione dalle foglie alla radice
- Introduciamo una particolare tecnica di parsing bottom-up che prende il nome di parsing SHIFT-REDUCE
- Rappresenta un tentativo di ridurre la stringa in input al non terminale iniziale della grammatica
- Ogni passo di riduzione consiste nell'individuare una sottostringa α che fa match con la parte destra di una qualche produzione $A \longrightarrow \alpha$ e sostituirla con il non terminale A
- Il procedimento corrisponde ad una derivazione rightmost al contrario

Bottom-up Parsing

- Consideriamo la seguente grammatica:

$$\begin{array}{lcl} S & \longrightarrow & aABe \\ A & \longrightarrow & Abc \mid b \\ B & \longrightarrow & d \end{array}$$

e la stringa $abbcde$

$abbcdea$	$abbcde$	$A \longrightarrow b$
$aAbcde$	$aAbcde$	$A \longrightarrow Abc$
$aAde$	$aAde$	$B \longrightarrow d$
$aABe$	$aABe$	$S \longrightarrow aABe$
S		

- Quanto visto corrisponde al seguente derivazione rightmost al contrario

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

Bottom-up Parsing

- Consideriamo di nuovo:
- Quanto visto corrisponde al seguente derivazione rightmost al contrario

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

- b è un HANDLE (maniglia) della stringa $w = abbcde$:
  - fa match con la parte destra della produzione $A \longrightarrow b$
  - se sostituiamo b con A otteniamo la forma sentenziale destra $aAbcde$
  - $aAbcde$ rappresenta un passo lungo una derivazione rightmost di w
- Non tutte le stringhe che matchano con la parte destra di una qualche produzione sono degli handle

$abbcdea$	$abbcde$	$A \longrightarrow b$
$abAcde$	$abAcde$	$B \longrightarrow d$
$abAcBe$		non matcha con la parte destra di nessuna produzione

Definizione di handle

Definition

Un handle di una forma sentenziale destra γ è (1) una produzione $A \to \beta$ e (2) una posizione in γ dove β può essere trovata e rimpiazzata con A per ottenere la precedente forma sentenziale destra lungo una derivazione rightmost di γ

$abbcdea$	$abbcde$	$A \to b$ in posizione 2
$aAbcde$	$aAbcde$	$A \to Abc$ in posizione 2
$aAde$	$aAde$	$B \to d$ in posizione 3
$aABe$	$aABe$	$S \to aABe$ in posizione 1
S		

Un esempio

- Consideriamo la grammatica:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

γ	handle
id + **id** * **id**	$E \rightarrow \mathbf{id}$ (1)
E + **id** * **id**	$E \rightarrow \mathbf{id}$ (3)
E + E * **id**	$E \rightarrow \mathbf{id}$ (5)
E + E * E	$E \rightarrow E * E$ (3)
E + E	$E \rightarrow E + E$ (1)
E	

$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + E * E \Rightarrow_{rm} E + E * \mathbf{id} \Rightarrow_{rm} E + \mathbf{id} * \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id} * \mathbf{id}$$

Derivazioni rightmost e handle

- In questo contesto una derivazione rightmost al contrario può essere ottenuta per "HANDLE PRUNING"
- Una stringa w è generata dalla grammatica solo se $w = \gamma_n$ dove γ_n è la n-esima forma sentenziale destra di una qualche derivazione rightmost per w

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$$

- Cerchiamo un handle β_n in γ_n e lo sostituiamo con il lato sinistro di una qualche produzione $A_n \rightarrow \beta_n$ per ottenere γ_{n-1}
- Ripetiamo il procedimento per $\gamma_{n-1}, \ldots, \gamma_1$ finchè non otteniamo una formale sentenziale che contiene il solo simbolo iniziale S

Implementazione di un parsing Shift-Reduce

- Usa uno stack per memorizzare simboli della grammatica ed un buffer per memorizzare la stringa in input
- Il simbolo \$ viene usato per marcare l'ultima posizione dello stack e la fine dell'input
- La configurazione iniziale per questo tipo di parsing è:

$$\text{STACK:\$} \qquad \text{INPUT:}w\$$$

  dove w è la stringa da parsare.
- Sposta ("shift") zero o più simboli di w nello stack finchè non si forma un handle β

$$\text{STACK:\$}\alpha\ \beta \qquad \text{INPUT:}w'\$$$

- Riduce l'handle β con l'appropriato non terminale (testa della produzione $A \rightarrow \beta$)

$$\text{STACK:\$}\alpha\ A \qquad \text{INPUT:}w'\$$$

Implementazione di un parsing Shift-Reduce

- Ripete questo ciclo di shift-reduce finchè non trova un errore o lo stack non contiene il solo simbolo iniziale S della grammatica e la stringa in input è vuota

$$\text{STACK: } \$S \qquad \text{INPUT: } \$$$

- Esegue quattro possibili operazioni:
  1. shift: sposta il prossimo simbolo in input al top dello stack
  2. reduce: riconosce un handle al top dello stack e decide con quale non terminale rimpiazzarlo
  3. accept: annuncia il completamento del parsing con successo
  4. error: scopre un errore sintattico e avvia una procedura di recovery dell'errore.

Consideriamo la grammatica:

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

e simuliamo (un possibile) comportamento di un parser shift-reduce in corrispondenza della stringa *abbcde*

\$	*abbcde*\$	shift
\$*a*	*bbcde*\$	shift
\$*ab*	*bcde*\$	reduce $A \rightarrow b$
\$*aA*	*bcde*\$	shift
\$*aAb*	*cde*\$	shift ???
\$*aAbc*	*de*\$	reduce $A \rightarrow Abc$
\$*aA*	*de*\$	shift
\$*aAd*	*e*\$	reduce $B \rightarrow d$
\$*aAB*	*e*\$	shift
\$*aABe*	\$	reduce $S \rightarrow aABe$
\$*S*	\$	accept

Viable-Prefixes

- Viable-Prefixes: l'insieme dei prefissi di una forma sentenziale destra che possono comparire sullo stack di un parsing shift-reduce
- Un viable prefix è un prefisso di una forma sentenziale destra che non contiene simboli oltre l'handle più a destra (ogni volta che si forma un handle questo viene rimosso dallo stack)
- Data la solita grammatica

$$\begin{array}{lcl} S & \rightarrow & aABe \\ A & \rightarrow & Abc \mid b \\ B & \rightarrow & d \end{array}$$

e una derivazione rightmost per la stringa $abbcde$

$$S \Rightarrow_{rm} \underline{aABe} \Rightarrow_{rm} aA\underline{d}e \Rightarrow_{rm} a\underline{Abc}de \Rightarrow_{rm} a\underline{b}bcde$$

Viable prefixes: $a, aA, aAB, aABe, aAd, aAb, aAbc, ab$
Non Viable prefixes: $aAde, aAbcd, Abc, abb, aAA$

$	$abbcde$	shift
$a	$bbcde$	shift
$ab	$bcde$	reduce $A \longrightarrow b$
$aA	$bcde$	shift
$aAb	cde	shift ???
$aAbc	de	reduce $A \longrightarrow Abc$
$aA	de	shift
$aAd	e	reduce $B \longrightarrow d$
$aAB	e	shift
$aABe	$	reduce $S \longrightarrow aABe$
$S	$	accept

Tutte le stringhe nello stack sono viable prefixes

Quando shiftare? Quando ridurre?

- Qualche volta in cima allo stack appare una sequenza di simboli che sembra essere un handle (matcha con la parte destra di una qualche produzione)
- ma potremmo non avere shiftato un numero sufficiente di simboli per identifica il vero handle

$$
\begin{array}{lll}
\ldots & & \\
\$aAb & cde\$ & \text{shift ???} \\
\$aAbc & de\$ & \text{reduce } A \longrightarrow Abc \\
\ldots & &
\end{array}
$$

se invece di shiftare riduciamo:

$$
\begin{array}{lll}
\ldots & & \\
\$aAb & cde\$ & \text{reduce } A \longrightarrow b \\
\$aAA & cde\$ & \text{non è un viable prefix} \\
\ldots & &
\end{array}
$$

Quando shiftare? Quando ridurre?

- Una corretta sequenza di passi shift/reduce deve preservare la seguente proprietá

  il contenuto dello stack deve essere un viable prefix

- Abbiamo bisogno di garantire che il modo in cui modifichiamo lo stack preservi la "viable prefix condition": dobbiamo essere in grado di stabilire se una certa sequenza di simboli è un viable prefix o meno
- Fatto importante: se l'input visto fino ad un certo punto può essere ridotto ad un viable prefix allora possiamo dire che la porzione di input vista non contiene errori (al meno per il momento)

Conflitti

- Ci sono grammatiche libere per le quali uno shift-reduce parser non può essere usato
- Conflitti shift-reduce: il parser raggiunge almeno uno stato in cui non è in grado di decidere, in base al contenuto dello stack e del simbolo corrente di input, se fare uno shift oppure ridurre
- Consideriamo la seguente grammatica:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\ & | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\ & | & \textbf{other} \end{array}$$

- Supponiamo di avere un parser shift-reduce nella configurazione:

  STACK: \$ . . . **if** expr **then** stmt INPUT else . . . \$
- In base a quello che segue l'else dell'input, potrebbe essere corretto ridurre **if** *expr* **then** *stmt* a *stmt* oppure fare lo shift dell'**else** poi cercare uno *stmt* per chiudere il condizionale

Conflitti

- Conflitti reduce-reduce: il parser decide che bisogna fare una riduzione, ma non è in grado di stabilire, in base sempre allo stack e al simbolo di lookahead, con quale produzione ridurre
- Le grammatiche per cui il parser si può trovare in una delle situazioni di conflitto appena descritte non appartengono alla classe LR(k) (vengono anche dette grammatiche non LR)

LR parsing

- LR(k) parsing: un metodo di parsing bottom-up efficiente che può essere usato per un'ampia classe di grammatiche libere da contesto
- LR(k)
  - L indica che l'input viene letto da sinistra a destra (Left to right)
  - R indica che viene ricostruita una derivazione rightmost al contrario (Reversed)
  - k indica il numero di simboli di lookahead usati; se (se $k = 1$ spesso viene omesso e quindi si parla semplicemente di LR parsing)

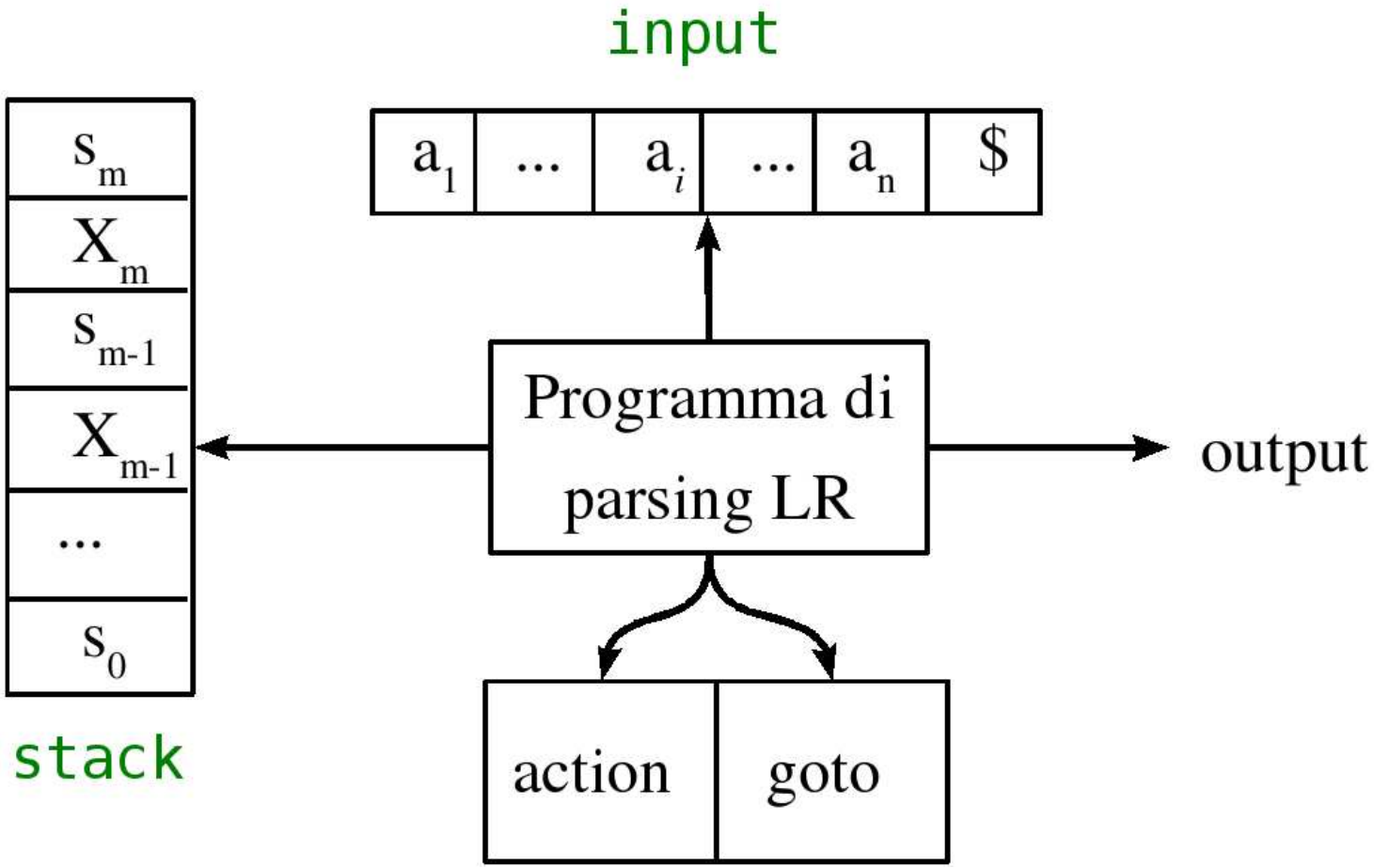
Vantaggi e svantaggi

- Può essere costruito un parser LR per tutti i costrutti dei linguaggi di programmazione per i quali può essere scritta una grammatica libera
- Nonostante sia il metodo di parsing shift-reduce senza backtracking più generale che si conosca, può essere implementato in maniera efficiente tanto quanto altri metodi di parsing shift-reduce meno generali
- La classe di grammatiche che possono essere analizzate da un parser LR include la classe di grammatiche possono essere analizzate con un parser LL predittivo
- La costruzione di un parser LR per un linguaggio di programmazione tipico è troppo complicata per essere fatta a mano
- Bisogna utilizzare un tool apposito, un generatore di parser LR, che applichi gli algoritmi che vedremo e definisca la tabella del parser (Yacc, Javacc)

Vantaggi e svantaggi

- Questi tool sono molto utili anche perchè danno informazione diagnostica se c'è qualche problema
- Ad esempio, in caso di grammatica ambigua, il tool restituisce informazioni utili per determinare dove si crea l'ambiguità)

Parser LR: struttura



Parser LR: struttura

È costituito da:

- Un input e un output
- Uno stack: memorizza stringhe della forma

$$s_0 X_1 s_1 X_2 s_2 \dots X_{m-1} s_{m-1} X_m s_m$$

  dove ogni X_i è un simbolo della grammatica (terminale e non) ed ogni s_i è un simbolo che rappresenta uno stato
- Ogni stato riassume informazioni sullo stack sottostante. Il simbolo al top dello stack è sempre un qualche stato s_m
- Una tabella di parsing divisa in due parti: una parte **action** e una parte **goto**
- Un programma di parsing LR

Parser LR: struttura

- Se s_m è lo stato correntemente al top dello stack ed a_i è il simbolo in input corrente, action$[s_m, a_i]$ può avere uno dei seguenti valori:
  - shift s, metti lo stato s in cima allo stack
  - riduci usando la produzione $A \rightarrow \alpha$
  - accetta
  - errore
- goto prende in input uno stato s ed un simbolo X della grammatica e produce un nuovo stato s' (goto$[s, X] = s'$)
- Il comportamento di un parser LR dipende dalla configurazione corrente
- Una configurazione è un coppia

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

dove: $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$ è l'attuale contenuto dello stack e $a_i a_{i+1} \ldots a_n \$$ è la porzione di input ancora da parsare

Algoritmo di parsing

- Sia $(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$ la configurazione corrente
- La successiva mossa del parser è determinata dallo stato s_m al top dello stack e dal simbolo a_i in input
- Se $\text{action}[s_m, a_i] = \text{shift } s$ il parser esegue uno shift e la configurazione successiva è:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n \$)$$

Algoritmo di parsing

- Sia $(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$ la configurazione corrente
- La successiva mossa del parser è determinata dallo stato s_m al top dello stack e dal simbolo a_i in input
- action$[s_m, a_i]$ = reduce $A \rightarrow \beta$: ha trovato un handle, una sottostringa che fa match con β, elimina l'handle dallo stack
  1. se $r = |\beta|$ (il numero di simboli di β), rimuove $2 * r$ simboli dallo stack: r simboli della grammatica ed r simboli di stato esponendo così lo stato s_{m-r}. Nello stack rimane la stringa: $s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r}$
  2. inserisce nello stack A e lo stato $s = \text{goto}[s_{m-r}, A]$

  La configurazione successiva è

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n \$)$$

Algoritmo di parsing

La successiva mossa del parser è determinata dallo stato s_m al top dello stack e dal simbolo a_i in input

- action$[s_m, a_i]$ =accept: il parsing termina con successo
- action$[s_m, a_i]$ = error: il parser ha scoperto un errore e chiama una procedura di recovery dell'errore

Costruzione della tabella di parsing

Esistono tre metodi principali:

- Simple LR (SLR): è il più debole dei tre in termini di grammatiche alle quali può essere applicato ma è il più facile da implementare
  - tabelle SLR: tabelle di parsing ottenute applicando questo metodo
  - parser SLR: LR parser che usa una tabella SLR
  - grammatiche SLR: grammatiche per le quali è possibile costruire un parser SLR
- LALR (Lookahead LR)
  - tabelle LALR, parser LALR, grammatiche LALR
- LR Canonico
  - tabelle LR canoniche, parser LR canonici, grammatiche LR canoniche

Grammatiche SLR, LALR, LR

Costruzione tabelle SLR

- È il metodo di costruzione di tabelle LR più semplice
- È il metodo meno potente: ha successo per meno grammatiche rispetto a tabelle LALR ed LR canoniche
- È il metodo più semplice da implementare

Item LR(0)

- Abbiamo visto che una generica configurazione di un parser LR è un coppia

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

- Gli stati che possiamo trovare nello stack sono insiemi di **item LR(0)**
- Un item LR(0) per una grammatica G è una qualsiasi produzione di G con un punto ($\bullet$) in qualche posizione della parte destra
- Es: la produzione $A \longrightarrow XYZ$ produce i seguenti LR(0) items:
  - $A \longrightarrow \bullet XYZ$
  - $A \longrightarrow X \bullet YZ$
  - $A \longrightarrow XY \bullet Z$
  - $A \longrightarrow XYZ \bullet$
- Ogni produzione della forma $A \longrightarrow \varepsilon$ produce all'item $A \longrightarrow \bullet$

Item LR(0)

- Se uno stato contiene un item $A \longrightarrow \alpha \bullet \beta$:
  - il parser si apetta di poter ridurre usando la produzione $A \longrightarrow \alpha\beta$
  - ha esaminato una porzione di input derivabile da α
  - si aspetta che l'input contenga una stringa derivabile da β
- Se uno stato contiene un item $A \longrightarrow \alpha \bullet c\beta$ e c é il corrente simbolo in input:
  - il parser shiftera c sullo stack
  - il prossimo stato conterrà l'item $A \longrightarrow \alpha c \bullet \beta$
- Se uno stato contiene un item $A \longrightarrow \alpha\bullet$:
  - il parser ridurrà usando la produzione $A \longrightarrow \alpha$

Item LR(0)

- GLi insiemi di item LR(0) sono gli stati del DFA in grado di riconoscere viable prefixes
- Una collezione di insiemi di item LR(0) prende il nome di collezione canonica LR(0)
- Per costruire una collezione canonica LR(0) abbiamo bisogno di introdurre:
  - il concetto di "grammatica aumentata"
  - due operazioni la closure e la goto

Grammatica aumentata

- Sia G grammatica con simbolo iniziale S
- Aggiungendo un nuovo simbolo non terminale iniziale S' ed una nuova produzione $S' \longrightarrow S$, otteniamo una nuova grammatica G' detta **grammatica aumentata** di G
- Questo accorgimento serve ad indicare al parser la fine del parsing
- Il parsing ha successo se e solo se l'input è terminato e c'è una riduzione con la produzione $S' \longrightarrow S$

L'operazione closure

- Sia I un insieme di LR(0) items per la grammatica G, allora *closure*(I) è l'nsieme di items LR(0) costruito applicando le due seguenti regole:

  1. inizialmente, ogni item di I viene aggiunto in *closure*(I)
  2. se $A \rightarrow \alpha \bullet B\beta \in$ *closure*(I) e $B \rightarrow \gamma$ è una produzione, allora aggiungi in *closure*(I) l'item $B \rightarrow \bullet\gamma$ (se non è già presente)

- In maniera meno formale:
  1. inizialmente, aggiunge in *closure*(I) ogni item di I
  2. cerca items con un non teminale (diciamo B) immediatamente a destra del punto, qualcosa della forma $A \rightarrow \alpha \bullet B\beta$
  3. individuato B e ogni possibile produzione per B, diciamo $B \rightarrow \gamma$, aggiunge in *closure*(I) l'item $B \rightarrow \bullet\gamma$ (se non è già presente)

L'operazione closure – intuizione

- la presenza di un item $A \rightarrow \alpha \bullet B\beta$ in *closure*(I) indica che, ad un certo punto del processo di parsing, ci aspettiamo di individuare nell'input una sottostringa derivabile dal non terminale B
- se $B \rightarrow \gamma$ è una produzione, allora è possibile che ci tale sottostringa sia derivabile da γ
- Per questo motivo aggiungiamo anche l'item $B \rightarrow \bullet\gamma$ in *closure*(I)

Un Esempio

- Consideriamo la seguente grammatica aumentata:

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

$$
\begin{aligned}
closure(\{E' \rightarrow \bullet E\}) \quad = \{ & \\
& E' \rightarrow \bullet E,
\end{aligned}
$$

Un Esempio

- Consideriamo la seguente grammatica aumentata:

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

$$
\begin{aligned}
closure(\{E' \rightarrow \bullet E\}) \quad = \{ & \\
& E' \rightarrow \bullet E, \\
& E \rightarrow \bullet E + T,
\end{aligned}
$$

Un Esempio

- Consideriamo la seguente grammatica aumentata:

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

$$
\begin{aligned}
closure(\{E' \rightarrow \bullet E\}) = \{ & \\
& E' \rightarrow \bullet E, \\
& E \rightarrow \bullet E + T, \\
& E \rightarrow \bullet T,
\end{aligned}
$$

Un Esempio

- Consideriamo la seguente grammatica aumentata:

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

$$
\begin{aligned}
closure(\{E' \rightarrow \bullet E\}) = \{ & \\
& E' \rightarrow \bullet E, \\
& E \rightarrow \bullet E + T, \\
& E \rightarrow \bullet T, \\
& T \rightarrow \bullet T * F, \\
& T \rightarrow \bullet F,
\end{aligned}
$$

Un Esempio

- Consideriamo la seguente grammatica aumentata:

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

$$
\begin{array}{lll}
closure(\{E' \rightarrow \bullet E\}) & = \{ & \\
 & & E' \rightarrow \bullet E, \\
 & & E \rightarrow \bullet E + T, \\
 & & E \rightarrow \bullet T, \\
 & & T \rightarrow \bullet T * F, \\
 & & T \rightarrow \bullet F, \\
 & & F \rightarrow \bullet (E), \\
 & & F \rightarrow \bullet \mathbf{id} \\
 & \} &
\end{array}
$$

L'operazione goto

- Sia I un insieme di LR(0) items ed X un simbolo (terminale o non) della grammatica G

$$goto(I, X) = closure(\{A \rightarrow \alpha X \bullet \beta \mid A \rightarrow \alpha \bullet X\beta \in I\})$$

- Intuitivamente
  1. cerca tutti gli item con il simbolo X immediatamente a destra del punto (items della forma $A \rightarrow \alpha \bullet X\beta$)
  2. sposta il punto in avanti ottenendo un insieme di items della forma $A \rightarrow \alpha X \bullet \beta$
  3. calcola la chiusura dell'insieme di items così ottenuto

L'operazione goto -un esempio

- Se I è un insieme di item che riconosce il viable prefix γ, $goto(I, X)$ è l'insieme di item in grado di riconoscere il viable prefix γX

$goto(\{E \to E \bullet + T\}, +) \quad =$

$closure(\{E \to E + \bullet T\}) \quad = \{$

$E \to E + \bullet T$

L'operazione goto -un esempio

- Se I è un insieme di item che riconosce il viable prefix γ, $goto(I, X)$ è l'insieme di item in grado di riconoscere il viable prefix γX

$$
\begin{aligned}
goto(\{E \to E \bullet + T\}, +) \quad &= \\
closure(\{E \to E + \bullet T\}) \quad &= \{ \\
& \qquad E \to E + \bullet T \\
& \qquad T \to \bullet T * F, \\
& \qquad T \to \bullet F,
\end{aligned}
$$

L'operazione goto -un esempio

- Se I è un insieme di item che riconosce il viable prefix γ, $goto(I, X)$ è l'insieme di item in grado di riconoscere il viable prefix γX

$$
\begin{aligned}
goto(\{E \rightarrow E \bullet + T\}, +) &= \\
closure(\{E \rightarrow E + \bullet T\}) &= \{ \\
&\qquad E \rightarrow E + \bullet T \\
&\qquad T \rightarrow \bullet T * F, \\
&\qquad T \rightarrow \bullet F, \\
&\qquad F \rightarrow \bullet (E), \\
&\qquad F \rightarrow \bullet \mathbf{id} \\
&\}
\end{aligned}
$$

Costruzione della collezione canonica di LR(0) items

procedure items(G'); G' è una grammatica aumentata
begin
    $C := \{I_0 = closure(\{S' \rightarrow \bullet S\})\}$
    repeat
        for each (insieme di item I in C e simbolo X tali che
                $goto(I, X)$ non è vuoto) do
                aggiungi $goto(I, X)$ a C
        until non possono essere aggiunti nuovi insiemi di item a C
end;

Collezione canonica: un esempio

Consideriamo la seguente grammatica aumentata:

$$S' \to S \quad S \to X$$
$$X \to (X) \quad X \to ()$$

$$I_0 = closure(\{S' \to \bullet S\}) = \{ \quad S' \to \bullet S,\ S \to \bullet X,\ X \to \bullet(X),\ X \to \bullet() \ \}$$

$$goto(I_0, S) = closure(\{S' \to S\bullet\}) = \{S' \to S\bullet\} = I_1$$

$$goto(I_0, X) = closure(\{S \to X\bullet\}) = \{S \to X\bullet\} = I_2$$

$$got(I_0, () = \quad closure(\{X \to (\bullet X), X \to (\bullet)\}) = \{X \to (\bullet X),\ X \to (\bullet),\ X \to \bullet(X),\ X \to \bullet() \ \} = I_3$$

Collezione canonica: un esempio

$$I_3 = \begin{array}{l} closure(\{X \to (\bullet X), X \to (\bullet)\}) = \\ \{X \to (\bullet X),\ X \to (\bullet), \\ X \to \bullet(X),\ X \to \bullet()\ \} \end{array}$$

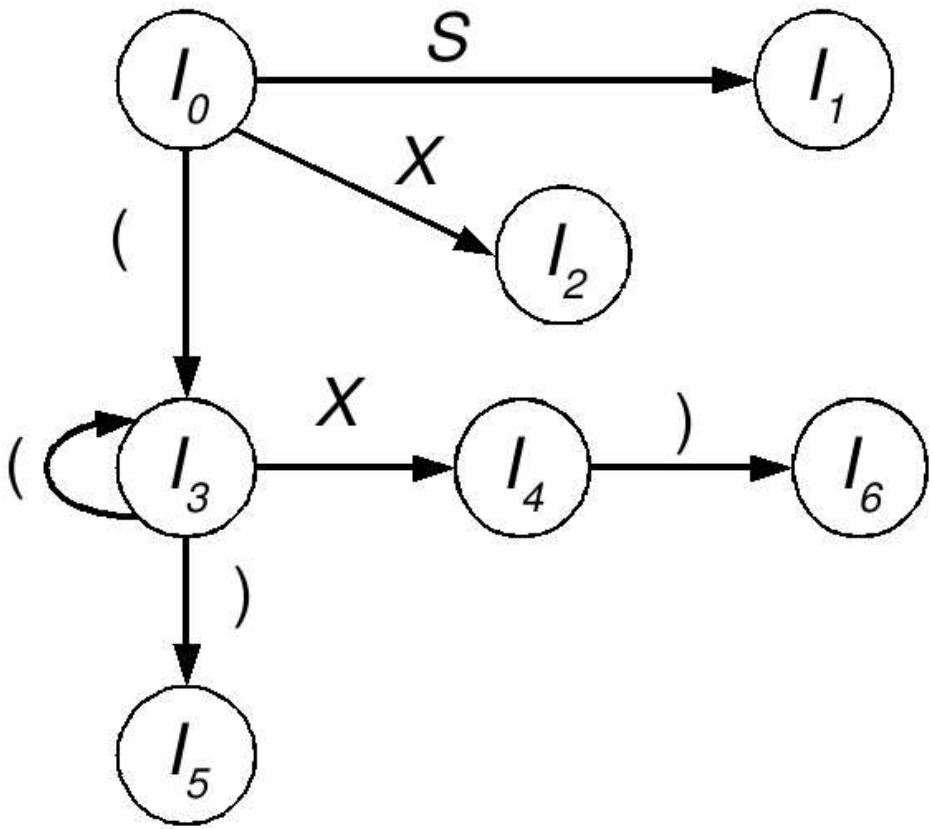
$$goto(I_3, X) = closure(\{X \to (X\bullet)\}) = \{X \to (X\bullet)\} = I_4$$

$$goto(I_3,)) = closure(\{X \to ()\bullet\}) = \{X \to ()\bullet\} = I_5$$

$$goto(I_3, () = \quad closure(\{X \to (\bullet X), X \to (\bullet)\}) = I_3$$

$$goto(I_4,)) = closure(\{X \to (X)\bullet\}) = \{X \to (X)\bullet\} = I_6$$

Automa associato alla funzione *goto*



Automa associato alla funzione *goto*

- Consideriamo, di nuovo, la grammatica aumentata G':

$$S' \to S \qquad S \to X$$
$$X \to (X) \qquad X \to ()$$

- Quali sono i possibili viable prefixes per G? Ricordo che: un viable prefix è un prefisso di una forma sentenziale destra che non contiene simboli oltre l'handle più a destra

$S' \Rightarrow_{rm} S \Rightarrow_{rm} X \Rightarrow_{rm} ()$ S, X, (, ()
$S' \Rightarrow_{rm} S \Rightarrow_{rm} X \Rightarrow_{rm} (X) \Rightarrow_{rm} (())$ (X, (X), ((, (()
$S' \Rightarrow_{rm} S \Rightarrow_{rm} X \Rightarrow_{rm} (X) \Rightarrow_{rm} ((X)) \Rightarrow_{rm} ((()))$ ((X, ((X), (((, (((()
$S' \Rightarrow_{rm} S \Rightarrow_{rm} X \Rightarrow_{rm} (X) \Rightarrow_{rm}\Rightarrow_{rm} ({}^{n}(X)){}^{n} \Rightarrow_{rm} ({}^{n}(()){}^{n})$

Automa associato alla funzione *goto*

- Sia $C = \{I_0, I_1, \ldots, I_n\}$ la collezione canonica di item LR(0) per una grammatica aumentata G'
- L'automa associato alla funzione *goto*, in cui I_0 è lo stato iniziale ed $I_1, \ldots, I_n$ sono stati finali, è un DFA in grado di riconoscere tutti i viable prefixes per G'
- L'algoritmo aveva intenzione di costruire proprio un automa del genere

Items validi

- Un item $A \longrightarrow \beta_1 \bullet \beta_2$ si dice valido per il viable prefix $\alpha\beta_1$ se esiste una derivazione rightmost della forma

$$S' \Rightarrow_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2 w$$

- Assumiamo $A \longrightarrow \beta_1 \bullet \beta_2$ valido per $\alpha\beta_1$:
  - se $\beta_2 \neq \varepsilon$ allora dobbiamo completare la formazione del handle sullo stack; la successiva operazione sarà uno shift
  - se $\beta_2 = \varepsilon$ allora $A \longrightarrow \beta_1$ è un handle e quindi dobbiamo ridurre usando questa produzione
- Abbiamo un conflitto quando due diversi item validi per uno stesso viable prefix dicono cose diverse: uno dice shifta e l'altro riduci
- Alcuni di questi conflitti possono essere risolti con altre metodologie di costruzione della parsing table, altri guardando il prossimo simbolo in input, altri, infine, non possono essere risolti affatto

Items validi

Come trovare gli item validi per un certo viable prefix???

Theorem

L'insieme degli item validi per un viable prefix γ è esattamente l'insieme di items raggiunto, a partire dallo stato iniziale, lungo un cammino etichettato con γ del DFA rappresentato dalla funzione goto fra gli stati della collezione canonica LR(0)

In base a questo teorema, l'insieme degli item validi per il viable prefix $(^n($, con $n \geq 0$, è

$$I_3 = \{X \to (\bullet X), X \to (\bullet), X \to \bullet(X), X \to \bullet()\}$$

Items validi

- L'item $\overbrace{X}^{A} \to \overbrace{(}^{\beta_1} \bullet \overbrace{X)}^{\beta_2}$ è valido per $\overbrace{(^n}^{\alpha}\overbrace{(}^{\beta_1}$; infatti:

$$S' \stackrel{*}{\Rightarrow}_{rm} \underbrace{(^n}_{\alpha}\overbrace{X}^{A}\underbrace{)^n}_{w} \Rightarrow_{rm} \underbrace{(^n}_{\alpha}\overbrace{(}^{\beta_1}\overbrace{X)}^{\beta_2}\underbrace{)^n}_{w}$$

- L'item $\overbrace{X}^{A} \to \bullet \overbrace{(X)}^{\beta_2}$ (qui $\beta_1 = \varepsilon$) è valido per $\overbrace{(^n(}^{\alpha}$; infatti:

$$S' \stackrel{*}{\Rightarrow}_{rm} \underbrace{(^n(}_{\alpha}\overbrace{X}^{A}\underbrace{))^n}_{w} \Rightarrow_{rm} \underbrace{(^n(}_{\alpha}\overbrace{(X)}^{\beta_2}\underbrace{))^n}_{w}$$

Items validi

- L'item $\overbrace{X}^{A} \to \overbrace{(}^{\beta_1} \bullet \overbrace{)}^{\beta_2}$ è valido per $\overbrace{(^n}^{\alpha}\overbrace{(}^{\beta_1}$; infatti:

$$S' \stackrel{*}{\Rightarrow}_{rm} \underbrace{(^n}_{\alpha}\overbrace{X}^{A}\underbrace{)^n}_{w} \Rightarrow_{rm} \underbrace{(^n}_{\alpha}\overbrace{(}^{\beta_1}\overbrace{)}^{\beta_2}\underbrace{)^n}_{w}$$

- L'item $\overbrace{X}^{A} \to \bullet \overbrace{()}^{\beta_2}$ (qui $\beta_1 = \varepsilon$) è valido per $\overbrace{(^n(}^{\alpha}$; infatti:

$$S' \stackrel{*}{\Rightarrow}_{rm} \underbrace{(^n(}_{\alpha}\overbrace{X}^{A}\underbrace{))^n}_{w} \Rightarrow_{rm} \underbrace{(^n(}_{\alpha}\overbrace{()}^{\beta_2}\underbrace{))^n}_{w}$$

Algoritmo

Ingredienti:

- Una grammatica aumentata G'
- Il DNA che riconosce i viable prefixes di G' (la funzione *goto*)
- La FOLLOW(A) per ogni non terminale A di G'

Output: la tabella di parsing (se la tabella ottenuta ha almeno un'entrata multidefinita la grammatica non è SLR(1))

Algoritmo

1. Costruisci la collezione canonica $C = \{I_0, I_1, \ldots, I_n\}$ per G'
2. Lo stato s_j del parser corrisponde all'insieme di items I_j
3. La parte *goto* della tabella, per ogni stato s_j e per ogni non terminale A, è costruita dalla funzione *goto* come segue:

   se $goto(I_j, A) = I_k$ (funzione) allora $goto[s_j, A] = s_k$ (tabella)
4. La parte *action* della tabella è costruita come segue:
   - se $A \to \alpha \bullet a\beta \in I_j$ e $goto(I_j, a) = s_k$ allora $action[s_j, a] = \text{shift } s_k$ (qui a è un terminale)
   - se $A \to \alpha\bullet \in I_j$, allora poni $action[s_j, a] = \text{reduce } A \to \alpha$ per ogni $a \in \text{FOLLOW}(A)$
   - se $S' \to S\bullet \in I_j$, allora poni $action[s_j, \$] = \text{accept}$
5. Ogni entrata indefinita corrisponde ad un errore
6. Lo stato iniziale corrisponde all'insieme di items contenente $S' \to \bullet S$

Costruzione tabella: un esempio

Consideriamo, di nuovo, la grammatica aumentata:

$$(1)\ S' \to S \quad (2)\ S \to X \quad (3)\ X \to (X) \quad (4)\ X \to ()$$

$$I_0 = \{S' \to \bullet S,\ S \to \bullet X,\ X \to \bullet(X),\ X \to \bullet()\ \}$$

$$goto(I_0, S) = I_1 = \{S' \to S\bullet\}$$
$$goto(I_0, X) = I_2 = \{S \to X\bullet\}$$
$$goto(I_0, () = I_3 = \{X \to (\bullet X),\ X \to (\bullet),\ X \to \bullet(X),\ X \to \bullet()\ \}$$

$$goto(I_3, X) = I_4 = \{X \to (X\bullet)\}$$
$$goto(I_3, () = I_3$$
$$goto(I_3,)) = I_5 = \{X \to ()\bullet\}$$

$$goto(I_4,)) = I_6 = \{X \to (X)\bullet\}$$

$$\text{FOLLOW}(S') = \text{FOLLOW}(S) = \{\$\} \text{ e FOLLOW}(X) = \{), \$\}$$

Costruzione tabella: un esempio

- Il parser ha 7 possibili stati $s_0, s_1, \ldots, s_6$ corrispondenti agli insiemi di item $I_0, I_1, \ldots, I_6$; lo stato iniziale è quello che corrisponde a I_0, cioè s_0
- Consideriamo l'item $I_0 = \{S' \to \bullet S,\ S \to \bullet X,\ X \to \bullet(X),\ X \to \bullet() \}$:
  - $goto(I_0, S) = I_1$ implica $goto[s_0, S] = s_1$
  - $goto(I_0, X) = I_2$ implica $goto[s_0, X] = s_2$
  - $X \to \bullet(X) \in I_0$ e $goto(I_0, () = I_3$ implicano $action[s_0, (] =$ shift s_3 (per brevità sulla tabella scriveremo s3)
  - $X \to \bullet() \in I_0$ e $goto(I_0, () = I_3$ implicano $action[s_0, (] =$ shift s_3 (di nuovo s3)
  - I_0 non ha item della forma $A \to \alpha\bullet$

Costruzione tabella: un esempio

- $I_1 = \{S' \to S\bullet\}$: settiamo una sola entrata della parte action, $action[s_1, \$] =$ accept
- $I_2 = \{S \to X\bullet\}$: in questo caso, poichè la FOLLOW$(S) = \{\$\}$, poniamo $action[s_2, \$] =$ reduce $S \to X$ (per comodità sulla tabella scriveremo r2, dove 2 è il numero della produzione $S \to X$)
- $I_3 = \{X \to (\bullet X),\ X \to (\bullet),\ X \to \bullet(X),\ X \to \bullet() \}$
  - $goto(I_3, X) = I_4$ implica $goto[s_3, X] = s_4$
  - Poichè $X \to \bullet(X) \in I_3$ e $goto(I_3, () = I_3$, $action[s_3, [] =$ shift s_3 (s3)
  - $X \to \bullet() \in I_3$ e $goto(I_3, () = I_3$ implicano $action[s_3, [] =$ shift s_3 (s3)
  - $X \to (\bullet) \in I_3$ 4 $goto(I_3,)) = I_5$ implicano $action[s_3,)] =$ shift s_5 (s5)

Costruzione tabella: un esempio

- $I_4 = \{X \rightarrow (X\bullet)\}$: in questo caso, poichè $goto(I_4,)) = I_6$, abbiamo $action[s_4,)] =$ shift s_6 (s6)
- $I_5 = \{X \rightarrow ()\bullet\}$: in questo caso, poichè la FOLLOW$(X) = \{\$,)\}$, poniamo $action[s_5, \$] = action[s_5,)] =$ reduce $X \rightarrow ()$ (r4, dove 4 è il numero della produzione $X \rightarrow ()$)
- $I_6 = \{X \rightarrow (X)\bullet\}$: in maniera analoga, FOLLOW$(X) = \{\$,)\}$ implica poniamo $action[s_6, \$] = action[s_6,)] =$ reduce $X \rightarrow (X)$ (r3, dove 3 è il numero della produzione $X \rightarrow (X)$)

La tabella

	(	)	$	S	X
s_0	s3			1	2
s_1			acc		
s_2			r2		
s_3	s3	s5			4
s_4		s6			
s_5		r4	r4		
s_6		r3	r3		

Parsing della stringa ((()))

STEP	STACK	INPUT	AZIONE
1	s_0	(((()))$	shift 3
2	$s_0(s_3$	((()))$	shift 3
3	$s_0(s_3(s_3$	(()))$	shift 3
4	$s_0(s_3(s_3(s_3$	()))$	shift 5
5	$s_0(s_3(s_3(s_3)s_5$	))$	reduce 4 $X \rightarrow ()$
6	$s_0(s_3(s_3Xs_4$	))$	shift 6
7	$s_0(s_3(s_3Xs_4)s_6$	)$	reduce 3 $X \rightarrow (X)$
8	$s_0(s_3Xs_4$	)$	shift 6
9	$s_0(s_3Xs_4)s_6$	$	reduce 3 $X \rightarrow (X)$
10	s_0Xs_2	$	reduce 2 $S \rightarrow X$
11	s_0Ss_1	$	accept

Grammatiche non SLR

- Grammatiche ambigue non possono essere LR e quindi non SLR
- Tuttavia ci sono grammatiche non ambigue che non sono SLR
- Consideriamo la seguente grammatica G'

$$
\begin{aligned}
S' &\to S \\
S &\to L = R \\
S &\to R \\
L &\to *R \\
L &\to \mathbf{id} \\
R &\to L
\end{aligned}
$$

è non ambigua ma nemmeno SLR

Collezione canonica per G'

- Costruiamo la collezione canonica di item LR(0) per G':

$$
\begin{aligned}
S' &\to S \\
S &\to L = R \\
S &\to R \\
L &\to *R \\
L &\to \mathbf{id} \\
R &\to L
\end{aligned}
$$

- $I_0 = \{S' \to \bullet S, S \to \bullet L = R, S \to \bullet R, L \to \bullet * R, L \to \bullet \mathbf{id}, R \to \bullet L\}$
- $I_1 = goto(I_0, S) = \{S' \to S\bullet\}$
- $I_2 = goto(I_0, L) = \{S \to L\bullet = R, R \to L\bullet\}$
- $I_3 = goto(I_0, R) = \{S \to R\bullet\}$
- $I_4 = goto(I_0, *) = \{L \to * \bullet R, R \to \bullet L, L \to \bullet * R, L \to \bullet \mathbf{id}\}$
- $I_5 = goto(I_0, id) = \{L \to \mathbf{id}\bullet\}$

Collezione canonica per G'

- Costruiamo la collezione canonica di item LR(0) per G':

$$
\begin{aligned}
S' &\to S \\
S &\to L = R \\
S &\to R \\
L &\to *R \\
L &\to \mathbf{id} \\
R &\to L
\end{aligned}
$$

- $I_6 = goto(I_2, =)\{S \to L = \bullet R, R \to \bullet L, L \to \bullet * R, L \to \bullet \mathbf{id}\}$
- $I_7 = goto(I_4, R) = \{L \to *R\bullet\}$
- $I_8 = goto(I_4, L) = \{R \to L\bullet\}$
- $goto(I_4, *) = I_4$, $goto(I_4, id) = I_5$
- $goto(I_6, R) = \{S \to L = R\bullet\} = I_9$
- $goto(I_6, L) = I_8$, $goto(I_6, *) = I_4$, $goto(I_6, \mathbf{id}) = I_5$

Collezione canonica per G'

- Analizziamo l'item $I_2 = goto(I_0, L) = \{S \to L\bullet = R, R \to L\bullet\}$
- Poichè $S \to L\bullet = R \in I_2$ e $goto(I_2, =) = I_6$, possiamo aggiungere l'entrata $action[s_2, =] =$ shift s_6
- Inoltre, poichè $R \to L\bullet \in I_2$ e FOLLOW$(R) = \{\$, =\}$, abbiamo anche $action[s_2, =] = action[s_2, \$] =$ reduce $R \to L$
- Abbiamo un conflitto shift/reduce in corrispondenza dell'entrata $action[s_2, =]$
- Il parser non ha abbastanza informazioni per decidere cosa fare se, nello stato s_2, legge il simbolo $=$

La tabella

	id	+	*	=	$	S	L	R
s_0	s5		s4			1	2	3
s_1					acc			
s_2				s6/r5	r5			
s_3					r2			
s_4	s5		s4				8	7
s_5				r4	r4			
s_6	s5		s4				8	
s_7				r3	r3			
- s_8				r5	r5			
s_5					r1			

Parsing della stringa $*\mathbf{id} = \mathbf{id}$

STEP	STACK	INPUT	AZIONE
1	s_0	$*\mathbf{id} = \mathbf{id}\$$	shift 4
2	$s_0 * s_4$	$\mathbf{id} = \mathbf{id}\$$	shift 5
3	$s_0 * s_4 \mathbf{id} s_5$	$= \mathbf{id}\$$	reduce $L \rightarrow \mathbf{id}$
4	$s_0 * s_4 L s_8$	$= \mathbf{id}\$$	reduce $R \rightarrow L$
5	$s_0 * s_4 R s_7$	$= \mathbf{id}\$$	reduce $L \rightarrow *R$
6	$s_0 L s_2$	$= \mathbf{id}\$$	s6/r5 assumiamo di ridurre con $R \rightarrow L$
6	$s_0 R s_3$	$= \mathbf{id}\$$	errore

Nel caso in cui decidiamo di effettuare al riduzione arriviamo in uno stato in cui il prossimo handle dovrebbe cominciare con R= il che non è possibile viste le produzione della nostra grammatica

Problema della metodologia SLR

- In generale, dato uno stato s_j, una tabella SLR indica una riduzione con una produzione $A \rightarrow \alpha$ se l'insieme di item I_j contiene l'item $A \rightarrow \alpha\bullet$ e il simbolo in input corrente a appartiene FOLLOW(A)
- Tuttavia, in alcune situazioni, quando lo stato s_j compare in testa allo stack, il viable prefix $\beta\alpha$ nello stack può non essere seguito da a in nessuna forma sentenziale destra
- Quindi la riduzione $A \rightarrow \alpha$ sarebbe sbagliata con il corrente l'input a

Soluzione: aggiungiamo informazione agli stati

- Ogni stato deve contenere, oltre agli item validi, anche i simboli di input che possono seguire un handle α riducibile al non terminale A
- Gli item sono delle coppie $[A \rightarrow \alpha \bullet \beta, a]$ dove:
  - $A \rightarrow \alpha\beta$ è una produzione ($A \rightarrow \alpha \bullet \beta$ è un LR(0) item) ed
  - a è un simbolo terminale oppure il \$
- Questi item che contengono anche un simbolo terminale sono chiamati item LR(1) (1 indica che si considera un simbolo di lookahead, cioè la seconda componente è un solo simbolo)
- Se l'item è della forma $[A \rightarrow \alpha\bullet, a]$ allora si esegue una riduzione solo se il simbolo in input è a
- Non riduciamo per tutti i simboli in FOLLOW(A), ma per un sottoinsieme (proprio) di questi