Compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2024/2025

1. Analisi sintattica (PARTE TERZA): Parsing shift-reduce
   - Generalità sul parsing bottom-up
   - Architettura di un parser LR
   - Parsing SLR(1)

Compilatori

Elementi generali

- Un parser generico di tipo bottom-up procede operando una sequenza di riduzioni a partire dalla stringa di input $\alpha_0 = \alpha$ e cercando di risalire così all'assioma iniziale.
- Al generico passo di riduzione il parser individua, nella stringa corrente α_i, un'opportuna sottostringa β che corrisponde alla parte destra di una produzione $A \rightarrow \beta$ e sostituisce β con A, così *riducendo* α_i ad α_{i+1}:
$$\alpha_i = \gamma\beta\delta, \qquad \alpha_{i+1} = \gamma A\delta$$
- Il processo termina con successo se, per un opportuno valore di i, risulta $\alpha_i = \mathcal{S}$.
- Nell'ambito del processo di riduzione il parser può costruire (dal basso verso l'alto) un albero di derivazione e/o produrre direttamente codice.

Parsing "Shift-reduce"

- Un parser *shift-reduce* è un parser di tipo bottom-up che fa uso di uno stack nel quale vengono memorizzati simboli (terminali o non terminali) della grammatica.
- Il nome deriva dal fatto che le due operazioni fondamentali eseguite del parser sono dette, appunto, *shift* (spostamento) e *reduce* (riduzione).
  - L'operazione shift legge un simbolo dallo stream di input e lo inserisce sullo stack.
  - L'operazione reduce sostituisce sullo stack gli ultimi k simboli inseriti (poniamo $X_1, \ldots, X_k$, con X_k sulla cima) con il simbolo A, naturalmente se esiste la produzione $A \rightarrow X_1 \ldots X_k$.
- Le sole altre operazioni che il parser esegue sono: accettare l'input o segnalare una condizione di errore.

Riduzioni e shift

- Per un parser di questo tipo la difficoltà consiste proprio nel decidere quando operare la riduzione e quando invece è necessario procedere con uno shift.
- Infatti, non è sempre vero che, quando sullo stack c'è la parte destra di una produzione, bisogna operare la riduzione.
- Un esempio relativo alla "solita" grammatica

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T \times F \mid F \\
F &\rightarrow \mathbf{n} \mid (E)
\end{aligned}
$$

chiarisce questo punto.

Esempio

- Consideriamo il problema del riconoscimento della stringa $\mathbf{n} \times \mathbf{n}$.
- La seguente tabella illustra il contenuto dello stack e l'input ancora da leggere se venisse applicato l'approccio "greedy" (errato) appena delineato.

Indice i	Stack	Input	Azione	α_i
0	$\$$	$\mathbf{n} \times \mathbf{n}\$$	shift	$\mathbf{n} \times \mathbf{n}\$$
0	$\$\underline{\mathbf{n}}$	$\times\mathbf{n}\$$	reduce	$\mathbf{n} \times \mathbf{n}\$$
1	$\$\underline{F}$	$\times\mathbf{n}\$$	reduce	$F \times \mathbf{n}\$$
2	$\$\underline{T}$	$\times\mathbf{n}\$$	reduce	$T \times \mathbf{n}\$$
3	$\$E$	$\times\mathbf{n}\$$	shift	$E \times \mathbf{n}\$$
3	$\$E\times$	$\mathbf{n}\$$	shift	$E \times \mathbf{n}\$$
3	$\$E \times \underline{\mathbf{n}}$	$\$$	reduce	$E \times \mathbf{n}\$$
4	$\$E \times \underline{F}$	$\$$	reduce	$E \times F\$$
5	$\$E \times \underline{T}$	$\$$	reduce	$E \times T\$$
6	$\$E \times E$	$\$$	error	$E \times E\$$

Handle (maniglie)

- Un parser di tipo shift-reduce deve individuare, come sottostringhe da ridurre a non terminale, esattamente quelle sequenze (e quelle produzioni) usate nella derivazione canonica destra.
- Tali sequenze devono inoltre essere individuate "nell'ordine giusto", e cioè l'ordine rovesciato rispetto alla corrispondente derivazione canonica destra.
- Queste sequenze (ma meglio sarebbe dire "produzioni") vengono chiamate *handle* (maniglie), di modo che il problema centrale della realizzazione di un tale parser può essere espresso sinteticamente come il problema di individuare le handle.

Esempio corretto

- La corretta riduzione per la stringa **n** $\times$ **n** è indicata di seguito

Indice i	Stack	Input	Azione	α_i
0	$\$$	**n** $\times$ **n**$\$$	shift	**n** $\times$ **n**$\$$
0	$\$\underline{\mathbf{n}}$	$\times$**n**$\$$	reduce	**n** $\times$ **n**$\$$
1	$\$\underline{F}$	$\times$**n**$\$$	reduce	$F \times$ **n**$\$$
2	$\$T$	$\times$**n**$\$$	shift	$T \times$ **n**$\$$
2	$\$T\times$	**n**$\$$	shift	$T \times$ **n**$\$$
3	$\$T \times \underline{\mathbf{n}}$	$\$$	reduce	$T \times$ **n**$\$$
4	$\$\underline{T \times F}$	$\$$	reduce	$T \times F\$$
5	$\$\underline{T}$	$\$$	reduce	$T\$$
6	$\$E$	$\$$	accept	$E\$$

- Si noti che, leggendo l'ultima colonna dal basso verso l'alto, in corrispondenza delle operazioni "reduce" si rivela la derivazione canonica destra di **n** $\times$ **n**.

Osservazioni

- Ad ogni dato istante, l'attuale forma di frase (la stringa α_i) si trova "parte sullo stack e parte ancora sullo stream di input".
- Più precisamente, se lo stack contiene una stringa $\alpha\beta_1$ (dal basso verso l'alto) e lo stream di input contiene la stringa $\beta_2\gamma$, allora la forma di frase "corrente" nella derivazione destra è $\alpha\beta_1\beta_2\gamma$.
- Se la prossima handle è la produzione $A \to \beta_1\beta_2$ allora:
  - se $\beta_2 = \epsilon$ allora la prossima mossa è la riduzione;
  - se $\beta_2 \neq \epsilon$ allora la prossima mossa è uno shift;
- Se la prossima handle non è $A \to \beta_1\beta_2$ allora il parser esegue uno shift o dichiara errore (come vedremo).

Il cuore computazionale del problema

- La difficoltà di progettazione di un parser shift-reduce parser sta tutta nella capacità di riconoscere quando è corretto operare uno shift e quando invece è corretto operare una riduzione.
- Il problema coincide con quello di determinare esattamente le handle. Infatti, se fossimo in grado di risolvere quest'ultimo sapremmo sempre quando operare uno shift e quando eseguire una riduzione.
- Dovremmo infatti "ridurre" quando e solo quando una maniglia appare sulla cima dello stack.
- Sfortunatamente ci sono grammatiche per le quali il paradigma shift-reduce non è applicabile, ad esempio grammatiche ambigue.

Osservazione)

- Si noti, di passaggio, che se considerassimo derivazioni canoniche sinistre, anziché destre, non potremmo contare sulla proprietà che la (o una) handle prima o poi si trovi sulla cima dello stack
- Si consideri il solito semplice esempio

Indice i	Stack	Input	Azione	Stringa α_i
0	$\$$	$\mathbf{n}+\underline{\mathbf{n}}\$$	shift	$\mathbf{n}+\mathbf{n}\$$
0	$\$\mathbf{n}$	$+\underline{\mathbf{n}}\$$	shift	$\mathbf{n}+\mathbf{n}\$$
0	$\$\mathbf{n}+$	$\underline{\mathbf{n}}\$$	shift	$\mathbf{n}+\mathbf{n}\$$
0	$\$\mathbf{n}+\underline{\mathbf{n}}$	$\$$	reduce	$\mathbf{n}+\mathbf{n}\$$
1	$\$\mathbf{n}+\underline{F}$	$\$$	reduce	$\mathbf{n}+F\$$
2	$\$\underline{\mathbf{n}}+T$	$\$$	reduce	$\mathbf{n}+T\$$
3	$\$\underline{F}+T$	$\$$	reduce	$F+T\$$
4	$\$\underline{T}+T$	$\$$	reduce	$T+T\$$
5	$\$\underline{E+T}$	$\$$	reduce	$E+T\$$
6	$\$E$	$\$$	accept	$E\$$

Prefissi ammissibili (viable prefix)

- Supponiamo ora di disporre già di un parser che non si lascia mai "sfuggire" la handle appena questa compare sulla cima dello stack e dunque che è sempre in grado di ricostruire la corretta derivazione canonica destra di una stringa del linguaggio
- Questo vuol dire che, per ogni data forma di frase α_i, la parte che sta sullo stack è un prefisso di α_i che non oltrepassa mai la handle stessa
- Si faccia riferimento all'esempio riportato nella slide 9 per visualizzare la situazione.
- Vediamo alcuni casi
  - Con la forma di frase $T \times F$ la handle è sulla cima dello stack e coincide con la forma di frase
  - Con la forma di frase $T \times$ **n** la handle è (indice $i = 2$) parte sullo stack e parte ancore nell'input ovvero $(i = 3)$ sulla cima dello stack

Prefissi ammissibili (viable prefix)

- Per un tale parser, i prefissi delle varie forme di frase che si possono leggere via via sullo stack sono detti *ammissibili*
- Senza far riferimento allo stack, possiamo dire che, in una derivazione canonica destra, i *prefissi ammissibili* sono quei prefissi di una qualsiasi forma di frase α che non superano l'ultimo carattere della handle in α
- Se la grammatica è ambigua, la definizione va però corretta perchè potrebbero esserci più handle e in tal caso il prefisso è ammissibile in α se non supera la handle più a destra in α. Un esempio:

$$\begin{array}{ccccccc} E & \Rightarrow & \underline{E+E} & \Rightarrow & E+\underline{\mathbf{id}} & \Rightarrow & \underline{E+E}+\mathbf{id} \\ E & \Rightarrow & \underline{E+E} & \Rightarrow & E+\underline{E+E} & \Rightarrow & E+E+\underline{\mathbf{id}} \end{array}$$

Qui il prefisso $E+E+$ è ammissibile per la forma di frase $E+E+\mathbf{id}$ anche se “supera” la handle $E+E$ perché nella forma di frase c'è ancora una handle più a destra

Compilatori

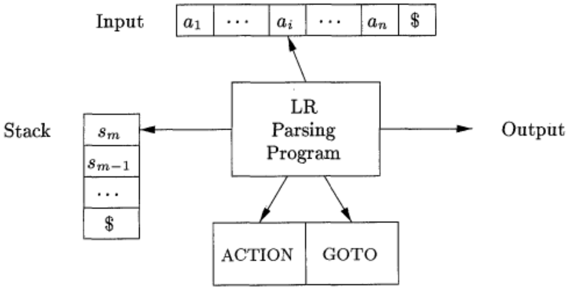
1. Analisi sintattica (PARTE TERZA): Parsing shift-reduce
   - Generalità sul parsing bottom-up
   - Architettura di un parser LR
   - Parsing SLR(1)

Parser LR

- Si tratta di una classe di parser di tipo shift-reduce che analizzano l'input da sinistra a destra (L) e che cercano di ricostruire una derivazione canonica destra (R) per l'input medesimo
- La diversa capacità di effettuare il parsing dipende dall'informazione contenuta in apposite tabelle di parsing che guidano il comportamento del programma.
- In questi appunti analizzeremo un solo tipo di parser LR, il più semplice, che prende (non a caso) il nome di (parser) $SLR(1)$.
- Per prima cosa vedremo però la "program structure" comune.

Struttura di un parser LR

- Un parser LR è caratterizzato da un programma di controllo che ha accesso ad uno stack e ad una tabella di parsing, oltre che a opportuni supporti di input e output.



Struttura di un parser LR (continua)

- Le tabelle prescrivono il comportamento del programma di controllo in funzione del contenuto dello stack e dei primi k caratteri presenti in input (per noi $k = 1$).
- Lo stack, a differenza dei parser shift-reduce visti precedentemente, contiene stati anziché simboli.
- Tuttavia, come vedremo, ad ogni stato è associato univocamente un simbolo della grammatica (l'inverso non è necessariamente vero).
- Come nel caso generico di parser shift-reduce, possiamo quindi ricostruire la forma di frase corrente (di una derivazione canonica destra) utilizzando i simboli memorizzati sullo stack concatenati con i simboli ancora sullo stream di input.

Tabelle di parsing

- Le tabelle di parsing di un parser LR hanno un numero di righe pari al numero di stati dell'automa che costituisce il controllo.
- Le colonne sono indicizzate dai simboli terminali e non terminali. Le colonne relative ai terminali formano quella che viene detta "parte $ACTION$" della tabella, mentre le altre formano la "parte $GOTO$".
- Nella parte action sono previste 4 tipi di azioni:
  - avanzamento di un carattere sullo stream di input e inserimento di uno stato in cima allo stack;
  - esecuzione di una riduzione;
  - accettazione dell'input;
  - rilevamento di un errore.
- La parte GOTO prescrive stati da inserire nello stack.

Funzionamento del parser

- Il funzionamento del parser è definito come segue.
- Inizialmente, lo stack contiene un solo stato (lo stato iniziale, naturalmente).
- Al generico passo, sia q lo stato in cima allo stack e x il prossimo carattere in input.
- Se $ACTION\,[q, x] =$ shift r, il parser avanza il puntatore di input e inserisce lo stato r sullo stack.
- Se $ACTION\,[q, x] =$ reduce i, il parser utilizza la i-esima produzione (secondo una numerazione arbitraria ma prefissata). Più precisamente, se $A \rightarrow \alpha$ è tale produzione, il parser rimuove $k_i = |\alpha|$ stati dallo stack e vi inserisce lo stato $GOTO\,[q', A]$ dove q' è lo stato sulla cima dello stack dopo le k_i rimozioni.
- Il parser si arresta in seguito ad accettazione o errore.

Esempio

- Consideriamo la grammatica che genera sequenze di parentesi bilanciate:

  $S \rightarrow (S)S$ Produzione 1

  $S \rightarrow \epsilon$ Produzione 2

  e consideriamo la seguente tabella di parsing (di cui vedremo più avanti la costruzione):

Stato	*ACTION*			*GOTO*
	(	)	$	S
0	shift 2	reduce 2	reduce 2	1
1			accept	
2	shift 2	reduce 2	reduce 2	3
3		shift 4		
4	shift 2	reduce 2	reduce 2	5
5		reduce 1	reduce 1	

- Consideriamo il comportamento del parser su input $()()$.

Esempio (continua)

Stack	Input	Azione
$\$0$	$()()\$$	shift 2
$\$02$	$)()\$$	reduce $S \to \epsilon$
$\$023$	$)()\$$	shift 4
$\$0234$	$()\$$	shift 2
$\$02342$	$)\$$	reduce $S \to \epsilon$
$\$023423$	$)\$$	shift 4
$\$0234234$	$\$$	reduce $S \to \epsilon$
$\$02342345$	$\$$	reduce $S \to (S)S$
$\$02345$	$\$$	reduce $S \to (S)S$
$\$01$	$\$$	accept

- Si ricordi che la riduzione con $S \to (S)S$ prima rimuove 4 stati dallo stack, quindi inserisce lo stato $GOTO[q', S]$, dove q' è lo stato che rimane in cima allo stack dopo le rimozioni.
- Analogamente, la riduzione con $S \to \epsilon$ rimuove 0 stati.

Compilatori

Parsing SLR(1)

- Il primo tipo di parser LR che analizziamo è detto *Simple LR parser* (o semplicemente *SLR*).
- È caratterizzato da tabelle di parsing di relativamente semplice costruzione (da cui il nome) ma che danno minori garanzie sulla possibilità di analisi di grammatiche libere.
- In altri termini, ci sono diverse grammatiche libere di interesse che non possono essere analizzate con parser SLR (e, segnatamente, $SLR(1)$).
- Si tratta comunque di un caso utile per capire la "logica" di un parser LR.

Item LR(0)

- Per comprendere come un parser SLR(1) prende le proprie decisioni (ovvero per capire come viene costruita la tabella di parsing) iniziamo ad introdurre il concetto di *item* $LR(0)$ per una grammatica G
- Un *item* $LR(0)$ (o semplicemente *item*) di G è una produzione di G in cui nella parte destra viene inserito un punto.
- Una produzione in cui la parte destra contiene k simboli (terminali o non terminali) dà origine a $k+1$ item in cui i punti sono inseriti all'inizio, dopo il primo simbolo,...,dopo l'ultimo simbolo
- Ad esempio, gli item associati alla produzione $S \to (S)S$ della grammatica che riconosce le sequenze bilanciate di parentesi sono: $S \to \cdot(S)S$, $S \to (\cdot S)S$, $S \to (S\cdot)S$, $S \to (S) \cdot S$ e $S \to (S)S\cdot$.
- Ad una produzione tipo $S \to \epsilon$ è associato il solo item $S \to \cdot$.

Item $LR(0)$

- Intuitivamente, il punto inserito in una produzione indica la posizione alla quale siamo arrivati nel processo di riconoscimento della parte destra della produzione stessa.
- Ad esempio, l'item $S \to (S) \cdot S$ indica che nell'input abbiamo riconosciuto una stringa generabile a partire da (S) e che ci "attendiamo" di riconoscere una stringa generabile da S.
- Un item con il puntino in fondo indica quindi che il processo di riconoscimento della parte destra nell'input è completato e dunque che (presumibilmente...) si può operare la riduzione.
- Un'opportuna collezione di insiemi di item $LR(0)$ fornisce i dati fondamentali per la costruzione di un automa utilizzabile per prendere le decisioni di parsing
- Nel caso in questione, in ordine alla costruzione del parser $SLR(1)$, la collezione viene chiamata *collezione canonica di item* $LR(0)$

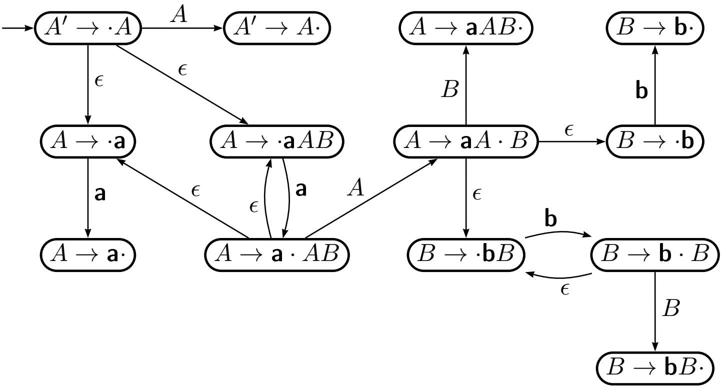
AFND $LR(0)$

- Utilizzando gli item costruiamo ora un automa non deterministico in cui gli stati sono proprio gli item mentre le transizioni sono così determinate:
  - Se uno stato s è associato all'item $\alpha \cdot X\beta$, allora si inserisce una transizione etichettata X da s allo stato t associato all'item $\alpha X \cdot \beta$
  - Se il simbolo X è un non terminale, si inserisce una transizione etichettata ϵ verso tutti gli stati t associati ad item in cui X è la testa della produzione
- Per esemplificare la costruzione, utilizziamo la seguente semplice grammatica G_{AB}, di valenza solo "didattica" (già aumentata con la produzione $A' \to A$)

$$\begin{array}{rcl} A' & \to & A \\ A & \to & \mathbf{a} \mid \mathbf{a}AB \\ B & \to & \mathbf{b} \mid \mathbf{b}B \end{array}$$

AFND $LR(0)$

- Se consideriamo tutti gli stati come stati finali, allora l'automa riconosce tutti e soli i prefissi ammissibili della grammatica



AFND $LR(0)$

- A titolo di esempio, consideriamo la derivazione canonica destra

$$A' \Rightarrow \underline{A} \Rightarrow \underline{\mathbf{a}AB} \Rightarrow \mathbf{a}A\underline{\mathbf{b}B} \Rightarrow \mathbf{a}A\mathbf{b}\underline{\mathbf{b}} \Rightarrow \mathbf{a}\underline{\mathbf{a}}\mathbf{bb}$$

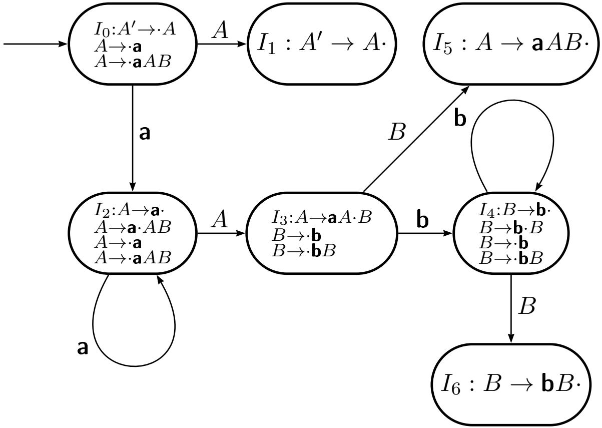
- Per ogni forma di frase, si può verificare che tutti i prefissi che non si estendono a destra della handle (indicata, al solito, con la sottolineatura) sono riconosciuti dall'automa
- Viceversa, nessuna sequenza di transizioni dell'automa è etichettata dai prefissi **aab** e (a maggior ragione) **aabb**, che non sono ammissibili
- Come ulteriore esempio, si provi a verificare che l'automa riconosce i prefissi ammissibili (e solo quelli) presenti nella derivazione canonica destra della stringa $\mathbf{a}^4\mathbf{b}^3$.

AFD $LR(0)$

- Utilizzando la *subset construction*, vista a suo tempo, possiamo poi realizzare l'automa deterministico equivalente
- Nell'automa deterministico gli stati sono costituiti non da singoli item bensì da *collezioni di item*
- Lo stato iniziale, sulla base di quanto "prescritto" dalla subset construction, è la chiusura dello stato dell'AFND corrispondente
- E poiché tutti gli stati dell'AFND sono finali, tali saranno anche tutti gli stati del corrispondente AFD

AFD $LR(0)$

- L'automa deterministico $LR(0)$ per la grammatica G_{AB} è illustrato di seguito:



AFD $LR(0)$

- Si noti che l'intersezione di due stati, ovvero di due collezioni di item, può non essere vuota.
- Ad esempio, l'item $B \rightarrow \cdot\mathbf{b}$ è presente nella collezione I_3 ma anche nella I_5
- Sono le collezioni nel loro insieme che devono essere distinte
- Naturalmente, in casi particolari un item può formare uno stato/collezione da solo.
- Questo è il caso, ad esempio, dell'item $B \rightarrow \mathbf{b}B\cdot$.
- Per una generica grammatica, si definiscono *kernel item* l'item iniziale $\mathcal{S}' \rightarrow \cdot\mathcal{S}$ e tutti gli item che non hanno il punto all'inizio
- *Nonkernel item* sono tutti gli altri, ovvero gli item che hanno il punto all'inizio, ad eccezione di $\mathcal{S}' \rightarrow \cdot\mathcal{S}$

Un secondo esempio

- Per la grammatica "aumentata"

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & (S)S \mid \epsilon \end{array}$$

sono definiti i seguenti insiemi di item:

$$\begin{array}{llcll}
I_0: & S' \rightarrow \cdot S & & I_3: & S \rightarrow (S\cdot)S \\
 & S \rightarrow \cdot(S)S & & & \\
 & S \rightarrow \cdot & & I_4: & S \rightarrow (S)\cdot S \\
 & & & & S \rightarrow \cdot(S)S \\
I_1: & S' \rightarrow S\cdot & & & S \rightarrow \cdot \\
 & & & & \\
I_2: & S \rightarrow (\cdot S)S & & I_5: & S \rightarrow (S)S\cdot \\
 & S \rightarrow \cdot(S)S & & & \\
 & S \rightarrow \cdot & & &
\end{array}$$

Costruzione "diretta" degli insiemi $LR(0)$

- L'automa deterministico $LR(0)$ e, segnatamente, le collezioni item che ne costituiscono gli stati, possono essere costruiti senza passare prima per l'automa non deterministico
- Ne diamo ora una descrizione informale
- La collezione iniziale I_0 contiene l'item $\mathcal{S}' \to \cdot\mathcal{S}$ e tutti gli item ottenuti dalle produzioni di $\mathcal{S}$ inserendo il punto all'inizio.
- Nell'ultimo esempio considerato, si aggiungono a $S' \to \cdot S$ due soli item (perché ci sono due produzioni relative ad S).
- Si procede poi ricorsivamente. Per ogni collezione I_j già formata:
- si considerano tutti i simboli della grammatica immediatamente alla destra del punto in item di I_j.
- Per ogni simbolo così individuato, si forma un gruppo I_k che contiene, inizialmente, gli item ottenuti spostando il punto alla destra del simbolo considerato.

Come costruire gli insiemi $LR(0)$ (continua)

- Ad esempio, fra gli item di I_0 (per la grammatica appena considerata) ci sono due soli simboli alla destra del punto, S e $($:

$$
\begin{aligned}
I_0 : \quad & S' \to \cdot S \\
& S \to \cdot(S)S \\
& S \to \cdot
\end{aligned}
$$

- Per ognuno di essi si creano due nuovi insiemi, I_1 e I_2, che contengono <u>inizialmente</u> un solo item ciascuno:

$$I_1 : \quad S' \to S\cdot$$

$$I_2 : \quad S \to (\cdot S)S$$

Come costruire gli insiemi $LR(0)$ (continua)

- Se il nuovo insieme I_k appena inizializzato contiene item in cui il punto precede un simbolo non terminale A, si aggiungono ad I_k tutti gli item ottenuti dalle produzioni di A inserendo il punto all'inizio.
- Quest'ultima operazione è detta *chiusura* dell'insieme I_k.
- Continuando l'esempio precedente, poiché l'insieme I_2 contiene l'item $S \to (\cdot S)S$, ad esso si aggiungono gli item $S \to \cdot(S)S$ e $S \to \cdot$:

$$\begin{array}{ll} I_2: & S \to (\cdot S)S \\ & S \to \cdot(S)S \\ & S \to \cdot \end{array}$$

- Il procedimento termina quando non ci sono più insiemi di item da considerare.

Funzioni $CLOSURE$ e $GOTO$

- Il procedimento appena descritto (in maniera alquanto discorsiva) può essere sinteticamente ricapitolato facendo uso delle due funzioni $CLOSURE$ e $GOTO$, che lavorano su insiemi di item.
- Dato un insieme di item I, $CLOSURE(I)$ si ottiene aggiungendo (ricorsivamente) ad I item del tipo $B \rightarrow \cdot\gamma$ sotto le seguenti condizioni:
  - in I esista inizialmente un item del tipo $A \rightarrow \alpha \cdot B\beta$, oppure,
  - ad I sia già stato aggiunto un item del tipo $A \rightarrow \cdot B\beta$.
- Il procedimento termina quando non si possono più aggiungere item sulla base delle precedenti regole.

Funzione $CLOSURE(I)$

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

Funzioni $CLOSURE$ e $GOTO$ (continua)

- Se I è un insieme di item e X un simbolo della grammatica $GOTO(I, X)$ è un insieme di item, che chiameremo J, calcolato nel seguente modo:
  - inizialmente di pone $J = \{\}$;
  - per ogni item $A \to \alpha \cdot X\beta$ in I, si aggiunse a J l'item $A \to \alpha X \cdot \beta$;
  - infine si pone $J \leftarrow CLOSURE(J)$.

Insiemi di item $LR(0)$

- Utilizzando le funzione $CLOSURE$ e $GOTO$ possiamo definire con precisione il calcolo degli insiemi di item per una grammatica aumentata.

$C \leftarrow \{CLOSURE(\{\mathcal{S}' \rightarrow \cdot\mathcal{S}\})\}$
repeat
  for each $I \in C$ **do**
    for each $X \in \mathcal{T} \cup \mathcal{N}$ **do**
      if $GOTO(I, X) \neq \{\}$ $\& GOTO(I, X) \notin C$ **then**
        $C \leftarrow C \cup \{GOTO(I, X)\}$
until No new state is added to C

Esempio

- Consideriamo la seguente grammatica (aumentata) che genera il linguaggio $\{\mathtt{a}^n\mathtt{b}^n | n \geq 1\}$:

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & \mathtt{a}S\mathtt{b} \mid \mathtt{ab} \end{array}$$

- L'insieme iniziale di item è $I_0 = CLOSURE\{S' \rightarrow \cdot S\} = \{S' \rightarrow \cdot S, S \rightarrow \cdot \mathtt{a}S\mathtt{b}, S \rightarrow \cdot \mathtt{ab}\}$.
- I simboli immediatamente a destra del punto in I_0 sono S e a, per cui calcoliamo i due insiemi:
  - $I_1 = GOTO(I_0, S) = \{S' \rightarrow S \cdot\}$;
  - $I_2 = GOTO(I_0, \mathtt{a}) = \{S \rightarrow \mathtt{a} \cdot S\mathtt{b}, S \rightarrow \mathtt{a} \cdot \mathtt{b},\ S \rightarrow \cdot \mathtt{a}S\mathtt{b}, S \rightarrow \cdot \mathtt{ab}\}$

Esempio (continua)

- L'insieme I_1 non dà origine ad altri insiemi di item (perché non ci sono simboli a destra del punto).
- Nel'insieme I_2 ci sono tre simboli distinti a destra del punto, per cui formiamo tre insiemi:
  - $I_3 = GOTO(I_2, S) = \{S \rightarrow \mathtt{a}S \cdot \mathtt{b}\}$;
  - $I_4 = GOTO(I_2, \mathtt{b}) = \{S \rightarrow \mathtt{ab}\cdot\}$;
  - $I_5 = GOTO(I_2, \mathtt{a}) = \{S \rightarrow \mathtt{a} \cdot S\mathtt{b}, S \rightarrow \mathtt{a} \cdot \mathtt{b},\ S \rightarrow \cdot\mathtt{a}S\mathtt{b}, S \rightarrow \cdot\mathtt{ab}\}$.
- Tuttavia, I_5 viene "scartato", in quanto coincide con I_2.
- Infine lavorando su I_3 si ottiene ("riusando" il simbolo I_5):
  - $I_5 = GOTO(I_3, \mathtt{b}) = \{S \rightarrow \mathtt{a}S\mathtt{b}\cdot\}$.

Esempio (continua)

- Ricapitolando, gli insiemi $LR(0)$ di item associati alla grammatica sono:

$$
\begin{array}{llcll}
I_0 : & S' \rightarrow \cdot S & \qquad & I_3 : & S \rightarrow \mathtt{a}S \cdot \mathtt{b} \\
 & S \rightarrow \cdot \mathtt{a}S\mathtt{b} & & & \\
 & S \rightarrow \cdot \mathtt{ab} & & I_4 : & S \rightarrow \mathtt{ab}\cdot \\
 & & & & \\
I_1 : & S' \rightarrow S\cdot & & I_5 : & S \rightarrow \mathtt{a}S\mathtt{b}\cdot \\
 & & & & \\
I_2 : & S \rightarrow \mathtt{a} \cdot S\mathtt{b} & & & \\
 & S \rightarrow \mathtt{a} \cdot \mathtt{b} & & & \\
 & S \rightarrow \cdot \mathtt{a}S\mathtt{b} & & & \\
 & S \rightarrow \cdot \mathtt{ab} & & &
\end{array}
$$

Esempio

- Da ultimo, consideriamo la costruzione degli insiemi di item $LR(0)$ per la grammatica aumentata

$$\begin{array}{rcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & (E) \mid \mathbf{n} \end{array}$$

  che, ricordiamo, non è adatta al parsing top-down.
- Nella slide seguente presentiamo direttamente la collezione degli insiemi di item ottenuta applicando l'algoritmo di costruzione degli insiemi di item.

Esempio (continua)

$$\begin{array}{ll} I_0: & E' \to \cdot E \\ & E \to \cdot E + T \\ & E \to \cdot T \\ & T \to \cdot T \times F \\ & T \to \cdot F \\ & F \to \cdot (E) \\ & F \to \cdot \mathbf{n} \end{array}$$

$$\begin{array}{ll} I_1: & E' \to E \cdot \\ & E \to E \cdot + T \end{array}$$

$$\begin{array}{ll} I_2: & E \to T \cdot \\ & T \to T \cdot \times F \end{array}$$

$$\begin{array}{ll} I_3: & T \to F \cdot \end{array}$$

$$\begin{array}{ll} I_4: & F \to (\cdot E) \\ & E \to \cdot E + T \\ & E \to \cdot T \\ & T \to \cdot T \times F \\ & T \to \cdot F \\ & F \to \cdot (E) \\ & F \to \cdot \mathbf{n} \end{array}$$

$$\begin{array}{ll} I_5: & F \to \mathbf{n} \cdot \end{array}$$

$$\begin{array}{ll} I_6: & E \to E + \cdot T \\ & T \to \cdot T \times F \\ & T \to \cdot F \\ & F \to \cdot (E) \\ & F \to \cdot \mathbf{n} \end{array}$$

$$\begin{array}{ll} I_7: & T \to T \times \cdot F \\ & F \to \cdot (E) \\ & F \to \cdot \mathbf{n} \end{array}$$

$$\begin{array}{ll} I_8: & E \to E \cdot + T \\ & F \to (E \cdot) \end{array}$$

$$\begin{array}{ll} I_9: & E \to E + T \cdot \\ & T \to T \cdot \times F \end{array}$$

$$\begin{array}{ll} I_{10}: & T \to T \times F \cdot \end{array}$$

$$\begin{array}{ll} I_{11}: & F \to (E) \cdot \end{array}$$

Automa $LR(0)$

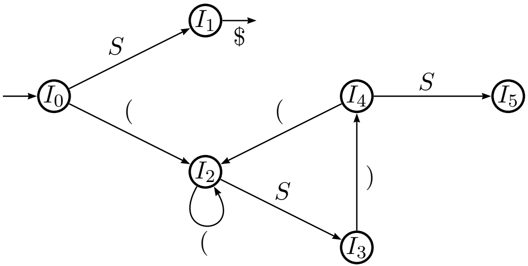
- Come già anticipato, le collezioni di item $LR(0)$ determinate con la procedura appena descritta costituiscono gli stati dell'automa $LR(0)$ (che, a sua volta, è alla base del parsing $SLR(1)$ che stiamo costruendo).
- Per completare la descrizione dell'automa è necessario definire la funzione δ di transizione.
- In realtà abbiamo già descritto tale funzione, che coincide "essenzialmente" con la funzione $GOTO$.
- Si noti che, tuttavia, che $GOTO(I, X)$ "costruisce" nuovi stati e dunque $J = GOTO(I, X)$ non viene aggiunto se risulta già definito,
- In tale caso vale comunqe $\delta(I, X) = J$.

Esempio

- L'automa $LR(0)$ per la grammatica

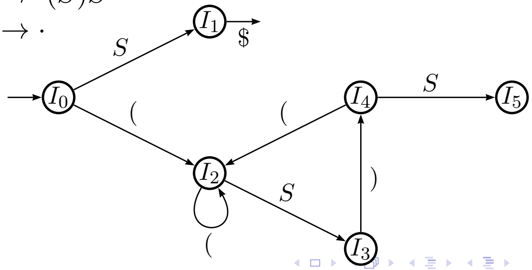
$$
\begin{array}{rcl}
S' & \rightarrow & S \\
S & \rightarrow & (S)S \mid \epsilon
\end{array}
$$

è:



Insiemi di item e automa

$$
\begin{array}{llcll}
I_0: & S' \rightarrow \cdot S & \quad & I_3: & S \rightarrow (S\cdot)S \\
 & S \rightarrow \cdot(S)S & & & \\
 & S \rightarrow \cdot & & I_4: & S \rightarrow (S)\cdot S \\
 & & & & S \rightarrow \cdot(S)S \\
I_1: & S' \rightarrow S\cdot & & & S \rightarrow \cdot \\
 & & & & \\
I_2: & S \rightarrow (\cdot S)S & & I_5: & S \rightarrow (S)S\cdot \\
 & S \rightarrow \cdot(S)S & & & \\
 & S \rightarrow \cdot & & &
\end{array}
$$

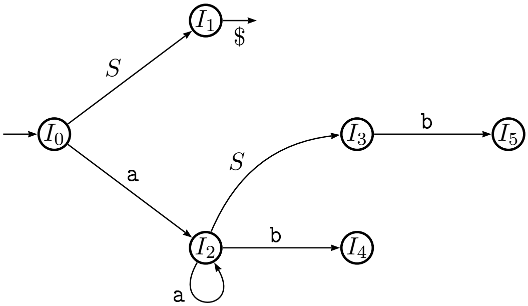


Esempio

- L'automa $LR(0)$ per la grammatica

$$
\begin{array}{rcl}
S' & \rightarrow & S \\
S & \rightarrow & \mathtt{a}S\mathtt{b} \mid \mathtt{ab}
\end{array}
$$

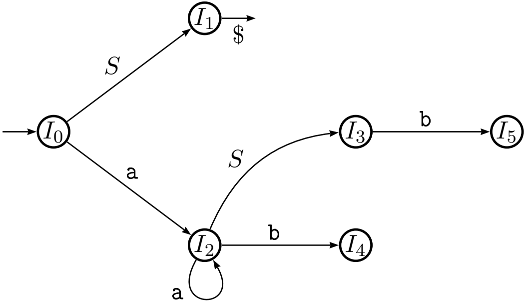
è:



Esempio

- Ricordiamo anche gli insiemi di item:

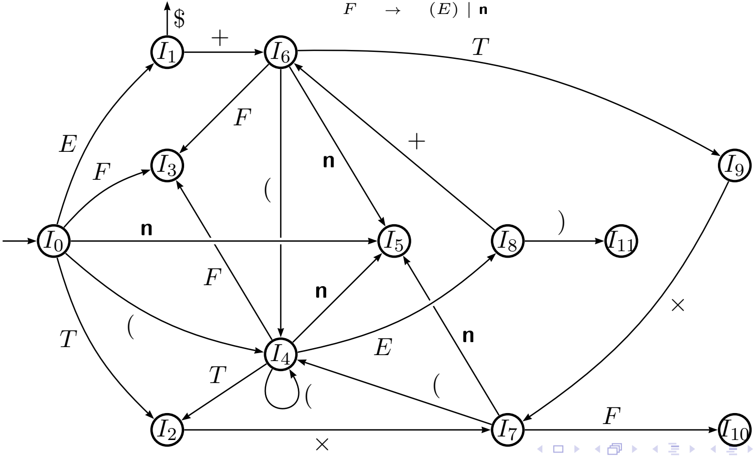
$$
\begin{array}{llllll}
I_0: & S' \to \cdot S & I_2: & S \to \mathtt{a} \cdot S\mathtt{b} & I_3: & S \to \mathtt{a}S \cdot \mathtt{b} \\
 & S \to \cdot \mathtt{a}S\mathtt{b} & & S \to \mathtt{a} \cdot \mathtt{b} & & \\
 & S \to \cdot \mathtt{ab} & & S \to \cdot \mathtt{a}S\mathtt{b} & I_4: & S \to \mathtt{ab}\cdot \\
 & & & S \to \cdot \mathtt{ab} & & \\
I_1: & S' \to S\cdot & & & I_5: & S \to \mathtt{a}S\mathtt{b}\cdot
\end{array}
$$



Esempio

- L'ultimo esempio è per la grammatica

$$\begin{array}{rcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & (E) \mid \mathbf{n} \end{array}$$



Gli insiemi di item

$$
\begin{aligned}
I_0 :\quad & E' \to \cdot E \\
& E \to \cdot E + T \\
& E \to \cdot T \\
& T \to \cdot T \times F \\
& T \to \cdot F \\
& F \to \cdot (E) \\
& F \to \cdot \mathbf{n}
\end{aligned}
$$

$$
\begin{aligned}
I_1 :\quad & E' \to E \cdot \\
& E \to E \cdot + T
\end{aligned}
$$

$$
\begin{aligned}
I_2 :\quad & E \to T \cdot \\
& T \to T \cdot \times F
\end{aligned}
$$

$$
I_3 :\quad T \to F \cdot
$$

$$
\begin{aligned}
I_4 :\quad & F \to (\cdot E) \\
& E \to \cdot E + T \\
& E \to \cdot T \\
& T \to \cdot T \times F \\
& T \to \cdot F \\
& F \to \cdot (E) \\
& F \to \cdot \mathbf{n}
\end{aligned}
$$

$$
I_5 :\quad F \to \mathbf{n} \cdot
$$

$$
\begin{aligned}
I_6 :\quad & E \to E + \cdot T \\
& T \to \cdot T \times F \\
& T \to \cdot F \\
& F \to \cdot (E) \\
& F \to \cdot \mathbf{n}
\end{aligned}
$$

$$
\begin{aligned}
I_7 :\quad & T \to T \times \cdot F \\
& F \to \cdot (E) \\
& F \to \cdot \mathbf{n}
\end{aligned}
$$

$$
\begin{aligned}
I_8 :\quad & E \to E \cdot + T \\
& F \to (E \cdot)
\end{aligned}
$$

$$
\begin{aligned}
I_9 :\quad & E \to E + T \cdot \\
& T \to T \cdot \times F
\end{aligned}
$$

$$
I_{10} :\quad T \to T \times F \cdot
$$

$$
I_{11} :\quad F \to (E) \cdot
$$

Item validi per un prefisso ammissibile

- Come può l'automa $LR(0)$ aiutare a risolvere il "dilemma" shift-reduce, ovvero aiutare nel riconoscimento delle handle?
- Consideriamo una generica grammatica G e consideriamo un prefisso ammissibile $\alpha\beta_1$
- Ricordiamo per comodità che questo vuol dire che esiste una derivazione canonica destra $S \stackrel{*}{\Rightarrow} \gamma$ tale che $\alpha\beta_1$ è un prefisso di γ che termina non oltre la handle presente in γ (o la handle più a destra, se in γ ce n'è più d'una e dunque la grammatica è ambigua)
- Un item $Z \rightarrow \beta_1 \cdot \beta_2$ si dice *valido* per $\alpha\beta_1$ se esiste una derivazione canonica destra $S \stackrel{*}{\Rightarrow} \alpha Z w \Rightarrow \alpha\beta_1\beta_2 w$, dove Z è il nonterminale più a destra e dunque w è una stringa di soli terminali
- In altri termini, un item è valido per un prefisso ammissibile, se i simboli che precedono il puntino sono anche gli ultimi del prefisso

Item validi e automa LR(0)

- Un risultato fondamentale nella teoria del parsing LR (che non dimostriamo) è il seguente
- Gli item validi per un prefisso ammissibile γ sono esattamente gli item che formano la collezione dello stato I che si raggiunge a partire dallo stato iniziale seguendo la sequenza di transizioni etichettate con γ
- Con riferimento alla grammatica G_{AB}, consideriamo, ad esempio, il prefisso ammissibile $\gamma = \mathbf{a}A\mathbf{b}$
- La sequenza di transizioni etichettate con i simboli di γ conduce dallo stato iniziale I_0 allo stato I_3
- Gli item validi per γ sono quindi precisamente i quattro item che formano la "collezione" I_3
- Che questi siano validi è verificato nella prossima slide

Item validi e automa LR(0)

- Per ogni item della forma $Z \Rightarrow \beta_1 \cdot \beta_2$, mostriamo che esiste una derivazione canonica destra $S \stackrel{*}{\Rightarrow} \alpha Z w \Rightarrow \alpha\beta_1\beta_2 w$ in cui $\alpha\beta_1 = \mathbf{a}A\mathbf{b}$
  1. Per l'item $B \rightarrow \mathbf{b}\cdot$ abbiamo la derivazione canonica destra
  $$A' \Rightarrow A \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{a}A\mathbf{b}$$
  e possiamo porre $\alpha = \mathbf{a}A$, $\beta_1 = \mathbf{b}$ e chiaramente $\beta_2 = \epsilon$
  2. Per l'item $B \rightarrow \mathbf{b} \cdot B$ abbiamo invece la derivazione
  $$A' \Rightarrow A \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{a}A\mathbf{b}B$$
  in cui ancora $\alpha = \mathbf{a}A$, $\beta_1 = \mathbf{b}$ ma $\beta_2 = B$
  3. Per l'item $B \rightarrow \cdot\mathbf{b}$ abbiamo invece
  $$A' \Rightarrow A \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{a}A\mathbf{b}B \Rightarrow \mathbf{a}A\mathbf{bb}$$
  con $\alpha = \mathbf{a}A\mathbf{b}$, $\beta_1 = \epsilon$ e $\beta_2 = \mathbf{b}$
  4. Infine, per l'item $B \rightarrow \cdot\mathbf{b}B$ abbiamo la derivazione
  $$A' \Rightarrow A \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{a}A\mathbf{b}B$$
  in cui $\alpha = \mathbf{a}A$ $\beta_1 = \epsilon$ w $\beta_2 = \mathbf{b}B$.

Item validi per un prefisso ammissibile

- Sapere che un item $Z \rightarrow \beta_1 \cdot \beta_2$ è valido per $\alpha\beta_1$ rappresenta un'informazione importante, anche se non decisiva, per stabilire se operare lo shift oppure eseguire una riduzione
- Intanto osserviamo che non siamo interessati a tutti i prefissi ammissibili
- Di fatto ci interessano solo i prefissi che possiano trovare sullo stack
- Se il prefisso $\alpha\beta_1$ si trova sullo stack:
  - se $\beta_2 \neq \epsilon$ allora la handle non è ancora tutta sullo stack e dunque l'operazione da fare è eseguire uno shift;
  - viceversa, se $\beta_2 = \epsilon$, allora la handle è già sullo stack e l'operazione da compiere è la riduzione con la produzione $Z \rightarrow \beta_1$
- Bisogna però considerare che per un dato prefisso potrebbe esserci più di un item valido
- Vediamo ancora l'esempio della collezione I_3 per la grammatica G_{AB}

Item validi e automa LR(0)

- Analizziamo dunque gli item in I_3 uno per uno, ricordando che il prefisso è $\gamma = \mathbf{a}A\mathbf{b}$
  1. Item $B \rightarrow \mathbf{b}\cdot$. La presenza di questo item imporrebbe una riduzione con la produzione $B \rightarrow \mathbf{b}$
  2. Item $B \rightarrow \mathbf{b} \cdot B$. In questo caso $\beta_1 = \mathbf{b}$ e $\beta_2 = B$. Questo item "chiama" uno shift
  3. Item $B \rightarrow \cdot\mathbf{b}$. Qui $\beta_1 = \epsilon$ e $\beta_2 = \mathbf{b}$. Anche questo item chiama uno shift.
  4. Item $B \rightarrow \cdot\mathbf{b}B$. Qui $\beta_1 = \epsilon$ e $\beta_2 = \mathbf{b}B$. E anche in questo caso viene chiamato uno shift

Costruzione della tabella di parsing SLR(1)

- La situazione illustrata nella precedente diapositiva sembra lasciare ancora aperto il problema della decisione shift/reduce
- Ci sono però alcune considerazioni da fare
- Innanzitutto, osserviamo che se una collezione include un item del tipo $Z \to \alpha \cdot A\gamma$, cioè un item in cui il puntino precede un nonterminale, allora a seguito dell'operazione di chiusura, la collezione conterrà anche un item del tipo $W \to \cdot x\delta$, cioè un item in cui il puntino precede un terminale (è immediato rendersene conto)
- In secondo luogo non abbiamo ancora tenuto conto dell'input!
- Si rifletta sul fatto che l'automa che abbiamo definito è indicato come "automa $LR(0)$" proprio perché nella sua costruzione l'input non viene preso in considerazione
- Nella costruzione del parser $SLR(1)$, a partire dall'automa $LR(0)$, prendiamo finalmente in considerazione un carettere di *lookahead*

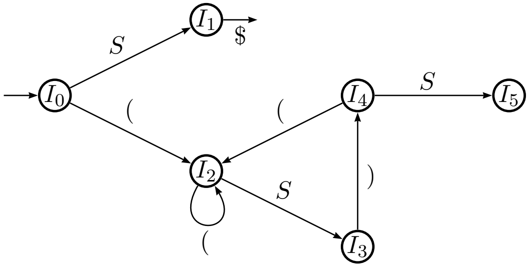
Tabelle di parsing $SLR(1)$

- Le tabelle di parsing $SLR(1)$, di cui ora diamo l'algoritmo di costruzione, incorporano le informazioni contenute nell'automa, oltre ad altre informazioni
- Fra le informazioni importanti, non considerate nell'automa, un ruolo fondamentale è costituito dalla conoscenza dell'insieme di simboli $FOLLOW(A)$, per ogni simbolo non terminale A della grammatica
- Le tabelle hanno tante righe quanti sono gli stati dell'automa e un numero di colonne pari al vocabolario della grammatica
- Le colonne sono suddivise in due parti, corrispondenti ai simboli terminali e nonterminali, chiamate rispettivamente parte *ACTION* e parte *GOTO*
- L'algoritmo esamina gli stati dell'automa e le transizioni uscenti da ciascuno stato

Tabelle di parsing $SLR(1)$

- Per ogni stato I_j, consideriamo le transizioni uscenti.
- Se esiste una transizione da I_j a I_k etichettata $X \in \mathcal{T}$ poniamo $ACTION\,[j, X] =$ shift k.
- Se esiste una transizione da I_j a I_k etichettata $X \in \mathcal{N}$ poniamo $GOTO\,[j, X] = k$.
- Se nell'insieme di item corrispondenti a I_j esiste un item $A \rightarrow \alpha\cdot$, allora poniamo $ACTION\,[j, X] =$ reduce $A \rightarrow \alpha$ per tutti i simboli X in $FOLLOW(A)$.
- Se I_j contiene l'item $\mathcal{S}' \rightarrow \mathcal{S}\cdot$ si pone $ACTION[j, \$] =$ accept.
- Se, in base a quanto sopra, in una posizione della tabella viene inserito più di un dato, allora la grammatica non è SLR(1)
- I conflitti, esclusivamente nella parte *ACTION*, possono essere di tipo shift-reduce

Esempio (grammatica per le parentesi)



Stato	*ACTION*			*GOTO*
	(	)	$	*S*
0	shift 2	reduce 2	reduce 2	1
1			accept	
2	shift 2	reduce 2	reduce 2	3
3		shift 4		
4	shift 2	reduce 2	reduce 2	5
5		reduce 1	reduce 1	

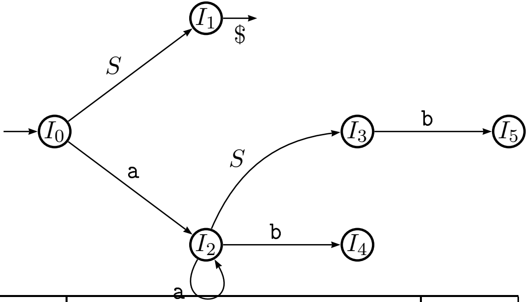
Esempio

- Riconsideriamo la grammatica

S	$\rightarrow$	aSb	Produzione 1
S	$\rightarrow$	ab	Produzione 2

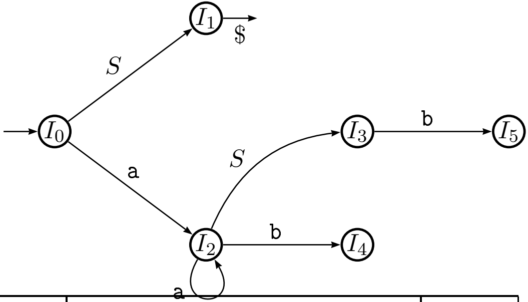
  in cui abbiamo numerato (arbitrariamente) le produzioni.
- Per tale grammatica l'algoritmo appena delineato produce la tabella di parsing evidenziata nella seguente diapositiva (in cui riportiamo, per comodità, anche l'automa $LR(0)$).
- È immediato anche verificare che $\mathrm{FOLLOW}(S) = \{\$, \mathtt{b}\}$

Esempio (continua)



Stato	*ACTION*			*GOTO*
	a	b	$	S
0	shift 2			1
1			accept	
2	shift 2	shift 4		3
3		shift 5		
4		reduce 2	reduce 2	
5		reduce 1	reduce 1	

Esempio (continua)



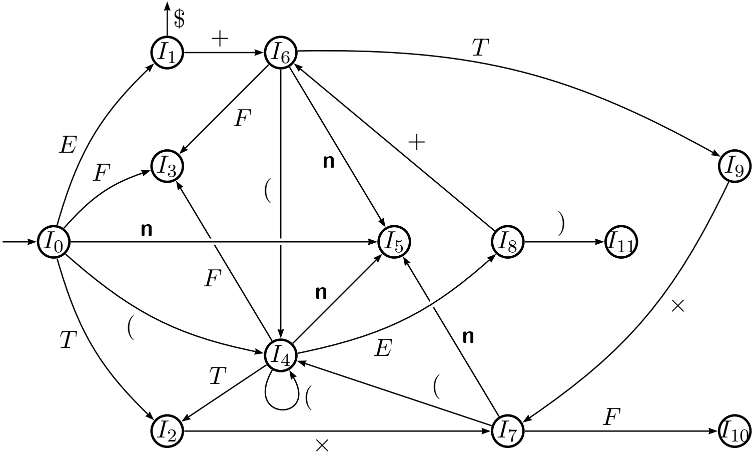
Stato	*ACTION*			*GOTO*
	a	b	$	S
0	shift 2			1
1			accept	
2	shift 2	shift 4		3
3		shift 5		
4		reduce 2	reduce 2	
5		reduce 1	reduce 1	

Esempio (continua)

- Consideriamo il comportamento del parser su input aabb

Stack	Input	Azione
$\$0$	aabb$	shift 2
$\$02$	abb$	shift 2
$\$022$	bb$	shift 4
$\$0224$	b$	reduce $S \rightarrow$ ab
$\$023$	b$	shift 5
$\$0235$	$	reduce $S \rightarrow$ aSb
$\$01$	$	accept

Ricordiamo l'automa per la grammatica delle espressioni



- $\text{FOLLOW}(E) = \{\$,), +\}$
- $\text{FOLLOW}(T) = \text{FOLLOW}(F) = \{\$,), +, *\}$

Esempio

- Diamo infine la tabella di parsing per la grammatica

E	$\to$	$E+T$	Prod. 1	T	$\to$	F	Prod. 4
E	$\to$	T	Prod. 2	F	$\to$	(E)	Prod. 5
T	$\to$	$T\times F$	Prod. 3	F	$\to$	**n**	Prod. 6

Stato	*ACTION*						*GOTO*		
	n	$+$	$\times$	(	)	\$	E	T	F
0	s 5			s 4			1	2	3
1		s 6				accept			
2		r 2	s 7		r 2	r 2			
3		r 4	r 4		r 4	r 4			
4	s 5			s 4			8	2	3
5		r 6	r 6		r 6	r 6			
6	s 5			s 4				9	3
7	s 5			s 4					10
8		s 6			s 11				
9		r 1	s 7		r 1	r 1			
10		r 3	r 3		r 3	r 3			
11		r 5	r 5		r 5	r 5			

Esempio (continua)

- Consideriamo il comportamento del parser su input $\mathbf{n} \times (\mathbf{n} + \mathbf{n})$

Stack	Input	Azione
$\$0$	$\mathbf{n}\times(\mathbf{n}+\mathbf{n})\$$	shift 5
$\$0\,5$	$\times(\mathbf{n}+\mathbf{n})\$$	reduce $F\rightarrow\mathbf{n}$
$\$0\,3$	$\times(\mathbf{n}+\mathbf{n})\$$	reduce $T\rightarrow F$
$\$0\,2$	$\times(\mathbf{n}+\mathbf{n})\$$	shift 7
$\$0\,2\,7$	$(\mathbf{n}+\mathbf{n})\$$	shift 4
$\$0\,2\,7\,4$	$\mathbf{n}+\mathbf{n})\$$	shift 5
$\$0\,2\,7\,4\,5$	$+\mathbf{n})\$$	reduce $F\rightarrow\mathbf{n}$
$\$0\,2\,7\,4\,3$	$+\mathbf{n})\$$	reduce $T\rightarrow F$
$\$0\,2\,7\,4\,2$	$+\mathbf{n})\$$	reduce $E\rightarrow T$
$\$0\,2\,7\,4\,8$	$+\mathbf{n})\$$	shift 6
$\$0\,2\,7\,4\,8\,6$	$\mathbf{n})\$$	shift 5
$\$0\,2\,7\,4\,8\,6\,5$	$)\$$	reduce $F\rightarrow\mathbf{n}$
$\$0\,2\,7\,4\,8\,6\,3$	$)\$$	reduce $T\rightarrow F$
$\$0\,2\,7\,4\,8\,6\,9$	$)\$$	reduce $E\rightarrow E+T$
$\$0\,2\,7\,4\,8$	$)\$$	shift 11
$\$0\,2\,7\,4\,8\,11$	$\$$	reduce $F\rightarrow(E)$
$\$0\,2\,7\,10$	$\$$	reduce $T\rightarrow T\times F$
$\$0\,2$	$\$$	reduce $E\rightarrow T$
$\$0\,1$	$\$$	accept