Linguaggi e compilatori
Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2023/2024

Linguaggi e compilatori

1. Analisi sintattica (PARTE TERZA): Parsing shift-reduce
   - Generalità sul parsing bottom-up
   - Parsing SLR(1)

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

• Non può essere usata con un parser top-down

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

chiarisce questo punto.

Esempio

- Shift -> prossimo carattere in input e messo nello stack (non cambia l'attuale forma di frase)
- Reduce -> L'operazione reduce sostituisce sullo stack gli ultimi k simboli inseriti

- 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.

di

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 $\mathbf{n} \times \mathbf{n}$ è indicata di seguito

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	$\$T$	$\times\mathbf{n}\$$	shift	$T \times \mathbf{n}\$$
2	$\$T\times$	$\mathbf{n}\$$	shift	$T \times \mathbf{n}\$$
3	$\$T \times \underline{\mathbf{n}}$	$\$$	reduce	$T \times \mathbf{n}\$$
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 $\mathbf{n} \times \mathbf{n}$.

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 $A \rightarrow \beta_1\beta_2$ 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 $A \rightarrow \beta_1\beta_2$ allora il parser esegue uno shift o dichiara errore (come vedremo).

Osservazioni (continua)

- L'osservazione più importante è che la prossima handle da utilizzare "prima o poi" si trova esattamente sulla cima dello stack.
- Questa proprietà vale perché consideriamo derivazioni canoniche destre; non varrebbe nel caso volessimo riprodurre una derivazione canonica sinistra.

Esempio

- Azioni eseguite (su input $\mathbf{n}+\mathbf{n}$) da un parser shift-reduce che “ricostruisce” una derivazione canonica sinistra, riconoscendo le handle.
- Come si può vedere, non è possibile garantire che le handle siano sempre sulla cima dello stack.

Indice i	Stack	Input	Azione	Stringa α_i
0	$\$ \leftarrow$	$\mathbf{n}+\underline{\mathbf{n}}\$$	shift	$\mathbf{n}+\mathbf{n}\$$
0	$\$\mathbf{n} \leftarrow$	$+\underline{\mathbf{n}}\$$	shift	$\mathbf{n}+\mathbf{n}\$$
0	$\$\mathbf{n}+ \leftarrow$	$\underline{\mathbf{n}}\$$	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	$E\$$

Il cuore computazionale del problema

- La difficoltà di progettazione del 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.

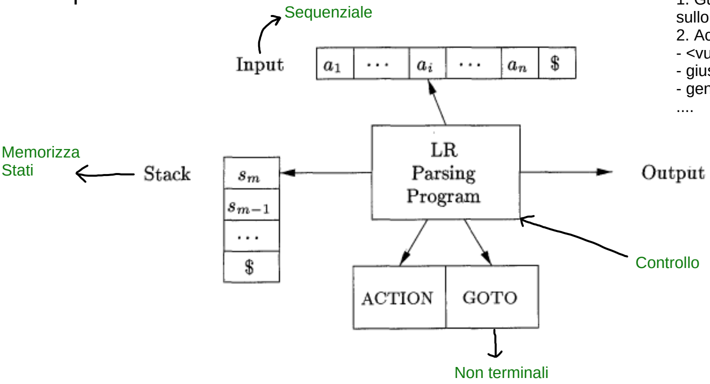
Linguaggi e compilatori

Parser LR

- Si tratta di una classe di parser di tipo shift-reduce (con analisi dell'input da sinistra a destra, "Left to Right"), caratterizzati da una struttura di programma comune ma con capacità di analisi diverse.
- 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 (essenzialmente un automa a stati finiti) che ha accesso ad uno stack e ad una tabella di parsing, oltre che a opportuni supporti di input e output.



1. Guarda lo stato in cima sullo stack e il simbolo in input
2. Accede alla tabella "action"
- <vuoto> -> ERROR
- giusto -> ACCEPT
- general -> SHIFT + Stato
....

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 $\boxed{A \to \alpha}$ è tale produzione, il parser rimuove $\boxed{k_i = |\alpha|}$ 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:

$$S \rightarrow (S)S \qquad \text{Produzione 1}$$
$$S \rightarrow \epsilon \qquad \text{Produzione 2}$$

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)

- In pratica basandosi sulla tabella di parsing sopra riesce a capire sa la sintassi è corretta
- Stato attuale

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

Parte dallo stato 0, guarda nella tabella sopra che cosa deve fare su input "("

Fa reduce S -> eps perchè nella tabella c'è scritto da stato 2 con input ")" -> reduce 2 (produzione)

In questi 2 step fa la pop() degli stati 5,4,3,2 poichè ad ogni step reduce fa la pop di k stati -> k = |a| dove a = A -> a

- 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.

Parsing SLR(1)

- Come detto, l'unico 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.

Automa $LR(0)$

- Il passo fondamentale consiste nella definizione di un automa (che di fatto sarà poi "trasferito" nella tabella di parsing), detto *automa LR(0)*.
- Data la grammatica G, la si "aumenta" con una produzione aggiuntiva, $\mathcal{S}' \to \mathcal{S}$ (il cui significato sarà chiaro più avanti).
- A partire dalle produzioni della grammatica aumentata, si definiscono poi speciali "oggetti", che chiameremo *item*.
- Un item è una produzione con inserito un marcatore nella parte destra, tipicamente un punto.
- Ad esempio, gli item associati alla produzione $S \to (S)S$ 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 $S \to \cdot$.

Automa $LR(0)$ (continua)

- Qual è il significato di un item associato ad una data produzione?
- Intuitivamente, esso 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 abbiamo riconosciuto una stringa descritta da (S) e che ci "attendiamo" di riconoscere una stringa descrivibile da S.
- Un item con il puntino in fondo indica quindi che il processo di riconoscimento della parte destra è completato e dunque che si può operare la riduzione (vedremo sotto quale altra condizione).

Automa $LR(0)$ (continua)

- Gli item vengono poi raggruppati in collezioni, ognuna delle quali definisce uno "stato" nel processo di riconoscimento.
- Ad esempio, una collezione per il parser della grammatica per le parentesi appena vista sarà costituito dagli item:

$$\begin{array}{lcl} S & \to & (S) \cdot S \\ S & \to & \cdot (S)S \\ S & \to & \cdot \end{array}$$

- Essa descrive la situazione in cui abbiamo riconosciuto (S) e ci attendiamo di riconoscere S (primo item), cioè ci attendiamo di riconoscere ancora un'istanza completa di $(S)S$ (secondo item) o, in alternativa, la stringa vuota (terzo item).
- Le collezioni di item costituiranno proprio gli stati dell'automa al cuore del parser.

Automa $LR(0)$ (continua)

- Si noti che l'intersezione di due collezioni può non essere vuota.
- Ad esempio, l'item $S \rightarrow (\cdot S)S$ forma gruppo ancora con $S \rightarrow \cdot(S)S$ e $S \rightarrow \cdot$ (per la stessa ragione di prima).
- 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 $S \rightarrow (S\cdot)S$.

Esempio Example1

- Per la grammatica "aumentata"

$$\boxed{\begin{array}{lcl} S' & \to & S \\ S & \to & (S)S \mid \epsilon \end{array}}$$

sono definiti i seguenti insiemi di item:

$$\begin{array}{llll} I_0: & S' \to \cdot S \quad \text{I1} & I_3: & S \to (S\cdot)S \quad \text{I4} \\ & S \to \cdot(S)S \quad \text{I2} & & \\ & S \to \cdot \quad \text{I3} & I_4: & S \to (S)\cdot S \quad \text{I5} \\ & & & S \to \cdot(S)S \quad \text{I2} \\ I_1: & S' \to S\cdot \quad \text{Finish} & & S \to \cdot \quad \text{Finish} \\ & & & \\ I_2: & S \to (\cdot S)S \quad \text{I3} & I_5: & S \to (S)S\cdot \quad \text{Finish} \\ & S \to \cdot(S)S \quad \text{I2} & & \\ & S \to \cdot \quad \text{Finish} & & \end{array}$$

- Il puntino indica che abbiamo riconosciuto un terminale o un non terminale

Come costruire gli insiemi $LR(0)$

- Diamo ora una descrizione dettagliata del procedimento di costruzione degli insiemi di item.
- L'insieme iniziale (che indicheremo sempre con I_0) contiene l'item $\mathcal{S}' \rightarrow \cdot\mathcal{S}$ e tutti gli item ottenuti dalle produzioni di $\mathcal{S}$ inserendo il punto all'inizio.
- Nell'esempio appena considerato, si aggiungono a $S' \rightarrow \cdot S$ due soli item (perché ci sono due produzioni relative ad S).
- Si procede poi ricorsivamente, lavorando ad ogni passo su un insieme I_j già formato.
- 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 prodedimento 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 $B \to \cdot\gamma$ sotto le seguenti condizioni:
  - in I esista inizialmente un item del tipo $A \to \alpha \cdot B\beta$, oppure,
  - ad I sia già stato aggiunto un item del tipo $A \to \cdot B\beta$.
- Il procedimento termina quando non si possono più aggiungere item sulla base delle precedenti regole.

Funzione $CLOSURE(I)$

SetOfItems $\textsc{closure}(I)$ {
    $J = I$;
    repeat
        for (each item $A \rightarrow \alpha{\cdot}B\beta$ in J)
            for (each production $B \rightarrow \gamma$ of G)
                if ($B \rightarrow {\cdot}\gamma$ is not in J)
                    add $B \rightarrow {\cdot}\gamma$ 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:
    - iniazialmente 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 $\{\mathtt{a}^n\mathtt{b}^n | n \geq 1\}$:

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

- 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)

Example2

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

$$I_0: \quad S' \to \cdot S \quad \text{I1}$$
$$S \to \cdot \mathtt{a}S\mathtt{b} \quad \text{I2}$$
$$S \to \cdot \mathtt{ab} \quad \text{I2}$$

$$I_1: \quad S' \to S\cdot \quad \text{Finish}$$

$$I_2: \quad S \to \mathtt{a}\cdot S\mathtt{b} \quad \text{I3}$$
$$S \to \mathtt{a}\cdot\mathtt{b} \quad \text{I4}$$
$$S \to \cdot \mathtt{a}S\mathtt{b} \quad \text{I0}$$
$$S \to \cdot \mathtt{ab} \quad \text{I0}$$

$$I_3: \quad S \to \mathtt{a}S\cdot\mathtt{b} \quad \text{I5}$$

$$I_4: \quad S \to \mathtt{ab}\cdot \quad \text{Finish}$$

$$I_5: \quad S \to \mathtt{a}S\mathtt{b}\cdot \quad \text{Finish}$$

Esempio

Example3

- 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)

$$
\begin{array}{ll}
I_0: & E' \to \cdot E \quad 1 \\
& E \to \cdot E + T \\
& E \to \cdot T \quad 2 \\
& T \to \cdot T \times F \\
& T \to \cdot F \\
& F \to \cdot (E) \\
& F \to \cdot \mathbf{n}
\end{array}
$$

E

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

T

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

F

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

(

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

n

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

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

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

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

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

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

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

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 $\delta(I, X) = J$.

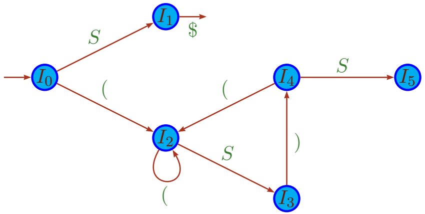
Esempio

- Caratteristica di questo automa che vedrà in input dei caratteri che in input non possono esserci.
  Ex: Non posso leggere in input il non terminale S.

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

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

è:

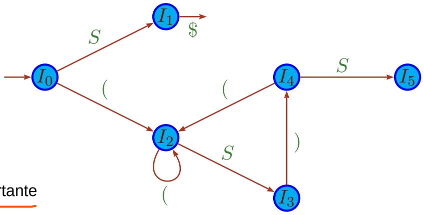


Insiemi di item e automa

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

- 1. Se dopo puntino c'è terminale metti nella colonna ACTION di dove ti sposti
  2. Se si trova davanti a un non terminale metti nella colonna GO TO di dove ti sposti
  3. Se c'è un puntino in fondo fai una reduce

Importante

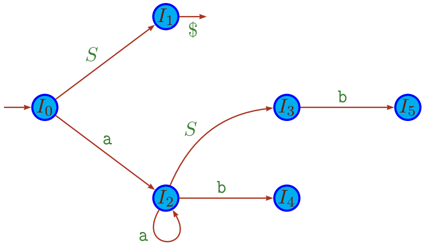


Esempio

- 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:

$$I_0: \quad S' \to \cdot S \qquad S \to \cdot \mathtt{a}S\mathtt{b} \qquad S \to \cdot \mathtt{ab}$$

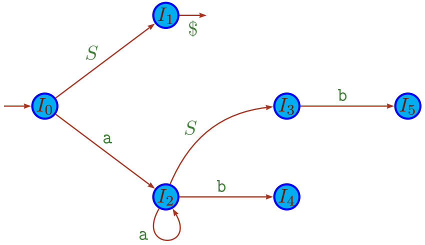
$$I_1: \quad S' \to S\cdot$$

$$I_2: \quad S \to \mathtt{a} \cdot S\mathtt{b} \qquad S \to \mathtt{a} \cdot \mathtt{b} \qquad S \to \cdot \mathtt{a}S\mathtt{b} \qquad S \to \cdot \mathtt{ab}$$

$$I_3: \quad S \to \mathtt{a}S \cdot \mathtt{b}$$

$$I_4: \quad S \to \mathtt{ab}\cdot$$

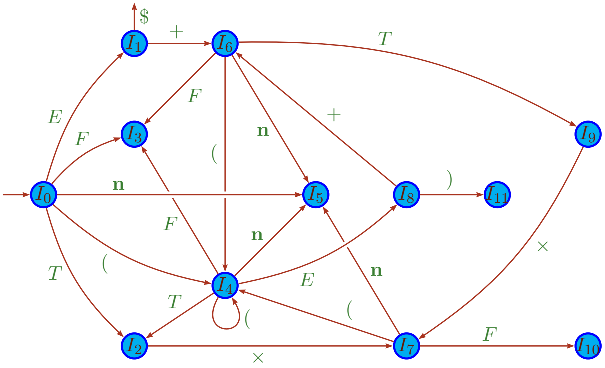
$$I_5: \quad S \to \mathtt{a}S\mathtt{b}\cdot$$



Esempio

- L'ultimo esempio è per la grammatica

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



Gli insiemi di item

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

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

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

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

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

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

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

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

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

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

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

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

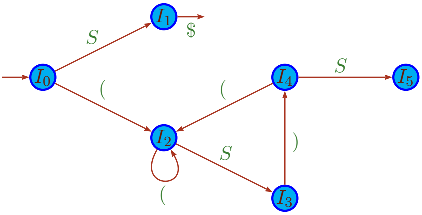
Tabelle di parsing $SLR(1)$

- Completiamo ora la descrizione del parser con l'algoritmo di definizione della tabella di parsing.
- Le tabelle incorporano le informazioni contenute nell'automa, che da solo non è sufficiente per eseguire l'analisi (si ricordi che un automa a stati finiti non è in grado di riconoscere linguaggi liberi (che non siano anche regolari).
- L'algoritmo esamina gli stati dell'automa e le transizioni uscenti da ciascuno stato.
- Esso necessita anche di conoscere, per ogni simbolo non terminale A, l'insieme di simboli $FOLLOW(A)$.

Tabelle di parsing $SLR(1)$ (continua)

- 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, ad un qualunque passo dell'algoritmo, si manifesta un cosiddetto *conflitto shft-reduce* (cioè si tenta di inserire in una entry della parte ACTION sia un'azione di shift che una di riduzione) allora la grammatica non è SLR(1).

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

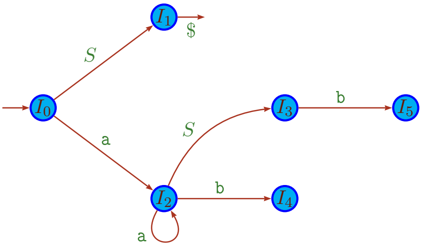
- Riconsideriamo la grammatica

  $S \rightarrow \mathtt{a}S\mathtt{b}$ Produzione 1

  $S \rightarrow \mathtt{ab}$ Produzione 2

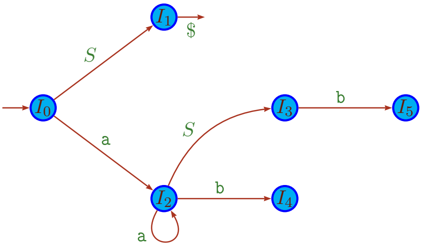
  in cui abbiamo numerato (arbitrariamente) le produzioni.
- Per tale grammatica l'algoritmo appena delineato produce la tabella di parsing evidenziata nella seguente diapositiva (in cui riportiamo, per comodità, anche l'automa $LR(0)$).
- È immediato anche verificare che $\mathrm{FOLLOW}(S) = \{\$, \mathtt{b}\}$

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)



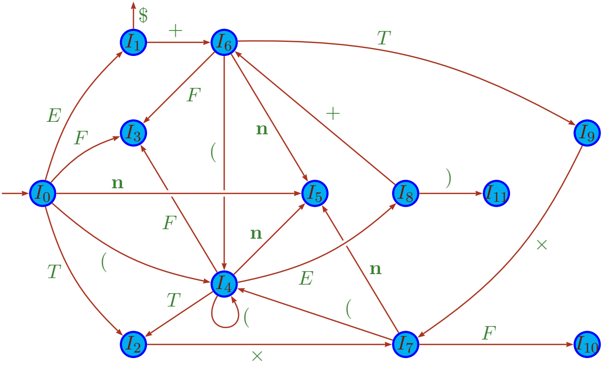
Stato	*ACTION*			*GOTO*
	a	b	$	S
0	shift 2			1
1			accept	
2	shift 2	shift 4		3
3		shift 5		
4		reduce 2	reduce 2	
5		reduce 1	reduce 1	

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 $S \to$ ab
$\$023$	b$	shift 5
$\$0235$	$	reduce $S \to$ aSb
$\$01$	$	accept

Ancora l'automa per comodità dell'ultimo esempio



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

Esempio

- Diamo infine la tabella di parsing per la grammatica

E	$\rightarrow$	$E+T$	Prod. 1	T	$\rightarrow$	F	Prod. 4
E	$\rightarrow$	T	Prod. 2	F	$\rightarrow$	(E)	Prod. 5
T	$\rightarrow$	$T \times F$	Prod. 3	F	$\rightarrow$	$\mathbf{n}$	Prod. 6

Stato	*ACTION*						*GOTO*		
	$\mathbf{n}$	$+$	$\times$	(	)	\$	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