

9. Analisi Sintattica

Nicola Fanizzi (*fanizzi@di.uniba.it*)

Dipartimento di Informatica
Università degli Studi di Bari

Corso di Linguaggi di Programmazione

3 maggio 2005

1 Introduzione

- Definizione
- Funzionalità del Parser

2 Analisi Sintattica Discendente

- Grammatiche $LL(k)$
- Analisi Discendente Guidata da Tabella
- Grammatiche $LL(1)$
- Gestione Errori
- Analisi Top-down in Discesa Ricorsiva

3 Analisi Sintattica Ascendente

- Grammatiche $LR(k)$
- Riduzione
- Algoritmo LR

Introduzione

L'analisi sintattica è l'attività del compilatore tesa a riconoscere la struttura del programma sorgente costruendo l'*albero sintattico* corrispondente mediante i simboli forniti dallo scanner

La componente del compilatore che si occupa di questa attività è l'**analizzatore sintattico** (o *parser*)

Funzionalità del Parser

Analisi Costruzione dell'albero di derivazione

- **discendente** (o *top-down*): dalla radice alle foglie
- **ascendente** (o *bottom-up*): dalle foglie alla radice

Segnalazione errori sintattici

- segnalazione precisa della posizione degli errori
- recupero in modo da portare a termine l'analisi se possibile
- efficienza del processo

Analisi Sintattica Discendente

Riconoscimento della struttura della frase in ingresso (*sorgente*) costruendo l'albero sintattico dalla radice alle foglie seguendo la

derivazione canonica sinistra

- ad ogni passo si espande il non terminale più a sinistra nella forma di frase generata dal parser fino a quel momento

Se la forma di frase è aXw con

- $a \in VT^*$
- $X \in VN$ *simbolo non terminale corrente* da espandere
- $w \in (VT \cup VN)^*$

difficoltà principale: scelta della parte destra da espandere nelle produzioni per il non terminale corrente

Necessità di fare *backtracking* (fonte di inefficienza)

Esempio.

data una grammatica G
con le seguenti produzioni:

- (1) $S \longrightarrow uXZ$
- (2) $X \longrightarrow yW$
- (3) $X \longrightarrow yV$
- (4) $W \longrightarrow w$
- (5) $V \longrightarrow v$
- (6) $Z \longrightarrow z$

consideriamo la frase
in ingresso $uyvz$

- 1° passo $S \xrightarrow{(1)} \underline{u}XZ$
 2° passo $uXZ \xrightarrow{(2)} \underline{uy}WZ$
 3° passo $uyWZ \xrightarrow{(4)} \underline{uyw}Z$
 la stringa uyw non è un prefisso
 stringa d'ingresso
 quindi bisogna fare *backtrack*
 2° passo $uXZ \xrightarrow{(3)} \underline{uy}VZ$
 3° passo $uyVZ \xrightarrow{(5)} \underline{uyv}Z$
 4° passo $uyvZ \xrightarrow{(6)} uyvz$

la stringa uyw non è un prefisso della stringa d'ingresso

quindi bisogna fare *backtracking*

Analisi Discendente Deterministica

Problemi dell'Analisi Discendente

- inefficienza
- ritrattazione azioni semantiche già intraprese

Rimedi:

- 1 trasformare la grammatica ai fini dell'analisi discendente
- 2 utilizzare l'informazione fornita dai simboli successivi
(*lookahead*)

Forme Normali per grammatiche libere

Grammatiche LL(k)

Classe di grammatiche (libere) in cui

- L: la stringa in ingresso è esaminata da sinistra a destra (LeftToRight)
- L viene costruita la derivazione canonica sinistra
- k numero di simboli di lookahead usati per poter scegliere la parte destra

Trasformazione

Problemi:

- 1 ricorsione sinistra nelle produzioni della grammatica
- 2 presenza di prefissi comuni in parti destre per lo stesso NT

Rimedi:

- 1 eliminazione della ricorsione sinistra (vedi algoritmo parte n. 7)
- 2 fattorizzazione sinistra:

$$A \longrightarrow yv \mid yw \quad \text{con } y, v \in V^+, w \in V^*$$

(uno dei due suffissi può essere vuoto e possono non avere prefissi comuni)

$$A \longrightarrow yA'$$

$$A' \longrightarrow v \mid w$$

Esempio.

Data la grammatica:

$$E \longrightarrow T \mid -T \mid E + T \mid E - T$$

$$T \longrightarrow F \mid T * F$$

$$F \longrightarrow i \mid (E)$$

eliminando le ricorsioni sinistre:

$$E \longrightarrow TE' \mid -TE'$$

$$E' \longrightarrow +TE' \mid -TE' \mid \epsilon$$

$$T \longrightarrow FT'$$

$$T' \longrightarrow *FT' \mid \epsilon$$

$$F \longrightarrow i \mid (E)$$

Analisi Discendente Guidata da Tabella

Si consideri una grammatica $LL(1)$ descritta da una tabella in cui

righe non terminali

colonne terminali

caselle parti destre delle produzioni / info errori

Tabella guida

	i	$+$	$-$	$*$	$($	$)$	$\$$
E	TE'		$-TE'$		TE'		
E'		$+TE'$	$-TE'$			ϵ	ϵ
T	FT'				FT'		
T'		ϵ	ϵ	$*FT'$		ϵ	ϵ
F	i				(E)		

Osservazioni:

- Derivazione canonica sinistra \longrightarrow occorre mantenere solo la parte destra dell'albero sintattico (ancora da espandere) in uno *stack*
- mediante un indice, il parser scorre la stringa in ingresso puntando al *prossimo simbolo* da riconoscere

Algoritmo

Inizialmente sullo stack c'è il simbolo distintivo della grammatica e l'indice punta al primo terminale sulla stringa in ingresso
il parser scorre la tabella in base al top dello stack e all'indice

- 1 se al top c'è un terminale, esso deve coincidere con quello puntato dall'indice della stringa d'ingresso, in tal caso il top viene cancellato e l'indice avanza al prossimo terminale;
- 2 se al top c'è un non terminale si sceglie la parte destra con cui espanderlo in base al simbolo terminale corrente secondo l'indice; si sostituisce il non terminale con i simboli trovati in tabella nella casella corrispondente e si fa avanzare l'indice;

Altrimenti si ricade in condizione di errore (codificabile nella tabella stessa)

Esempio (continua). $w = - i + i * i \$$

<i>stack</i>	<i>frase</i>	<i>casella</i>
\$E	<u>-</u> i + i * i \$	tab[E] [-]
\$E'T-		
\$E'T	- <u>i</u> + i * i \$	tab[T] [i]
\$E'T'F		tab[F] [i]
\$E'T'i		
\$E'T'	- i <u>+</u> i * i \$	tab[T'] [+]
\$E'		tab[E'] [+]
\$E'T+		
\$E'T	- i + <u>i</u> * i \$	tab[T] [i]
\$E'T'F		tab[F] [i]
\$E'T'i		
\$E'T'	- i + i <u>*</u> i \$	tab[T'] [*]
\$E'T'F*		
\$E'T'F	- i + i * <u>i</u> \$	tab[F] [i]
\$E'T'i		
\$E'T'	- i + i * i <u>\$</u>	tab[T'] [\$]
\$E'		tab[E'] [\$]
\$		

Implementazione

```
void parse(w)
lista_token w;
/* stringa da analizzare */
{
pila_simboli stack; /* pila */
int ip; /* indice prossimo simbolo */
parte_destra tab[num_nt][num_t];
/* tabella guida*/
simbolo x;

init(stack); push(stack,$); push(stack,S);
ip=1; w=strcat(w,$);

do
  if (terminale(top(stack)))
    if (top(stack) == w[ip])
      {pop(stack); ip++;}
    else error(1);
  else /* non terminale */
    if (tab[top(stack)][w[ip]] == 0)
      error(2);
    else if (tab[top(stack)][w[ip]] == ε)
      pop(stack);
    else { /* parte destra */
      pop(stack);
      for-each (x in tab[top(stack)][w[ip]])
        push(stack,x);
      }
  while (!empty(stack));
}
```

dove:

- le implementazioni dei tipi sono omesse;
- `pop()`, `push()`, `top()`, `empty()`, `init()`
funzioni primitive per pile;
- `terminale(s)` funzione booleana vera sse `s` è terminale;
- `foreach` struttura di controllo da implementare opportunamente

Grammatiche LL(1): insiemi First

Avendo una grammatica priva di ricorsioni sinistre e fattorizzata:
FIRST(x) insieme dei simboli terminali che possono essere prefissi
 di stringhe derivabili da $x \in V$

- $FIRST(t) = \{t\} \quad \forall t \in VT$
- $\forall X \in VN$ tale che $X \longrightarrow x_1|x_2|\dots|x_n \in P$
 $FIRST(X) = \{t \in VT \mid X \xRightarrow{+} tu, u \in V^*\} = \bigcup_{i=1}^n FIRST(x_i)$
- $\forall y = y_1y_2 \dots y_n, \quad y_i \in VN \cup VT$
 - se $y_1 \not\xRightarrow{+} \epsilon$ allora $FIRST(y) = FIRST(y_1)$
 - se $y_i \xRightarrow{+} \epsilon, i = 1, \dots, k$ e $y_{k+1} \not\xRightarrow{+} \epsilon$
allora $FIRST(y) = \bigcup_{i=1}^{k+1} FIRST(y_i)$
 - se $y_i \xRightarrow{+} \epsilon, \forall i = 1, \dots, n$
allora $FIRST(y) = \bigcup_{i=1}^n FIRST(y_i)$

Grammatiche LL(1): insiemi Follow

FOLLOW insieme dei terminali che in una derivazione possono seguire immediatamente $X \in VN$

Formalmente:

$$FOLLOW(X) = \{t \in VT \mid S \xRightarrow{+} uXtv \wedge u, v \in V^*\}$$

Se $Y \longrightarrow uXv \in P$ con $X \in VN$, $u, v \in V^*$

i due insiemi sono legati dalla relazione

$FOLLOW(X) \supseteq FIRST(v)$ se $v \not\xRightarrow{+} \epsilon$ ovvero

$FOLLOW(X) \supseteq FIRST(v) \cup FOLLOW(Y)$ altrimenti

Tali insiemi possono essere calcolati alitmicamente (vedi testo)

Condizioni LL(1)

Una grammatica si dice *LL(1)* sse, per ogni produzione $X \longrightarrow x_1 \mid x_2 \mid \dots \mid x_n$ sono soddisfatte le seguenti condizioni:

- 1 $FIRST(x_i) \cap FIRST(x_j) = \emptyset \quad \forall i \neq j$
 $FIRST(X) = \bigcup_{i=1}^n FIRST(x_i)$
- 2 esiste al più un solo x_j tale che $x_j \xRightarrow{+} \epsilon$ e, nel caso $FIRST(X) \cap FOLLOW(X) = \emptyset$

Parsing di grammatiche LL(1)

Si può dimostrare che le grammatiche $LL(1)$ sono non ambigue
Quindi il parser top-down di una grammatica $LL(1)$ è in grado di scegliere univocamente la parte destra in base al prossimo simbolo a della stringa in ingresso:

- 1 se $a \in FIRST(x_i)$
allora l'analizzatore espande X con la parte destra x_i
- 2 se $a \notin FIRST(x_i) \quad \forall i \in \{1, \dots, n\}$
ma $\exists j \in \{1, \dots, n\} \quad x_j \xRightarrow{*} \epsilon$
(e nessun'altra $x_k \xRightarrow{*} \epsilon, \quad k \neq j$)
e $a \in FOLLOW(X)$ allora espandi X con ϵ
- 3 altrimenti si segnala la situazione di errore

Costruzione Tabella per il Parser

- Per ogni regola della grammatica $LL(1)$

$$X \longrightarrow x_1 \mid x_2 \mid \dots \mid x_n$$

Si pone

$$tab[X][t] = x_i \quad \forall t \in FIRST(x_i)$$

- Se mediante la grammatica

$$x_j \xRightarrow{*} \epsilon$$

allora

$$tab[X][b] = \epsilon \quad \forall b \in FOLLOW(X)$$

Gestione Errori (analisi top-down guidata da tabella)

Tipi di errore

- ❶ mancata corrispondenza tra terminale corrente e quello al top dello stack
- ❷ accesso ad un elemento della tabella che risulta vuoto

Trattamento

- ❶ nel primo caso si hanno due alternative:
 - ❶ scartare un certo numero dei prossimi simbolo in ingresso finchè si trovino simboli per far riprendere l'analisi
 - ❷ inserire (virtualmente) il simbolo mancante in modo da riprendere l'analisi (senza causare altri errori)
- ❷ nel secondo caso non ci si può basare solo sullo stato corrente: coppia (top dello stack, simbolo terminale corrente) ma occorre tener conto dell'analisi già effettuata (se ne trova traccia sullo stack) usando *criteri euristici*

Analisi Top-down in Discesa Ricorsiva

Tecnica rapida di scrittura di *procedure ricorsive* di riconoscimento in base alle produzioni della grammatica *LL(1)*

Lo *stack* viene realizzato implicitamente dal meccanismo di gestione delle chiamate delle procedure del parser associate ad ogni non terminale

agenda

- passaggio da BNF alle procedure senza gestione errori
- passaggio da EBNF alle procedure senza gestione errori
- passaggio da EBNF alle procedure con gestione errori

In questi casi l'analizzatore lessicale ha una più stretta interazione con l'analizzatore sintattico

da BNF al Parser senza gestione errori

Implementazione.

Si costruisce, per ogni non terminale, una procedure corrispondente

Per ogni non terminale con produzioni: $X \longrightarrow x_1|x_2|\cdots|x_n$:

Se $X \not\Rightarrow^+ \epsilon$ allora la procedura X_p da scrivere sarà:

```
void X_p() {  
    if (token.code in FIRST(x1))  
        { /* codice relativo a x1 */ }  
    else if (token.code in FIRST(x2))  
        { /* codice relativo a x2 */ }  
    ...  
    else if (token.code in FIRST(xn))  
        { /* codice relativo a xn */ }  
    else error();  
}
```


Se $X \xRightarrow{+} \epsilon$ allora la procedura X_p da scrivere sarà:

```
void X_p() {  
    if (token.code in FIRST(x1))  
        { /* codice relativo a x1 */ }  
    else if (token.code in FIRST(x2))  
        { /* codice relativo a x2 */ }  
    ...  
    else if (token.code in FIRST(xn))  
        { /* codice relativo a xn */ }  
    else if (!token.code in FOLLOW(X))  
        error();  
}
```

Osservazioni:

- token: variabile (globale) utilizzata dallo scanner per passare il prossimo simbolo
- per quanto riguarda il codice relativo ad ogni parte destra x_i
 - si inseriscono tante istruzioni quanti sono i simboli di x_i (terminali e non) nell'ordine in cui compaiono
 - per ogni NT è prevista la chiamata della relativa procedura
 - per ogni terminale è prevista l'istruzione:

```
if (token.code == t_c) scan(token); else error(n);
```

per il primo (terminale) si omette il test perchè già effettuato in lookahead: si chiama `scan(token)` leggere il prossimo token
- nel main del parser: `init_scan(token); S_p();`
inizializza lo scanner e fa partire il riconoscimento del simbolo iniziale della grammatica

Richiamo sulla (E)BNF

- Nelle produzioni (libere) si separa la parte sinistra da quella destra usando $::=$ invece di \longrightarrow
- uso dei **metasimboli** $(,), [,], \{ , \}$:
 - gruppo $X ::= (x \mid y)z$ equivale a $X \longrightarrow xz \mid yz$
 - opzionalità $X ::= [x]yz$ equivale a $X \longrightarrow xyz \mid yz$
 - chiusura $X ::= \{x\}y$ equivale a $X \longrightarrow y \mid xy \mid xxy \mid \dots$
 - chiusura lim. $\{x\}_m^n$ come sopra ma per un numero di ripetizioni tra m e n
- Quando si vogliono usare questi simboli letteralmente essi vanno racchiusi tra apici

da EBNF al Parser senza gestione errori

Esempio.

Data la grammatica:

$E ::= T \mid E + T$

$T ::= F \mid T * F$

$F ::= i \mid (E)$

eliminando le ricorsioni

sinistre:

$E ::= T EE$

$EE ::= + T EE \mid \epsilon$

$T ::= F TT$

$TT ::= * F TT \mid \epsilon$

$F ::= i \mid (E)$

Per guidare il parser

si calcolano gli insiemi

FIRST e FOLLOW:

$\text{FIRST}(T EE) = [I_C, \text{LEFT_PAR_C}]$

$\text{FIRST}(+T EE) = [\text{PLUS_C}]$

$\text{FIRST}(FTT) = [I_C, \text{LEFT_PAR_C}]$

$\text{FIRST}(*FTT) = [\text{MUL_C}]$

$\text{FIRST}((E)) = [\text{LEFT_PAR_C}]$

$\text{FIRST}(i) = [I_C]$

$\text{FOLLOW}(EE) = [\text{RIGHT_PAR_C}, \text{EOF_C}]$

$\text{FOLLOW}(TT) = [\text{PLUS_C}, \text{RIGHT_PAR_C}, \text{EOF_C}]$

Implementazione.

```
parser() {  
    init_scan(token);  
    E_p();  
    if (token.code != EOF_C)  
        error(1);  
}  
  
E_p() {  
    if (token.code in first(T EE))  
        { T_p(); EE_p(); }  
    else error(1);  
}  
  
EE_p() {  
    if (token.code in first(+T EE))  
        { scan(token); T_p(); EE_p(); }  
    else if (!token.code in follow(EE))  
        error(3);  
}
```

```
T_p() {  
    if (token.code in first(F TT))  
        { F_p(); TT_p(); }  
    else error(1);  
}  
  
TT_p() {  
    if (token.code in first(*F TT))  
        { scan(token); F_p(); TT_p(); }  
    else if (!token.code in follow(TT))  
        error(3);  
}  
  
F_p() {  
    if (token.code in first((E)))  
        {  
            scan(token); E_p();  
            if (token.code == RIGHT_PAR_C)  
                scan(token);  
            else error(4);  
        }  
    else if (token.code in FIRST(i))  
        scan(token);  
    else error(1);  
}
```

da [E]BNF al Parser, con gestione errori

La gestione degli errori all'analisi ricorsiva discendente si può effettuare con l'aggiunta, *a ciascuna procedura*, di

- un'istruzione di *prologo*
- un'istruzione di *epilogo*

Funzione di salto skip_to().

```
skip_to(code_set s) {  
    if (!token.code in s) {  
        begin_skip_msg();  
        /* messaggio inizio skip */  
        do  
            scan(token);  
        while (!token.code in s);  
        end_skip_msg();  
        /* messaggio uscita skip */  
    }  
}
```


Osservazioni.

- **s**: il codice di prologo salta i prossimi simboli se non sono contenuti nell'insieme dei simboli *validi* fornito:
 - elementi di FIRST(X)
 - simboli terminali di sincronizzazione (per proseguire l'analisi);
più preciso (tiene conto della particolare derivazione seguita)
- **z**: in uscita si garantisce l'appartenenza del token ad un insieme di terminali *susseguenti*
- $s \subseteq z$

Esempio.

Data la grammatica EBNF:

$E ::= T \{ + T \}$

$T ::= F \{ * F \}$

$F ::= (i \mid (E))$

L'implementazione del parser ricorsivo-discendente sarà:

```
parser3() {  
    first[E] = first[T] = first[F] = [I_C, LEFT_PAR_C];  
    init_scan(token);  
    E_p([EOF_C], [EOF_C]);  
}
```

```
E_p(code_set s,z) {  
    skip_to(s+first[E]);  
    if (token.code in first[E]) {  
        T_p(s,[PLUS_C]+z);  
        while (token.code == PLUS_C) {  
            scan(token);  
            T_p(s,[PLUS_C]+z);  
        }  
    } else error(1);  
    skip_to(z);  
}
```

```
T_p(code_set s,z) {  
    skip_to(s+first[T]);  
    if (token.code in first[T]) {  
        F_p(s,[MUL_C]+first[F]+z);  
        while (token.code in ([MUL_C]+first[F]) {  
            if (token.code == MUL_C) scan(token);  
            else error(2);  
            F_p(s,[MUL_C]+first[F]+z);  
        }  
    } else error(1);  
    skip_to(z);  
}
```

```
F_p(code_set s,z) {  
    skip_to(s+first[T]);  
    if (token.code in first[F])  
        if (token.code == LEFT_PAR_C) {  
            scan(token);  
            E_p(s+[RIGHT_PAR_C], [RIGHT_PAR_C]+z);  
            if (token.code == RIGHT_PAR_C) scan(token);  
            else error(4);  
        } else scan(token);  
    else error(1);  
    skip_to(z);  
}
```

ottimizzabile eliminando le istruzioni di sincronizzazione ridondanti

Analisi Sintattica Ascendente

L'analisi sintattica **ascendente** (o *bottom-up*) consente di trattare una classe di grammatiche libere più ampia di quella gestita tramite tecniche top-down e consente una più sofisticata gestione degli errori

In questo caso, si intende costruire l'albero di derivazione a partire dalle foglie (token) risalendo fino al simbolo distintivo

L'albero viene costruito mediante riduzioni successive:

la **riduzione** è l'operazione inversa rispetto alla derivazione

Grammatiche LR(k)

L'Analisi Sintattica Ascendente si avvale in genere della classe di grammatiche $LR(k)$ (spesso per $k = 1$)

- L: la stringa di ingresso viene esaminata da sinistra (**L**eft) verso destra
- R: viene effettuata la *riduzione destra* (**R**ight) processo inverso della derivazione canonica destra
- k: numero di simboli (di *lookahead*) successivi alla parte già riconosciuta della stringa in ingresso utili alla decisione da prendere

Riduzione

Il parser bottom-up effettua l'analisi riducendo la frase in ingresso al simbolo iniziale della grammatica

forma di frase corrente f_i

- si individua una sottostringa che coincide con la parte destra di una produzione della grammatica
- si sostituisce questa sottostringa in f_i con la parte sinistra ottenendo f_{i+1} (forma di frase ridotta di f_i)

La sequenza di forme di frase costituisce una **riduzione destra** della stringa in ingresso

Una forma di frase può contenere varie parti destre di produzioni:

- **parte destra riducibile di f_i** (detta *handle*): la sottostringa che ridotta produce f_{i+1}
- **prefisso LR riducibile di f_i** : un prefisso che contiene la parte destra riducibile come suffisso (ossia non ha altri simboli più a destra)
- **prefisso LR**: un qualunque prefisso di un prefisso riducibile;
- **prefisso LR candidato** alla riduzione: un prefisso LR che ha come suffisso la parte destra di una produzione

Osservazione: prefissi riducibili \subseteq prefissi LR candidati

Proprietà

Una volta effettuata la riduzione di $f_i = axw$
dove ax è il prefisso riducibile ($a, x \in V^*$ e $w \in VT^*$)
nella forma di frase $f_{i+1} = aXw$
mediante la regola $X \rightarrow x$
la stringa aX è ancora un prefisso LR di f_{i+1}

Quindi:

si può utilizzare una pila per memorizzare il prefisso riducibile corrente

$S \longrightarrow E$
 $E \longrightarrow E + T \mid T$
 $T \longrightarrow T * F \mid F$
 $F \longrightarrow i \mid (E)$

Sequenza di riduzioni:

$i + i * i$
 $F + i * i$
 $T + i * i$
 $E + i * i$
 $E + F * i$
 $E + T * i$
 $E + T * F$
 $E + T$
 E
 S

Derivazione canonica destra:

$$\begin{array}{lclclclclcl} S & \Rightarrow & E & & \Rightarrow & E + T & & \Rightarrow & E + T * F & \Rightarrow \\ & & & & & & & & & \\ & \Rightarrow & E + T * i & \Rightarrow & E + F * i & \Rightarrow & E + i * i & \Rightarrow & & \\ & \Rightarrow & T + i * i & \Rightarrow & F + i * i & \Rightarrow & i + i * i & \Rightarrow & & \end{array}$$

Algoritmo

Algoritmi *shift-reduce* (es. alg. LR canonico [Knuth, 65])

- il parser funziona come un PDA
- analisi guidata da tabella (di difficile costruzione)
- **osservazione:** a dispetto della lunghezza dell'input, della forma di frase corrente e della profondità dello stack corrente il numero di situazioni possibili è ridotto:
una per ogni simbolo della grammatica

Tabelle-Guida LR(k)

Si suppone di avere sullo stack il prefisso LR corrente,
le azioni possibili in tabella sono

- sposta** quando il prefisso LR presente sullo stack non è riducibile, si legge il prossimo simbolo in ingresso ponendolo sullo stack
- riduci** quando lo stack compare un prefisso riducibile, si sostituisce la parte destra riducibile con il rispettivo non terminale della parte destra
- accetta** lo stack contiene il simbolo iniziale;
la stringa in input viene accettata
- errore** viene richiamata un'apposita procedura di gestione degli errori

Osservazioni.

- Se la parte destra è riducibile allora si trova certamente nella parte alta dello stack (suffisso del prefisso)
- Per decidere se il prefisso candidato sullo stack sia proprio quello riducibile, il parser LR(1) usa il prossimo simbolo nella stringa di ingresso (*lookahead*)

Esempio (cont.):

$$\overrightarrow{i + i * i}$$

						i	F	T	*	i	F			
i	F	T	E	+	+	+	+	+	+	+	+	E	S	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	

- 1 sposta i
- 2 riduci $F \rightarrow i$
- 3 riduci $T \rightarrow F$
- 4 riduci $E \rightarrow T$
- 5 sposta +
- 6 sposta i
- 7 riduci $F \rightarrow i$
- 8 riduci $T \rightarrow F$
- 9 sposta *
- 10 sposta i
- 11 riduci $F \rightarrow i$
- 12 riduci $T \rightarrow T * F$
- 13 riduci $E \rightarrow E + T$
- 14 riduci $S \rightarrow E$

Una grammatica è adatta all'analisi bottom-up LR(k) se il parser, rilevando un prefisso candidato in cima allo stack, decide univocamente l'azione da intraprendere in base ai prossimi k simboli in ingresso

Tipologie di parsing bottom-up

LR(k) metodo potente ma oneroso nella costruzione della tabella

SLR(k) metodo più debole ma di facile implementazione tabella compatta

LALR(k) metodo quasi al pari di LR(k) ma con tabella compatta come nel caso precedente

Es. Yacc genera simili tabelle per l'analizzatore sintattico