Le Grammatiche LL(K)

15 maggio 2007

Analisi Sintattica

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 dall'analisi lessicale

- Le tecniche di parsing verificano se una stringa di token può essere generata da una certa grammatica.
- La componente del compilatore che si occupa di questa attività è l'**analizzatore sintattico** (o ***parser***)

Attività del Parser

Costruzione dell'albero sintattico in modo

- **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 (della derivazione) 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

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^o **passo** $S \overset{(1)}{\Longrightarrow} \underline{u}XZ$

2^o **passo** $uXZ \overset{(2)}{\Longrightarrow} \underline{uy}WZ$

3^o **passo** $uyWZ \overset{(4)}{\Longrightarrow} \underline{uyw}Z$

la stringa $\underline{uyw}$ non è un prefisso della stringa d'ingresso
quindi bisogna fare *backtracking*

2^o **passo** $uXZ \overset{(3)}{\Longrightarrow} \underline{uy}VZ$

3^o **passo** $uyVZ \overset{(5)}{\Longrightarrow} \underline{uyv}Z$

4^o **passo** $uyvZ \overset{(6)}{\Longrightarrow} \underline{uyvz}$

Analisi Sintattica Deterministica

É preferibile avere un parser deterministico.

(Sa sempre la "mossa" giusta da intraprendere ad ogni momento)

Occorre

1. Porre vincoli alla grammatica per renderla adatta all'analisi
2. Utilizzare l'informazione fornita dai simboli successivi (simboli di lookahead) alla parte di stringa in ingresso già riconosciuta, per guidare l'analizzatore nella scelta della parte destra con cui espandere il simbolo non terminale corrente.

Grammatiche LL(k)

Classe di grammatiche (libere) per cui é possibile costruire analizzatori sintattici deterministici

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

Grammatiche LL(1): il lookahead symbol è il token successivo a quello attualmente elaborato

Esempio

data una grammatica G

con le seguenti produzioni:

(1) $S \longrightarrow print(E)$
(2) $S \longrightarrow while(B)S$
(3) $S \longrightarrow L$
(4) $E \longrightarrow id$
(5) $E \longrightarrow num$
(6) $B \longrightarrow E > E$
(7) $L \longrightarrow SL$
(8) $L \longrightarrow \epsilon$

Se volessimo scoprire cosa é derivabile da S

- se il primo token é "print", il secondo deve essere "(", seguito da una stringa derivabile da "E", seguito da ")"
- se il rpimo token é "while".....

Trasformazione delle grammatiche per l'analisi top-down

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
2. fattorizzazione sinistra:
   $A \longrightarrow yv \mid yw \qquad$ con $y, v \in V^+$, $w \in V^*$
   (uno dei due suffissi può essere vuoto e possono non avere prefissi comuni)
   $A \longrightarrow yA'$
   $A' \longrightarrow v \mid w$

Trasformazione delle grammatiche per l'analisi top-down

Eliminazione della ricorsione sinistra

- Ricorsioni dirette
  $Y \longrightarrow Yx \mid v$, $x \in V^+$, Y non è un prefisso di v.
  Lo stesso linguaggio è generato dalle regole :
  $Y \longrightarrow vY'$
  $Y' \longrightarrow xY' \mid \epsilon$
  In generale
  se esiste la regola con ricorsione sinistra
  $Y \longrightarrow Yx_1 \mid Yx_2 \mid ... \mid Yx_n \mid v_1 \mid v_2 \mid ... \mid v_m$
  questa puó essere sostituita con le regole equivalenti:
  $Y \longrightarrow (v_1 \mid v_2 \mid ... \mid v_m)Y'$
  $Y' \longrightarrow (x_1 \mid x_2 \mid ... \mid x_n)Y' \mid \epsilon$

Esempio

Data la grammatica:

$E \longrightarrow T|-T|E+T|E-T$

$T \longrightarrow F|T*F$

$F \longrightarrow i|(E)$

eliminando le ricorsioni sinistre:

$E \longrightarrow TE'|-TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow i|(E)$

Eliminazione della ricorsione sinistra

- Ricorsioni indirette

  La grammatica:

  $S \longrightarrow Xa \mid bS \mid c$

  $X \longrightarrow Sb \mid aXb$

  presenta una ricorsione sinistra indiretta.

  Sostituendo le parti destre di X in S:

  $S \longrightarrow Sba \mid aXba \mid bS \mid c$

  $X \longrightarrow Sb \mid aXb$

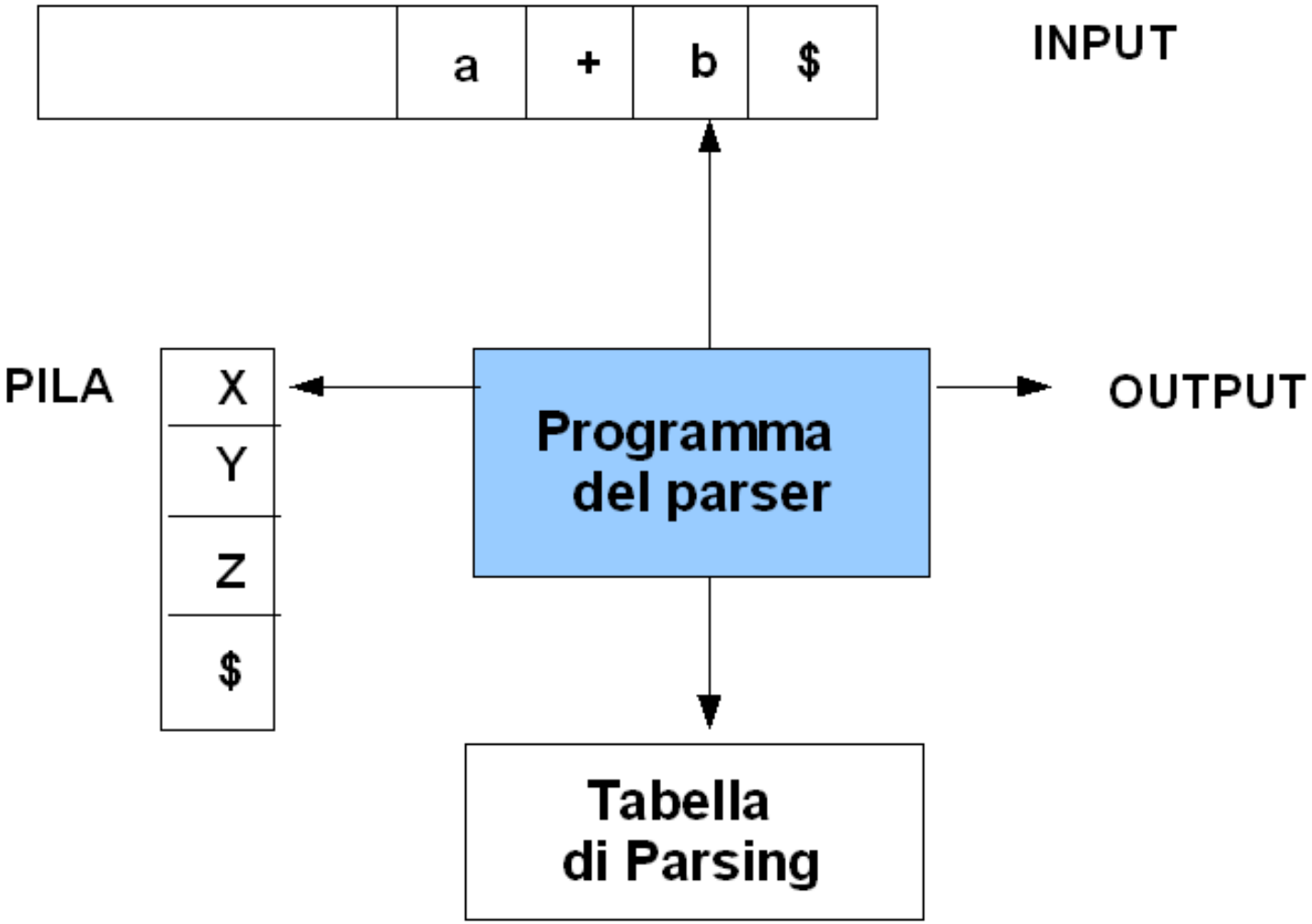
  La ricorsione diventa diretta e si puó eliminare

  $S \longrightarrow aXbaS' \mid bSS' \mid cS'$

  $S' \longrightarrow baS' \mid \epsilon$

  $X \longrightarrow Sb \mid aXb$

Analisi Discendente Guidata da Tabella



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

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)		

Algoritmo

Inizialmente sullo stack c'e' 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'e' 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'e' 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 $

stack	frase	casella
$E	-i + i * i $	tab[E][-]
$E'T-		
$E'T	- i + i * i $	tab[T][i]
$E'T'F		tab[F][i]
$E'T'i		
$E'T'	- i + i * i $	tab[T'][+]
$E'		tab[E'][+]
$E'T+		
$E'T	- i + i * i $	tab[T][i]
$E'T'F		tab[F][i]
$E'T'i		
$E'T'	- i + i * i $	tab[T'][*]
$E'T'F*		
$E'T'F	- i + i * i $	tab[F][i]
$E'T'i		
$E'T'	- i + i * i $	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]] == ∅)
      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

FIRST() e FOLLOW()

La costruzione della tabella del parser é assistita da due funzioni associate alla grammatica (priva di ricorsioni sinistre e fattorizzata).

FIRST(x) é l' insieme dei simboli terminali che possono essere prefissi di stringhe derivabili da x

Esempio

data una grammatica G
con le seguenti produzioni:

(1) $S' \longrightarrow S\$$
(2) $S \longrightarrow AB$
(3) $S \longrightarrow Cf$
(4) $A \longrightarrow ef$
(5) $A \longrightarrow \epsilon$
(6) $B \longrightarrow hg$
(7) $C \longrightarrow DD$
(8) $C \longrightarrow fi$
(9) $D \longrightarrow g$

- $FIRST(S') = \{e, f, g, h\}$
- $FIRST(S) = \{e, f, g, h\}$
- $FIRST(A) = \{e, \epsilon\}$
- $FIRST(B) = \{h\}$
- $FIRST(C) = \{f, g\}$
- $FIRST(C) = \{g\}$
- inoltre... $S \Rightarrow AB \Rightarrow Ahg$
  - $Ahg \Rightarrow hg$
  - $Ahg \Rightarrow efhg$

  dunque anche $FIRST(AB) = \{e, h\}$
- $FIRST(eFB) = \{e\}$
- $FIRST(AC) = \{e, f, g\}$
- $FIRST(AA) = \{e, \epsilon\}$
- ....

FIRST(x) é l' insieme dei simboli terminali che possono essere prefissi di stringhe derivabili da x (con x stringa di terminali e non terminali)

- se $x = \epsilon$, allora $FIRST(x) = \epsilon$
- se il primo simbolo in x é un terminale "a", allora $FIRST(x) = \{a\}$
- se $x = Ax'$, con A non terminale e x' stringa di terminali e non terminali (anche vuota), allora:
  - Se $FIRST(A)$ non contiene ϵ, allora $FIRST(x) = FIRST(A)$
  - Se $FIRST(A)$ contiene ϵ, allora $FIRST(x) = (FIRST(A) - \{\epsilon\}) \cup FIRST(x')$

Da questa definizione segue un algoritmo per il calcolo dei vari insiemi FIRST

FOLLOW(X) insieme dei terminali che in una derivazione possono seguire immediatamente X

Esempio

data una grammatica G

con le seguenti produzioni:

(1) $S' \longrightarrow S\$$
(2) $S \longrightarrow AB$
(3) $S \longrightarrow Cf$
(4) $A \longrightarrow ef$
(5) $A \longrightarrow \epsilon$
(6) $B \longrightarrow hg$
(7) $C \longrightarrow DD$
(8) $C \longrightarrow fi$
(9) $D \longrightarrow g$

quale terminale puó seguire "A" in una derivazione?
ad esempio:
$S' \Rightarrow S\$ \rightarrow AB\$ \rightarrow Ahg\$$ quindi h é nel $FOLLOW(A)$

- $FOLLOW(S') = \{\}$
- $FOLLOW(S) = \{\$\}$
- $FOLLOW(A) = \{h\}$
- $FOLLOW(B) = \{\$\}$
- $FOLLOW(C) = \{f\}$
- $FOLLOW(D) = \{f, g\}$

Tali insiemi possono essere calcolati algoritmicamente

Condizioni LL(1)

Una grammatica si dice $LL(1)$ sse, per ogni produzione $X \longrightarrow x_1|x_2|\ldots|x_n$ 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 $x_j \stackrel{+}{\Longrightarrow} \epsilon$ 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 della stringa in ingresso:

1. se il prossimo simbolo appartiene a $FIRST(x_j)$ l'analizzatore espande X con la parte destra x_j
2. se il prossimo simbolo non appartiene a $FIRST(x_j) \qquad \forall i = 1, \ldots, n$
   ma $x_k \stackrel{*}{\Longrightarrow} \epsilon$ (e quindi nessun altro $x_j \stackrel{*}{\Longrightarrow} \epsilon$)
   e il simbolo successivo appartiene a $FOLLOW(X)$
   allora espandi X con ϵ
3. altrimenti si segnala la situazione di errore

Costruzione Tabella per il Parser

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

$$X \longrightarrow x_1 | x_2 | \ldots | x_n$$

Si pone

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

- Se mendiante la grammatica

$$x_j \stackrel{*}{\Longrightarrow} \epsilon$$

allora

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

Parsing di Grammatiche LL(1)

Esempio: alcuni statements simpleJava Si consideri la grammatica libera con le seguenti produzioni:

(1) $S' \longrightarrow S\$$
(2) $S \longrightarrow id(L);$
(3) $S \longrightarrow if(E)\ S\ else\S$
(4) $L \longrightarrow \epsilon$
(5) $L \longrightarrow EC$
(6) $C \longrightarrow \epsilon$
(7) $C \longrightarrow ,EC$
(8) $E \longrightarrow id$
(9) $E \longrightarrow num$

Calcoliamo i FIRST e FOLLOW per la grammatica:

- S': $FIRST(S') = \{id, if\}$; $FOLLOW(S') = \{\}$
- S: $FIRST(S) = \{id, if\}$; $FOLLOW(S') = \{\$, else\}$
- L: $FIRST(L) = \{id, num, \epsilon\}$; $FOLLOW(L) = \{)\}$
- C: $FIRST(C) = \{,\ ,\epsilon\}$; $FOLLOW(C) = \{)\}$
- E: $FIRST(E) = \{id, num\}$; $FOLLOW(S') = \{),\ ,\}$

Parsing di Grammatiche LL(1)

Esempio: alcuni statements simpleJava Si consideri la grammatica libera con le seguenti produzioni:

Tabella guida

	id	*num*	(	)	;	*if*	*else*	,
S'	S$					S$		
S	id(L)					if (E) S else S		
L	EC	EC		ϵ				
C				ϵ			,EC	
E	id	num						

Gestione Errori (analisi top-down guidata da tabella)

Tipi di errore

1. mancata match tra terminale corrente e quello al top dello stack
2. accesso ad un elemento della tabella che risulta vuoto

Trattamento

1. nel primo caso si hanno due alternative:
   1. scartare un certo numero dei prossimi simbolo in ingresso finchè si trovino simboli per far riprendere l'analisi
   2. inserire (virtualmente) il simbolo mancante in modo da riprendere l'analisi (senza causare altri errori)
2. 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