Linguaggi di Programmazione
Corso C

Parte n.9
Analisi Sintattica

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

Dipartimento di Informatica
Università degli Studi di Bari

Analizzatore Sintattico

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)

Costruzione

- **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 (programma 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

Esempio

data una grammatica con 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 Discendente Deterministica

Analisi Discendente, problemi

- 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 in cui

L**:** la stringa in ingresso è esaminata da sinistra (Left) a destra

L**:** viene costruita la derivazione canonica sinistra

k**:** numero di simboli di lookahead per poter scegliere la parte destra

Trasformazione

problemi:

1. ricorsione sinistra nelle produzioni della grammatica
2. presenza di prefissi comuni in parti destre relative allo stesso NT

rimedi:

1. eliminazione della ricorsione sinistra (vedi algoritmo parte n. 7)
2. fattorizzazione sinistra:
   $A \longrightarrow yv|yw$ con $y, v \in V^+$, $w \in V^*$
   (uno dei due suffissi può essere vuoto e non hanno prefissi comuni)
   $A \longrightarrow yA'$
   $A' \longrightarrow v|w$

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:

$E \longrightarrow TE'|-TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow i|(E)$

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)		

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'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)

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);
         foreach (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

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'][$]
$E'		tab[E'][$]
$		

Grammatiche LL(1)

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 $X \longrightarrow x_1|x_2|\ldots|x_n \in P$
  $FIRST(X) = \{t \in VT \mid X \stackrel{+}{\Longrightarrow} tu,\ u \in V^*\} =$

$$= \bigcup_{i=1}^{n} FIRST(x_i)$$

- $\forall y = y_1 y_2 \cdots y_n, \quad y_i \in VN \cup VT$
  - se $y_1 \stackrel{+}{\not\Longrightarrow} \epsilon$ allora $FIRST(y) = FIRST(y_1)$
  - se $y_i \stackrel{+}{\Longrightarrow} \epsilon, i = 1, \ldots, k$ e $y_{k+1} \stackrel{+}{\not\Longrightarrow} \epsilon$
    allora $FIRST(y) = \bigcup_{i=1}^{k+1} FIRST(y_i)$
  - se $y_i \stackrel{+}{\Longrightarrow} \epsilon, \forall i = 1, \ldots, n$
    allora $FIRST(y) = \bigcup_{i=1}^{n} FIRST(y_i)$

FOLLOW insieme dei terminali che in una derivazione possono seguire immediatamente $X \in VN$
Formalmente: $FOLLOW(X) = \{t \in VT \mid S \stackrel{+}{\Longrightarrow} uXtv \wedge u, v \in V^*\}$

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 $v \stackrel{+}{\not\Longrightarrow} \epsilon$ ovvero
$FOLLOW(X) \supseteq FIRST(v) \cup FOLLOW(Y)$ altrimenti

Tali insiemi possono essere calcolati algoritmicamente (vedi testo)

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
   $FIRST(X) \cap FOLLOW(X) = \emptyset$

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 $FIRST(x_j) \forall i = 1, \ldots, n$
   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 $X \longrightarrow x_1|x_2|\ldots|x_n$ della grammatica $LL(1)$

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

se $x_j \stackrel{*}{\Longrightarrow} \epsilon$ allora $tab[X][b] = \epsilon \qquad \forall b \in FOLLOW(X)$

Gestione Errori

nell’analisi top-down guidata da tabella

tipi di errore

1. mancata corrispondenza del simbolo terminale in cima allo stack con quello indirizzato dall’indice
2. accesso ad un elemento della tabella che risulta vuoto

trattamento

1. nel primo caso si hanno due alternative:
   (a) scartare un certo numero dei prossimi simbolo in ingresso finchè si trovino simboli per far riprendere l’analisi
   (b) 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

L’informazione sull’analisi già effettuata è di difficile reperimento

se ne trova traccia sullo stack

bisogna basarci su 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

Analisi ricorsiva discendente per BNF senza gestione errori

Si costruisce, per ogni non terminale, una procedure contrassegnata dal suo nome con l'aggiunta del suffisso "_p"

Implementazione Algoritmo

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

1. se $X \overset{+}{\not\Longrightarrow} \epsilon$ 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 x2 */ }
     else error();
}
```

2. se $X \overset{+}{\Longrightarrow} \epsilon$ 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 x2 */ }
     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 `xi`
  1. tante istruzioni quanti sono i simboli di `xi` (terminali e non) nell'ordine in cui vi compaiono
  2. per un simbolo non terminale viene effettuata la chiamata della procedura relativa a quel simbolo
  3. per un simbolo terminale viene effettuata la seguente istruzione:
     `if (token.code = t_c) scan(token); else error(n);`
     per il primo simbolo (terminale) della parte si omette il test perchè già effettuato in lookaead, quindi basta chiamare `scan(token)` per passare al prossimo token
- nel main del parser:
  `init_scan(token); S_p;`
  che inizializza lo scanner e fa partire il riconoscimento del simbolo iniziale della grammatica

Esempio

Data la grammatica:

E $\longrightarrow$ T | E + T

T $\longrightarrow$ F | T * F

F $\longrightarrow$ i | (E)

eliminando le ricorsioni sinistre:

E $\longrightarrow$ T EE

EE $\longrightarrow$ + T EE | ϵ

T $\longrightarrow$ F TT

TT $\longrightarrow$ * F TT | ϵ

F $\longrightarrow$ i | (E)

calcolando gli insiemi FIRST e FOLLOW

FIRST(T EE)	=	[I_C, LEFT_PAR_C]
FIRST(+T EE)	=	[PLUS_C]
FIRST(FTT)	=	[I_C, LEFT_PAR_C]
FIRST(*FTT)	=	[MUL_C]
FIRST((E))	=	[LEFT_PAR_C]
FIRST(i)	=	[I_C]
FOLLOW(EE)	=	[RIGHT_PAR_C, EOF_C]
FOLLOW(TT)	=	[PLUS_C, RIGHT_PAR_C, 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)
         else error(4);
      }
   else if (!token.code in FIRST(i))
      scan(token);
      else error(1);
}
```

Analisi ricorsiva discendente per [E]BNF 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***

esempio:

```
X_p(code_set s,z) {
/* X_ NT che può produrre la stringa vuota */
/* s ins. simboli di sincronizzazione */
/* z ins. simboli di susseguenti */
   skip_to(s+first(X)));
   /* token contiene un simbolo di first(X) ∪ s*/
   if (token.code in first(X)) {
   /* codice relativo al corpo di X_p */
   } else error(...);
   skip_to(z); /* epilogo */
}

Y_p(code_set s,z) {
/* Y_ NT che può produrre la stringa vuota */
   skip_to(s+first(Y)+z));
   /* token contiene un simbolo di first(X) ∪ s ∪ z*/
   if (token.code in first(X)) {
   /* codice relativo al corpo di Y_p */
   }
   skip_to(z); /* epilogo */
}
```

```
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 $\longrightarrow$ T { + T }

T $\longrightarrow$ F { * F }

F $\longrightarrow$ (i | (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);
}
```

si può ottimizzare 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 sofisticata gestione degli errori

Essa si avvale in genere della classe di grammatiche LR(1)

Grammatiche LR(k)

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 la *riduzione destra* della stringa in ingresso

- una forma di frase può contenere varie parti destra di produzioni:
  si chiama *parte destra riducibile di* f_i la sottostringa che ridotta produce f_{i+1}
- il *prefisso LR riducibile* di f_i è un prefisso che contiene la parte destra riducibile come suffisso (cioè non ha altri simboli a destra di essa)
- un *prefisso LR* è un qualunque prefisso di un prefisso riducibile;
- se un prefisso LR ha come suffisso una parte destra di produzione si dirà *prefisso LR candidato* alla riduzione

Osservazione

prefissi riducibili $\subseteq$ prefissi LR candidati

Proprietà

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

Quindi:
si può utilizzare una pila per memorizzare il prefisso riducibile corrente

Esempio

S $\longrightarrow$ E
E $\longrightarrow$ E + T | T
T $\longrightarrow$ T * F | F
F $\longrightarrow$ i | (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:

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

Tabelle-Guida LR(k)

Sullo stack si suppone di avere il prefisso LR corrente

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

Se la parte destra è riducibile allora si trova certamente nella parte alta dello stack

Per decidere se il prefisso candidato sullo stack sia quello riducibile,
il parser LR(1) usa il prossimo simbolo nella stringa di ingresso (look-ahead)

Esempio (continua)

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

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

Tabelle Guida per l’Analisi LR(k)

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 simbolo 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