Linguaggi di Programmazione
Corso C

Parte n.12
Analisi Semantica e Traduzione

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

Dipartimento di Informatica
Università degli Studi di Bari

Analisi Semantica e Traduzione

- raccolta informazioni sugli identificatori per costruire la tavola dei simboli con segnalazione degli eventuali errori semantici
- traduzione del sorgente in codice intermedio per la generazione di un codice oggetto efficiente

Si prenderà in considerazione un'analisi semantica guidata dalla analisi sintattica: *traduzione guidata dalla sintassi*

Analisi Semantica

1. raccolta informazioni sugli identificatori nella *tavola dei simboli*

   occorre progettare il contenuto interno della tavola organizzando le informazioni da includere

2. verifica della correttezza nell'impiego degli identificatori e dei costrutti del linguaggio in generale

   controlli statici (a compile-time):

   - identificatori dichiarati prima dell'uso ed in modo univoco
   - compatibilità operandi
   - chiamate congruenti con le dichiarazioni (visibilità)

3. segnalazione degli eventuali errori semantici

   non appena possibile per evitare sia di riportare lo stesso errore sia di segnalare errori in cascata

Analisi Semantica guidata dalla Sintassi

L'analisi sintattica può fare da guida per l'analisi semantica e la traduzione in codice intermedio

Le grammatiche libere vanno estese con

- l'associazione di *attributi* ai simboli della grammatica e
- l'integrazione delle *regole sintattiche* con *regole semantiche*

Grammatiche ad Attributi

Ogni simbolo di una grammatica può avere degli

attributi

= variabili di un linguaggio di programmazione con un nome ed un tipo

L'albero sintattico verrà esteso
in ciascun nodo, si riportano i valori degli attributi del simbolo
calcolati dalle regole semantiche associate alle regole sintattiche

Regole Semantiche

In genere sono poste a destra di una produzione
e racchiuse tra i metasimboli { e }

All'interno ci sono sequenze di istruzioni (tipicamente assegnazioni) dette

azioni semantiche

Classi di Attributi

- attributi **ereditati**
- attributi **sintetizzati**

Data una regola sintattica $X \longrightarrow x_1 x_2 \ldots x_n$ ed un'azione semantica associata A questa può assegnare un valore:

1. ad un attributo sintetizzato del simbolo non terminale X
   A è del tipo
   $$s := f(z_1, z_2, \ldots, z_m)$$
   con s attributo sintetizzato di X
   e ciascun z_i è ereditato di X o sintetizzato di x_j $(1 \leq j \leq n)$
2. ad un attributo ereditato di un simbolo non terminale nella parte destra
   A è del tipo
   $$e := f(z_1, z_2, \ldots, z_m)$$
   con s attributo ereditato di x_j $(1 \leq j \leq n)$
   e ciascun z_i è ereditato di X o sintetizzato di x_j $(1 \leq j \leq n)$

Osservazioni

- Un nodo N dell'albero sintattico può ricevere attraverso un attributo ereditato un'informazione che può dipendere da un attributo ereditato del nodo padre e da qualche attributo sintetizzato di altri nodi aventi lo stesso padre di N
- Un attributo ereditato porta informazione dal contesto del nodo ricevente
- Un attributo sintetizzato determina un flusso ascendente di informazione elaborata da qualche discendente
- Simboli terminali = foglie
  possono avere solo attributi sintetizzati (forniti dallo scanner)

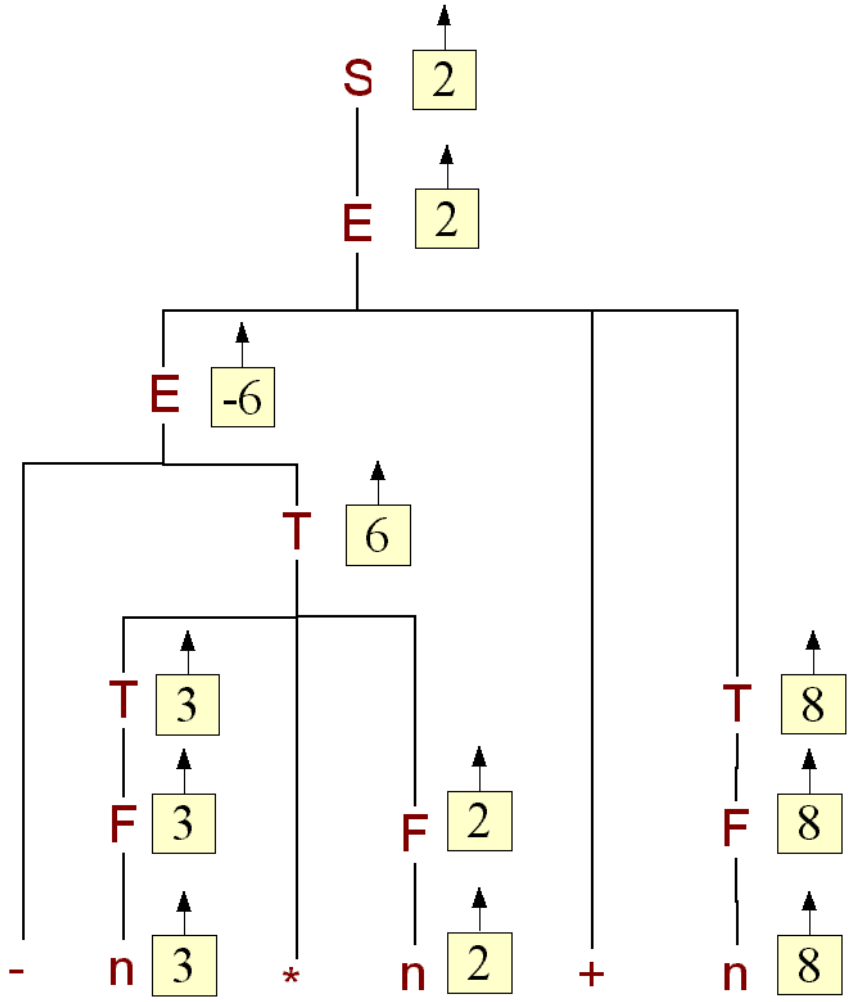
Esempio

regola sintattica	regola semantica
S $\longrightarrow$ E	{S.V:=E.V}
E $\longrightarrow$ E + T	{E1.V:=E2.V+T.V}
E $\longrightarrow$ T	{E.V:=T.V}
E $\longrightarrow$ -T	{E.V:=-T.V}
T $\longrightarrow$ T * F	{T1.V:=T2.V*F.V}
T $\longrightarrow$ F	{T.V:=F.V}
F $\longrightarrow$ (E)	{F.V:=E.V}
F $\longrightarrow$ n	{F.V:=n.IVAL}

Osservazioni

- attributi indicati con la notazione *nome_simbolo.nome_attributo*
- se uno stesso simbolo compare più volte in una regola sintattica, nella regola semantica le varie istanze vengono numerate progressivamente
- gli attributi sintetizzati associati ai simboli terminali rappresentano l'informazione fornita dallo scanner

Esempio (continua)



Tipi di Grammatiche ad Attributi

Attributi:

- *normali*: il risultato semantico prodotto è dato dal valore degli attributi sintetizzati del simbolo iniziale della grammatica S
- *opzionali*: grammatiche che producono effetti collaterali
  es. scrittura di un valore calcolato `{write(E.V)}`

Ordine di calcolo dell'albero sintattico:

- *costruzione totale*:
- *costruzione parziale*: ordine top-down o bottom-up

Calcolo attributi:
Azione semantica possibile quando le informazioni sono disponibili cioè quando tutti gli attributi usati sono stati avvalorati

$$\text{Ordine azioni semantiche} \neq \text{Ordine tra regole sintattiche}$$

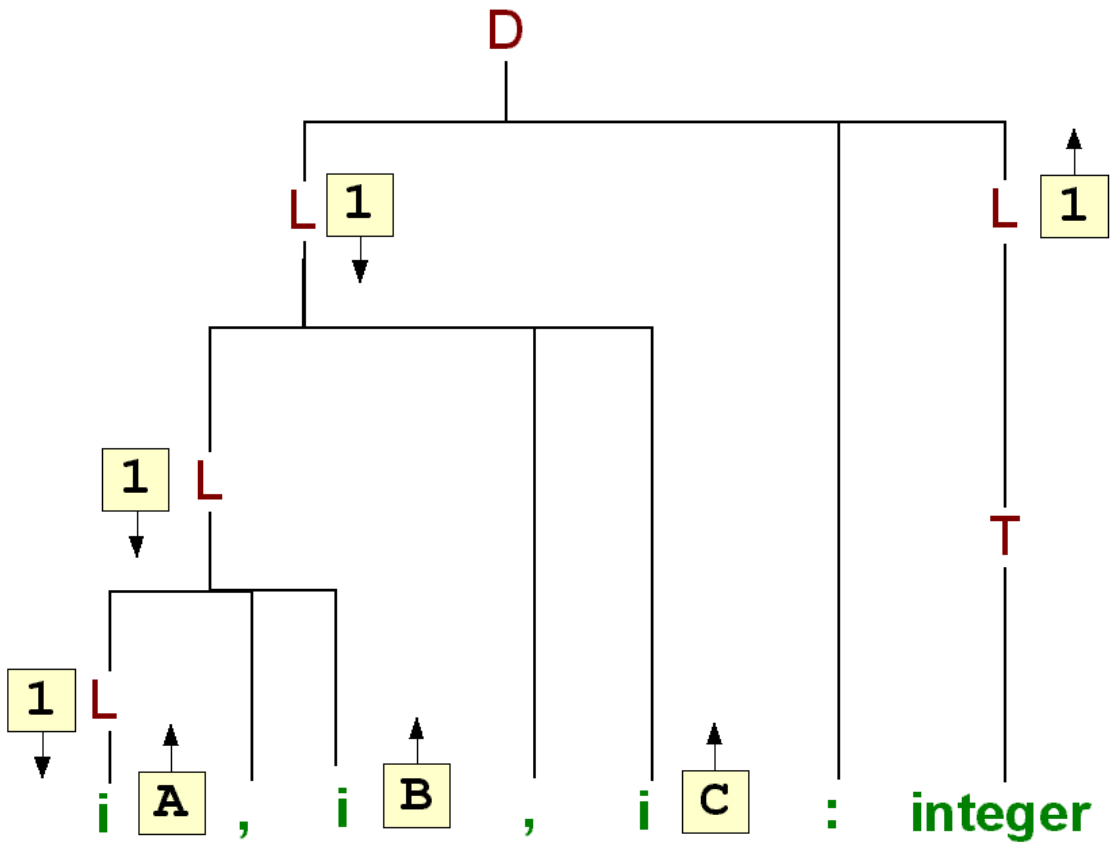
(si utilizza una tecnica basata sul *grafo delle dipendenze* tra attributi)

Esempio

regola sintattica	regola semantica
D ⟶ L : T	`{L.TY:=T.TY}`
T ⟶ integer	`{T.TY:=1}`
T ⟶ real	`{T.TY:=2}`
L2 ⟶ L1, i	`{L2.TY:=L1.TY; writeln(i.NAME, L1.TY)}`
L ⟶ i	`{writeln(i.NAME, L1.TY)}`

Osservazioni

- `T.TY` attr. sintetizzato intero
- `L.TY` attr. ereditato intero



Restrizioni sugli Attributi

- *Grammatiche ad attributi di tipo **S***:
  contengono solo attributi sintetizzati
- *Grammatiche ad attributi di tipo **L***:
  esclusivo il flusso di informazioni da sinistra a destra
  per ogni regola

  $$X \longrightarrow x_1 x_2 \cdots x_n$$

  gli attributi ereditati da ciascun x_i possono dipendere
  - da quelli sintetizzati dei simboli x_j con $j < i$ e
  - da quelli ereditati di X

Per le grammatiche di tipo S e di tipo L

è sufficiente considerare l'informazione sintattica contenuta nello stack

Pertanto si possono applicare le tecniche di analisi LL o LR

Strutture Dati per l'Analisi Semantica

Tabella dei simboli: formata da *descrittori di identificatori*

chiave=nome identificatore

1. tipo pre-definito: occupazione di memoria
2. tipo array:
   - occupazione di memoria
   - puntatore ad un *descrittore di array*
     - numero indici
     - valore parte costante
     - puntatore al descrittore del tipo dell'array
     - puntatore ai descrittori degli indici (inf,sup)
3. tipo puntatore:
   - occupazione memoria
   - puntatore all'elemento *descrittore del tipo* puntato
4. tipo record:
   - occupazione memoria
   - lista dei campi (come per le variabili)

5. dichiarazione variabili e parametri:
   - offset rispetto al blocco
   - puntatore descrittore di tipo
6. contesti di programma:
   - occupazione del blocco di attivazione relativo
   - puntatore a *descrittore di modulo*:
     - indice della quadrupla di inizio
     - occupazione parametri
     - puntatore alla lista id. locali (a procedure o funzioni) e al risultato
7. forward reference
8. costanti
   - intere o assimilabili
   - reali

Errori Semantici

1. id. non dichiarato
2. id. già dichiarato
3. tipo non valido
4. estremi indice non validi
5. operando non valido
6. operatore non ammesso
7. nome tipo non valido
8. indice non valido
9. numero di indici errato
10. espressione non valida
11. assegnazione errata
12. nome procedura o funzione errato
13. numero parametri errato
14. operando non intero o reale
15. operando non booleano
16. tipi non compatibili
17. tipi non uguali
18. operando non puntatore
19. argomento non valido
20. campo record non valido
21. operando non array
22. operando non record
23. parametro attuale dal tipo diverso dal richiesto
24. parametro attuale di modo diverso

Procedura Gestione Errori

`error(codice_errore, id )`

Se il parametro id è NULL_ID allora il messaggio d’errore viene sempre stampato

Altrimenti viene consultata una lista di errori e si stampa il messaggio se il codice non vi appartiene già

Ottimizzazione

Il codice prodotto dall'Analisi Semantica può essere migliorato prima della traduzione in codice oggetto

Obiettivi:

- ridurre il numero di istruzioni nel formato intermedio (quadruple)
- ridurre il fabbisogno di memoria temporanea
- rendere più efficiente l'esecuzione

Tipi di Ottimizzazione

- ***Uso immediato e propagazione delle costanti***
  Insieme alla fase di analisi semantica:
  - sostituzione del riferimento di una costante simbolica con il suo valore effettivo
  - conversioni di tipo
  - calcolo espressioni aritmetiche (o logiche) che coinvolgono solo costanti:
    ```
    const int pi = 3.14;
    ...
    perimetro = 2*pi*r;
    ```
    porta alla generazione delle quadruple:
    ```
    (*,6.28,R,T1)
    (MOV,T1,-,perimetro)
    ```

  Ciò equivale all'espansione di macro (*constant folding*)

- ***Eliminazione del codice inutile***
  blocchi di codice che non saranno mai eseguiti:
  ```
  const int debug = FALSE;
  ...
  if (debug) {...}  /* if eliminabile */
  ```
  La propagazione delle costanti può rivelare tali blocchi inutili

- ***Manipolazione delle espressioni***
  Codice più efficiente, trasformando le espressioni in forme equivalenti:
  ```
  z = (-x-y)/(p-q);
  ```
  tradotta (per risparmiare una operazione di negazione):
  ```
  z = (x+y)/(q-p);
  ```

- ***Eliminazione delle espressioni comuni***
  Una espressione può comparire più volte senza cabiar valore in un blocco di codice; allora la si calcola una sola volta:
  ```
  x = y*y-w*(p+q);
  z = y*y+2+8/(p+q);
  ```
  diventa
  ```
  t1 = y*y;
  t2 = p+q;
  x = t1-w*t2;
  z = t1+2+8/t2;
  ```

- ***Eliminazione delle copie***
  copia = assegnazione del valore di una variabile ad un'altra variabile
  `v2 := v1`
  Se `v2` non è più usata, si elimina la copia e si sostituiscono le occorrenze di `v2` con `v1`

- ***Ottimizzazione dei cicli***

  Analisi statistica $\longrightarrow$ la maggior parte del tempo spesa nei cicli.

  1. Spostamento *codice costante* (invariante rispetto al ciclo):

```
for (i=1; i<=50; i++) {
     delta = x+y;
     a[i] = a[i]+delta;
}
```

  diventa

```
delta = x+y;
for (i=1; i<=50; i++)
     a[i] = a[i] + delta;
```

  2. *Variabili indotte*: assumono valori dipendenti dall'indice del ciclo si possono spesso eliminare o se ne può facilitare il calcolo:

```
for (j=1; j<=20; j++) {
     alfa = j*5;
     a[alfa] = b;
}
```

  diventa

```
alfa = 0;
for (j=1; i<=20; j++){
     alfa = alfa + 5;  /* + anziche' * */
     a[alfa] = b;
}
```

- ***Fattorizzazione del codice comune***
  Una istruzione o un blocco può essere presente in rami alternativi del flusso di controllo
  Questa istruzione (o blocco) può essere estratto e spostato prima della diramazione:
  `if (i>j) x = a + b; else y = a + b;`
  diventa
  ```
  t = a + b;
  if (i>j) x = t; else y = t;
  ```

- ***Calcolo parte costante indirizzo di elemento di array***

- ***Valutazione accorciata di espressioni logiche***