Linguaggi di Programmazione
Corso C

Parte n.11

Tabella dei Simboli e Formato Intermedio

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

Dipartimento di Informatica
Università degli Studi di Bari

Analisi Semantica

I prodotti principali dell'Analisi Semantica sono

Tavola dei Simboli informazioni sugli identificatori nel sorgente

Formato Intermedio rappresentazione elementare adatta alla generazione del codice oggetto

Tavola dei Simboli

Il compilatore deve tenere traccia di

- identificatori utilizzati nel sorgente
- loro attributi

Queste informazioni sono memorizzate in una apposita tabella detta

tavola dei simboli

Il compilatore consulta la tavola dei simboli ogni volta che incontra un identificatore:

- se esso non è presente nella tavola (prima occorrenza), allora si introduce un nuovo elemento nella tabella con una serie di attributi
- se esso è già presente nella tavola, allora se ne aggiornano, eventualmente, i suoi attributi

Nelle coppie

```
(nome, attributi)
```

il `nome` è di solito una stringa che fa da chiave per la ricerca degli elementi

Progetto della Tavola dei Simboli

Difficoltà:

- numero identificatori non noto a priori
- ricerca ed inserimento efficienti
- omogeneizzazione dei record per ottimizzare lo spazio

Realizzazione con Liste

Liste di record `(nome, attributi)`

- non ordinate:
  inserzione a fondo lista;
  ricerca a partire dalla testa
  gestione semplice ma inefficiente per grosse tabelle
- ordinate:
  ricerca ordinata
  inserzione più onerosa rispetto al caso precedente
- riorganizzazione dinamica:
  l'elemento cercato viene posto in testa alla lista perchè si presume che le istruzioni vicine nel sorgente potranno farvi riferimento

Realizzazione con Alberi Binari

Alberi in cui i nodi sono record `(nome, attributi)`
con due puntatori al sottoalbero sinistro ed al sottoalbero destro

- organizzazione: l'albero mantiene un ordinamento ponendo nel sotto albero sinistro gli elementi minori e in quello destro gli elementi maggiori del nodo considerato
- inserimento: sempre più oneroso man mano che l'albero cresce
- ricerca: molto efficiente dell'ordine del logaritmo del numero dei record

Realizzazione con Tabelle Ordinate

Tabelle in cui gli elementi `(nome, attributi)` sono ordinati alfabeticamente in base al `nome`

- ricerca: molto efficiente dell'ordine del logaritmo del numero dei record tramite l'algoritmo di ricerca binaria
- inserimento: oneroso se la dimensione massima non è nota

Realizzazione con Tabelle Hash

Tabelle di n elementi `(nome, attributi)`

La corrispondenza nome-posizione viene calcolata da un'apposita

funzione hash

dato un nome fornisce un intero nel range 0..n-1

Deve essere calcolabile in maniera efficiente e deve essere in grado di distribuire uniformemente i record nella tabella

- ricerca: in assenza di *collisioni* il tempo è costante
- inserimento: tempo costante a meno di casi di collisione

Le collisioni sono trattate in vari modi:

- spostandosi in avanti o indietro rispetto alla posizione calcolata fino a trovare un elemento non impiegato
- mantenere per ogni posizione una lista degli elementi in collisione

Realizzazione a Livelli

Rispecchia le regole di visibilità dei blocchi del programma

Viene mantenuto uno *stack* di tavole dei simboli
(realizzate secondo le tecniche precedenti):

- quando si apre un nuovo contesto si genera una nuova tabella da porre in cima allo stack;
  questa diventa quella corrente e gli identificatori vanno inseriti nella stessa
- all'uscita dal contesto si opera il pop dello stack e si torna alla tabella precedente
- la ricerca va eseguita dal contesto più interno a quello più esterno

Alternativa: singola tabella-hash

- occorre aggiungere alla chiave anche l'attributo indicante il livello del sottoprogramma contenente l'identificatore
- all'uscita dal contesto si eliminano gli identificatori definiti a quel livello (si può creare una lista ausiliaria degli id. allo stesso livello)

Formato Intermedio dei Programmi

Il codice intermedio viene prodotto in fasi di analisi semantica e traduzione per poter poi passare all'ottimizzazione ed alla generazione del codice

Istruzioni elementari (simili ad un linguaggio assembler)

Requisiti:

- possono rappresentare ogni costrutto del linguaggio
- facilitano l'ottimizzazione
- facilitano la traduzione in codice oggetto

Le variabili verranno di seguito rappresentate col loro nome sebbene in realtà debbano rappresentare puntatori agli elementi della tavola dei simboli

Formato a 3 Indirizzi (2 di operandi e 1 per il risultato)

Forma tipica:

`Z:= X OPR Y`

dove:

- `Z`, `X` e `Y` sono variabili dichiarate o temporanee
- `OPR` è un operatore aritmetico

espressioni aritmetiche più generali vanno spezzate:

esempio: `A := X + Y * Z` viene tradotta come

1. `T1 := Y * Z`
2. `T2 := X + T1`
3. `A := T2`

con `T1` e `T2` var. temporanee utilizzate per spezzare l'espressione immagazzinate nei registri del processore o in apposite zone di memoria

Altre istruzioni:

- `Z := OPR X` (assegnazione con operatore unario)
  L'operatore può essere il meno unario o un operatore di conversione
- `Z := X` (copia)
- `goto q` (salto incondizionato)
  dove `q` è l'indice dell'istruzione da eseguire in seguito
- `if X REL Y goto q` (salto condizionato)
  dove `REL` è una operatore relazionale
- `Z := X[Y]` (lettura elem. array)
- `X[Y] := Z` (scrittura elem. array)
- `T := ^V OPR n` (lettura indirizzo)
  l'indirizzo di `V` cui può essere sottratto o aggiunto `n` è copiato in `T`
- `Z := X^` (lettura var. puntata)
- `X^ := Z` (scrittura var. puntata)
- `param X` (passaggio parametro)
  `X` copiato nello stack
- `call S` (chiamata sottoprogramma)
- `start S` (prologo (sotto-)programma)
- `halt` oppure `return` (epilogo (sotto-)programma)
- `result F T` (allocazione spazio per risultato funzione)
  la funzione `F` restituisce un risultato per raccogliere il quale viene allocata la var. temporanea `T`

Realizzazione con record

- quadruple
- triple
- triple indirette

Quadruple $(\mathbf{OPR}, \mathbf{ARG1}, \mathbf{ARG2}, \mathbf{RESULT})$

esempio: `A := B + C * D` viene tradotta come

OPR	ARG1	ARG2	RESULT
*	C	D	T1
+	B	T1	T2
:=	T2		A

Triple e Triple Indirette $(\mathbf{OPR}, \mathbf{ARG1}, \mathbf{ARG2})$

Il risultato dell'operazione è denotato dall'indice della tripla stessa

esempio: `A := B + C * D` viene tradotta come

INDICE	OPR	ARG1	ARG2
0	*	C	D
1	+	B	(0)
2	:=	(1)	A

Variante:
codice intermedio =
sequenza di indici di triple che rimandano alla tabella delle triple

Le triple non sono così duplicate:
basta introdurre un riferimento indiretto quando serve

Notazione Polacca (o Postfissa)

Consente di calcolare rapidamente espressioni senza analizzarle per trovare i vincoli di precedenza degli operatori:

ogni operatore segue gli operandi cui si applica

sono pertanto eliminate le parentesi

esempio: `A := -x - B * (C + D)` viene tradotta come

`^A X -u B C D + * - :=`

dove il meno unario viene rappresentato da `-u` per differenziarlo da `-`
e `^A` rappresenta l'indirizzo di `A`

[N.B.] Gli operatori compaiono nello stesso ordine (ma in verso opposto) rispetto al programma sorgente

Il **calcolo dell'espressione**, basato sull' uso di uno stack, scandisce via via la stringa dell'espressione:

- operando: si fa il push sullo stack
- operatore binario: si applica ai due operandi prelevati in cima allo stack e si fa il push del risultato
- operatore unario: si applica all' operando in cima allo stack e si fa il push del risultato
- assegnazione: lo stack contiene in cima il valore dell'espressione da assegnare alla variabile il cui indirizzo è l'elemento successivo da prelevare dallo stack