Tabella dei Simboli

Giovedì 13 Dicembre 2012

Tabella dei simboli (Symbol table)

- Il compilatore, nelle sue varie fasi, deve tenere traccia:
  - degli identificatori utilizzati nel codice sorgente;
  - dei loro attributi (tipo e caratteristiche).

Tabella dei simboli (Symbol table)

- Queste informazioni sono memorizzate in un'apposita struttura dati detta

Tabella dei simboli (***Symbol table***)

da consultare ogni volta che si incontra un identificatore:

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

Quando costruire ed interagire con la TS

- La TS è consultata in tutte le fasi del processo di compilazione.
- La costruzione della TS avviene, almeno in linea teorica, a partire dall'analisi lessicale e completata nelle fasi successive.
  - Nell'analisi lessicale:
    - è creata una entry nella tabella per ogni nuovo identificatore incontrato (variabili, tipi, nomi di funzioni, etichette, campi, ….);
    - a questa entry non è associato alcun valore: di solito non si conosce né il tipo né le caratteristiche.
  - Nell'analisi sintattica e semantica:
    - la TS viene riempita con informazioni sui tipi e sulle caratteristiche degli identificatori.
- Viene coinvolta anche nella fase di generazione del codice

Struttura della tabella dei simboli

- La TS è una struttura dati astratta per rappresentare insiemi di coppie

  < nome_id, lista_attributi >

  ove:

  nome_id, che di solito è una stringa, rappresenta la chiave per accedere a ***lista_attributi*** cioè alle informazioni associate in modo univoco all'identificatore stesso.

Operazioni principali sulla TS

- La TS, che può essere considerata una struttura dati “dizionario”, ha tre operazioni di base:
  - ***Insert*** (inserimento)

    per memorizzare le informazioni associate ad un identificatore.
  - ***Lookup*** (accesso e/o ricerca)

    per ritrovare le informazioni associate ad un determinato identificatore.
  - ***Delete*** (cancellazione)

    per rimuovere le informazioni associate ad un determinato identificatore quando esso non è più visibile.

Problemi

- Il problema principale è quello di individuare l'organizzazione più idonea della struttura dati che implementa la TS.
- In genere, due sono i principali fattori che possono influenzare la scelta dell'organizzazione di una struttura dati:
  - lo spazio di memoria occupato
  - la velocità di accesso
- Poiché la TS normalmente è consultata migliaia di volte nel corso della compilazione, la velocità di accesso avrà maggiore priorità rispetto allo spazio di memoria occupato.
- Bisogna considerare, inoltre, che anche le specifiche del linguaggio sorgente posso influenzare l'organizzazione della TS.

Struttura della Tabella dei simboli

- Illustreremo le principali organizzazioni della TS e vedremo come le caratteristiche del linguaggio sorgente possono influenzare tali strutture.

Realizzazione della Tabella dei simboli

- La TS può essere organizzata come:
  - un semplice array
  - una lista concatenata
  - un albero binario di ricerca
  - una tabella hash

La Tabella dei simboli come *ARRAY*

- Vantaggi:
  - semplicità di implementazione
  - facile accesso alle sue componenti
  - spazio occupato in memoria relativo ai soli dati
- Svantaggi:
  - dimensione della struttura da fissare a priori (ciò implica di fissare un limite sul numero di identificatori che si possono gestire)
  - ricerche non particolarmente efficienti
- La struttura ad array è raramente usata tranne che per semplici compilatori.

La TS come *lista concatenata*

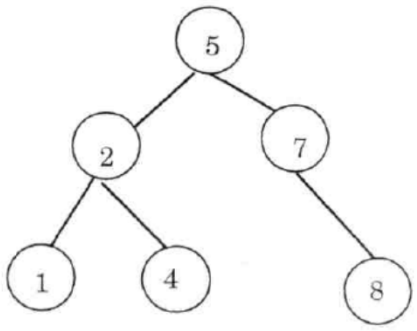
- In questo caso la struttura è dinamica.
- La ricerca resta lineare ma è possibile ottimizzarla:
  - mantenendo la lista ordinata
  - utilizzando il metodo della ***riorganizzazione dinamica***

  Secondo questo metodo le voci a cui si accede sono spostate via via verso la testa della lista cosicché i simboli più usati avranno un tempo di accesso inferiore (si presume che le istruzioni vicine nel sorgente potranno farvi riferimento).
- Gestione semplice ma inefficiente per grosse tabelle.

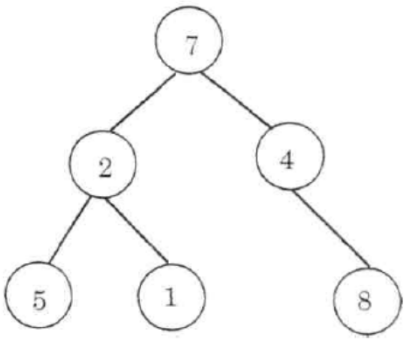
La TS come ABR

- Gli ABR sono particolari alberi binari tali che per ogni nodo mantengono un ordinamento ponendo nel sottoalbero di sinistra gli elementi minori e in quello di destra tutti gli elementi maggiori del nodo considerato.
- Si dimostra che se l'albero è costruito in modo casuale ha un altezza di approssimativamente 1.39log n, dove n è il numero dei nodi.
  Le tre operazioni hanno un costo proporzionale all'altezza. ☺
  Frequenti cancellazioni possono sbilanciare l'albero. ☹
- Supportano efficienti operazioni di ordinamento.

Albero binario di ricerca:



Albero binario:



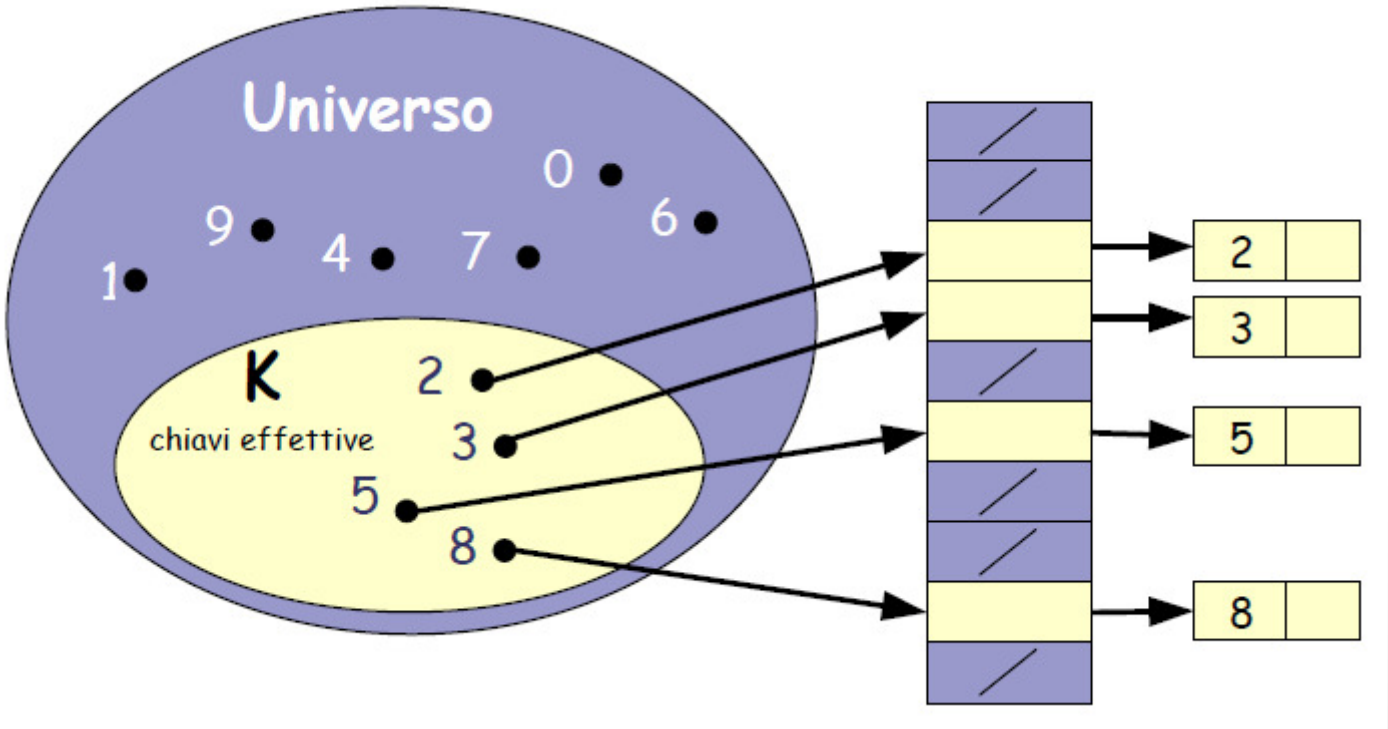

```
struct treenode {
    int key;
    struct treenode *sx;
    struct treenode *dx;
};

typedef struct treenode TREE;
typedef TREE *TREEPTR;
```

La TS ad accesso diretto

- Si suppone che l'universo delle chiavi associate agli n elementi da memorizzare siano interi in [0..m-1] con n<=m
- Assumiamo che le chiavi siano tutte distinte
- Si definisce un array V di dimensione m tale che se un certo elemento x ha chiave k, allora V[k]=x
- Le caselle che non corrispondono ad alcuna chiave avranno NULL
- Le chiavi sono usate come indici per spostarci nella struttura dati
- *Il tempo richiesto per ogni azione è costante*
- *Se m è molto grande può diventare impraticabile!!!*
- *Può comportare un enorme spreco di memoria!!!*

La TS ad accesso diretto



Fattore di carico

- Misuriamo il grado di riempimento di una tabella introducendo il fattore di carico:
  $\alpha = N/M$
  dove M è la dimensione della tabella e N è il numero di chiavi effettivamente utilizzate
- Esempio: tabella con nomi di studenti indicizzati da numeri di matricola a 6 cifre
  $N = 100$, $M = 10^6$, $\alpha = 0,0001 = 0,01\%$

E se le chiavi non sono interi?

La TS come *tabella hash*

- Idea: Dimensionare la tabella in base al numero di elementi attesi ed utilizzare una speciale funzione (funzione hash) per indicizzare la tabella
- La maggior parte dei compilatori usa questo tipo di organizzazione: in particolare la TS è una "hash table with chaining".
- Questo tipo di organizzazione consente di implementare le tre operazioni fondamentali con un numero molto contenuto di accessi indipendente dalla dimensione della tabella.

La TS come *tabella hash*

- Una ricerca basata su hashing è completamente diversa da quella basata su confronti:

  invece che spostarci nella struttura dati in funzione dell'esito dei confronti fra chiavi, cerchiamo di accedere agli elementi della tabella in modo diretto tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella.

- *E' necessario determinare la **funzione di hash** che trasforma una chiave in un indirizzo della tabella.*

La funzione *hash*

- Un indirizzo hash è un possibile valore dell'indice di un vettore di elementi destinati ad ospitare gli elementi della tabella.

- La corrispondenza ***nome_id – posizione*** viene calcolata da un'apposita funzione detta funzione hash:

$$h : S \rightarrow \{ 0 .. M-1\}$$

dove S è l'insieme delle chiavi

- Una funzione hash ideale è facilmente calcolabile e approssima una funzione casuale: per ogni valore di input, i possibili valori di output dovrebbero essere in qualche senso equiprobabili.

La TS come *tabella hash*

- L'idea è quella di implementare la TS come un array così definito:

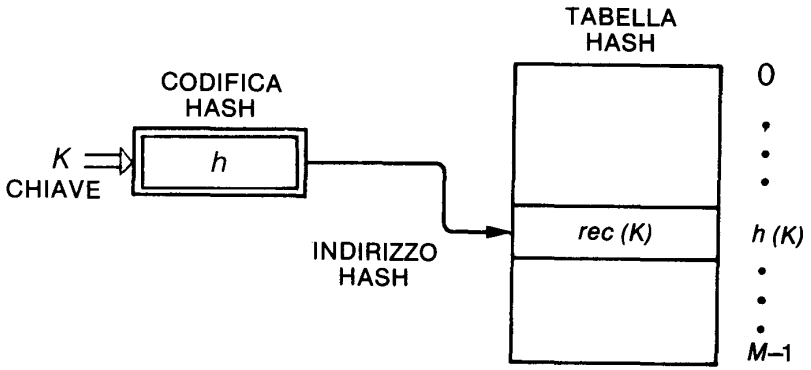
```
#define N .. ; //lunghezza massima della tabella
     struct Item{
            int key;
            . . .
            };
     typedef struct Item ITEM;
     typedef ITEM *HASH_TAB;

     void HT_init(HASH_TAB *st;int M);
     {int i;
     *st=(HASH_TAB)malloc(M*sizeof(ITEM));
     for (i=0;i<M;i++) *st[i]=NullItem;
     }
```

- **Oppure**

```
struct ITEM *hashtab[N];
```

La TS come *tabella hash*



Con l'hashing, un elemento con chiave k viene memorizzato nella cella h(k).

Pro: riduciamo lo spazio necessario per memorizzare la tabella

Contra: perdiamo la corrispondenza tra chiavi e posizioni in tabella

Le tabelle hash possono soffrire del fenomeno delle collisioni

La TS come *tabella hash*

Operazioni:

```
insert(ITEM e; key k);
    st[h(k)]=e

delete(key k);
   st[h(k)]=NULL;

ITEM search(key k);
 return st[h(k)]
```

La TS come *tabella hash: collisioni*

- La situazione ideale si ha quando la funzione h è iniettiva:

$$\text{per ogni } K, K' \in S: \quad K \neq K' \Rightarrow h(K) \neq h(K')$$

- In questo caso ogni chiave ha il suo indirizzo hash distinto da quello di tutte le altre chiavi: ogni elemento della tabella può essere raggiunto con un solo accesso (funzione hash perfetta).
- Naturalmente si richiede che M sia non minore del numero delle possibili chiavi, cosa che in genere non accade.
- Ciò potrebbe non sempre essere possibile, per cui si accetta il fatto che due o più chiavi “collidano” cioè abbiano lo stesso indirizzo hash.

La TS come *tabella hash: collisioni*

- Le chiavi possibili sono in genere moltissime e non note a priori (per esempio le variabili che un utente inserirà in un programma, o i cognomi dei clienti di una ditta), quindi si ha |Insieme delle chiavi| >> M.

- Dunque è inevitabile che nell'applicazione si presentino due o più chiavi K1, K2 ... con lo stesso indirizzo hash, cioè h(K1)=h(K2)=...

  Nasce quindi un problema di *collisioni*: solo una delle chiavi (tipicamente la prima che si è presentata, diciamo K1) potrà essere allocata in A[h(K1)], e le altre saranno poste altrove.

- Dobbiamo quindi affrontare tre problemi:

  1) come si sceglie la dimensione M;

  2) come si calcola la funzione h;

  3) come si risolvono le collisioni.

Scelta di M e fattore di carico

- Se N è il numero variabile degli elementi e M è la dimensione fissata del vettore, si definisce fattore di carico α=N/M
- Spesso, M si sceglie come potenza di due o come numero primo. Ciò dipende dalla funzione hash.

Esempi di funzioni hash (hash table di taglia M)

- Se le chiavi sono numeri reali casuali in [0,1), la funzione hash è h(k)= $\lfloor kM \rfloor$
- Se le chiavi sono numeri reali casuali in [s,t), allora h(k)= $\lfloor (k-s)/(t-s) * M \rfloor$
- Se le chiavi sono intere, h(k)=k%M (hashing modulare). Se $M=10^p$, h(k) rappresenta le p cifre meno significative. Una buona scelta sarebbe M un numero primo.
- Se le chiavi sono stringhe, allora la funzione hash può essere:

```
int hash(char *s) {
/* calcola il valore HASH di s;*/
int h=0,a=127;
for(;*s!='\0';s++)
   h=(a*h+*s)%HASHSIZE; /*HASHSIZE deve essere un primo*/
return h;}
```

Ogni carattere contribuisce in modo unico al risultato finale. Per avere una buona (cioè uniforme) distribuzione degli oggetti nella tabella (e avere collisioni distribuite) è conveniente utilizzare dei numeri primi per la base delle potenze e per la dimensione delle tabelle.

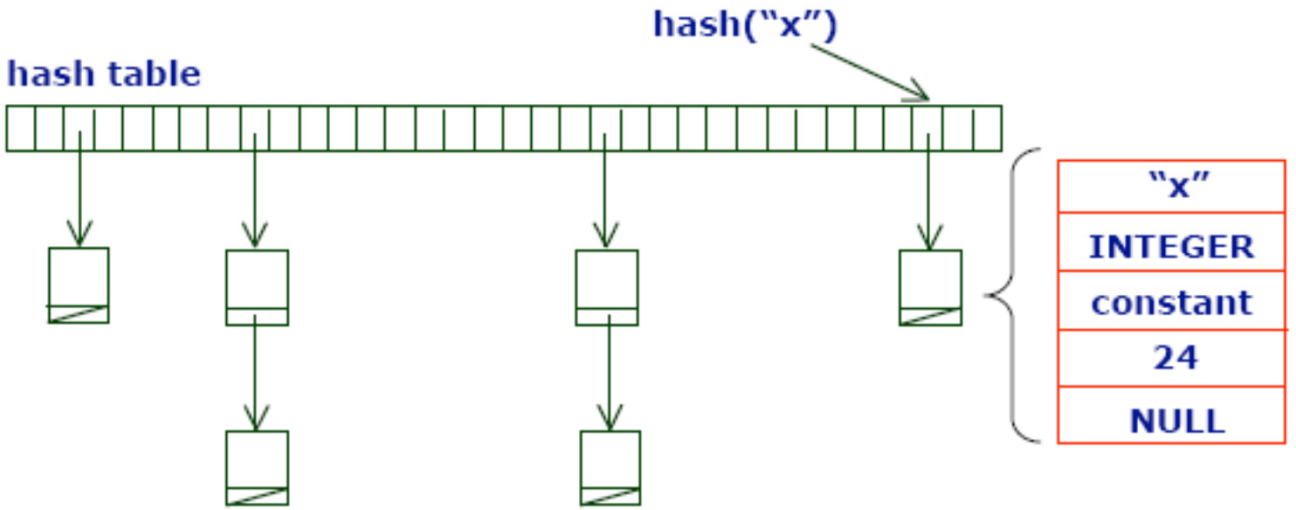
Metodi classici di risoluzione delle collisioni

- Liste di collisione(hash table with chaining): gli elementi collidenti sono contenuti in liste esterne alla tabella; T[i] punta alla lista di elementi

  tali che h(k) = i
- Indirizzamento aperto: tutti gli elementi sono contenuti nella tabella; se una cella è occupata, se ne cerca un’altra libera

Risoluzione delle collisioni: "hash table with chaining"

- La TS sarà un array di puntatori ciascuno dei quali punta ad una lista concatenata (chain o bucket) in cui ogni nodo memorizza le informazioni relative ad ogni identificatore il cui indirizzo hash collide.



Hash table with chaining

- La probabilità che il numero di chiavi in ciascuna lista sia pari al fattore di carico è molto vicina a 1.
- E' un metodo molto efficiente per $\alpha>1$.
- Costo della ricerca (caso medio): nell'ipotesi di uniformità semplice della funzione hash, una ricerca richiede in media un tempo $\Theta(1+\alpha)$

Osservazioni

- Alcuni compilatori scelgono di utilizzare una sola tabella dei simboli (che contiene qualunque informazione), che è dunque abbastanza complessa.

- Altri compilatori scelgono di separare le informazioni in differenti tabelle (tabella per i tipi, per la portata delle dichiarazioni, etc.)

Comportamento e implementazione della ST

- Il comportamento di una symbol table dipende pesantemente dalle proprietà delle dichiarazioni del linguaggio che deve essere tradotto;
- Il funzionamento delle operazioni di insert e delete, quando devono essere chiamate, quali attributi inserire, variano da linguaggio a linguaggio.

Dichiarazioni

Esistono 4 principali tipi di dichiarazioni in un linguaggio di programmazione:

Dichiarazioni di costanti: `const int A=2;`

(in C esiste anche #define che però viene gestito dal preprocessore)

Associano valori a nomi; In Pascal si richiede che siano statiche, ovvero computabili in fase di compilazione; il compilatore sostituisce il nome col valore. Quando sono dinamiche (per es. C) si può assegnare valore solo una volta.

Dichiarazioni di tipi:

In Pascal `type ciccio=array[1..6] of char;`

In C ci sono dichiarazioni struct e union che definiscono delle vere e proprie strutture di tipo:

```
struct nodo{
char *dname;
struct nodo * next;}
```

Oppure typedef: `typedef struct nodo *ptrnodo;`

Associano nomi a tipi nuovi o alias di tipi.

Dichiarazioni

Dichiarazioni di variabili:
In Pascal `var c:integer;`
In C `int c;`
Associano nomi a tipi. Comunemente fanno uso di attributi relativi alla visibilità (scope).

Dichiarazioni di procedure/funzioni:
Sono dichiarazioni di costanti di tipo procedure o funzioni;

Può essere usata una sola TS per tutti i tipi di dichiarazione, diverse TS per ogni tipo di dichiarazione ma esistono linguaggi (come il C e il Pascal) che hanno differenti TS per diverse regioni del programma (procedure o funzioni).

Attributi di visibilità

Nel caso di variabili, un attributo implicito è la visibilità di una dichiarazione, o la regione del programma dove la dichiarazione si applica.
Un attributo collegato alla visibilità è la locazione di memoria per la variabile dichiarata.

Un altro è il tempo di vita della variabile: in C tutte le variabili esterne a funzioni sono allocate staticamente (prioritaria all'inizio dell'esecuzione). Il tempo di vita è quello del programma.

Nel caso di variabili dichiarate dentro una funzione, il tempo di vita dura quanto la funzione.

Visibilità e struttura a blocchi

- Nei linguaggi senza dichiarazioni annidate (Fortran) la TS così come l'abbiamo descritta è più che sufficiente.

- Molti linguaggi di programmazione moderni sono strutturati a blocchi. Un blocco è un costrutto che può contenere dichiarazioni. E' possibile quindi avere anche dichiarazioni annidate. Pertanto bisogna gestire con la TS anche il fatto che una dichiarazione ha una sua portata (il tempo di vita).

Regole di visibilità e struttura a blocchi

- **Dichiarazione prima dell'uso:** una variabile deve essere dichiarata prima di essere usata. Ciò consente alla symbol table di essere costruita in fase lessicale e di generare errori durante la fase di parsing.

- **Regola del blocco annidato più vicino:** date diverse dichiarazioni con lo stesso nome la dichiarazione valida è quella del blocco più interno.

Risoluzione del problema

- Esistono due approcci per tener conto delle regole di visibilità nella TS:
  - avere una TS per ogni ambiente o blocco;
  - avere una TS globale.

Una TS per ogni ambiente

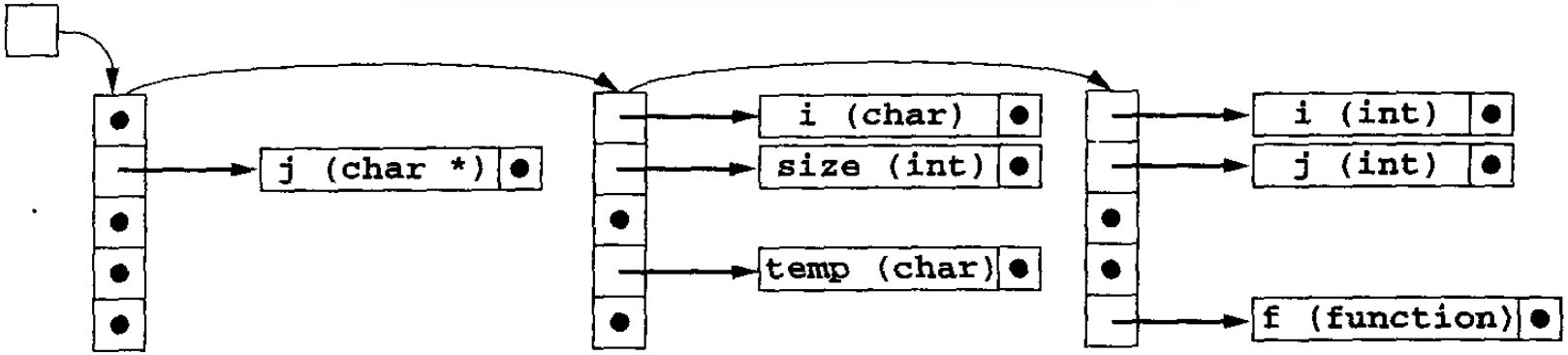
- E’ probabilmente più semplice pensare ad una TS per ogni ambiente.
- Infatti quando la compilazione di un determinato blocco è finita, la TS contenente l’ambiente locale, può essere cancellata.

La pila degli ambienti

- Ciò suggerisce di utilizzare una “pila degli ambienti”: ogni volta che si definisce un nuovo ambiente una nuova TS è aggiunta nella pila.
- Quando il compilatore termina con tale ambiente, la TS relativa può essere eliminata.
- L’uso della pila assicura che vengano rispettate le regole di visibilità.

```
int i,j;

int f(int size)
{ char i, temp;
   ...
   { char * j;
      ...
   }
}
```



Problemi con la pila degli ambienti

- nei compilatori a più passate si potrebbe avere la necessità di rivisitare gli ambienti già cancellati;
- alcuni compilatori creano la “cross-reference table” che contiene per ogni variabile dove è definita e dove è referenziata. Tale tabella è ovviamente costruita alla fine della compilazione e alcune informazioni potrebbero non essere più disponibili. La soluzione potrebbe essere quella di salvare opportunamente gli ambienti cancellati (per es. in un file temporaneo)
- se sono presenti più TS abbiamo bisogno di più hash table normalmente tutte della stessa dimensione: può capitare, quindi, che per alcuni ambienti con pochi identificatori lo spazio di memoria inutilizzato sia veramente tanto.

Una TS globale

- In questo caso bisogna associare all'identificatore un'informazione relativa all'ambiente a cui appartiene.
- Ciò può essere fatto associando ad ogni ambiente un numero che verrà incrementato ogni volta che inizia un nuovo ambiente.
- Ogni nuovo identificatore sarà inserito all'inizio della lista in modo tale che in cima ci saranno gli identificatori dell'ultimo ambiente analizzato e ciò in accordo con le regole di visibilità.
- La cancellazione degli ambienti già analizzati non presenta particolari difficoltà, fermo restando le osservazioni fatte in precedenza.

Osservazioni sul contenuto della TS

- Per ogni elemento:
  - nome dell'identificatore
  - elenco degli attributi

Nome dell'identificatore

- In alcuni linguaggi di programmazione la lunghezza degli identificatori è limitata.
- In questo caso è possibile memorizzare il nome dell'identificatore in un vettore di caratteri di lunghezza massima pari a quella consentita per la lunghezza degli identificatori stessi.
- Lo spreco di memoria risulta irrisorio se il limite per la lunghezza degli identificatori è ragionevole (per es. 8 caratteri).
- Tale soluzione non può essere accettata per quei linguaggi che ammettono identificatori di lunghezza arbitraria.

La soluzione

- I nomi degli identificatori vengono memorizzati, opportunamente delimitati, in un unico vettore di caratteri di grandi dimensione (detto “array dei nomi”).

- Nella TS non sarà presente il nome dell’identificatore ma un puntatore all’array dei nomi.