Lezione 15
22 novembre 1999

Argomenti trattati
- Comportamento dinamico dei programmi Pascal.
- Organizzazione della memoria: stack e record di attivazione.
- Attivazione e disattivazione di sottoprogrammi.

15.1 Comportamento dinamico dei programmi Pascal

Abbiamo visto che in Pascal la visibilità delle risorse viene determinata in base al testo del programma. In altre parole, possiamo dire che essa dipende dalla *struttura statica* del programma, cioè da come sono organizzati e innestati tra loro i blocchi che lo costituiscono. Il comportamento di un programma, e in particolare l'attivazione e disattivazione di sottoprogrammi, dipendono invece dalla particolare esecuzione considerata. Ad esempio, una chiamata di procedura può essere inserita nel ramo `THEN` di un costrutto `IF` e può dunque avvenire solo in base al verificarsi di una determinata condizione.

Consideriamo il seguente programma:

```
PROGRAM p (input, output);
   VAR
      x, w: integer;

   PROCEDURE a (VAR k: integer);
   BEGIN {a}
      k := k + x
   END; {a}


   PROCEDURE b (VAR j: integer);
      VAR
         x: integer;
   BEGIN {b}
      x := 2 * j;
      a(x);
      j := j DIV 2 + x
```

```
   END; {b}

BEGIN {p}
   readln(x, w);
   IF x > w THEN
      a(w)
   ELSE
      b(w);
   writeln(x, w)
END. {p}
```

In base al valore della condizione valutata nel programma principale, si possono avere due sequenze di esecuzione, la prima in cui il programma `p` richiama la procedura `a`, la seconda in cui il programma `p` richiama la procedura `b` che a sua volta richiama la procedura `a`.

Osserviamo pertanto che, in base ai valori forniti da `input`, è possibile che una stessa procedura sia richiamata da blocchi differenti di codice. In entrambi i casi, l'identificatore `x`, utilizzato all'interno di `a`, denota la variabile `x` dichiarata nel programma principale. Questo è dovuto al fatto che la visibilità degli identificatori dipende dalla struttura statica del programma, cioè dal modo in cui sono organizzati e innestati tra loro i vari blocchi e non da come avvengono le chiamate.

A livello astratto, possiamo immaginare ciascun sottoprogramma come una particolare macchina, dotata di una propria memoria privata. In questo modo, la chiamata di un sottoprogramma corrisponde all'attivazione della macchina corrispondente. La macchina, oltre ad accedere alla propria memoria, può accedere alla memoria di altre macchine già attivate, corrispondenti al blocco in cui il sottoprogramma è stato dichiarato e via via ai blocchi piú esterni. Nell'esempio precedente, la macchina `PROCEDURE a` può accedere, oltre alle proprie risorse, anche alle risorse della macchina `PROGRAM p`. Nella prima delle due sequenze d'esecuzione che abbiamo evidenziato, la macchina `a` viene attivata dalla macchina `p`; nella seconda, la macchina `a` viene attivata invece dalla macchina `b`. L'attivazione delle macchine, o meglio dei sottoprogrammi, dipende quindi dalla particolare esecuzione, e pertanto è *dinamica*.

Studieremo ora come, in realtà, le differenti macchine corrispondenti ai sottoprogrammi, possano essere realizzate disponendo di un'unica macchina. In altre parole, evidenzieremo come la Macchina Pascal sia in grado di gestire l'attivazione e disattivazione dei sottoprogrammi e dei relativi dati.

Ricordando che un programma è costituito da *dati* e *azioni*, osserviamo che durante l'esecuzione è necessario disporre di due informazioni fondamentali:

- L'*environment pointer* (o *puntatore all'ambiente*) che individua l'ambiente di esecuzione, cioè l'ambiente che contiene i dati locali utilizzati dal sottoprogramma in esecuzione.
- L'*instruction pointer* (o *puntatore all'istruzione*) che individua l'istruzione in corso d'esecuzione.

L'esecuzione di un programma inizia con la creazione di un ambiente in cui verranno memorizzate le variabili dichiarate nel programma principale. Si passa quindi ad eseguire il codice del programma, facendo puntare l'instruction pointer alla prima linea di codice del programma. Durante l'esecuzione, l'instruction pointer viene modificato sulla base delle strutture di controllo utilizzate (sequenza, selezione o iterazione). Viene inoltre modificato quando si abbia la chiamata o il rientro da un sottoprogramma.
In particolare, quando si ha una *chiamata di procedura*, l'attivazione della procedura avviene compiendo le seguenti operazioni:

- memorizzazione dell'instruction pointer (per permettere il rientro);
- creazione di un nuovo ambiente per i dati locali della procedura;

- scambio dei parametri;
- spostamento dell'instruction pointer all'inizio del codice della procedura.

Quando viene raggiunta la fine del codice della procedura, la disattivazione e il *rientro dalla procedura* avvengono effettuando le seguenti operazioni:

- distruzione dell'ambiente relativo alla procedura;
- spostamento dell'instruction pointer al punto memorizzato al momento della chiamata.

La creazione e la gestione in memoria degli ambienti relativi ai sottoprogrammi avviene utilizzando uno *stack* di *record di attivazione*.

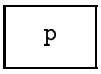
15.2 Stack e record di attivazione

Uno *stack* è un insieme di oggetti organizzati in una *pila* (si pensi ad una pila di piatti o una pila di libri): è possibile aggiungere o eliminare elementi solo in cima alla pila. Una struttura a pila è detta anche struttura LIFO (*Last In First Out*), in quanto il primo elemento che può essere eliminato è quello che è stato inserito per ultimo.

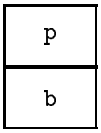
La memoria della macchina Pascal è organizzata come uno *stack di record di attivazione*, in cui ogni record di attivazione rappresenta la memoria "locale" di ciascun blocco del programma, cioè l'ambiente per i dati locali del blocco. All'inizio dell'esecuzione del programma, lo stack contiene il record di attivazione del programma principale. Quando viene chiamato un sottoprogramma, viene aggiunto in cima allo stack un record di attivazione per il sottoprogramma chiamato. Tale record viene distrutto all'uscita dal sottoprogramma (si noti che il primo sottoprogramma che può terminare è sempre l'ultimo ad essere stato attivato).

Consideriamo di nuovo il programma `p` precedente. Disegnamo l'organizzazione dello stack in record di attivazione, durante l'esecuzione del programmma. Per comodità, rappresenteremo sempre lo stack al contrario, ponendo cioè l'elemento corrispondente alla cima dello stack in basso.

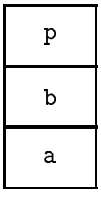
All'inizio dell'esecuzione lo stack conterrà solamente il record d'attivazione di `p`:



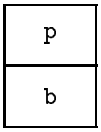
Supponiamo che vengano inseriti valori di input che rendano falsa la condizione dell'`IF`. Verrà eseguito il ramo `ELSE`, provocando l'attivazione della `PROCEDURE b`. Per memorizzarne i dati, verrà posto, in cima allo stack, un nuovo record d'attivazione:



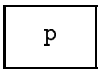
All'interno di `b` viene richiamata la `PROCEDURE a`. Attivando `a`, la struttura dello stack diventerà:



Quando viene raggiunta la fine del codice di `a`, cioè la linea `END; {a}`, il record di attivazione di `a` viene distrutto e si riprende l'esecuzione dal punto, in questo caso in `b`, in cui è avvenuta la chiamata. Pertanto la struttura dello stack torna ad essere:



Infine, al termine dell'esecuzione di `b`, anche il record di attivazione di `b` viene eliminato dallo stack:



In ogni istante, nel record di attivazione che si trova in cima allo stack (che chiameremo *record di attivazione corrente*), si trovano i dati locali del sottoprogramma in esecuzione. Le variabili visibili dal sottoprogramma, ma definite in blocchi esterni, possono essere reperite in altri record di attivazione. Si noti che, comunque, la visibilità degli identificatori dipende dalla *struttura statica* del programma, e non da come avvengono le chiamate. Pertanto, nel programma dell'esempio precedente, la variabile `x` utilizzata nella procedura `a` verrà *sempre* reperita nel record di attivazione di `p` (anche nel caso in cui `a` sia richiamata da `b`).

Struttura del record di attivazione

Abbiamo visto che il *record di attivazione* di un sottoprogramma (o programma) `s` è la struttura di memoria associata a ciascuna attivazione di `s`. Tale struttura contiene:

1. *Informazioni relative all'attivazione del sottoprogramma.*
   Queste informazioni vengono impiegate per:

   - reperire in altri record di attivazione le variabili dichiarate in blocchi esterni rispetto al blocco in esecuzione (variabili globali);
   - ricevere i risultati delle `FUNCTION` richiamate dal sottoprogramma o dal programma in esecuzione;
   - effettuare correttamente il *rientro* dai sottoprogrammi richiamati dal blocco in esecuzione.

   Non presentiamo in dettaglio tutte queste informazioni, ma ci limiteremo, negli esempi successivi, ad evidenziarne solo alcune.

2. *Dati del sottoprogramma.*
   Nel record di attivazione trovano posto i dati dichiarati localmente al sottoprogramma, cioè:

   - le *variabili locali*, i cui valori vengono memorizzati direttamente nel record di attivazione;
   - i *parametri per valore*, i cui valori vengono memorizzati direttamente nel record di attivazione (inizializzati con i parametri attuali specificati al momento della chiamata);
   - i *parametri per riferimento*, per i quali, nel record di attivazione, viene memorizzato un riferimento al parametro attuale che è stato specificato al momento della chiamata, in maniera che il sottoprogramma possa operare *direttamente* sul parametro attuale.

Evoluzione dello stack durante l'esecuzione

Mostriamo ora l'evoluzione dello stack durante l'esecuzione del programma `p` precedente. In ogni record di attivazione, oltre ai dati locali, inseriamo un campo di nome `ritorno`, utile per memorizzare il punto in cui si dovrà riprendere l'esecuzione al rientro da un sottoprogramma (indirizzo di ritorno). Per comodità, riportiamo il testo del programma, marcando con commenti i due possibili punti di rientro:

```
PROGRAM p (input, output);
   VAR
      x, w: integer;

   PROCEDURE a (VAR k: integer);
   BEGIN {a}
      k := k + x
   END; {a}


   PROCEDURE b (VAR j: integer);
      VAR
         x: integer;
   BEGIN {b}
      x := 2 * j;
      a(x);
{*}   j := j DIV 2 + x
   END; {b}

BEGIN {p}
   readln(x, w);
   IF x > w THEN
      a(w)
   ELSE
      b(w);
{+}writeln(x, w)
END. {p}
```

Supponiamo di eseguire il programma `p` sui valori di `input` `10` e `4`.

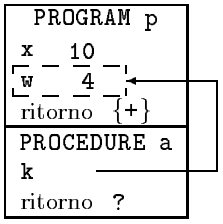
Dopo l'esecuzione dell'istruzione `readln` del programma principale, il contenuto dello stack è:

PROGRAM p
x 10
w 4
ritorno ?

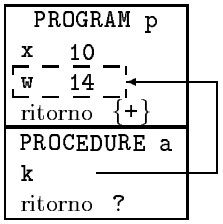
Poiché il contenuto della variabile `x` è maggiore di quello della variabile `w`, viene eseguito in ramo `THEN` dell'`IF`. Pertanto, viene chiamata la procedura `a`, passando, come parametro attuale, la variabile `w`:

- nel campo `ritorno` del record di attivazione di `p` viene memorizzato il punto in cui si dovrà riprendere l'esecuzione al rientro dalla procedura, cioè il punto in cui si trova l'istruzione successiva alla chiamata, che in questo caso è l'istruzione marcata con `{+}`:
- viene creato, in cima allo stack, un record di attivazione per `a`;
- si effettua il passaggio dei parametri: in questo caso, il parametro formale `k` di `a` è un riferimento al parametro attuale `w` utilizzato nella chiamata.

A questo punto è possibile iniziare l'esecuzione del codice di `a`. Il contenuto dello stack è:



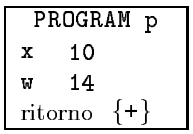
In `a` viene eseguito l'assegnamento `k := k + x`. Dalla precedente figura, osserviamo che l'identificatore `k` si riferisce alla variabile `w` del programma pricipale. L'identificatore `x` non è dichiarato localmente e quindi va reperito nell'ambiente esterno ad `a`. In base alle regole di visibilità, tale identificatore corrisponde alla variabile `x` del programma principale. Pertanto, l'effetto dell'istruzione è di assegnare `14` alla variabile `w`:



Raggiunta la fine del codice della procedura `a`, si effettua il *rientro*:

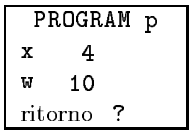
- viene distrutto il record di attivazione che si trova in cima allo stack, cioè quello di `a`;
- si riprende l'esecuzione dal punto di ritorno, memorizzato nel record di attivazione di `p`, cioè dall'istruzione marcata con `{+}`.

Al rientro da `a`, il contenuto dello stack è dunque:

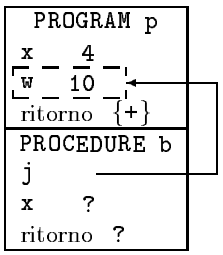


Presentiamo ora l'evoluzione dello stack, nel caso i valori forniti in input siano `4` e `10`.

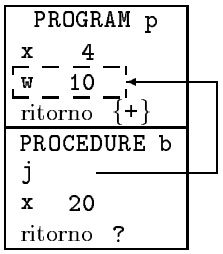
Dopo l'esecuzione dell'istruzione `readln` del programma principale, il contenuto dello stack è:



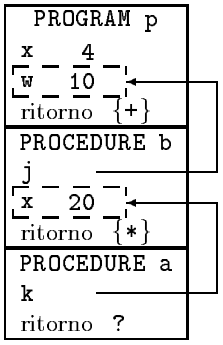
Nella successiva istruzione di selezione viene eseguito il ramo `ELSE`, effettuando la chiamata `b(w)`. Il punto di rientro (in questo caso l'istruzione marcata con `{+}`) viene memorizzato nel record di attivazione del blocco chiamante, cioè `p`. Pertanto, subito dopo l'attivazione di `b`, prima di iniziarne l'esecuzione del codice, il contenuto dello stack è:



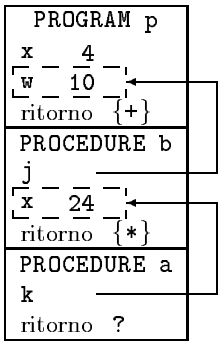
Eseguendo il successivo assegnamento, `x := 2 * j`, il contenuto dello stack diventa:



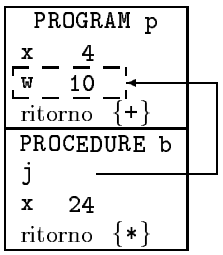
Viene quindi effettuata la chiamata `a(x)`. Il parametro formale `k` per riferimento sarà dunque utilizzato per denotare il parametro attuale fornito nella chiamata, cioè la variabile `x` dichiarata localmente alla procedura `b`. Pertanto, dopo la chiamata di `a`, il contenuto dello stack è:



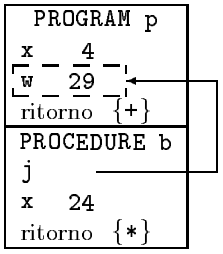
All'interno della procedura `a` viene eseguito l'assegnamento `k := k + x`, che utilizza il parametro per riferimento `k` e la variabile `x`. Dal testo del programma possiamo osservare che la variabile `x` visibile in `a` è quella dichiarata nel programma principale. Dunque, il valore da utilizzare per `x` nel calcolo dell'espressione, è quello contenuto nel record di attivazione di `p`, cioè `4`. D'altra parte, il risultato dell'assegnamento modifica `k`, che è un riferimento alla variabile `x` di `b`. Pertanto, dopo l'esecuzione dell'assegnamento, il contenuto dello stack è:



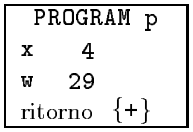
Al rientro da `a`, se ne distrugge il record di attivazione. Pertanto il contenuto dello stack è:



L'esecuzione riprende quindi dal punto segnato nel campo `ritorno` del record di attivazione restato in cima allo stack, cioè dall'istruzione marcata con {*}, l'assegnamento `j := j DIV 2 + x`, il cui effetto è:



Infine, al rientro da `b`, resta come unico record di attivazione quello di `p`:



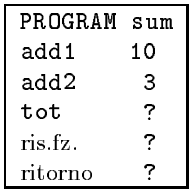
Presentiamo ora un semplice esempio in cui viene utilizzata una `FUNCTION`. Per permettere di memorizzare i valori restituiti dalle `FUNCTION` estendiamo i record di attivazione inserendo un nuovo campo (indicato con `ris.fz.` per risultato funzione). Il valore restituito da una funzione verrà scritto nel campo `ris.fz.` del record di attivazione del blocco chiamante.

```
PROGRAM sum (input, output);
   VAR
      add1, add2, tot: integer;

   FUNCTION somma (x, y: integer): integer;
   BEGIN {somma}
      somma := x + y
   END; {somma}

BEGIN {sum}
   readln(add1, add2);
   tot := somma(add1, add2) {*};
   writeln(tot)
END. {sum}
```

Simuliamo l'esecuzione del programma `sum` sui dati `10` e `3`. Come sempre, all'inizio dell'esecuzione lo stack contiene solo il record d'attivazione del programma principale. In particolare, dopo l'esecuzione dell'istruzione `readln`, il contenuto dello stack sarà:



L'istruzione successiva è un assegnamento. Il valore da assegnare è dato dal risultato della funzione `somma` sui valori di `add1` e `add2`. Dopo la chiamata e il passaggio dei parametri (che è per valore), il contenuto dello stack risulta essere:

PROGRAM	sum
add1	10
add2	3
tot	?
ris.fz.	?
ritorno	{*}
FUNCT.	**somma**
x	10
y	3
ris.fz.	?
ritorno	?

Eseguendo l'istruzione `somma := x + y`, viene calcolato il risultato della funzione, che viene posto nel campo `ris.fz.` dell'ambiente chiamante (cioè nel record di attivazione di `sum`). Il contenuto dello stack diventa quindi:

PROGRAM	sum
add1	10
add2	3
tot	?
ris.fz.	13
ritorno	{*}
FUNCT.	**somma**
x	10
y	3
ris.fz.	?
ritorno	?

Raggiunta la fine della `FUNCTION` somma, ne viene eliminato il record di attivazione dallo stack:

PROGRAM	sum
add1	10
add2	3
tot	?
ris.fz.	13
ritorno	{*}

A questo punto è possibile riprendere l'esecuzione dal punto di ritorno, completando l'assegnamento indicato con {*}, servendosi del valore memorizzato nel campo `ris.fz.`. Il contenuto dello stack diviene quindi:

PROGRAM	sum
add1	10
add2	3
tot	13
ris.fz.	13
ritorno	{*}

L'esecuzione termina con l'istruzione di scrittura.

Esercizi

1. Simulare "manualmente" l'esecuzione del programma `p` dell'esercizio 7 della Lezione 14, disegnando l'evoluzione dello stack.

2. Simulare "manualmente" l'esecuzione dei programmi `p1`, `p2` e `p3` dell'esercizio 9 della Lezione 14, disegnando l'evoluzione dello stack. Si presti particolare attenzione alle differenze tra i programmi `p2` e `p3`: in esecuzione, i record di attivazione vengono creati e distrutti nello stesso ordine; tuttavia, a causa della diversa organizzazione del testo, gli static link delle due procedure di nome `c` sono differenti. Come conseguenza, le due esecuzioni producono risultati diversi.

3. Simulare "manualmente" l'esecuzione dei programmi `p1` e `p2` dell'esercizio 11 della Lezione 14, disegnando l'evoluzione dello stack.

4. Simulare "manualmente" l'esecuzione del programma per la semplificazione di frazioni, presentato nella Lezione 10, su coppie di valori di input scelte a piacere, disegnando l'evoluzione dello stack.

5. Simulare "manualmente" l'esecuzione del seguente programma, con input `1 2 3`, disegnando l'evoluzione dello stack.

```
PROGRAM p (input, output);
   VAR
      x, y, w: integer;

   PROCEDURE a (VAR i: integer; j: integer);
   BEGIN {a}
      j := j + w;
      i := i * j
   END; {a}

   PROCEDURE b (VAR u, v, z: integer);
      VAR
         w: integer;

      PROCEDURE c;
      BEGIN {c}
         a(z, w - 1);
         w := w * 2
      END; {c}

   BEGIN {b}
      IF u > v THEN
         a(u, v)
      ELSE
         a(v, u);
      w := z;
      c;
      z := w
   END; {b}

BEGIN {p}
   readln(x, y, w);
   a(x, y);
   b(x, y, w);
   writeln(x)
END. {p}
```

6. Si consideri la seguente intestazione di procedura:

   ```
   PROCEDURE b (x: real; VAR z: real; w: integer);
   ```

   Per ognuna delle seguenti chiamate della procedura `b`, scrivere delle dichiarazioni di variabili ed eventualmente di tipo, in modo che la chiamata risulti corretta. Se ciò non fosse possibile spiegare il motivo.

   - `b(a[1], p, succ(a[3]))`
   - `b(x / y, y, ord(x IN [x]))`
   - `b(x, a, ord(x IN y[x]))`
   - `b(a, x, ord(x IN y[x]))`

7. Per ognuno dei seguenti frammenti di codice individuare delle dichiarazioni di variabile ed eventualemente di tipo, nonché eventuali intestazioni di procedura o funzione, in modo che le istruzioni che vi appaiono risultino corrette dal punto di vista della compatibilità dei tipi. Se ciò non fosse possibile, spiegare il motivo.

   - ```
     x := y(z + 1) + 1
     ```
   - ```
     x := y[z + 1] + 1
     ```
   - ```
     x := y{z + 1} + 1
     ```
   - ```
     a := b + c;
     IF x IN a THEN x := chr('a')
     ```
   - ```
     a(x);
     b(x, y);
     c(x + y + 1)
     ```
   - ```
     a(x);
     b(x, y);
     c(x + y + [1])
     ```
   - ```
     x := alfa(x);
     x := ord(x)
     ```
   - ```
     x := chr(alfa(x))
     ```