di Sergio Polini

Le «sequenze»

Eccoci finalmente al termine del nostro programma: in questa e nella prossima puntata vedremo come implementare i comandi di QUED, esaurendo così l'argomento.
In realtà il programma ha una struttura molto semplice (leggere ed eseguire i comandi dell'utente), mentre tutta la difficoltà sta nei «dettagli»: che strutture di dati adottare, come eseguire l'analisi lessicale, sintattica e semantica dell'input. Il lavoro fatto finora, non a caso durato qualche mese, è stato necessario: ora siamo finalmente in grado di riprendere il nostro cammino «top down», di riprendere cioè il discorso dalla struttura del corpo principale del programma per arrivare alla implementazione dei singoli comandi

Passiamo in rassegna il nostro bagaglio.

Abbiamo i meccanismi per la redirezione di input e output e per l'interpretazione dei parametri della riga comando (visti nel numero di luglio/agosto dello scorso anno), un «modulo» di manipolazione di liste circolari doppie (QLIST.INC, pubblicato a settembre), nonché «modulli» di analisi lessicale (QALEX.INC, novembre) e sintattica (QPARS.INC, dicembre): dovevamo essere sicuri delle nostre tecniche prima di poter disegnare lo schema generale del programma.

Avevamo visto infatti che il metodo «top down» può essere usato come tale solo dall'esperto, da chi abbia già le idee molte chiare su come realizzare un certo programma: non può prevedere l'uso di liste circolari doppie chi non abbia familiarità con la loro manipolazione, lo stesso vale per l'uso di una «grammatica» dell'input o di una macchina a stati finiti.

I momenti di «bottom up» che ci siamo concessi, confortati da quel po' di teoria discussa lo scorso ottobre, ci hanno ormai reso «esperti» (almeno un pochino...), e possiamo quindi ripartire «dall'alto».

Il ciclo principale

Vi devo chiedere di tornare al corpo principale del programma, già pubblicato a suo tempo ma riprodotto per vostra comodità nella figura 1.

Si chiama in primo luogo una routine di inizializzazione, che prepara la redirezione dell'I/O e interpreta i parametri della riga comando.

Se qualcosa non va per il verso giusto, viene assegnato un codice d'errore alla variabile *Stato* e si torna al DOS, altrimenti parte un ciclo **repeat-until**.

Abbiamo ormai le idee ben chiare su quello che questo deve fare: leggere il comando dato dall'utente e quindi passarlo alla procedura *Q_ParseCS*; questa chiamerà ripetutamente la routine di analisi lessicale per il suo lavoro di analisi sintattica e semantica dell'input, e quindi, se va tutto bene, chiamerà *Q_Esegui* o *Q_EseguiGlob*. Molti programmi dominati dalla struttura dell'input adottano uno schema analogo.

Si esce dal ciclo quando alla variabile *Stato* viene assegnato, al ritorno da *Q_ParseCS*, il valore FINEDATI.

Questo accade quando si dà un comando di uscita incondizionata dal programma, con «Q», oppure quando si dà il comando «q» dopo aver salvato le eventuali modifiche apportate al testo in memoria.

Un errore lessicale, sintattico o semantico, oppure un errore verificatosi durante l'esecuzione di un comando, fa sì che *Stato* contenga un codice d'errore; in tal caso si chiama una procedura che avverte l'utente che qualcosa non va.

Tutto ruota intorno a *Q_ParseCS*. Occorre tuttavia fare anche un'altra considerazione.

Come avevamo visto a settembre, una lista circolare è spesso accompagnata da due variabili di tipo puntatore: una (*NodoZeroPtr*) punta al nodo «zeresimo», posto tra l'ultimo e il primo nodo della lista, un'altra (*NodoCorrPtr*) punta al nodo corrente. La prima serve solo a localizzare rapidamente l'inizio e

```
begin
  Q_Inizializza(Stato);
  if Stato <> OK then begin
    writeln(StdErr,Msg[Stato]); halt(1)
  end;
  repeat
    CopiaNCPtr := NodoCorrPtr;
    if MostraPrompt then write(StdErr,PromptStr,' ');
    Q_ReadCS(Stato);
    if Stato = OK then Q_ParseCS(Stato);
    if not(Stato in [FINEDATI,OK]) then begin
      Q_MsgErrore(Stato); NodoCorrPtr := CopiaNCPtr
    end
  until Stato = FINEDATI
end.
```

Figura 1
Il «main body» del programma QUED.PAS.

Comando	Riga corrente dopo l'esecuzione
a (append)	L'ultima riga aggiunta
c (change)	L'ultima riga aggiunta
d (delete)	La riga successiva all'ultima cancellata (*)
e (edit)	L'ultima riga del file letto dal disco
E (Edit)	L'ultima riga del file letto dal disco
f (file-name)	Invariata
h (help)	Invariata
H (Help)	Invariata
i (insert)	L'ultima riga inserita
m (move)	L'ultima riga mossa
n (number)	L'ultima riga visualizzata
p (print)	L'ultima riga visualizzata
P (Prompt)	Invariata
q (quit)	Invariata
Q (Quit)	--
r (read)	L'ultima riga del file letto da disco
s (substitute	La riga su cui viene operata la sostituzione
S (Silent)	Invariata
t (copy)	L'ultima riga copiata
w (write)	Invariata
=	Invariata
(*) Se le righe cancellate erano alla fine del testo, riga corrente diventa la nuova ultima riga.	

Figura 2 - Tabella degli effetti di diversi comandi sulla riga corrente.

la fine del testo in memoria, l'altra si incarica di ricordare «dove siamo».

Molti comandi cambiano la riga corrente, nel senso che «corrente» diventa in genere l'ultima riga su cui un comando ha agito.

Può tuttavia accadere che si verifichi un errore durante l'esecuzione di un comando; per evitare che si perda così l'orientamento (che cioè ci si trovi su una riga senza sapere come), è consigliabile salvare il puntatore alla riga corrente prima di chiamare *Q_ParseCS*, in modo che sarà facile tornare «dove eravamo» in caso di errore.

Occorrerà poi un po' d'attenzione nella scrittura delle routine di esecuzione dei vari comandi, nel senso che bisogna badare anche ad aggiornare ovunque necessario la variabile *NodoCorrPtr*. A questo scopo conviene evitare di andare «a tentoni», è cioè meglio costruirsi prima di tutto una tabella come quella di figura 2.

Lettura e scrittura di «sequenze»

Avevamo potuto collaudare le routine di analisi lessicale grazie ad un apposito programma di prova, ma non abbiamo fatto lo stesso con le routine di QPARS.INC.

Il motivo è presto detto: poiché un momento molto importante dell'analisi sintattica e semantica è rappresentato dalla verifica della correttezza degli «indirizzi» (il riferimento ad una riga inesistente è un errore di tipo semantico), abbiamo bisogno di un qualche testo in memoria per poter fare le nostre prove.

Ecco quindi che vi propongo innanzi tutto il file QFILE.INC, contenente le due procedure *Q_ReadFile* e *Q_WriteFile*.

Quest'ultima è particolarmente semplice: si apre il file con *rewrite* (e quindi lo si crea se non esiste), si scrivono una alla volta tutte le righe comprese tra quelle puntate dai parametri *RP1* e *RP2* (per default la prima e l'ultima del testo in memoria), si incrementa contemporaneamente una variabile *n*; alla fine, se la variabile boleana *MostraTotRighe* vale TRUE (viene cambiata in FALSE con il comando «S»), si mostra il numero delle righe scritte (registrato in *n*).

Per scorrere tutte le righe comprese tra *RP1* e *RP2* viene usato un ciclo **while**: si tratta di un dettaglio apparentemente insignificante, ma che può offrirci lo spunto per ricordare alcune interessanti considerazioni di Wirth.

Come esiste una corrispondenza tra gli «oggetti» di un programma e le sue strutture di dati, esiste anche una corrispondenza tra strutture di dati e istruzioni.

Abbiamo visto ad esempio che, se devo gestire un insieme ordinato di elementi dello stesso tipo e di cui già conosco la numerosità, nulla si presta meglio di un array, mentre se non posso determinare a priori il numero di quegli oggetti può convenirmi una lista concatenata; aggiungiamo ora che, da questo punto di vista, un file su disco assomiglia un po' ad una lista, nei casi in cui non è determinato a priori il numero delle righe, se è un file *Text*, o dei record.

Le liste del tipo che abbiamo visto finora (sì, ne vedremo altri tipi) e i file (in particolare quelli di tipo *Text*) sono per Wirth ambedue esempi di una struttura di dati più astratta: la *sequenza*.

Per «sequenza con tipo base T» si intende o la sequenza vuota o la concatenazione di una sequenza e di un'istanza del tipo T. Notate la differenza: un array è un insieme ordinato di un numero predeterminato di elementi tutti appartenenti ad un certo tipo; una sequenza invece cresce dinamicamente mediante successive concatenazioni.

Si richiedono per questo motivo schemi di allocazione dinamica della memoria, sia nel senso di usare sempre più RAM mediante progressivo uso di uno *heap* (con le procedure *new* o *getmem*), sia nel senso di occupare sempre più spazio sul disco.

Proprio da questo modo di vedere di Wirth deriva la differenza in Pascal tra le istruzioni **for** e **while**. In altri linguaggi (in primo luogo il C), i cicli **for** e **while** sono perfettamente equivalenti, sono solo due modi diversi di scrivere la stessa cosa; in Pascal c'è invece una netta differenza: si usa un ciclo **for** quando è noto il numero delle iterazioni, si usa un ciclo **while** quando non si può sapere per quante volte il ciclo dovrà essere eseguito; si usa un ciclo **for** per scorrere un array, si usa un ciclo **while** per scorrere una lista concatenata o un file.

Un'ultima osservazione, forse questa davvero banale: la differenza tra un ciclo **while** e un ciclo **repeat** risiede solo nel fatto che il primo potrebbe non essere eseguito nemmeno una volta (anche una sequenza vuota è una sequenza), l'altro viene eseguito almeno una volta.

Ecco perché nel corpo principale del programma abbiamo usato un ciclo **repeat**: anche la sequenza dei comandi dati dall'utente è una «sequenza» nel senso appena detto, ma la presenza o meno di comandi (la possibilità cioè della «sequenza vuota») dipende da quello che passa nella testa dell'utente, e dobbiamo lasciargli la possibilità di scegliere tra sequenza vuota (uscire subito dal programma) e non-vuota (fare prima qualcosa), proprio eseguendo il ciclo almeno una volta. Potremmo forse dire che il ciclo **repeat** è l'istruzione «corrispondente» a sequenze rappresentate da successivi input da parte dell'utente.

Ecco comunque che, essendo liste e file ambedue «sequenze», *Q_ReadFile* ha una struttura molto simile a quella di *Q_WriteFile* nonostante scorra un file invece che una lista: si apre il file con *reset* invece che con *rewrite*, è

ovviamente diversa la condizione che governa il ciclo **while**, si usa *readln* invece che *writeln*, si chiama la procedura *Q_AddRiga* (contenuta in QLIST.INC) per aggiungere ogni riga letta dal file alla lista in memoria oltre a incrementare un puntatore, si rende «corrente» l'ultima riga letta e aggiunta. Il resto è praticamente identico.

Visualizzazione del testo

Una volta aggiunte a QUED le routine di QFILE.INC, ci mancano solo le procedure *Q_Esegui* e *Q_EseguiGlob* per poter compilare il programma.

Spesso, in questi casi, si usa verificare la correttezza sintattica del programma inserendo procedure «vuote» (fatte cioè solo di un «**begin end;**») al posto di quelle mancanti: si può in tal modo verificare che il compilatore accetti quanto abbiamo finora scritto senza protestare per errori quali un punto e virgola mancante, un nome di variabile scritto male, ecc.

Possiamo tuttavia fare anche qualcosa di più interessante: cominceremo a vedere una versione «minima» del file QCOMND.INC (figura 4), alla quale aggiungeremo il mese prossimo le ultime routine.

Compilando il programma con questa versione ridotta di QCOMD.INC otteniamo un programma eseguibile, in grado di farci scorrere un file che abbiamo indicato nella riga comando (ad esempio con «qued pippo.txt»).

In tal modo potremo verificare non solo il funzionamento dei primi comandi, ma anche quello delle routine contenute in QPARS.INC.

Vediamo in primo luogo *Q_EseguiGlob*.

Questa procedura non fa altro che scorrere con un ciclo **while** (non a caso molto simile a quello visto in *Q_WriteFile...*) una sublista, verificando se vale TRUE il campo *Glob* di ogni nodo. Ricorderete che la volta scorsa abbiamo visto la procedura *Q_Marca* che «marca», rendendo vero il campo *Glob*, ogni riga su cui va eseguito un comando globale.

Ecco quindi che per tutte le righe della sublista in cui *Glob* è vero viene chiamata *Q_Esegui*. Tutto qui.

La procedura *Q_Esegui* non è molto più complicata; in pratica tutto si riduce ad una istruzione **case**.

Vi sono solo due accorgimenti da prendere.

Sappiamo che l'esplicita indicazione di una sublista prima di un comando orientato è opzionale; se la sublista manca vengono assunti dei valori di default per la prima e l'ultima riga su cui deve agire un comando.

Meglio: alcuni comandi operano con riferimento ad una sola riga anche se se ne indicano due; in questi casi, come per i comandi «append» e «insert», si considera o l'unica riga specificata o solo la seconda se l'utente ne ha indicate due; se manca qualsiasi indicazione si assumono dei valori di default.

A questi pensa *Q_SubLista* (in QPARS.INC), che per prima cosa assegna ai suoi due parametri variabile *RP1* e *RP2* gli indirizzi della prima e dell'ultima riga del testo, prima cioè di verificare, chiamando *Q_Indirizzo*, se l'utente ha esplicitamente indicato altri indirizzi.

Quei valori di default non vanno però bene per tutti i comandi; *Q_Esegui* chiama quindi subito una procedura *Q_Default* per mettere le cose a posto.

Altro accorgimento: quasi tutti i comandi (fanno eccezione quelli che ammettono l'indicazione di un nome di file, come «r» o «f») possono essere seguiti da una «n» o una «p»; l'effetto è quello di visualizzare dopo l'esecuzione la riga corrente, preceduta o meno dal suo numero di riga.

La presenza della «n» o della «p» viene verificata da *Q_ParseCS*, che chiama appunto *Q_Esegui* con i parametri booleani *nFlag* e *pFlag* (*gFlag* serve solo per il comando «substitute»): se uno di questi vale TRUE si visualizza la riga corrente dopo l'esecuzione del comando.

Per il resto, tutta *Q_Esegui* sta in una istruzione **case**; il mese prossimo la completeremo aggiungendo i «casi» mancanti, ora ci limitiamo alla esecuzione dei comandi «q», «Q», «n» e «p».

I primi due non fanno altro che assegnare il valore FINEDATI a *Stato*, preparando così l'uscita dal programma, gli altri due prima assegnano FALSE ai flag appena ricordati (per evitare di visualizzare due volte una stessa riga), poi chiamano una procedura.

Q_nPrint e *Q_Print* sono molto simili. In primo luogo si controlla che la prima riga da visualizzare non sia in

```
{ QFILE.INC }

procedure Q_ReadFile(RP: NPtr; var Stato: integer);
var
  f: text;  s: AnyStr;  n: integer;
begin
  assign(f, NomeFile); {$I-} reset(f); {$I+}
  if IOResult <> 0 then Stato := ERRIO
  else begin
    NodoCorrPtr := RP; Stato := OK; n := 0;
    while (not eof(f)) and (Stato = OK) do begin
      {$I-} readln(f,s); {$I+}
      if IOResult <> 0 then Stato := ERRIO
      else begin
        Q_AddRiga(s, NodoCorrPtr, Stato);
        if Stato = OK then n := n + 1
      end
    end;
    close(f);
    if (Stato = OK) and MostraTotRighe then begin
      writeln(StdOut,n); flush(StdOut)
    end
  end
end;

procedure Q_WriteFile(RP1, RP2: NPtr; var Stato: integer);
var
  f: text;  n: integer;
begin
  assign(f, NomeFile); {$I-} rewrite(f); {$I+}
  if IOResult <> 0 then Stato := ERRIO
  else begin
    Stato := OK; n := 0;
    while (RP1 <> RP2^.Next) and (Stato = OK) do begin
      {$I-} writeln(f, RP1^.Txt^); {$I+}
      if IOResult <> 0 then Stato := ERRIO
      else n := n + 1;
      RP1 := RP1^.Next
    end;
    close(f);
    if (Stato = OK) and MostraTotRighe then begin
      writeln(StdOut,n); flush(StdOut)
    end
  end
end;
```

Figura 3 - Il file QFILE.INC di QUED.

```
{ QCOMD.INC }

procedure Q_nPrint(RP1, RP2: NPtr; var Stato: integer);
var
  p: NPtr;  n: integer;
begin
  if RP1 = NodoZeroPtr then Stato := ERRNUM
  else begin
    p := NodoZeroPtr^.Next; n := 1;
    while p <> RP1 do begin n := n + 1; p := p^.Next end;
    while p <> RP2^.Next do begin
      writeln(n,#9,p^.Txt^);
      n := n + 1; p := p^.Next
    end;
    NodoCorrPtr := RP2; Stato := OK
  end
end;

procedure Q_Print(RP1, RP2: NPtr; var Stato: integer);
begin
  if RP1 = NodoZeroPtr then Stato := ERRNUM
  else begin
    while RP1 <> RP2^.Next do begin
      writeln(RP1^.Txt^); RP1 := RP1^.Next
    end;
    NodoCorrPtr := RP2; Stato := OK
  end
end;

procedure Q_Default(Cmd: char; var RP1, RP2: NPtr);
begin
  if NumAddr = 0 then
    case Cmd of
      APPEND, INS    : RP2 := NodoCorrPtr;
      CHANGE, DEL, NUMERA, PRINT, MOVETO, COPYTO, SUBST:
                       begin RP1 := NodoCorrPtr; RP2 := RP1 end;
      READF, NUMLIN : RP2 := NodoZeroPtr^.Prev;
      FINECMD        : begin RP1 := NodoCorrPtr^.Next; RP2 := RP1 end
    end
  else if NumAddr = 1 then RP2 := RP1
end;

procedure Q_Esegui(Cmd: char; RP1, RP2, RP3: NPtr;
                   gFlag, nFlag, pFlag, Glob: boolean; var Stato: integer);
var
  p: NPtr; n: integer;
begin
  Stato := OK;
  if not Glob then Q_Default(Cmd, RP1, RP2);
  case Cmd of
    QUITIF: if not Cambiamenti then Stato := FINEDATI
            else Stato := ERRNOSALV;
    QUIT  : Stato := FINEDATI;
    NUMERA: begin
              nFlag := FALSE; pFlag := FALSE; Q_nPrint(RP1, RP2, Stato)
            end;
    FINECMD,PRINT : begin
              nFlag := FALSE; pFlag := FALSE; Q_Print(RP1, RP2, Stato)
            end;
  end;
  if Stato = OK then
    if nFlag then Q_nPrint(NodoCorrPtr, NodoCorrPtr, Stato)
    else if pFlag then Q_Print(NodoCorrPtr, NodoCorrPtr, Stato)
end;

procedure Q_EseguiGlob(ch: char; RP1, RP2: NPtr;
                       gFlag, nFlag, pFlag: boolean; var Stato: integer);
begin
  Stato := OK;
  while (RP1 <> RP2^.Next) and (Stato = OK) do begin
    if RP1^.Glob = TRUE then begin
      RP1^.Glob := FALSE;
      Q_Esegui(ch, RP1, RP1, nil, gFlag, nFlag, pFlag, TRUE, Stato)
    end;
    RP1 := RP1^.Next
  end
end;
```

realtà la «zeresima»: questo potrebbe accadere, ad esempio, se si desse un comando «n» o «p» senza alcun testo in memoria. Poi si procede, con il solito ciclo **while**, a mostrare su video le righe indicate dall'utente o, in mancanza, la riga corrente.

Alla fine si rende corrente l'ultima riga visualizzata, obbedendo così alla tabella della figura 2.

Q_nPrint ha però qualcosa in più. Le liste sono in alcuni casi molto più vantaggiose di un array, ma non consentono l'accesso immediato ad un loro elemento: per accedere al dodicesimo nodo, ad esempio, devo far scorrere un puntatore dal primo nodo in poi per undici volte (o dal «zeresimo» in poi per dodici volte).

Questo lavoro viene in realtà svolto dalla procedura *Q_GotoRiga* di QPARS.INC, ma ora bisogna «rifarlo» per poter mostrare il numero di riga di ognuna delle righe visualizzate.

A rigore ciò non è strettamente necessario: si sarebbe potuto evitarlo complicando un po' le routine di analisi sintattica; ho preferito tuttavia mantenere le cose quanto più possibile semplici, anche perché non mi è sembrato di riscontrare alcun apprezzabile svantaggio in termini di velocità di esecuzione del programma.

Collaudo

Così siamo pronti. Se volete collaudare questa prima, ridotta, versione di QUED, avete ormai tutto quanto occorre: abbiamo già visto infatti sia il file principale (QUED.PAS) che tutti i file da includere in questo: QFILE.INC, QLIST.INC, QALEX.INC, QCOMD.INC, sia pure in versione «mini», QPARS.INC.

Ricordate di lanciare il programma indicando un file da leggere nella riga comando, e... cercate di dare solo i comandi fin qui visti!

A chi preferisse aspettare la versione completa chiedo di pazientare ancora un mese. Per l'ultima volta! MC

Figura 4 - Una versione «minima» del file QCOMD.INC, destinata ad essere completata il mese prossimo. La versione ridotta ci consente intanto di verificare il corretto funzionamento del modulo di analisi sintattica (in QPARS.INC) e di un primo gruppo di comandi.