di Sergio Polini

Analisi lessicale dell'input

Ricapitoliamo. Abbiamo visto che non esiste un unico metodo per sviluppare un programma, che occorre essere consapevoli dell'esistenza di diversi possibili approcci, che bisogna imparare ad adottare quello di volta in volta più efficace. Per QUED siamo partiti dall'analisi delle tre fasi fondamentali di ogni programma (input, elaborazione, output) per definire le caratteristiche generali dell'interazione del nostro editor con l'ambiente esterno. Aiutati da «momenti» di bottom-up (la messa a punto dei meccanismi di I/O redirection, nonché delle routine di elaborazione dei parametri della riga comando e di gestione di liste circolari doppie), abbiamo così compiuto il primo passo di una progettazione top-down; ci siamo però accorti subito dopo che, proseguendo meccanicamente in quel modo, rischiavamo di pervenire ad un programma in cui la scansione dei comandi dati dall'utente si sarebbe propagata su quasi tutte le funzioni e procedure. Ci siamo cioè resi conto che QUED è un programma in cui predomina la struttura dell'input, e quindi è un programma che bisogna sviluppare «da sinistra verso destra»

«Dall'alto verso il basso», «dal basso verso l'alto», «da sinistra verso destra», «da destra verso sinistra»: espressioni che hanno certamente un denso significato per chi si occupi teoricamente della metodologia della programmazione, ma che rischiano di ridursi a parole vuote se non accompagnate da adeguate tecniche. Già il mese scorso abbiamo subito precisato che, in pratica, procedere «da sinistra verso destra» vuol dire partire dalla definizione del lessico e della sintassi dell'input, e abbiamo presentato la grammatica dei comandi di QUED. Ora si tratta di mostrare come si possa effettivamente pervenire a programmi concreti e funzionanti partendo da concetti altrimenti un po' astratti.

Una precisazione. La tecnica che illustreremo, nonostante possa apparire piuttosto specialistica, ha in realtà un vasto campo di applicazione: può essere utilizzata per elaborare qualsiasi tipo di input, dal semplice carattere fino a «frasi» complesse quali, ad esempio, i comandi dati ad un DBIII o le formule di uno spreadsheet. Dopo QUED svilupperemo insieme un altro programma nel quale, oltre a gestire liste diverse da quelle fin qui viste, adotteremo le tecniche di analisi lessicale e sintattica dell'input per leggere e interpretare un tipo particolare di file. Ma procediamo con ordine.

Qualche definizione

Un *alfabeto* non è altro che un insieme finito di simboli, una *frase* è una sequenza finita di simboli appartenenti all'alfabeto. Una *grammatica* è un insieme finito di regole (dette *produzioni*) che definiscono i criteri secondo i quali una data sequenza di simboli rappresenta una frase sintatticamente corretta. Un *linguaggio* è l'insieme (finito o infinito) di tutte le frasi accettate da una grammatica. La volta scorsa avevamo visto una definizione di grammatica apparentemente diversa: l'insieme delle regole che consentono di *generare* le frasi di un linguaggio. In realtà possiamo anche dire che una frase appartiene ad un linguaggio se può essere generata dalla grammatica di questo.

Le produzioni di una grammatica comprendono *simboli non terminali* (su cui torneremo il mese prossimo) e *simboli terminali* o *token*; questi possono essere o singoli simboli dell'alfabeto (un carattere, una virgola, ecc.) o *lessemi*, ovvero classi di simboli: i caratteri «1», «2» e «3», se consecutivi, costituiscono un numero; il token NUMERO indica quindi tutte le sequenze di caratteri appartenenti all'insieme ['0'..'9'].

Da un punto di vista sintattico non ha importanza se un NUMERO vale 14 o 123, l'importante è che quel numero, quale che sia il suo valore, compaia solo dove una qualche produzione ammette possa esservi un NUMERO. Il compito dell'analisi lessicale è quindi quello di preparare l'analisi sintattica «riconoscendo» un token NUMERO in una sequenza di simboli; in pratica, avremo una routine di analisi sintattica che chiama ripetutamente una funzione, Q_Token in QUED, la quale scorre l'input, saltando caratteri irrilevanti come spazi e tabulazioni (se e quando sono effettivamente irrilevanti), fino a riconoscere un token; quando ciò avviene, la funzione ritorna un codice convenzionale corrispondente al token riconosciuto e, se questo è un lessema, ne conserva il valore in una variabile globale per quando servirà. Quel numero, ad esempio, potrebbe essere un numero di riga. La routine di analisi sintattica ottiene da Q_Token l'informazione che è stato riconosciuto un token NUMERO nella stringa digitata dall'utente (quella che nella grammatica di QUED abbiamo chiamato «stringa-comando») e controlla che la frase in cui è contenuto sia sintatticamente corretta. Solo a questo punto si può eseguire il comando, ad esempio visualizzare la ventiquattresima riga del testo in memoria; solo a questo punto serve sapere che quel NUMERO

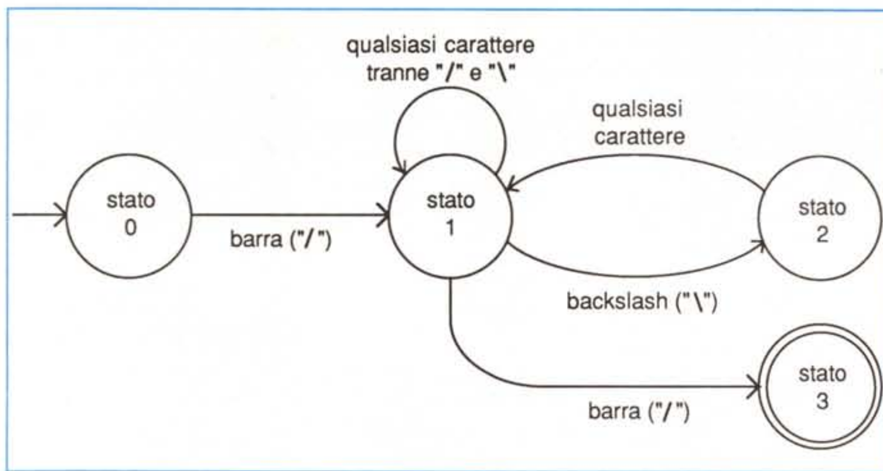

Figura 1 - Il diagramma della macchina a stati finiti che riconosce il token «/STRINGA/».

era un 24. I lessemi sono sequenze di simboli che corrispondono ad un certo *pattern* (se proprio volete un termine italiano provate con «modello»): un NUMERO è ad esempio una sequenza di cifre, ma talvolta il pattern è un po' meno banale. Una STRINGA potrebbe essere una qualsiasi sequenza di caratteri, ma in QUED appare spesso racchiusa tra due barre o tra due punti interrogativi: questo comporta che nella sequenza «/STRINGA/», ad esempio, non avrei modo di sostituire STRINGA con una stringa che contenesse una barra. Definiamo quindi un carattere di *escape*: stabiliamo che qualsiasi carattere preceduto da una «barra rovesciata» (*backslash*) va preso alla lettera, non va interpretato (una barra preceduta da un backslash è un normale carattere, non è la barra che indica la fine della stringa). Il pattern di /STRINGA/ è quindi:
una barra seguita da una qualsiasi sequenza di
a) caratteri diversi dal backslash, oppure
b) caratteri qualsiasi preceduti da un backslash, seguita da un'altra barra.

Questi sono in sostanza i lessemi di QUED, ed hanno, per nostra fortuna, una importante proprietà: possono essere rappresentati mediante *regular expressions* (mi rifiuto di tradurre con «espressioni regolari»). Qui mi fermo. Una definizione rigorosa ci porterebbe troppo lontano, ed è sufficiente ai nostri fini sapere che le regular expressions possono essere riconosciute da *macchine a stati finiti* (che vedremo subito). Per il resto, potrebbe anche darsi che ritorneremo un giorno sull'argomento per vedere come costruire un «compilatore» di regular expressions...

Macchine a stati finiti

Una macchina a stati finiti è il modello matematico di un sistema che ammette input e output discreti (cioè non continui; in parole povere: caratteri invece che un getto d'inchiostro) e che può trovarsi in uno qualsiasi di un numero finito di «stati». Vi sono uno stato iniziale, stati intermedi ed uno o più stati finali; il passaggio da uno stato ad un altro dipende solo da due cose: lo stato attuale e l'input. Non c'è bisogno di una «memoria», in quanto il percorso seguito per arrivare ad un certo stato non ha alcuna influenza su quanto accadrà dopo un certo input, se non in quanto ha fatto sì che si arrivasse allo stato attuale. Pensate ad un ascensore: non importa come sia arrivato al quarto piano; quel che conta è che se spingo il bottone «1» scenderà al primo. Le macchine a stati finiti sono usate molto spesso per l'analisi lessicale proprio per il loro funzionamento molto semplice: in pratica si tratta solo di preparare un diagramma delle transizioni da uno stato all'altro e di tradurlo poi in un ciclo del tipo

```
while S <= n do
  case S of
    0: if input = 'a' then S := 7;
    ...
    n: if input = 'f' then S := 2
  end;
```

```
{ TESTALEX.PAS }

program TestAlex;
{R+}
{$I QDICH.INC}
{$I QALEX.INC}
var
  ch: char;
begin
  assign(StdIn,'INP:'); reset(Stdin);
  repeat
    write('stringa-comando (Ctrl-Z per finire): ');
    Q_ReadCS(Stato);
    if Stato = FINEDATI then halt;
    repeat
      ch := Q_Token;
      if ch in [RIGACORR, VIRGOLA, ULTRIGA, PIU, MENO, APPEND, CHANGE,
        DEL, ERRMSG, HELP, INS, MOVETO, NUMERA, PRINT, PROMPT, QUITIF,
        QUIT, SILENT, COPYTO, NUMLIN] then
          writeln('   Token: ',ch)
      else if ch = FINECMD then writeln('   Token: FINECMD')
      else if ch in CIFRE then
        writeln('   Token: ',ch,'  Numero  : ', Numero)
      else if ch in [CERCANEXT,CERCAPREV,EDIT,EDITIF,NOMFIL,READF,WRITEF,
        GLOBSI,GLOBNO] then
          writeln('   Token: ',ch,'  Stringa1: ',Stringa1)
      else if ch = SUBST then begin
        writeln('   Token: ',ch,'  Stringa1: ',Stringa1,
                                '  Stringa2: ',Stringa2);
        ch := Q_Token;
        if ch in ['g','n','p'] then writeln('   Token: ',ch)
        else if ch = FINECMD then writeln('   Token: FINECMD')
        else writeln('   Token: NONTOK')
      end
      else if ch = NONTOK then
        writeln('   Token: NONTOK')
      else writeln('   Errore interno')
    until ch in [NONTOK,FINECMD]
  until false
end.
```

Figura 3 - Il file TESTALEX.PAS, contenente un programma di prova delle routine di QALEX.INC.

Torniamo al pattern di /STRINGA/. Nella figura 1 è rappresentata una macchina a stati finiti che altro non è che una formalizzazione della definizione discorsiva data prima. Ci troviamo nello stato iniziale quando non abbiamo riconosciuto alcun token (ad esempio perché abbiamo appena finito di riconoscerne uno, e ricominciamo quindi da capo); l'input di una barra ci porta allo stato 1, al quale torniamo ogni volta che abbiamo in input un carattere che non sia né una barra né un backslash. Un backslash ci porta allo stato 2, dal quale si torna allo stato 1 quale che sia il carattere in input (anche una barra). Si raggiunge lo stato 3 (quello finale) se si è nello stato 1 e se il carattere in input è una barra. A rigore le cose non sono sempre così semplici: nella figura 1 ogni carattere può portare da uno stato ad un solo altro, ma nulla vieta che uno stesso input possa portare dallo stato 1, ad esempio, o allo stato 2 o allo stato 3, generando così una ambiguità che può essere sciolta solo dopo che un altro input sia tale da portare ad uno stato 4 o 5. Si parla in questo caso di macchine a stati finiti *non deterministiche*, nettamente più difficili da maneggiare.

Per fortuna i pattern di QUED non richiedono che si arrivi a tanto: ogni volta che si legge un carattere si sa subito che o si è appena riconosciuto un token (ad esempio la lettera «a» per il comando «append») o che si è all'inizio del riconoscimento di un ben determinato token («1» è l'inizio di un NUMERO, «?» non può essere altro che l'inizio di «?STRINGA?», ecc.).

Implementazione

Devo fare una confessione. In principio volevo evitare di proporvi argomenti che potrebbero anche risultare ostici a qualcuno (ma sòlo ai più impressionabili); avevo quindi scritto le routine di analisi lessicale «alla buona». Risultato: notevole effervescenza di bug. Rimedio: riscrivere tutto da capo con le tecniche opportune. Tempo impiegato: un paio d'ore. Bug: scomparsi (e qui faccio gli scongiuri perché... non si sa mai!).

Per dirla in altro modo: così come non esiste un solo metodo per sviluppare programmi, non esiste neppure un solo modo (sarebbe troppo generoso parlare di tecniche, in certi casi) per scrivere una routine. L'importante è scegliere di volta in volta quello corretto, tanto più che certe tecniche basta sapere che esistono per poterle adoperare: una macchina a stati finiti deterministica si riduce a niente più che una **case** dentro un ciclo **while**. Potete controllare agevolmente che la procedura Q_Stringa «assomiglia molto» al diagramma della figura 1.

In realtà non è «identica»: ho voluto consentire all'utente di omettere la barra finale se STRINGA non è seguita da altri caratteri, ho aggiunto uno stato «di comodo» in cui i caratteri vengono man mano aggiunti alla variabile incaricata di conservare il valore di STRINGA, ho aggiunto uno stato di errore se la stringa-comando termina con un backslash, ecc. Credo comunque che sia ancora facile almeno intravedere la «macchina» incorporata nella procedura.

Non può poi creare problemi il fatto che ho estratto da Q_Token le due

```
{ QALEX.INC }

var
  CmdStr      : AnyStr;                                      { stringa-comando }
  LenCS       : byte absolute CmdStr;                        { length(CmdStr) }
  i           : integer;             { indice del carattere corrente in CmdStr }
  Esito       : integer;

procedure Q_ReadCS(var Stato: integer);
{ Legge la stringa dallo standard input }
begin
  i := 0; Numero := 0; Stringa1 := ''; Stringa2 := '';
  Stato := OK;
  if eof(StdIn) then Stato := FINEDATI
  else readln(StdIn,CmdStr);
end;

function Q_NextChar: char;
begin
  i := i + 1;
  if i > LenCS then Q_NextChar := FINECMD else Q_NextChar := CmdStr[i]
end;

procedure Q_Numero(var n, Esito: integer);
{ Ritorna il numero che si trova a partire da CmdStr[i] }
var
  j, Errore: integer;
begin
  j := i + 1; Esito := OK;
  {R-} while (j <= LenCS) and (CmdStr[j] in CIFRE) do j := j + 1; {R+}
  val(copy(CmdStr,i,j-i), n, Errore);
  if Errore = 0 then i := j - 1      { CmdStr[i] e' l'ultima cifra del numero }
  else Esito := ERRNUM
end;

procedure Q_Stringa(Dir: char; var St: AnyStr; Ricerca: boolean;
                    var Esito: integer);
{ Estrae la stringa St, preceduta da '/' o '?', da CmdStr }
const
  StrCorr: AnyStr = '';        { valore corrente della stringa per funzioni di }
var                            {   ricerca }
  j, S: integer;                { S e' lo "stato" del diagramma di transizioni }
begin
  Esito := OK; St := ''; j := i + 1; S := 0;
  while S <> 4 do
    case S of
      0: if j > LenCS then S := 4             { se manca il secondo '/' o '?' }
         else if CmdStr[j] = ESCAPE then S := 1
         else if CmdStr[j] = Dir then S := 4                { se '/../' o '?..?' }
         else S := 2;
      1: begin
           j := j + 1;
           if j > LenCS then S := 3            { se CmdStr termina con ESCAPE }
           else S := 2
         end;

      2: begin St := St + CmdStr[j]; j := j + 1; S := 0 end;
      3: begin Esito := ERRESCAPE; S := 4 end
    end;
  i := j;
  { Istruzioni seguenti non eseguite per la 2^ stringa del comando SUBST      }
  if (Esito = OK) and Ricerca then
    if St = '' then
      if StrCorr = '' then Esito := ERRSTRING   { manca un valore precedente }
      else St := StrCorr
    else StrCorr := St
end;

function Q_Token: char;
{ Estrae il "token" successivo dalla stringa-comando }
const
  TAB = #9;       SPAZIO = ' ';
  CMD_SEMPLICI : set of char =
    [RIGACORR, VIRGOLA, ULTRIGA, PIU, MENO, APPEND, CHANGE, DEL, ERRMSG,
     HELP, INS, MOVETO, NUMERA, PRINT, PROMPT, QUITIF, QUIT, SILENT,
     COPYTO, NUMLIN, FINECMD];
  UltToken    : char = FINECMD;             { memorizza l'ultimo token estratto }
var
```

procedure Q_Numero e Q_Stringa. A proposito: Q_Numero non usa una macchina a stati finiti perché... non si deve nemmeno esagerare. La procedura Val consente non solo di determinare comodamente il valore di un NUMERO, ma anche di controllare che si tratti di un valore accettabile come integer. C'è solo un punto che merita un cenno particolare: normalmente una «g» è il primo carattere del prefisso dei comandi da eseguire globalmente, ma dopo il comando di sostituzione (s/STRINGA/STRINGA/) indica che la seconda stringa va sostituita alla prima in tutta la riga. Ecco una situazione in cui avrei bisogno di una «macchina con memoria». Poco male: ho risolto il problema semplicemente memorizzando in una variabile UltToken l'ultimo token riconosciuto. Chi vedesse per la prima volta routine del genere non si scoraggi: il programma che vi proporrò dopo QUED sarà più semplice quanto ad analisi lessicale e sintattica.

```
  ch, Token    : char;
  S            : integer;                  { "stato" del diagramma di transizioni }
begin
  Esito := OK; S := 0;
  while S <> 12 do
    case S of
      0: begin
           ch := Q_NextChar; Token := ch;
           if ch in [TAB,SPAZIO] then S := 0
           else if ch in CMD_SEMPLICI then S := 12
           else if ch in CIFRE then S := 1
           else if ch in [CERCANEXT,CERCAPREV] then S := 2
           else if ch in [EDIT,EDITIF,NOMFIL,READF,WRITEF] then S := 3
           else if ch = SUBST then S := 5
           else if ch in [GLOBSI,GLOBNO] then S := 8
           else S := 11
         end;
      1: begin Q_Numero(Numero, Esito); S := 12 end;
      2: begin Q_Stringa(ch, Stringa1, TRUE, Esito); S := 12 end;
      3: begin
           ch := Q_NextChar;
           if ch = FINECMD then S := 12
           else if ch in [TAB,SPAZIO] then S := 4
           else begin Esito := ERRSPAZIO; S := 12 end
         end;
      4: begin
          ch := Q_NextChar;
          if ch in [TAB,SPAZIO] then S := 4
          else if ch = FINECMD then S := 12
          else begin
            Stringa1 := copy(CmdStr, i, 255); i := LenCS; S := 12
          end
        end;

     5: begin
          ch := Q_NextChar;
          if ch in [TAB,SPAZIO] then S := 5
          else S := 6
        end;
     6: begin
          if ch <> CERCANEXT then Esito := ERRSINT
          else Q_Stringa(ch, Stringa1, TRUE, Esito);
          if Esito = OK then begin i := i - 1; S := 7 end
          else S := 12
        end;
     7: begin
          ch := Q_NextChar;
          if ch <> CERCANEXT then Esito := ERRCMD
          else Q_Stringa(ch, Stringa2, FALSE, Esito);
          S := 12
        end;
     8: if UltToken = SUBST then S := 12  { dopo 's/../../', 'g' e' un flag }
        else S := 9;
     9: begin
          ch := Q_NextChar;
          if ch in [TAB,SPAZIO] then S := 9 else S := 10
        end;
    10: begin
          if ch <> CERCANEXT then Esito := ERRSINT
          else Q_Stringa(ch, Stringa1, TRUE, Esito);
          S := 12
        end;
    11: begin Esito := ERRCMD; S := 12 end
   end;
 if Esito <> OK then Token := NONTOK;
 UltToken := Token;
 Q_Token := Token;
end;

function LexError: integer;
{ Chiamata da Q_ParseCS per conoscere il tipo di errore che ha portato }
{ alla mancata individuazione di un token corretto }
begin
  LexError := Esito
end;
```

Figura 2 - Il file QALEX.INC, contenente il modulo di analisi lessicale di QUED.

Incremental testing

E così abbiamo visto quale tecnica si può usare per scrivere un «modulo» di analisi lessicale dell'input. Parlo di «modulo» perché il resto del programma può tranquillamente disinteressarsi di quello che fanno Q_Token e le altre procedure: basta che Q_Token ritorni il token di volta in volta riconosciuto assegnando eventualmente gli opportuni valori ad alcune variabili globali. Gli unici altri contatti sono rappresentati dalla procedura Q_ReadCS, che legge la stringa-comando (è nel modulo perché ogni volta che ne viene letta una nuova si inizializza a zero la variabile «i», indice del carattere corrente della stringa-comando), e della funzione LexError, il cui scopo è quello di «nascondere» la variabile Esito.

Questa modularità presenta notevoli vantaggi. Non solo è possibile riscrivere completamente le routine di analisi lessicale lasciando inalterato il resto (vi ho prima raccontato che mi è capitato di doverlo fare), ma abbiamo anche maggiori probabilità di arrivare ad un programma corretto.

Testare un editor è un'impresa estremamente ardua, in quanto sono pressoché infinite le diverse situazioni che possono capitare; se invece riusciamo a scomporre il programma in moduli diventa tutto più facile. Insieme a QALEX.INC vi propongo quindi TESTALEX.PAS (anch'esso disponibile su MC-Link col nome TESTALEX.ARC): un breve programma di prova delle routine di analisi lessicale (oltre a QALEX.INC viene «incluso» anche QDICH.INC, pubblicato il mese scorso).

Se gli date «pippo» vi dirà che le prime quattro lettere sono token (per i comandi «print» e «insert»), ma che non riconosce, giustamente, la «o». Può tuttavia essere più interessante provare con cose del tipo:

```
g/\/STRINGA\//s//\/stringa\//gp
```

Se tutto va bene, potremo più avanti preoccuparci della correttezza del programma completo sicuri che, se qualcosa non va, non è colpa dell'analisi lessicale.

Coraggio: avete un mese di tempo per controllare se la nostra macchina a stati finiti funziona. MC