di Sergio Polini

Quando in input ci sono nomi di file

La volta scorsa abbiamo aggiunto un'altra tessera al nostro mosaico del metodo, introducendo quasi di soppiatto gli "structure chart" di Yourdon e Constantine. Ora torneremo sui problemi dell'analisi lessicale dell'input, già trattata a novembre, per riprendere alcuni aspetti allora solo accennati. Ne approfitteremo anche per spendere qualche parola sulla materia prima di GREP, il terzo programma di utilità di derivazione Unix fornito insieme alle ultime versioni dei compilatori Borland

Riprendiamo le definizioni date a novembre, con qualche integrazione.

Abbiamo bisogno in primo luogo di un concetto elementare, in termini del quale definire tutti gli altri: il *simbolo*. Nel nostro caso potremo pensare al set di caratteri ASCII, ma si potrebbe pensare anche a segnali elettrici o nervosi (le macchine a stati finiti, su cui torneremo tra un attimo, furono usate in origine per costruire modelli delle reti di neuroni).

Una *stringa* (o *parola*) è una sequenza finita di simboli, la sua *lunghezza* è il numero dei simboli che la compongono. Fa spesso comodo poter parlare di una stringa vuota, che indicheremo con una "**e**" in neretto, con lunghezza pari a zero.

Un *alfabeto* è un insieme finito di simboli, un *linguaggio formale* è un insieme, finito o infinito, di stringhe di simboli di un alfabeto.

Per ogni linguaggio vi è il problema di definire e riconoscere le stringhe che effettivamente gli appartengono. Pensiamo a due linguaggi L1 e L2, composti di stringhe di simboli tratti, rispettivamente, da A1, alfabeto di soli caratteri numerici, e da A2, un normale alfabeto a..z. È facile vedere che "abcd" appartiene a L2 ma non a L1, "1234" appartiene a L1 ma non a L2, "a1b2" non appartiene né all'uno né all'altro. Naturalmente le cose non sono sempre così banali.

Regular expressions

Sappiamo già (ne abbiamo discusso a dicembre) che un linguaggio può essere visto come un insieme di frasi, a loro volta definibili come sequenze di simboli. Se però dobbiamo considerare la struttura delle frasi sconfiniamo nell'analisi sintattica, su cui torneremo il mese prossimo; inoltre non tutti i linguaggi hanno frasi. Alcuni linguaggi, in particolare, possono essere definiti mediante una notazione molto semplice, che consente agevolmente di esprimere in forma sintetica tutte le relative stringhe.

Dato un qualsiasi alfabeto, indicheremo con la "**e**" in neretto la stringa vuota, con le altre lettere minuscole i suoi simboli.

Il linguaggio più semplice è indubbiamente L(**e**), che comprende la sola stringa nulla. Subito dopo viene L(**a**), che comprende solo "**a**".

Introduciamo ora alcuni operatori:

1) "¦", che sta per "oppure";
2) concatenazione (un simbolo dopo l'altro);
3) "*", che sta per "il simbolo precedente ripetuto zero o piu' volte".

A questi possiamo aggiungere, per comodita':

4) "+", che sta per "il simbolo precedente ripetuto una o piu' volte";
5) "?", che sta per "il simbolo precedente ripetuto zero o una volta";
6) "{m,n}", che sta per "il simbolo precedente ripetuto da m a n volte".

Parlo di comodità perché, ad esempio, *x+* equivale a *xx**, *x?* equivale a ***x¦e*** ecc.

Abbiamo così quello che ci serve per costruire "espressioni regolari", le quali denotano linguaggi chiamati "insiemi regolari". Qualche esempio. Con l'espressione *(a¦b)(a¦b)* indichiamo un linguaggio le cui stringhe hanno tutte due simboli, dei quali sia il primo che il secondo possono essere una "a" o una "b", quindi: "aa", "ab", "ba" e "bb". Con *(0¦1)*00(0¦1)** indichiamo tutte le stringhe di 0 e 1 in cui vi siano almeno due zeri consecutivi, così come con 10+ tutti i multipli di 10.

Quando si ha a che fare con espressioni in cui compaiono molti simboli si può usare un'abbreviazione: invece di *a¦b¦c¦d...* e cosi' via sino a z, si fa prima con *[a-z]*. Si possono inoltre attribuire dei nomi alle espressioni, chiamando ad esempio **lettera** l'espressione *[A-Za-z]*, in quanto non si tratta di altro che di una

qualsiasi lettera maiuscola o minuscola. Analogamente **cifra** sarà l'espressione [*0-9*]. Il programma GREP lavora appunto su espressioni come quelle che stiamo descrivendo, anche se accetta una sintassi un po' diversa (la "**g**" e la "**p**" di GREP non sono altro che i comandi global e print dell'editor di Unix a cui ci siamo ispirati per QUED, "re" sta per regular expression).

Può capitare di dover trattare espressioni composte a loro volta di altre espressioni; sono utili in tali casi le cosiddette definizioni regolari: si tratta di una successione di definizioni, in ognuna delle quali si assegna un nome ad una espressione che viene scritta usando, oltre ai simboli, anche i nomi precedentemente definiti.

Se ad esempio adottiamo come alfabeto i simboli del set di caratteri ASCII, possiamo così definire un identificatore in Pascal:

lettera = [A-Za-z]
cifra = [0-9]
identificatore = lettera(lettera!cifra)*

Ovvero: una lettera seguita da zero o più lettere o cifre.

Notate che abbiamo incluso nella definizione di **lettera** anche il trattino di sottolineatura, come veniva fatto nell'appendice I del manuale del Turbo Pascal 3.0.

Ora forse siamo in grado di capire un po' meglio una cosa solo accennata a novembre: le espressioni regolari consentono di descrivere in forma sintetica i *lessemi*, ovvero particolari sequenze di simboli che, in quanto corrispondono ad una certo *pattern*, possono essere riconosciute come *token*. Ricorderete che i token sono i *simboli terminali* di una grammatica e che, se il compito dell'analisi sintattica è quello di accertare che la successione dei token obbedisca a certe regole, il compito dell'analisi lessicale è proprio quello di riconoscere i token nel flusso di caratteri in input.

Macchine di Moore

È relativamente facile riconoscere un identificatore se il suo *pattern* è indicato da una espressione regolare, in quanto le espressioni regolari godono di una interessante proprietà: il linguaggio che esse denotano può essere "accettato" da macchine a stati finiti, le quali, a loro volta, possono essere implementate con un semplice ciclo **while**.

Ricordiamo che una macchina a stati finiti è il modello di un sistema che ammette input e output discreti e che può trovarsi in uno qualsiasi di un numero finito di "stati". Si distinguono uno stato iniziale, stati intermedi e stati finali o "accettanti"; il passaggio da uno stato all'altro, come dicevamo a novembre, dipende solo da due cose: lo stato in cui si trova la macchina e l'input. Vi avevo allora proposto l'esempio di un

```
[C:\TP] mmalex
Stringa ('fine' per finire): nomefile
NOMEFILE: nomefile
Stringa ('fine' per finire): nomefile.
NOMEFILE: nomefile.
Stringa ('fine' per finire): nomefile.est
NOMEFILE: nomefile.est
Stringa ('fine' per finire): 123456789

Input : 123456789
ERRORE:          ^

Stringa ('fine' per finire): nomefile.1234

Input : nomefile.1234
ERRORE:              ^

Stringa ('fine' per finire): \subdir1\subdir2\nomefile.est
NOMEFILE: \subdir1\subdir2\nomefile.est
Stringa ('fine' per finire): \subdir1\\subdir2\nomefile.est

Input : \subdir1\\subdir2\nomefile.est
ERRORE:          ^

Stringa ('fine' per finire): \subdir1\123456789\nomefile.est

Input : \subdir1\123456789\nomefile.est
ERRORE:                   ^

Stringa ('fine' per finire):    comando arg1 arg2 arg3
SPAZIO RIGACMD: comando arg1 arg2 arg3
Stringa ('fine' per finire): target: source1 source2
NOMEFILE: target DUEPUNTI NOMEFILE: source1 NOMEFILE: source2
```

Figura 1 - Compilando MMALEX.PAS dopo aver definito "Main" (ad esempio con: tpc /DMain mmalex) si ottiene, invece di una unit, un programma MMALEX.EXE che consente di verificare il corretto funzionamento del modulo di analisi lessicale del nostro MiniMake. Nella figura un esempio di esecuzione.

```
{$IFDEF Main}
program MMAlex;
{$ELSE}
unit MMAlex;

interface
{$ENDIF}
uses Dos;
const
  SPAZIO    = ' ';
  FINERIGA  = #13;
  DUEPUNTI  = ':';
  NOMEFILE  = 'f';
  RIGACMD   = 'c';
  ERRORE    = #0;
var
  Riga    : string;
  NumRiga : word;
  Indx    : word;
  Nome    : PathStr;
  Comando : ComStr;
  FineFile: boolean;
{$IFDEF Main}
  t       : char;
{$ELSE}
procedure ReadNextRiga;
function  NextToken: char;

implementation
{$ENDIF}

const
  TAB       = #9;
  PUNTO     = '.';
  BACKSLASH = '\';
  CARNOMF   : set of char =
   [':','@','$','%','''','-','0'..'9','@','A'..'Z','^','a'..'z','{','}','~'];

procedure ReadNextRiga;
begin
  if Eof then FineFile := TRUE
  else begin
    Readln(Riga);
    Inc(NumRiga);
    Indx := 0
  end
end;

function NextChar: char;
begin
  Inc(Indx);
  if Indx > Length(Riga) then NextChar := FINERIGA
  else NextChar := Riga[Indx]
end;

function NextToken: char;
const
  USCITA   = 7;
  MAXINDX  = 255;
  UltToken : char = FINERIGA;
var
  Stato : integer;
  Len   : integer;
  c     : char;
  Token : char;
begin
  Stato := 0;
  Len   := 0;
  Nome  := '';
  while Stato <> USCITA do
    case Stato of
      0: begin
           c := NextChar;
           if c in [TAB,SPAZIO] then begin
             Len := 0;
             if (Indx > 1) and (UltToken <> BACKSLASH) then begin
               Token := SPAZIO; Stato := USCITA
             end
             else Stato := 0
           end
           else if (c = FINERIGA) or (c = DUEPUNTI) then begin
             Token := c; Stato := USCITA
           end
           else if c = BACKSLASH then Stato := 1
           else if c = PUNTO then Stato := 2
           else if c in CARNOMF then begin
             if UltToken = SPAZIO then Stato := 6
             else Stato := 4
           end
           else begin
             Token := ERRORE; Stato := USCITA
           end
         end;
      1: if Indx > Length(Riga) then begin
           if (Riga[Indx-1] <> TAB) and (Riga[Indx-1] <> SPAZIO) then begin
             Token := ERRORE; Stato := USCITA
           end
           else begin
             UltToken := BACKSLASH; ReadNextRiga; Stato := 0
           end
         end
         else begin
           Nome := Nome + c; Len := 0;
           c := NextChar;
           if c = PUNTO then Stato := 2
           else if c in CARNOMF then Stato := 4
           else begin
             Token := ERRORE; Stato := USCITA
           end
         end;

      2: if not (UltToken in [FINERIGA, SPAZIO, BACKSLASH]) then begin
           Token := ERRORE; Stato := USCITA
         end
         else begin
           Nome := Nome + c; Len := 0;
           c := NextChar;
           if c = PUNTO then Stato := 3
           else if c = BACKSLASH then Stato := 1
           else begin
             Token := ERRORE; Stato := USCITA
           end
         end;
      3: begin
           Nome := Nome + c;
           c := NextChar;
           if c <> BACKSLASH then begin
             Token := ERRORE; Stato := USCITA
           end
           else Stato := 1
         end;
      4: if (Len >= 8) or (Length(Nome) >= SizeOf(PathStr) - 1) then begin
           Token := ERRORE; Stato := USCITA
         end
         else begin
           Nome := Nome + c; Inc(Len);
           c := NextChar;
           if c in CARNOMF then Stato := 4
           else if c = PUNTO then begin
             Len := 0; Stato := 5
           end
           else if c = BACKSLASH then Stato := 1
           else begin
             Dec(Indx); Token := NOMEFILE; Stato := USCITA
           end
         end;
      5: if (Len >= 4) or (Length(Nome) >= SizeOf(PathStr) - 1) then begin
           Token := ERRORE; Stato := USCITA
         end
         else begin
           Nome := Nome + c; Inc(Len);
           c := NextChar;
           if c in CARNOMF then Stato := 5
           else if Length(Nome) < SizeOf(PathStr) then begin
             Dec(Indx); Token := NOMEFILE; Stato := USCITA
           end
           else begin
             Token := ERRORE; Stato := USCITA
           end
         end;
      6: if (Length(Riga) - Indx + 1) >= SizeOf(ComStr) then begin
           Token := ERRORE; Stato := USCITA
         end
         else begin
           Comando := Copy(Riga, Indx, MAXINDX);
           Indx := MAXINDX;
           Token := RIGACMD; Stato := USCITA
         end
    end;
    UltToken  := Token;
    NextToken := Token
end;

          begin
            NumRiga  := 0;
            FineFile := FALSE;
          {$IFDEF Main}
            repeat
              Write('Stringa (''fine'' per finire): ');
              ReadNextRiga;
              repeat
                t := NextToken;
                case t of
                  ERRORE  : begin
                              Writeln;
                              Writeln('Input : ',Riga);
                              Writeln('ERRORE: ','^':Indx);
                              Writeln
                            end;
                  FINERIGA: Writeln;
                  SPAZIO  : Write('SPAZIO ');
                  DUEPUNTI: Write('DUEPUNTI ');
                  NOMEFILE: Write('NOMEFILE: ',Nome,' ');
                  RIGACMD : Write('RIGACMD: ',Comando)
                end
              until (t = ERRORE) or (t = FINERIGA)
            until Riga = 'fine'
          {$ENDIF}
          end.
```

Figura 2 - Il sorgente di MMALEX.PAS.

Errata corrige

C'era un errore nella funzione *Print* del listato di MMSIM.PAS, pubblicato il mese scorso. La riga:

```
Writeln(#9,c^.Cmd);
```

va sostituita con:

```
Writeln(#,c^.Comando,' ',c^.Argomenti);
```

Cercherò di raccontarvi la storia di questo errore quando parleremo, in una prossima puntata, della procedura *Exec*.

ascensore: per andare dal piano (stato) attuale ad un altro, basta che premete il bottone corrispondente a quest'ultimo. Ogni stato rappresenta la configurazione del sistema in quel momento, e racchiude in sé tutte le informazioni necessarie per la transizione ad un altro stato. Ciò comporta che vi sia un limitatissimo bisogno di "memoria" (all'ascensore non serve ricordare tutte le tappe percorse per giungere al piano dove si trova).

Questa faccenda della memoria è di notevole importanza, in quanto ha diretta influenza sul programma che poi dovremo scrivere. Ne vedremo altri esempi in un contesto completamente diverso quando, tra un paio di mesi, distingueremo tra procedure e processi iterativi e ricorsivi.

Vediamo ora perché gli stati finali vengono detti anche "accettanti". Per passare dallo stato iniziale ad uno stato finale attraverso alcuni stati intermedi, è necessario che i successivi input corrispondano a quelli di volta in volta previsti per la transizione da uno stato ad un altro; se per un simbolo in input non è prevista alcuna transizione, la stringa che lo comprende viene rifiutata; se invece ciò non accade, si potrà giungere ad uno stato finale. Sarà proprio un tale esito a confermarci che la stringa sottoposta al test appartiene al linguaggio denotato dalla espressione regolare equivalente alla nostra macchina, può quindi essere "accettata".

Tutto si basa su questa equivalenza tra macchine a stati finiti ed espressioni regolari, di cui vi risparmio la dimostrazione rigorosa (i curiosi possono dare un'occhiata a J.E. Hopcroft & J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979). Vi propongo solo qualche spunto: in una macchina a stati finiti si passa da uno stato all'altro in funzione dell'input; in una espressione regolare si passa da un simbolo all'altro mediante operatori per i quali è facile vedere un'analogia con la macchina: (*a* | *b*) vuol dire che, se sono nello stato i, passerò allo stato *j* se in input c'è una "a", allo stato *k* se in input c'è una "b"; *ab* (la concatenazione) comporta che se sono nello stato che ha riconosciuto la "a", passerò al successivo se e solo se in input c'è una "b"; *a** (zero o piu' "a") vuol dire che rimarrò in un certo stato fino a che in input avrò delle "a".

A rigore all'analisi lessicale non basta sapere che una stringa è "accettata": occorre che la macchina abbia tanti stati finali quanti sono i token che si possono presentare nell'input e che, quando sia giunta ad uno di questi, emetta come output il nome del token riconosciuto. Macchine che hanno un output associato con gli stati si chiamano "macchine di Moore", e l'alfabeto da cui è tratto l'output nel nostro caso è rappresentato dalle costanti dichiarate nella *interface* della unit MMALEX (figura 2).

QALEX.INC e MMALEX.PAS

I nomi di file sono stringhe di un alfabeto che comprende solo alcuni dei simboli ASCII (il set CARNOMF nel listato di MMALEX.PAS); se chiamiamo tali simboli genericamente **caratteri**, ogni nome di file è una parola di un linguaggio denotato dalla seguente espressione:

nomefile = **carattere**{1,8}(.**carattere**{0,3})?

Ovvero: da 1 a 8 caratteri, opzionalmente seguiti da un punto a sua volta seguito da un numero di caratteri variabile da 0 a 3 (vi lascio la definizione del **pathname**, cioè del nome del file preceduto dalla indicazione della subdirectory in cui si trova).

La necessità di *contare* il numero dei caratteri rappresenta la prima differenza tra il modulo di analisi lessicale del MiniMake e quello di QUED. Un'altra riguarda gli spazi. Spesso nell'analisi lessicale spazi e tabulazioni (non poche volte anche i caratteri di fine riga) vengono considerati solo *separatori*, hanno importanza solo in quanto segnalano la fine di un *token*. Nella sintassi del MiniMake, invece, vedremo che le righe in cui vengono indicati i comandi devono iniziare con almeno uno spazio o un tab, costringendoci ad attribuire un ruolo diverso a tali caratteri secondo la loro posizione.

Ancora un'altra differenza con il *backslash*: usato in QUED come carattere di *escape*, viene usato nel MiniMake per congiungere righe successive. Se una lista di source non sta tutta in una riga, si può proseguire su righe successive terminando tutte, tranne l'ultima, con una barra rovesciata.

Infine, viene usata una variabile *Indx* per tenere traccia del carattere su cui è scattata una condizione di errore, in modo da poter riproporre all'utente la riga incriminata con una chiara indicazione del punto dolente (v. figura 1).

Ne segue un codice più semplice di quello di QUED quanto a numero di stati, ma più articolato nel dettaglio. Ne segue anche che potrete fare riferimento congiuntamente ai sorgenti di QUED e di MiniMake per trarne lo spunto per affrontare un ampio numero di situazioni. Non vi dovrebbe essere difficile, ad esempio, "sfrondare" MMALEX.PAS in modo da ricavarne un modulo per l'input controllato di nomi di file. Non abbiamo certo esaurito l'argomento: l'analisi lessicale di un sorgente FORTRAN, ad esempio, è notevolmente più complessa di quanto abbiamo visto su queste pagine, ma in fondo non capita tutti i giorni di dover scrivere un compilatore FORTRAN...

Ho taciuto un'altra differenza rispetto a QUED, in quanto originata esclusivamente dalla diversa versione del compilatore che ho usato. Poiché QUED era stato scritto con il Turbo Pascal 3.0, il test del modulo di analisi lessicale veniva condotto mediante un distinto programma (TESTALEX.PAS). Ora abbiamo le versioni 4.0 e 5.0 e la compilazione condizionale, di cui abbiamo approfittato anche la volta scorsa: basta accedere al menu *Options/Compiler/Conditional defines* dell'ambiente integrato, o usare l'opzione /D del compilatore TPC.EXE, per definire "Main"; ne seguirà la compilazione delle sezioni di codice comprese tra *$IFDEF Main* e *$ELSE* o *$ENDIF*, secondo i casi, e quindi la creazione di un programma di prova dal nome MMALEX.EXE (v. figura 1).