di Sergio Polini

# Sintassi e semantica dell'input

*Soggetti, predicati, complementi vari, ecc. Siamo ben abituati a vedere una «struttura sintattica» dietro ogni frase del linguaggio parlato. Sappiamo ad esempio che possiamo qualificare un soggetto accompagnandolo con un attributo, o specificare un complemento oggetto con un complemento indiretto. Quando ascoltiamo una frase non perdiamo tempo ad analizzarla, eppure se la capiamo è proprio perché riconosciamo le parole dietro i suoni, e dietro le parole una struttura sintattica; solo dopo di ciò possiamo capirne anche il significato. «Pietro saluta Paolo» non ha lo stesso significato di «Paolo saluta Pietro» proprio perché è diversa la «categoria sintattica» di Paolo e Pietro nelle due frasi.*
*Qualcosa di simile accade anche con i linguaggi formali, anche se, trovandoci in un mondo un po' diverso da quello del linguaggio naturale, dobbiamo fare qualche sforzo in più per attribuire un senso preciso e concreto a concetti per altro verso ben familiari*

La volta scorsa avevamo rimandato la discussione dei «simboli non-terminali», in quanto avevamo preferito concentrarci sui «simboli terminali» riconosciuti durante l'analisi lessicale. Un primo aiuto per capire di che si tratta ci viene da un sinonimo: i simboli non-terminali vengono anche chiamati «categorie sintattiche».

Possiamo dire in breve che ogni «frase» nasconde una struttura, e che ogni «parola» di una frase acquista un suo significato proprio in virtù della sua collocazione in questa struttura. I token sono le parole di linguaggi come quello dei comandi di QUED, i simboli non-terminali ci dicono quale successione di token, quale «struttura» di token può essere accettata come sintatticamente corretta, preparando così il terreno alla successiva interpretazione semantica.

## Dall'italiano... al latino

Consideriamo la grammatica molto semplice della figura 1, tanto semplice che può generare una sola frase: «Pietro saluta Paolo». Supponiamo di dover tradurre il tutto in latino.

«Frase» è il simbolo non-terminale da cui partiamo, per osservare che può essere sostituito da due altri non-terminali: «soggetto» e «gruppo-del-predicato». L'analisi della prima categoria sintattica è molto breve: «soggetto» si può trasformare solo nel terminale «Pietro». Il «gruppo-del-predicato» può invece essere ulteriormente scomposto nei suoi elementi «predicato» e «complemento-oggetto», i quali peraltro ci conducono poi subito ai terminali «saluta» e «Paolo».

Abbiamo così completato l'analisi sintattica della nostra frase, che possiamo rappresentare con un «albero sintattico» come quello della figura 2. La figura aiuta forse meglio a capire perché si parla di simboli «terminali» e «non-terminali»: questi ultimi corrispondono ai nodi interni dell'albero, quelli alle sue «foglie» (avremo modo di parlare più diffusamente di alberi tra qualche mese).

Non possiamo comunque limitarci ai risultati sin qui ottenuti. I verbi vanno coniugati, in latino i nomi vanno declinati, ecc. In generale, quando si deve tradurre da una lingua ad un'altra occorre considerare, accanto alla categoria sintattica delle varie parole e dei diversi costrutti, anche tutta una serie di «attributi» (da non confondere con gli attributi come categoria sintattica) quali genere, numero, caso, e così via; alcuni attributi sono propri della singola parola («Pietro» è singolare e maschile... per il solo fatto di essere Pietro), altri competono ad una parola solo in quanto questa deve coesistere con altre («saluta» è singolare perché «Pietro» è singolare, cioè perché il verbo deve concordare morfologicamente con il suo soggetto; «Paolo» avrà un caso «accusativo» perché è il complemento oggetto di «saluta», ecc.). Possiamo tradurre la nostra frase solo dopo aver individuato il valore corretto dei vari attributi: «Pietro» sarà nominativo, «saluta» sarà singolare, «Paolo» sarà accusativo, la frase completa sarà «Petrus salutat Paulum».

Vi ho proposto il latino perché la declinazione dei nomi consente di vedere anche più chiaramente che in italiano l'importanza degli attributi. In italiano non sarebbe priva di ambiguità una frase come «Pietro Paolo saluta», in latino «Petrus Paulum salutat» può avere un solo significato.

```
frase                   ::= soggetto gruppo-del-predicato
gruppo-del-predicato    ::= predicato complemento-oggetto
soggetto                ::= Pietro
predicato               ::= saluta
complemento-oggetto     ::= Paolo
```

*Figura 1 - Una grammatica molto semplice.*

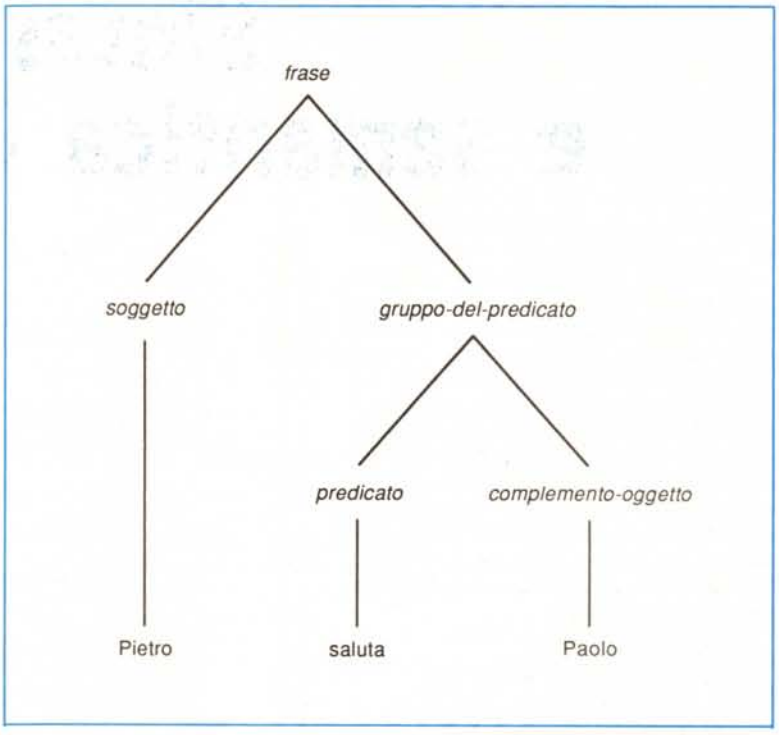


*Figura 2 - La struttura sintattica (secondo la grammatica della figura 1) della frase «Pietro saluta Paolo».*

In conclusione, per tradurre una frase dall'italiano al latino si deve prima individuare la sua struttura accertando che vengano rispettate le regole sintattiche, quindi assegnare il corretto valore ad un insieme di attributi mediante l'applicazione di «regole semantiche» (così dette anche se possono non aver nulla a che vedere con il «significato»). A rigore, si dovrebbe trasformare la grammatica in uno «schema di traduzione» inserendo nelle sue produzioni le opportune «azioni semantiche», ma questo è un aspetto che per ora è meglio rimandare ad un eventuale futuro. Quel che tuttavia non deve sfuggire è che si possono riscontrare due tipi di errori in una frase, sintattici e semantici: «Pietro domani Paolo» è sintatticamente sbagliata in quanto «domani» non è un predicato, «Pietro salutano Paolo» è semanticamente sbagliata (ripeto: qui per semantico si intende tutto ciò che riguarda gli «attributi», e il significato è solo uno dei possibili attributi).

## Dalle formule ai valori

Un altro esempio può magari essere utile, soprattutto perché è proprio a portata di mano.

Anche le formule di uno spreadsheet hanno un linguaggio e una grammatica: i token sono rappresentati da numeri, operatori, parentesi, riferimenti a caselle e funzioni predefinite (seno, coseno, ecc.); le produzioni ci dicono come possono essere combinati operatori ed operandi tenendo conto della associatività e della precedenza degli operatori (in «4+3*2», ad esempio, si deve prima eseguire la moltiplicazione, poi l'addizione).

Anche qui abbiamo un problema (facile facile) di attributi: l'attributo di un numero è il suo valore, quello del riferimento ad una casella è il valore della formula in questa contenuta, l'attributo di un «ramo» dell'albero sintattico è il valore della sub-espressione cui si riferisce (l'attributo di «3*2» è «6»), l'attributo di tutta la formula non è altro che il suo valore.

Quello che è meno facile è decidere che tipo di grammatica scrivere, quale metodo di analisi sintattica adottare. Avrete capito che, se parlo di esempio «a portata di mano» è perché la Borland ha sempre regalato agli utenti del Turbo Pascal il MicroCalc, un micro-spreadsheet che, se anche non può sostituire un Lotus, risulta tuttavia estremamente interessante per chi voglia dedicare un po' del suo tempo allo studio del sorgente. Bene: nel MicroCalc del Turbo Pascal 3.0 si usa un «recursive-descent parser», in quello del Turbo Pascal 4.0 si usa uno «shift-reduce parser». Due approcci completamente diversi, ognuno con i suoi pregi e difetti, tanto per dare un'idea della varietà delle scelte possibili.

Non è il caso di illustrare qui i diversi metodi di analisi sintattica («parsing» in inglese), né le tecniche per scrivere una grammatica adatta al tipo di parser che si è scelto, anche perché i comandi di QUED hanno una sintassi molto più semplice di quella delle formule di uno spreadsheet. Vi rimando quindi (almeno per ora) allo splendido *Compilers, Principles, Techniques and Tools* di A.V. Aho, R. Sethi e J. D. Ullman (Addison-Wesley), o, per un primo approccio, al solito *Algorithms + data Structures = Programs* di Wirth (contiene l'illustrazione di un semplice «recursive-descent parser»).

Vi prego comunque di dare un'occhiata al sorgente del MicroCalc distribuito con il Turbo Pascal 3.0; lì si sottintende una grammatica basata su cinque simboli non-terminali: «Expression», «SimpleExpression», «Term», «Factor» e «SignedFactor». La valutazione delle formule è affidata ad una procedura «Evaluate» che opera mediante cinque funzioni, una per ognuno di quei non-terminali. Potreste almeno guardarvi Wirth: la grammatica è più semplice (solo «Expression», «Term» e «Factor»), ed anche il programma (tre procedure con lo stesso nome dei non-terminali).

Se non avete voglia di leggere altro oltre a MC, almeno credetemi sulla parola: in alcuni casi basta scrivere la grammatica per avere già bella e pronta la struttura di quella parte del programma che si incaricherà dell'analisi sintattica. Il file QPARS.INC. di QUED (figura 3) è stato scritto proprio in questo modo.

## Dai comandi alle azioni

L'utente di QUED digita dei comandi, il programma deve «tradurli» in azioni, deve cioè operare secondo i desideri dell'utente come espressi in quella che abbiamo chiamato stringa-comando.

Esiste un numero finito di comandi, ma l'utente può digitare infinite stringhe-comando: basta pensare che ogni comando può essere preceduto da una sublista, ovvero dalla indicazione della prima e dell'ultima delle righe su cui il comando va eseguito, che ogni riga può essere indicata con il suo numero di riga assoluto (ad esempio un «4» per la quarta), con una stringa, con la distanza da un'altra riga («/pippo/—5»), ecc.

Proprio per questo abbiamo preferito partire dalla grammatica del linguaggio dei comandi di QUED: meglio lavorare sulla struttura sintattica della stringa-comando che portarsi appresso per tutto il programma il problema delle possibili infinite varianti. Non è difficile infatti costruire le nostre routine partendo dalla grammatica (vi ricordo che è stata pubblicata nel numero di ottobre): in essa abbiamo non-terminali come «sublista», «indirizzo», «flag»; in QPARS.INC abbiamo le procedure Q_Sublista, Q_Indirizzo, Q_Flag.

Naturalmente non si tratta solo di omonimie, e per verificarlo basta scorrere la grammatica e il sorgente.

La prima produzione della grammatica ci dice che la stringa comando può essere o un comando assoluto o un comando orientato; un comando assoluto non può essere preceduto da una sublista, un comando orientato invece sì; se la stringa comando comincia con una sublista e prosegue con un comando assoluto abbiamo quindi un errore di sintassi.

I comandi orientati possono essere o globali o locali; i primi sono preceduti da un «prefisso» che precisa su quali righe va eseguito o non va eseguito un comando. Non tutti i comandi orientati possono essere eseguiti globalmente; quelli per i quali ciò è possibile sono detti «ripetibili». Se l'eventuale prefisso non è seguito da un comando ripetibile si ha un errore di sintassi.

I comandi orientati locali possono a loro volta essere di due tipi: o ripetibili non ripetuti (cioè non preceduti dal «prefisso») o non ripetibili. Per il resto assomigliano molto, da un punto di vista sintattico, ai comandi assoluti: un carattere (ad esempio «a» per «append») eventualmente seguito da una STRINGA e/o da un «flag» (anch'esso opzionale).

Per semplificare il lavoro del parser ho considerato come un unico token combinazioni quali «e STRINGA» o «r STRINGA» (vi rimando al numero di ottobre per una spiegazione dei comandi «edit» e «read»), e soprattutto il comando di sostituzione («s/STRINGA/STRINGA/»); in questo modo quando il

```
{ QPARS.INC }

procedure Q_GotoRiga(var RP: NPtr; n: integer; var S: integer);
{ Va alla n-esima riga dopo (o prima se n < 0) quella indicata da RP }
begin
  S := OK;
  if n > 0 then
    while (n > 0) and (S = OK) do begin
      n := n - 1; RP := RP^.Next; if RP = NodoZeroPtr then S := ERRNUM
    end
  else
    while (n < 0) and (S = OK) do begin
      n := n + 1; RP := RP^.Prev; if RP = NodoZeroPtr then S := ERRNUM
    end
end;

procedure Q_CercaStringa(Dir: char; var RP: Nptr; var S: integer);
{ Cerca Stringa1 a partire dalla linea successiva o precedente }
begin
  RP := NodoCorrPtr; S := ERRCERCA;
  repeat
    if Dir = CERCANEXT then RP := RP^.Next else RP := RP^.Prev;
    if pos(Stringa1, RP^.Txt) <> 0 then S := OK
  until (RP = NodoCorrPtr) or (S = OK)
end;

procedure Q_Indirizzo(var RP: NPtr; var S: integer; var c: char);
var
  Num: integer;  Segno: char;
begin
  S := FINEDATI;                                    { per il caso: riga ::= '' }
  if c in CIFRE then begin
    RP := NodoZeroPtr; Q_GotoRiga(RP, Numero, S); c := Q_Token
  end
  else case c of
    RIGACORR : begin RP := NodoCorrPtr; S := OK; c := Q_Token end;
    ULTRIGA  : begin RP := NodoZeroPtr^.Prev; S := OK; c := Q_Token end;
    CERCANEXT,
    CERCAPREV: begin Q_CercaStringa(c, RP, S); c := Q_Token end
  end;
  if (S in [FINEDATI,OK]) and (c in [PIU,MENO]) then begin
    if S = FINEDATI then RP := NodoCorrPtr;
    Num := 1; Segno := c; c := Q_Token;
    if c in CIFRE then begin
      Num := Numero; c := Q_Token
    end;
    if Segno = MENO then Num := - Num;
    Q_GotoRiga(RP, Num, S)
  end
end;

procedure Q_ControllaSubLista(RP1, RP2: NPtr; var S: integer);
{ Controlla che la prima riga della sublista non sia successiva alla seconda }
var
  p: NPtr;
begin
  p := RP1;
  while (p <> RP2) and (S = OK) do begin
    p := p^.Next; if p = NodoZeroPtr then S := ERRSUBLST
  end
end;

procedure Q_SubLista(var RP1, RP2: NPtr; var n, S: integer; var c: char);
begin
  n := 0;                  { se non vengono indicati gli estremi della sublista }
  RP1 := NodoZeroPtr^.Next; RP2 := NodoZeroPtr^.Prev;  { default provvisorio }
  Q_Indirizzo(RP1, S, c);
  if S = OK then begin
    n := 1;
    if c = VIRGOLA then begin
      c := Q_Token; RP2 := NodoZeroPtr^.Next; Q_Indirizzo(RP2, S, c);
      if S = OK then begin
        n := 2; Q_ControllaSubLista(RP1, RP2, S)
      end
    end
  end;
  if S = FINEDATI then S := OK                      { se riga ::= '', n = 0 }
end;

procedure Q_Flag(tok, c: char; var g,n,p: boolean; var S: integer);
begin
  S := OK; g := FALSE; n := FALSE; p := FALSE;
  if c = GLOBSI then
    if tok = SUBST then begin g := TRUE; c := Q_Token end
    else S := ERRSUFF;              { 'g' ammesso solo dopo s/STRINGA/STRINGA/ }
  if (S = OK) and (c <> FINECMD) then begin
    case c of
      NUMERA: n := TRUE;
      PRINT : p := TRUE;
      else    S := ERRSUFF
    end;
    if Q_Token <> FINECMD then S := ERRSUFF
  end
end;
```

```
procedure Q_Marca(RP1, RP2: NPtr; c: char);
var
  P1, P2, P3: NPtr;
  Mark: boolean;
begin
  Mark := (c = GLOBSI);                          { vero se 'g', falso se 'v' }
  P1 := NodoZeroPtr^.Next;
  if NumAddr = 0 then begin                      { se non specificati estremi, }
    RP1 := P1; RP2 := NodoZeroPtr^.Prev          {   si assume tutto il buffer }
  end;
  P2 := RP1; P3 := RP2^.Next;
  while P1 <> P2 do begin
    P1^.Glob := FALSE; P1 := P1^.Next
  end;
  while P2 <> P3 do begin
    P2^.Glob := (pos(Stringa1,P2^.Txt^) <> 0) = Mark;
    P2 := P2^.Next
  end;
  while P3 <> NodoZeroPtr do begin
    P3^.Glob := FALSE; P3 := P3^.Next
  end
end;


procedure Q_ParseCS(var Stato: integer);
{ Esegue l'analisi sintattica dei comandi, che poi esegue }
const
  CMD_ASSOLUTI: set of char =
    [ERRMSG, HELP, PROMPT, QUITIF, QUIT, SILENT, EDITIF, EDIT, NUMFIL];
  CMD_RIPETIBILI: set of char = [DEL, NUMERA, PRINT, SUBST];
  CMD_NON_RIPETIBILI: set of char =
    [FINECMD, APPEND, INS, CHANGE, NUMLIN, MOVETO, COPYTO, READF, WRITEF];
var
  ch, ch2: char;
  RPtr1, RPtr2, RPtr3: NPtr;
  gFlag, nFlag, pFlag: boolean;
begin
  ch := Q_Token;
  Q_SubLista(RPtr1, RPtr2, NumAddr, Stato, ch);
  if Stato = OK then begin
    if (NumAddr <> 0) and (ch in CMD_ASSOLUTI) then Stato := ERRNADDR
    else if ch in [GLOBSI,GLOBNO] then begin
      Q_Marca(RPtr1, RPtr2, ch);
      ch := Q_Token;
      if ch = FINECMD then ch := PRINT;          { g/STRINGA/ == g/STRINGA/p }
      if ch = NONTOK then Stato := LexError
      else if not (ch in CMD_RIPETIBILI) then Stato := ERRGLOB
      else begin
        Q_Flag(ch, Q_Token, gFlag, nFlag, pFlag, Stato);
        if Stato = OK then
          Q_EseguiGlob(ch, RPtr1, RPtr2, gFlag, nFlag, pFlag, Stato)
      end
    end
    else if ch in CMD_ASSOLUTI+CMD_RIPETIBILI+CMD_NON_RIPETIBILI then begin
      ch2 := Q_Token;
      if ch in [MOVETO,COPYTO] then begin
        Q_Indirizzo(RPtr3, Stato, ch2);
        if Stato = FINEDATI then Stato := ERRDEST
      end;
      if Stato = OK then Q_Flag(ch, ch2, gFlag, nFlag, pFlag, Stato);
      if Stato = OK then
        Q_Esegui(ch, RPtr1, RPtr2, RPtr3, gFlag, nFlag, pFlag, FALSE, Stato)
    end
    else Stato := LexError
  end
end;
```

*Figura 3 - Il file QPARS.INC, contenente le routine di analisi sintattica di QUED.*

parser viene informato del riconoscimento di un token come «s» sa già di poter disporre dei necessari «attributi», ad esempio, per restare al comando di sostituzione, sia della stringa da cercare (nella variabile globale Stringa1) che di quella che dovrà sostituirla (Stringa2). In altri termini, una volta riconosciuto un comando assoluto o locale, il parser deve solo vedere se questo è seguito da una «n», una «p» o una «g» (un flag, appunto) prima di chiamare la procedura che lo eseguirà. C'è una sola eccezione: tra i comandi non ripetibili vi sono anche i comandi-movimento (quelli che copiano o muovono una o più righe da un punto all'altro del testo), i quali richiedono che venga specificato un indirizzo *dopo* il comando (si tratta del punto del testo in cui andranno copiate o trasferite le righe della «sublista»).

Potete verificare che la struttura della procedura Q_ParseCS non è altro che una traduzione in Pascal di tutta questa chiacchierata. Viene chiamata una procedura Q_SubLista per verificare se è presente l'indicazione di una sublista, si prosegue vedendo se compare un prefisso «g» o «v», in caso affermativo si chiama una procedura Q_EseguiGlob, altrimenti Q_Esegui (previo controllo della presenza dell'indirizzo dopo i comandi «m» e «t»), in tutti e due i casi dopo aver chiamato una procedura Q_Flag.

Oltre a Q_ParseCS, Q_SubLista e Q_Flag, in QPARS.INC trovate Q_Marca, che «marca» le righe su cui va poi eseguito un comando globale, e un grappolo di procedure controllate da Q_SubLista. A questa vengono passate come parametri due variabili globali e tre locali a Q_ParseCS; quelle globali sono *NumAddr*, numero degli indirizzi specificati (sarà 0 se non c'è sublista, 1 se viene indicato solo un indirizzo, 2 se vengono indicate sia la riga iniziale che quella finale), e *Stato* (per gli eventuali codici di errore); quelle locali sono *RPtr1* e *RPtr2*, puntatori alla prima e ultima riga dell'eventuale sublista, e *ch*, contenente il valore che Q_ParseCS riceve da Q_Token. Q_SubLista chiama subito Q_Indirizzo; questa non fa nulla (salvo assegnare FINEDATI a *Stato*) se il token contenuto in *ch* non ha niente a che vedere con un indirizzo, altrimenti esegue le sue «azioni semantiche» avvalendosi delle procedure Q_GotoRiga e Q_CercaStringa: cerca infatti la riga indicata dall'utente con un numero, con uno dei simboli «.» o «?» (che indicano rispettivamente la riga corrente e l'ultima riga del testo in memoria), con una stringa, e/o con una distanza positiva o negativa rispetto ad un'altra riga; se non la trova assegna a *Stato* un codice d'errore, che dobbiamo interpretare come errore «semantico» (indicare una riga che non esiste è un po' come pronunciare una parola senza senso). Se *Stato* vale OK, Q_SubLista sa che l'utente ha digitato un indirizzo; verifica quindi se questo è seguito da una virgola e da un secondo indirizzo e, in caso affermativo, chiama Q_ControllaSubLista per controllare che la prima riga della sublista non sia successiva alla seconda (sarebbe un altro tipo di errore semantico).

Le procedure Q_Esegui e Q_EseguiGlob, chiamate da Q_ParseCS se non vengono riscontrati errori sintattici o semantici, sono contenute nel file QCOMD.INC, che vedremo il mese prossimo.