Progetto di Linguaggi di programmazione
Seconda parte
Analizzatore sintattico

May 25, 2007

1 La grammatica del Lisp Kit

Per permettere agli studenti una più facile interazione con i docenti sul progetto (e anche il corso) abbiamo creato un blog molto semplice. L'indirizzo è http://corso-linguaggi-file.blog.kataweb.it/. Ho già inserito un post con qualche consiglio su come convenga trattare gli eventuali errori nell'input che potrebbero venire trovati dall'analizzatore lessicale.

La seconda parte del progetto consiste nella realizzazione in ML di un analizzatore sintattico. Esso prende in input la stringa di token prodotta dall'analizzatore lessicale (prima parte) applicato ad un programma in Lispkit. La seguente grammatica libera da contesto G_{LK} descrive la sintassi dei programmi Lisp Kit:

```
1 Prog::= let  Bind in Exp end | letrec Bind in Exp end

2 Bind::= var = Exp X
3 X :: = and Bind | epsilon

4 Exp ::= Prog | var Y |const | List |
          op ( Seq_Exp ) | lambda (Seq_Var ) Exp
5 Y :: = ( Seq_Exp ) | epsilon

6 Seq_Exp::= Exp Z
7 Z::= Seq_Exp | epsilon


8 Seq_Var::= var W
9 W :: = Seq_Var | epsilon

10 List :: = [ Const_List ] | nil
```

```
11 Const_List ::= const V | List V
12  V :: = ::Const_list | epsilon
```

I nonterminali sono numerati da 1 a 12 e le diverse produzioni del nonterminale i, vengono indicate con i.j. Per esempio, per il nonterminale Exp, con 4.2 indicheremo la produzione Exp::= var Y. I nonterminali iniziano con lettere maiuscole, mentre i terminali sono composti da sole minuscole. La grammatica G_{LK} non esprime le costanti semplici dei programmi Lisp Kit, cioè le costanti intere, T (True), F (False) e le stringhe. Il motivo è che l'analizzatore lessicale si prende già cura di queste costanti. Per esempio la costante 123 verrà rappresentata con la coppia $token\_lexema$ $(NM, M(NUM(123)))$ (dove 123 indica l'intero 123 e non la sequenza di 3 caratteri numerici) e quindi sarebbe inutile che G_{LK} si occupasse di generare le 3 cifre di 123. Lo stesso vale per le altre costanti semplici. Questo fatto corrisponde alla presenza nella grammatica del terminale $const$ che rappresenta tutte le costanti semplici. In pratica, questo significa che se stiamo cercando di riconoscere un'espressione (Exp) e troviamo in input una coppia $token\_lexema$ che corrisponde ad una costante semplice, allora dobbiamo usare la produzione 4.3, cioè $Exp ::= const$. Esattamente lo stesso accade per le variabili che devono venire analizzate dall'analizzatore lessicale. Di nuovo in G_{LK} le variabili sono modellate dal terminale var. Nello stesso modo il terminale op rappresenta tutti gli operatori. Nell'Appendice chiameremo questi simboli terminali **collettivi**. Questo trattamento speciale non vale invece per le liste costanti che, potendo contenere liste innestate non potrebbe venire controllata completamente dall'analizzatore lessicale e quindi questo controllo viene fatto da G_{LK}. Il nonterminale che genera le liste costanti è List (10). Di conseguenza, anche la costante nil che rappresenta la lista vuota generato da List (10.2).

Il significato delle produzioni della grammatica G_{LK} è semplice. Le produzioni di $Prog$ indicano che i programmi Lisp Kit consistono di espressioni let o $letrec$, ciascuna delle quali consiste di una sequenza di bind (generati dal nonterminale $Bind$), dopo i bind viene la key work "in", seguita da un'espressione che è il corpo dell'espressione, dopo la quale viene "end". La produzione $Bind ::= Var = Exp\, X$ e le produzioni per X specificano che ogni bind è una coppia $Var = Exp$. Infatti ogni bind dichiara una nuova variabile (Var) inizializzata da un'espressione (Exp). Queste variabili sono le variabili locali delle espressioni let o $letrec$. Il nonterminale X serve a generare un bind aggiuntivo (preceduto da "and") oppure a concludere (con la parola vuota $epsilon$) la lista dei bind.

A proposito delle produzioni relative al nonterminale Exp (4), notiamo i seguenti punti:

- la produzione 4.1 dice che un'espressione può essere un programma, cioè un'espressione let o $letrec$;
- la 4.2 $Exp ::= var\ Y$, serve a generare espressioni che consistono di una sola variabile, quando $Y ::= epsilon$, oppure genera invocazioni di funzione, quando $Y ::= (Seq\_Exp)$. Si intende che Seq_Exp generi i parametri attuali dell' invocazione.

- la produzione 4.3 dice che un'espressione può essere una costante e la 4.4 che può essere una lista la cui sintassi è spiegata dalle produzioni di $List$ (10).
- a questo punto le altre produzioni dovrebbero essere facilmente comprensibili senza ulteriori spiegazioni.

2 Cosa deve fare l'analizzatore sintattico

Il progetto del corso di Linguaggi dell'anno scorso consisteva nel realizzare un compilatore che traduceva programmi Lisp Kit in corrispondenti programmi per la macchina astratta SECD. Poichè partire dal Lisp Kit vero avrebbe richiesto uno sforzo eccessivo, abbiamo definito una rappresentazione molto semplice dei programmi Lisp Kit (detta Forma Semplice (FS)) e abbiamo considerato la compilazione da linguaggio FS a linguaggio SECD. La trasformazione da linguaggio Lisp Kit normale a FS doveva venir fatta "a mano". Quest'anno il progetto chiede proprio di automatizzare questo passaggio da Lisp Kit a FS. Quindi si tratta di realizzare una analizzatore lessicale (Parte 1 del progetto) seguito da un analizzatore sintattico nella Parte 2 del progetto che stiamo spiegando con il presente documento. L'analizzatore sintattico deve compiere le seguenti 2 azioni:

1. deve controllare che le stringhe in input siano programmi Lisp Kit corretti rispetto alla grammatica G_{LK} e
2. contemporaneamente al controllo sintattico costruisce la corrispondente espressione FS.

Esaminiamo questi 2 punti separatamente.

2.1 Come controllare che l'input sia corretto

Questa è l'azione di analisi sintattica vera e propria. Per realizzare il parser per G_{LK} si deve usare la tecnica vista nel corso che descriviamo nel seguito:

1. In appendice trovate le funzioni `FIRST` e `FOLLOW` di G_{LK}. Usando queste funzioni dovete scrivere la tabella di parsing. Se avrete fatto i conti correttamente, la tabella dovrebbe mostrare che G_{LK} è LL(1).
2. Dato che G_{LK} è LL(1), con l'aiuto della tabella dovrete completare la realizzazione di un analizzatore sintattico di tipo ricorsivo discendente senza backtrack per G_{LK}. Un tale analizzatore consiste di una funzione per ogni nonterminale di G_{LK} (più delle funzioni ausiliarie). In appendice trovate la maggior parte delle funzioni che realizzano un tale analizzatore. A voi si chiede solamente di realizzare le funzioni corrispondenti ai noterminali $Prog$ e Exp. Come funzionano queste funzioni lo capirete guardando quella date in appendice. In generale ciascuna di queste funzioni deve rispondere ai seguenti requisiti:
   - la funzione riceve in input una lista L di coppie $tokens\_lexema$ (prodotta dall'analizzatore lessicale a partire da un programma Lisp Kit);

- la funzione si comporta nel modo seguente: se la parte di L che viene esaminata non è sintatticamente corretta, allora deve lanciare un'opportuna eccezione (nelle funzioni date in appendice ci sono esempi di semplici eccezioni), mentre se questa parte è sintatticamente corretta, allora la funzione deve restituire due cose:
  - la espressione FS corripondente alla parte del programma L esaminato; come devono essere queste espressioni FS viene spiegato nella Sezione 2.2
  - la parte di L non consumata dalla funzione, cioè la parte del programma che resta da analizzare.

2.2 Come costruire le espressioni FS

Per capire questa seconda parte è necessario definire come sono fatte le espressioni FS. Esse sono definite dal seguente tipo `Sexpr` che usa il tipo $s\_espressione$ per le costanti:

```
datatype s_espressione = NUM of int |
                         STRINGA of string |
                         T | F | NIL |
                         DOT of s_espressione * s_espressione

datatype Sexpr = Var of string |
                 Quote of s_espressione |
                 Op of string * Sexpr list |
                 If of Sexpr * Sexpr * Sexpr |
                 Lambda of string list * Sexpr |
                 Call of Sexpr * Sexpr list   |
                 Let of Sexpr * string list * Sexpr list
                 Letrec of Sexpr * string list * Sexpr list
```

Il tipo `Sexpr` rappresenta i programmi Lisp Kit, ma introduce costruttori che "spiegano" le diverse parti del programma e le sotto-parti sono spesso ordinate in modo diverso da quello dei programmi Lisp Kit (ricordate che la forma FS è definita per rendere semplice la compilazione verso la SECD).

Vediamo qualche altro caso:

- i valori di tipo $s\_espressione$ rappresentano le costanti del linguaggio Lisp Kit. La costante T è rappresentata dal valore T, mentre una stringa costante $''pippo''$ è rappresentata da $STRINGA(\text{“}pippo'')$; Solo le liste costanti necessitano di qualche spiegazione aggiuntiva: la lista $[2,[5],10]$ viene rappresentato dalla $s\_espressione$ $DOT(NUM(2), DOT(DOT(NUM(5), NIL), DOT(NUM(10), NIL)))$. Nelle $Sexpr$ i valori costanti vengono rappresentati con il costruttore $QUOTE(costante)$ dove $costante$ rappresenta una qualsiasi $s\_espressione$ come quelle descritte prima.

- L'identificatore x viene rappresentato da $VAR(\text{“}x'')$;
- l'applicazione di un operatore, come per esempio $ADD(2,3)$ deve diventare la $Sexpr\ OP(\text{“}ADD'', [NUM(2), NUM(3)])$
- il costruttore If ha un uso ovvio: i suoi tre argomenti sono la condizione, il ramo then ed il ramo else del condizionale;
- Nel costruttore $Lambda$ il primo argomento è la lista delle variabili legate dal $Lambda$ ed il secondo argomento è il corpo della $Lambda$;
- il costruttore $Call$ modella le invocazioni di funzioni e quindi il primo argomento deve essere il nome della funzione da invocare ed il secondo la lista dei parametri attuali dell'invocazione;
- Consideriamo il costruttore Let. Il suo primo argomento è il corpo della definizione Let, il secondo è la lista delle variabili locali dichiarate nella Let e il terzo parametro è la lista delle espressioni che inizializzano queste variabili. Si deve osservare che queste 3 cose si trovano anche nella forma Lisp Kit della Let, ma in un ordine diverso. Consideriamo il fatti la produzione 1.1 di G_{LK} che specifica la sintassi del Let in Lisp Kit: $Prog ::= let\ Bind\ in\ Exp\ end$, il nonterminale Exp genera il corpo della Let, cio il primo argomento del costruttore Let di $Sexpr$, mentre il nonterminale $Bind$ genera una sequenza di dichiarazioni locali della forma seguente: $x1 = e1\ and\ x2 = e2\ and\ ...\ and\ xk = ek$, nella $Sexpr$ la lista delle variabili $[x1, ..., xk]$ corrisponde al secondo argomento del costruttore Let, mentre la lista $[e1, ..., ek]$ corrisponde al terzo argomento di Let. Da questa descrizione si evince che la funzione che corrisponde a $Prog$ del parser, quando sceglie la produzione 1.1, dovr (tra le altre cose) invocare le funzioni corrispondenti a $Bind$ e Exp e queste funzioni dovranno restituirgli i 3 pezzi (corpo, lista di variabili dichiarate, lista delle espressioni che li inizializzano) con cui la funzione $Prog$ potrà costruire la $Sexpr\ Let$(corpo, lista di variabili dichiarate, lista delle espressioni che li inizializzano) richiesta.
- Il costruttore $Letrec$ è simile al Let. le differenze sono solo semantiche e si capiranno guardando il compilatore e l'interprete.

Modalità di consegna

Devono essere consegnati due file: uno contenente la tabella di parsing e l'altro con il codice dell' analizzatore lessicale e di quello sintattico. Per quanto riguarda l'analizzatore sintattico, consegnate le 2 funzioni che vi sono richieste (ed eventualmente altre funzioni ausiliarie che servissero a queste 2) assieme a quelle che vi sono state date (vedi Appendice). Ovviamente le vostre funzioni devono funzionare correttamente assieme alle nostre ed assieme devono formare il richiesto analizzatore sintattico (e traduttore in FS).

Il primo file DEVE essere chiamato: "CognomeIniziale_tabella.*" (l'estensione è a vostra discrezione). Il secondo file va chiamato: "CognomeIniziale_lexsin.sml". I

file non nominati nel modo richiesto non verranno corretti. I file vanno consegnati almeno 5 giorni lavorativi prima dell'esame orale che si intende sostenere. Il progetto viene discusso col docente all'orale. La consegna va effettuata con l'apposito comando: "consegna progetto_linguaggi", da eseguire nel vostro folder che contiene SOLO i file da consegnare.

Esempi di traduzione Lisp Kit $\rightarrow$ FS

```
ESEMPIO 1

Codice LispKit
"let x=5 and y = 6 in IF ( LEQ ( x y )  SUB ( y x ) SUB ( x y ) ) end  $"


Lista di tokens prodotta dall'analizzatore lessicale:
[(LET, S "let"), (ID, S "x"), (SYM, S "="), (NM, M(NUM 5)), (AND, S "and"),
(ID, S "y"), (SYM, S "="), (NM, M(NUM 6)), (IN, S "in"), (OP, S "IF"),
(SYM, S "("), (OP, S "LEQ"), (SYM, S "("), (ID, S "x"), (ID, S "y"),
(SYM, S ")"), (OP, S "SUB"), (SYM, S "("), (ID, S "y"), (ID, S "x"),
(SYM, S ")"), (OP, S "SUB"), (SYM, S "("), (ID, S "x"), (ID, S "y"),
(SYM, S ")"), (SYM, S ")"), (END, S ''end''), (SYM, S "$")]


Sexpr prodotta dall'analizzatore sintattico
    Let(If(Op("LEQ", [Var "x", Var "y"]), Op("SUB", [Var "y", Var "x"]),
            Op("SUB", [Var "x", Var "y"])), ["x", "y"],
         [Quote(NUM 5), Quote(NUM 6)]



ESEMPIO 2

Codice LispKit
"let N = 3 and L = LAMBDA ( P Q R ) DIV (ADD (
ADD ( MUL ( P P ) MUL ( Q Q ) ) MUL ( R R ) ) N ) in L ( 2 4 6 ) end $"


Lista di tokens prodotta dall'analizzatore lessicale:
[(LET, S "let"), (ID, S "N"), (SYM, S "="), (NM, M(NUM 3)), (AND, S "and"),
 (ID, S "L"), (SYM, S "="), (LAMBDA, S "LAMBDA"), (SYM, S "("),
 (ID, S "P"), (ID, S "Q"), (ID, S "R"), (SYM, S ")"), (OP, S "DIV"),
 (SYM, S "("), (OP, S "ADD"), (SYM, S "("), (OP, S "ADD"), (SYM, S "("),
 (OP, S "MUL"), (SYM, S "("), (ID, S "P"), (ID, S "P"), (SYM, S ")"),
 (OP, S "MUL"), (SYM, S "("), (ID, S "Q"), (ID, S "Q"), (SYM, S ")"),
 (SYM, S ")"), (OP, S "MUL"), (SYM, S "("), (ID, S "R"), (ID, S "R"),
 (SYM, S ")"), (SYM, S ")"), (ID, S "N"), (SYM, S ")"), (IN, S "in"),
 (ID, S "L"), (SYM, S "("), (NM, M(NUM 2)), (NM, M(NUM 4)), (NM, M(NUM 6)),
 (SYM, S ")"), (END, S "end"), (SYM, S "$")]

Sexpr prodotta dall'analizzatore sintattico
    Let(Call(Var "L", [Quote(NUM 2), Quote(NUM 4), Quote(NUM 6)]), ["N", "L"],
         [Quote(NUM 3),
          Lambda(["P", "Q", "R"],
```

```
                Op("DIV",
                   [Op("ADD",
                       [Op("ADD",
                           [Op("MUL", [Var "P", Var "P"]),
                            Op("MUL", [Var "Q", Var "Q"])]),
                        Op("MUL", [Var "R", Var "R"])]), Var "N"]))])
```

ESEMPIO 3

Codice LispKit

```
"letrec
FACT = LAMBDA ( X ) IF ( EQ ( X 0 ) 1 MUL ( X FACT ( SUB ( X 1 ) ) ) )
and G = LAMBDA ( H L ) IF ( EQ ( L  NIL ) L
CONS ( H ( CAR ( L ) ) G ( H CDR ( L ) ) ) )
in G ( FACT [2::3::4::5] ) end $"
```

Lista di tokens prodotta dall'analizzatore lessicale:

```
[(LETREC, S "letrec"), (ID, S "FACT"), (SYM, S "="), (LAMBDA, S "LAMBDA"),
     (SYM, S "("), (ID, S "X"), (SYM, S ")"), (OP, S "IF"), (SYM, S "("),
     (OP, S "EQ"), (SYM, S "("), (ID, S "X"), (NM, M(NUM 0)), (SYM, S ")"),
     (NM, M(NUM 1)), (OP, S "MUL"), (SYM, S "("), (ID, S "X"), (ID, S "FACT"),
     (SYM, S "("), (OP, S "SUB"), (SYM, S "("), (ID, S "X"), (NM, M(NUM 1)),
     (SYM, S ")"), (SYM, S ")"), (SYM, S ")"), (SYM, S ")"), (AND, S "and"),
     (ID, S "G"), (SYM, S "="), (LAMBDA, S "LAMBDA"), (SYM, S "("),
     (ID, S "H"), (ID, S "L"), (SYM, S ")"), (OP, S "IF"), (SYM, S "("),
     (OP, S "EQ"), (SYM, S "("), (ID, S "L"), (Nil, M NIL), (SYM, S ")"),
     (ID, S "L"), (OP, S "CONS"), (SYM, S "("), (ID, S "H"), (SYM, S "("),
     (OP, S "CAR"), (SYM, S "("), (ID, S "L"), (SYM, S ")"), (SYM, S ")"),
     (ID, S "G"), (SYM, S "("), (ID, S "H"), (OP, S "CDR"), (SYM, S "("),
     (ID, S "L"), (SYM, S ")"), (SYM, S ")"), (SYM, S ")"), (SYM, S ")"),
     (IN, S "in"), (ID, S "G"), (SYM, S "("), (ID, S "FACT"), (SYM, S "["),
     (NM, M(NUM 2)), (SYM, S "::"), (NM, M(NUM 3)), (SYM, S "::"),
     (NM, M(NUM 4)), (SYM, S "::"), (NM, M(NUM 5)), (SYM, S "]"), (SYM, S ")"),
     (END, S "end"), (SYM, S "$")]
```

Sexpr prodotta dall'analizzatore sintattico

```
    Letrec(Call(Var "G",
                [Var "FACT",
                 Quote(DOT(NUM 2, DOT(NUM 3, DOT(NUM 4, DOT(NUM 5, NIL)))))]),
           ["FACT", "G"],
           [Lambda(["X"],
                   If(Op("EQ", [Var "X", Quote(NUM 0)]), Quote(NUM 1),
                      Op("MUL",
                         [Var "X",
                          Call(Var "FACT",
                               [Op("SUB", [Var "X", Quote(NUM 1)])])]))),
            Lambda(["H", "L"],
                   If(Op("EQ", [Var "L", Quote NIL]), Var "L",
                      Op("CONS",
                         [Call(Var "H", [Op("CAR", [Var "L"])]),
                          Call(Var "G",
                               [Var "H", Op("CDR", [Var "L"])])])))])
```

Appendice: FIRST e FOLLOW di G_{LK}

Nelle definizioni di FIRST e FOLLOW che seguono abbiamo introdotto dei simboli terminali speciali che chiameremo **collettivi**. Si tratta dei simboli $const$, var e op. Il primo sta per ogni costante semplice, il secondo per ogni identificatore e l'ultimo per ogni operatore. Usare questi 3 simboli (al posto delle reali stringhe che essi rappresentano) è una grossa semplificazione e questo è infatti il motivo per cui le abbiamo usate. Un'altra semplificazione che facciamo è quella di considerare le key word come un unico simbolo. Quindi per esempio "let" viene considerato un unico simbolo (d'altronde è racchiusa in un'unica coppia $token\_lexema$).

```
First(Prog)={let,letrec}
Follow(Prog)={$,end,),in}

First(Bind)= {var}
Follow(Bind)= {in}

First(X)={and, epsilon}
Follow(X)= {in}

First(Exp)={let,letrec,var,const,op,lambda,[,nil}
Follow(Exp)={and, end,),in,let,letrec,var,const,op,lambda,[,nil}

First(Y)={(,epsilon}
Follow(Y)={and, end,),in,let,letrec,var,const,op,lambda,[,nil}

First(Seq_Exp)= First(Exp)={let,letrec,var,cst,op,lambda,[,nil}
Follow(Seq_Exp)={)}

First(Z)={let,letrec,var,const,op,lambda,epsilon,[,nil}
Follow (Z)={)}

First(Seq_Var)={var}
Follow(Seq_Var)={)}

First(W)={var,epsilon}
Follow(W)={)}

First(List) = {[,nil}
Follow(List)={and, end,),in,let,letrec,var,const,op,lambda,[,nil,::,]}

First (Const_List) ={[,nil,const}
Follow(Const_List) ={]}

First(V) = {::, epsilon}
Follow(V) = {]}
```

Appendice: Un (grosso) pezzo dell'analizzatore sintattico

Nella descrizione precedente ho detto che l'analizzatore sintattico consiste di una funzione per ogni simbolo nonterminale. Questo resta vero, ma vedrete in quanto segue che oltre a queste funzioni ci sono molte altre funzioni ausiliarie e anche funioni che corripondono a quei simboli terminali di G_{LK} che abbiamo chiamato collettivi nell'Appendice precedente e cioè $const, var, op$.

```
datatype token = LET | IN | END | LETREC | AND |
                 LAMBDA | OP | ID | SYM | NM| STR | BOOL | Nil| Notoken



datatype s_espressione = NUM of int |
                         STRINGA of string |
                         T | F | NIL |
                         DOT of s_espressione * s_espressione


datatype Sexpr = Var of string |
                 Quote of s_espressione |
                 Op of string * Sexpr list |
                 If of Sexpr * Sexpr * Sexpr |
                 Lambda of string list * Sexpr |
                 Call of Sexpr * Sexpr list |
                 Let of Sexpr * string list * Sexpr list|
                 Letrec of Sexpr * string list * Sexpr list




datatype meta_S = M of s_espressione | S of string

type token_lexema = token * meta_S

exception e of string;

(*  Analizzatore lessicale da fare *)
```

```
(* estrae la constante dal costruttore di  M meta_S *)
fun quoting(M(Y)) = Y |quoting(S(X))= raise e("quoting applicato al costrutto

and

(* estrae la stringa dal costruttore di  S meta_S *)
unS(S(Y)) = Y | unS(M(Y))=  raise e("estrazione stringa da costruttore M")


and

(* estrae la stringa dal costruttore di  Var di Sexpr *)
unVar(Var(Y))=Y | unVar(_) = raise e("estrazione nome variabile
da costruttore errato")

and

(*testa e un token rappresenta una costante semplice *)
constant(t:token): bool =
t=NM orelse t=STR orelse t=Nil orelse t=BOOL

and

(* testa se un token appartiene a FIRST di Exp *)
expfirst(t:token): bool =
constant(t) orelse t=LET orelse t=LETREC orelse t=OP orelse t=LAMBDA
orelse t=ID

and

(* funzione corrispondente al terminale const *)
const(tkl:token_lexema list): s_espressione*token_lexema list =
    let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
    in
      case tkhd of
           NM   => (quoting(lxhd),tl(tkl)) | (* numero *)
           STR  => (quoting(lxhd),tl(tkl))  | (* stringa *)
           BOOL => (quoting(lxhd),tl(tkl)) |  (* T of F *)
           _ => raise e("const applicato a una non-costante")
    end

and
```

```
(*funzione corrispondente al terminale var *)
var(tkl:token_lexema list): Sexpr*token_lexema list=
     let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
     in
       if tkhd = ID then (Var(unS(lxhd)),tl(tkl)) else raise e("non e'una var
     end

and

(*funzione corrispondente al nonterminale Seq_Var *)
seqvar(tkl:token_lexema list): Sexpr list * token_lexema list=
 let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
 in
       if  tkhd = ID then (*se e' una variabile*)
        let
         val (sv,tv)=var(tkl) (*prendi primo elemento della sequenza*)
         val (ls,ts)=w(tv)  (*riconosci ricorsivamente il resto della
 sequenza *)
        in
         (sv::ls,ts) (*concatena e restituisci il resto della lista token_lex
        end
       else raise e("la sequenza di variabili non inizia con una variabile ")
end

and

(*funzione ausiliaria di seqvar*)
w(tkl:token_lexema list): Sexpr list * token_lexema list=
 let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
 in
       if  tkhd = ID then seqvar(tkl)
       else ([],tkl)
 end




and
```

```
v(tkl:token_lexema list)=
if hd(tkl) = (SYM, S("::"))
then constlist(tl(tkl))
else (NIL,tkl)


and

constlist(tkl:token_lexema list)=
  let
    val tkhd = #1(hd(tkl))
    val lxhd = #2(hd(tkl))
  in
    if (constant(tkhd))
  then
       let
           val (sc,tc) = const(tkl)
           val (sv,tv) = v(tc)
         in
       (DOT(sc,sv),tv)
     end
    else
      if ( hd(tkl)=(SYM,S("[")) orelse hd(tkl)=(Nil,M(NIL)) ) then
       let
        val (sl,tl) = lista(tkl)
        val (sv,tv) = v(tl)
       in
         (DOT(sl,sv),tv)
       end
      else raise e("token non compatibile con lista di costanti")
 end



and

(* funzione corrispondente al nonterminale list*)
lista(tkl:token_lexema list)=
 let
    val tkhd = #1(hd(tkl))
    val lxhd = #2(hd(tkl))
in
    if tkhd = Nil then (NIL,tl(tkl)) (*lista vuota *)
    else if (tkhd = SYM andalso lxhd=S("[")) (* altrimenti inizia con[ *)
         then
```

```
              let
                 val (cl,tr) = constlist(tl(tkl)) (* riconosco gli
elementi della lista *)
              in
                if (hd(tr) <> (SYM,S("]"))) (* deve rimanere ] *)
                then raise e("lista non chiusa correttamente")
                else (cl,tl(tr))
              end
          else raise e("non e' una lista")

end




and

(* funzione corrispondente al nonterminale Exp :  DA FARE *)
exp(tkl)= .......

and

(* funzione corrispondente al non terminale Seq_Exp *)
seqexp(tkl:token_lexema list): Sexpr list * token_lexema list=
 let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
 in
       if  (expfirst(tkhd) orelse (tkhd = SYM andalso lxhd=S("[")))
       then (* comincia con una espressione*)
        let
         val (se,te)=exp(tkl)
         val (ls,ts)=z(te)
        in
         (se::ls,ts)
        end
       else raise e("la sequenza di espressioni non inizia con una espression
end

and

(* funzione corrispondente al non terminale Z*)
z(tkl:token_lexema list): Sexpr list * token_lexema list=
 let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
```

```
 in
       if  (expfirst(tkhd) orelse (tkhd = SYM andalso lxhd=S("[")))
       then seqexp(tkl)
       else ([],tkl)
 end

and

(* funzione corrispondente al non terminale Bind *)
bind(tkl:token_lexema list): string list * Sexpr list *token_lexema list=
let
       val tkhd = #1(hd(tkl))
       val lxhd = #2(hd(tkl))
     in
      if(tkhd = ID) then
       let
         val sr = unS(lxhd); (* prima variabile *)
         val (er,ter) = exp(tl(tl(tkl))) (* espressione corrispondente
 alla prima variabile *)
         val (fw1,fw2,tf) = x(ter) (*riconosci il resto del bind*)
       in
         (sr::fw1,er::fw2,tf)
       end
     else raise e("il bind non comicia con un identificatore ")
end

and

(* funzione corrispondente al non terminale X*)
x(tkl:token_lexema list): string list * Sexpr list *token_lexema list=
let
          val tkhd = #1(hd(tkl))
          val lxhd = #2(hd(tkl))
        in
         case tkhd of
          AND  => bind(tl(tkl)) |
          _   => ([],[],tkl)
        end

and
 (* funzione corrispondente al nonterminale Prog: DA FARE  *)
prog(tkl:token_lexema list): Sexpr*token_lexema list=
```