

```

1
2 (* ----- *)
3 (* ----- Prima Parte ----- *)
4 (* ----- *)
5 (* ----- *)
6
7
8 (* ----- *)
9 (* ----- Dichiarazioni ----- *)
10 (* ----- *)
11
12 load "Int";
13
14 datatype token = LET | IN | END | LETREC | AND | LAMBDA | OP | ID | SYM | NM | STR |
  BOOL | Nil | Notoken
15
16 datatype s_espressione = NUM of int | STRINGA of string | T | F | NIL | DOT of
  s_espressione * s_espressione
17
18 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
19
20 datatype meta_S = M of s_espressione | S of string
21
22 type token_lexema = token * meta_S
23
24 exception e of string;
25
26
27 (* ----- *)
28 (* ----- Funzioni lparte ----- *)
29 (* ----- *)
30
31 fun lexi(s:string, tk1: token_lexema list): token_lexema list =
32 let
33 val L = explode s
34
35 fun isletter(c: char): bool =
36   if c = #"a" orelse c = #"b" orelse c = #"c" orelse c = #"d" orelse c = #"e"
37   orelse c = #"f" orelse c = #"g" orelse c = #"h" orelse c = #"i"
38   orelse c = #"j" orelse c = #"k" orelse c = #"l" orelse c = #"m"
39   orelse c = #"n" orelse c = #"o" orelse c = #"p" orelse c = #"q"
40   orelse c = #"r" orelse c = #"s" orelse c = #"t" orelse c = #"u"
41   orelse c = #"v" orelse c = #"w" orelse c = #"x" orelse c = #"y"
42   orelse c = #"z" orelse
43   c = #"A" orelse c = #"B" orelse c = #"C" orelse c = #"D" orelse c = #"E"
44   orelse c = #"F" orelse c = #"G" orelse c = #"H" orelse c = #"I"
45   orelse c = #"J" orelse c = #"K" orelse c = #"L" orelse c = #"M"
46   orelse c = #"N" orelse c = #"O" orelse c = #"P" orelse c = #"Q"
47   orelse c = #"R" orelse c = #"S" orelse c = #"T" orelse c = #"U"
48   orelse c = #"V" orelse c = #"W" orelse c = #"X" orelse c = #"Y"
49   orelse c = #"Z" orelse c = #"_"
50   then true
51   else false;
52 fun isnumero(c: char): bool =
53   if c = #"~" orelse c = #"0" orelse c = #"1" orelse c = #"2" orelse c = #"3" orelse c
54   = #"4" orelse c = #"5" orelse c = #"6" orelse c = #"7" orelse c = #"8" orelse c =
55   #"9"
56   then true
57   else false;
58 fun isKey(c: string): bool =
59   if c = "LET" orelse c = "IN" orelse c = "END" orelse c = "AND" orelse c = "NIL"
60   orelse c = "ADD" orelse c = "SUB" orelse c = "MUL" orelse c = "DIV"
61   orelse c = "REM" orelse c = "EQ" orelse c = "LEQ" orelse c = "CAR" orelse c =
62   "CDR" orelse c = "CONS" orelse c = "ATOM" orelse c = "IF" orelse c = "LAMBDA"
63   orelse
64   c = "in" orelse c = "end" orelse c = "and" orelse c = "nil" orelse c = "add"
65   orelse c = "sub" orelse c = "mul" orelse c = "div" orelse c = "rem" orelse c =
66   "eq" orelse c = "leq" orelse c = "car" orelse c = "cdr" orelse c = "cons"
67   orelse c = "atom" orelse c = "if" orelse c = "lambda"
68   then true
69   else false;
70
71 fun ccStringa(nil,R) = (nil ,nil)
72 | ccStringa(x::l,R) = if x <> #"\" then ccStringa(l,x::R) else (nil,nil)
73
74
75 fun ccNumero(nil,R) = (nil,R)
76 | ccNumero(x::l,R) = if isnumero(x) then ccNumero(l,x::R) else (x::l,R)

```

```

67
68 fun creaTokenLexema(c:string): token_lexema =
69     if c = "LET" then (LET,S c ) else if
70     c = "IN" then (IN,S c ) else if
71     c = "END" then (END,S c ) else if
72     c = "LETREC" then (LETREC,S c ) else if
73     c = "AND" then (AND,S c ) else if
74     c = "NIL" then (Nil,M NIL ) else if
75     c = "T" then (BOOL,M T ) else if
76     c = "F" then (BOOL,M F ) else if
77     c = "ADD" then (OP,S c ) else if
78     c = "SUB" then (OP,S c ) else if
79     c = "MUL" then (OP,S c ) else if
80     c = "DIV" then (OP,S c ) else if
81     c = "REM" then (OP,S c ) else if
82     c = "EQ" then (OP,S c ) else if
83     c = "LEQ" then (OP,S c ) else if
84     c = "CAR" then (OP,S c ) else if
85     c = "CDR" then (OP,S c ) else if
86     c = "CONS" then (OP,S c ) else if
87     c = "ATOM" then (OP,S c ) else if
88     c = "IF" then (OP,S c ) else if
89     c = "LAMBDA" then (LAMBDA,S c ) else if
90
91     c = "let" then (LET,S c ) else if
92     c = "in" then (IN,S c ) else if
93     c = "end" then (END,S c ) else if
94     c = "letrec" then (LETREC,S c ) else if
95     c = "and" then (AND,S c ) else if
96     c = "nil" then (Nil,M NIL ) else if
97     c = "t" then (BOOL,M T ) else if
98     c = "f" then (BOOL,M F ) else if
99     c = "add" then (OP,S c ) else if
100    c = "sub" then (OP,S c ) else if
101    c = "mul" then (OP,S c ) else if
102    c = "div" then (OP,S c ) else if
103    c = "rem" then (OP,S c ) else if
104    c = "eq" then (OP,S c ) else if
105    c = "leq" then (OP,S c ) else if
106    c = "car" then (OP,S c ) else if
107    c = "cdr" then (OP,S c ) else if
108    c = "cons" then (OP,S c ) else if
109    c = "atom" then (OP,S c ) else if
110    c = "if" then (OP,S c ) else if
111    c = "lambda" then (LAMBDA,S c )
112    else (ID,S c );
113
114
115 fun ccKey(nil,L) = (nil,"")
116 | ccKey(x::l,L) =
117     (* ho letto uno spazio quindi è una ID *)
118     if x = #" " then
119         let
120             (* ho il problema che in ID come questa *)
121             (* FACT il programma mi prenderebbe: *)
122             (* (BOOL,M F) , (ID,S "ACT") e non va *)
123             (* bene. *)
124             val stringa = implode(L)
125         in
126             if stringa = "T " then (l,"T")
127             else if stringa = "F " then (l,"F")
128             else (l,implode(L))
129         end
130     else if isletter(x) then (* controllo che sia un carattere *)
131         if isKey(implode(L@[x])) then
132             (* controllo che non sia LETREC *)
133             if hd(l) = #"R" andalso hd(tl(l)) = #"E" andalso hd(tl(tl(l))) = #"C"
134             orelse hd(l) = #"r" andalso hd(tl(l)) = #"e" andalso hd(tl(tl(l))) =
135             #"c"
136             then (tl(tl(tl(l))),implode
137                 (L@[x]@(hd(l)::hd(tl(l))::hd(tl(tl(l))))::nil)) ) (* "LETREC" *)
138             else ( l,implode(L@[x]) )
139         else ccKey(l,L@[x])
140     else raise e("non e' un carattere") (* se non è un carattere torno *)
141
142 (* FUNZIONE PRINCIPALE *)
143 fun principale(nil,tkl: token_lexema list) = tkl
144 | principale(x::l,tkl: token_lexema list) =
145     if isletter(x) then

```

```

145         let
146             val(resto,stringa)=ccKey(x::l,[])
147             val risultato = creaTokenLexema(stringa)
148         in
149             principale(resto,risultato::tkl )
150         end
151     else if isnumero(x) then
152         let
153             val(resto,numeroRis)=ccNumero(x::l,[])
154             val stringaRis = implode(numeroRis)
155             val intrRis = valOf(Int.fromString(stringaRis))
156         in
157             principale( resto , ( NM,M(NUM(intrRis)) )::tkl )
158         end
159
160     else
161         case x of
162             (* SEGNI DI PUNTEGGIATURA *)
163             #"(" => principale(l, (SYM,S (str x))::tkl ) |
164             #")" => principale(l, (SYM,S (str x))::tkl ) |
165             #"[" => principale(l, (SYM,S (str x))::tkl ) |
166             #"]" => principale(l, (SYM,S (str x))::tkl ) |
167             #"=" => principale(l, (SYM,S (str x))::tkl ) |
168             #":" => if hd(l) = #":" then principale(tl(l), (SYM,S "::-")::tkl )
169                     else raise e("errore :: ho un solo :") |
170             #"$" => principale(l, (SYM,S (str x) )::tkl ) |
171             (* STRINGHE *)
172             #"\" =>
173                 let
174                     val(resto,stringa)=ccStringa(x::l,[])
175                     val stringaRis = implode(stringa)
176                 in
177                     principale(resto , (SYM,S (stringaRis))::tkl )
178                 end |
179             (* cioè se è uno spazio !! --> lo ignoro !! *)
180             _ => principale(l, tk1 )
181
182     in
183         rev (principale(L,tkl))
184 end;
185
186
187 (* ----- *)
188 (* ----- Seconda Parte ----- *)
189 (* ----- *)
190 (* ----- *)
191
192
193 (* ----- *)
194 (* ----- Funzioni 2parte----- *)
195 (* ----- *)
196
197
198 (* estrae la costante dal costruttore di M meta_S *)
199 fun quoting(M(Y)) = Y | quoting(S(X))= raise e("quoting applicato al costruttore")
200 and
201 (* estrae la stringa dal costruttore di S meta_S *)
202 unS(S(Y)) = Y | unS(M(Y))= raise e("estrazione stringa da costruttore M")
203 and
204 (* estrae la stringa dal costruttore di Var di Sexpr *)
205 unVar(Var(Y))=Y | unVar(_) = raise e("estrazione nome variabile da costruttore errato")
206 and
207 (*testa se un token rappresenta una costante semplice *)
208 constant(t:token): bool = t=NM orelse t=STR orelse t=BOOL
209 and
210 (* ***** *)
211 (* *mancava: orelse t=Nil***** *)
212 (* ***** *)
213 (* testa se un token appartiene a FIRST di k *)
214 expfirst(t:token): bool = constant(t) orelse t=LET orelse t=LETREC orelse t=OP orelse
t=LAMBDA orelse t=ID orelse t=Nil
215 and
216 (* funzione corrispondente al terminale const *)
217 const(tk1:token_lexema list): s_espressione*token_lexema list =
218 let
219     val tkhd = #1(hd(tk1))
220     val lxhd = #2(hd(tk1))
221 in
222     case tkhd of
223         NM => (quoting(lxhd),tl(tk1)) | (* numero *)

```

```

224     STR => (quoting(lxhd),tl(tkl)) | (* stringa *)
225     BOOL => (quoting(lxhd),tl(tkl)) | (* T of F *)
226     _ => raise e("const applicato a una non-costante")
227 end
228 and
229 (*funzione corrispondente al terminale var *)
230 var(tkl:token_lexema list): Sexpr*token_lexema list=
231 let
232     val tkhd = #1(hd(tkl))
233     val lxhd = #2(hd(tkl))
234 in
235     if tkhd = ID then (Var(unS(lxhd)),tl(tkl)) else raise e("non e'una variabile")
236 end
237 and
238 (*funzione corrispondente al nonterminale Seq_Var *)
239 seqvar(tkl:token_lexema list): Sexpr list * token_lexema list=
240 let
241     val tkhd = #1(hd(tkl))
242     val lxhd = #2(hd(tkl))
243 in
244     if tkhd = ID then (*se e' una variabile*)
245         let
246             val (sv,tv)=var(tkl) (*prendi primo elemento della sequenza*)
247             val (ls,ts)=w(tv) (*riconosci ricorsivamente il resto della sequenza *)
248             in (sv::ls,ts) (*concatena e restituisci il resto della lista token_lexe *)
249         end
250     else raise e("la sequenza di variabili non inizia con una variabile ")
251 end
252 and
253 (*funzione ausiliaria di seqvar*)
254 w(tkl:token_lexema list): Sexpr list * token_lexema list=
255 let
256     val tkhd = #1(hd(tkl))
257     val lxhd = #2(hd(tkl))
258 in
259     if tkhd = ID then seqvar(tkl)
260     else ([],tkl)
261 end
262 and
263 v(tkl:token_lexema list)=
264     if hd(tkl) = (SYM, S("::")) then constlist(tl(tkl))
265     else (NIL,tkl)
266 and
267 constlist(tkl:token_lexema list)=
268 let
269     val tkhd = #1(hd(tkl))
270     val lxhd = #2(hd(tkl))
271 in
272     if (constant(tkhd)) then
273         let
274             val (sc,tc) = const(tkl)
275             val (sv,tv) = v(tc)
276             in (DOT(sc,sv),tv) end
277         else
278             if ( hd(tkl)=(SYM,S("[") ) orelse hd(tkl)=(Nil,M(NIL)) ) then
279                 let
280                     val (sl,tl) = lista(tkl)
281                     val (sv,tv) = v(tl)
282                     in (DOT(sl,sv),tv) end
283                 else raise e("token non compatibile con lista di costanti")
284             end
285         and
286         (* funzione corrispondente al nonterminale list*)
287         lista(tkl:token_lexema list)=
288         let
289             val tkhd = #1(hd(tkl))
290             val lxhd = #2(hd(tkl))
291         in
292             if tkhd = Nil then (NIL,tl(tkl)) (*lista vuota *)
293             else if (tkhd = SYM andalso lxhd=S("[") ) (* altrimenti inizia con[ *) then
294                 let
295                     val (cl,tr) = constlist(tl(tkl)) (* riconosco gli elementi della lista *)
296                 in
297                     if (hd(tr) <> (SYM,S("]"))) (* deve rimanere ] *)
298                     then raise e("lista non chiusa correttamente")
299                     else (cl,tl(tr))
300                 end
301             else raise e("non e' una lista")
302         end
303

```

```

304 (* -----FUNZIONE EXP DA FARE----- *)
305 and
306 (* funzione corrispondente al nonterminale Exp : DA FARE *)
307 exp(tkl:token_lexema list):Sexpr*token_lexema list=
308 let
309     val tkhd = #1(hd(tkl))
310     val lxhd = #2(hd(tkl))
311 in
312     case tkhd of
313     LET => prog(tkl) |
314     LETREC => prog(tkl) |
315     ID =>
316     (* Dentro a ID considero il caso di Y          *)
317     if hd(tl(tkl)) = (SYM ,S("(")) then
318         let
319             val (lista, resto) = seqexp(tl(#2(var(tkl))))
320         in
321             if hd(resto) = (SYM ,S("(")) then
322                 (Call(#1(var(tkl)) , lista), tl(resto))
323             else raise e("manca ) nel caso Y di ID")
324         end
325     else var(tkl) |
326     OP => if hd(tl(tkl)) = (SYM ,S("(")) then
327         (* ho controllato che ci sia la parentesi dopo l'operatore *)
328         (* ora:poichè IF-THE-ELSE e' particolare lo risolto a parte *)
329         if unS(lxhd) = "IF" then
330             let
331                 (* tl(tl(tkl)) perche': non passo a exp OP e ( sia *)
332                 (* perche sarebbe un errore logico che mi causerebbe *)
333                 (* tra l'altro un loop infinito *)
334                 val (risultato_condizione , resto_condizione) = exp(tl(tl(tkl)))
335                 val (risultato_then , resto_then) = exp(resto_condizione)
336                 val (risultato_else , resto_else) = exp(resto_then)
337             in
338                 (* controllo che ci sia la parentesi ) *)
339                 if hd(resto_else) = (SYM ,S("(")) then
340                     (* tl(resto_else) perche: non gli passo la parentesi ) *)
341
342                     (If(risultato_condizione,risultato_then,risultato_else),tl(resto_else))
343                     else raise e("Manca la parentesi ) ad IF")
344                 end
345             else
346                 (* per tutti gli altri operandi posso usare lo stesso costruito *)
347                 let
348                     val (risultato,resto) = seqexp(tl(tl(tkl)))
349                     val operatore = unS(lxhd)
350                 in
351                     if hd(resto) = (SYM ,S("(")) then
352                         (Op(operatore,risultato),tl(resto))
353                     else raise e("Manca la parentesi ) a OP")
354                 end
355             else raise e("Manca la parentesi ( dopo l'operatore") |
356     LAMBDA => if hd(tl(tkl)) = (SYM ,S("(")) then
357         (* controllo che ci sia ( delle variabili di LAMBDA *)
358         let
359             (* mangio le variabili tra le parentesi ( ) *)
360             val (variabili,resto) = seqvar(tl(tl(tkl)))
361         in
362             (* controllo che ci sia ) delle variabili di LAMBDA *)
363             if hd(resto) = (SYM ,S("(")) then
364                 let
365                     (* ovviamnte ora devo mangiare le exp del LAMBDA *)
366                     val (risultato_exp,resto_exp)=exp(tl(resto))
367                     (* mi serve la lista dei nomi delle variabili durante *)
368                     (* la costruzione del Lambda. *)
369                     fun SexprToString(nil:Sexpr list): string list = [] |
370                     SexprToString(listaSexpr:Sexpr list): string list =
371                         unVar(hd(listaSexpr)::SexprToString(tl(listaSexpr)))
372                 in
373                     (Lambda(SexprToString(variabili),risultato_exp),resto_exp)
374                 end
375             else raise e("Manca la parentesi ) dopo LAMBDA")
376         end
377     else raise e("Manca la parentesi ( dopo LAMBDA") |
378     _ => if constant(tkhd) then
379         let val (risultato,resto) = const(tkl) in (Quote(risultato), resto) end
380     else if (hd(tkl)=(SYM,S("[") ) orelse hd(tkl)=(Nil,M(NIL)) ) then
381         (* Considero cioè i cue casi: SYM,S("[") e Nil,M(NIL) *)
382         let val (risultato,resto) = lista(tkl) in (Quote(risultato), resto) end

```

```

381         else raise e("errore CASE di exp: tkhd no match")
382     end
383     (* -----FINE --FUNZIONE EXP DA FARE----- *)
384
385     and
386     (* funzione corrispondente al non terminale Seq_Exp *)
387     seqexp(tkl:token_lexema list): Sexpr list * token_lexema list=
388         let
389             val tkhd = #1(hd(tkl))
390             val lxhd = #2(hd(tkl))
391         in
392             if (expfirst(tkhd) orelse (tkhd = SYM andalso lxhd=S("[ ])))
393             then (* comincia con una espressione*)
394                 let
395                     val (se,te)=exp(tkl)
396                     val (ls,ts)=z(te)
397                 in (se::ls,ts) end
398             else raise e("la sequenza di espressioni non inizia con una espressione")
399         end
400     and
401     (* funzione corrispondente al non terminale Z*)
402     z(tkl:token_lexema list): Sexpr list * token_lexema list=
403         let
404             val tkhd = #1(hd(tkl))
405             val lxhd = #2(hd(tkl))
406         in
407             if (expfirst(tkhd) orelse (tkhd = SYM andalso lxhd=S("[ ])))
408             then seqexp(tkl)
409             else ([],tkl)
410         end
411     and
412     (* funzione corrispondente al non terminale Bind *)
413     bind(tkl:token_lexema list): string list * Sexpr list *token_lexema list=
414         let
415             val tkhd = #1(hd(tkl))
416             val lxhd = #2(hd(tkl))
417         in
418             if(tkhd = ID) then
419                 let
420                     val sr = unS(lxhd); (* prima variabile *)
421                     val (er,ter) = exp(tl(tl(tkl))) (* espressione corrispondente alla prima
422                     variabile *)
423                     val (fw1,fw2,tf) = x(ter) (*riconosci il resto del bind*)
424                 in (sr::fw1,er::fw2,tf) end
425             else raise e("il bind non comincia con un identificatore ")
426         end
427     and
428     (* funzione corrispondente al non terminale X*)
429     x(tkl:token_lexema list): string list * Sexpr list *token_lexema list=
430         let
431             val tkhd = #1(hd(tkl))
432             val lxhd = #2(hd(tkl))
433         in
434             case tkhd of
435             AND => bind(tl(tkl)) |
436             _ => ([],[],tkl)
437         end
438     (* -----FUNZIONE PROG DA FARE----- *)
439     and
440     (* funzione corrispondente al nonterminale Prog: DA FARE *)
441     prog(tkl:token_lexema list): Sexpr*token_lexema list=
442         let
443             val tkhd = #1(hd(tkl)) (* estraggo: token *)
444             val lxhd = #2(hd(tkl)) (* estraggo: meta_S *)
445         in
446             (* controllo che il primo token sia Let o Letrec, *)
447             (* se non e' cosi lancio l'eccezione *)
448             if ( tkhd = LET orelse tkhd = LETREC ) then
449                 let
450                     (* devo lanciare BIND,che genera la sequenza *)
451                     (* di dichiarazioni locali nella forma x1=e1 *)
452                     (* Per tener traccia del risultato uso delle *)
453                     (* variabili locali che mi tengono il *)
454                     (* risultato della funzione BIND. *)
455                     (* Ovviamente a bind gli passiamo la token_ *)
456                     (* lexema list che rimane dopo aver letto *)
457                     (* Let o Letrec *)
458                     (* *)
459                     (* MEMO => bind(tkl:token_lexema list): *)

```

```

460         (* string list*Sexpr list*token_lexema list *)
461     val (lista_stringhe,lista_Sexp,resto_bind) = bind(tl(tkl))
462 in
463     (* controllo che ora ci sia IN *)
464     if ( #1(hd(resto_bind)) = IN ) then
465     let
466         (* come prima,ma ora lanciola funzione exp *)
467         (* se tutto va bene una volta eseguita exp *)
468         (* mi devo trovare END *)
469         val (risultato_exp,resto_exp) = exp(tl(resto_bind))
470     in
471         if ( #1(hd(resto_exp)) = END ) then
472             (* OK va tutto bene non mi rimane altro che *)
473             (* restituire il risultato in base se ho *)
474             (* trovato LET o LETREC. *)
475             (* tkhd e' la testa della list del primo let *)
476             if ( tkhd = LET ) then
477                 (* Dove: *)
478                 (* risultato_exp => corpo del let *)
479                 (* lista_stringhe => le varibili del LET *)
480                 (* lista_Sexp => i valori delle variabili *)
481                 (Let(risultato_exp,lista_stringhe,lista_Sexp),tl(resto_exp))
482             (* quindi se non e' LET e' LETREC *)
483             else
484                 (Letrec(risultato_exp,lista_stringhe,lista_Sexp),tl(resto_exp))
485             else raise e("mi aspettavo END ")
486         end
487     else raise e("mi aspettavo IN ")
488
489     end
491     else raise e("prog non comincia con Let o con Letrec ")
492 end;
493
494 (* ----- *)
495 (* ----- *)
496 (* -----PROVA ESECUZIONE PROGRAMMA ----- *)
497 (* ----- *)
498 (* ----- *)
499
500 val stringaInput="let N = 3 and L = LAMBDA ( P Q R ) DIV (ADD ( ADD ( MUL ( P P ) MUL (
501 Q Q ) ) MUL ( R R ) ) N ) in L ( 2 4 6 ) end $";
502 lexi(stringaInput,[]);
503 prog(lexi(stringaInput,[]));
504
505 val stringaInput="letrec FACT = LAMBDA ( X ) IF ( EQ ( X 0 ) 1 MUL ( X FACT ( SUB ( X 1
506 ) ) ) ) and G = LAMBDA ( H L ) IF ( EQ ( L NIL ) L CONS ( H ( CAR ( L ) ) G ( H CDR ( L
507 ) ) ) ) in G ( FACT [2::3::4::5] ) end $";
508 lexi(stringaInput,[]);
509 prog(lexi(stringaInput,[]));
510
511

```