



UNIVERSITÀ DEGLI STUDI DI UDINE

Linguaggi e Compilatori 2

Progetto d'Esame

GRUPPO 6

Docente:

Prof. MARCO COMINI

Allievo:

ANDREA COMAND - 100668

ALBERTO MARTURANO - 102619

MARCO FRANCESCHETTI - 86955

ANNO ACCADEMICO 2012-2013

Esercizio 1

R	lineare sx	lineare dx	duplicante	eliminante	collassante
R1	✗	✓	✗	✗	✓
R2	✗	✗	✗	✗	✗
R3	✓	✗	✓	✓	✗
R4	✗	✓	✗	✗	✗
TRS	✗	✗	✓	✓	✓

Dato che il *TRS* non è lineare sinistro allora non è nemmeno ortogonale.

Dato che nella *R3* il primo argomento è una chiamata di funzione (e non un costruttore) la funzione *h* non è constructor based.

Per la stessa ragione anche la funzione *g* non è constructor based.

Tre coppie di regole sono in overlapping:

- *R1, R3* → testimone: *h (h (H [x]) [x]) z*
- *R2, R3* → testimone: *h (h z [x]) [0, h z [x]]*
- *R3, R4* → testimone: *h (h z [x]) [h z [x]]*

Esercizio 2

Questo è il codice WAM corrispondente al programma Prolog fornito.

```

safe/3:      try_me_else safeA/3
2          get_var X0 A1
4          get_str []/0 A2
4          get_var X1 A3
6          proceed

safeA/3:     trust_me
8          alloc 4
10         get_var Y0 A1
10         get_str [|]/2 A2
12         uni_var X0
12         uni_var Y1
14         get_var Y2 A3
14         put_val X0 A2
16         call noattack/3
16         put_var Y3 A1
16         put_str +/2 A2
18         set_val Y2
18         put_str 1/0 X1
20         set_val X1
20         call is/2
22         put_val Y0 A1

```

```

24          put_val Y1 A2
25          put_val Y2 A3
26          call safe/2
27          dealloc

28 noattack/3: alloc 3
29          get_var Y0 A1
30          get_var Y1 A2
31          get_var Y2 A3
32          call #\=/2
33          put_val Y1 A1
34          put_str +/2 A2
35          set_val Y0
36          set_val Y2
37          call #\=/2
38          put_val Y0 A1
39          put_str +/2 A2
40          set_val Y1
41          set_val Y2
42          call #\=/2
43          dealloc

44 brep/3:    try_me_else brepA
45          get_str z/0 A1
46          get_var X0 A2
47          get_str []/2 A3
48          uni_var X1
49          uni_val X0
50          get_str 0/0 X1
51          proceed

54 brepA/3:   retry_me_else brepB
55          get_str s/1 A1
56          uni_var X0
57          get_str z/0 X0
58          get_str []/2 A2
59          uni_var X1
60          uni_var X2
61          get_str []/2
62          uni_var X3
63          uni_var X4
64          get_str []/0 X4
65          get_str 1/0 X1
66          get_str 2/0 X3
67          get_str []/0 A3
68          proceed

```

```

70 brepB/3:    trust_me
71     alloc 4
72     get_str s/1 A1
73     uni_var X0
74     get_str s/1 X0
75     uni_var X1
76     get_var Y0 A2
77     get_var Y1 A3
78     put_val X1 A1
79     put_var Y2 A2
80     put_var Y3 A3
81     call div2/3
82     put_str s/1 A1
83     set_val Y2
84     put_str [|]/2 A2
85     set_val Y3
86     set_val Y0
87     put_val Y1 A3
88     call brep/3
89     dealloc
90
91 brep/2:    alloc 2
92     get_str s/1 A1
93     uni_var X0
94     get_str s/1 X0
95     uni_var X1
96     get_str [|]/2 A2
97     uni_var X2
98     uni_var Y0
99     get_str 1/0 X2
100    put_var Y1 A2
101    put_str s/1 A3
102    set_val X2
103    put_str z/0 X2
104    call half/3
105    put_val Y1 A1
106    put_val Y0 A2
107    call brep/2
108   dealloc

```

Esercizio 3

Per prima cosa cerchiamo t_1, t_2, t_3 .

```

h t1 [_ , _] = Nothing
2 h (Right x) t2 = Just (t3 + y)

```

Affinché h sia induttivamente sequenziale, $t1 = \text{Left } _$. Per gli altri due parametri abbiamo più scelta: $t2$ deve essere una lista in cui compare y e $t3$ deve essere un intero.

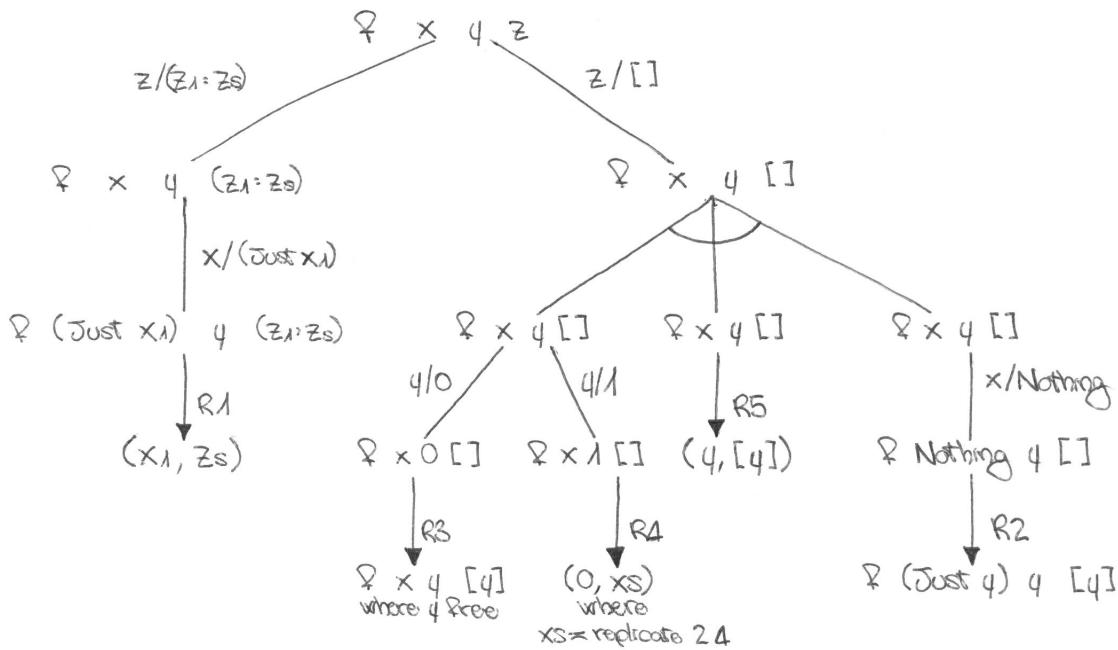
Visto che per via della query x è un intero, abbiamo messo $t2 = [y]$ e $t3 = \text{Left } x$, ottenendo:

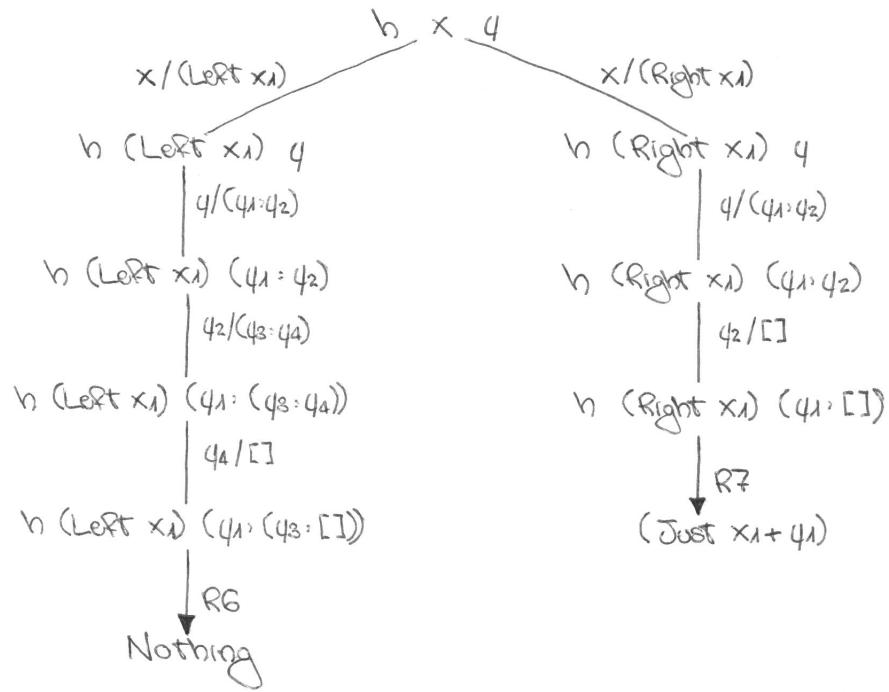
```

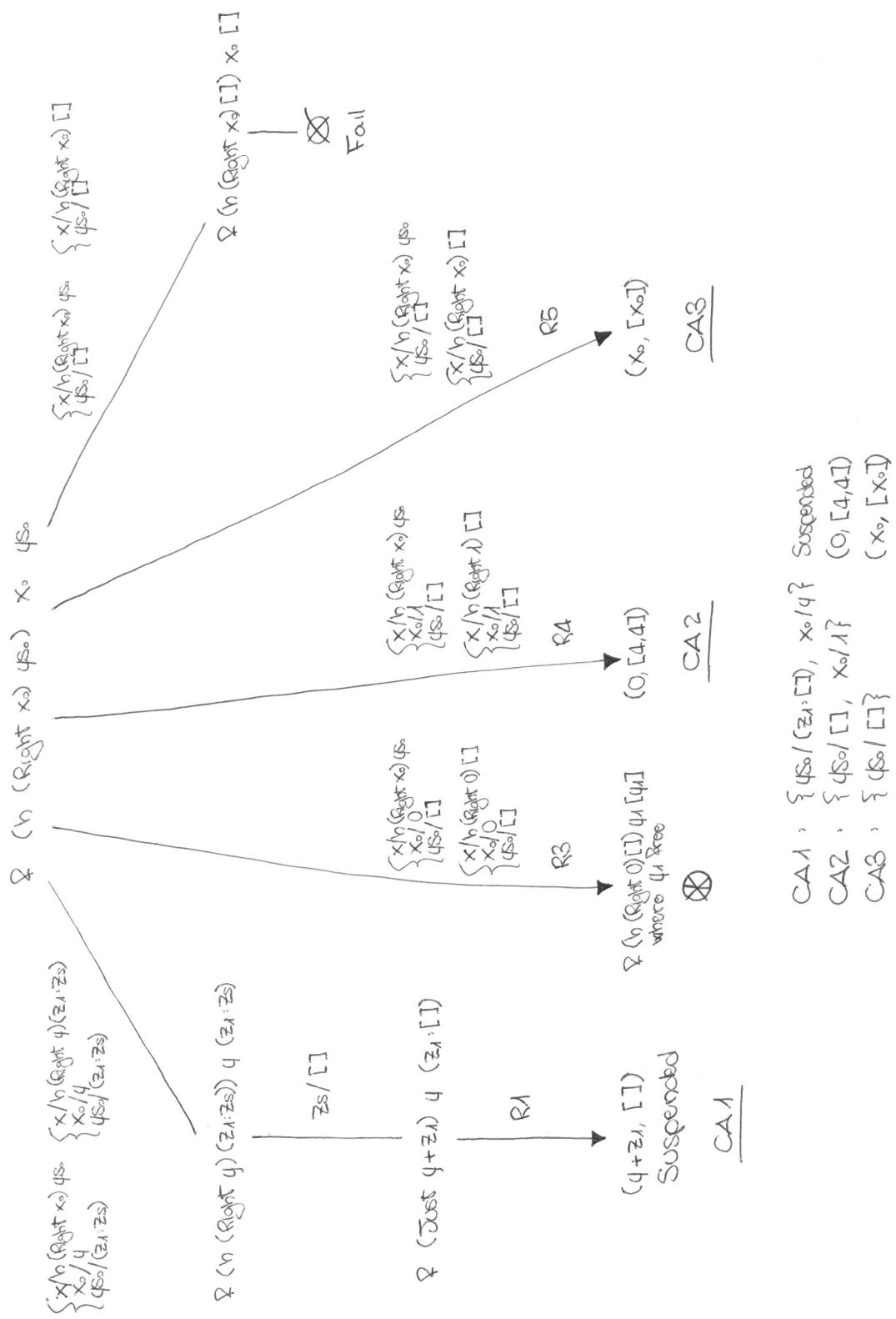
h (Left _) [_ , _] = Nothing
2 h (Right x) [y] = Just (x + y)

```

Seguono i definitional tree per le due funzioni e l'albero di needed narrowing per q .







$\vdash h (\text{Right } q) \ [1] \ q_1 [q_1] \quad \text{where } q_1 \text{ free}$
 ↓
~~⊗~~
 Fail

$h (\text{Right } q) (z_1, z_s)$
 ↓
 $z_s / []$
 ↓
 $R7$
 $(\text{Just } q + z_1)$

Esercizio 4

Rappresentazione delle queries

La prima query è la seguente:

```
g 2 (Right (Just success)) 1 [2]
```

Questa query ne in Haskell ne in Curry da alcun risultato. Infatti l'unica regola per g necessita che il secondo argomento sia **Left** z . Haskell quindi dirà che il pattern non è esaustivo e ritornerà un'eccezione.

Curry invece dirà che non ci sono soluzioni, visto che non ci sono regole in cui si può entrare.

La seconda query è la seguente:

```
g 2 (Left 5) (error "A") [5]
```

Haskell riesce a matchare con successo la regola con sostituzioni

$x/2$
 $z/5$
 $n/(error "A")$
 $y/[5]$

viene ora valutata la guardia, che diventa:

```
1 f (Just 2) (error "A") [5, 5] =:= (2, [5])
```

viene dunque ora eseguita la chiamata a f . Si entra nella prima regola con sostituzioni:

$$\begin{aligned}x/2 \\y/(error "A") \\z/5 \\zs/[5]\end{aligned}$$

La chiamata a f restituisce (x, zs) , ovvero $(2, [5])$. È ora possibile terminare la valutazione della guardia di g .

```
(2, [5]) =:= (2, [5])
```

Tale valutazione ritorna risultato **True** quindi viene restituito x ovvero 2, che è dunque il risultato della query.

Curry come Haskell ha successo nel pattern-matching ed entra nella regola di g con sostituzioni:

$$\begin{aligned}x/2 \\z/5 \\n/(error "A") \\y/[5]\end{aligned}$$

viene ora valutata la guardia, che diventa:

```
f (Just 2) (error "A") [5, 5] =:= (2, [5])
```

Curry, cerca ora di valutare il primo operando. Visto che l'ultimo argomento è una lista non vuota, può solamente utilizzare la prima regola di f . Curry dunque entra in quella regola con sostituzioni:

$$\begin{aligned}x/2 \\y/(error "A") \\z/5 \\zs/[5]\end{aligned}$$

La chiamata a f restituisce (x, zs) , ovvero $(2, [5])$. È ora possibile terminare la valutazione della guardia di g .

```
(2, [5]) =:= (2, [5])
```

Tale valutazione ritorna risultato **success** quindi viene restituito x ovvero 2, che è dunque il risultato della query.

Haskell e Curry si comportano quindi allo stesso modo in entrambe le queries.

Spazio di Ricerca

Segue lo spazio di ricerca per la query `g x y v z where x, y, v, z free`
Poiché g ha un sola regola, si entra in essa e si ottengono le sostituzioni:

$$\begin{array}{l} x_1/x \\ y/\text{Left}z_1 \\ n_1/v \\ y_1/z \end{array}$$

Si valuta dunque la guardia, che diventa:

```
f (Just x) v [z1, z1] =:= (x, z)
```

Per valutare il primo operando della guardia, poiché la lista è non vuota, solo la prima regola di f viene matchata. Si entra quindi in quella regola e si ottengono le sostituzioni:

$$\begin{array}{l} x_2/x \\ y_2/v \\ z_2/z_1 \\ zs/[z_1] \end{array}$$

La chiamata a f restituisce (x_2, zs) , ovvero $(z_1, [z_1])$. È ora possibile terminare la valutazione della guardia di g .

```
(x2, [z1]) =:= (x, z)
```

Che ha successo e genera le sostituzioni

$$\begin{array}{l} z/[z_1] \\ x/x_2 \end{array}$$

Abbiamo dunque la soluzione:

```
1 Result: x
2 Bindings:
3 x=x
4 y=Left z1
5 v=v
6 z=[z1]
```

Riscrittura del programma

Sia in Haskell che in Curry, le queries su g danno lo stesso risultato. Infatti la query a f nella guardia di g è deterministica, visto che matcha soltanto con la prima regola di f .

Riscriviamo ora il programma f affinché la sua esecuzione in Curry sia equivalente all'esecuzione del programma in Haskell. Riscrivere la versione Haskell per dare gli stessi risultati di quella di Curry non è fattibile (a meno di

simulare il non determinismo, ottenendo comunque qualcosa di molto diverso dal programma iniziale).

Il programma originale è il seguente.

```

1 f (Just x) y (z:zs) = (x, zs)
2 f Nothing y [] = f (Just y) y [y]
3 f x 0 [] = f x y [y] where y free
4 f x 1 [] = (0, xs) where xs = replicate 2 4
5 f x y [] = (y, [y])
6
7 g x (Left z) n y
8   | f (Just x) n [z, z]
9   :=: (x, y) = x

```

La seconda regola di *f*, eccetto per il primo argomento, è più generale delle ultime tre. Quindi in Haskell le ultime 3 vengono eseguite solo se la seconda non può essere eseguita, ovvero nel caso in cui il primo argomento è un Just. L'ultima regola inoltre viene eseguita solo se le due precedenti non possono essere eseguite a causa del secondo argomento. Di conseguenza possiamo riscrivere il programma nella seguente versione, dove viene tolta parte del non determinismo nel caso in cui sia eseguito in curry.

```

1 f (Just x) y (z:zs) = (x, zs)
2 f Nothing y [] = f (Just y) y [y]
3 f x@(Just x') 0 [] = f x y [y] where y free
4 f x@(Just x') 1 [] = (0, xs) where xs = replicate 2 4
5 f x@(Just x') y [] | y /= 0 && y /= 1 = (y, [y])
6
7 g x (Left z) n y
8   | f (Just x) n [z, z]
9   :=: (x, y) = x

```

La terza regola contiene una variabile libera, tuttavia il suo valore è ininfluente. Infatti viene eseguita una chiamata ricorsiva che può solamente usare la prima regola, visto che l'ultimo argomento è una lista non vuota:

```

1 f x@(Just x') 0 [] -> f x y [y] where y free -> (x',
[] )

```

Il programma diventa quindi:

```

1 f (Just x) y (z:zs) = (x, zs)
2 f Nothing y [] = f (Just y) y [y]
3 f x@(Just x') 0 [] = (x', [])
4 f x@(Just x') 1 [] = (0, xs) where xs = replicate 2 4
5 f x@(Just x') y [] | y /= 0 && y /= 1 = (y, [y])
6
7 g x (Left z) n y

```

```

| f (Just x) n [z, z]
9 := (x, y) = x

```

La prima funzione ora si comporta alla stessa maniera sia in Haskell che in Curry.

La seconda funzione può essere riscritta in maniera più semplice: infatti la chiamata a f nella guardia può entrare solamente nella prima regola.

```

1 f (Just x) n [z, z] = (x, [z])

```

La guardia può quindi essere riscritta come:

```

1 f (Just x) n [z, z] := (x, y) -> (x, [z]) == (x, y)
      -> [z] == y

```

Il programma alla fine diventa dunque:

```

1 f (Just x) y (z:zs) = (x, zs)
2   f Nothing y [] = f (Just y) y [y]
3   f x@(Just x') 0 [] = (x', [])
4   f x@(Just x') 1 [] = (0, xs) where xs = replicate 2 4
5   f x@(Just x') y [] | y /= 0 && y /= 1 = (y, [y])
6
7 g x (Left z) n y
     | y == [z] = x

```

Esercizio 5

Per rappresentare la soluzione del gioco del Contadino con la Capra, i Cavoli e il Lupo si è scelto di fare uso di una lista di mosse, a loro volta rappresentate ognuna con una tripla i cui elementi possono avere valore 0 o 1.

La tripla rappresenta, nell'ordine, la Capra, i Cavoli e il Lupo; il valore 1 indica che l'elemento corrispondente viene spostato da una riva a quella opposta, mentre il valore 0 indica che l'elemento non viene spostato. Il Contadino è sempre coinvolto negli spostamenti della barca, di conseguenza non è rappresentato nelle mosse.

Onde evitare un numero infinito di mosse, si è fatto uso del predicato *statoPassato* per tenere traccia delle posizioni sulle rive di Capra, Cavoli e Lupo; una mossa valida viene aggiunta alla lista delle mosse che costituiscono una soluzione solo se non conduce ad una configurazione di posizioni già vista in precedenza.

La validità di una mossa è verificata attraverso il predicato *noGood*: esso rappresenta i vincoli del gioco, ad esempio l'impossibilità di avere Capra e Cavoli insieme senza il Contadino.

Gli stati del gioco sono rappresentati, similmente alle mosse, mediante una quadrupla in cui il primo elemento rappresenta il Contadino e i rimanenti seguono l'ordinamento delle triple sopra esposto. In questo caso, i valori possibili per gli

elementi sono sempre 0 e 1, e indicano la presenza del corrispondente elemento su una o sull'altra riva. Si è scelto di indicare la configurazione iniziale con la quadrupla $(1,1,1,1)$ e quella finale con $(0,0,0,0)$.

Per ulteriori dettagli implementativi si rimanda al codice commentato.

Esercizio 6

Per eseguire il codice, lanciare `generatorsOff` dopo avere modificato la definizione di `f` nel file sorgente. Il codice va lanciato con MCC, in quanto utilizziamo le loro librerie per i Set.

I passaggi fondamentali della risoluzione dell'esercizio sono i seguenti:

- la generazione del sottogruppo di tutte le permutazioni π (diverse dall'identità) tali che $f(x) = f(x\pi)$;
- la ricerca dei generatori per tale sottogruppo.

La funzione `filt` risolve il primo punto. Essa si serve della funzione `gen` la quale genera tutte le possibili permutazioni per liste di quattro elementi. `filt` restituisce quindi (non deterministicamente) tutte le permutazioni π che soddisfano $f(x) = f(x\pi)$ per ogni x del dominio di f .

Per dettagli sull'implementazione si rimanda al codice commentato.

La funzione `generatorsOff` ottiene un generator set del gruppo sopracitato e scrive le permutazioni in un formato univoco. Per farlo, sfrutta la funzione `findGenerators`: essa genera l'insieme dei generatori ricevuto in input calcolando un punto fisso. Per farlo sfrutta una funzione che mantiene l'insieme di permutazioni da generare, un insieme candidato ad essere l'insieme generatore e un insieme di permutazioni generabili con l'insieme candidato ad essere generatore. Ad ogni passo, tutte le permutazioni generabili sono composte tra di loro per ottenere il nuovo insieme di permutazioni generabili. Quando si arriva al punto fisso, ovvero si ha generato tutto il generabile con l'insieme candidato a generatore corrente, si aggiunge un nuovo generatore. Il processo termina quando si ha generato tutto il sottogruppo fornito in input.

Per risolvere l'esercizio, inoltre, sono state definite delle istruzioni di supporto necessarie al soddisfacimento del secondo punto.

La funzione `normalizeCycles` riscrive una permutazione eliminando i cicli con meno di due elementi e modificando ogni ciclo rimanente ruotandolo in modo da portare l'elemento minore al primo posto, e poi convertendolo in prodotto di scambi di cui il primo elemento è fisso (es. $[[1, 2, 3]] = [[1, 3], [1, 2]]$).

Utilizziamo questa forma per disporre di una rappresentazione univoca delle permutazioni.

Vengono definite altre funzioni fondamentali: la funzione `compose` che compone due permutazioni, la funzione `toSingleCycle` che scrive una permutazione come singolo ciclo (quando possibile) e la funzione `toCycles` che data una lista di posizioni la converte in permutazione (dove ogni ciclo è uno scambio) (es `toCycles [2,3,4,1] -> [[1,4],[1,3],[1,2]]`).

Per i dettagli sull'implementazione si rimanda al codice commentato.

Esercizio 7

Una versione non ambigua della grammatica fornita è la seguente:

- $SL \rightarrow S FS$
- $FS \rightarrow ; \mid SL \mid \epsilon$
- $S \rightarrow \text{while} (E) S \mid \text{print} (E) \mid \{ SL \} \mid e$
- $E \rightarrow T FT$
- $FTt \rightarrow + E \mid \epsilon$
- $T \rightarrow id \mid num$

I First sets dei nonterminali di tale grammatica sono i seguenti:

- $FIRST(SL) = \{\text{while}, \text{print}, \{, e\}$
- $FIRST(FS) = \{; , \epsilon\}$
- $FIRST(S) = \{\text{while}, \text{print}, \{, e\}$
- $FIRST(FT) = \{+ , \epsilon\}$
- $FIRST(E) = \{id, num\}$
- $FIRST(T) = \{id, num\}$

I Follow sets sono i seguenti:

- $FOLLOW(SL) = \{\} , \$\}$
- $FOLLOW(FS) = \{\} , \$\}$
- $FOLLOW(S) = \{\} , ; , \$\}$
- $FOLLOW(FT) = \{\}\}$
- $FOLLOW(E) = \{\}\}$
- $FOLLOW(T) = \{+ ,)\}$

Nel seguito è presentata la tabella di parsing.

Tabella di Parsing

	\$	num	id	+	e	}	{)	(print	while	;
SL					SL → S FS		SL → S FS			SL → S FS	SL → S FS	
FS	FS → ε					FS → ε						FS → ; SL
S					S → e		S → { SL }			S → print (E)	S → while (E) S	
E		E → T FT	E → T FT									
FT				FT → + E				FT → ε				
T		T → num	T → id									

Durante l'esecuzione del parser sull'input si rende necessario adottare una strategia di error recovery, in particolare nel momento in cui viene letto in input il terminale *e* immediatamente dopo la lettura di una prima occorrenza dello stesso terminale. Per la gestione della error recovery si è usato il Panic Mode, illustrato di seguito.

Si costruisce, per ogni nonterminale *X* della grammatica, l'insieme di sincronizzazione $sync(X) = FOLLOW(X)$; successivamente si procede eliminando un simbolo di input alla volta finché non si incontra un simbolo *a* appartenente al *Follow set* del nonterminale correntemente in cima allo stack; solo allora si elimina dallo stack il nonterminale presente al top di esso.

Nel nostro caso quindi consideriamo $sync(FS) = FOLLOW(FS) = \{\}, \$\}$; eliminiamo il simbolo *e* che generava il conflitto, trovando ora in input il nuovo simbolo *्*. Notiamo che esso appartiene all'insieme di sincronizzazione di *FS*, di cui possiamo quindi fare il *pop* dallo stack; l'esecuzione del parser può ora riprendere normalmente. Segue la simulazione del parsing sull'input proposto.

Simulazione del Parsing sull'input

STACK	{ while (id) { e e } } \$
SL \$	↑
S FS \$	↑
{ SL } FS \$	↑
SL } FS \$	↑
S FS } FS \$	↑
while (E) S FS } FS \$	↑
(E) S FS } FS \$	↑
E) S FS } FS \$	↑
T FT) S FS } FS \$	↑
id FT) S FS } FS \$	↑
FT) S FS } FS \$	↑
) S FS } FS \$	↑
S FS } FS \$	↑
{ SL } FS } FS \$	↑
SL } FS } FS \$	↑
S FS } FS } FS \$	↑
e FS } FS } FS \$	↑
FS } FS } FS \$	↑
ERROR RECOVERY	
FS } FS } FS \$	↑
} FS } FS \$	↑
FS } FS \$	↑
} FS \$	↑
FS \$	↑
\$	↑

Esercizio 8

Una grammatica equivalente a quella fornita è la seguente:

- $C \rightarrow C \& C1 \mid C1$
- $C1 \rightarrow !C1 \mid E \text{ rel } E \mid \text{pred} (Es)$
- $E \rightarrow E + E1 \mid E1$
- $E1 \rightarrow E1 * E2 \mid E2$
- $E2 \rightarrow - E2 \mid (E) \mid \text{num} \mid id$

- $Es \rightarrow E, Es | E$

I First e Follow sets dei nonterminali di tale grammatica sono i seguenti:

- $FIRST(C) = \{!, pred, -, (\, num, id\}$
- $FIRST(C1) = \{!, pred, -, (\, num, id\}$
- $FIRST(E1) = \{-, (\, num, id\}$
- $FIRST(E2) = \{-, (\, num, id\}$
- $FIRST(Es) = \{-, (\, num, id\}$
- $FIRST(E) = \{-, (\, num, id\}$
- $FOLLOW(C) = \{\&, \$\}$
- $FOLLOW(C1) = \{\&, \$\}$
- $FOLLOW(E1) = \{*, rel, +,), virgola, \&, \$\}$
- $FOLLOW(E2) = \{*, rel, +,), virgola, \&, \$\}$
- $FOLLOW(Es) = \{\}\}$
- $FOLLOW(E) = \{rel, +,), virgola, \&, \$\}$

Si è assunto che l'operatore $\&$ abbia precedenza inferiore rispetto all'operatore $!$, e che gli operatori $rel, +, *$ e $-$ siano in ordine di precedenza crescente.
Le tabelle delle Action e dei Goto del parser SLR per questa grammatica, assieme agli stati dell'automa, sono riportate di seguito.

Tabella di Parsing (Action)

	\$,	id	num)	(-	*	+	pred	rel	!	&
0			s11	s10		s9	s8			s7		s6	
1									s19		s18		
2	r(E1 → E2)	r(E1 → E2)			r(E1 → E2)			r(E1 → E2)	r(E1 → E2)		r(E1 → E2)		r(E1 → E2)
3	r(E → E1)	r(E → E1)			r(E → E1)			s17	r(E → E1)		r(E → E1)		r(E → E1)
4	r(C → C1)												r(C → C1)
5	accept												s16
6			s11	s10		s9	s8			s7		s6	
7						s14							
8			s11	s10		s9	s8						
9			s11	s10		s9	s8						
10	r(E2 → num)	r(E2 → num)			r(E2 → num)			r(E2 → num)	r(E2 → num)		r(E2 → num)		r(E2 → num)
11	r(E2 → id)	r(E2 → id)			r(E2 → id)			r(E2 → id)	r(E2 → id)		r(E2 → id)		r(E2 → id)
12					s26				s19				
13	r(E2 → - E2)	r(E2 → - E2)			r(E2 → - E2)			r(E2 → - E2)	r(E2 → - E2)		r(E2 → - E2)		r(E2 → - E2)
14			s11	s10		s9	s8						
15	r(C1 → ! C1)												r(C1 → ! C1)
16			s11	s10		s9	s8			s7		s6	
17			s11	s10		s9	s8						
18			s11	s10		s9	s8						
19			s11	s10		s9	s8						
20	r(E → E + E1)	r(E → E + E1)			r(E → E + E1)			s17	r(E → E + E1)		r(E → E + E1)		r(E → E + E1)
21	r(C1 → E rel E)								s19				r(C1 → E rel E)
22	r(E1 → E1 * E2)	r(E1 → E1 * E2)			r(E1 → E1 * E2)			r(E1 → E1 * E2)	r(E1 → E1 * E2)		r(E1 → E1 * E2)		r(E1 → E1 * E2)
23	r(C → C and C1)												r(C → C and C1)
24		s28			r(Es → E)				s19				
25					s27								
26	r(E2 → (E))	r(E2 → (E))			r(E2 → (E))			r(E2 → (E))	r(E2 → (E))		r(E2 → (E))		r(E2 → (E))
27	r(C1 → pred (Es))												r(C1 → pred (Es))
28			s11	s10		s9	s8						
29					r(Es → E , Es)								

Tabella di Parsing (Goto)

	C	C1	E1	E2	Es	E
0	s5	s4	s3	s2		s1
1						
2						
3						
4						
5						
6		s15	s3	s2		s1
7						
8				s13		
9			s3	s2		s12
10						
11						
12						
13						
14			s3	s2	s25	s24
15						
16		s23	s3	s2		s1
17				s22		
18			s3	s2		s21
19			s20	s2		
20						
21						
22						
23						
24						
25						
26						
27						
28			s3	s2	s29	s24
29						

$$I_0 = \{ \overline{S} \rightarrow .C, C \rightarrow .C \& C1, C \rightarrow .C1, C1 \rightarrow !C1, C1 \rightarrow .E \text{ rel } E, C1 \rightarrow .\text{pred}(Es), \\ E \rightarrow .E + E1, E \rightarrow .E1, E1 \rightarrow .E1 * E2, E1 \rightarrow .E2, E2 \rightarrow .-E2, E2 \rightarrow .(E), \\ E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_1 = \{ C1 \rightarrow E . \text{rel } E, E \rightarrow E . + E1 \}$$

$$I_2 = \{ \overline{E1} \rightarrow E2 . ? \}$$

{ , *, +, &, *, rel, \$ }

$$I_3 = \{ \overline{E} \rightarrow E1 . , E1 \rightarrow E1 * E2 ? \}$$

{ , +, &, *, rel, \$ }

$$I_4 = \{ \overline{C} \rightarrow C1 . ? \}$$

{ &, \$ }

$$I_5 = \{ C \rightarrow C . \& C1, \overline{C} \rightarrow C . ? \}$$

{ \$ }

$$I_6 = \{ \overline{C1} \rightarrow !.C1, C1 \rightarrow .!C1, C1 \rightarrow .E \text{ rel } E, C1 \rightarrow .\text{pred}(Es), E \rightarrow .E + E1, \\ E \rightarrow .E1, E1 \rightarrow .E1 * E2, E1 \rightarrow .E2, E2 \rightarrow .-E2, E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_7 = \{ C1 \rightarrow \text{pred}.(Es) \}$$

$$I_8 = \{ \overline{E2} \rightarrow - . E2, E2 \rightarrow . - E2, E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_9 = \{ \overline{E2} \rightarrow (. E), E \rightarrow .E + E1, E \rightarrow .E1, E1 \rightarrow .E1 * E2, E1 \rightarrow .E2, E2 \rightarrow .-E2, \\ E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_{10} = \{ \overline{E2} \rightarrow \text{num}. ? \}$$

{ , *, +, &, *, rel, \$ }

$$I_{11} = \{ \overline{E2} \rightarrow \text{id}. ? \}$$

{ , *, +, &, *, rel, \$ }

$$I_{12} = \{ E \rightarrow E . + E1, \overline{E2} \rightarrow (E) ? \}$$

$$I_{13} = \{ \overline{E2} \rightarrow - E2 . ? \}$$

{ , *, +, &, *, rel, \$ }

$$I_{14} = \{ \overline{C1} \rightarrow \text{pred}.(Es), Es \rightarrow .E, Es, Es \rightarrow .E, E \rightarrow .E + E1, E \rightarrow .E1, \\ E1 \rightarrow .E1 * E2, E1 \rightarrow .E2, E2 \rightarrow .-E2, E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_{15} = \{ \overline{C1} \rightarrow !C1 . ? \}$$

{ &, \$ }

$$I_{16} = \{ \overline{C} \rightarrow C . \& C1, C1 \rightarrow .!C1, C1 \rightarrow .E \text{ rel } E, C1 \rightarrow .\text{pred}(Es), E \rightarrow .E + E1, \\ E \rightarrow .E1, E1 \rightarrow .E1 * E2, E1 \rightarrow .E2, E2 \rightarrow .-E2, E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_{17} = \{ \overline{E1} \rightarrow E1 * . E2, E2 \rightarrow . - E2, E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_{18} = \{ \overline{C1} \rightarrow E \text{ rel }. E, E \rightarrow .E + E1, E \rightarrow .E1, E1 \rightarrow .E1 * E2, E1 \rightarrow .E2, E2 \rightarrow .-E2, \\ E2 \rightarrow .(E), E2 \rightarrow .\text{num}, E2 \rightarrow .\text{id} \}$$

$$I_{19} = \{ \underbrace{E \rightarrow E + . E_1}, \underbrace{E_1 \rightarrow . E_1 * E_2}, \underbrace{E_1 \rightarrow . E_2}, \underbrace{E_2 \rightarrow . - E_2}, \underbrace{E_2 \rightarrow . (E)}, \\ \underbrace{E_2 \rightarrow . \text{num}}, \underbrace{E_2 \rightarrow . \text{id}} \}$$

$$I_{20} = \{ \underbrace{E \rightarrow E + E_1.}_{\{\), +, &, , rel, \$\}}, \underbrace{E_1 \rightarrow E_1. * E_2?}_{\{\), *, +, &, , rel, \$\}}$$

$$I_{21} = \{ \underbrace{C_1 \rightarrow E \text{ rd } E.}_{\{\&, \$\}}, \underbrace{E \rightarrow E. + E_1?}_{\{\&, \$\}}$$

$$I_{22} = \{ \underbrace{E_1 \rightarrow E_1 * E_2.}_{\{\), *, +, &, , rel, \$\}}$$

$$I_{23} = \{ \underbrace{C \rightarrow C \& C_1.}_{\{\&, \$\}}$$

$$I_{24} = \{ \underbrace{E \rightarrow E. + E_1}, \underbrace{E_S \rightarrow E.}_{\{\}\}}, \underbrace{E_S \rightarrow E., E_S?}_{\{\}\}}$$

$$I_{25} = \{ \underbrace{C_1 \rightarrow \text{pred}(E_S.)?}_{\{\&, \$\}}$$

$$I_{26} = \{ \underbrace{E_2 \rightarrow (E).?}_{\{\), *, +, &, , rel, \$\}}$$

$$I_{27} = \{ \underbrace{C_1 \rightarrow \text{pred}(E_S).?}_{\{\&, \$\}}$$

$$I_{28} = \{ \underbrace{E_S \rightarrow E., E_S?}_{\{\}\}}, \underbrace{E_S \rightarrow . E, E_S}, \underbrace{E_S \rightarrow . E, E_S?}_{\{\}\}}, \underbrace{E \rightarrow . E + E_1}, \underbrace{E \rightarrow . E_1}, \\ \underbrace{E_1 \rightarrow . E_1 * E_2}, \underbrace{E_1 \rightarrow . E_2}, \underbrace{E_2 \rightarrow . - E_2}, \underbrace{E_2 \rightarrow . (E)}, \underbrace{E_2 \rightarrow . \text{num}}, \underbrace{E_2 \rightarrow . \text{id}} \}$$

$$I_{29} = \{ \underbrace{E_S \rightarrow E, E_S?}_{\{\}\}}$$

Nell'immagine precedente sono illustrati i 30 stati dell'automa relativo al parser SLR. Osserviamo che, per ogni kernel item, il Follow set della testa della regola è uguale all'insieme dei relativi lookahead tokens. Ne consegue che le tabelle di parsing dei parser SLR e LALR coincidono, e che quindi le rispettive esecuzioni producono gli stessi effetti.

Si notino, nell'immagine, i lookahead tokens in corrispondenza dei kernel items; questi ultimi sono sottolineati in modo da distinguerli.

Nel seguito mostriamo l'esecuzione del parsing bottom-up sull'input proposto.

Simulazione del Parsing sull'input

STACK	num * id + num rel id & pred (num + id) & id id rel num \$	ACTION
0	↑	Shift 10
0 10	↑	Reduce E2 → num
0 2	↑	Reduce E1 → E2
0 3	↑	Shift 17
0 3 17	↑	Shift 11
0 3 17 11	↑	Reduce E2 → id
0 3 17 22	↑	Reduce E1 → E1 * E2
0 3	↑	Reduce E → E1
0 1	↑	Shift 19
0 1 19	↑	Shift 10
0 1 19 10	↑	Reduce E2 → num
0 1 19 2	↑	Reduce E1 → E2
0 1 19 20	↑	Reduce E → E + E1
0 1	↑	Shift 18
0 1 18	↑	Shift 11
0 1 18 11	↑	Reduce E2 → id
0 1 18 2	↑	Reduce E1 → E2
0 1 18 3	↑	Reduce E → E1
0 1 18 21	↑	Reduce C1 → E rel E
0 4	↑	Reduce C → C1
0 5	↑	Shift 16
0 5 16	↑	Shift 7
0 5 16 7	↑	Shift 14
0 5 16 7 14	↑	Shift 10
0 5 16 7 14 10	↑	Reduce E2 → num
0 5 16 7 14 2	↑	Reduce E1 → E2
0 5 16 7 14 3	↑	Reduce E → E1
0 5 16 7 14 24	↑	Shift 19
0 5 16 7 14 24 19	↑	Shift 11
0 5 16 7 14 24 19 11	↑	Reduce E2 → id
0 5 16 7 14 24 19 2	↑	Reduce E1 → E2
0 5 16 7 14 24 19 20	↑	Reduce E → E + E1
0 5 16 7 14 24	↑	Reduce Es → E
0 5 16 7 14 25	↑	Shift 27
0 5 16 7 14 25 27	↑	Reduce C1 → pred (Es)
0 5 16	↑	
ERROR RECOVERY		
0 5	↑	Shift 16
0 5 16	↑	Shift 11
0 5 16 11	↑	
ERROR RECOVERY		
0 5	↑	
0 5	↑	
0 5	↑	Accept

Notiamo che, in due momenti del parsing dell'input, si rende necessario adottare una strategia di error recovery. La strategia utilizzata è il Panic Mode, il cui funzionamento è illustrato nel seguito.

Scegliamo un nonterminale N della grammatica, e procediamo facendo il *pop* dallo stack fintantoché, detto X lo stato correntemente in cima allo stack, $goto[X, N] = Y$ non sia definita, e poniamo Y in cima allo stack, senza tuttavia eliminarvi X . Successivamente scartiamo simboli in input fino a che non ne incontriamo uno appartenente al Follow set del nonterminale N scelto: a questo punto proseguiamo normalmente con il parsing.

Nel nostro caso quindi, nella prima delle due occorrenze di errore, lo stack è formato dagli stati 0, 5, 16 (16 in cima) e abbiamo scelto il nonterminale C . Iniziamo eliminando il 16: osserviamo che $goto[5, C]$ non è definito e di conseguenza eliminiamo anche il 5. Ora al top dello stack abbiamo lo 0, e osserviamo che $goto[0, C]$ è definito ed è 5. Poniamo quindi 5 in cima allo stack; essendo il simbolo $\&$, correntemente in input, appartenente al Follow set di C , proseguiamo con il parsing senza operare eliminazioni di simboli dall'input.

Nel secondo caso di intervento della error recovery, lo stack è formato dagli stati 0, 5, 16, 11 (11 in cima) e anche in questo caso abbiamo scelto il nonterminale C . Eliminiamo dallo stack, nell'ordine, gli stati 11, 16 e 5 non essendo per essi definita la *goto* con il nonterminale C . Ora, come visto nel caso precedente, $goto[0, C] = 5$: poniamo 5 in cima allo stack e procediamo scorrendo l'input rimanente eliminandovi i simboli *id*, *rel*, *num* in quanto non facenti parte del Follow set di C . Il simbolo $\$$ invece appartiene a tale insieme, e quindi ora il parsing può riprendere, terminando immediatamente in quanto la Action corrispondente allo stato 5 per questo input è *Accept*.

Esercizio 9

La Grammatica

La grammatica da noi parsata è la seguente:

```

⟨Prog⟩ ::= ⟨ListComm⟩

⟨Decl⟩ ::= var ⟨ListLIdent⟩ : ⟨Type⟩ ;
           | var ⟨ListLIdent⟩ : ⟨Type⟩ = ⟨RExpr⟩ ;
           | def ⟨LIdent⟩ (⟨ListParam⟩) : ⟨Type⟩ ;
           | def ⟨LIdent⟩ (⟨ListParam⟩) : ⟨Type⟩ = ⟨Comm⟩ ;
           | def ⟨LIdent⟩ (⟨ListParam⟩) ;
           | def ⟨LIdent⟩ (⟨ListParam⟩) = ⟨Comm⟩ ;
           | val ⟨ListLIdent⟩ : ⟨Type⟩ = ⟨RExpr⟩ ;

```

```

⟨Type⟩ ::= Int
          | Float
          | Char
          | String
          | Boolean
          | Pointer [ ⟨Type⟩ ]
          | Array [ ⟨Type⟩ , ⟨Integer⟩ ]
⟨Param⟩ ::= ⟨PassType⟩ ⟨LIdent⟩ : ⟨Type⟩
          | ⟨LIdent⟩ : ⟨Type⟩
⟨PassType⟩ ::= value
              | valres
              | const
⟨RExpr⟩ ::= ⟨RExpr⟩ + ⟨RExpr⟩
          | ⟨RExpr⟩ - ⟨RExpr⟩
          | ⟨RExpr⟩ * ⟨RExpr⟩
          | ⟨RExpr⟩ / ⟨RExpr⟩
          | ⟨RExpr⟩ ^ ⟨RExpr⟩
          | ⟨RExpr⟩ % ⟨RExpr⟩
          | ( ⟨RExpr⟩ )
          | - ⟨RExpr⟩
          | ⟨LExpr⟩
          | ⟨LIdent⟩ ( ⟨ListRExpr⟩ )
          | & ⟨LExpr⟩
          | ⟨RExpr⟩ < ⟨RExpr⟩
          | ⟨RExpr⟩ > ⟨RExpr⟩
          | ⟨RExpr⟩ == ⟨RExpr⟩
          | ⟨RExpr⟩ <= ⟨RExpr⟩
          | ⟨RExpr⟩ >= ⟨RExpr⟩
          | ⟨RExpr⟩ != ⟨RExpr⟩
          | ⟨RExpr⟩ && ⟨RExpr⟩
          | ⟨RExpr⟩ || ⟨RExpr⟩
          | ! ⟨RExpr⟩
          | ( ⟨Type⟩ ) ⟨RExpr⟩
          | ⟨Basic⟩
⟨Basic⟩ ::= ⟨Integer⟩
          | ⟨Double⟩
          | ⟨Char⟩
          | ⟨String⟩
          | ⟨Boolean⟩
⟨Boolean⟩ ::= true
          | false
⟨LExpr⟩ ::= ⟨LIdent⟩
          | ⟨LExpr⟩ [ ⟨RExpr⟩ ]
          | * ⟨RExpr⟩

```

```

⟨Comm⟩ ::= ⟨CommSing⟩
          | { ⟨ListComm⟩ }

⟨CommSing⟩ ::= ⟨LIdent⟩ : ⟨CommSing⟩
              | if (⟨RExpr⟩) ⟨Comm⟩
              | if (⟨RExpr⟩) ⟨Comm⟩ else ⟨Comm⟩
              | while (⟨RExpr⟩) ⟨Comm⟩
              | for (⟨Asgn⟩ ; ⟨RExpr⟩ ; ⟨Asgn⟩) ⟨Comm⟩
              | ⟨Asgn⟩ ;
              | ⟨LIdent⟩ (⟨ListRExpr⟩) ;
              | ⟨Decl⟩
              | goto ⟨LIdent⟩ ;
              | return ⟨RExpr⟩ ;
              | return ;
              | break ;
              | continue ;

⟨Asgn⟩ ::= ⟨LExpr⟩ = ⟨RExpr⟩

⟨ListParam⟩ ::= ε
              | ⟨Param⟩
              | ⟨Param⟩ , ⟨ListParam⟩

⟨ListRExpr⟩ ::= ε
              | ⟨RExpr⟩
              | ⟨RExpr⟩ , ⟨ListRExpr⟩

⟨ListLIdent⟩ ::= ε
              | ⟨LIdent⟩
              | ⟨LIdent⟩ , ⟨ListLIdent⟩

⟨ListComm⟩ ::= ε
              | ⟨Comm⟩ ⟨ListComm⟩

```

Type System

Il nostro type system utilizza 2 ambienti, uno per le variabili V e uno per le label L . L'ambiente Γ è l'unione dei due.

Le regole il cui nome comincia per Lab hanno un solo compito: portare su l'ambiente delle label. Queste regole non hanno bisogno dell'ambiente. Una volta che si è determinato l'insieme delle label esistenti in una certa porzione del programma, possono essere applicate le altre regole. In questo modo si può saltare a label non ancora definite. Durante l'analisi di semantica statica la propagazione delle label è leggermente diversa per poter generare messaggi d'errore più significativi. L'operatore \oplus_d compone ambienti disgiunti, generando un errore in caso in cui l'intersezione tra i due sia non vuota. L'operatore \oplus_s unisce gli ambienti effettuando lo shadowing, preferendo gli elementi del secondo. L'ambiente ci permette di sapere il tipo di passaggio di una variabile e il suo tipo.

Il vincolo di costante è considerato uno dei possibili tipi di passaggio. Inoltre l'ambiente ci permette di sapere se siamo in un loop.

Le regole per il type system sono le seguenti:

Un programma è ben formato se dopo aver raccolto le label con le apposite regole si è in grado di dedurre che con tali labels nell'ambiente, la lista di comandi che lo forma è ben formata. $V_{starting}$ è l'ambiente che contiene tutte le funzioni predefinite.

$$(Prog) \frac{\vdash_{LabListComm} listComm : L \quad \langle L, V_{starting} \rangle \vdash_{ListComm} listComm : \Delta}{\langle \emptyset, V_{starting} \rangle \vdash_{Prog} listComm}$$

Le labels che compongono una lista di comandi si ricavano facendo le unioni delle labels contenute in ogni comando.

$$(LabListComm) \frac{\vdash_{LabComm} comm : L_1 \quad \vdash_{LabListComm} listComm : L_2}{\vdash_{LabListComm} comm; listComm : L_1 \oplus_d L_2}$$

$$(LabListCommEmpty) \frac{}{\vdash_{LabListComm} [] : \emptyset}$$

Un comando può essere un comando singolo o un blocco di comandi. Nel caso sia un blocco di comandi le labels interne non vengono esportate.

$$(LabCmdSing) \frac{\vdash_{LabCommSing} commSing : L}{\vdash_{LabComm} commSing : L}$$

$$(LabCmdMult) \frac{}{\vdash_{LabComm} listComm : \emptyset}$$

Un comando singolo può avere una label. In tal caso la prossima regola la estrae. La seconda regola si applica a ogni comando senza label

$$(LabCommSingWLabel) \frac{\vdash_{LabCommSing} commSing : L}{\vdash_{LabCommSing} \text{label } lab : commSing : L \oplus_d \{lab\}}$$

$$(LabCommSingWOLabel) \frac{}{\vdash_{LabCommSing} commSingwithoutlabel : \emptyset}$$

Queste sono tutte le regole necessarie per propagare le labels. Ora seguono le rimanenti regole.

Una lista di comandi è ben formata se il comando che inizia tale lista è ben formato e se la lista di comandi rimanenti con le modifiche all'ambiente effettuate dal primo comando è ben formata. Una lista vuota è sempre ben formata.

$$(ListComm) \frac{\Gamma \vdash_{Comm} comm : \Gamma_1 \quad \Gamma_1 \vdash_{ListComm} listComm : \Gamma_2}{\Gamma \vdash_{ListComm} comm; listComm : \Gamma_2}$$

$$(ListCommEmpty) \frac{}{\Gamma \vdash_{ListComm} [] : \Gamma}$$

Un comando può essere un comando singolo o un blocco di comandi. Un blocco di comandi è ben formato se la lista di comandi che lo compone è ben formata. L'ambiente locale non viene esportato.

$$(CmdSing) \frac{\Gamma \vdash_{CommSing} commSing : \Gamma_1}{\Gamma \vdash_{Comm} commSing : \Gamma_1}$$

$$(CmdMult) \frac{\vdash_{LabListComm} listComm : L_1 \quad \langle L_1 \oplus_d L_0, V \rangle \vdash_{ListComm} listComm : \Gamma_1}{\langle L_0, V \rangle \vdash_{Comm} listComm : \langle L_0, V \rangle}$$

Ora seguono le regole per i vari tipi di comandi.
Un comando con label è semanticamente equivalente a uno senza label.

$$(CmdWLabel) \frac{\Gamma \vdash_{CommSing} commSing : \Gamma_1}{\Gamma \vdash_{CommSing} \text{label } lab : commSing : \Gamma_1}$$

Un *If* è corretto se la guardia è booleana e se il corpo è corretto.

$$(CmdIf) \frac{\Gamma \vdash_{RExpr} guard : Boolean \quad \Gamma \vdash_{Comm} body : \Gamma_1}{\Gamma \vdash_{CommSing} \text{if } (guard) body : \Gamma}$$

$$(CmdIfElse) \frac{\Gamma \vdash_{RExpr} guard : Boolean \quad \Gamma \vdash_{Comm} body : \Gamma_1 \quad \Gamma \vdash_{Comm} elseBody : \Gamma_2}{\Gamma \vdash_{CommSing} \text{if } (guard) body \text{ else } elseBody : \Gamma}$$

Un *While* è ben formato se la guardia è booleana e se il corpo è ben formato assumendo che sia all'interno di un loop.

Un *for* è ben formato se lo sono gli assegnamenti, se la guardia è booleana e se il corpo, assumendo che sia in un loop, è ben formato.

$$(CmdWhile) \frac{\Gamma \vdash_{RExpr} guard : Boolean \quad (\Gamma \oplus_s \{isInLoop : true\}) \vdash_{Comm} body : \Gamma_1}{\Gamma \vdash_{CommSing} \text{while } (guard) body : \Gamma}$$

$$(CmdFor) \frac{\begin{array}{c} \Gamma \vdash_{Asgn} init \quad \Gamma \vdash_{RExpr} guard : Boolean \\ \Gamma \vdash_{Asgn} post \quad (\Gamma \oplus_s \{isInLoop : true\}) \vdash_{Comm} body : \Gamma_1 \end{array}}{\Gamma \vdash_{CommSing} \text{for } (init; guard; post) body : \Gamma}$$

Un assegnamento è ben formato se i tipi sono compatibili e se la *LExpr* non ha il vincolo di essere costante. Per assegnare un *Int* ad un *Float* questa regola va usata assieme alla regola di conversione implicita. L'ambiente ci permette di sapere se una *LExpr* o una *RExpr* ha il vincolo di costante. Una *RExpr* ha il vincolo di costante se e solo se è una *LExpr* che ha il vincolo di costante.

$$(CmdAsgn) \frac{\Gamma \vdash_{Asgn} assign}{\Gamma \vdash_{CommSing} assign : \Gamma}$$

$$(Assign) \frac{\Gamma \vdash_{LExpr} lExpr : \tau \quad \Gamma \vdash_{RExpr} rExpr : \tau}{\Gamma \vdash_{Asgn} lExpr = rExpr}$$

where *lExpr* non ha il vincolo di costante

Una chiamata di procedura è ben formata se i tipi degli argomenti sono coerenti con i quelli dei parametri formali. Nel caso in cui un parametro sia passato per valres, allora tale parametro attuale deve essere una *RExpr* che è una *LExpr* senza vincolo di costante e in questo caso non può essere effettuato il cast implicito.

$$(CmdProc) \frac{\begin{array}{c} \Gamma \vdash_{RExpr} arg_i : \tau_i \quad \forall i \in 1 \dots n \\ \Gamma \vdash_{LExpr} proc : Function(\tau_1, \dots, \tau_i, \dots, \tau_n) \rightarrow \tau_r \end{array}}{\Gamma \vdash_{CommSing} proc(arg_1, \dots, arg_n) : \Gamma}$$

where se *arg_i* è passato per valres → *arg_i* è una *LExpr*, non ha il vincolo di costante ed è esattamente di tipo *τ_i* (non viene eseguito il cast implicito)

Un comando può essere anche una dichiarazione.

$$(CmdDecl) \frac{\Gamma \vdash_{Decl} decl : \Gamma_1}{\Gamma \vdash_{CommSing} decl : \Gamma_1}$$

Un comando *goto* è ben formato se la label a cui si riferisce esiste ed è visibile.

$$(CmdGoTo) \frac{}{\langle L, V_{starting} \rangle \vdash_{CommSing} goto label : \langle L, V_{starting} \rangle}$$

where *label* ∈ *L*

Un comando *return* è ben formato se siamo all'interno del corpo di una definizione di funzione o di procedura e il tipo ritornato è coerente con il tipo di ritorno della funzione.

$$(CmdReturnFun) \frac{\begin{array}{c} \Gamma \vdash_{LExpr} fun : Function(\tau_1, \dots, \tau_n) \rightarrow \tau_r \quad \Gamma \vdash_{RExpr} value : \tau_r \\ \Gamma \vdash_{CommSing} return value : \Gamma \end{array}}{\Gamma \vdash_{CommSing} return value : \Gamma}$$

where Siamo nel corpo della funzione *fun*

$$(CmdReturnProc) \frac{\Gamma \vdash_{LExpr} proc : Function(\tau_1, \dots, \tau_n) \rightarrow Void}{\Gamma \vdash_{CommSing} return : \Gamma}$$

where Siamo nel corpo della funzione *proc*

Un *break* o un *continue* è ben formato se siamo all'interno di un ciclo.

$$(CmdBreak) \frac{}{\Gamma \vdash_{CommSing} \text{break} : \Gamma} \\ \text{where Siamo in un loop}$$

$$(CmdContinue) \frac{}{\Gamma \vdash_{CommSing} \text{continue} : \Gamma} \\ \text{where Siamo in un loop}$$

Le seguenti sono le regole per le dichiarazioni. Queste regole inseriscono nell'ambiente gli elementi dichiarati. Nella nostra grammatica ammettiamo dichiarazioni di multiple variabili in una sola dichiarazione. Per semplicità queste dichiarazioni non saranno incluse nel type system. Inoltre ammettiamo anche le dichiarazioni di prototipi per poter definire funzioni mutuamente ricorsive.

Una dichiarazione di variabile è sempre ben formata se non si effettua l'inizializzazione delle variabili dichiarate. Quando si effettua l'inizializzazione delle variabili, una dichiarazione è ben formata se il tipo dichiarato è consistente con quello dell'espressione di inizializzazione.

$$(DeclVar) \frac{}{\Gamma \vdash_{Decl} (\text{var } v : \tau) : \Gamma \oplus_s \{v : \tau\}}$$

$$(DeclVarInit) \frac{\Gamma \vdash_{RExpr} \text{init} : \tau}{\Gamma \vdash_{Decl} (\text{var } v : \tau = \text{init}) : \Gamma \oplus_s \{v : \tau\}}$$

$$(DeclValInit) \frac{\Gamma \vdash_{RExpr} \text{init} : \tau}{\Gamma \vdash_{Decl} (\text{var } c : \tau = \text{init}) : \Gamma \oplus_s \{c : \tau\}} \\ \text{where } c \text{ è inserita nell'ambiente come avente il vincolo di costante}$$

Nel caso di una dichiarazione di funzione o di procedura, il corpo va valutato dopo aver aggiunto all'ambiente i parametri formali (con il passaggio del tipo giusto) e la funzione stessa.

$$(DeclFun) \frac{}{\Gamma \vdash_{Decl} (\text{def } fun(\text{arg}_1 : \tau_1, \dots, \text{arg}_n : \tau_n) : \tau_r) : \Gamma \oplus_s \{fun : \text{Function}(\tau_1, \dots, \tau_n) \rightarrow \tau_r\}} \\ \text{where I parametri formali sono inseriti nell'ambiente assieme al modo in cui sono passati}$$

$$(DeclFunInit) \frac{\Gamma \oplus_s \{\text{arg}_1 : \tau_1, \dots, \text{arg}_n : \tau_n, fun : \text{Function}(\tau_1, \dots, \tau_n) \rightarrow \tau_r, \text{isInLoop} : \text{false}\} \vdash_{Comm} \text{body} : \Gamma_1}{\Gamma \vdash_{Decl} (\text{def } fun(\text{arg}_1 : \tau_1, \dots, \text{arg}_n : \tau_n) : \tau_r = \text{body}) : \Gamma \oplus_s \{fun : \text{Function}(\tau_1, \dots, \tau_n) \rightarrow \tau_r\}} \\ \text{where I parametri formali sono inseriti nell'ambiente assieme al modo in cui sono passati}$$

$$(DeclProc) \frac{\Gamma \vdash_{Decl} (\text{def } proc(arg_1 : \tau_1, \dots, arg_n : \tau_n)) : \Gamma \oplus_s \{proc : Function(\tau_1, \dots, \tau_n) \rightarrow Void\}}{\text{where I parametri formali sono inseriti nell'ambiente assieme al modo in cui sono passati}}$$

$$(DeclProcInit) \frac{\Gamma \oplus_s \{arg_1 : \tau_1, \dots, arg_n : \tau_n, proc : Function(\tau_1, \dots, \tau_n) \rightarrow Void, isInLoop : false\} \vdash_{Comm} body : \Gamma_1}{\Gamma \vdash_{Decl} (\text{def } proc(arg_1 : \tau_1, \dots, arg_n : \tau_n) = body) : \Gamma \oplus_s \{proc : Function(\tau_1, \dots, \tau_n) \rightarrow Void\}}$$

where I parametri formali sono inseriti nell'ambiente assieme al modo in cui sono passati

Seguono ora le regole per le LExpr.

$$(LExpId) \frac{}{\Gamma, x : \tau \vdash_{LE} x : \tau}$$

$$(LExpArr) \frac{\Gamma \vdash_{RExpr} i : Int \quad \Gamma \vdash_{LExpr} a : Array(\tau, size)}{\Gamma \vdash_{LE} a[i] : \tau}$$

where Se a ha il vincolo di costante, anche $a[i]$ lo avrà.

$$(LExpDeref) \frac{\Gamma \vdash_{RExpr} v : Pointer(\tau)}{\Gamma \vdash_{LE} *v : \tau}$$

Per finire, le regole per le RExpr.

$$(REBool) \frac{\Gamma \vdash_{RExpr} x : Boolean \quad \Gamma \vdash_{RExpr} y : Boolean \quad op \in \{\&\&, ||\}}{\Gamma \vdash_{RExpr} x op y : Boolean}$$

$$(RERelOpInt) \frac{\Gamma \vdash_{RExpr} x : Int \quad \Gamma \vdash_{Int} y : Float \quad op \in \{>, <, \leq, \geq\}}{\Gamma \vdash_{RExpr} x op y : Boolean}$$

$$(RERelOpFloat) \frac{\Gamma \vdash_{RExpr} x : Float \quad \Gamma \vdash_{RExpr} y : Float \quad op \in \{>, <, \leq, \geq\}}{\Gamma \vdash_{RExpr} x op y : Boolean}$$

$$(REArithOpFloat) \frac{\Gamma \vdash_{RExpr} x : Float \quad \Gamma \vdash_{RExpr} y : Float \quad op \in \{+, -, *, /, ^\}}{\Gamma \vdash_{RExpr} x op y : Float}$$

$$(REArithOpInt) \frac{\Gamma \vdash_{RExpr} x : Int \quad \Gamma \vdash_{RExpr} y : Int \quad op \in \{+, -, *, /, ^, \% \}}{\Gamma \vdash_{RExpr} x op y : Int}$$

$$(REEqu) \frac{\Gamma \vdash_{RExpr} x : \tau \quad \Gamma \vdash_{RExpr} y : \tau \quad op \in \{==, !=\}}{\Gamma \vdash_{RExpr} x op y : Boolean}$$

$$(RENegFloat) \frac{\Gamma \vdash_{RExpr} x : Float}{\Gamma \vdash_{RExpr} -x : Float}$$

$$(RENegInt) \frac{\Gamma \vdash_{RExpr} x : Int}{\Gamma \vdash_{RExpr} -x : Int}$$

$$(RENegBool) \frac{\Gamma \vdash_{RExpr} x : Boolean}{\Gamma \vdash_{RExpr} !x : Boolean}$$

$$(REExpr) \frac{\Gamma \vdash_{LExpr} x : \tau}{\Gamma \vdash_{RExpr} x : \tau}$$

$$(REFun) \frac{\begin{array}{c} \Gamma \vdash_{RExpr} arg_i : \tau_i \quad \forall i \in 1 \dots n \\ \Gamma \vdash_{LExpr} fun : Function(\tau_1, \dots, \tau_i, \dots, \tau_n) \rightarrow \tau_r \end{array}}{\Gamma \vdash_{RExpr} fun(arg_1, \dots, arg_n) : \tau_r}$$

where se arg_i è passato per valres $\rightarrow arg_i$ è una LExpr, non ha il vincolo di costante ed è esattamente di tipo τ_i (non viene eseguito il cast implicito)

$$(RERef) \frac{\Gamma \vdash_{LExpr} x : \tau}{\Gamma \vdash_{RExpr} &x : Pointer(\tau)}$$

Per quanto riguarda i tipi base, l'unico cast implicito è il seguente, tra Int e Float.

$$(Spec) \frac{\Gamma \vdash_{RExpr} x : Int}{\Gamma \vdash x_{RExpr} : Float}$$

$$(RECast) \frac{\Gamma \vdash_{RExpr} x : \tau}{\Gamma \vdash_{RExpr} (\tau_1)x : \tau_1}$$

where $(\tau_1, \tau) \in \{ (Int, Float), (Float, Int), (Char, Int), (Int, Char), (Float, Char), (Char, Float), (String, Char) \}$

Environment

L'environment è stato implementato attraverso il tipo `Env`, definito come mappa dove la chiave è una stringa e l'elemento è `EnvElem`.

Ogni `EnvElem` è caratterizzato da un identificatore, un tipo, un tipo di passaggio e un indirizzo di memoria.

L'environment tiene traccia di tutte le definizioni di variabili, costanti, procedure, funzioni e parametri formali. Ogni nodo del parse tree è dotato dei due attributi `envIn` ed `envOut`. La funzione di `envIn` è di contenere l'environment che vale per quel nodo, mentre quella di `envOut` è di contenere il nuovo environment, eventualmente modificato dal nodo stesso.

Nel caso tipico, l'`envIn` di un nodo viene generato dal nodo genitore e l'`envOut` di un nodo viene sintetizzato dagli `envOut` dei nodi figli. Questo permette ad esempio, in una lista di comandi, di passare l'environment attraverso questi ultimi incrementandolo gradualmente.

È importante puntualizzare che nel caso vi sia un blocco (anonimo, definizione di funzione, corpo dell'`if`) l'environment di uscita è uguale a quello di ingresso dato che le definizioni effettuate nel blocco valgono solamente nel blocco.

All'inizio l'environment è inizializzato con la funzione `initEnv` che restituisce un environment contenente le funzioni predefinite di scrittura e lettura richieste.

Funzioni e procedure

Le procedure sono definite come funzioni aventi come tipo di ritorno il tipo Void. L'overloading non è stato implementato e quindi in ogni environment vi può essere un solo elemento avente un determinato nome.

Dato che l'environment è incrementale, per ammettere la mutua ricorsione è stato necessario implementare i prototipi di funzione e procedura. I prototipi, nell'environment, hanno pass-type **PassValue**, e si distinguono dalle definizioni di funzioni con corpo dato che il loro pass-type è **PassConst**. Quando viene definita una funzione per la quale esisteva già un prototipo, se il tipo della funzione è uguale a quello del prototipo viene utilizzato il vecchio indirizzo di memoria (quello del prototipo), altrimenti la funzione appena definita viene considerata una nuova funzione che prende il posto del prototipo e viene quindi generato un nuovo indirizzo di memoria.

Environment per le Labels

L'environment delle label si comporta in maniera simile all'environment delle definizioni, l'unica differenza è che una label deve essere visibile anche dai comandi che precedono il comando che la definisce, e non solo in seguito alla definizione. Come nel caso dell'environment generale, una label creata all'interno di un blocco vale solamente in quel blocco ed in eventuali blocchi interni ad esso.

Per implementare questo meccanismo sono state utilizzati tre attributi: **labelEnvIn**, **labelEnvOut**, **labelEnv**.

L'attributo **labelEnvIn** in un nodo rappresenta l'environment delle etichette definite fino a quel punto. L'attributo **labelEnvOut** in un nodo rappresenta il nuovo environment delle etichette eventualmente modificato dal nodo stesso e da eventuali figli con l'aggiunta di nuove etichette. L'attributo **labelEnv** rappresenta invece l'environment definitivo delle etichette valide per quel nodo. Vengono dunque collezionate tutte le label in un blocco utilizzando gli attributi **labelEnvIn** e **labelEnvOut**. Una volta raccolte tutte le labels, il nodo che rappresenta il blocco imposta l'attributo **labelEnv**, che viene passato top-down.

Tipi

Ogni nodo del parse tree è dotato di un tipo indicato nell'attributo **tipo**. Per le r-expr e le l-expr il contenuto di **tipo** è semplicemente il tipo dell'espressione, mentre per tutti gli altri nodi il tipo è Void.

I tipi vengono controllati per generare eventuali errori. Per via della laziness, il tipo di ogni nodo deve essere definito in base al tipo dei nodi precedenti, per far sì che un eventuale errore venga effettivamente valutato.

Sono stati implementati i vari tipi base più i tipi composti, come il puntatore **TypePointer** che è definito a partire da un altro tipo, e il tipo **TypeArray** che è definito a partire da un altro tipo ed un intero che ne rappresenta la dimensione. Vi è inoltre il tipo **TypeFunc** che viene utilizzato solo nel salvataggio delle de-

finizione di funzione e procedura nell'environment e quindi non viene mai parsato.

Status

Il dato **Stat** è stato definito per mantenere le informazioni, utili per la generazione del TAC, relative alle label ausiliarie generate, allo stato della memoria e ai registri temporanei.

Sono state inoltre definite delle funzioni che restituiscono nuove etichette, locazioni di memoria temporanee e non temporanee ed il nuovo stato che tiene conto della modifica allo stato della memoria.

Per dettagli implementativi si rimanda al codice commentato.

Ogni nodo è dotato dell'attributo **statusIn** che rappresenta lo stato corrente del nodo e **statusOut** che rappresenta il nuovo stato eventualmente modificato dal nodo stesso. L'uso tipico è di impostare lo **statusIn** dei nodi sottostanti in base al proprio **statusIn** e definire il proprio **statusOut** in base allo **statusOut** dei nodi sottostanti.

All'inizio lo stato è impostato dalla funzione **initStat** che tiene conto dello spazio impiegato per gli indirizzi utilizzati per le funzioni predefinite.

Booleani

La valutazione delle r-expr booleane è tale da supportare la cortocircuitazione. Per implementare questo, ogni r-expr è dotata degli attributi **true** e **false**. Tali attributi contengono le label alla quale saltare nel caso l'espressione restituisca *true* oppure *false*. Per esempio considerando la r-expr $A \parallel B$, se A valuta *true*, allora si salta direttamente alla label indicata nell'attributo **true** che ci porterà alla prossima istruzione, altrimenti se A valuta *false* si salta alla label salvata nell'attributo **false** che ci farà valutare anche B . Se però la r-expr booleana viene usata come valore e non in una guardia, allora bisogna scrivere del codice ausiliario che usa tali salti per produrre il valore necessario.

Per ulteriori dettagli si rimanda al codice commentato.

Three Address Code

Abbiamo assunto le seguenti specifiche per il TAC, oltre a quelle classiche:

- L'operatore di dereference può essere utilizzato più volte in sequenza (es $**_x$)
- L'istruzione **return** salta fuori dalla funzione a cui si riferisce, quindi non è necessario inserire una ulteriore istruzione di salto.

Abbiamo inoltre scelto di valutare le espressioni e gli argomenti da sinistra verso destra in quanto a nostro parere risulta più naturale.