RELAZIONE DEL PROGETTO DI LINGUAGGI E COMPILATORI
(PARTE 4 GRUPPO 6)
di
Andrea ******, matricola n. *****
Massimiliano De Luise, matricola n. ******
Massimo *******, matricola n. ******

Indice

1 Convenzioni utilizzate nella relazione

Di seguito con il nome **MINE** (*Mine Is Not E*) si farà riferimento al linguaggio creato a partire dalla sintassi concreta del linguaggio **E**.
Nella descrizione della grammatica i non-terminali sono racchiusi tra $\langle$ e $\rangle$, i simboli ::= (produzione), | (unione) e ϵ (regola vuota) appartengono alla notazione di BNFC, tutti gli altri simboli sono terminali.

2 Struttura generale del progetto

Nella cartella relativa all'esercizio del progetto sono presenti i seguenti file:

- grammar.cf − sorgente di BNFC relativo alla parte parziale del progetto
- new_grammar.cf − sorgente di BNFC relativo alla parte finale del progetto
- AbsGrammar.hs − modulo haskell contenente la grammatica del linguaggio MINE
- LexGrammar.x − sorgente del lexer Alex per il lexing del linguaggio MINE
- ParGrammar.y − sorgente del parser Happy per il parsing del linguaggio MINE
- Printer.hs − sorgente che crea l'eseguibile da utilizzare per il *pretty printer*
- TypeChecking.hs − modulo haskell che implementa il typechecking
- TACGen.hs − modulo haskell che implementa la geenrazione di codice intermedio
- Type.hs − modulo haskell contenente tipi di dato comuni tra vari file
- SkelGrammar.hs e PrintGrammar.hs − moduli haskell necessari al funzionamento del progetto

Oltre ai file riportati sopra, è presente una cartella testcases contenente dei test effettuati durante lo sviluppo di MINE.

3 Linguaggio

3.1 Sintassi di base

Il linguaggio creato ha la stessa sintassi concreta del linguaggio **E**.
Un programma definito in questo linguaggio equivale ad una sequenza di comandi all'interno di un blocco (sezione di codice racchiuso tra due parentesi graffe). I singoli comandi sono separati dal carattere *newline*.
Riguardo la sintassi di MINE è importante tenere in considerazione i seguenti punti:

- **Case Sensitivity** − MINE è case sensitive, il che significa che l'identificativo **Hello** e **hello** sono differenti tra loro
- **Nomi** − i nomi delle variabili, delle costanti, delle funzioni e delle procedure devono iniziare con una lettera e possono contenere soltanto caratteri alfanumerici e i caratteri _ e '. Inoltre i nomi non possono corrispondere a parole chiave del linguaggio.

3.2 Parole riservate

Le parole riservate, usate all'interno del linguaggio, sono le seguenti:

String	boolean	break	char
continue	def	else	false
float64	if	int	ref
return	true	val	valres
var	void	while	for
in	const	do	switch
match	checked		

3.3 Commenti

I commenti mono-linea cominciano con #.
I commenti multi-linea sono racchiusi tra `/*` e `*/`.

3.4 Spazi bianchi

Una linea contenente soltanto caratteri di spaziatura, o una linea commentata, viene totalmente ignorata da MINE

3.5 Type system

Il linguaggio ha i seguenti tipi base predefiniti: void, boolean, char, int, float64 e String. Sono presenti inoltre i tipi t[n] (array di dimensione n i cui elementi sono di tipo t) e t ~ (puntatore ad un dato di tipo t).
Valgono le seguenti compatibilità:

- Il tipo int è compatibile con il tipo float64
- Se il tipo t è compatibile con il tipo t' allora per ogni n si ha che il tipo t[n] è compatibile con il tipo t'[n]

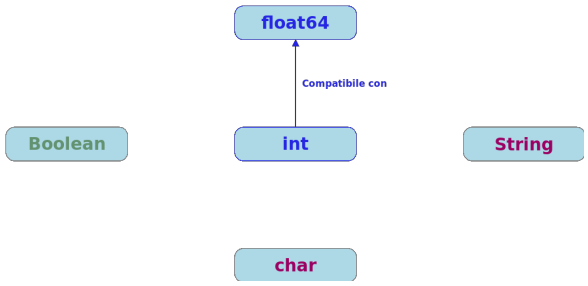

Figura 1: Compatibilità tra i tipi base presenti in MINE

3.6 Operatori di base

Gli operatori matematici sono utlizzati nelle espressioni matematiche nello stesso modo col quale vengono utilizzati nell'algebra. La seguente tabella elenca gli operatori matematici presenti.
Si assuma che la variable A abbia valore 2 e la variabile B abbia valore 3.

Operatore	Descrizione	Esempio
+ (Addizione)	Somma i valori presenti ai lati dell'operatore	A + B restituirà 5
- (Sottrazione)	Sottrae il valore di destra da quello a sinistra dell'operatore	A - B restituirà -1
* (Moltiplicazione)	Moltiplica i valori presenti ai lati dell'operatore	A * B restituirà 6
/ (Divisione)	Divide il valore di sinistra per quello a destra dell'operatore	A / B restituirà 2 0
% (Modulo)	Divide il valore di sinistra per quello a destra dell'operatore e restituisce il resto	A % B restituirà 2
** (Potenza)	Eleva a potenza il valore di sinistra per quello a destra dell'operatore	A**B restituirà 8
++ (Incremento)	Incrementa il valore dell'operando di 1	A++ farà valere A 3
−− (Decremento)	Decrementa il valore dell'operando di 1	A−− farà valere A 1

Di seguito sono elencati gli operatori relazionali supportati da MINE.
Si assuma che la variabile A abbia valore 10 e la variabile B abbia valore 20.

Operatore	Descrizione	Esempio
== (Uguale a)	Controlla se i due operandi sono uguali o no	A == B restituirà falso
!= (Diverso da)	Controlla se i due operandi sono diversi o no	A != B restituirà vero
>(Maggiore di)	Controlla se l'operando sinistro è maggiore dell'operando di destra o no	A >B restituirà falso
<(Minore di)	Controlla se l'operando sinistro è minore dell'operando di destra o no	A <B restituirà vero
>= (Maggiore o uguale a)	Controlla se l'operando sinistro è maggiore o uguale dell'operando di destra o no	A >= B restituirà falso
<= (Minore o uguale a)	Controlla se l'operando sinistro è minore o uguale dell'operando di destra o no	A <= B restituirà vero

La seguente tabella elenca gli operatori logici.
Si assuma la variabile A abbia valore true e la variabile B abbia valore false.

Operatore	Descrizione	Esempio
&& (And logico)	Restituisce l'and logico tra i due operandi	A && B è falso
|| (Or logico)	Restituisce l'or logico tra i due operandi	A || B è vero
! (Not logico)	Restituisce il not logico dell'operando	!A è falso

Di seguito gli operatori di assegnamento supportati da MINE:

Operatore	Descrizione	Esempio
:=	Assegna il valore di destra alla variabile	A := 2
+=	Assegna la somma tra il valore di destra e quello della variabile alla variabile stessa	A += 2
-=	Assegna la sottrazione tra il valore della variabile e il valore a destra alla variabile stessa	A -= 2
*=	Assegna la moltiplicazione tra il valore di destra e quello della variabile alla variabile stessa	A *= 2
/=	Assegna la divisione tra il valore della variabile e il valore a destra alla variabile stessa	A /= 2
%=	Assegna il modulo tra il valore della variabile e il valore a destra alla variabile stessa	A %= 2
**=	Assegna la potenza tra il valore della variabile e il valore a destra alla variabile stessa	A **= 2
&=	Assegna il risutato dell'and logico tra il valore della variabile e il valore a destra alla variabile stessa	A &= true
|=	Assegna il risutato dell'or logico tra il valore della variabile e il valore a destra alla variabile stessa	A |= true

È presente inoltre un operatore ternario, denominato con "?", che rappresenta il costrutto if-then-else per la categoria sintattica delle espressioni. La sintassi è la seguente:

```
{
var x:int := readInt("stdin")
var y:float64 := `x == 42` ? `10.2` ^ `4.28`
}
```

Listing 1: Operatore ternario

In esso la condizione da valutare e le espressioni risultato sono racchiuse da apici inversi (carattere ascii 96), la condizione è separata dalle espressioni dal carattere ?, e le due espressioni risultato sono divise tra loro dal carattere cappello (carattere ascii 94).

La seguente tabella riporta le precedenze e le associatività dei singoli operatori. Un operatore ha maggior precedenza rispetto ad un altro, e quindi verrà valutato prima di questo, se appare prima nella tabella.

Categoria	Operatore	Associatività
Unario (post)	++, −−	nonassoc
Unario (pre)	++, −−	nonassoc
Unario	∼	nonassoc
Unario	-	nonassoc
Binario	**	destra
Binario	*, /	sinistra
Binario	+, -, %	sinistra
Binario	==, !=, >, >=, <, <=	nonassoc
Unario	!	nonassoc
Binario	&&	sinistra
Binario	||	sinistra
Ternario	?	nonassoc

Caratteristiche degli operatori di base

Nel linguaggio MINE gli operatori di base hanno le seguenti caratteristiche:

- L'uguaglianza tra stringhe è da intendersi come uguaglianza tra caratteri e non come uguaglianza tra gli indirizzi delle stringhe. Si consideri il seguente esempio:

```
1 {
2 var x:boolean := false
3 def y:String := "hello"
4 if (y == readString("stdin")) {
5   x := true
6 }
7 }
```

Listing 2: Uguaglianza tra stringhe in MINE

  In esso la condizione dell'if alla riga 4 ha valore vero se la stringa letta da stdin risulta uguale lessograficamente a hello.

- L'operatore di incremento/decremento può essere annidato, si prenda come esempio il seguente codice:

```
1 {
2 var x:int := 10
3 var y:int := (++x++)++
4 }
```

Listing 3: Operatore inc/dec in MINE

  Al termine dell'esecuzione del codice, la variabile x conterrà il valore 13, mentre la variabile y conterrà il valore 11

- Sul tipo di dato puntatore sono definiti soltanto l'operatore di uguaglianza e l'operatore di disuguaglianza
- Sul tipo di dato char son definiti soltanto gli operatori di uguaglianza e confronto
- L'operatore di divisione se applicato a due numeri interi restituisce a sua volta un numero intero, se invece uno dei due operandi è di tipo float64, allora anche il risultato è del medesimo tipo
- L'operatore di dereferenza ha minore priorità rispetto all'accesso degli array. Si prenda come esempio il seguente codice, nella quale le parentesi tonde sono obbligatorie:

```
1 {
2 var x:int[] := [1,2,3]
3 var y:int[3]~ := &x
4 var z:int := (~y)[0]
5 }
```

Listing 4: Operatore dereferenza con accesso ad array

- L'operatore ternario ? può restituire qualsiasi tipo di dato, e non può comparire come elemento di una matrice

3.7 Cicli

MINE supporta i seguenti tipi di costrutti di cicli:

- **ciclo while** – Ripete un comando o un gruppo di comandi finchè una certa condizione è vera. La condizione viene testata prima di eseguire il body
- **ciclo do while** – Variante del ciclo while
- **ciclo for** – Ripete un comando o un gruppo di comandi un numero finito di volte

E i seguenti comandi di controllo:

- **break** – Termina il ciclo e trasferisce il controllo al comando immediatamente successivo al ciclo considerato
- **continue** – Causa un'interruzione dell'esecuzione del ciclo, viene valutata la guardia ed eventualmente ripetuto il *body*

La sintassi dei cicli è la seguente:

```
{
var x:boolean := true
while(x) {
  x := false
}
}
```

Listing 5: Esempio di un ciclo while

```
{
var x:boolean := true
do {
  x := false
} while (x)
}
```

Listing 6: Esempio di un ciclo do while

```
{
var x:int := 0
for i in 1..3 {
  x++
}
}
```

Listing 7: Esempio di un ciclo for

Nei cicli for non è possibile modificare la variabile di controllo, questa infatti è a tutti gli effetti una variabile locale al blocco definita con def. Inoltre non sono presenti cicli determinati decrescenti, ovvero cicli for nei quali la variabile di iterazione decresce ad ogni fine ciclo.

3.8 Costrutti di selezione

MINE supporta i seguenti costrutti di selezione:

- **if** – Il costrutto if consiste in un'espressione booleana seguita da uno o più comandi
- **if else** – Il costrutto if può essere seguito da un comando opzionale else, che esegue il suo *body* se la guardia è falsa
- **case** – Permette di testare l'equivalenza tra una data espressione e diversi valori

La sintassi dei costrutti di selezione è la seguente:

```
{
def x:boolean := false
def y:int := 20
if (y < 30) {
  x := true
}

def z:boolean := true
if (x && !z) {
  y := 42
} else {
  y := -1
}
}
```

Listing 8: Esempio di costrutti if

```
{
var x:int := 0
def y:char := readChar("stdin")
switch (y) {
  match 'a' { x++ }
  match 'b' {
      x := 42 }
  match _ { x := 24 }
}
}
```

Listing 9: Esempio di costrutto case

Nel costrutto case sono obbligatorie le parentesi tonde attorno all'espressione di verifica, è inoltre obbligatorio che sia presente sempre uno ed un solo ramo di default (match _), e che questo sia l'ultimo. I blocchi del costrutto sono da intendersi come comprensivi dell'istruzione di break, ciò significa che verrà eseguito sempre un solo blocco tra tutti.

3.9 Dichiarazioni

Sono possibili quattro tipi di dichiarazioni:

- Dichiarazione di costanti – indicata dalla keyword def seguita dal nome della variabile, dal carattere :, dal tipo della variabile, dal simbolo := e da un valore o da un'espressione da associare alla costante
- Dichiarazione di variabili – indicata dalla keyword var seguita dal nome della variabile, dal carattere :, dal tipo della variabile, dal simbolo := e da un valore o da un'espressione da associare alla costante
- Dichiarazione di funzioni – indicata dalla keyword def seguita dal nome della funzione, da una lista di modalità nome:tipo racchiusa tra parentesi tonde rappresentante i parametri di input della funzione, dal carattere :, dal tipo restituito dalla funzione, seguito dal body della funzione (racchiuso tra graffe) compreso del comando di return
- Dichiarazione di procedure – indicata dalla keyword def seguita dal nome della funzione, da una lista di modalità nome:tipo racchiusa tra parentesi tonde rappresentante i parametri di input della funzione, dal carattere :, dal tipo void, seguito dal body della procedura (racchiuso tra graffe) compreso (opzionalmente) del comando di return

Le modalità di passaggio dei parametri possibili sono: val, res, valres, const. È possibile omettere la modalità di passaggio, in questo caso si suppone sia val.
Più formalmente:

- Dichiarazione di costanti – indicata dalla keyword def seguita da un elemento della forma $\langle Ident \rangle$: $\langle TypeSpec \rangle := \langle ComplexRExpr \rangle$
- Dichiarazione di variabili – indicata con la keyword var seguita da un elemento della forma $\langle Ident \rangle$: $\langle TypeSpec \rangle := \langle ComplexRExpr \rangle$
- Dichiarazione di funzioni e procedure – indicata con la keyword def seguita da un elemento della forma $\langle Ident \rangle$ ($\langle ListOfParameters \rangle$) : $\langle RetType \rangle$ $\langle BlockDecl \rangle$

Si veda l'Appendice per una descrizione esaustiva della grammatica del linguaggio.

La tipizzazione utilizzata dal linguaggio MINE è statica, inoltre nella dichiarazione è sempre obbligatorio inserire il tipo. Sono possibili dichiarazioni di funzioni ricorsive e mutualmente ricorsive.

La sintassi delle dichiarazioni è la seguente:

```
{
def foo(x:int):int {
  return x**2
}
def z:int := foo(8)
}
```

Listing 10: Esempio di dichiarazioni

Array checked

È presente la possibilità di dichiarare array con una variante sintattica di quella descritta precedentemente. L'utilizzo della parola chiave checked prima di una definizione di array, genera nel TAC dei controlli a *run-time* sull'ammissibilità degli indici (cioè viene controllato se il valore di un indice è compreso nei limiti previsti dalla dimensione dell'array).

```
{
checked var x:int[] := [1,2,3]
}
```

Listing 11: Dichiarazione di un array checked

Quando un array (checked o non checked) viene passato come parametro ad una funzione, all'interno del *body* di questa l'array viene gestito come checked.
Se si desidera migliorare le *performance*, a discapito della sicurezza, è possibile comunque assegnare l'array passato come parametro ad una nuova variabile locale alla funzione, e successivamente utlizzare tale variabile.

Scoping

Lo scoping delle variabili e delle costanti funziona nel seguente modo:

- Le variabili/costanti sono visibili nel blocco di dichiarazione, dalla riga della dichiarazione a seguire
- Le variabili/costanti sono visibili nei blocchi interni successivi alla riga di dichiarazione della variabile stessa

Lo scoping per le funzioni e per le procedure funziona nel seguente modo:

- È presente l'hoisting per le funzioni/procedure, ovvero queste risultano visibili in tutto il blocco nel quale vengono dichiarate
- Le funzioni/procedure sono inoltre visibili in tutti i blocchi interni al blocco di dichiarazione

```
{
def x:int := foo(1) /* x vale 42 */
def foo(x:int):int { return 42 * x }
def z:boolean := k /* errore, k non visibile */
{
  def y:int := foo(1) + x /* y vale 84 */
}
def k:boolean := true
}
```

Listing 12: Scoping nel linguaggio

Il linguaggio possiede scoping statico, si prenda come esempio il seguente frammento di codice che non risulta avere errori di *type checking*:

```
{
def x:int := 10
def foo():int { return x }
{
  def x:char := 'a'
  def y:int := foo()
}
}
```

Listing 13: Scoping statico del linguaggio

Considerato che il tipo char non è compatibile con il tipo int (come specificato a 3.5), in un linguaggio con scoping dinamico il codice 13 non sarebbe corretto.

Overloading

Per quanto riguarda l'overloading delle funzioni/procedure, è possibile definire una funzione/procedura omonima ad una già esistente. Affinchè questo sia possibile la nuova funzione deve o richiedere un numero diverso di parametri in input, oppure richiedere che almeno un parametro sia di un tipo non compatibile con il tipo del medesimo parametro della funzione/procedura già esistente.
L'overloading delle variabili/costanti non è consentito.

```
{
var x:int := 10
def foo(x:float64):int { return 42 }
def foo(y : boolean):int {
  var result: int := 30
  if (y) {
    result := -30
  }
  return result
}
var x:boolean := true /* errore, x definita prima */
def foo(x:int):boolean { return x > 10 } /* errore, foo definita prima (il tipo int e' compatibile
    col tipo float64) */
}
```

Listing 14: Overloading nel linguaggio

Overriding

Per quanto riguarda l'overriding delle variabili/costanti è possibile ridefinire una data variabile/costante soltanto in un blocco interno rispetto a quello di dichiarazione di essa.
L'overriding di funzioni/procedure non è consentito.

```
{
var x:int := 10
def foo(x:int):int { return 42 * x }
{
  var x:int := 24
  def foo(x:int):int { return x%2 } /* errore, foo definita prima */
}
}
```

Listing 15: Overriding nel linguaggio

Limitazioni nelle dichiarazioni

Il linguaggio MINE presenta le seguenti limitazioni per quanto riguarda le dichiarazioni:

- Le funzioni/procedure possono prendere in input qualsiasi tipo di dato ad eccezione di array di dimensione non definita
- Le funzioni/procedure possono restituire qualsiasi tipo di dato ad eccezione di array (di dimensione definita o indefinita)

- Se una variabile di tipo array è definita con def, allora non è possibile modificarne il contenuto, nè riassegnando un nuovo valore, nè modificando l'array al suo interno

```
{
def x:int[] := [1,2,3]
x[0] := 4 /* non corretto */
x[0]++ /* non corretto */
x := [5,6,7] /* non corretto */
}
```

Listing 16: Esempio dichiarazioni array 2

- Se un puntatore viene definito come def, non è possibile modificarne il contenuto direttamente utilizzando il puntatore stesso

```
{
var x:int := 10
def y:int~ := &x
x++ /* corretto */
(~y)++ /* non corretto */
}
```

Listing 17: Esempio puntatori 2

3.10 Funzioni predefinite

Nel linguaggio MINE sono presenti le seguenti funzioni predefinite, esse sono utilizzabili senza necessità di dichiarazione:

- writeInt – preso in input un intero n ed una stringa S scrive n nel file S
- writeChar – preso in input un carattere c ed una stringa S scrive c nel file S
- writeFloat – preso in input un numero m ed una stringa S scrive m nel file S
- writeString – prese in input due stringhe S, S' scrive S nel file S'
- readInt – presa in input una stinga S, legge un numero intero dal file S e lo restituisce
- readChar – presa in input una stinga S, legge un carattere dal file S e lo restituisce
- readFloat – presa in input una stinga S, legge un numero dal file S e lo restituisce
- readString – presa in input una stinga S, legge una stringa S' dal file S e la restituisce

4 Implementazione

Per velocizzare e facilitare l'implementazione di MINE si è inizialmente utilizzato il tool BNFC (il cui file sorgente è stato nominato grammar.cf), e successivamente si sono modificati/raffinati i file da esso prodotti.

4.1 Makefile

È stato creato come da richiesta il file Makefile. Questo dispone dei seguenti comandi:

- make − compila tutti i file necessari al funzioamento del progetto
- make demo − esegue il programma utilizzando il file demo (presente nella cartella testcases) come codice sorgente
- make do_tests − esegue una batteria di test e controlla se questi sono corretti (per una descrizione più dettagliata dei test si rimanda a 6)
- make testout − esegue il programma su un file di test e stampa il risultato in formato String di Haskell (questo comando viene utilizzato durante la creazione dei test)
- make clean − clean dei file creati

È possibile passare il parametro TEST=n al comando make testout per eseguire la computazione sul test numero n. Tutti i test sono salvati nella cartella testcases.

4.2 Lexing

Il lexer generato da BNFC è stato modificato al fine di non considerare i caratteri di tabulazione e di spaziatura, e di considerare più caratteri $\backslash n$ consecutivi come un carattere unico. Inoltre i caratteri $\backslash r$ sono trasformati in $\backslash n$, ciò viene fatto per compatibilità con Windows e OS Classic.

Un'ulteriore importante modifica riguarda i caratteri di *newline*. Il linguaggio **E**, infatti, utilizza come separatore per i diversi comandi tale carattere, ma è possibile inserire $\backslash n$ anche all'interno dei singoli comandi. Ciò crea il problema di dover discriminare i caratteri $\backslash n$ che indicano la fine di un comando, dai caratteri $\backslash n$ che fanno parte del comando stesso. Come esempio di ciò si consideri il seguente frammento di codice che risulta essere legale nel linguaggio **E**:

```
1  var x :=
2
3
4      10
5
6  def y := [1,
7        2,
8
9        3]
10
11 def z := 10 +
12           20
```

Listing 18: Frammento di codice scritto in E

Per ovviare a questo problema si è deciso di utilizzare come separatore dei comandi il carattere ; (non utilizzato altrove nel linguaggio), e di inserire tale carattere automaticamente al posto di *newline* nelle posizioni in cui questo è stato inserito per indicare la fine di un comando.
L'inserimento automatico di tale carattere viene effettuato nel file LexGrammar.x dalla funzione preProcessing, che viene chiamata sulla lista dei token creata dal lexer. Questa funzione inserisce il nuovo carattere di terminazione dei comandi:

- dopo ogni parentesi graffa chiusa
- dopo ogni parentesi tonda chiusa se questa è seguita da un invio e se le parentesi tonde e quadre sono bilanciate
- dopo ogni parentesi quadra chiusa se questa è seguita da un invio e se le parentesi quadre sono bilanciate
- all'interno di una RExp, se le parentesi sono bilanciate ed è presente un $\backslash n$ non preceduto da un'operazione
- dopo break, continue, return se seguiti da $\backslash n$

Come esempio di ciò si consideri il seguente programma scritto in MINE

```
{
def x:int := 1 +
2


def y:int[] := [1,(2+
4)+
1,


5]

}
```

Listing 19: Frammento di codice in MINE prima della funzione preProcessing

Questo viene tradotto nel seguente frammento di codice:

```
{
def x:int := 1 + 2;

def y:int[] := [1,(2+4)+1,5];

};
```

Listing 20: Funzione preProcessing applicata al frammento di codice 19

Attuando questa strategia di sostituzione dei caratteri, però, è possibile inserire il ; nel codice sorgente tra un comando e l'altro. Questo non è consentito dalla sintassi di E, e quindi per evitare che ciò sia possibile è stato modificato il lexer in modo tale da trasformare i ; in caratteri di spazio. Siccome i caratteri di spazio originali (ovvero presenti nel codice sorgente) vengono eliminati e la grammatica non li consente, il lexer rileverà un errore quando legge tale carattere.

4.3 Albero di sintassi astratta

L'albero di sintassi astratta generato da BNFC (presente nel file AbsGrammar.hs) è stato modificato al fine di renderlo più compatto, con la conseguenza di facilitarne la creazione e la visita.
È stato inoltre introdotto un parametro polimorfo all'interno dei nodi dell'albero, questo viene usato inizialmente per salvare la posizione dei token, successivamente per salvare il tipo dei vari nodi, infine per salvare il codice TAC relativo ai nodi.
Infatti il parser crea un albero di tipo Program Posn, questo viene passato come input al typechecking che lo trasforma in un albero di tipo Program TypeCheckRes, a sua volta questo viene passato al generatore di TAC che restituisce un albero di tipo Program TAC.
L'introduzione della *record syntax* nel codice prodotto da BNFC ha permesso di facilitare la creazione e l'interrogazione dei vari tipi di dato utilizzati.

4.4 Parsing

Il sorgente del parser prodotto da BNFC (presente nel file ParGrammar.y) è stato modificato affinche venga restituito l'albero di sintassi astratta descritto precedentemente. Per costruire l'albero il parser inserisce nei singoli nodi le posizioni relative rilevate dal lexer.

Controllo conflitti

Sono presenti 2 conflitti shift/reduce e 1 conflitto reduce/reduce. Di seguito viene riportata l'analisi effettuata su di questi:

- Conflitto stato 67 – In questo stato il Parser esegue sempre lo *shift* se legge la parentesi tonda chiusa, quindi ogni coppia di parentesi che racchiude una LExpr viene riconosciuta come parte della LExpr stessa.
  Questo non è un problema perchè uno Stmt non necessita di essere disambiguato mediante l'utilizzo di una coppia di parentesi.

- Conflitto stato 69 – In questo stato se al parser arriva il token contenente la parentesi graffa chiusa, questo non è in grado di distinguere se deve riconoscere una lista vuota di Stmt oppure no. Entrambe le riduzioni vanno bene, in quanto la scelta di una di esse rispetto all'altra non influenza il risultato dell'esecuzione del codice haskell generato, questo è dovuto al fatto che vengono comunque effettuate delle concatenazioni.
- Conflitto stato 119 – Non sono presenti problemi di ambiguità nel caso di una singola LExpr all'interno di una RExpr, di conseguenza anche nel caso in cui le parentesi tonde fossero state inserite a livello della RExpr queste risulterebbero superflue. L'esecuzione dello *shift* seguita dalla riduzione da LExpr a BLexpr e successivamente da BLexpr a LExpr non causa problemi di ambiguità.

4.5 Type checking

Per effettuare il type checking del codice è stato creato il modulo TypeChecking.hs. Questo importa il modulo Types.hs, la cui creazione è stata necassaria per evitare cicli di importazione, infatti i tipi di dato presenti in esso servono sia in TypeChecking.hs, sia in AbsGrammar.hs, e il primo dei due importa il secondo. All'interno di TypeChecking.hs vengono definiti i seguenti tipi di dato:

```
type Env = Map String [EnvEntry]

data EnvEntry
    = Var {varPos:: LexGrammar.Posn, varType:: Type, varEditable:: Bool, varSize:: Integer, varMod
    :: AbsGrammar.Modality, varLineMod:: Int, varCheck:: Bool}
    | Fun {funPos:: LexGrammar.Posn, funParams:: [Param], funType:: Type, funSize:: Integer}
    deriving (Show)

data TypeCheckRes
    = TypeInfo {getEnv:: Env, getType:: Type, getPos:: LexGrammar.Posn, getSize:: Integer, getId::
    String, getLocal:: Int}
    | Errs {getErrs:: [String]}
    deriving (Show)

data Param = Param {getParamType:: Type, getParamPos:: LexGrammar.Posn, getParamModality::
    AbsGrammar.Modality, getParamString:: String, getParamSize:: Integer}
  deriving (Eq, Ord, Show, Read)
```

Listing 21: Frammento di codice del file TypeChecking.hs

Il tipo di dato Env rappresenta l'environment, questo risulta essere una mappa chiave-valore, la cui chiave è una stringa e il cui valore è una lista di entry. La chiave rappresenta o un ID di una funzione/procedura o un ID di una variabile/costante oppure un token speciale (per esempio return per indicare che ci troviamo all'interno del *body* di una funzione, oppure while per indicare che ci troviamo all'interno del *body* di un comando di ciclo). Il valore dell'environment rappresenta invece la lista delle dichiarazioni di funzioni/procedure e variabili/costanti. È stata utilizzata una lista in quanto, come specificato in 3.9, è presente l'overloading di funzioni.
Il tipo di dato TypeCheckRes rappresenta il risultato del typechecking su un dato nodo, questo può essere un'insieme di informazioni oppure una lista di stringhe di errore.

All'interno di questo modulo sono inoltre presenti le diverse funzioni che ne permettono il funzionamento, queste possono essere divise in due categorie: getTypeFrom e compute.
Nella prima categoria sono presenti le funzioni che, preso un nodo dell'albero di tipo Program Posn in input assieme ad eventuali altri parametri, creano il tipo di dato TypeCheckRes relativo a quel nodo.
Nella seconda categoria invece sono presenti le funzioni che, preso un nodo dell'albero di tipo Program Posn in input assieme ad un environment, utilizzano l'opportuna funzione getTypeFrom per costruire il relativo nodo dell'albero di tipo Program TypeCheckRes.

```
getTypeFromJumpStmt :: JumpStmt LexGrammar.Posn -> Env -> TypeCheckRes
getTypeFromJumpStmt (Break pos) env = case Data.Map.lookup "while" env of
  Just entry -> TypeInfo env (SimpTyp T_Void) pos dimVoid "" 0
  Nothing -> Errs ["unexpected break at " ++ (show pos)]

getTypeFromJumpStmt (Continue pos) env = case Data.Map.lookup "while" env of
  Just entry -> TypeInfo env (SimpTyp T_Void) pos dimVoid "" 0
  Nothing -> Errs ["unexpected continue at " ++ (show pos)]

getTypeFromJumpStmt (RetExpVoid pos) env = case Data.Map.lookup "return" env of
  Just ((Var rPos rTy const dim _ _ _):xs) -> if (subType rTy (SimpTyp T_Void)) then (TypeInfo env
    (SimpTyp T_Void) pos dimVoid "" 0) else (Errs ["type " ++ (show rTy) ++ " at " ++ (show pos) ++
     " not compatible with type void declared for function"])
```

```
  Nothing -> Errs ["unexpected return at " ++ (show pos)]

getTypeFromJumpStmt (RetExp pos rExpr) env = case (Data.Map.lookup "return" env, getTypeFromRExpr
    rExpr env) of
  (Just ((Var rPos rTy const dim _ _ _):xs), Errs strs) -> Errs strs
  (Just ((Var rPos rTy const dim _ funRow _):xs), TypeInfo _ tr _ _ _ modr) -> case tr of
    Point _ _ -> if modr > funRow
                 then
                   Errs ["not sure if returned pointer points to a global variable at " ++ (show
    pos)]
                 else
                   if (subType tr rTy) then (TypeInfo env (SimpTyp T_Void) pos dimVoid "" 0) else
    (Errs ["couldn't match expected type " ++ (show rTy) ++ " with actual type " ++ (show tr) ++ "
    at " ++ (show pos)])
    _ -> if (subType tr rTy) then (TypeInfo env (SimpTyp T_Void) pos dimVoid "" 0) else (Errs ["
    couldn't match expected type " ++ (show rTy) ++ " with actual type " ++ (show tr) ++ " at " ++
    (show pos)])
  (Nothing, _) -> Errs ["unexpected return at " ++ (show pos)]

-                    ---------------------------------------------------

computeJumpStmt:: JumpStmt Posn  -> Env -> JumpStmt TypeCheckRes
computeJumpStmt input@(Break _) env = Break (getTypeFromJumpStmt input env)
computeJumpStmt input@(Continue _) env = Continue (getTypeFromJumpStmt input env)
computeJumpStmt input@(RetExpVoid _) env = RetExpVoid (getTypeFromJumpStmt input env)
computeJumpStmt input@(RetExp _ re) env = RetExp (getTypeFromJumpStmt input env) (computeRExpr re
    env)
```

Listing 22: Coppia di funzioni getTypeFromJumpStmt e computeJumpStmt

L'albero risultante dopo la procedura di typechecking ha tipo Program TypeCheckRes.

Dichiarazioni di funzioni

Le dichiarazioni di funzioni/procedure vengono gestite nel seguente modo.

Quando viene effettuato il typechecking di un blocco B viene chiamata la funzione makeFunctionEnv, questa funzione controlla se le definizioni di funzioni/procedure presenti all'interno di B sono corrette (per esempio vengono effettuati i controlli descritti nella sezione successiva), in caso di verifica positiva le funzioni vengono aggiunte all'*environment*.

Successivamente viene chiamata la funzione checkStmt che, attraverso la funzione getTypeFromStmt, effettua il typechecking di tutti i comandi presenti in B.
Durante l'esecuzione di getTypeFromStmt le funzioni, ovviamente, non vengono aggiunte nuovamente all'*environment*, ma viene effettuato il typechecking del loro *body*.

Controlli effettuati

Oltre ai controlli canonici effettuati nella fase di typechecking, sono presenti i seguenti controlli:

- Controllo riguardo puntatori restituiti dalle funzioni – nelle funzioni che restituiscono puntatori, viene effettuato un controllo per verificare che il puntatore restituito non faccia riferimento ad una variabile interna alla funzione. Per esempio il seguente frammento di codice genera un errore:

```
{
def x:int := 10
def foo():int~ {
  def y:int := 11
  return &y
}
}
```

Listing 23: Puntatore ad una variabile locale restituito

  Mentre il seguente codice risuta essere legale:

```
{
def x:int := 10
def foo():int~ {
  return &x
}
}
```

Listing 24: Puntatore ad una variabile non locale restituito

Esistono delle situazioni nelle quali, nonstante venga restituito un puntatore che fa riferimento ad una variabile non locale, viene segnalato un errore. È possibile infatti effettuare questo controllo in modo più lasco, ovvero non segnalando situazioni effettivamente problematiche, oppure in modo più restrittivo, segnalando situazioni che di per sè non sarebbero problematiche. Durante lo sviluppo di MINE si è deciso di considerare la seconda opzione.

- Controllo riguardo parametri formali duplicati – nelle dichiarazioni di funzioni/procedure viene controllato se sono presenti degli identificativi duplicati nei parametri formali. Per esempio il seguente codice genera un errore:

```
{
def foo(x:int, y:boolean, x:int):int {
  return 42
}
}
```

Listing 25: Controllo duplicazione parametri formali

- Controllo di ridefinizione dei parametri formali – nelle dichiarazioni di funzioni/procedure viene controllato se i parametri formali vengono ridefiniti all'interno della funzione/procedura, in caso affermativo viene generato un errore. Il seguente codice per esempio non risulta essere legale:

```
{
def foo(x:int):int {
  var x:int := 10
  return x
}
}
```

Listing 26: Controllo ridefinizione parametri formali

4.6 Generazione codice intermedio

Per la generazione del codice intermedio è stato creato il modulo TACGen.hs. All'interno di esso vengono definiti i seguenti tipi di dato:

```
data TAC
  = TacAssignBinOp {getAddrResult:: Addr, getBinOp:: TBinOp, getFstAddr:: Addr, getSndAddr:: Addr,
    getTypeAssign:: String}
  | TacAssignRelOp {getAddrResult:: Addr, getBoolBinOp:: TRelOp, getFstAddr:: Addr, getSndAddr::
    Addr, getTypeAssign:: String}
  | TacAssignUnOp {getAddrResult:: Addr, getUnOp:: TUnOp, getFstAddr:: Addr, getTypeAssign:: String
    }
  | TacCopy Addr Addr String
  | TacJump Lab
  | TacConJump Lab Bool Addr
  | TacRelConJump Lab Bool TRelOp Addr Addr
  | TacParam Addr String
  | TacProcCall Addr Int
  | TacFunCall Addr Int Addr String
  | TacArrayRead Addr Addr Addr String
  | TacArrayWrite Addr Addr Addr String
  | TacAssignRef Addr Addr
  | TacAssignDeref Addr Addr String
  | TacWritePointee Addr Addr String
  | TacLabel Lab
  | TacRetVoid
  | TacRet Addr String
  | TacString Addr [Addr]
  | TacComment String
  | TacError String
```

```
  | TacExit

data TBinOp = IntAdd | FloatAdd | LongAdd | IntSub | FloatSub | LongSub | IntMul | FloatMul |
    IntDiv | FloatDiv | IntPow | FloatPow | Mod

data TRelOp = Or | And | EqInt | EqLong | EqChar | EqFloat | EqBoolean | NeqInt | NeqLong | NeqChar
     | NeqFloat | NeqBoolean | LtInt | LtLong | LtChar | LtFloat | LtBoolean | LtEInt | LtELong |
    LtEChar | LtEFloat | LtEBoolean | GtInt | GtLong | GtChar | GtFloat | GtBoolean | GtEInt |
    GtELong | GtEChar | GtEFloat | GtEBoolean

data TUnOp = Neg | IntToFloat | PreInc | PostInc | PreDecr | PostDecr
  deriving (Show)

data TACS = TACS {text :: [TAC], functions :: [TAC], dataMem :: [TAC]}
```

Listing 27: Frammento di codice del file TACGen.hs

Il tipo di dato TAC è la rappresentazione del three-address code, i costruttori e gli argomenti associati ad esso rappresentano le diverse operazioni possibili.
Il tipo di dato TACS rappresenta invece una tripla di liste di TAC nella quale il primo argomento è l'insieme dei comandi, il secondo argomento è l'insieme relativo alle dichiarazioni di funzioni, e infine il terzo argomento è l'insieme relativo alle dichiarazioni di stringhe. La creazione di questo tipo di dato è stata fatta con lo scopo di stampare prima tutte stringhe utilizzate nel programma, poi tutte le dichiarazioni di funzioni, e infine il resto del codice.

All'interno di questo modulo sono inoltre presenti le diverse funzioni che ne permettono il funzionamento. Queste preso un nodo dell'albero di tipo Program TypeCheckRes in input, assieme ad altri eventuali parametri, costruiscono il relativo nodo dell'albero di tipo Program TACS.
Per questioni di efficienza, i diversi TAC presenti nei nodi sono inseriti al contrario e su di essi viene fatto il *reverse* solo prima di essere stampati.

```
getTACFromJumpStmt:: JumpStmt TypeCheckRes -> Integer -> Lab -> Lab -> (JumpStmt TACS, Integer)
getTACFromJumpStmt jump c beg end = case jump of

  AbsGrammar.Break _ -> (AbsGrammar.Break (TACS [TacJump end] [] []), c)
  AbsGrammar.Continue _ -> (AbsGrammar.Continue (TACS [TacJump beg] [] []), c)
  AbsGrammar.RetExpVoid _ -> (AbsGrammar.RetExpVoid (TACS [TacRetVoid] [] []), c)

  AbsGrammar.RetExp tres rexpr ->
    let tacr = fst $ getValFromRExpr rexpr c in
    let d = snd $ getValFromRExpr rexpr c in
    let rtacs = (rexprContent tacr) in
    let env = getEnv tres in
    let tre = varType . head $ findWithDefault [] "return" env in
    let texp = getType $ rexprContent rexpr in

      if tre == texp
        then (AbsGrammar.RetExp (rtacs {text = ((TacRet (rexprAddr tacr) (getTACType tre)):(text
    rtacs))}) tacr, d)
        else let newT = newTemp d in

            (AbsGrammar.RetExp (rtacs {text = ((TacRet newT (getTACType tre)):(cast newT (rexprAddr
     tacr) tre texp):(text rtacs))}) tacr, d + 1)
```

Listing 28: Esempio di funzione getTac presente nel modulo TACGen.hs

Preprocessing

È stata implementata la seguente attività di *preprocessing* durante la generazione del three-address code: se viene effettuata un'operazione tra numeri interi, allora nel TAC relativo a tale operazione comparirà il risultato finale e non i passaggi intermedi di essa. Come esempio di ciò si consideri il seguente frammento di codice in MINE:

```
{
var x:int := (10 + 2**3) / (15 + 1 - 10)
x := (10 * 10) % (12 - 2)

def y:int[] := [1 + 41, 50 % 2, 120]
x := y[11-10]
}
```

Listing 29: Frammento di codice in MINE

Il codice TAC relativo è il seguente:

```
# static data
# text
  x@2,5 =int 3
L3:
  x@2,5 =int 0
L2:
  y@5,5[0] =int 42
  y@5,5[4] =int 0
  y@5,5[8] =int 120
L1:
  t9 =int 1 mul_int 4
  t10 =int y@5,5[t9]
  x@2,5 =int t10
L0:
# functions
```

Listing 30: Codice TAC relativo al programma 29

Si può notare come i valori asegnati alle variabili x e y, siano il risultato delle operazioni relative presenti nel codice 29.

Fall-through

È stata implementata la tecnica del *fall-through*, come esempio si prenda il seguente codice:

```
{
var x:boolean := false
if (x) {
  def y:int := 0
} else {
  def y:int := 1
}
}
```

Listing 31: Tecnica del fall-through

Questo viene tradotto nel seguente modo:

```
# static data
# text
  x@2,5 =boolean false
L1:
  ifFalse x@2,5 goto L3
  y@4,13 =int 0
  goto L0
L3:
  y@6,13 =int 1
L0:
# functions
```

Listing 32: TAC relativo al programma 31

5 Esecuzione del programma

I passi necessari per visualizzare il codice TAC (o eventuali errori) relativo ad un determinato sorgente di MINE presente in un file locale, sono i seguenti:

1. collocarsi con il comando cd nella cartella esercizio
2. eseguire il comando make
3. eseguire il comando ./Mine path dove *path* è il percorso al file sorgente

Se si vuole immettere direttamente codice MINE dallo standard input i passi sono i medesimi, ma al comando presente nel punto 3 non bisogna passare nessun parametro. Quando si ha terminato di scrivere il programma bisogna immettere il carattere di terminazione file (ctrl+d sui sistemi unix).

Se si desidera invece visualizzare il *pretty printer* del codice sorgente, allora i passi sono i seguenti:

1. collocarsi con il comando cd nella cartella esercizio
2. eseguire il comando ghc Printer.hs
3. eseguire il comando ./Printer path dove *path* è il percorso al file sorgente

Se si vuole immettere direttamente codice MINE dallo standard input i passi sono i medesimi, ma al comando presente nel punto 3 non bisogna passare nessun parametro. Quando si ha terminato di scrivere il programma bisogna immettere il carattere di terminazione file (ctrl+d sui sistemi unix).

6 Test

Per controllare che le varie versioni del codice non introducano nuovi errori rispetto alle versioni già testate, è stato sviluppato un programma haskel, Test.hs.
Questo programma controlla se l'output del file Mine corrisponde all'output che il medesimo file produceva nelle versione precedenti.
La metodologia con la quale si è effettuato il testing è riportata di seguito.

6.1 Creazione di nuovi testcase

Di seguito è riportata la procedura di creazione di nuovi *testcase*:

1. creazione di un file, nominato n, all'interno della cartella testcases contenente codice sorgente di MINE
2. esecuzione del comando Mine testcases/n
3. controllo dell'effettiva correttezza dell'output ricevuto
4. in caso il controllo abbia esito positivo, esecuzione del comando make testout TEST=n
5. copia dell'output del comando presente al punto precedente nell'opportuno file Test.hs

6.2 Verifica dei testcase

Grazie alla creazione del file Test.hs, ed alla sua integrazione all'interno del Makefile, la procedura di verifica dei *testcase* risulta molto semplice. Questa consiste nell'esecuzione del comando make do_tests. Verranno stampati a schermo dei feedback positivi in caso di correttezza dei test, ed un messaggio di errore (riportante il *testcase* fallito) nel caso di non correttezza.

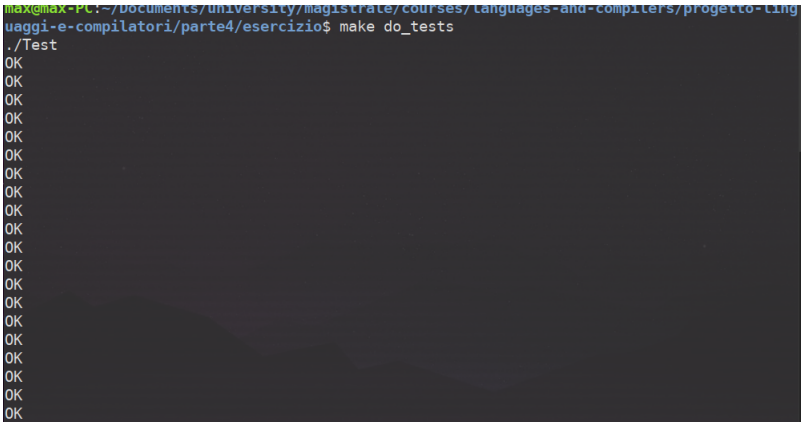

Figura 2: Esecuzione del comando make do_tests

6.3 Test significativi

Di seguito vengono riportati alcuni dei test effettuati, presenti nella cartella testcases, e il relativo output del programma.

Test 2

```
{
def guard:boolean := true
var x:int := 5
if(guard) {
  x += 4
}
}
```

Listing 33: Testcase numero 2

```
# static data
# text
  guard@2,5 =boolean true
L2:
  x@3,5 =int 5
L1:
  ifFalse guard@2,5 goto L0
  t5 =int x@3,5 add_int 4
  x@3,5 =int t5
L0:
# functions
```

Listing 34: Risultato del testcase numero 2

Test 10

```
{
var x:String := "qwerty"
var f:int :=2
if (x == "qwe") {
  while(true) {
    if (f < 200) {
     f **= 2
    } else {
     break
    }
  }
}
}
```

Listing 35: Testcase numero 10

```
# static data
str6:
  "qwe"
str4:
  "qwerty"
# text
  x@2,5 =addr str4
L2:
  f@3,5 =int 2
L1:
  param x@2,5_addr
  param str6_addr
  t9 =int call mine$strcmp@1,1:2
  ifFalse t9 eq_int 0 goto L0
  goto L10
L11:
  ifFalse f@3,5 lt_int 200 goto L12
  t16 =int f@3,5 pow_int 2
  f@3,5 =int t16
  goto L10
L12:
  goto L0
L10:
  goto L11
L0:
# functions
```

Listing 36: Risultato del testcase numero 10

Test 5

```
{
var x:int := 5
  if (x == 5) {
    x := false
  }
}
```

Listing 37: Testcase numero 5

```
type boolean in (4,7) not compatible with type int in (4,4)
```

Listing 38: Risultato del testcase numero 5

Test 16

```
{
def x (valres y: int, ref z : int, w : int): int {
  return y * z * w
}
var a: int := 1
var b: int := 2
var c: int := 3
def res: int := x(a, b, c)
}
```

Listing 39: Testcase numero 16

```
# static data
# text
L4:
  a@5,5 =int 1
L3:
  b@6,5 =int 2
L2:
  c@7,5 =int 3
L1:
  t9 =addr &a@5,5
  param t9_addr
  t10 =addr &b@6,5
  param t10_addr
  param c@7,5_int
  t14 =int call x@2,13:3
  res@8,5 =int t14
L0:
# functions
x@2,13:
# init
  y@2,22$valres =int *y@2,22
# code of x@2,13
  t6 =int *z@2,34
  t7 =int y@2,22$valres mul_int t6
  t8 =int t7 mul_int w@2,43
# postamble
  *y@2,22 =int y@2,22$valres
  return_int t8
# postamble
  *y@2,22 =int y@2,22$valres
error "control of function x@2,13 should not reach this point."
```

Listing 40: Risultato del testcase numero 16

Test 28

```
{
var x:int := 20
def y:int[] := [x]
for i in y[0]+y[0]**2 .. x+y[0] {
    x := i
}
}
```

Listing 41: Testcase numero 28

```
# static data
# text
  x@2,5 =int 20
L2:
  y@3,5[0] =int x@2,5
L1:
  t5 =int 0 mul_int 4
  t6 =int y@3,5[t5]
  t9 =int 0 mul_int 4
```

```
  t10 =int y@3,5[t9]
  t13 =int t10 pow_int 2
  t14 =int t6 add_int t13
  t15 =int 0 mul_int 4
  t16 =int y@3,5[t15]
  t19 =int x@2,5 add_int t16
  t20 =int t19
  i@4,5 =int t14
L4:
  if i@4,5 gt_int t20 goto L23
  x@2,5 =int i@1,1
  i@4,5 =int i@4,5 add_int 1
  goto L4
L23:
L0:
# functions
```

Listing 42: Risultato del testcase numero 28

Test 33

```
{
var x:int :=5
def foo( b:int) : int {
  return 5
}
 do {
  var t:int~ := &x
  var m:int := foo(~t)
  } while (x != 6)

}
```

Listing 43: Testcase numero 33

```
# static data
# text
  x@2,5 =int 5
L2:
L1:
L5:
  t10 =addr &x@2,5
  t@7,7 =addr t10
L8:
  t11 =int *t@7,7
  param t11_int
  t13 =int call foo@3,5:1
  m@8,7 =int t13
L4:
  if x@2,5 neq_int 6 goto L5
L0:
# functions
foo@3,5:
# code of foo@3,5
  return_int 5
error "control of function foo@3,5 should not reach this point."
```

Listing 44: Risultato del testcase numero 33

Test 34

```
# this is a in-line comment
{
def x:int := readInt("stdin")
var res:float64 := 0
switch(x%5) {
    match 1 { res := 100 }
    match 2 { res := 120 }
    match 3 { writeString("stdout", "No sense")
              res := 90
            }
    match _ {}
}
```

```
13
14 def ok: boolean := `res <= 100` ? `true` ^ `false`
15 }
```

Listing 45: Testcase numero 34

```
1  # static data
2  str32:
3    "No sense"
4  str5:
5    "stdin"
6  str31:
7    "stdout"
8  # text
9    param str5_addr
10   t7 =int call readInt@1,1:1
11   x@3,5 =int t7
12 L3:
13   res@4,5 =float64 IntToFloat 0
14 L2:
15   t9 =int x@3,5 mod_int 5
16   t8 =int t9
17   ifFalse t8 eq_int 1 goto L12
18   t18 =float64 IntToFloat 100
19   res@4,5 =float64 t18
20 L16:
21   goto L1
22 L12:
23   ifFalse t8 eq_int 2 goto L11
24   t24 =float64 IntToFloat 120
25   res@4,5 =float64 t24
26 L22:
27   goto L1
28 L11:
29   ifFalse t8 eq_int 3 goto L10
30   param str31_addr
31   param str32_addr
32   call writeString@1,1:2
33 L29:
34   t35 =float64 IntToFloat 90
35   res@4,5 =float64 t35
36 L28:
37   goto L1
38 L10:
39 L1:
40   t41 =float64 IntToFloat 100
41   ifFalse res@4,5 lte_float t41 goto L37
42   t39 =boolean true
43   goto L38
44 L37:
45   t39 =boolean false
46 L38:
47   ok@14,5 =boolean t39
48 L0:
49 # functions
```

Listing 46: Risultato del testcase numero 34

A Appendice

La seguente appendice contiene la descrizione della grammatica del linguaggio generata da BNFC.

Parole e simboli riservati

Le parole riservate, usate nella grammatica, sono le seguenti:

`String`	`boolean`	`break`
`char`	`checked`	`const`
`continue`	`def`	`do`
`else`	`false`	`float64`
`for`	`if`	`in`
`int`	`match`	`ref`
`return`	`switch`	`true`
`val`	`valres`	`var`
`void`	`while`	

I simboli usati nella grammatica sono i seguenti:

	`;`	`,`
`(`	`)`	`||`
`&&`	`!`	`==`
`!=`	`<`	`<=`
`>`	`>=`	`+`
`-`	`*`	`/`
`%`	`**`	`&`
`` ` ``	`?`	`^`
`~`	`++`	`--`
`[`	`]`	`:`
`:=`	`..`	`{`
`}`	`_`	`*=`
`+=`	`/=`	`-=`
`**=`	`&=`	`|=`

Commenti

I commenti mono-linea cominciano con `#`.
I commenti multi-linea sono racchiusi tra `/*` and `*/`.

La struttura sintattica della grammatica

⟨*Boolean*⟩ ::= `true`
| `false`

⟨*BasicType*⟩ ::= `boolean`
| `char`
| `float64`
| `int`
| `void`
| `String`

⟨*Modality*⟩ ::= ϵ
| `val`
| `ref`
| `valres`
| `const`

⟨*NewLine*⟩ ::=

⟨*Program*⟩ ::= ⟨*BlockDecl*⟩ `;`

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

⟨*RExpr*⟩ ::= ⟨*RExpr1*⟩
| ⟨*RExpr*⟩ || ⟨*RExpr1*⟩
| ‘ ⟨*RExpr*⟩ ‘ `?` ‘ ⟨*RExpr*⟩ ‘ `^` ‘ ⟨*RExpr*⟩ ‘

⟨*RExpr1*⟩ ::= ⟨*RExpr2*⟩
| ⟨*RExpr1*⟩ `&&` ⟨*RExpr2*⟩

⟨*RExpr2*⟩ ::= ⟨*RExpr3*⟩
| `!` ⟨*RExpr3*⟩

⟨*RExpr3*⟩ ::= ⟨*RExpr4*⟩

⟨*RExpr4*⟩ ::= ⟨*RExpr5*⟩

⟨*RExpr5*⟩ ::= ⟨*RExpr6*⟩
| ⟨*RExpr6*⟩ == ⟨*RExpr6*⟩
| ⟨*RExpr6*⟩ `!=` ⟨*RExpr6*⟩
| ⟨*RExpr6*⟩ < ⟨*RExpr6*⟩
| ⟨*RExpr6*⟩ <= ⟨*RExpr6*⟩
| ⟨*RExpr6*⟩ > ⟨*RExpr6*⟩
| ⟨*RExpr6*⟩ >= ⟨*RExpr6*⟩

⟨*RExpr7*⟩ ::= ⟨*RExpr8*⟩
| ⟨*RExpr7*⟩ + ⟨*RExpr8*⟩
| ⟨*RExpr7*⟩ − ⟨*RExpr8*⟩

⟨*RExpr8*⟩ ::= ⟨*RExpr9*⟩
| ⟨*RExpr8*⟩ `*` ⟨*RExpr9*⟩
| ⟨*RExpr8*⟩ `/` ⟨*RExpr9*⟩
| ⟨*RExpr8*⟩ `%` ⟨*RExpr9*⟩

⟨*RExpr9*⟩ ::= ⟨*RExpr10*⟩
| ⟨*RExpr10*⟩ `**` ⟨*RExpr9*⟩

⟨*RExpr10*⟩ ::= ⟨*RExpr11*⟩

⟨*RExpr11*⟩ ::= ⟨*RExpr12*⟩
| − ⟨*RExpr12*⟩
| `&` ⟨*LExpr*⟩

⟨*RExpr12*⟩ ::= ⟨*RExpr13*⟩
| ⟨*Ident*⟩ `(` ⟨*ListRExpr*⟩ `)`

⟨*RExpr13*⟩ ::= ⟨*RExpr14*⟩
| ⟨*Integer*⟩
| ⟨*Char*⟩
| ⟨*String*⟩
| ⟨*Double*⟩
| ⟨*Boolean*⟩

⟨*RExpr14*⟩ ::= `(` ⟨*RExpr*⟩ `)`
| ⟨*LExpr*⟩

⟨*LExpr*⟩ ::= ⟨*LExpr1*⟩
| `~` ⟨*RExpr*⟩
| ++ ⟨*LExpr1*⟩
| −− ⟨*LExpr1*⟩

⟨*LExpr1*⟩ ::= ⟨*LExpr2*⟩
| ⟨*LExpr2*⟩ ++
| ⟨*LExpr2*⟩ −−

⟨*LExpr2*⟩ ::= `(` ⟨*LExpr*⟩ `)`
| ⟨*BLExpr*⟩

⟨*BLExpr*⟩ ::= ⟨*BLExpr*⟩ `[` ⟨*RExpr*⟩ `]`
| ⟨*Ident*⟩

⟨*TypeSpec*⟩ ::= ⟨*BasicType*⟩
| ⟨*CompoundType*⟩

⟨*CompoundType*⟩ ::= ⟨*TypeSpec*⟩ `[` ⟨*Integer*⟩ `]`
| ⟨*TypeSpec*⟩ `[ ]`
| ⟨*TypeSpec*⟩ `~`

⟨*VarDeclInit*⟩ ::= `var` ⟨*Ident*⟩ `:` ⟨*TypeSpec*⟩ `:=` ⟨*ComplexRExpr*⟩
| `def` ⟨*Ident*⟩ `:` ⟨*TypeSpec*⟩ `:=` ⟨*ComplexRExpr*⟩

⟨*CheckedDecl*⟩ ::= `checked` ⟨*VarDeclInit*⟩
| ⟨*VarDeclInit*⟩

⟨*ComplexRExpr*⟩ ::= ⟨*RExpr*⟩
| `[` ⟨*ListComplexRExpr*⟩ `]`

⟨*ListComplexRExpr*⟩ ::= ⟨*ComplexRExpr*⟩
| ⟨*ComplexRExpr*⟩ `,` ⟨*ListComplexRExpr*⟩

⟨*ListParameter*⟩ ::= ϵ
| ⟨*Parameter*⟩
| ⟨*Parameter*⟩ `,` ⟨*ListParameter*⟩

⟨*Parameter*⟩ ::= ⟨*Modality*⟩ ⟨*Ident*⟩ `:` ⟨*TypeSpec*⟩

⟨*ListStmt*⟩ ::= ϵ
| ⟨*Stmt*⟩
| ⟨*Stmt*⟩ `;` ⟨*ListStmt*⟩

⟨*Stmt*⟩ ::= ⟨*BlockDecl*⟩
| ⟨*Ident*⟩ `(` ⟨*ListRExpr*⟩ `)`
| ⟨*JumpStmt*⟩
| `while (` ⟨*RExpr*⟩ `)` ⟨*BlockDecl*⟩
| `do` ⟨*BlockDecl*⟩ `while (` ⟨*RExpr*⟩ `)`
| `for` ⟨*Ident*⟩ `in` ⟨*RExpr*⟩ `..` ⟨*RExpr*⟩
| ⟨*SelectionStmt*⟩
| ⟨*LExpr*⟩ ⟨*Assignment-op*⟩ ⟨*RExpr*⟩
| ⟨*LExpr*⟩
| ⟨*CheckedDecl*⟩
| `def` ⟨*Ident*⟩ `(` ⟨*ListParameter*⟩ `)` `:` ⟨*TypeSpec*⟩ ⟨*BlockDecl*⟩
| `switch (` ⟨*RExpr*⟩ `)` ⟨*SwitchBlock*⟩
| `(` ⟨*Stmt*⟩ `)`
| ⟨*Stmt*⟩ `;`

⟨*SwitchBlock*⟩ ::= `{` ⟨*ListSwitchMatch*⟩ `}`

⟨*SwitchMatch*⟩ ::= `match` ⟨*RExpr*⟩ ⟨*BlockDecl*⟩
| `match _` ⟨*BlockDecl*⟩

⟨*ListSwitchMatch*⟩ ::= ϵ
| ⟨*SwitchMatch*⟩
| ⟨*SwitchMatch*⟩ `;` ⟨*ListSwitchMatch*⟩

⟨*BlockDecl*⟩ ::= `{` ⟨*ListStmt*⟩ `}`

⟨*Assignment-op*⟩ ::= `:=`
| `*=`
| `+=`
| `/=`
| `-=`
| `**=`
| `&=`
| `|=`

⟨*JumpStmt*⟩ ::= `break`
| `continue`
| `return`
| `return` ⟨*RExpr*⟩

⟨*SelectionStmt*⟩ ::= `if (` ⟨*RExpr*⟩ `)` ⟨*BlockDecl*⟩
| `if (` ⟨*RExpr*⟩ `)` ⟨*BlockDecl*⟩ `else` ⟨*BlockDecl*⟩