Descrizione del linguaggio Scala40

Sara Biavaschi (130512)
Federico Bulzoni (142242)
Simone Scaboro (143191)

La seguente relazione ha lo scopo di fornire una descrizione del linguaggio **Scala40** costruito a partire dal linguaggio **Scala**, e del front-end di compilatore che è stato sviluppato.

La prima sezione riguarda la descrizione del linguaggio. Verranno presentati la struttura lessicale, la struttura sintattica e i vincoli di semantica statica del linguaggio. Sono evidenziate solo le caratteristiche salienti, omesse le scelte standard e le richieste presenti nella consegna.

Successivamente verranno descritte le soluzioni non standard utilizzate per il compilatore, ovvero le due monadi `Writer a` e `State a`. Verranno descritti i due principali moduli che vanno a comporre il compilatore front-end del linguaggio **Scala40**, ovvero il modulo di analisi di semantica statica e il modulo di generazione del Three Address Code. In questa sezione viene data una descrizione generale dei moduli; per una descrizione più dettagliata si rimanda al codice nei rispettivi file.

Per le istruzioni di compilazione e di test si rimanda al file `README.md` nella cartella `Scala40Compiler`.

1 Struttura lessicale di Scala40

Parole riservate

Le parole riservate in **Scala40** sono le seguenti:

`if`, `else`, `do`, `while`, `def`, `return`, `var`, `Array`, `False`, `True`, `until`, `by`, `Null`, `Char`, `String`, `Int`, `Float`, `Bool`, `val`, `ref`, `res`, `valres`.

Identificatori

Un identificatore $\langle Ident \rangle$ è una lettera o il carattere '_' seguiti da una sequenza arbitraria di lettere, cifre e del carattere '_'.

Letterali

Vi sono letterali per numeri interi, numeri in virgola mobile, singoli caratteri, booleani, stringhe. Essi seguono le convenzioni della maggior parte dei linguaggi

di programmazione.

$$\begin{aligned}\langle \textit{Literal} \rangle ::= & \langle \textit{Int} \rangle \\ & \mid \langle \textit{Float} \rangle \\ & \mid \langle \textit{Char} \rangle \\ & \mid \langle \textit{Bool} \rangle \\ & \mid \langle \textit{String} \rangle \\ & \mid \texttt{Null}\end{aligned}$$

Commenti

I commenti in **Scala40** sono di due tipi:

- i commenti di una riga sono sequenze di caratteri che iniziano con `//` e finiscono al termine della riga;
- i commenti multi-riga sono sequenze di caratteri che iniziano con `/*` e terminano con `*/`. Non possono essere annidati.

Caratteri di spaziatura

I token possono essere separati dai caratteri di spaziatura standard o commenti.

2 Struttura sintattica di Scala40

- Un *programma* è una sequenza di dichiarazioni.
- Una *dichiarazione* ha una delle seguenti forme:
  - *Dichiarazione di variabili*

$$\begin{aligned}\langle \textit{Decl} \rangle ::= & \texttt{var } \langle \textit{Ident} \rangle \texttt{: } \langle \textit{TypeSpec} \rangle \texttt{ = } \langle \textit{Expr} \rangle \texttt{ ;} \\ & \mid \texttt{var } \langle \textit{Ident} \rangle \texttt{: } \langle \textit{TypeSpec} \rangle \texttt{;}\end{aligned}$$

  - *Dichiarazione di funzioni e procedure*

$$\begin{aligned}\langle \textit{Decl} \rangle ::= & \texttt{def } \langle \textit{Ident} \rangle \ \langle \textit{ParamClauses} \rangle \texttt{: } \langle \textit{TypeSpec} \rangle \texttt{ = } \langle \textit{Expr} \rangle \texttt{ ;} \\ & \texttt{def } \langle \textit{Ident} \rangle \ \langle \textit{ParamClauses} \rangle \texttt{: } \langle \textit{TypeSpec} \rangle \texttt{ = } \langle \textit{Block} \rangle \\ & \texttt{def } \langle \textit{Ident} \rangle \ \langle \textit{ParamClauses} \rangle \texttt{ = } \langle \textit{Expr} \rangle \texttt{ ;} \\ & \texttt{def } \langle \textit{Ident} \rangle \ \langle \textit{ParamClauses} \rangle \texttt{ = } \langle \textit{Block} \rangle\end{aligned}$$

  dove $\langle \textit{TypeSpec} \rangle$ è una specifica di tipo, che ha la forma

$$\begin{aligned}\langle \textit{TypeSpec} \rangle ::= & \langle \textit{SimpleType} \rangle \\ & \mid \texttt{* } \langle \textit{TypeSpec} \rangle \\ & \mid \texttt{Array [} \langle \textit{TypeSpec} \rangle \texttt{] (} \langle \textit{Int} \rangle \texttt{)} \\ \langle \textit{SimpleType} \rangle ::= & \texttt{Bool} \mid \texttt{Char} \mid \texttt{Int} \mid \texttt{Float} \mid \texttt{String}\end{aligned}$$

mentre l'elemento ⟨*ParamClauses*⟩ è una sequenza, non vuota, di ⟨*ParamClause*⟩, e ciascun ⟨*ParamClause*⟩ ha la forma

⟨*ParamClause*⟩ ::= `(` ⟨*Params*⟩ `)`

⟨*Params*⟩ è una sequenza, che può essere vuota, di elementi separati da virgola della forma

⟨*Param*⟩ ::= ⟨*Mode*⟩ ⟨*Ident*⟩ `:` ⟨*TypeSpec*⟩

dove ⟨*Mode*⟩ indica la modalità di passaggio del parametro

⟨*Mode*⟩ ::= `val` | `ref` | `res` | `valres`

Ad esempio, una specifica di tipo valida è `Array[*Int](2)`, che indica un array di puntatori ad interi di dimensione 2. Ad esempio, una definizione di funzione valida è

```
def foo(val a: Array[*Int](2), val p: *Int)(val x: Int): Int =
*a[1] + *a[2] + *p + x;
```

Essa prende come parametri un array di puntatori ad interi, un puntatore ad un intero e un intero e restituisce un intero.

- Un *blocco* è una sequenza di istruzioni racchiuse fra parentesi graffe.

  ⟨*Block*⟩ ::= `{` ⟨*StmtList*⟩ `}`

- Una *istruzione* ha la forma:

  ⟨*Stmt*⟩ ::= ⟨*Decl*⟩
  | ⟨*Block*⟩
  | ⟨*LExpr*⟩ ⟨*OpAssign*⟩ ⟨*Expr*⟩ `;`
  | `if (` ⟨*Expr*⟩ `)` ⟨*Stmt*⟩
  | `if (` ⟨*Expr*⟩ `)` ⟨*Stmt*⟩ `else` ⟨*Stmt*⟩
  | `for (` ⟨*Ident*⟩ `<-` ⟨*Expr*⟩ `until` ⟨*Expr*⟩ `by` ⟨*Expr*⟩ `)` ⟨*Stmt*⟩
  | `do` ⟨*Stmt*⟩ `while (` ⟨*Expr*⟩ `) ;`
  | `break ;`
  | `continue ;`
  | `return ;`
  | `return` ⟨*Expr*⟩ `;`
  | ⟨*Ident*⟩ ⟨*Args*⟩ `;`

  dove, nell'ultima istruzione, che corrisponde alla chiamata di procedura o funzione, ⟨*Args*⟩ è una sequenza, non vuota, di ⟨*Arg*⟩ della forma

  ⟨*Arg*⟩ ::= `(` ⟨*ExprList*⟩ `)`

e ⟨*ExprList*⟩ è una sequenza, che può essere vuota, di ⟨*Expr*⟩ separate da virgola.

Invece ⟨*OpAssign*⟩ è uno dei seguenti operatori di assegnamento:

⟨*OpAssign*⟩ ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `^=`

- Le *left expressions* del linguaggio hanno la seguente forma:

  ⟨*LExpr*⟩ ::= ⟨*Ident*⟩
  | ⟨*LExpr*⟩ `[` ⟨*Expr*⟩ `]`
  | `*` ⟨*LExpr*⟩
  | `(` ⟨*LExpr*⟩ `)`

  L'operatore accesso ad array `[]` ha la precedenza sull'operatore di dereference `*`. Quindi ad esempio `*a[1]` è sintatticamente equivalente a `*(a[1])`.

- Le *right expressions* del linguaggio hanno la seguente forma:

  ⟨*Expr*⟩ ::= ⟨*Literal*⟩
  | ⟨*LExpr*⟩
  | `&` ⟨*LExpr*⟩
  | `Array (` ⟨*ExprList*⟩ `)`
  | ⟨*Ident*⟩ ⟨*Args*⟩
  | ⟨*Expr*⟩ ⟨*BinOp*⟩ ⟨*Expr*⟩
  | ⟨*UnOp*⟩ ⟨*Expr*⟩
  | `(` ⟨*Expr*⟩ `)`| ⟨*Ident*⟩ ⟨*Args*⟩

  ⟨*BinOp*⟩ ::= `||` | `&&` | `<` | `<=` | `>` | `>=` | `==` | `!=` | `+` | `-` | `*` | `/` | `%` | `^`
  ⟨*UnOp*⟩ ::= `!` | `-`

  Gli operatori hanno precedenze e associatività standard.

3 Vincoli di semantica statica di Scala40

Scoping

Il linguaggio ha scoping statico con visibilità dal punto di dichiarazione in poi. Una variabile/funzione dichiarata all'interno di un blocco è visibile all'interno dell'intero blocco e nei sotto-blocchi in esso contenuti, nel caso in cui una stessa variabile/funzione sia stata dichiarata più volte all'interno di blocchi annidati, viene considerata la dichiarazione nel blocco più vicino al punto di utilizzo. Gli identificatori di variabili e funzioni/procedure devono essere univoci all'interno di uno stesso scope. Lo spazio dei nomi è unico, ossia non è permesso dichiarare una variabile e una funzione/procedura con lo stesso nome.

	Bool	Char	Int	Float	String
Bool	T	T	T	T	F
Char	F	T	T	T	F
Int	F	F	T	T	F
Float	F	F	F	T	F
String	F	F	F	F	T

Table 1: Tabella compatibilità tra tipi di base: una entry (t_i, t_j) ha valore T se il tipo t_i può essere convertito al tipo t_j, ha valore F altrimenti.

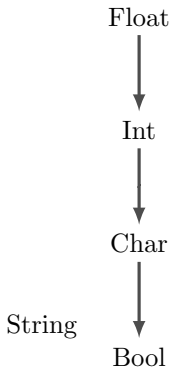

Figure 1: Rappresentazione grafica della compatibilità tra tipi base.

Vincoli riguardanti i tipi

Il linguaggio ha 5 tipi base: `Bool`, `Char`, `Int`, `Float` e `String`; le compatibilità tra i tipi di base sono riportate in tabella 1. La rappresentazione grafica delle compatibilità è in figura **??**.

Oltre ai tipi di base sono presenti due tipi composti: `*typ`, puntatore a `typ` e `Array[typ](dim)`, array di dimensione `dim` di elementi di tipo `typ`, dove `typ` è un tipo base o un tipo composto e `dim` è un intero; il tipo speciale `*Void` è utilizzato internamente per codificare il valore `Null`. È possibile assegnare il valore `Null` a qualsiasi elemento di tipo puntatore a `typ` qualunque sia il tipo `typ` puntato. I tipi composti seguono le seguenti regole di compatibilità:

- un elemento di tipo `*typ1` è compatibile unicamente con elementi di tipo `*typ2`, un elemento di tipo `*typ1` è compatibile con un elemento di tipo `*typ2` se e solo se `typ1` è compatibile con `typ2` o se `typ1` = `Void`;
- un elemento di tipo `Array[typ1](dim1)` è compatibile unicamente con elementi di tipo `Array[typ2](dim2)`, un elemento di tipo `Array[typ1](dim1)` è compatibile con un elemento di tipo `Array[typ2](dim2)` se e solo `typ1` è compatibile con `typ2` e `dim1` = `dim2`.

Le regole di compatibilità tra tipi sono codificate all'interno della funzione `compatible` nel modulo `StaticAnalysis.hs`. Sia `compatible typ1 typ2` equivalente

Operatore	Vincoli	Tipo
$!exp$	$\tau(exp) = Bool$	$Bool$
$-exp$	$\tau(exp) \leq Float$	`if` $\tau(exp) = Float$ `then` $Float$ `else` Int

Table 2: Regole di derivazione per gli operatori unari.

Classe	Membri
Numerici	`+`, `-`, `*`, `/`, `^`, `%`
Booleani	`||`, `&&`
Relazionali	`<`, `<=`, `>`, `>=`, `==`, `!=`

Table 3: Classificazione operatori binari.

a $typ_1 \leq typ_2$ nella notazione utilizzata in seguito, sia inoltre $\tau(exp)$ il tipo della espressione exp. Le regole di derivazione per gli operatori unari sono riportare in tabella 2.

Gli operatori binari sono classificati in `numerici`, `booleani`, `relazionali` come riportato in tabella 3. Le regole di derivazione per gli operatori binari sono definite in base alle possibili classi, tali regole sono riportate in tabella 5.

Le regole di inferenza per le altre espressioni sono riportate in tabella **??**.

Riguardo alle L-espressioni, ogni identificatore all'interno di un programma `Scala40` ha associato un tipo, nel caso delle variabili tale tipo è il tipo di dichiarazione della variabile, nel caso delle funzioni è il loro tipo di ritorno. Le regole di derivazione per gli operatori su L-espressioni di accesso ad un array, referenziamento e dereferenziamento sono riportate in tabella **??**.

Negli statement di assegnamento è richiesto che il tipo della R-espressione sia compatibile con il tipo della L-espressione a cui si vuole assegnare. Negli statement condizionali è richiesto che la condizione sia una espressione compatibile con il tipo $Bool$, se così non è viene lanciato un errore.

Vincoli riguardanti procedure e funzioni

Alle procedure è assegnato come tipo di ritorno il tipo interno `Void`. La signature della procedura principale di un programma `Scala40` è fissata a `main () : Void`, se si tenta di dichiarare una funzione/procedura con identificatore `main` e signature diversa da `main () : Void` viene riportato un errore. Le chiamate di procedura non possono essere utilizzate come espressioni in quanto non hanno alcun valore di ritorno.

Riguardo al passaggio dei parametri attuali nelle chiamate di funzioni/procedure

Operatore	Vincoli	Tipo
e_1 op e_2 t.c. $op \in Numerici$	$max(\tau(e_1), \tau(e_2)) \leq Float$	`if` $\tau(max(\tau(e_1), \tau(e_2))) = Float$ `then` $Float$ `else` Int
e_1 op e_2 t.c $op \in Booleani$	$\tau(e_1) \leq Bool \wedge \tau(e_2) \leq Bool$	$Bool$
e_1 op e_2 t.c. $op \in Relazionali$	$\tau(e_1) \leq \tau(e_2) \wedge \tau(e_2) \leq \tau(e_1)$	$Bool$

Table 4: Regole di derivazione per gli operatori binari.

Operatore	Vincoli	Tipo
Array($e_1, \ldots, e_n$)	$\forall_i(\tau(e_i) \leq max(\}\tau(e_1), \ldots, \tau(e_n)\})$	$Array[max(\{\tau(e_1), \ldots, \tau(e_n)\})][n]$
if e_c then e_T else e_F	$e_c \leq Bool \wedge e_T \leq max(\tau(e_T), \tau(e_F)) \wedge e_F \leq max(\tau(e_T), \tau(e_F))$	$max(\tau(e_T), \tau(e_F)$

Table 5: Regole di derivazione per l'operatore di creazione di array e l'operatore condizionale.

Operatore	Vincoli	Tipo
$*lexp$	$\tau(lexp) = *typ$ dove typ è un tipo qualsiasi	typ
$\&lexp$		$*\tau(lexp)$
$lexp[e_a]$	$\tau(e_a) \leq Bool \wedge \tau(lexp) = Array[typ](dim)$ con typ e dim qualsiasi	typ

Table 6: Regole di derivazione per gli operatori su L-espressioni.

si richiede che il parametro attuale sia una L-espressione nel caso in cui il corrispondente parametro formale sia dichiarato con modalità per `riferimento`, per `risultato` o per `valore-risultato`. Nel caso in cui il parametro formale sia dichiarato con modalità per `valore` non ci sono vincoli sul corrispondente parametro attuale. Si richiede una corrispondenza bi-univoca tra la firma di una chiamata di procedura/funzione e la corrispondente firma di procedura/funzione, ogni parametro attuale oltre a rispettare i vincoli sulla modalità deve avere tipo compatibile con il tipo del corrispondente parametro formale.

Negli statement `return exp` si richiede che il tipo della espressione `exp` sia compatibile con il tipo dello scope in cui l'istruzione è contenuta, se così non è un errore viene lanciato.

Non è permesso utilizzare statement `return exp` all'interno di procedure, sono chiaramente permessi gli statement `return` senza espressione di ritorno; vale il viceversa nel caso delle funzioni.

Altri vincoli forti

Gli statement `break` e `continue` sono utilizzabili solamente all'interno di cicli indeterminati (`while` e `do ... while`), nel caso ne si rilevi un utilizzo inappropriato viene lanciato un errore.

All'interno di un ciclo `for` la variabile di iterazione è considerata come una implicita dichiarazione locale del corpo.

Vincoli deboli

La violazione dei seguenti vincoli non impedisce il proseguimento alle successive fasi della compilazione, tuttavia viene visualizzato un messaggio di avviso in quanto si ritiene che il comportamento ottenuto possa non essere quello cercato.

- All'interno del blocco di codice di una funzione, deve essere presente almeno una istruzione `return exp`, con $\tau(exp)$ compatibile con il tipo di ritorno della funzione per ogni possibile percorso di esecuzione del blocco (per ogni valutazione delle guardie delle istruzioni `if`).

- Il programma deve contenere la dichiarazione di una procedura con firma `main ()` al top level.
- Se una funzione viene chiamata senza assegnarne il valore di ritorno ad un'espressione.

4 Soluzioni non standard

4.1 Gestione del log tramite la monade Writer

La gestione del log merita un discorso dedicato, per comprendere appieno cosa è stato svolto bisogna partire da una analisi del modulo `Errors.hs`.

Un elemento di log, che può essere un `Warning` oppure un `Error`, contiene l'eccezione e la posizione in cui si è verificata. Le eccezioni sono di tipo `TCException` e ogni eccezione ha un messaggio di notifica dedicato.

Il modulo `Environment.hs` che fornisce l'interfaccia al modulo di analisi statica per la gestione dell'environment del programma, utilizza una versione modificata della monade di default per la gestione degli errori utilizzata da BNFC. Nella versione da noi utilizzata in caso di errore viene ritornato un oggetto di tipo `TCException` invece che un oggetto `String`.

All'interno del modulo `StaticAnalysis.hs` è invece la monade `Writer` ad occuparsi di immagazzinare gli elementi di log creati durante l'analisi statica del programma. La monade `Writer` si occupa di immagazzinare gli elementi di log all'interno di una lista di `LogElement`. Un nuovo elemento di log viene aggiunto alla lista tramite la funzione `tell`.

Giusto per chiarezza sintattica, nel nostro codice è stata aggiunta una funzione `saveLog` che è poco più che un alias della funzione `tell`, ed è stato definito l'alias `Logger a` per `Writer [LogElement] a`. Ogni funzione del modulo di analisi statica che può incorrere in un'eccezione durante l'analisi dell'albero di sintassi astratta, ritorna dunque un elemento `Logger a`.

4.2 Gestione delle istruzioni TAC tramite la monade State

Per la gestione della creazione delle istruzioni TAC durante la creazione di quest'ultimo, è stata utilizzata la monade `State a`. Questa permette di manipolare l'inserimento di nuove istruzioni, la creazione di nuove etichette e di nuovi temporanei, in modo equiparabile ad un cambio di stato in una computazione. La creazione e l'inserimento di una nuova istruzione nel codice TAC corrispongono, appunto, ad una modifica dello stato corrente. La monade `State a` utilizza due funzioni per operare su quest'ultimo: `get`, per ottenere lo stato, e `put` per aggiornarlo.

La monade `State a` utilizzata durante la creazione del Three Address Code opera su stati formati da tuple di sei elementi:

- un intero che indica l'indice dell'ultimo temporaneo creato;
- un intero che indica l'ultima etichetta creata;

- la lista di istruzioni TAC che compongono il TAC in output;
- una lista di `FunState`. Un oggetto `FunState` è composto da:
  - una lista di istruzioni TAC (`funCode`): contiene le istruzioni TAC che compongono la funzione;
  - un tipo (`funType`): tipo di ritorno della funzione. Se non ci si trova in una funzione (scope globale) il valore di default è il tipo `Void`. Questo campo è utile per controllare se vi sia la necessità di eseguire l'operazione di cast sull'espressione ritornata da un eventuale return;
  - una lista di parametri (`funParams`): parametri della funzione. Questo elemento è utile per la gestione del postambolo e dei relativi assegnamenti necessari a seconda del metodo di passaggio deciso.

  Questo oggetto viene usato durante la dichiarazione di funzione.
- una coppia di etichette `(label_continue, label_break)`: dove l'etichetta `label_continue` è quella da indicare nel `Goto` in caso venga trovata un'istruzione `continue`. Indica l'etichetta del ciclo corrente nel quale l'istruzione è contenuta. Una volta usciti dal ciclo, l'etichetta viene reipostata a quella precedente, per garantire la correttezza in caso di cicli indeterminati annidati. Quando non ci si trova in un ciclo il valore impostato è quello di un'etichetta fittizia: `Label -1`. L'etichetta `label_break` è quella da indicare nel `Goto` in caso venga trovata un'istruzione `break`. Per le modalità di utilizzo sono le medesime di `label_continue`;
- una coppia di etichette `(label_arr, label_void)`: dove l'etichetta `label_arr` indica la posizione nel flusso della funzione che restituisce l'errore nel caso di discrepanze tra l'indice di accesso ad un array e la dimensione di quest'ultimo; mentre `label_void` è l'etichetta che indica la posizione nel flusso della funzione che restituisce un errore nel caso in cui non venga trovato, come ultima istruzione di una funzione, un return.

Lo stato viene modificato nel momento in cui: si crea una nuova etichetta e/o un nuovo temporaneo tramite le funzioni `newLabel` e `newTemp`, rispettivamente (per evitare duplicazioni tra le istruzioni) e quando una nuova istruzione TAC di una funzione viene inserita nella lista delle istruzioni di quella funzione (tramite la funzione `out`). Vi è una modifica dello stato nel momento in cui tutti gli statetement di una funzione sono stati tradotti in istruzione TAC: l'insieme delle istruzioni di quella funzione (`funCode` in `FunState`) vengono aggiunte in testa all'insieme delle istruzioni TAC principale (quelle che compongono il TAC in output), tramite la funzione `pushCurrentStream`. Inoltre, lo stato si modifica quando vengono modificate le etichette, il tipo della funzione corrente (`funType` di `FunState`) e i parametri della funzione corrente (`funParams` in `FunState`). In particolare la modifica e accesso dell'etichetta per l'istruzione `continue` (`break`) sono, rispettivamente, `setContinue` (`setBreak`) e `getContinue` (`getBreak`). L'etichetta della funzione di controllo degli array viene gestita tramite le funzioni

`setOutOfBounds` e `getOutOfBounds` e, l'etichetta di gestione del `return` mancance, tramite `setEndOfNonVoid` e `getEndOfNonVoidLabel`. Il tipo della funzione viene manipolato tramite le funzioni `setFunType` e `getFunType`, mentre i parametri, vengono gestiti tramite le funzioni `genFunParams` e `setFunParams`.

Tutte queste funzioni servono per modificare lo stato ed ottenere da esso il valore attuate di uno specifico campo.

Nel file `StateManager.hs` sono presenti tutte le funzioni di manipolazione dello stato e la definizione dell'alias `TacState a` per `State (Int, Int, [TAC], [FunState], (Label, Label), (Label, Label)) a`, dove `TAC` è un'istruzione TAC.

5 Descrizione soluzione

5.1 Il modulo di analisi statica

In questa sezione viene fornita una breve descrizione della logica retrostante il funzionamento del modulo di analisi statica `StaticAnalysis.hs`.

Il modulo di analisi statica fornisce un'interfaccia alle restanti componenti del compilatore, che permette di ottenere un albero di sintassi astratta annotato ed eventualmente una lista di log a partire da un programma **Scala40** rappresentato in sintassi astratta.

Per la gestione dei log si rimanda alla sezione riguardante le tecniche non standard utilizzate.

Il modulo è stato progettato seguendo un approccio *top-down*, andando di volta in volta a risolvere i sotto-problemi in modo ricorsivo.

Il modulo di analisi statica fa largo uso del modulo *Environment.hs* che gli offre un'interfaccia per la gestione dell'environment del programma sotto analisi.

L'environment di un programma è una pila di *scope*, dove ogni scope è formato da: una *tabella di lookup* che mette in corrispondenza gli identificatori definiti all'interno dello scope con le informazioni a riguardo, un *tipo* che eredita dalla funzione in cui lo scope è stato creato ed infine un *valore booleano* che indica se nello scope è presente o meno un `return` con tipo compatibile con quello dello scope.

Il modulo `Environment.hs` fornisce tutte le funzioni necessarie per la gestione dell'environment di un programma. La funzione `lookup` permette di ottenere le informazioni di un identificatore precedentemente dichiarato nello scope corrente, o in un *super-scope* nel quale lo scope corrente è contenuto; nel caso in cui l'identificatore cercato non sia trovato un eccezione viene ritornata. La funzione `update` permette di inserire una nuova corrispondenza *identificatore-info* nello scope corrente. Viene ritornata un'eccezione, nel caso in cui l'identificatore che si sta cercando di inserire fosse già stato dichiarato in precedenza all'interno dello scope. Per ottenere il tipo dello scope corrente è presente la funzione *getScopeType*, la funzione *hasReturn* ritorna invece il valore booleano che indica se nello scope corrente è presente o meno un `return` di tipo coerente con quello

dello scope; quando un tale `return` viene trovato la funzione `setReturnFound` permette di impostare tale valore booleano a `True`. La funzione `addScope` aggiunge un nuovo scope all'environment, prendendo come argomento il tipo dello scope che si sta aggiungendo. Qui è importante porsi una domanda:

Quale tipo viene passato ad `addScope`?

La risposta è: *dipende*. Per convenzione, se lo scope è lo scope globale del programma, esso ha tipo `TSimple_TypeVoid`; se lo scope viene creato da una dichiarazione di funzione, allora il tipo passato ad `addScope` è quello della funzione dichiarata. Infine, se lo scope viene aggiunto durante la creazione di un blocco di statements, allora il tipo passato ad `addScope` è quello dello scope in cui il blocco di statements è racchiuso. Non è difficile vedere che tali accorgimenti verificano la proprietà stabilita per il tipo di uno scope, ossia che esso coincida con il tipo della funzione in cui è racchiuso.

Forniamo ora una breve panoramica del funzionamento del modulo di analisi statica `StaticAnalysis.hs`. La funzione principale del modulo è `typeCheck` che preso in input un programma rappresentato in sintassi astratta di **Scala40**, restituisce tale programma annotato ed una lista di log (eventualmente vuota). Il compito di annotare gli elementi della sintassi astratta e di aggiungere eventuali log alla lista di log ogni qual volta venga rilevata una eccezione, è affidato alle funzioni `infer*` (per esempio: `inferDecl`, `inferStm`, `inferExp`, ecc.), tali funzioni preso un elemento della sintassi astratta effettuano i controlli per verificare che sia coerente con le specifiche del linguaggio; nel caso non lo siano, tramite la funzione `saveLog`, viene aggiornata la lista dei log. Al termine dei controlli l'elemento analizzato viene ritornato arricchito con annotazioni quali: le informazioni sugli identificatori utilizzati e il tipo delle espressioni coinvolte.

Le funzioni `infer*` hanno tutte un comportamento naturalmente ricorsivo, si prenda per esempio il caso di una dichiarazione di variabile con assegnamento di un valore, per poter inferire la dichiarazione e determinare se è valida o meno è necessario inferire l'espressione che si sta cercando di assegnare alla variabile e verificare che i tipi siano compatibili.

È stato deciso che, nel caso in cui si stia inferendo un elemento che coinvolge un'espressione che è già stato verificato essere problematica, non vengono aggiunte al log nuove eccezioni riguardanti l'elemento analizzato. Questa scelta è giustificata dal desiderio di eliminare ridondanza negli errori, dato che un'espressione errata, potrebbe portare ad una lunga serie di errori a catena in tutti gli elementi in cui essa è contenuta.

Per ottenere questo comportamento, è stato inserito nella sintassi astratta un tipo interno `SType_Error` che viene assegnato alle espressioni che generano un'eccezione, o alle espressioni le cui sotto-espressioni generano eccezioni. Se un qualsiasi elemento ha come sotto-espressione una espressione di tipo `SType_Error` non vengono generate ulteriori eccezioni riguardo all'interazione di tale sotto-espressione con l'elemento considerato.

Le annotazioni aggiunte dal modulo di analisi statica riguardano esclusivamente le espressioni e i parametri attuali nelle chiamate di funzione/procedura. Le *R*-espressioni annotate sono identificate dagli elementi di sintassi astratta interni

`ExpTyped` e le L-espressioni annotate sono identificate dagli elementi di sintassi astratta interni `LExpTyped`, entrambi gli elementi condividono la stessa struttura, un'espressione tipata contiene l'espressione, il suo tipo e la locazione in cui viene utilizzata. Una caratteristica ulteriore delle espressioni tipate, è che la locazione contenuta all'interno di un identificatore (presente in un'espressione) non è la locazione di utilizzo, che invece è in un campo apposito, ma è la locazione di dichiarazione dell'identificatore. La locazione di dichiarazione di un identificatore utilizzato all'interno di una espressione risulta essere fondamentale per la successiva fase di generazione del codice TAC.

I parametri attuali annotati sono identificati dagli elementi di sintassi astratta interni `ParExpTyped` che oltre ad avere l'informazione della espressione passata come parametro contengono anche il tipo del corrispondente parametro formale nella funzione/procedura chiamata.

Il modulo `Typed.hs` contiene la definizione degli elementi tipati con le loro proprietà.

5.2 Descrizione soluzione - Three Address Code

Il Three Address Code (TAC) viene costruito a partire dall'albero annotato risultante dall'analisi di semantica statica. Per la costruzione viene fatto largo uso della monade State, descritta nella sezione precedente.

Gli indirizzi utilizzati per le istruzioni del TAC sono dei seguenti tipi:

- Letterali: per valori costanti di tipo base.
- Indirizzi di variabile e di funzioni: hanno la seguente forma:

  `ident@loc`

  dove `ident` è il loro identificatore e `loc` la locazione di dichiarazione.
- Temporanei: utilizzati per identificare espressioni.

Inoltre, nel TAC sono presenti le etichette che vengono utilizzate per indicare una locazione univoca nel flusso di esecuzione. Nel caso delle funzioni, come etichetta, viene utilizzato il loro indirizzo.

La costruzione del Three Address Code inizia dalla funzione `genProg` che ha i seguenti compiti:

- Inserire un'etichetta in fondo a tutte le istruzioni (e il relativo `Call` dopo le dichiarazioni globali) se durante l'analisi di semantica statica non è stato trovato una funzione main da cui far partire la computazione. In alternativa, viene inserita un'istruzione `Goto` all'etichetta del main;
- iniziare la discesa nell'albero annotato per poter costruire le singole istruzioni, tramite la funzione `genDecls`.

La funzione `genDecls` prende in input una lista di dichiarazioni e le scorre producendo le relative istruzioni TAC. Questo avviene tramite la funzione `genDecl`; questa divide le dichiarazioni nei quattro casi possibili:

Tipo	Valore di default
Int	0
Float	0.0
Char	\0
String	""
Bool	False
Pointer	Null
Array	-

Table 7: Valori di default

- Dichiarazione di variabile con assegnamento: viene creato un nuovo indirizzo a partire dall'identificatore della variabile e dalla sua locazione di dichiarazione. Gli viene assegnato un valore tramite la funzione `genExpAddr` che permette di evitare la creazione di temporanei inutili quando ci troviamo nel caso di un'espressione semplice (es. $x = 3$, $*y = x$, $x = y + z$ ecc.) ed esegue i necessari cast.

- Dichiarazione di variabile: come nel caso precente costruiamo un indirizzo a partire da identificatore e locazione, e gli assegnamo un valore di default () tramite la funzione `buildDefaultValue`.

- Dichiarazione di funzione/procedura: in questo caso viene creata una nuova etichetta per la funzione. Viene chiamata la funzione `genBlock` che (tramite `genStms`) controlla la presenta dell'istruzione `return` (nel caso delle procedure verrà aggiunto un'istruzione di return vuoto) e costruisce tutte le istruzioni TAC relative al corpo della funzione. Al termine di questo processo le istruzioni TAC che sono state create vengono estratte tramite operazione di pop dalla lista di liste di istruzioni dello stato e inserite in testa al codice globale del TAC. Questo viene fatto per far sì che una funzione dichiarata nel corpo di un'altra venga, inserita sopra quest'ultima a livello di Three Address Code, e non al suo interno.

  Inoltre, se la dichiarazione si trova nello scope globale e l'identificatore utilizzato è `main`, viene controllato che la funzione abbia la firma specifica `def main()`.

Ogni funzione è costituita da un insieme di statement (contenuti in un blocco). La funzione che si occupa di percorrere tutti gli statement di un blocco è `genStms`, che restituisce `True` se l'ultimo statement è un `return` (per evitare che nel TAC compaiano due istruzioni `return` consecutive). Nel contempo vengono esaminati gli statement singolarmente tramite la funzione `genStm`. Quest'ultima prende in esame tutti i possibili statement:

- Dichiarazione: viene riutilizzata `genDecl` per la gestione delle dichiarazioni interne al blocco.

- Blocchi interni: viene riutilizzata la funzione `genBlock`.

- Assegnamenti: un assegnamento ha la seguente forma:

  `Lexpression = RExpression`

  Le `l-expression` comprendono: accessi ad array, variabili puntatori e identificatori di variabile. La loro gestione avviene tramite la funzione `genLexp`. Le `r-expression` oltre che le `LExpression`, comprendono le espressioni.

- Chiamata di procedura: vengono create le istruzioni che indicano i parametri passati alla funzione (tramite la funzione `genParams` che crea istruzioni `Param addr`) e viene creata un'istruzione `Call` del TAC. La funzione `genParams` si occupa anche di deferenziare la variabile passata alla funzione se il metodo di passaggio richiesto è `ref`, `res`, `valres`. Al momento della stampa, le funzione già definite nel linguaggio (es. `writeInt`) avranno, al posto della locazione specificata nell'indirizzo, la parola `default` (es. `writeInt@default`).

- Return: se l'istruzione `return` ritorna un'espressione essa viene gestita tramite `genExpAddr`, che in caso di espressioni complesse (non binaria tra due identificatori o unaria) restituisce un temporaneo.

- Break e Continue: viene creata un'istruzione `Goto` con associata un'etichetta che indica l'inizio del ciclo in caso di `continue` e l'istuzione subito dopo il ciclo nel caso di `break`.

- For: oltre che all'istruzione di controllo per le varie iterazioni, viene inserito in coda a tutte le istruzioni TAC relative al corpo del ciclo, l'istruzione di incremento dell'iteratore.

- Do-While: viene modificato lo stato con le etichette relative ai `break` e `continue`. In questo caso le etichette create sono tre: quella relativa all'inizio del ciclo (utilizzata dalla condizione di iterazione), quella che indica la posizione del controllo dell'iterazione (utile per il `Goto` del `continue`) e quella che indica l'istruzione successiva alla posizione di controllo (utile per il `Goto` del `break`). Al termine delle istruzioni relative al ciclo, il valore dello stato relativo alle etichette `break` e `continue` viene ristabilito al precedente valore;

- While: come nel caso precedente vengono create le etichette per il `break` e il `continue`, che, vengono gestite come nel caso precedente. In questo caso le etichette create sono solo due: quella che rimanda all'inzio del ciclo e quella che rimanda alla prima istruzione dopo il ciclo;

- If: banalmente vengono create le istruzioni relative agli statement relativi all'`if` e all'`else`. Se il corpo dell'`else` è vuoto, allora l'etichetta `labelElse` dopo le istruzioni del caso `if` non viene generata.

Sia per quanto riguarda gli `If` e i cicli (`For, Do-While` e `While`) la gestione delle etichette avviene tramite la funzione `genCondition`, che, presa l'espressione della condizione e le due etichette che indicano dove spostarsi in caso essa sia vera o meno, gestitisce la creazione dei `Goto` ed esegue il controllo del flusso. La funzione `genCondition` viene utilizzata in particolar modo per evitare controlli superflui (es. nella condizione `a && b` se `a` è falsa, non serve valutare anche `b` e il controllo può passare all'etichetta relativa al caso `False` della condizione).

La generazione delle istruzioni riguardanti le espressioni viene gestita tramite la funzione `genExpAddr` che restituisce l'indirizzo TAC dell'espressione: un letterale nel caso di espressioni composte da un solo letterale o un nuovo temporaneo nel caso di dereferenziazioni, LExpression, accessi ad array o espressioni simili a quelle descritte nel caso dell'assegnamento. Se il tipo dell'espressione tipata restituita durante la fase di analisi statica è diverso (ma sicuramente compatibile) con quello della sottoespressione che si sta analizzando, allora viene eseguito un `cast` implicito.

Terminate tutte le dichiarazioni presenti nello scope globale, vengono inserite in testa alla lista di istruzioni TAC tutte le dichiarazioni di variabili globali seguite dall'istruzione `Goto` alla funzione `main` se presente, oppure all'etichetta `l0` in fondo a tutte le istruzioni (evitando così che tutte le altre funzioni vengano eseguite sequenzialmente).

La stampa ordinata delle istruzioni TAC avviene tramite le funzioni in `PrintTAC.hs`. Per aumentare la leggibilità del codice TAC sono stati aggiunti numerosi commenti, ad esempio quelli relativi all'inizio e fine di una funzione o dei cicli. Relativamente ai commenti, al fianco delle operazioni di addizione, sottrazione, moltiplicazione, divisione, modulo è stato aggiunto un commento con il nome preciso dell'istruzione che sta venendo eseguita, ad esempio `MulInt` per la moltiplicazione tra interi o `PlusFloat` per la somma di float. Ciò è stato fatto per aumentare la comprensibilità delle espressioni, che altrimenti, con i soli simboli `+`, `-`, `*`, `/`, `%`, sarebbero risultate essere indistiguibili nei due casi (Float e Int)