Descrizione del linguaggio Scala40

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

La seguente relazione ha lo scopo di fornire una completa descrizione del liguaggio **Scala40** costruito a partire dal linguaggio **Scala**. Verranno descritte le soluzioni non standard utilizzate, 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. L'ultima sezione della relazione riguarda la desrizione del linguaggio. Verrà presentata la struttura lessicale, la struttura sintattica e la semantica statica del linguaggio.

1 Soluzioni non standard

1.1 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`.

1.2 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 quattro 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 liste di istruzioni TAC. Ogni lista contiene un insieme di istruzioni di una singola funzione.

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`). Inoltre, 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 vengono aggiunte in testa all'insieme delle istruzioni TAC principale (quelle che compongono il TAC in output), tramite la funzione `pushCurrentStream`.

Nel file `ThreeAddressCode.hs` è stato utilizzato `TacState a` come alias per `State (Int, Int, [TAC], [[TAC]]) a`, dove `TAC` è un'istruzione TAC.

2 Descrizione soluzione

2.1 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 combacino.

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 è compresa.

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, sia R-espressioni (elementi `Exp`), sia L-espressioni (elementi `LExp`. Le espressioni annotate sono identificate rispettivamente dagli elementi di sintassi astratta interni: `ETyped` per le R-espressioni e da `LExpTyped` per le L-espressioni. 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.

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

2.2 Descrizione soluzione - Three Address Code

Il Three Address Code (TAC) viene costruito a partire dall'albero annotato risultante dall'analisi di semantica statica (Type Checking). Per la costruzione viene fatto largo uso della monade State, descritta nella sezione precedente. Nel TAC gli indirizzi utilizzati nelle istruzioni sono:

- 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 `Goto` 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:

- 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 `genExpAssign` 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.).

- 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.

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:

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

Table 1: Valori di default

- 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 `LExpression` comprendono: accessi ad array, variabili puntatori e identificatori di variabile. La loro gestione avviene tramite la funzione `genLexp`. Le `RExpression` 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.
- Return: se l'istruzione `return` ritorna un'espressione essa viene gestita tramite `genExp`, che in caso di espressioni complesse (non binaria tra due identificatori o unaria) restituisce un temporaneo.
- If e While: in entrambi i casi la gestione delle etichette viene gestita 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 generazione delle istruzioni riguardanti le espressioni viene gestita tramite la funzione `genExp` 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 (gestite tramite la funzione `genExpAssign`).

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).

3 Descrizione del linguaggio Scala40

Nella descrizione del linguaggio sono evidenziate solo le caratteristiche salienti, omesse le scelte standard e le richieste presenti nella consegna.

4 Struttura lessicale di Scala40

Parole riservate

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

`if`, `else`, `do`, `while`, `def`, `return`, `var`, `Array`, `False`, `True`, `Null`, `Char`, `String`, `Int`, `Float`, `Bool`.

Identificatori

Un identificatore $\langle Ident \rangle$ è una lettera seguita 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.

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

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.

5 Struttura sintattica di Scala40

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

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

  - *Dichiarazione di funzioni e procedure*

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

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

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

  mentre l'elemento $\langle ParamClauses\rangle$ è una sequenza, non vuota, di $\langle ParamClause\rangle$, e ciascun $\langle ParamClause\rangle$ ha la forma

$$\langle ParamClause\rangle ::= \ \texttt{(} \ \langle Args\rangle \ \texttt{)}$$

  $\langle Args\rangle$ è una sequenza, che può essere vuota, di elementi separati da virgola della forma

$$\langle Arg\rangle ::= \ \langle Ident\rangle \ \texttt{:} \ \langle TypeSpec\rangle$$

  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(a: Array[*Int](2),p: *Int)(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.

$$\langle Block\rangle ::= \ \{ \ \langle StmtList\rangle \ \}$$

- Una *istruzione* ha la forma:

$$\begin{aligned}\langle Stmt\rangle ::= &\ \langle Decl\rangle\\ |\ &\langle Block\rangle\\ |\ &\langle LExpr\rangle\ \langle OpAssign\rangle\ \langle Expr\rangle\ \texttt{;}\\ |\ &\texttt{if (}\ \langle Expr\rangle\ \texttt{)}\ \langle Stmt\rangle\\ |\ &\texttt{if (}\ \langle Expr\rangle\ \texttt{)}\ \langle Stmt\rangle\ \texttt{else}\ \langle Stmt\rangle\\ |\ &\texttt{while (}\ \langle Expr\rangle\ \texttt{)}\ \langle Stmt\rangle\\ |\ &\texttt{do}\ \langle Stmt\rangle\ \texttt{while (}\ \langle Expr\rangle\ \texttt{) ;}\\ |\ &\texttt{return ;}\\ |\ &\texttt{return}\ \langle Expr\rangle\ \texttt{;}\\ |\ &\langle Ident\rangle\ \langle ParamsList\rangle\ \texttt{;}\end{aligned}$$

  dove, nell'ultima istruzione, che corrisponde alla chiamata di procedura o funzione, $\langle ParamsList\rangle$ è una sequenza, non vuota, di $\langle Params\rangle$ della forma

$$\langle Params\rangle ::= \texttt{(}\ \langle ExprList\rangle\ \texttt{)}$$

  e $\langle ExprList\rangle$ è una sequenza, che può essere vuota, di $\langle Expr\rangle$ separate da virgola.

  Invece $\langle OpAssign\rangle$ è uno dei seguenti operatori di assegnamento:

$$\langle OpAssign\rangle ::= \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=}$$

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

$$\begin{aligned}\langle LExpr\rangle ::= &\ \langle Ident\rangle\\ |\ &\langle LExpr\rangle\ \texttt{[}\ \langle Expr\rangle\ \texttt{]}\\ |\ &\texttt{*}\ \langle LExpr\rangle\\ |\ &\texttt{(}\ \langle LExpr\rangle\ \texttt{)}\end{aligned}$$

  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:

$$\begin{aligned}\langle Expr\rangle ::= &\ \langle Literal\rangle\\ |\ &\langle LExpr\rangle\\ |\ &\texttt{\&}\ \langle LExpr\rangle\\ |\ &\texttt{Array (}\ \langle ExprList\rangle\ \texttt{)}\\ |\ &\langle Ident\rangle\ \langle ParamsList\rangle\\ |\ &\langle Expr\rangle\ \langle BinOp\rangle\ \langle Expr\rangle\\ |\ &\langle UnOp\rangle\ \langle Expr\rangle\\ |\ &\texttt{(}\ \langle Expr\rangle\ \texttt{)}\end{aligned}$$

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

Gli operatori hanno precedenze e associatività standard.

6 Vincoli di semantica statica di Scala40

Scoping

Il linguaggio ha scoping statico con visibilità dal punto di dichiarazione in poi. Le regole per la visibilità di dichiarazioni locali sono le usuali.

Gli identificatori di variabili e funzioni/procedure devono essere unici. Lo spazio dei nomi è unico, ossia non è permesso dichiarare una variabile e una funzione/procedura con lo stesso nome. Ovviamente questo vale per identificatori con visibilità non disgiunta.

Vincoli riguardanti i tipi

Il linguaggio ha 5 tipi base e 2 costruttori di tipo come indicato nella definizione di ⟨*TypeSpec*⟩. Non sono previste compatibilità fra tipi diversi. Il valore `Null` può essere assegnato solo a left-expression il cui tipo ha un puntatore al top-level.

I tipi degli operatori binari sono i seguenti.

`+`, `-`, `*`, `/`, `^`, `%`	`Int` × `Int` → `Int` o `Float` × `Float` → `Float`
`||`, `&&`	`Bool` × `Bool` → `Bool`
`<`, `<=`, `>`, `>=`	`Int` × `Int` → `Bool` o `Float` × `Float` → `Bool`
`==`, `!=`	$\tau \times \tau \rightarrow$ `Bool` dove τ è un qualsiasi tipo

I tipi degli operatori unari sono i seguenti.

`!`	`Bool` → `Bool`
`-`	`Int` → `Int` o `Float` → `Float`

Le guardie di istruzioni `if`, `do-while` e `while` devono avere tipo `Bool`.

Altri vincoli

TODO. Return.