UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in Informatica

PROGETTO LC2 PARTE 3
GRUPPO 2

Rosario Maione 105606 - maione.rosario@spes.uniud.it
Matteo Bernardon 106027 - bernardon.matteo@spes.uniud.it
Mihai Daniel Iftimie 105576 - iftimie.mihaidaniel@spes.uniud.it

ANNO ACCADEMICO 2014-2015

Analisi di Lasso e assunzioni

Il linguaggio Lasso

Di seguito vengono elencati, per ogni punto critico della sintassi concreta di Lasso, come sono stati rappresentati i vari costrutti nella nostra grammatica in modo da rendere di più facile lettura il resto del documento e i file della realizzazione, aiutando anche a capire le varie scelte progettuali fatte:

- Le variabili vengono dichiarate con la sintassi

  $var(c :: integer = 8)$

  e vengono richiamate anteponendo il carattere $ all'identificatore.

- Quando in un if si usa $else$ questo non prende nessuna condizione, mentre quando si usa l'$else\ if$ questo viene usato in lasso dando all'$else$ una condizione fra parentesi tonde.

- Gli array vengono rappresentati con la sintassi

  $array(4,'c',"casa")$

  Da notare come un array in Lasso sia polimorfo.

- Il ciclo while viene usato con la sintassi

  $while(exp) => \{corpo\}$

- Come costrutto iterativo lasso non prevede il for ma il loop, che viene usato con la sintassi

  $loop(int) => \{corpo\}$.

- Non essendo supportati i puntatori sono stati introdotti seguendo la sintassi e la logica dei puntatori di C.

- Lasso non pervede le modalità di passaggio di parametri value, valueresult e constant, quindi sono state introdotte inserendo la modalità nella dichiarazione della variabile prima dell'identificatore, con la seguente sintassi

  $define\ pippo :: char(constant\ pluto :: char) => \{...\}$

Se la modalità non viene indicata il parametro ha di default la modalità di passaggio per valore.

Assunzioni fatte

Di seguito vengono riportate le assunzioni fatte per quanto riguarda vincoli semantici e sintattici:

- Le procedure *write* e *read* prendono un solo parametro per volta.
- Le funzioni vengono identificate solo dall'identificatore e non dalla firma
- Le dichiarazioni di variabili avvengono sempre una alla volta(in modo da utilizzare la sitassi con *var* di Lasso).
- Può essere dichiarata una funzione all'interno del corpo di un'altra funzione, in questo caso però la funzione interna potrà essere richiamata solo localmente al blocco della funzione esterna.
- Una funzione e una variabile possono avere lo stesso identificatore, questo perchè hanno modalità di chiamata diverse.
- Non viene effettuato alcun casting automatico tra tipi, questo vuol dire che il casting su operatori aritmetici e relazionali viene effettuato (infatti questi hanno più tipi supportati) ma non viene effettuato sugli operatori di assegnamento, come $* =$ o $/ =$.

Grammatica

Di seguito viene presentata la grammatica di Lasso:

Parole riservate e simboli

Le parole riservate usate sono le seguenti:

```
False     True        array
block     boolean     case
char      const       decimal
define    else        goto
if        integer     iterate
loop      loop_abort  loop_continue
main      match       not
pointer   read        return
string    val         valres
var       void        while
write
```

I simboli usati sono i seguenti:

```
=>   {    }
(    )    ||
&&   ==   !=
<    <=   >
>=   +    −
*    /    %
,    &    ++
−−   $    ::
=    :    *=
+=   /=   −=
```

Struttura semantica

⟨*Program*⟩ ::= `main =>` `{` ⟨*ListLine*⟩ `}`

⟨*ListLine*⟩ ::= ϵ
| ⟨*Line*⟩ ⟨*ListLine*⟩

⟨*Boolean*⟩ ::= `True`
| `False`

⟨*RExpr*⟩ ::= ⟨*RExpr1*⟩
| ⟨*RExpr*⟩ || ⟨*RExpr1*⟩

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

⟨*RExpr2*⟩ ::= ⟨*RExpr3*⟩
| `not` ⟨*RExpr3*⟩

⟨*RExpr3*⟩ ::= ⟨*RExpr4*⟩
| ⟨*RExpr4*⟩ == ⟨*RExpr4*⟩
| ⟨*RExpr4*⟩ `!=` ⟨*RExpr4*⟩
| ⟨*RExpr4*⟩ < ⟨*RExpr4*⟩
| ⟨*RExpr4*⟩ <= ⟨*RExpr4*⟩
| ⟨*RExpr4*⟩ > ⟨*RExpr4*⟩
| ⟨*RExpr4*⟩ >= ⟨*RExpr4*⟩

⟨*RExpr4*⟩ ::= ⟨*RExpr5*⟩
| ⟨*RExpr4*⟩ + ⟨*RExpr5*⟩
| ⟨*RExpr4*⟩ − ⟨*RExpr5*⟩

⟨*RExpr5*⟩ ::= ⟨*RExpr6*⟩
| ⟨*RExpr5*⟩ `*` ⟨*RExpr6*⟩
| ⟨*RExpr5*⟩ / ⟨*RExpr6*⟩
| ⟨*RExpr5*⟩ `%` ⟨*RExpr6*⟩

⟨*RExpr6*⟩ ::= ⟨*RExpr7*⟩
| − ⟨*RExpr7*⟩

⟨*RExpr7*⟩ ::= ⟨*RExpr8*⟩
| ⟨*FunCall*⟩

⟨*RExpr8*⟩ ::= ⟨*RExpr9*⟩
| `*` ⟨*LExpr*⟩
| `&` ⟨*LExpr*⟩

⟨*RExpr9*⟩ ::= ⟨*RExpr10*⟩
| ⟨*Integer*⟩
| ⟨*Char*⟩
| ⟨*String*⟩
| ⟨*Double*⟩
| ⟨*Boolean*⟩
| `array (` ⟨*ListRExpr*⟩ `)`

⟨*RExpr10*⟩ ::= `(` ⟨*RExpr*⟩ `)`
| ⟨*LExpr*⟩

⟨*FunCall*⟩ ::= ⟨*Ident*⟩ `(` ⟨*ListRExpr*⟩ `)`

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

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

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

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

⟨*BLExpr*⟩ ::= `$` ⟨*Ident*⟩
| `$` ⟨*Ident*⟩ `(` ⟨*RExpr*⟩ `)`

⟨*Decl*⟩ ::= `var (` ⟨*Ident*⟩ `::` ⟨*Type*⟩ `)`
| `var (` ⟨*Ident*⟩ `::` ⟨*Type*⟩ = ⟨*RExpr*⟩ `)`
| `define` ⟨*Ident*⟩ `::` ⟨*Type*⟩ `(` ⟨*ListParameter*⟩ `) => {` ⟨*ListLine*⟩
`return` ⟨*RExpr*⟩ `}`
| `define` ⟨*Ident*⟩ `:: void (` ⟨*ListParameter*⟩ `) => {` ⟨*ListLine*⟩ `}`

⟨*Type*⟩ ::= `boolean`
| `char`
| `decimal`
| `integer`
| `string`
| `array`
| `pointer` ⟨*Type*⟩

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

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

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

⟨*Line*⟩ ::= `block => {` ⟨*ListLine*⟩ `}`
| ⟨*Ident*⟩ `(` ⟨*ListRExpr*⟩ `)`
| ⟨*JumpStmt*⟩
| ⟨*IterStmt*⟩
| ⟨*SelectionStmt*⟩
| ⟨*LExpr*⟩ ⟨*Assignment-op*⟩ ⟨*RExpr*⟩
| ⟨*Decl*⟩
| `write (` ⟨*RExpr*⟩ `)`
| `read (` ⟨*RExpr*⟩ `)`
| ⟨*Ident*⟩ `:`
| `goto` ⟨*Ident*⟩

⟨*Assignment-op*⟩ ::= =
| *=
| +=
| /=
| −=

⟨*JumpStmt*⟩ ::= `loop_abort`
| `loop_continue`

⟨*SelectionStmt*⟩ ::= `if (` ⟨*RExpr*⟩ `) => {` ⟨*ListLine*⟩ `}`
| `if (` ⟨*RExpr*⟩ `) => {` ⟨*ListLine*⟩ ⟨*Else*⟩ `}`
| `match (` ⟨*RExpr*⟩ `) => {` ⟨*Case*⟩ `}`

⟨*Else*⟩ ::= `else` ⟨*ListLine*⟩
| `else (` ⟨*RExpr*⟩ `)` ⟨*ListLine*⟩
| `else (` ⟨*RExpr*⟩ `)` ⟨*ListLine*⟩ ⟨*Else*⟩

⟨*Case*⟩ ::= `case` ⟨*ListLine*⟩
| `case (` ⟨*ListRExpr*⟩ `)` ⟨*ListLine*⟩
| `case (` ⟨*ListRExpr*⟩ `)` ⟨*ListLine*⟩ ⟨*Case*⟩

⟨*IterStmt*⟩ ::= `while (` ⟨*RExpr*⟩ `) => {` ⟨*ListLine*⟩ `}`
| `loop (` ⟨*RExpr*⟩ `) => {` ⟨*ListLine*⟩ `}`
| `iterate (` ⟨*RExpr*⟩ `) => {` ⟨*ListLine*⟩ `}`

Type System

Assiomi e insiemi usati

Tipi $\dfrac{}{\Gamma \vdash x : T}$ con $x = T \in Type$

booleani $\dfrac{}{\Gamma \vdash Bool : boolean}$

Identificatore $\dfrac{}{\Gamma \vdash Ident \notin Dom(\Gamma)}$

$Type = \{boolean, char, decimal, integer, string, array, pointerType\}$
$Bool = \{True, False\}$
$Modality = \{\epsilon, val, valres, const\}$
$JumpStmp = \{loop\_abort, loop\_continue\}$

Regole

Le regole vengono presentate ruotate di 90 gradi per migliorare la leggibilità una volta stampata la relazione

Vuoto

$$\frac{}{\Gamma \vdash \phi}$$

Program

$$\frac{\Gamma \vdash ListLine}{\Gamma \vdash \mathsf{main} => ListLine}$$

Or

$$\frac{\Gamma \vdash RExpr1 : boolean \qquad \Gamma \vdash RExpr2 : boolean}{\Gamma \vdash RExpr1 || RExpr2 : boolean}$$

And

$$\frac{\Gamma \vdash RExpr1 : boolean \qquad \Gamma \vdash RExpr2 : boolean}{\Gamma \vdash RExpr1 \&\& RExpr2 : boolean}$$

Not

$$\frac{\Gamma \vdash RExpr : boolean}{\Gamma \vdash \mathsf{not} RExpr : boolean}$$

==

$$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 == RExpr2 : boolean} \qquad T1 = T2 \in Type$$

!=

$$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 ! = RExpr2 : boolean} \qquad T1 = T2 \in Type$$

<

$$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 < RExpr2 : boolean}$$

$conT1 \leq T2 se \in \{integer, decimal\}$

con $T1 = T2 se \in char$

<=

$$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 <= RExpr2 : boolean}$$

$conT1 \leq T2 se \in \{integer, decimal\}$

con $T1 = T2 se \in char$

>

$$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 > RExpr2 : boolean}$$

$conT1 \leq T2 se \in \{integer, decimal\}$

con $T1 = T2 se \in char$

>=	$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 >= RExpr2 : boolean}$	$conT1 \leq T2 se \in \{integer, decimal\}$ con $T1 = T2 se \in char$
+	$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 + RExpr2 : T2}$	$T1 \leq T2 \in \{integer, decimal\}$ con $integer < decimal$
-	$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 - RExpr2 : T2}$	$T1 \leq T2 \in \{integer, decimal\}$ con $integer < decimal$
*	$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1 * RExpr2 : T2}$	$T1 \leq T2 \in \{integer, decimal\}$ con $integer < decimal$
/	$\frac{\Gamma \vdash RExpr1 : T1 \qquad \Gamma \vdash RExpr2 : T2}{\Gamma \vdash RExpr1/RExpr2 : decimal}$	$T1, T2 \in \{integer, decimal\}$
%	$\frac{\Gamma \vdash RExpr1 : integer \qquad \Gamma \vdash RExpr2 : integer}{\Gamma \vdash RExpr1\%RExpr2 : integer}$	
negativo	$\frac{\Gamma \vdash RExpr : T}{\Gamma \vdash -RExpr : T}$	$T \in \{integer, decimal\}$
LExpr =	$\frac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr = RExpr : T1}$	$T1 = T2 \in Type$
LExpr *=	$\frac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr* = RExpr : T1}$	$T1 = T2 \in \{integer, decimal\}$
LExpr +=	$\frac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr+ = RExpr : T1}$	$T1 = T2 \in \{integer, decimal\}$
LExpr /=	$\frac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr/ = RExpr : T1}$	$T1 = T2 \in \{integer, decimal\}$

LExpr -=	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr- = RExpr : T1}$	$T1 = T2 \in \{integer, decimal\}$
preincremento	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr = + + RExpr : T1}$	$T1 = T2 \in \{integer, decimal\}$
predecremento	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr = - - RExpr : T1}$	$T1 = T2 \in \{integer, decimal\}$
postincremento	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr = RExpr + + : T1}$	$T1 = T2 \in \{integer, decimal\}$
postdecremento	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr = RExpr - - : T1}$	$T1 = T2 \in \{integer, decimal\}$
()	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash LExpr = (RExpr) : T1}$	$T1 = T2 \in Type$
BLExpr	$\dfrac{\Gamma \vdash Ident}{\Gamma \vdash \$Ident}$	
BLExpr array	$\dfrac{\Gamma \vdash Ident \qquad \Gamma \vdash RExpr : integer}{\Gamma \vdash \$Ident(RExpr)}$	
LExpr BLExpr	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash BLExpr : T2}{\Gamma \vdash LExpr = BLExpr : T1}$	$T1 = T2 \in Type$
Funcall	$\dfrac{\Gamma \vdash Ident \qquad \Gamma \vdash ListRExpr}{\Gamma \vdash Ident(ListRExpr) : T}$	$T \in Type$, T rappresenta il tio di ritorno della funzione
RExpr Funcall	$\dfrac{\Gamma \vdash LExpr : T1 \qquad \Gamma \vdash Funcall : T2}{\Gamma \vdash LExpr = Funcall : T1}$	$T1 = T2 \in Type$

puntatore *

$$\frac{\Gamma \vdash LExpr : Type}{\Gamma \vdash *LExpr : pointerType}$$

puntatore &

$$\frac{\Gamma \vdash LExpr : Type}{\Gamma \vdash \&LExpr : pointerType}$$

var decl

$$\frac{\Gamma \vdash Ident \qquad \Gamma \vdash T : Type}{\Gamma \vdash \mathsf{var}(Ident :: T) : Type}$$

var decl ass

$$\frac{\Gamma \vdash Ident \qquad \Gamma \vdash T : T1 \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash \mathsf{var}(Ident :: T = (RExpr)) : T1} \qquad T1 = T2 \in Type$$

fundecl

$$\frac{\Gamma \vdash Ident \qquad \Gamma \vdash T : T1 \qquad \Gamma \vdash ListParameter \qquad \Gamma \vdash ListLine \qquad \Gamma \vdash RExpr : T2}{\Gamma \vdash \mathsf{define} Ident :: T(ListParameter) => \{Listline \mathsf{return} RExpr\} : Type} \qquad T1 = T2 \in T$$

procdecl

$$\frac{\Gamma \vdash Ident \qquad \Gamma \vdash ListParameter \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{define} Ident :: \mathsf{void}(ListParameter) => \{Listline\}}$$

Parameter

$$\frac{\Gamma \vdash Ident \qquad \Gamma \vdash Modality \qquad \Gamma \vdash T : Type}{\Gamma \vdash ModalityIdent :: T : Type}$$

Block

$$\frac{\Gamma \vdash ListLine}{\Gamma \vdash \mathsf{block} => ListLine}$$

If

$$\frac{\Gamma \vdash RExpr : boolean \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{if}(RExpr) => \{ListLIne\}}$$

If else

$$\frac{\Gamma \vdash RExpr : boolean \qquad \Gamma \vdash ListLine \qquad \Gamma \vdash Else}{\Gamma \vdash \mathsf{if}(RExpr) => \{ListLIneElse\}}$$

else

$$\frac{\Gamma \vdash ListLine}{\Gamma \vdash \mathsf{else} ListLine}$$

else if
$$\frac{\Gamma \vdash RExpr : boolean \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{else}(RExpr) ListLine}$$

else if finale
$$\frac{\Gamma \vdash RExpr : boolean \qquad \Gamma \vdash ListLine \qquad \Gamma \vdash Else}{\Gamma \vdash \mathsf{else}(RExpr) ListLine Else}$$

match
$$\frac{\Gamma \vdash RExpr : Type \qquad \Gamma \vdash Case}{\Gamma \vdash \mathsf{match}(RExpr) => \{Case\}}$$

case
$$\frac{\Gamma \vdash ListLine}{\Gamma \vdash \mathsf{case} ListLine}$$

case listline1
$$\frac{\Gamma \vdash ListRExpr \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{case}(ListRExpr) ListLine}$$

case listline2
$$\frac{\Gamma \vdash ListRExpr \qquad \Gamma \vdash ListLine \qquad \Gamma \vdash Case}{\Gamma \vdash \mathsf{case}(ListRExpr) ListLine Case}$$

while
$$\frac{\Gamma \vdash RExpr : boolean \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{while}(RExpr) => \{ListLine\}}$$

loop
$$\frac{\Gamma \vdash RExpr : integer \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{loop}(RExpr) => \{ListLine\}}$$

iterate
$$\frac{\Gamma \vdash RExpr : array \qquad \Gamma \vdash ListLine}{\Gamma \vdash \mathsf{iterate}(RExpr) => \{ListLine\}}$$

write
$$\frac{\Gamma \vdash RExpr : T}{\Gamma \vdash \mathsf{write}(RExpr)} \qquad T \in \{integer, decimal, char, string\}$$

read
$$\frac{\Gamma \vdash RExpr : T}{\Gamma \vdash \mathsf{read}(RExpr)} \qquad T \in \{integer, decimal, char, string\}$$

Label $\dfrac{\Gamma \vdash \mathit{Ident}}{\Gamma \vdash \mathit{Ident}:}$

goto $\dfrac{\Gamma \vdash \mathit{Ident}}{\Gamma \vdash \mathsf{goto}\mathit{Ident}}$

Descrizione della soluzione

Per ottenere un primo prototipo è stato usato BNFC, raffinando poi manualmente i file prodotti automaticamente dal tool. Di seguito vengono descritti solo i file presentati nella soluzione più significativi e/o modificati dopo la generazione automatica:

- **lasso.cf**
  File usato come input di BNFC. Contiene la descrizione della sintassi astratta nella forma richiesta.

- **Lexlasso.x**
  Contiene il lexer con tutti i token necessari.

- **Parlasso.y**
  Contiene il parser con l'albero sintattico attributato. All'interno vengono implementati i controlli di semantica statica, il type checker e le istruzioni per generare il Three Address Code. Nei prossimi capitoli verranno descritti in dettaglio tutti gli elementi significativi di questo file.

- **Printlasso.hs**
  Pretty printer, modulo di supporto utile a mostrare il programma di input dopo la scomposizione effettuata dall'intero processo. Rispetto a quello generato dal BNFC, non adatto alla nostra grammatica, sono state apportate delle modifiche per rendere la presentazione più chiara e fedele all'input. Abbiamo riscontrato problemi con il pretty printer di default, questo perchè il nostro linguaggio usa caratteri come il $ e non mette i ; alla fine della riga.

- **Testlasso.hs**
  File principale che contiene il main con le direttive per ottenere il risultato delparser, lanciare il pretty printer e mostrare il TAC.

- **DataStructures.hs**
  Modulo di supporto contenente le strutture dati per gli attributi e le funzioni di controllo per il type checker, il pretty printer per il TAC, le strutture dati e le funzioni di controllo per il TAC.

- **Makefile**
  Oltre che a compilare con il comando *make*, è possibile lanciare i test preparate lanciando il comando *make test* associandolo ad un numero da 1 a 6 (i.e. *make test1*)

Type Checker

Attributi per il Type Checker:

- **vEnvLoc e vEnvGlob :: [VarEnv]**
  Con VarEnv =
  | Var Ident Type Scope
  | Const Ident Type Scope

  Ambienti, locale e globale, per le variabili

- **fEnvLoc e fEnvGlob :: [FunEnv]**
  Con FunEnv =
  | Fun Ident Type [Type] Scope
  | Proc Ident [Type]

  Ambienti, locale e globale, per le funzioni

- **lEnvLoc e lEnvGlob :: [LabEnv]**
  Con LabEnv =
  | Lab Ident Scope
  | Got Ident String Scope

  Ambienti, per label e goto

- **typeAtt :: Type**
  Attributo per gestire il tipo di un nodo

- **typeAttList :: [Type]**
  Attributo che raccoglie una lista di attributi, usato ad esempio nel passaggio di parametri.

- **noType :: Bool**
  Attributo per la gestione degli errori di tipo, se il valore è True significa che il nodo o ha un problema sul tipo o ce l'ha uno dei suoi figli. Questa propagazione dell'errore sul tipo serve a mostrare in output l'errore più interno di ogni singola riga, che è molto più significativo del più esterno.

- **trueLexpr :: Bool**
  Attributo usato per sapere se a un nodo di tipo left-expression viene assegnato qualche valore

- **inLoop :: Bool**
  Attributo usato per sapere se il nodo si trova in un ciclo o meno

Three Address Code

Attributi per il Three Address Code:

- **code :: [StructTAC]**
  Con StructTAC =
  | NullOp String String
  | UnOp String String String
  | BinOp String String String String
  | UncondJump Int
  | CondJump String Int
  | RelCondJump String String String Int
  | FunDecl String String Int
  | FunCall String String String Int [String]
  | ArrayLeft String String String
  | ArrayRight String String String
  | LabelT Int
  | Return String
  | LabelProg String
  | GotoProg String

  Attributo che contiene la struttura dati per generare il codice intermedio

- **addr :: String**
  Contiene il nome delle variabili temporanee

- **listAddr :: [String]**
  Lista con nomi di variabili temporanee, viene usata per poter dichiarare i parametri prima di richiamare una procedura o una funzione

- **countDown :: (int,int)**
  Coppia di interi in cui il primo elemento conta le nuove variabili introdotte nel codice intermedio, il secondo invece conta le label introdotte

- **countUp :: (int,int)**
  Stesso meccanismo di countDown ma necessario al passaggio di attributi tra nodi.

Il TAC presentato non è completo, per ragioni di tempo alcune istruzioni non vengono tradotte in codice intermedio. Per altre invece è stata implementata tutta la logica di generazione e inserimento nella struttura dati, ma non siamo riusciti a completare l'ultima sessione di debugging. Abbiamo inoltre riscontrato difficoltà più serie, come il fatto di aver seguito la sintassi di Lasso per quanto riguarda gli array polimorfi, questa scelta si è rivelata solo in un secondo momento sbagliata, questo perchè la gestione di più

tipi, e il conseguente calcolo dello spazio per ogni posizione dell'array, è risultata molto complessa.