Università degli Studi di Udine

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

Corso di Laurea Magistrale in Informatica

Relazione del progetto
per l'insegnamento
Linguaggi e compilatori

ProgettoLC parte5
Gruppo 2

Candidati:

Filippo Callegari

callegari.filippo@spes.uniud.it

Matricola 128602

Talissa Dreossi

dreossi.talissa@spes.uniud.it

Matricola 128080

Luca Rosso

rosso.luca.1@spes.uniud.it

Matricola 127888

Professore dell'insegnamento:

Marco Comini

Anno Accademico 2019–2020 - Secondo Semestre

1 Scelte implementative

L'implementazione del auL, seppur semplice, denota alcune particolarità, dovuta, tra le altre cose, a voler utilizzare meno attributi possibili in Happy, e di creare un linguaggio solido.

La valutazione della dereferenziazione viene effettuata leggendo la parte di AST creata, dando precedenza all'array e successivamente al puntatore, in quanto il tipo complesso (ad esempio, *Int[]) viene memorizzato come array di puntatori.

Viene creato e manipolato passo-passo un SDT: questo permette di creare sia l'AST che il TAC contemporaneamente.

In auL sono presenti le modalità di passaggio dei parametri by value (val), by constant (const), by result (res) and by value/result (valres) e riferimento (gestito come definizione di Higher-Type). Se la modalità non è definita, verrà usata la modalità di default che è by reference per array e stringhe e by value per tutti gli altri tipi.

Un programma auL è composto da una lista di Statement e/o delle funzioni: il file per essere accettato deve contenere almeno una delle due. All'interno del blocco delle funzioni, invece, similmente ai blocchi degli statement, non è richiesto sia presente alcuna riga di codice.

2 Lessico di auL

in controllo sintattico viene lasciato completamente al compilatore. Noi abbiamo sviluppato solamente la gestione degli errori sintattici. si definisce quindi che:

- un'etichetta auL sia quindi definita come
  `token LIdent (letter|'_'lower)(letter|digit|'_')*;;`
- i letterali che rappresentano valori costanti di tipo intero, floating point, carattere e stringhe seguono le normali convenzioni, i letterali false e true rappresentano i booleani, `'nil'` per il valore dei puntatori.

Parole riservate

Le parole riservate in auL sono:

```
Bool       Char       Float       Int
String     Void       and         break
const      continue   do          else
elseif     end        false       for
function   if         local       nil
not        or         readChar    readFloat
readInt    readString repeat      res
return     then       true        until
val        valres     while       writeChar
writeFloat writeInt   writeString
```

Simboli riservati

I simboli riservati sono:

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

Commenti

Sono presenti sia commenti in-line che commenti multi-line. Sono definiti quindi dalle seguenti espressioni regolari

```
-- Toss single line comments
"--" [.]* ;
-- Toss multi line comments
"--[" ("=")* "[" ([$u # [\]]])* ("]") ("=")* "]" ;
```

3 Sintassi di auL

In auL un programma è definito come una lista non vuota di statement o funzioni.

```
Program ::= [PGlobl] ;
PGlobl ::= Stm ;
PGlobl ::= FuncD ;
```

Le dichiarazioni di entità dell'ambiente sono definite quindi come `Decl`, `Local`, ed in ambiente globale, come `FuncD`:

```
Decl ::= BasicType LExp VarInit ;

BasicType ::= "Bool" | "Char" | "Float"
            | "Int" | "String" | "Void" ;

LExp ::= LIdent | "*" LExp | LIdent [Dim] ;

Dim ::= "[" RExp "]" ;

VarInit ::= {- empty -} | "=" RExp | "=" Array ;

Array ::= "{" [Array] "}" | "{" [VType] "}" ;

-- le regole interne a VType sono valori assumibili
VType ::= Boolean | Char | Double | Integer | String | PtrVoid;

Local ::= "local" Decl;
```

```
FuncD ::= CompoundType "function" LIdent "(" [ParamF] ")" Block "end";

ParamF ::= Modality BasicType LExp;

CompoundType ::= BasicType | "*"  CompoundType
               |  BasicType [Bracks] ;
Bracks ::= "[" "]";
```

Nella manipolazione della left expression, si dà priorità alla dereferenziazione come array, per essere seguita dalla dereferenziazione per "`*`". Se fosse possibile operare con i puntatori, sommando o moltiplicando, la dereferenziazione come array sarebbe zucchero sintattico per l'espressione `*(*(Ptr +i)+j)` .

Gli statement sono definiti come:

```
Stm ::= Decl ";";
Stm ::= Local ";";
Stm ::= LExp "=" RExp ";";
Stm ::= "while" RExp EBlk ;
Stm ::= "repeat" Block "until" RExp ";";

Stm ::= "for" LIdent "=" RExp "," RExp Increment EBlk ;
Increment ::= {-empty-} | "," RExp ;

Stm ::= "if" RExp "then" Block [ElseIf] Else "end" ;
ElseIf ::={-empty-} | "elseif" RExp "then" Block ;
Else ::={-empty-} | "else" Block;

Stm ::= FuncWrite ";" | FuncRead ";" | Func ";";
FuncWrite ::= "writeInt" "(" RExp ")"
            | "writeFloat" "(" RExp ")"
            | "writeChar" "(" RExp ")"
            | "writeString" "(" RExp ")";
FuncRead  ::= "readInt" "(" ")"
            | "readFloat" "(" ")"
            | "readChar" "(" ")"
            | "readString" "(" ")";
Func      ::= LIdent "(" [RExp] ")";

Stm ::= EBlk ;
EBlk ::= "do" Block "end";

Stm ::= "return" RValue ";";
Stm ::= "break" ";";
Stm ::= "continue" ";";

Block ::= [Stm];
```

Per motivi di getsione dell'ambiente si è deciso di definire una regola fittizia (e quindi non presente in BNFC ma solo in Happy) di `Block`: questa permette quindi di controllare e gestire il return nelle funzioni. Il blocco definito all'interno di uno statement o dichiarazione di funzione è valido anche se vuoto, mentre la lista `PGlob` non sarà mai accettata se non è presente almeno uno statement o una funzione. Lo statement di `return` diventa, quindi, disponibile solo all'interno di funzioni.

Il `break` ed il `continue` vengono resi disponibile all'utilizzo solo all'interno di cicli indeterminati.

Si nota quindi come non sia possibile eseguire in qualunque ambiente la sola esecuzione di una right expression, ma si possa eseguire qualunque genere di funzione.

Il `For` ha opzionale la parte di `Increment`: nel caso in cui non sia presente si assume che l'incremento sia di `1 : Int`.

La condizione di iterazione per il `for`

```
for a = 1,10,2 do
  [Stm]
end
```

diventa quindi equivalente alla scrittura

```
Int a = 1;
while (a < 10) do
  [Stm]
  a = a + 2;
end
```

Si dettagliano ulteriormente alcune peculiarità implementative nel capitolo omonimo.

Le right expression sono definite quindi come segue:

```
Rexp ::= RExp ? RExp : RExp
       | RExp Op2 RExp
       | Op1 Rexp
       | FuncRead
       | Func
       | LExp
       | ::Integer::
       | ::Float::
       | ::String::
       | ::Char::
       | nil
       | true
```

```
        | false
        | ( RExp ) ;

Op2 ::= or | and
      | == | ~= | > | < | >= | <=
      | + | - | * | / | % | ^
      | .. ;

Op1 ::= not
      | -
      | # ;
```

Precedenza degli operatori

Mostriamo quindi le associatibità e le precedenze per ogni operatore. Si ricorda che 0 è il valore più basso (e quindi ha la precedenza minima) e 10 è il valore più alto (precedenza massima).

Operatore	Precedenza	Associatività
operatore condizionale (? :)	0	non associativo
or	1	sx
and	2	sx
not	3	sx
==, ∼=, >, >=, <, <=	4	non associativo
+, -	5	sx
∗, /, %	6	sx
^	7	dx
- (unario)	8	sx
#	9	sx
&	10	sx

4 Scoping e binding di auL

Lo scoping di auL è prettamente statico, similmente a C. È possibile definire quindi sole funzioni ricorsive semplici e non mutuamente ricorsive. Il binding del linguaggio, essendo compilato, è statico.

La visibilità dei nomi, siano queste di funzioni o di variabili, è immediatamente successivo alla loro dichiarazione ed a tutti i loro blocchi innestati (a meno di redifinizione nei suoi blocchi). È possibile bypassare la definizione di un qualunque nome attraverso lo statement *local*: local infatti permette di **definire una variabile** riutilizzando un nome già utilizzato, anche nel medesimo blocco.

Non è possibile dichiarare funzioni innestate: quando si entra in un blocco legato ad un blocco, si perde la possibilità di definirne.

Non è possibile definire variabili locali come costanti, ma è possibile definire parametri di funzione come tali.

La definizione di variabili (ed array o puntatori) permette l'inizializzazione. Unico vincolo è che durante l'inizializzazione non è possibile autoriferirsi (a meno non ci si trovi in una situazione di utilizzo dello statement *local*). La definizione di una variabile senza conseguente inizializzazione crea un valore generico al suo interno (undefined behaviour), ed in sè non genera un errore. Non si richiede che il valore di inizializzazione sia noto a *compile time*.

5 Tipi e regole di auL

5.1 Definizione dei tipi

In auL abbiamo identificato i seguenti tipi di base

- Int
- Char
- String
- Float
- Boolean

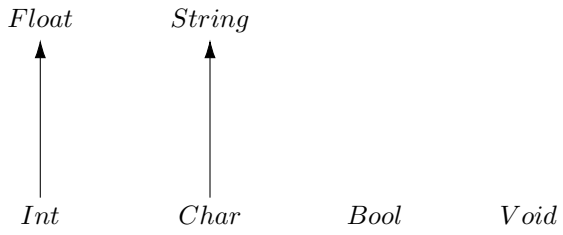
Inoltre c'è la possibilità di definire array di dimensione prefissata e di definire dei puntatori a tipo qualsiasi. Questi sono quindi noti come "higher-types".

Abbiamo anche definito il tipo Void per gestire il tipo di ritorno delle funzioni che non restituiscono alcun valore.

5.2 Compatibiltà tra i tipi

Definiamo come *tipi canonici* i tipi $C_\tau = \{Int, Float, Bool, Char, String\}$.

Mostriamo quindi la gerarchia di tipo:



Quindi deduciamo che:

- se assegniamo un *Int* ad un *Float*, questo sarà accettato;
- ogni C_τ accetterà ogni C_τ;
- se τ_1 è un *higher-type* puntatore, questo accetterà un τ_2 *higher-type* array, a patto che l'"*altezza*" del puntatore sia la medesima dell'"*altezza*" e $C_{\tau_1} = C_{\tau_2}$;
- per completezza nel manipolare il tipo canonico "*String*" si permette di riferire una singola lettera di questa come fosse un array di "*Char*", ma non sono consentite operazioni tra "*Char*" e "*String*"[1].

6 Type system

6.1 Regole per le costanti

Sia "a" un valore per un tipo "τ". Allora

$$\frac{\exists a \in \{\tau : Int, \tau : Bool, \tau : Float, \tau : Char, \tau : String\}}{\Gamma \vdash_{val} a : \tau}$$

Esemplifichiamo ulteriormente "a" un valore per un tipo "$\tau : Bool$":

$$\frac{a = \text{`}true\text{'}: \tau \qquad \tau : Bool}{\Gamma \vdash_{val} a : \tau}$$

$$\frac{a = \text{`}false\text{'}: \tau \qquad \tau : Bool}{\Gamma \vdash_{val} a : \tau}$$

Esemplifichiamo ulteriormente "a" un valore per un tipo puntatore a "$\tau : C_\tau$":

$$\frac{a = \text{`}nil\text{'}: \tau \qquad \tau : Pointer(C_\tau)}{\Gamma \vdash_{val} a : Pointer(C_\tau)}$$

6.2 Regole per le Right-Expressions

S.p.d.g. siano a e b delle *Right expression*. Esemplifichiamo quindi il type system per le *Right Expressions*.

- $\tau \in \{Int, Float\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau \qquad -a : \tau}{\Gamma \vdash_{rexp} -a : \tau}$$

[1] Si è deciso in comune accordo che la dereferenziazione di una stringa generi **comunque** un tipo "*Char*", in quanto non sarebbe altrimenti possibile operare in nessuna maniera con le stringhe, come, ad esempio, per far diventare maiuscola una stringa minuscola. Per togliere tale possibilità è sufficiente commentare riga 232 del file "*Env.hs*".

- $\tau_1, \tau_2 \in \{Int, Float, Char\}$, $\tau_\diamond$ rispetta la gerarchia di tipo e $\odot \in \{+, -\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau_1 \qquad \Gamma \vdash_{rexp} b : \tau_2 \qquad a \odot b : \tau_\diamond}{\Gamma \vdash_{rexp} a \odot b : \tau_\diamond}$$

- $\tau_1, \tau_2 \in \{Int, Float\}$, $\tau_\diamond$ rispetta la gerarchia di tipo e $\odot \in \{*, \div, \hat{}\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau_1 \qquad \Gamma \vdash_{rexp} b : \tau_2 \qquad a \odot b : \tau_\diamond}{\Gamma \vdash_{rexp} a \odot b : \tau_\diamond}$$

- $\tau_1, \tau_2 \in \{Int, Float\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau_1 \qquad \Gamma \vdash_{rexp} b : \tau_2 \qquad a\%b : Int}{\Gamma \vdash_{rexp} a\%b : Int}$$

- $\tau \in \{String\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau \qquad \Gamma \vdash_{rexp} b : \tau \qquad a..b : String}{\Gamma \vdash_{rexp} a..b : String}$$

- $\tau \in \{Int, Float, Char\}$ e $\odot \in \{\leq, \geq, <, >\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau \qquad \Gamma \vdash_{rexp} b : \tau \qquad a \odot b : Bool}{\Gamma \vdash_{rexp} a \odot b : Bool}$$

- $\tau_1, \tau_2 = C_\tau$ e $\odot \in \{==, \neq\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau_1 \qquad \Gamma \vdash_{rexp} b : \tau_2 \qquad a \odot b : Bool}{\Gamma \vdash_{rexp} a \odot b : Bool}$$

- dove τ è puntatore a C_τ e $\odot \in \{==, \neq\}$

$$\frac{\Gamma \vdash_{val} a : \text{`}nil\text{'} \qquad \Gamma \vdash_{rexp} b : \tau \qquad a \odot b : Bool}{\Gamma \vdash_{rexp} a \odot b : Bool}$$

- il caso riflessivo per a e b a quello sopra;
- τ_1, τ_2 sono puntatore a C_τ e $\odot \in \{==, \neq\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau_1 \qquad \Gamma \vdash_{rexp} b : \tau_2 \qquad a \odot b : Bool}{\Gamma \vdash_{rexp} a \odot b : Bool}$$

- τ è puntatore a C_τ

$$\frac{\Gamma \vdash_{rexp} a : \tau \qquad \#a : Int}{\Gamma \vdash_{rexp} \#a : Int}$$

- $\odot \in \{and, or\}$

$$\frac{\Gamma \vdash_{rexp} a : Bool \qquad \Gamma \vdash_{rexp} b : Bool \qquad a \odot b : Bool}{\Gamma \vdash_{rexp} a \odot b : Bool}$$

- regola unaria del not

$$\frac{\Gamma \vdash_{rexp} a : Bool \qquad not\ a : Bool}{\Gamma \vdash_{rexp} not\ a : Bool}$$

- siano $a_1, \ldots, a_n$ espressioni, id identificativo di una funzione con arietà n e $\tau_\diamond \in C_\tau$

$$\frac{\Gamma \vdash_{rexp} a_1 : \tau_1, \ldots, \Gamma \vdash_{rexp} a_n : \tau_n \qquad \Gamma \vdash_{rexp} id(a_1 : \tau_1 \times \ldots \times a_n : \tau_n) : \tau_\diamond}{\Gamma \vdash_{rexp} id(a_1, \ldots, a_n) : \tau_\diamond}$$

- id identificativo di una variabile, τ il suo tipo (sia higher-type che C_τ)

$$\frac{\Gamma \vdash_{lexp} id : \tau}{\Gamma \vdash_{rexp} id : \tau}$$

- id identificativo di una variabile, τ il suo tipo (sia higher-type che C_τ) e $*\tau$ il puntatore a τ

$$\frac{\Gamma \vdash_{lexp} id : \tau}{\Gamma \vdash_{rexp} \&id : *\tau}$$

- sia $\phi \in \{readInt() : Int, readFloat() : Float, readChar() : Char, readString() : String\}$ funzioni del linguaggio per ricevere input dal mondo esterno, ed associamo loro il tipo di ritorno

$$\frac{\Gamma \vdash_{rexp} \phi() : \tau}{\Gamma \vdash_{rexp} \phi() : \tau}$$

- val identificativo di un valore, τ il suo tipo (sia higher-type che C_τ)

$$\frac{\Gamma \vdash_{val} val : \tau}{\Gamma \vdash_{rexp} val : \tau}$$

- sia c un ulteriore *R. Exp.*. Allora, definito l'operatore condizionale si avrà:

$$\frac{\Gamma \vdash_{rexp} a : Bool \qquad \Gamma \vdash_{rexp} b : \tau \qquad \Gamma \vdash_{rexp} c : \tau}{\Gamma \vdash_{rexp} a\ \texttt{?}\ b\ \texttt{:}\ c : \tau}$$

6.3 Regole per le Left-Expression

S.p.d.g. sia a una Left Expression. Esemplifichiamo il type system per le Left Expression.

- sia $*\tau : Pointer(C_\tau)$

$$\frac{\Gamma \vdash_{lexp} a : *\tau}{\Gamma \vdash_{lexp} *a : \tau}$$

- sia $\tau[\,] : Array(C_\tau)$, e sia $E \dashv_{rexp} \Gamma : Int$

$$\frac{\Gamma \vdash_{lexp} a : \tau[\,] \qquad \Gamma \vdash_{rexp} E : Int}{\Gamma \vdash_{lexp} a[E] : \tau}$$

- sia $*\tau : Poiter(C_\tau)$, e sia $E \dashv_{rexp} \Gamma : Int$

$$\frac{\Gamma \vdash_{lexp} a : *\tau \qquad \Gamma \vdash_{rexp} E : Int}{\Gamma \vdash_{lexp} a[E] : \tau}$$

6.4 Regole per la definizione di funzioni

S.p.d.g. sia C un insieme (anche vuoto) di Statement. Esemplifichiamo il type system per la definizione di funzioni.

- sia $\tau \in \{HT(C_\tau), C_\tau \cup \{Void\}\}$, Γ l'*Environment* e $\hat{\Gamma}$ l'*environment* nuovo

$$\frac{\Gamma, id \vdash \emptyset : \bot^2 \qquad \Gamma \vdash_{cmd} C : Void}{\Gamma \vdash_{fnct} id(a_1 : \tau_1, \ldots, a_n : \tau_n)\{C\} : \tau \Rightarrow (\Gamma, id(a_1 : \tau_1 \times a_n : \tau_n) : \tau) \models \hat{\Gamma}}$$

$$\frac{\Gamma, id \vdash \emptyset : \bot \qquad \Gamma \vdash_{cmd} C : Void}{\Gamma \vdash_{fnct} id(\,)\{C\} : \tau \Rightarrow (\Gamma, id(\,) : \tau) \models \hat{\Gamma}}$$

2. con "$\Gamma, id \vdash \emptyset : \bot$" si indica che non è mai stata definita nessuna funzione o variabile con nome definito in "id".

6.5 Regole per la definizione di variabili

S.p.d.g. sia Γ l'*Environment* e $\hat{\Gamma}$ l'*environment* nuovo. Esemplifichiamo il type system per la definizione di funzioni.

- sia $\tau \in \{HT(C_\tau), C_\tau\}$

$$\frac{\Gamma, id \vdash \emptyset : \bot}{\Gamma \vdash_{var} id : \tau \Rightarrow (\Gamma, id : \tau) \models \hat{\Gamma}}$$

$$\frac{\Gamma, id \vdash \emptyset : \bot \qquad \Gamma \vdash_{rexp} b : Int}{\Gamma \vdash_{var} id[b] : \tau \Rightarrow (\Gamma, id : \tau) \models \hat{\Gamma}}$$

- sia $\tau \in \{Pointer(C_\tau), C_\tau\}$

$$\frac{\Gamma, id \vdash \emptyset : \bot \qquad \Gamma \vdash_{rexp} E : \tau}{\Gamma \vdash_{var} id = E : \tau \Rightarrow (\Gamma, id : \tau) \models \hat{\Gamma}}$$

- sia $\tau : HT(C_\tau)$

$$\frac{\dfrac{\Gamma \vdash_{val} a_i : \tau, \forall i \in [1, n] \qquad \Gamma \vdash_{val} E = \{a_1 : \tau \times \ldots \times a_n : \tau\} : \tau}{\Gamma \vdash_{val} E : \tau} \qquad \Gamma, id \vdash \emptyset : \bot \qquad \Gamma \vdash_{rexp} b : Int}{\Gamma \vdash_{var} id[b] = E : \tau \Rightarrow (\Gamma, id : \tau) \models \hat{\Gamma}}$$

6.6 Regole per i comandi

S.p.d.g., per tutti i comandi appartenenti a "*auL*", possiamo affermare che il tipo che ritorna sia sempre e solo indefinito.

$$\frac{}{\Gamma \vdash_{cmd} C : \bot}$$

7 Altri vincoli del auL

- Una chiamata di funzione in una espressione non può avere tipo di ritorno void, perché le espressioni non possono avere tipo void;
- **break** e **continue** possono comparire solo nel corpo di un loop indeterminato. In caso di loop annidati, questi statement si riferiscono al ciclo più interno;
- poichè il return è prerogativa nelle sole funzioni, e non è possibile verificare che venga eseguito ogni return dichiarato nei blocchi (come "*if-then-elseif-else*", si è deciso che sia sufficiente lasciare che c'è almeno un blocco che lo contiene perchè la funzione di tipo non-*Void* abbia il return richiesto. Rimane a carico del programmatore garantirne il corretto funzionamento;
- nel caso in cui il tipo della funzione sia definito come *Void*, si controlla solo che il *return* non ritorni effettivamente nulla. Oltre questo, viene considerato valido se in questo genere di funzioni non compaia il return.
- viene controllato, ad ogni chiamata di funzione, che il numero di parametri attuali coincida con il numero di parametri formali, e che i tipi siano compatibili;
- non è possibile sovrascrivere una costante, perciò negli assegnamenti si controlla che la lhs non sia una costante;

- sono implementati i passaggi per riverimento, valore, risultato, costante, valore/risultato. Il passaggio per valore è definibile attraverso alla dichiarazione dei parametri di funzione del valore in ingresso come puntatore (e quindi "*"). I vincoli sulle "modalità" del parametro sono che se la variabile è definita come `costante` non sarà possibile modificare il suo valore, mentre se la variabile è in modalità "risultato" non sarà possibile accedere al suo valore.

8 Peculiarità implementative

Cicli determinati ed inderminati

Nell'implementazione si è scelto di inserire i 2 costrutti canonici di ciclo indeterminato, quali "`while`" e "`repeat`" e il ciclo determinato quale "`for`". Quest'ultimo ha una particolarità: non viene richiesto un tipo esplicito nella definizione della variabile locale usata poichè verrà utilizzato come tipo quello della prima espressione (ovvero il tipo che inizializza la variabile). Questo permette di poter creare dei cicli come

```
[...]
  for a = 0.5, 10 do
    writeFloat(a);
  end
[...]
```

senza doversi preoccupare del tipo.

Operatore condizionale o "if ternario"

L'implementazione dell'operatore condizionale è stata legata al fatto che questo ritorna un tipo e quindi è legato ad una *R. Expr.* viene quindi inserito nelle *R. Expr.*. In "Lua" non esiste normalmente questo operatore, ed è stato scelto di implementarlo, come per il C:

```
[...]
  LExp = RExp ? RExp : RExp;
[...]
```

la prima *R. Expr.* sarà legata ad un test che sia di tipo "*Bool*", mentre le altre due espressioni è richiesto che abbiano necessariamente tipi tra loro compatibili.

La scelta della costruzione sintattica di questo operatore è legata alla decisione di non introdurre conflitti nel parser.

9 TAC

Per la gestione del TAC si è definito un tipo di dato TAC come segue:
`data TAC = Rules RulesTac | LRules LabelTac RulesTac`

```
deriving (Eq, Show)
```

Ovvero `TAC` può essere o una regola oppure una regola etichettata da una label (`LabelTac` è infatti una stringa): quello che restituiremo all'interno del parser sarà una lista di `TAC` perchè appunto prevediamo la presenza di più regole. Le `RulesTac` sono le possibili istruzioni del linguaggio tradotte in TAC:

- `AssgmBin` e `AssgmUn` sono rispettivamente l'assegnamento di un'operazione binaria, di un'operazione unaria mentre e `Assgm` è l'assegnamento generico del tipo x = y.
- `Cast a1 t1 t2 a2` indica che sto facendo un cast a `a2` da `t2` a `t1` il cui risultato viene assegnato a `a1`
- `VarDecl` è la dichiarazione di variabile, come `Local` è la dichiarazione di variabile locale
- `Goto` è l'istruzione che mi indica su quale label spostarmi, mentre `CondTrue` e`CondFalse` mi servono rispettivamente ad indicare su quale label spostarmi se la condizione è vera o falsa
- `ProcCall` e `FuncCall` sono rispettivamente chiamata a procedura (a cui passo il numero di argomenti ma che non assegno a nessun altro temporaneo) e la chiamata a funzione (che quindi assegna il risultato ad un temporaneo)
- `Func` serve a indicare la dichiarazione di funzione e il numero di parametri che vengono poi caricati da `Load`
- `ArrayDef` indica la dichiarazione di un array
- `ListDimension` indica la lista delle dimensioni degli array
- `ArrayEl` viene usato per gli assegnamenti del tipo x = y[i] o y[i] = x
- `ListElem` e `ListRexp` servono per la gestione delle liste di elementi degli array e delle liste degli argomenti per le chiamate a funzioni
- `AssignAddress` e `AssignPointer` vengono usati rispettivamente per istruzioni del tipo x = &y e x = *y, mentre `DerefAssign` per quelle del tipo *x = y
- `NoOperation` è la semplice operazione vuota
- `ReturnTac`, `Break`, `Comment` e `Error` sono rispettivamente l'istruzione di return, di break, di commento e di errore
- `GetArg` viene utilizzato per il caricamento degli argomenti nelle operazioni

Il tipo di dato `TAC` ha poi bisogno di altri tipi di dato tra cui ricordiamo i più rilevanti:

- `ArgOp` che può essere un nome di variabile (con relativa posizione), un temporaneo, un intero, un float, un booleano, un carattere o una stringa

- `BinaryOp` e `UnaryOp` sono operazioni binarie e unarie con relativo tipo, mentre `RelationOp` sono le operazioni di relazione anch'esse con relativo tipo

Il tipo `State` serve per tenere traccia dello stato corrente di temporanei e label. Può essere modificato tramite `skipState` nel momento in cui viene utilizzato un temporaneo o una label così da non riutilizzarle.
Per la generazione del TAC è stato necessario, come vedremo nella prossima sezione, definire degli attributi per la grammatica.

10 Implementazione di auL

Dapprima si è iniziato a definire una grammatica completa di tutte le feature desiderate da auL attraverso l'ausilio di BNFC. Successivamente, dopo aver migrato la grammatica di base di Happy generata da BNFC ad una attributata, si è iniziato a definire l'environment (*Env.hs*). Finito di sviluppare le regole base per le dichiarazioni di tipi canonici e le right-expression, si è implementata in maniera iterativa tutte le altre funzionalità pensate per il linguaggio.

La grammatica attributata di Happy si compone di:

- alcuni attributi per la generazione dell' AST;
- alcuni attributi per la gestione dell'environment, tra cui il tipo uscente dalle RExp, l'env. locale, env. esterno e env. uscente (ovvero le eventuali modifiche dell'environment locale). Il controllo del tipo e della presenza del return in una funzione viene effettuato attraverso la manipolazione dell'env. loc del blocco-funzione, dove si controlla se effettivamente viene trovato o meno, e se è del tipo desiderato.
- gli attributi per il TAC, in particolare
  - *code* che restituisce il TAC
  - *condTrue* e *condFalse* che restituiscono le label su cui spostarsi in caso che la condizione sia vera o falsa
  - *statein* e *stateout* che indicano lo stato in entrata e quello in uscita
  - *addr* che indica l'`ArgOp` dell'elemento
  - *nextLabel* che indica la label al blocco di istruzioni successivo (es. se mi trovo in un ciclo while, la *nextLabel* indica la label per uscire dal ciclo)
  - *listDim*, *listElem* e *listRexp* che sono liste di `ArgOp` per la dimensione degli array, per gli elementi degli array e per gli argomenti delle funzioni.

Parallelamente, dopo aver sviluppato l'environment, si è iniziato a scrivere il file per il TAC e la sua gestione.

11 Conflitti

Non sono presenti conflitti nella grammatica definita da BNFC.