ProgettoLC parte5parz Gruppo 2 Relazione

Gruppo 2

September 16, 2020

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 mi permette di creare quindi sia un AST che un TAC contemporaneamente.

In auL sono presenti le modalità di passaggio dei parametri by value (val), by reference (ref), by constant (const), by result (res) and by value/result (valres). 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       do          else        elseif
end         false       for         function
if          in          local       name
nil         not         or          readChar
readFloat   readInt     readString  ref
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 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. Lo scoping 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 la regola del *local*: local infatti permette di **definire una variabile** riutilizzando un nome già utilizazato, anche nel medesimo blocco.

Non è possibile dichiarare funzioni innestate: quando si entra in un blocco legato ad uno statement, si perde la possibilitò 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 la loro inizializzazione: limite dell'inizializzazione è che non possono autoriferirsi durante la loro definizione (a meno non ci si trovi in una situazione di utilizzo dello statement *local*). La loro non 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*.

4 Tipi e regole di auL

4.1 Definizione dei tipi

In auL abbiamo identificato i seguenti tipi di base

- Int
- Char
- String
- Float
- Boolean

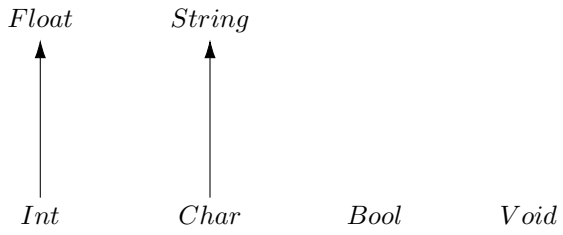
Inoltre c'è la possibilità di definire array di dimensione prefissata e di definire dei puntatori a tipo qualsiasi. Questi sono quindi noti come "higher-types".

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

4.2 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;
- se "*dereferenziamo*" un tipo *String* attraverso come ad un array, questo accetterà un carattere;
- 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 $C_{\tau_1} = C_{\tau_2}$.

5 Type system

5.1 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 "$\tau : C_\tau$":

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

5.2 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*.

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

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

- $\tau_1, \tau_2 \in \{Int, Float, Char\}$, $\tau_\diamond$ 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 $\odot \in \{*, \div, \hat{}\}$

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

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

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

- $\tau_1, \tau_2 \in \{Char, String\}$

$$\frac{\Gamma \vdash_{rexp} a : \tau_1 \qquad \Gamma \vdash_{rexp} b : \tau_2 \qquad a..b : String}{\Gamma \vdash_{rexp} a..b : String}$$

- $\tau_1, \tau_2 = C_\tau$ e $\odot \in \{\leq, \geq, <, >, ==, \neq\}$

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

- 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)

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

5.3 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 $E \dashv_{rexp} \Gamma : Int$

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

5.4 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

$$\frac{\Gamma, id \vdash \emptyset : \bot^1 \qquad \Gamma \vdash_{cmd} C : Void}{\Gamma \vdash_{fnct} id(a_1 : \tau_1, \ldots, a_n : \tau_n)\{C\} : \tau \Rightarrow (\Gamma, id(a_1 : \tau_1 \times a_n : \tau_n) : \tau) \models \hat{\Gamma}}$$

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

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

5.5 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 $\tau : HT(C_\tau)$

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

5.6 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.

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

6 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. 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;
- **da mettere la roba del name/const/...**.

7 TAC

Blablablablablablablablablabla...

8 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 nel linguaggio.

La grammatica attributata 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 **blablablabla**

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

9 Conflitti

Non sono presenti conflitti nella grammatica definita da BNFC.