Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Corso di Laurea in Informatica

Relazione di Linguaggi

Candidati:
Davide Imola
Matricola VR386238

Andrea Slemer
Matricola VR386253

Anno Accademico 2016–2017

Indice

Capitolo 1

Testo elaborato

Estendere l'interprete OCAML studiato durante le lezioni del corso di Linguaggi scegliendo una delle tre semantiche studiate: operazionale, denotazionale o iterativa. La soluzione dovrá contenere il codice base imperativo e funzionale di blocchi, procedure e funzioni.

L'estensione del linguaggio didattico dovrá prevedere:

- Il tipo stringa di caratteri: string
- La costante string nella semantica come sequenza di caratteri alfanumerici
- Il calcolo della lunghezza di una stringa: len x
- La concatenazione di stringhe: s1@s2
- L'operazione di sottostringa: subs (string, index1, index2)

La possibilitá di introdurre ulteriori estenzioni/operazioni prendendo ispirazione da operazioni note in linguaggi di scripting é lasciata allo studente.

L'estensione prevede anche l'implementazione del comando 'Reflect string', il quale riceve in input come unico parametro una stringa e richiama l'interprete sulla stringa, la quale viene vista come sequenza di comandi eseguibili.

Capitolo 2

Struttura del programma

L'interprete utilizza una semantica denotazionale. Per rendere piú efficente l'invocazione dell'interprete in fase di testing, abbiamo utilizzato un file di supporto chiamato 'main.ml', il quale ha il compito di richiamare tutte le parti dell'interprete che risiedono in file diversi:

2.1 Linguaggio funzionale

Abbiamo adottato per una prima stesura del programma un linguaggio funzionale. La scelta ci ha permesso di costruire una prima versione dell'elaborato finale in modo agevole, in quanto il linguaggio funzionale permette una piÃź semplice e facile individuazione degli errori. Per un limite della struttura non siamo riusciti a portare a termine l'elaborato, poiché la funzione 'Reflect string' richiedeva che la stringa ricevuta in ingresso fosse interpretata come sequenza di comandi, il che é incompatibile con la struttura del linguaggio funzionale.

2.2 Linguaggio imperativo

Abbiamo continuato lo sviluppo del software usando un linguaggio imperativo e, riuscendo a trasporre il software funzionale prodotto, abbiamo avuto una base di partenza per l'elaborato finale.

```
#use "syntax.ml";;

#use "env/env_interface.ml";;
#use "env/env_semantics.ml";;
open Funenv;;

#use "stack/stack_interface.ml";;
#use "stack/stack_semantics.ml";;
open SemStack;;

#use "stack/stackm_interface.ml";;
#use "stack/stackm_semantics.ml";;
open SemStack_Modificable;;

#use "domains.ml";;
#use "operations.ml";;
#use "semantics.ml";;
```

Figura 2.1: Caso funzionale: 'main.ml'

```
#use "syntax.ml";;

#use "env/env_interface.ml";;
#use "env/env_semantics.ml";;
open Funenv;;

#use "stack/stack_interface.ml";;
#use "stack/stack_semantics.ml";;
open SemStack;;

#use "stack/stackm_interface.ml";;
#use "stack/stackm_semantics.ml";;
open SemStack_Modificable;;

#use "store/store_interface.ml";;
#use "store/store_semantics.ml";;
open Funstore;;

#use "domains.ml";;
#use "operations.ml";;
#use "semantics.ml";;
```

Figura 2.2: Caso imperativo: 'main.ml'

2.3 Struttura files del software

- **syntax.ml:** contiene i domini sintattici e definisce la sintassi di tutte le operazioni e i comandi definiti.
- **env_interface.ml:** contiene la signature dell'ambiente.
- **env_semantics.ml:** contiene l'implementazione dell'interfaccia dell'ambiente, definita in *env_ interface.ml*.
- **stack_interface.ml:** contiene la signature dello stack **non modificabile**.
- **stack_semantics.ml:** contiene l'implementazione dell'interfaccia dello stack **non modificabile**, definita in *stack_ interface.ml*.
- **stackm_interface.ml:** contiene la signature dello stack **modificabile**.
- **stackm_semantics.ml:** contiene l'implementazione dell'interfaccia dello stack **modificabile**, definita in *stackm_ interface.ml*.
- **store_interface.ml:** contiene la signature della memoria.
- **store_semantics.ml:** contiene l'implementazione dell'interfaccia della memoria, definita in *store_ interface.ml*.
- **domains.ml:** contiene i domini semantici (esprimibili *eval*, dichiarabili *dval*, memorizzabili *mval*) e le conversioni di tipo.
- **operations.ml:** contiene l'operazione *typecheck*, usato per verificare il tipo dei parametri in ingresso ad ogni operazione, e l'implementazione tutte le altre operazioni divise per tipologia (Operazioni *base*, Operazioni sulle *stringhe*, Operazioni di supporto private e non disponibili all'esecuzione, Operazione *parser* e *parserCom* di supporto al comando *Reflect*).
- **semantics.ml:** contiene la definizione della semantica di ogni parte sintattica necessaria all'interprete.

Capitolo 3

Modifiche al software di base

3.1 Tipo stringa: inserimento

Per soddisfare l'inserimento del tipo **stringa** é stato aggiunto sia nella sintassi (all'interno del tipo *exp*) e anche nella semantica (all'interno di tutti e tre i tipi presenti nel file *domains.ml*).

```
(* TYPE EXPRESSABLE *)
type exp =
      (* CONSTANTS *)
      | Eint of int
      | Ebool of bool
      | Estring of string
      | Den of ide
```

Figura 3.1: String nel tipo exp: 'syntax.ml'

```
(* TYPE EVAL: EXPRESSIBLE VALUES *)
type eval =
        | Int of int
        | Bool of bool
        | String of string
        | Funval of efun
        | Novalue
and efun = (dval list) * (mval store) -> eval
```

Figura 3.2: String nel tipo eval: 'domains.ml'

3.2 Tipo stringa: operazioni

Tutte le operazioni sulle espressioni sono state implementate nel file *operazioni.ml* il quale contiene:

- Una funzione che implementa il check di tipo, il quale prevede il controllo di tre tipologie di dato: *int*, *bool* e *string*.

```
(* SUPPORT FUNCTION - TYPECHECK *)
let typecheck (x, y) = match x with
      | "int" -> (match y with
          | Int(u) -> true
          | _ -> false )
      | "bool" -> (match y with
          | Bool(u) -> true
          | _ -> false )
      | "string" -> (match y with
          | String(u) -> true
          | _ -> false )
      | _ -> failwith ("not a valid type")
```

Figura 3.3: Funzione Typecheck: 'operations.ml'

- Una operazione **'len string'** che restiusce un intero che rappresenta il valore della lunghezza della stringa passata come parametro. Per lo sviluppo di questa funzione abbiamo fatto affidamento alla funzione **String.length** di OCaml.

```
(* COMPUTE THE LENGTH OF THE STRING X *)
let len x = if typecheck("string",x)
               then (match x with | String(x) -> Int(String.length x)
                                  | _ -> failwith ("len match error"))
               else failwith ("len type error")
```

Figura 3.4: Funzione Len: 'operations.ml'

- Una operazione **'conc string string'** che restituisce una stringa, la quale é la concatenazione delle due stringhe passate come parametri. Per lo sviluppo di questa funzione abbiamo fatto affidamento alla funzione *String.concat* di OCaml, questo ci ha permesso di giustapporre due stringhe inserendole in una lista, utilizzando come stringa di separazione la **stringa vuota ""**.

```
(* CONCATENATE STRING X TO STRING Y *)
let conc (x,y) = if typecheck("string",x) && typecheck("string",y)
                 then (match (x,y) with | (String(x), String(y)) -> String(String.concat "" [x; y])
                                        | _ -> failwith ("concat match error"))
                 else failwith ("concat type error")
```

Figura 3.5: Funzione Conc: 'operations.ml'

- Una operazione **'subs string int int'** che restituisce una stringa, la quale é composta dal pezzo di stringa passata come parametro che intercorre dal carattere in posizione del primo intero (compreso), fino al carattere in posizione del secondo intero (non compreso). Per lo sviluppo di questa funzione abbiamo fatto affidamento alla funzione **String.sub** di OCaml, questo ci ha permesso di evitare molti controlli sull'idoneitá dei tre parametri ricevuti in ingresso.

```
(* CUT A STRING X AND FROM INDEX "i1" (included) TO INDEX "i2" (not included) *)
let subs (x,i1,i2) = if typecheck("string",x) && typecheck("int",i1) && typecheck("int",i2)
                     then (match (x,i1,i2) with | (String(x), Int(i1), Int(i2)) -> String(String.sub x i1 ((i2-i1)+1))
                                                | _ -> failwith ("subs match error"))
                     else failwith ("subs type error")
```

Figura 3.6: Funzione Subs: 'operations.ml'

- Una operazione **'charat string int'** che restitusce un carattere, il quale é estratto dalla stringa passata come parametro e risiede nella posizione indicata dall'intero. Il carattere viene memorizzato comunque in formato *String*.

```
(* RETURN THE CHAR IN Y POSITION OF THE STRING X*)
let charat (x,y) = if typecheck("string",x) && typecheck("int",y)
                   then (match (x,y) with | (String(x), Int(y)) -> String(String.sub x y 1)
                                          | _ -> failwith ("charat match error"))
                   else failwith ("charat type error")
```

Figura 3.7: Funzione CharAt: 'operations.ml'

- Una operazione **'streq string string'** che restituisce un valore *'boolean'*, il quale é calcolato confrontando le due stringhe passate come parametro. Viene utilizzato come supporto la funzione *'String.compare'* la quale restiusce un valore intero (*negativo* se x $<$ y, *uguale a zero* se x $=$ y, o *positivo* se x $>$ y).

```
(* COMPARE IF THE TWO STRINGS ARE EQUALS *)
let streq (x,y) = if typecheck("string",x) && typecheck("string",y)
                  then (match (x,y) with | (String(x), String(y)) -> iszero(Int(String.compare (x) y))
                                         | _ -> failwith ("streq match error"))
                  else failwith ("streq type error")
```

Figura 3.8: Funzione StrEq: 'operations.ml'

3.3 Comando Reflect

Il comando **'Reflect string'** riceve in ingresso una stringa e la interpreta come una sequenza di *espressioni* e *comandi*, richiamando su di essa l'interprete stesso.

Reflect permette di effettuare uno scan della stringa da sinistra a destra, eseguendo ogni token riconosciuto come comando o espressione ed aspettandosi dopo ogni sotto-stringa riconosciuta i parametri definiti nella semantica corrispondente.

Il comando é stato implementato nella semantica dei comandi il quale controlla che la stringa sia formattata correttamente controllando che il numero di parentesi **'('** e **')'** sia di egual numero, inoltre controlla se il primo token della stringa rappresenta un comando o un'espressione.

Nel caso la funzione *'isCommand()'* dovesse riconoscere un comando *(restituendo un boolan con valore 'true')*, verrebbe invocata la funzione *parserCom()*, altrimenti verrebbe richiamata la funzione *parser*.

Per visualizzare il risultato viene allocata e dichiarata una variabile di output, chiamata *"result"*.

```
(* Reflect use function parser to valuate string e *)
| Reflect(e) -> let g = sem e r s in
  if typecheck("string",g) && eq_int(occurrence(g,String("(")),occurrence(g,String(")"))) && len(g)>=Int(5) && isCommand(g)
    then let st_stack = emptystack(100,Novalue) in
         let op_stack = emptystack(100,Undefinedstack) in
         let com = parserCom(g,op_stack,st_stack) in
         semc com r s
  else if typecheck("string",g) && eq_int(occurrence(g,String("(")),occurrence(g,String(")"))) && len(g)>=Int(5)
    then let st_stack = emptystack(100,Novalue) in
         let op_stack = emptystack(100,Undefinedstack) in
         let exp = parser(g,op_stack,st_stack) in
         if empty(st_stack) || eq_string(subs(top(st_stack),Int(0),Int(0)),String(")"))
             then semc (Assign(Den "result",exp)) r s
             else failwith ("parser error")
  else failwith ("string not valid")
```

Figura 3.9: Comando Reflect: 'semantics.ml'

3.4 Parser

La funzione *parser* ha bisogno di tre parametri:

```
(* Function parser for command Reflect for functions *)
let rec parser (e,op_stack,st_stack) =
```

- **e:** contiene la stringa che il parser deve analizzare.
- **op_stack:** contiene i token che vengono riconosciuti come espressioni e non sono ancora stati eseguiti.
- **st_stack:** contiene il resto della stringa che deve essere ancora analizzato.

La soluzione implementata scansiona la stringa in modo sequenziale.

La funzione riconosce i vari token grazie ad una funzione di *'substring'* che punta sempre a leggere la prima parola della stringa rimanente.

Caso base: la funzione incontra la fine della stringa o un tipo terminale *(Den, Eint, Ebool, Estring)*

Caso ricorsivo: la funzione incontra un'espressione che necessita di uno o piú parametri. In questo caso il token viene memorizzato e si procede a successive scansioni per la lettura dei parametri. Solo quando tutti i parametri sono stati letti correttamente l'espressione viene eseguita.

3.5 ParserCom

La funzione *parserCom* utilizza gli stessi tre paramteri della funzione *parser*, ma viene chiamata ogni volta che viene riconosciuto che il primo token della stringa **e** é un comando.

Capitolo 4

Esempi