Progettazione e Sviluppo di Interpreti

Dispensa per il corso di Paradigmi di Programmazione

22 ottobre 2021

Indice

1 Introduzione allo sviluppo di interpreti **2**
- 1.1 Componenti principali da realizzare in un interprete . . . 2
- 1.2 Ciclo di interpretazione . . . 2
- 1.3 Un interprete (risolutore) di espressioni artimetiche . . . 3
- 1.4 Lo Scanner (o Lexer, o tokenizzatore) . . . 4
- 1.5 Il Parser . . . 7
- 1.6 L'interprete . . . 9
- 1.7 Dalla semantica SOS al codice, sistematicamente . . . 12
- 1.8 Mettiamo le cose insieme . . . 12

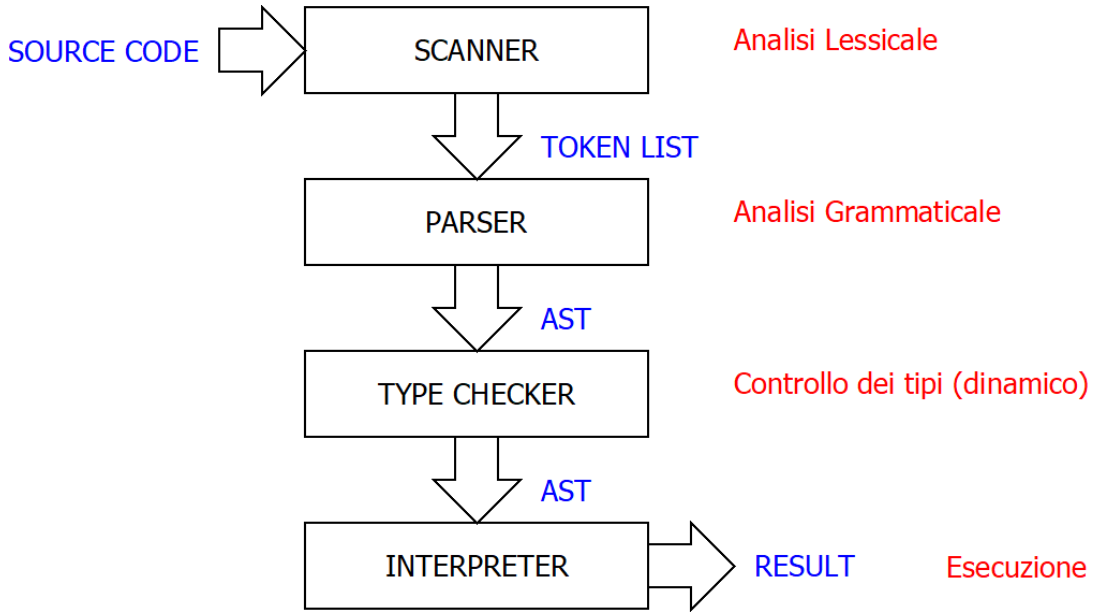
2 Un interprete del λ-calcolo **14**
- 2.1 La sintassi del λ-calcolo . . . 14
- 2.2 La semantica del λ-calcolo . . . 14
- 2.3 Implementazione della capture-avoiding substitution . . . 14
- 2.4 Implementazione di FV(e) . . . 15
- 2.5 Generatore di identificatori "freschi" . . . 15
- 2.6 Implementazione della semantica . . . 17
- 2.7 PER ESERCIZIO... . . . 19
- 2.8 Funzioni di utilità per la stampa delle λ-espressioni . . . 19

3 Un interprete di MiniCaml **21**
- 3.1 Definizioni di tipi di dato usati nell'interpete . . . 21
- 3.2 Ambiente e valori esprimibili . . . 22
- 3.3 Type Checking . . . 23
- 3.4 Eccezione in caso di errori durante l'esecuzione . . . 24
- 3.5 Operazioni primitive . . . 24
- 3.6 Interprete . . . 26
- 3.7 Esempio di esecuzione: fattoriale . . . 27

Capitolo 1

Introduzione allo sviluppo di interpreti

1.1 Componenti principali da realizzare in un interprete



1.2 Ciclo di interpretazione

L'interprete esegue le operazioni elementari del programma *una dopo l'altra*

Le fasi di *scanning/parsing* possono essere svolte a livello di:

- *intero programma*
- *singole istruzioni/espressioni*

Lo stesso vale per i *controlli di tipo* (e altre analisi)

- *analisi statica* (intero programma)
- *controlli dinamici* (per ogni istruzione/espressione, all'esecuzione)
- *analisi statica + controlli dinamici*

Ad esempio: L'interprete *JavaScript* (Node.js) prima esegue analizza la sintassi di *tutto il programma* e poi interpreta

```
console.log(1);
console.log(2;  //manca la parentesi
```

Risultato: non stampa `1`, perchè prima di eseguire controlla la sintassi di tutto il programma:

```
console.log(2;
            ^
SyntaxError: missing ) after argument list
```

Altro esempio: Il *toplevel di Ocaml* esegue il parsing ed esegue espressione per espressione

[1]:
```
let x = 10 ;;
lett y = 20 ;; (* let scritto male... *)
```

[1]:
```
val x : int = 10
```

```
File "[1]", line 2, characters 0-4:
2 | lett y = 20 ;; (* let scritto male... *)
    ^^^^
Error: Unbound value lett
```

Prima ha parsato ed eseguito `let x = 10`, poi ha parsato `lett y = 20` trovando l'errore

1.3 Un interprete (risolutore) di espressioni artimetiche

Sintassi delle espressioni aritmetiche (solo su interi, *non serve type checking*)

```
Exp ::= n | Exp op Exp | (Exp)
op ::= + | - | * | /
```

Definiamo un tipo algebrico per rappresentare alberi di sintassi astratta (AST) per le espressioni definite da questa grammatica:

[2]:
```
type op = Add | Sub | Mul | Div ;;
type exp =
    | Val of int
    | Op of op*exp*exp;;
```

[2]:
```
type op = Add | Sub | Mul | Div
```

[2]:
```
type exp = Val of int | Op of op * exp * exp
```

Un paio di esempi di espressioni:

```
exp1 = (3*7)-5
exp2 = 3*(7-5)
```

[3]:
```
let exp1 = Op (Sub, (Op (Mul, Val 3, Val 7)), Val 5) ;;
let exp2 = Op (Mul, Val 3, (Op (Sub, Val 7, Val 5))) ;;
```

[3]:
```
val exp1 : exp = Op (Sub, Op (Mul, Val 3, Val 7), Val 5)
```

[3]:
```
val exp2 : exp = Op (Mul, Val 3, Op (Sub, Val 7, Val 5))
```

Scriviamo innanzitutto una funzione di utilità che (ri)trasforma l'AST in formato testuale:

[4]:
```
let rec to_string e =
    let symbol o =
        match o with | Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
    in
    match e with
    | Val n -> string_of_int n
    | Op (o,e1,e2) -> "("^ (to_string e1) ^ (symbol o) ^ (to_string e2) ^")" ;;
```

[4]:
```
val to_string : exp -> string = <fun>
```

[5]:
```
to_string exp1 ;;
to_string exp2 ;;
```

[5]:
```
- : string = "((3*7)-5)"
```

[5]:
```
- : string = "(3*(7-5))"
```

1.4 Lo Scanner (o Lexer, o tokenizzatore)

Rappresentiamo ogni simbolo che può apparire in una espressione con un *token*

[6]:
```
type token =
    | Tkn_NUM of int      (* numero n *)
    | Tkn_OP of string    (* operatore + - * / *)
    | Tkn_LPAR            (* simbolo ( *)
    | Tkn_RPAR            (* simbolo ) *)
    | Tkn_END;;           (* fine dell'espressione *)
```

[6]:
```
type token =
    Tkn_NUM of int
  | Tkn_OP of string
  | Tkn_LPAR
  | Tkn_RPAR
  | Tkn_END
```

Lo *scanner* trasforma:

- la *rappresentazione testuale* dell'espressione (stringa)
- in una *lista di token*

Nell'eseguire questa trasformazione lo scanner *controlla* che nell'espressione non siano stati utilizzati *simboli non previsti*

Lo scanner non effettua (ancora) un controllo grammaticale:

- `)(3++(88` è corretta per lo scanner
- `[(3+2)-1]` non è corretta per lo scanner (`[` e `]` sono simboli non previsti)

Implementazione dello scanner (funzione `tokenize`)

Scandisce ricorsivamente la stringa un carattere per volta e produce il token corrispondente

- Se trova più simboli numerici (`0-9`) uno dopo l'altro li *aggrega* in un unico token numerico (es. `Tkn_NUM 231`)
- Solleva l'eccezione `ParseError` se incontra un simbolo non previsto

[7]:
```
exception ParseError of string*string;;
```

[7]:
```
exception ParseError of string * string
```

[8]:
```
(* scanner *)
let tokenize s =
    (* funzione che scandisce ricorsivamente s,
       dove pos è la posizione del carattere corrente *)
    let rec tokenize_rec s pos =
        if pos=String.length s then [Tkn_END] (* caso base: fine lista *)
        else
            let c = String.sub s pos 1 (* estrae il carattere corrente *)
            in
            (* si richiama ricorsivamente prima di gestire il carattere corrente *)
            let tokens = tokenize_rec s (pos+1)
            in
                (* trasforma il carattere corrente in un token*)
                match c with
                | " " -> tokens
                | "(" -> Tkn_LPAR::tokens
                | ")" -> Tkn_RPAR::tokens
                | "+" | "-" | "*" | "/" -> (Tkn_OP c)::tokens
                | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ->
                    (* accorpa cifre consecutive *)
                    (match tokens with
                    | Tkn_NUM n::tokens' ->
                            Tkn_NUM (int_of_string (c^(string_of_int n)))::tokens'
                    | _ -> Tkn_NUM (int_of_string c)::tokens
                    )
                | _ -> raise (ParseError ("Tokenizer","unknown symbol: "^c))
    in
        tokenize_rec s 0 ;;
```

[8]:
```
val tokenize : string -> token list = <fun>
```

Esempio d'uso dello scanner:

[9]:
```
let t1 = tokenize "(34 + 41) - (2223 * 2)";;
```

[9]:
```
val t1 : token list =
  [Tkn_LPAR; Tkn_NUM 34; Tkn_OP "+"; Tkn_NUM 41; Tkn_RPAR; Tkn_OP "-";
   Tkn_LPAR; Tkn_NUM 2223; Tkn_OP "*"; Tkn_NUM 2; Tkn_RPAR; Tkn_END]
```

[10]:
```
let t2 = tokenize "(34 + 41) - ([2223 * 2)";;
```

```
Exception: ParseError ("Tokenizer", "unknown symbol: [").
Raised at file "[8]", line 26, characters 29-76
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from file "[8]", line 11, characters 25-47
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Funzioni di utilità per stampare un token o una lista di token:

[11]:
```
let string_of_token t =
    match t with
    | Tkn_NUM n -> "Tkn_NUM "^(string_of_int n)
    | Tkn_OP s -> "Tkn_OP "^s
    | Tkn_LPAR -> "Tkn_LPAR"
    | Tkn_RPAR -> "Tkn_RPAR"
    | Tkn_END -> "Tkn_END" ;;

let print_token t =
    print_endline (string_of_token t);;

let print_tokenlist tl =
    List.iter (fun t -> (print_token t)) tl;;
```

[11]:
```
val string_of_token : token -> string = <fun>
```

[11]:
```
val print_token : token -> unit = <fun>
```

[11]:
```
val print_tokenlist : token list -> unit = <fun>
```

[12]:
```
print_tokenlist t1;;  (* ( 34 + 41 ) - ( 2223 * 2 ) *)
```

```
Tkn_LPAR
Tkn_NUM 34
Tkn_OP +
Tkn_NUM 41
Tkn_RPAR
Tkn_OP -
Tkn_LPAR
```

```
Tkn_NUM 2223
Tkn_OP *
Tkn_NUM 2
Tkn_RPAR
Tkn_END
```

```
[12]: - : unit = ()
```

"Problemi" di questa implementazione dello scanner

La funzione `tokenize_rec` è ricorsiva, ma *non tail-recursive*...

- potenziali problemi con espressioni estremamente lunghe

ESERCIZIO: Pensare ad *implementazioni alternative*:

- tail-recursive (ci vuole un accumulatore...)
- iterative (usando costrutti imperativi di OCaml)
- usando la funzione `String.iter` per trasformare ogni carattere in un token e la funzione `List.fold_left` per accorpare i token numerici consecutivi (o qualcosa di simile). Richiede due passate...

1.5 Il Parser

Il parser

- *controlla* che l'espressione sia *sintatticamente corretta*
  - appartenga al linguaggio definito dalle regole BNF
- *genera* l'abstract syntax tree (*AST*)

Come si realizza un parser:

- la grammatica deve essere resa *non ambigua*
- se il linguaggio è *molto semplice*, si implementa un *parser a discesa ricorsiva*
- se il linguaggio *non è molto semplice*, si usa un *parser generator* (software che genera il codice del parser a partire dalla grammatica)
  - Flex/Bison per generare un parser in C
  - PEG.js, ATNLR4 o Jison per generare un parser in JavaScript
  - ocamllex/ocamlyacc o Menhir per generare un parser in OCaml
  - ATNLR4 o JavaCC per generare un parser in Java
  - ...

Ridefiniamo la grammatica delle espressioni in modo *non ambiguo*

- Da così:

```
Exp ::= n | Exp op Exp | (Exp)
op ::= + | - | * | /
```

- A così:

```
Exp ::= Term + Exp | Term - Exp | Term
Term ::= Factor * Term | Factor / Term | Factor
Factor ::= n | (Exp)
```

ossia:

```
Exp ::= Term [ + Exp | - Exp ]
Term ::= Factor [ * Term | / Term ]
Factor ::= n | (Exp)
```

dove `[...]` indica che il contenuto è opzionale

Implementazione del parser (funzione `parse`)

Implementiamo un parser a discesa ricorsiva:

- una *funzione* per ogni *categoria sintattica* (`exp`, `term` e `factor`)
- le funzioni sono *mutuamente ricorsive*
- le funzioni si richiamano l'una con l'altra e *consumano token* secondo quanto indicato dalla grammatica
- ogni chiamata di funzione restituisce un nodo dell'AST
  - l'*AST* viene in questo modo corrisponde all'*albero delle chiamate*

[13]:
```
(* parser *)
let parse s =

   (* usiamo un riferimento per scandire la lista dei token (ottenuta da tokenize)␣
 ↪*)
    let tokens = ref (tokenize s) in

    (* restituisce il primo token senza rimuoverlo *)
    let lookahead () = match !tokens with
        | [] -> raise (ParseError ("Parser","lookahead error"))
        | t::_ -> t
    in

    (* elimina il primo token *)
    let consume () = match !tokens with
        | [] -> raise (ParseError ("Parser","consume error"))
        | t::tkns -> tokens := tkns
    in

    (* funzioni mutuamente ricorsive che seguono dalla grammatica *)
    (* Exp ::= Term [ + Exp | - Exp ] *)
    let rec exp () =
        let t1 = term() in
        match lookahead () with
        | Tkn_OP "+" -> consume(); Op (Add,t1,exp())
        | Tkn_OP "-" -> consume(); Op (Sub,t1,exp())
        | _ -> t1

    (* Term ::= Factor [ + Term | - Term ] *)
    and term () =
        let f1 = factor() in
        match lookahead() with
        | Tkn_OP "*" -> consume(); Op (Mul,f1,term())
        | Tkn_OP "/" -> consume(); Op (Div,f1,term())
        | _ -> f1

    (* Factor ::= n | ( Exp ) *)
    and factor () =
        match lookahead() with
        | Tkn_NUM n -> consume(); Val n
        | Tkn_LPAR -> consume(); let e = exp() in
                      (match lookahead() with
                      | Tkn_RPAR -> consume(); e
                      | _ -> raise (ParseError ("Parser","RPAR error"))
                      )
        | _ -> raise (ParseError ("Parser","NUM/LPAR error"))
```

```
   (* Si comincia chiamando exp che fa tutto il lavoro e restituisce la radice␣
↪dell'AST *)
   in
       let ast = exp() in
       (* controlliamo che al termine sia rimasto solo Tkn_END *)
       match lookahead() with
       | Tkn_END -> ast
       | x -> print_tokenlist !tokens; raise (ParseError ("Parser","parse error"));
↪;
```

[13]:
```
val parse : string -> exp = <fun>
```

Esempi di esecuzione del parser:

[14]:
```
let ast = parse "32 + 24 * 12 * (3-1) +2" ;;
```

[14]:
```
val ast : exp =
  Op (Add, Val 32,
   Op (Add, Op (Mul, Val 24, Op (Mul, Val 12, Op (Sub, Val 3, Val 1))),
    Val 2))
```

[15]:
```
let err = parse "16 + (7 - 2" ;;
```

```
Exception: ParseError ("Parser", "RPAR error").
Raised at file "[13]", line 43, characters 35-71
Called from file "[13]", line 30, characters 17-25
Called from file "[13]", line 22, characters 17-23
Called from file "[13]", line 24, characters 46-51
Called from file "[13]", line 49, characters 18-23
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Discussione sui parser

Implementare un parser non è sempre così semplice:

- rendere la grammatica non ambigua non è sempre ovvio
- con la discesa ricorsiva è facile imbattersi in situazioni di *ricorsione infinita*
  - Es: se avessimo definito `Exp ::= Exp + Term | ...` la funzione `exp` per prima cosa si sarebbe chiamata ricorsivamente...
- a volte non è sufficiente leggere un singolo token per capire quale regola grammaticale applicare
  - *lookhaed dinamico*
- ...

proprio per tutto questo: * esistono i *parser generator* (il cui funzionamento non è argomento di questo corso) * per i linguggi che vedremo *NON* implementeremo un parser * *partiremo dall'AST*

1.6 L'interprete

E ora implementiamo l'interprete delle espressioni.

Si parte dalla *definizione della semantica*.

L'abbiamo già definita almeno un paio di volte, in modi diversi:

- facciamo un po' di ordine...

Structural Operational Semantics (SOS)

La definizione di una relazione di transizione per descrivere il comportamento dei programmi (o espressioni) scritti in un certo linguaggio segue solitamente l'approccio della *Structural Operational Semantics (SOS)*, o Semantica Strutturale Operazionale.

- *Semantics*: è una descrizione del "significato" del linguaggio. Nei linguaggi di programmazione il significato è dato dal comportamento dei programmi scritti in quel linguaggio
- *Operational*: descrive il comportamento dei programmi tramite una relazione di transizione ($\rightarrow$) che cattura le operazioni che vengono svolte passo-passo durante l'esecuzione
- *Structural*: la relazione di transizione è definita usando regole di inferenza basate sulla struttura sintattica (una o più regole per ogni costurutto definito dalla grammatica del linguaggio)

Approcci alla definizione: small-step e big-step

Esistono due approcci principali alla definizione di una semantica in stile SOS

Small-step semantics:

- ogni passo della relazione di transizione si esegue una singola operazione
- una computazione è una sequenza di passi
- esempio: $((3 + (5 * 2)) - 1) \rightarrow_{ss} ((3 + 10) - 1) \rightarrow_{ss} (13 - 1) \rightarrow_{ss} 12$

LA COMPUTAZIONE SI SVILUPPA LUNGO LA SEQUENZA DI PASSI

Big-step semantics:

- la relazione di transizione *descrive in un solo passo l'intera computazione*
- le singole operazioni sono descritte nell'albero di derivazione di quella transizione
- esempio:

$$\frac{\dfrac{\vdots}{3 \rightarrow_{bs} 3} \quad \dfrac{\vdots}{(5*2) \rightarrow_{bs} 10} \quad 3+10=13}{(3+(5*2)) \rightarrow_{bs} 13} \quad 1 \rightarrow_{bs} 1 \quad 13 - 1 = 12}{((3 + (5 * 2)) - 1) \rightarrow_{bs} 12}$$

LA COMPUTAZIONE SI SVILUPPA DISCENDENDO L'ALBERO DI DERIVAZIONE

Semantica delle espressioni

Semantica *small-step*:

$$n \rightarrow_{ss} n \qquad \frac{E_1 \rightarrow_{ss} E_1'}{E_1 \, op \, E_2 \rightarrow_{ss} E_1' \, op \, E_2}$$

$$\frac{E_2 \rightarrow_{ss} E_2'}{n \, op \, E_2 \rightarrow_{ss} n \, op \, E_2'} \qquad \frac{n_1 \text{op } n_2 = n}{n_1 \, op \, n_2 \rightarrow_{ss} n}$$

Semantica *big-step*:

$$n \rightarrow_{bs} n \qquad \frac{E_1 \rightarrow_{bs} n_1 \quad E_2 \rightarrow_{bs} n_2 \quad n_1 \text{op } n_2 = n}{E_1 \, op \, E_1 \rightarrow_{bs} n}$$

Implementazione dell'inteprete (funzione `eval`)

Consideriamo inizialmente la semantica *big-step* (più semplice).

$$n \rightarrow_{bs} n \qquad \frac{E_1 \rightarrow_{bs} n_1 \;\; E_2 \rightarrow_{bs} n_2 \;\; n_1 \,\mathrm{op}\, n_2 = n}{E_1 \, op \, E_1 \rightarrow_{bs} n}$$

[16]:
```
(* interprete big-step *)
let rec eval e =
    match e with
    | Val n -> Val n
    | Op (op,e1,e2) ->
        (* chiamate ricorsive che calcolano le derivazioni per e1 ed e2 *)
        match (eval e1, eval e2) with
        | (Val n1, Val n2) -> (match op with   (* calcola n1 op n2 *)
                               | Add -> Val (n1+n2)
                               | Sub -> Val (n1-n2)
                               | Mul -> Val (n1*n2)
                               | Div -> Val (n1/n2)
                              )
        (* caso (inutile) aggiunto solo per rendere esaustivo il pattern matching␣
 ↪*)
        | _ -> failwith "Errore impossibile che si verifichi" ;;
```

[16]:
```
val eval : exp -> exp = <fun>
```

NOTA: l'interprete non restituisce un intero (es. `n`), ma un *valore* nella rappresentazione AST (es. `Val n`)

- per coerenza con $\rightarrow_{bs}$ che è definita su $Exp \times Val$ con $Val \subset Exp$.

Esempio d'uso dell'interprete:

[17]:
```
(* AST dell'espressione 3+(5*2) *)
let exp = Op (Add , Val 3, Op ( Mul , Val 5, Val 2)) ;;

eval exp;;
```

[17]:
```
val exp : exp = Op (Add, Val 3, Op (Mul, Val 5, Val 2))
```

[17]:
```
- : exp = Val 13
```

Implementazione dell'inteprete (funzione `eval_ss`)

Consideriamo ora la semantica *small-step*

$$n \rightarrow_{ss} n \qquad \frac{E_1 \rightarrow_{ss} E_1'}{E_1 \, op \, E_2 \rightarrow_{ss} E_1' \, op \, E_2} \qquad \frac{E_2 \rightarrow_{ss} E_2'}{n \, op \, E_2 \rightarrow_{ss} n \, op \, E_2'} \qquad \frac{n_1 \,\mathrm{op}\, n_2 = n}{n_1 \, op \, n_2 \rightarrow_{ss} n}$$

[18]:
```
(* interprete small-step *)
let rec eval_ss e =
        match e with
        | Val n -> Val n
        | Op (op,e1,e2) ->
            (match (e1,e2) with
```

```
            (* se gli operandi sono entrambi valori, calcola n1 op n2 *)
            | (Val n1,Val n2) -> Val (match op with
                                      | Add -> n1 + n2     | Sub -> n1 - n2
                                      | Mul -> n1 * n2     | Div -> n1 / n2 )
            (* se e1 può fare un passo (cioè se è un Op e non un valore) *)
            | (Op (_,_,_),_) -> Op (op,(eval_ss e1),e2)
            (* altrimenti, se e1 è un valore ma e2 può fare un passo *)
            | (Val _,Op (_,_,_))  -> Op (op,e1,(eval_ss e2))
            ) ;;
```

[18]:
```
val eval_ss : exp -> exp = <fun>
```

Esempio di esecuzione del nuovo interprete (small-step):

[19]:
```
(* AST dell'espressione 3+(5*2) *)
let exp = Op (Add , Val 3, Op ( Mul , Val 5, Val 2)) ;;

eval_ss exp;;             (* fa un passo *)
eval_ss (eval_ss exp);;   (* fa due passi *)
```

[19]:
```
val exp : exp = Op (Add, Val 3, Op (Mul, Val 5, Val 2))
```

[19]:
```
- : exp = Op (Add, Val 3, Val 10)
```

[19]:
```
- : exp = Val 13
```

Il risultato è lo stesso calcolato, in un solo passo, dall'interprete precedente (big-step):

[43]:
```
eval exp ;;
```

[43]:
```
- : exp = Val 13
```

1.7 Dalla semantica SOS al codice, sistematicamente

Questi due esempi mostrano un *approccio sistematico di implementazione*:

1. Si usa il *pattern matching* per considerare i vari tipi di *nodo dell'AST*
2. Ogni tipo di *nodo* corrisponde a un *costrutto sintattico* definito dalla grammatica
3. Per ogni caso del pattern matching (costrutto sintattico) si *identificano le regole della semantica* relative ad esso (la semantica è syntax-driven)
4. Si verificano le *precondizioni* delle varie regole, possibilmente *richiamando ricorsivamente l'interprete* (per le regole che non sono assiomi)
5. Quando si trova una regola le cui precondizioni sono verificate, si calcola *risultato della transizione*

1.8 Mettiamo le cose insieme

Scanner + Parser + Interprete

[35]:
```
(* con inteprete big-step *)
let exec1 s = eval (parse s) ;;
```

```
(* con inteprete small-step *)
let exec2 s =
    let rec eval_rec ast = (* chiusura transitiva *)
        match ast with
        | Val n -> Val n
        | _ -> eval_rec ( eval_ss ast )
    in
        eval_rec (parse s) ;;
```

[35]:
```
val exec1 : string -> exp = <fun>
```

[35]:
```
val exec2 : string -> exp = <fun>
```

[33]:
```
exec1 "3+2*(6-2)" ;;
exec2 "3+2*(6-2)" ;;
```

[33]:
```
- : exp = Val 11
```

[33]:
```
- : exp = Val 11
```

Piccola variante: interprete small-step che mostra tutti i passi:

[41]:
```
let solve s =
    let rec solve_rec ast =
        match ast with
        | Val n -> (to_string ast)
        | _ -> (to_string ast)^" = "^(solve_rec (eval_ss ast))
    in
        solve_rec (parse s);;

solve "3+2*(6-2)" ;;
```

[41]:
```
val solve : string -> string = <fun>
```

[41]:
```
- : string = "(3+(2*(6-2))) = (3+(2*4)) = (3+8) = 11"
```

Capitolo 2

Un interprete del λ-calcolo

2.1 La sintassi del λ-calcolo

$$e ::= x \mid \lambda x.e \mid e\, e$$

Definiamo il tipo degli identificatori e dell'AST delle espressioni:

[1]:
```
type id = string

type exp =
    Var of id
  | Lam of id * exp
  | App of exp * exp
```

[1]:
```
type id = string
```

[1]:
```
type exp = Var of id | Lam of id * exp | App of exp * exp
```

2.2 La semantica del λ-calcolo

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'}$$

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

2.3 Implementazione della capture-avoiding substitution

Richiamo della definizione:

$$
\begin{array}{rcl}
x\{x := e\} & \equiv & e \\
y\{x := e\} & \equiv & y \text{ se } y \neq x \\
(e_1 e_2)\{x := e\} & \equiv & (e_1\{x := e\})(e_2\{x := e\}) \\
(\lambda y.e_1)\{x := e\} & \equiv & \lambda y.(e_1\{x := e\}) \\
 & & \text{se } y \neq x \text{ e } y \notin FV(e) \\
(\lambda y.e_1)\{x := e\} & \equiv & \lambda z.((e_1\{y := z\})\{x := e\}) \\
 & & \text{se } y \neq x \text{ e } y \in FV(e) \text{ e } z \text{ fresca}
\end{array}
$$

2.4 Implementazione di FV(e)

$$FV(x) = x \qquad FV(\lambda x.e) = FV(e) \setminus \{x\}$$

$$FV(e_1\, e_2) = FV(e_1) \cup FV(e_2)$$

Funzione che calcola ricorsivamente l'insieme delle *variabili libere* dell'espressione e rappresentato come lista di identificatori:

[2]:
```
(* computes the free (non-bound) variables in e *)
let rec fvs e =
  match e with
      Var x -> [x]
    | Lam (x,e) -> List.filter (fun y -> x <> y) (fvs e)
    | App (e1,e2) -> (fvs e1) @ (fvs e2)
;;
```

[2]:
```
val fvs : exp -> id list = <fun>
```

Esempi d'uso:

[3]:
```
(* TESTS *)
fvs (Var "x") = ["x"];;
fvs (Lam ("x",Var "y")) = ["y"];;
fvs (Lam ("x",Var "x")) = [];;
fvs (App (Lam ("x", Var "z"), Var "y")) = ["z"; "y"];;
```

[3]:
```
- : bool = true
```

[3]:
```
- : bool = true
```

[3]:
```
- : bool = true
```

[3]:
```
- : bool = true
```

2.5 Generatore di identificatori "freschi"

La capture avoiding substitution richiede in alcuni casi di generare identificatori *freschi* (nuovi, non presenti nell'espressione)

[4]:
```
(* generates a fresh variable   *)
let newvar =
```

```
  let x = ref 0 in
  fun () ->
    let c = !x in
    incr x;
    "v"^(string_of_int c)
```

[4]:
```
val newvar : unit -> string = <fun>
```

`newvar` è una funzione senza parametri con una *variabile modificabile* `x` nella sua *chiusura*!

Esempi d'uso:

[5]:
```
(* TESTS *)
newvar ();;
newvar ();;
```

[5]:
```
- : string = "v0"
```

[5]:
```
- : string = "v1"
```

Ora siamo pronti per implementare la capture-avoiding substitution:

[6]:
```
(* substitution: subst e y m means
  "substitute occurrences of variable y with m in the expression e" *)
let rec subst e y m =
  match e with
      Var x ->
    if y = x then m (* replace x with m *)
    else e (* variables don't match: leave x unchanged *)
    | App (e1,e2) -> App (subst e1 y m, subst e2 y m)
    | Lam (x,e) ->
      if y = x then (* don't substitute under the variable binder *)
        Lam(x,e)
      else if not (List.mem x (fvs m)) then (* no need to alpha convert *)
        Lam (x, subst e y m)
      else (* need to alpha convert *)
        let z = newvar() in (* assumed to be "fresh" *)
        let e' = subst e x (Var z) in (* replace x with z in e *)
        Lam (z,subst e' y m) (* substitute for y in the adjusted term, e' *)
```

[6]:
```
val subst : exp -> id -> exp -> exp = <fun>
```

[7]:
```
(* TESTS *)
let m1 =  (App (Var "x", Var "y"));;  (* x y *)
let m2 = (App (Lam ("z",Var "z"), Var "w"));; (* (lambda z . z) w *)
let m3 = (App (Lam ("z",Var "x"), Var "w"));; (* (lambda z . x) w *)
let m4 = (App (App (Lam ("z",Var "z"), Lam ("x", Var "x")), Var "w"))
            (* (lambda z . z) (lambda x . x) w *)

let m1_zforx = subst m1 "x" (Var "z");;
let m1_m2fory = subst m1 "y" m2
let m2_ughforz = subst m2 "z" (Var "ugh")
let m3_zforx = subst m3 "x" (Var "z")
let m1_m3fory = subst m1 "y" m3
```

```
[7]: val m1 : exp = App (Var "x", Var "y")

[7]: val m2 : exp = App (Lam ("z", Var "z"), Var "w")

[7]: val m3 : exp = App (Lam ("z", Var "x"), Var "w")

[7]: val m4 : exp = App (App (Lam ("z", Var "z"), Lam ("x", Var "x")), Var "w")

[7]: val m1_zforx : exp = App (Var "z", Var "y")

[7]: val m1_m2fory : exp = App (Var "x", App (Lam ("z", Var "z"), Var "w"))

[7]: val m2_ughforz : exp = App (Lam ("z", Var "z"), Var "w")

[7]: val m3_zforx : exp = App (Lam ("v2", Var "z"), Var "w")

[7]: val m1_m3fory : exp = App (Var "x", App (Lam ("z", Var "x"), Var "w"))
```

2.6 Implementazione della semantica

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'} \qquad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

```
[8]: (* beta reduction. *)
let rec reduce e =
  match e with
      App (Lam (x,e1), e2) -> subst e1 x e2 (* direct beta rule *)
    | App (e1,e2) ->
      let e1' = reduce e1 in (* try to reduce a term in the lhs *)
      if e1'<>e1 then App(e1',e2)
      else App (e1,reduce e2) (* didn't work; try rhs *)
    | Lam (x,e) -> Lam (x, reduce e) (* reduce under the lambda (!) *)
    | _ -> e (* no opportunity to reduce *)

[8]: val reduce : exp -> exp = <fun>

[9]: (* TESTS *)
let m2red = reduce m2 ;; (* (lambda z . z) w *)
let m3red = reduce m3 ;; (* (lambda z . x) w *)
let m4red1 = reduce m4 ;; (* (lambda z . z) (lambda x . x) w *)
let m4red2 = reduce m4red1 ;; (* vedi sopra *)
let m13sred = reduce m1_m3fory ;; (* x ((lambda z . x) w) *)
```

```
[9]: val m2red : exp = Var "w"

[9]: val m3red : exp = Var "x"

[9]: val m4red1 : exp = App (Lam ("x", Var "x"), Var "w")

[9]: val m4red2 : exp = Var "w"

[9]: val m13sred : exp = App (Var "x", Var "x")
```

DOMANDA: Ma la semantica che abbiamo implementato è effettivamente questa?

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'}$$

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

Non esattamente...

- L'*ordine dei pattern* nel pattern matching *privilegia l'applicazione funzionale* alla riduzione di e_1 e di e_2

```
match e with
      App (Lam (x,e1), e2) -> subst e1 x e2 (* direct beta rule *)
    | App (e1,e2) -> ...
```

- Inoltre, questo modo di gestire l'applicazione funzionale dice che `e2` può essere ridotto *solo se* `e1` *non è riducibile*:

```
App (e1,e2) ->
      let e1' = reduce e1 in (* try to reduce a term in the lhs *)
      if e1'<>e1 then App(e1',e2)
      else App (e1,reduce e2) (* didn't work; try rhs *)
```

Quindi la semantica che abbiamo implementato in realtà è questa:

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \neq_\alpha \lambda x.e_3 \quad e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_1 \neq_\alpha \lambda x.e_3 \quad e_1 \not\to \quad e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'}$$

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

dove:

- $=_\alpha$ è l'α-equivalenza
- $e_1 \not\to$ significa $\nexists e_1'.e_1 \to e_1'$.

D'altra parte la definizione originale è *non deterministica*

- la stessa espressione può fare due riduzioni diverse

Esempio:

$$((\lambda x.x)\, y)\, ((\lambda x.x)\, y) \to y\, ((\lambda x.x)\, y)$$
$$((\lambda x.x)\, y)\, ((\lambda x.x)\, y) \to ((\lambda x.x)\, y)\, y$$

Invece *l'interprete è un programma* e la sua esecuzione deve essere *deterministica*

- si può dimostrare, sfruttando la proprietà di confluenza del λ-calcolo (i.e., Church-Rosser) che nonostante le condizioni aggiunte non si perdono computazioni possibili

2.7 PER ESERCIZIO...

- Implementare la *chiusura transitiva* della semantica, che consente di eseguire intere computazioni invece che singoli passi. Che succede se le diamo in pasto l'espressione $\Omega = (\lambda x.xx)(\lambda x.xx)$?
- Implementare la semantica *call-by-value* del λ-calcolo
- Implementare la seguente *semantica big-step* del λ-calcolo e verificare che succedere dandole in pasto l'espressione Ω:

$$\frac{e \to_{bs} e'}{\lambda x.e \to_{bs} \lambda x.e'} \qquad x \to_{bs} x$$

$$\frac{e_1 \to_{bs} \lambda x.e_1' \quad e_2 \to_{bs} e_2' \quad e_1'\{x := e_2'\} \to_{bs} e'}{e_1\, e_2 \to_{bs} e'}$$

2.8 Funzioni di utilità per la stampa delle λ-espressioni

[10]:
```
(* pretty printing *)
open Format;;

let ident = print_string;;
let kwd = print_string;;

let rec print_exp0 = function
  | Var s ->  ident s
  | lam -> open_hovbox 1; kwd "("; print_lambda lam; kwd ")"; close_box ()

and print_app = function
  | e -> open_hovbox 2; print_other_applications e; close_box ()

and print_other_applications f =
  match f with
  | App (f, arg) -> print_app f; print_space (); print_exp0 arg
  | f -> print_exp0 f

and print_lambda = function
  | Lam (s, lam) ->
      open_hovbox 1;
      kwd "\\"; ident s; kwd "."; print_space(); print_lambda lam;
      close_box()
  | e -> print_app e;;
```

[10]:
```
val ident : string -> unit = <fun>
```

[10]:
```
val kwd : string -> unit = <fun>
```

[10]:
```
val print_exp0 : exp -> unit = <fun>
val print_app : exp -> unit = <fun>
val print_other_applications : exp -> unit = <fun>
val print_lambda : exp -> unit = <fun>
```

[11]:
```
(* TESTS *)
print_lambda m1; print_newline ();;
print_lambda m2; print_newline ();;
```

```
x y
```

[11]:
```
- : unit = ()
```

```
(\z. z) w
```

[11]:
```
- : unit = ()
```

Capitolo 3

Un interprete di MiniCaml

La spiegazione sul linguaggio MiniCaml e sullo sviluppo di questo interprete è nel materiale didattico del corso (slides). In queste note saranno sottolineati solo alcuni aspetti implementativi.

3.1 Definizioni di tipi di dato usati nell'interpete

Tipi per la sintassi astratta del linguaggio:

[1]:
```
(* Identificatori *)
type ide = string;;

(* I tipi *)
type tname =  TInt | TBool | TString | TClosure | TRecClosure | TUnBound

(* Abstract Expressions = espressioni nella sintassi astratta,
   compongono l'Albero di Sintassi Astratta *)
type exp =
    | EInt of int
    | CstTrue
    | CstFalse
    | EString of string
    | Den of ide
    (* Operatori binari da interi a interi*)
    | Sum of exp * exp
    | Diff of exp * exp
    | Prod of exp * exp
    | Div of exp * exp
    (* Operatori da interi a booleani *)
    | IsZero of exp
    | Eq of exp * exp
    | LessThan of exp*exp
    | GreaterThan of exp*exp
    (* Operatori su booleani *)
    | And of exp*exp
    | Or of exp*exp
    | Not of exp
    (* Controllo del flusso, funzioni *)
    | IfThenElse of exp * exp * exp
    | Let of ide * exp * exp
    | Letrec of ide * ide  * exp * exp
    | Fun of ide * exp
    | Apply of exp * exp
```

```
[1]: type ide = string
```

```
[1]: type tname = TInt | TBool | TString | TClosure | TRecClosure | TUnBound
```

```
[1]: type exp =
         EInt of int
       | CstTrue
       | CstFalse
       | EString of string
       | Den of ide
       | Sum of exp * exp
       | Diff of exp * exp
       | Prod of exp * exp
       | Div of exp * exp
       | IsZero of exp
       | Eq of exp * exp
       | LessThan of exp * exp
       | GreaterThan of exp * exp
       | And of exp * exp
       | Or of exp * exp
       | Not of exp
       | IfThenElse of exp * exp * exp
       | Let of ide * exp * exp
       | Letrec of ide * ide * exp * exp
       | Fun of ide * exp
       | Apply of exp * exp
```

3.2 Ambiente e valori esprimibili

Ambiente come tipo polimorfo:

- L'ambiente associa identificatori a valori. L'implementazione più semplice (come nelle slides del corso) è tramite una lista di coppie (identificatore,valore) e una funzione `lookup` fornisce il valore associato ad un dato identificatore
- I questo caso l'ambiente è implementato come una funzione vera e propria "aggiornabile" tramite la funzione `bind` (sotto)
- L'ambiente è definito come tipo polimorfo per consentire la mutua ricorsione con la definizione del tipo dei valori, evT

```
[2]: (* Ambiente polimorfo *)
     type 't env = ide -> 't
```

```
[2]: type 't env = ide -> 't
```

```
[3]: (* Evaluation types = tipi  esprimibili *)
     type evT =
         | Int of int
         | Bool of bool
         | String of string
         | Closure of ide * exp * evT env
         | RecClosure of ide * ide * exp * evT env
         | UnBound
```

[3]:
```
type evT =
    Int of int
  | Bool of bool
  | String of string
  | Closure of ide * exp * evT env
  | RecClosure of ide * ide * exp * evT env
  | UnBound
```

[4]:
```
(* Ambiente vuoto *)
let emptyenv = function x -> UnBound
```

[4]:
```
val emptyenv : 'a -> evT = <fun>
```

[5]:
```
(* Binding fra una stringa x e un valore primitivo evT *)
let bind (s: evT env) (x: ide) (v: evT) =
    function (i: ide) -> if (i = x) then v else (s i)
```

[5]:
```
val bind : evT env -> ide -> evT -> ide -> evT = <fun>
```

3.3 Type Checking

[6]:
```
(* Funzione da evT a tname che associa a ogni valore il suo descrittore di tipo  *)
let getType (x: evT) : tname =
    match x with
    | Int(n) -> TInt
    | Bool(b) -> TBool
    | String(s) -> TString
    | Closure(i,e,en) -> TClosure
    | RecClosure(i,j,e,en) -> TRecClosure
    | UnBound -> TUnBound
```

[6]:
```
val getType : evT -> tname = <fun>
```

[7]:
```
(* Type-checking *)
let typecheck ((x, y) : (tname*evT)) =
    match x with
    | TInt -> (match y with
               | Int(u) -> true
               | _ -> false
              )
    | TBool -> (match y with
                | Bool(u) -> true
                | _ -> false
               )
    | TString -> (match y with
                  | String(u) -> true
                  | _ -> false
                 )
    | TClosure -> (match y with
                   | Closure(i,e,n) -> true
                   | _ -> false
```

```
                    )
    | TRecClosure -> (match y with
                      | RecClosure(i,j,e,n) -> true
                      | _ -> false
                     )
    |TUnBound -> (match y with
                  | UnBound -> true
                  | _ -> false
                 )
```

[7]:
```
val typecheck : tname * evT -> bool = <fun>
```

3.4 Eccezione in caso di errori durante l'esecuzione

[8]:
```
(* Errori a runtime *)
exception RuntimeError of string
```

[8]:
```
exception RuntimeError of string
```

3.5 Operazioni primitive

[9]:
```
(* PRIMITIVE del linguaggio *)

(* Controlla se un numero è zero *)
let is_zero(x) = match (typecheck(TInt,x),x) with
    | (true, Int(v)) -> Bool(v = 0)
    | (_, _) -> raise ( RuntimeError "Wrong type")

(* Uguaglianza fra interi *)
let int_eq(x,y) =
    match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Bool(v = w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

(* Somma fra interi *)
let int_plus(x, y) =
    match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Int(v + w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

(* Differenza fra interi *)
let int_sub(x, y) =
    match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Int(v - w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

(* Prodotto fra interi *)
let int_times(x, y) =
    match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Int(v * w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")
```

```
(* Divisione fra interi *)
let int_div(x, y) =
    match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) ->
                    if w<>0 then Int(v / w)
                            else raise (RuntimeError "Division by zero")
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

(* Operazioni di confronto *)
let less_than(x, y) =
    match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Bool(v < w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

let greater_than(x, y) =
    match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Bool(v > w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

(* Operazioni logiche *)
let bool_and(x,y) =
    match (typecheck(TBool,x), typecheck(TBool,y), x, y) with
    | (true, true, Bool(v), Bool(w)) -> Bool(v && w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

let bool_or(x,y) =
    match (typecheck(TBool,x), typecheck(TBool,y), x, y) with
    | (true, true, Bool(v), Bool(w)) -> Bool(v || w)
    | (_,_,_,_) -> raise ( RuntimeError "Wrong type")

let bool_not(x) =
    match (typecheck(TBool,x), x) with
    | (true, Bool(v)) -> Bool(not(v))
    | (_,_) -> raise ( RuntimeError "Wrong type")
```

```
[9]: val is_zero : evT -> evT = <fun>
```

```
[9]: val int_eq : evT * evT -> evT = <fun>
```

```
[9]: val int_plus : evT * evT -> evT = <fun>
```

```
[9]: val int_sub : evT * evT -> evT = <fun>
```

```
[9]: val int_times : evT * evT -> evT = <fun>
```

```
[9]: val int_div : evT * evT -> evT = <fun>
```

```
[9]: val less_than : evT * evT -> evT = <fun>
```

```
[9]: val greater_than : evT * evT -> evT = <fun>

[9]: val bool_and : evT * evT -> evT = <fun>

[9]: val bool_or : evT * evT -> evT = <fun>

[9]: val bool_not : evT -> evT = <fun>
```

3.6 Interprete

```
[10]: (* Interprete *)
let rec eval (e:exp) (s:evT env) : evT =
    match e with
    | EInt(n) -> Int(n)
    | CstTrue -> Bool(true)
    | CstFalse -> Bool(false)
    | EString(s) -> String(s)
    | Den(i) -> (s i)

    | Prod(e1,e2) -> int_times((eval e1 s), (eval e2 s))
    | Sum(e1, e2) -> int_plus((eval e1 s), (eval e2 s))
    | Diff(e1, e2) -> int_sub((eval e1 s), (eval e2 s))
    | Div(e1, e2) -> int_div((eval e1 s), (eval e2 s))

    | IsZero(e1) -> is_zero (eval e1 s)
    | Eq(e1, e2) -> int_eq((eval e1 s), (eval e2 s))
    | LessThan(e1, e2) -> less_than((eval e1 s),(eval e2 s))
    | GreaterThan(e1, e2) -> greater_than((eval e1 s),(eval e2 s))

    | And(e1, e2) -> bool_and((eval e1 s),(eval e2 s))
    | Or(e1, e2) -> bool_or((eval e1 s),(eval e2 s))
    | Not(e1) -> bool_not(eval e1 s)

    | IfThenElse(e1,e2,e3) ->
      let g = eval e1 s in
          (match (typecheck(TBool,g),g) with
          |(true, Bool(true)) -> eval e2 s
          |(true, Bool(false)) -> eval e3 s
          |(_,_) -> raise ( RuntimeError "Wrong type")
          )
    | Let(i, e, ebody) -> eval ebody (bind s i (eval e s))
    | Fun(arg, ebody) -> Closure(arg,ebody,s)
    | Letrec(f, arg, fBody, leBody) ->
      let benv = bind (s) (f) (RecClosure(f, arg, fBody,s)) in
          eval leBody benv
    | Apply(eF, eArg) ->
      let fclosure = eval eF s in
          (match fclosure with
          | Closure(arg, fbody, fDecEnv) ->
              let aVal = eval eArg s in
              let aenv = bind fDecEnv arg aVal in
              eval fbody aenv
```

```
            | RecClosure(f, arg, fbody, fDecEnv) ->
                let aVal = eval eArg s in
                let rEnv = bind fDecEnv f fclosure in
                let aenv = bind rEnv arg aVal in
                eval fbody aenv
            | _ -> raise ( RuntimeError "Wrong type")
            )
```

[10]:
```
val eval : exp -> evT env -> evT = <fun>
```

3.7 Esempio di esecuzione: fattoriale

[11]:
```
let myRP =
   Letrec("fact", "n",
          IfThenElse(Eq(Den("n"),EInt(0)),
                     EInt(1),
                     Prod(Den("n"),
                          Apply(Den("fact"),
                                Diff(Den("n"),EInt(1))))),
          Apply(Den("fact"),EInt(3)));;
```

[11]:
```
val myRP : exp =
  Letrec ("fact", "n",
   IfThenElse (Eq (Den "n", EInt 0), EInt 1,
    Prod (Den "n", Apply (Den "fact", Diff (Den "n", EInt 1)))),
   Apply (Den "fact", EInt 3))
```

[12]:
```
eval myRP emptyenv;;
```

[12]:
```
- : evT = Int 6
```