Contents

1 Base base

L'evaluator è "basato" sull'evaluator presente al capitolo `4.1` di Structure and Interpretation of Computer Programs l'evaluator presente nel libro non è purtroppo progettato in maniera molto object oriented, e le idee di fondo di questo vanno riviste ai fini di non creare abominii.

1.1 Idee del cazzo di lisp da tenere a mente

lisp è un linguaggio che sa essere abbastanza stronzo a volte, specie per quel povero crisitano che sta leggendo questo documento per capire che cazzo di codice ho scritto (pace all'anima sua)

ci sono due/tre particolarità di questo essere[1] da tenere a mente quando andrete a soffrirvi sto interpreter

1. La struttura dati base del lisp è la lista (concatenata), la struttura base che rappresenta il codice è la stessa cazzo di lista. questa struttura è descritta da un insieme di coppie di puntatori che contengono

   - un puntatore `head` (o `car`) che punta al primo elemento della lista
   - un puntatore `tail` (o `cdr`) che punta alla cella successiva della lista

   i nomi `car` e `cdr` sono più comuni, più che altro che sono lì dall'alba dei tempi e la gente ci si è abituata.

2. Lisp è *"expression oriented"*, vale a dire, ogni espressione, ogni CAZZO di espressione, ritorna un valore, un if, un loop, un print, una definizione, un assegnamento... tutto.

[1] queste cazzate sarebbero più da scheme/common lisp, che sono quelli che ho usato come "ispirazione"

3. In lisp le funzioni sono variabili, anche un pover `def fizzbuzz (n)` sotto è una variabile (ahimè, una lambda), a cui è stato dato un nome per pietà

4. In lisp le funzioni tengono traccia di tutte le variabili locali presenti quando queste sono state definite (vedre (su google) i termini `Closure`[2] e `Lexical Binding`)

1.2 Quello del libro

L'evaluator del libro è incentrato 2 funzioni

`eval` valuta un'espressione, ritornando il valore che assume

`apply` chiama questa funzione con questi argomenti ("applica" la funzione agli argomenti), e ritornami il valore ritornato dalla funzione

Le strutture dati principali dell'evaluator, oltre all'albero del codice che grazialcazzo, sono

`environment` "ambiente" dell'esecuzione, banalmente che cazzo di variaibili sono definite e che valore hanno, questo include, oltre alla variabili, le funzioni definite fino a quel momento.

`procedure` la funzione

1.3 Funzioni del libro

1.3.1 Eval

1. Nel libro il codice del libro per la funzione `eval` è sotto copiaincollato

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters

                                        (lambda-body exp)
                                        env))
```

[2] specificando che cerchi roba di programmazione, altrimenti ti vengono cose da Poggiolini sulla chiusura di un insieme rispetto a un'operazione

```
         exp)
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)

         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))

        (else
         (error "Unknown expression type: EVAL" exp))))
```

2. Tradotto in java questo, per mortali che toccano erba ed escono di casa, si traduce in una gigantesca catena di `if-else`

```
Value evaluate(Expression exp, Environment env) {
    if(isSelfEvaluating(exp))
        return exp;
    if(isVariable(exp))
        return env.lookupVariabile(exp);
    if(isQuoted(exp))
        return textOfQuotation(exp);
    if(isAssignment(exp)) {
        env.evalAssignment(exp);
        return cazzoNeSo;
    }
    if(isDefinition(exp)) {
        env.evalDefinition(exp);
        return cazzoNeSo;
    }
    if(isIf(exp)) {
        return evaluateIf(exp, env);
    }
    if(isLambda(exp)) {
        return new Procedure(Procedure.parametersOfExpression(exp),
                             Procedure.bodyOfExpression(exp),
                             env);
    }
    if(isSequence(exp)) {
        return evalSequence(exp);
    }
    if(isCond(exp)) {
        Expression ifs = translateCondToIfChain(exp);
        return evaluate(ifs);
    }
    if(isFuncall(exp)) {
        operator = evaluate(expressionOperator(exp));
        operands = evaluateAllInList(expressionOperands(exp));
```

```
        return applyFunction(operator, operands);
    }
    // arrivati qui abbiamo finito le operazioni che sappiamo fare
    // quindi non so che cazzo dirti,
    // se arrivo qui vuo dire che non so gestire l'espressione data
    // quindi ti becchi un'eccezione
    throw new InvalidArgumentException("cannot evaluate given
    ↪  expression");
}
```

visto che non siamo yanderedev, questo si può riarchitetturare (tra gli altri modi) con un

(a) Factory disabile

```
public interface Evaluator {
    public Value evaluate(Expression exp, Environment env);
}

Value evaluate(Expression exp, Environment env) {
    Evaluaotr ev = EvaluatorPicker.determineEvaluator(exp);
    return ev.evaluate(exp, env);
}

/* ... */

class EvaluatorPicker {
    public Evaluator determineEvaluator(exp) {
        if(isCompositeExpression(exp)) {
            return determineFromOperator(exp.getOperator());
        }
        else { // vale a dire, se l'espressione è atomica (un
        ↪  singolo nome o un letterale)
            if(isSymbol(exp))
                return new LookupEvaluator();
            else
                return new ConstantEvaluator();
        }
    }

    private Evaluator determineFromOperator(op) {
        String opName = op.getName();
        Evaluator ev = opnameEvaluatorMap.get(opname);
        if(ev == notFound) {
            // se non è un'operatore predefinito allora era una
            ↪  funzione normale
            return new FunctionApplicationEvaluator();
```

```
            // NOTA : non mi ricordo come si fa a dire in java
            ↪  "non è nella map"
            // quando lo trovo te lo metto bellino
        }
    }

    private HasMap opnameEvaluatorMap =
    {
        {"quote" : new QuoteEvaluator()},
        {"if" : new IfEvaluator()},
        {"set" : new AssignmentEvaluator()},
        {"define" : new DefinitionEvaluator()},
        {"lambda" : new LambdaEvaluator()},
        {"sequence" : new SequenceEvaluator()},
        {"cond" : new CondEvaluator()}
    };
}
```

Se hai idee migliori di questo factory disabile sarei molto felice di sentirle, io le sto un po' finendo.

(b) Factory meno disabile Sarebbe anche possibile avere un `Evaluable` che viene costruito in base alla form, maragi definito come

```
public interface Evaluable {
    Value evaluate(Environment env);
}
```

NOTA 1: non so se sarbbe meglio passare l'env qui o averlo interno, per adesso mettiamo che viene passato, potrebbe cambiare

NOTA 2: poi bisognerebbe trovare anche il modo di unificare il fatto che viene costruita da un'espressione, magari sarebbe meglio avere un'interfaccia che unisce `evaluate()` e un `fromExpression()`

```
public class ConstantExpression implements Evaluable {
    /* ... */
}

public class FunctoinApplication implements Evaluable {
    /* ... */
}
```

(al momento manco gli sbatti per scrivere questa cosa per intero, t'attacchi al cazzo, poi quando si implementa saranno cazzi)

1.3.2 Apply

per definire apply, ovvero l'applicazoine di procedure, dobbiamo un attimo vedere come sono definite le procedure all'interno del programma

1. Procedure le procedure[3] sono, come detto sopra, delle variabili in questo caso degli oggetti, questi avranno

   - degli argomenti
   - un body (cazzo di codice contiene sta funzione)

   risparmiando i dettagli saranno fatte tipo così

   ```
   class Procedure {
       LispList functionArguments;
       LispCode functionBody;

       Environment definitionEnvironment

           public Procedure(LispList arguments, LispList functionBody,
                            Environment definitionEnvironment) {
           this.functionArguments = functionArguments;
           this.functionBody = functionBody;
           this.definitionEnvironment = definitionEnvironment;
       }

       /* getter e setter e sticazzi */
   }
   ```

   quello che ci interessa adesso è il metodo `call`

2. Tornando ad `apply` dato il `Procedure.call(arguments)` sopra definito è abbastanza facile definire `apply`, prima però la definizione del libro, per documentazoine

   ```
   (define (apply procedure arguments)
     (cond ((primitive-procedure? procedure)
            (apply-primitive-procedure procedure arguments))
           ((compound-procedure? procedure)
            (eval-sequence
             (procedure-body procedure)
             (extend-environment
              (procedure-parameters procedure)
              arguments
   ```

[3] salvo quelle builtin, e magari delle foreign call, a cui devo pensare dopo come cazzo fare

```
          (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type: APPLY" procedure))))
```

che tradotta brutalmente darebbe

```
Value applyProcedure(Procedure proc, LispList args) {
    if (proc.isPrimitiveProcedure()) {
        return applyPrimitive(proc, args);
    }
    if (proc.isCompoundProcedure()) {
        return evalSequence(proc.getBody(),
                            ↪  proc.getEnvironment().extend(proc.getArglist,
                                                            args));
    }
}
```

> NOTA : `Environment.extend()` rende un `Environment` con gli stessi binding di `this` a cui si aggiungono poi tutti i nomi dell'`arglist` a cui vengono assegnati tutti i valor dell'`args` dato, veda il lettore se va bene come interfaccia

si nota qui un dispatch esplicito sul "tipo", una bestemmia per ogni apprezzatore di orientazione a oggetti[4], una traduzione un po' più javanese potrebbe essere.

```
public interface LispCallable {
    public Value call(LispList givenArguments);
}

public class BuiltinFunction implements LispCallable {
    /* devo ancora vedere come fare la cosa del builtin, mi spiace
    ↪  */

    @Override
    public Value call(LispList givenArguments) {
        /* il dispatch so' cazzo tua farlo mo' */
    }
}

public class UserDefinedFunction implements LispCallable {
    private LispCode functionBody;
    private LispList formalParameters;
```

[4] e di un minimo di scalabilità

```
    // nome figo per dire "i nomi degli argomenti che devi passarmi"
    private Environment definitionEnvironment;

    public UserDefinedFunction(LispCode functionBody,
                               LispList formalParameters,
                               Environment definitionEnvironment) {
        this.functionBody = functionBody;
        this.formalParameters = formalParameters;
        this.definitionEnvironment = definitionEnvironment;
    }

    @Override
    public Value call(LispList givenArguments) {
        Environment extendedEnvironment =
            definitionEnvironment.extend(formalParameters,
            ↪  givenArguments);
        return evalSequence(functionBody, extendedEnvironment);
    }
}
```

e, avendo tradotto l'infame dispatch in polimorfismo come piace al vicario, si può tradurre `apply` semplicemente come un

```
Value apply(Procedure proc, LispList vals) {
    return proc.call(vals);
}
```