SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione di un dialetto di LISP

Ingegneria dell'Informazione, Informatica e Statistica
Corso di laurea in Informatica

Alessandro Scandone
Matricola 1700455

Relatore
Adolfo Piperno

Anno Accademico 2021/2022

Implementazione di un dialetto di LISP
Tesi di Laurea Triennale. Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: alescandone@gmail.com

A nonno. A nonna. A mia madre.

Sommario

La finalità di questo elaborato è di definire e implementare un dialetto di LISP. Si tratta di un linguaggio definito inizialmente [1] nel 1960 (rendendolo quindi uno dei primi linguaggi di programmazione della storia) come sistema di calcolo formale, ispirato dal lambda calcolo di Alonzo Church. Il nome LISP, acronimo di *list-processor*, richiama la struttura dati principale usata dal linguaggio: una semplice *linked-list*. Tale struttura dati costituisce la rappresentazione dello stesso codice sorgente. In altre parole, il programma `(+ 1 2)`, che viene interpretato - con notazione prefissa - come il numero 3, si può vedere come una lista costituita da 3 elementi: il simbolo `+`, il numero *1*, e il numero *2*. Tale proprietà (comunemente detta *omoiconicità*) permette potenti funzionalità di metaprogrammazione, poiché il linguaggio può manipolare la propria sorgente come una qualunque struttura dati. Storicamente LISP ha introdotto diversi concetti nei linguaggi di programmazione [2] [3], tra cui l'idea delle funzioni come *first-class citizen*, la garbage collection, le espressioni *if*, e altro ancora. Sorprendentemente, a differenza di altri suoi coetanei, si tratta di un linguaggio ancora estremamente attuale. Le caratteristiche citate ne hanno favorito una grande diffusione di dialetti diversi. Un esempio degno di nota è Clojure, che prende l'idea di *code-as-data*, e la porta all'estremo, con altre strutture dati persistenti [4]. Tali caratterestiche lo rendono un linguaggio estremamente flessibile, con campi di applicazione che vanno dalla programmazione web a data science. I concetti di metaprogrammazione nati nel LISP trovano applicazione anche in moltissimi altri linguaggi, come Julia [5], Elixir [6] e tantissimi altri. Il dialetto implementato (che prende ispirazione ma segue scelte diverse rispetto a quello iniziale di McCarthy) definisce anche un proprio modello della concorrenza, che prende fortemente spunto proprio da quello di Elixir [7], comunemente detto *actor model* (e che a sua volta ha delle origini antecedenti rispetto ad Elixir).

L'architettura dell'interprete è composta dalle seguenti fasi [8] [9] [10]:

1. Parsing: la sorgente del programma (es. una stringa) viene trasformata in una struttura dati

2. Compilazione: la struttura dati elaborata precedentemente viene trasformata in una serie di istruzioni primitive

3. Esecuzione: una virtual machine esegue le istruzioni ottenute precedentemente

Le conseguenze di quanto detto sono principalmente due. In primo luogo, mentre tipicamente l'AST del programma è una struttura dati distinta rispetto ai valori runtime, in questo caso sono entrambi rappresentati dallo stesso tipo. In secondo luogo, il passo 2 (compilazione), ha accesso alla piena capacità computazionale del passo 3 (esecuzione), tramite il sistema delle macro. Queste due caratteristiche rendono possibile riordinare arbitrariamente la sintassi a tempo di compilazione. Ad esempio, non abbiamo bisogno di definire un costrutto ad-hoc che valuti l'*and* logico dei valori passati, come avviene negli altri linguaggi, (in cui se il primo valore è falso, il secondo non viene valutato). Avendo a disposizione l'espressione *if* come primitiva, possiamo considerare l'operatore *and* come zucchero sintattico. Ovvero trattare questa espressione:

```
(and a b c)
```

Come zucchero sintattico per quest'altra:

```
(if a
  (if b
    c
    false)
  false)
```

Tale regola sintattica può essere definita all'interno del linguaggio stesso tramite, appunto, una macro. [11] Le macro possono essere adoperate per una grande varietà di utilizzi, come implementare costrutti più ergonomici (ad esempio lo switch-case), creare appositi DSL (*Domain-specific language*) o definire interi nuovi costrutti (come il pattern matching), il tutto senza dover modificare l'implementazione originaria del linguaggio, che si deve occupare di un insieme di operazioni minimale.

L'implementazione segue quindi il seguente principio di design: se è possibile, un costrutto dovrebbe essere implementato come funzione. Altrimenti, se possibile, come macro. Altrimenti, come special form (vedremo di cosa si tratta). Questo processo permette di costruire l'intero linguaggio a partire da poche primitive.
Vedremo quindi in che modo un blocco di codice come il seguente:

```
(defun max (a b)
  (if (> a b)
    a
    b))
```

viene prima trasformato in una struttura dati (la lista che ha come primo elemento il simbolo `defun`, e così via). Le macro vengono poi espanse, trasformando il programma scritto nel seguente:

```
(def max
  (lambda* (a b)
    (do (if (> a b)
        a
        b))))
```

il quale a sua volta viene trasformato in una serie di istruzioni a basso livello simili a queste:

```
000 GetLocal 0
001 GetLocal 1
002 GreatherThan
003 JumpIfNot 6
004 GetLocal 0
005 Jump 7
006 GetLocal 1
007 Return
```

le quali, infine, vengono eseguite dall'apposita virtual machine.

Struttura della tesi

Inizialmente, nella sezione 1, ci occuperemo di definire a livello formale quale debba essere la semantica del linguaggio da implementare. Vedremo inoltre quali funziona-

lità di metaprogrammazione offre il linguaggio, come funziona il suo modello della concorrenza e quali primitive sono necessarie nella libreria standard per implementare il linguaggio completo in maniera incrementale. Nella sezione 2 implementeremo la virtual machine citata precedentemente nella fase di esecuzione, per poi implementarne nella sezione 3 il relativo compilatore. Infine, come appendice, è presente una referenza delle funzionalità del linguaggio host utilizzate per implementare il tutto.

Indice

Capitolo 1

Semantica del linguaggio

La sintassi del linguaggio è costituita principalmente delle cosiddette *s-expression*.

```
; Un symbol è una s-expression.
a
; Altri letterali sono numbers e strings. I seguenti sono s-expressions:
42
42.0
-42
"str"
; Una lista di s-expression èuna s-expression
(a b () (0 1) "abc" (d e () f))
```

Viene naturale interpretare questa notazione come una struttura dati ad albero, che a sua volta possiamo considerare come l'albero della sintassi astratta di un linguaggio. Per esempio *(f a b)* può essere considerata l'applicazione della funzione f ai due argomenti a e b (e viene chiamato *form*). Sono presenti 5 costrutti sintattici che fanno eccezione a questa regola, e che quindi vengono detti *special forms*: `do`, `def`, `if`, `lambda`, e `quote`.

1.1 Semantica operazionale

Per denotarne la semantica operazionale definiamo una relazione (con notazione infissa) $\mapsto$, che opera sul prodotto cartesiano:

$$(Env, Env, Value) * (Env, Value)$$

dove *Value* è l'insieme di tutti i valori del linguaggio, (*Symbol* è l'insieme di tutti i valori del linguaggio di tipo *symbol*, e così via) e *Env* è un sottoinsieme di $\{(n, v) | n \in Symbol, v \in Value\}$, tale che esiste al più una coppia *(n, v)* per ogni symbol n. Meno formalmente, la relazione opera sui seguenti valori:

$$(\text{env globale}, \text{scope lessicale}, \text{valore}) \mapsto (\text{env globale}, \text{valore risultante})$$

e rappresenta la valutazione di un valore. Questa funzione è generalmente chiamata *eval*. Vediamo quindi le regole di inferenza del linguaggio che vogliamo implementare:

1.1.1 Costanti

$$\frac{v \in String \cup Number \cup Function \cup \{\texttt{true}, \texttt{false}, \texttt{()}\}}{(\_, env, v) \mapsto (env, v)}$$

In altre parole, le stringhe, numeri, funzioni, i simboli `true` e `false` e la lista vuota, vengono valutati come sè stessi, senza operare sull'environment.

```
true ; => true
"abc" ; => "abc"
42 ; => 42
() ; => ()
```

Da questo punto in poi useremo la notazione `nil` per indicare la lista vuota (`()`)

1.1.2 `def` special form

Il form `(def name value)` crea una variabile globale di nome *name*, con il valore della valutazione di *value*, e viene valutato come `nil`.

$$\frac{name \in Symbol \quad (l, env, v) \mapsto (env', v')}{(l, env, (\texttt{def}\ name\ v)) \mapsto (env' \leftarrow (name, v'), \texttt{nil})}$$

Nella notazione usata, il simbolo $\leftarrow$ applicato agli environment indica l'aggiunta di un binding, mantenendo l'invariante specificata per l'insieme *Env* (quindi se esiste già un binding con quel simbolo, viene sovrascritto). La stessa notazione verrà usata per gli scope lessicali.

1.1.3 Simboli

Un simbolo `s` (diverso da `true` o `false`) viene valutato tramite il lookup del suo valore prima nello scope lessicale, poi nell'environment globale.

$$\frac{s \in Symbol \quad (s, \mathrm{v}) \in l}{(l, env, s) \mapsto (env, \mathrm{v})}$$

$$\frac{s \in Symbol \quad (s, \mathrm{v}) \in env}{(l, env, s) \mapsto (env, \mathrm{v})}$$

```
(def full-name "John McCarthy") ; => nil

full-name ; => "John McCarthy"
```

1.1.4 `do` special form

Un blocco `(do x ... z)` le espressioni x, ..., z, e viene valutato come il risultato della valutazione di z. Utile per eseguire side-effects.

$$\frac{(l, env, x_1) \mapsto (env_1, x_1') \quad \forall i : (l, env_{i-1}, x_i) \mapsto (env_i, x_i')}{(l, env, (\texttt{do}\ x_1, ..., x_n)) \mapsto (env_n, x_n')}$$

```
(do (def x 42) x) ; => 42
```

1.1.5 `if` special form

`(if condition x y)` ha la classica semantica delle espressioni *if*. Vengono valutati come *falsy* i valori `false` e `nil`.

$$\frac{(l, env, b) \mapsto (\texttt{false}, env')}{(l, env, (\texttt{if } b\ x\ y)) \mapsto (env', y)}$$

$$\frac{(l, env, b) \mapsto (\texttt{nil}, env')}{(l, env, (\texttt{if } b\ x\ y)) \mapsto (env', y)}$$

$$\frac{}{(l, env, (\texttt{if } b\ x\ y)) \mapsto (env', x)}$$

```
(if true 0 1) ; => 0
(if "abc" 0 1) ; => 0
(if false 0 1) ; => 1
(if () 0 1) ; => 1
```

1.1.6 `quote` special form

L'espressione speciale `quote` permette di mantenere delle espressioni non valutate:

$$\frac{}{(\_, env, (\texttt{quote } v)) \mapsto (env, v)}$$

Da qui in avanti utilizzeremo la notazione `'expr` è per indicare l'espressione `(quote expr)`.

```
(quote (a b c)) ; => (a b c)
'(a b c) ; zucchero sintattico per esempio sopra
'(if true 0 1) ; => (if true 0 1)
'x ; => x
'42 ; => 42
```

1.1.7 `lambda` special form

Usiamo la notazione $clo[\_, \_, \_]$ per denotare una valore di tipo *closure*

$$\frac{p \in Symbol}{(l, env, (\texttt{lambda } (p)\ body)) \mapsto (env, clo[l \cup env, p, body])}$$

1.1.8 Applicazione

Se non si applica nessuna delle regole precedenti, un form viene interpretato come normale applicazione di funzione.

$$\frac{(l, env, f) \mapsto (env', clo[l', p, b]) \quad (l' \leftarrow (p, x'), env'', b) \mapsto (env''', v)}{(l, env, (f\ x)) \mapsto (env''', v)}$$

```
(def identity (lambda (x) x))
(identity 42) ; => 42

(def const (lambda (x) (lambda (y) x)))
((const 0) 1) ; => 0

(def glob 42)
((lambda () glob)) ; => 42
```

1.2 Sintassi speciale lambda

Oltre alla regola descritta sopra, la special form `lambda` supporta anche:

1.2.1 Multipli argomenti

```
; astrazione
(lambda (a b c) ...)
; applicazione
(f x y z)
```

1.2.2 Funzioni variadiche

```
; astrazione
(lambda (&rest args) ...)
; applicazione
(f) ; args avrá come valore la lista ()
(f x y z) ; args avrá come valore la lista (x y z)
```

1.2.3 Argomenti opzionali

```
; astrazione
(lambda (&opt a b) ...)
; applicazione
(f x y) ; a avrá come valore x, b avrá come valore y
(f x) ; a avrá come valore x, b avrá come valore ()
(f) ; sia a che b avranno come valore ()
```

1.3 Metaprogrammazione

Il costrutto usato per la definizione delle macro è `defmacro`. La sua esecuzione avviene completamente durante lo step di compilazione. Ad esempio:

```
(defmacro my-macro () 42)

(+ 1 (my-macro))
```

diventa (in fase di compilazione) il programma `(+ 1 42)` (che a sua volta viene valutato, a tempo di esecuzione, come il valore `43`) senza che lo step di esecuzione sia a conoscenza dell'esistenza di questa macro.
In generale, la forma `(defmacro name (`p_1 `...` p_n`) body)` definisce, a tempo di compilazione, la macro *name*, che permette, sempre a tempo di compilazione, di trasformare l'espressione `(name` x_1 `...` x_n`)` nel risultato della funzione `(lambda (`p_1 `...` p_n`) body)` applicata agli argomenti x_1, ..., x_n.

Per esempio, nell'implementazione della libreria standard del linguaggio viene offerto lo zucchero sintattico `defun`, definita come macro in questo modo:

```
(defmacro defun (name params &rest body)
  (list 'def name
    (cons 'lambda (cons params body))))
```

In altre parole, scrivere

```
(defun sum (x y) (+ x y))
```

è equivalente a

```
(def sum (lambda (x y) (+ x y)))
```

Sfortunatamente descrivere il valore di ritorno in questo modo non è particolarmente leggibile. Al fine di poter avere una maggiore ergonomia nell'uso delle macro e di non dover costruire le liste manualmente con operazioni come `cons`, `list`, o simili, introduciamo `backquote`. Come vedremo più in avanti, non si tratta di una primitiva, ma è a sua volta implementata come macro tramite i costrutti già descritti. La seguente suite di test di integrazione definisce le specifiche della macro.

```
behavior of "backquote macro"
it should "behave as quote" in {
  expectToEvalAs("(backquote 1)", 1)
  expectToEvalAs("(backquote a)", Symbol("a"))
  expectToEvalAs(
    "(backquote (1 a))",
    List.of(1, Symbol("a"))
  )
}

it should "interpolate values inside unquote" in {
  expectToEvalAs("(backquote (unquote 42))", 42)
  expectToEvalAs("(def x 42) (backquote (unquote x))", 42)
  expectToEvalAs(
    "(def x 42) (backquote (a (unquote x)))",
    List.of(Symbol("a"), 42)
  )
  expectToEvalAs(
    "(def x 42) (backquote (a (y (unquote x))))",
    List.of(Symbol("a"), List.of(Symbol("y"), 42))
  )
}

it should "interpolate lists inside unquote-splicing" in {
  expectToEvalAs(
```

```
    "(backquote (1 2 (unquote-splicing '(3 4))))",
    List.of(1, 2, 3, 4)
  )
  expectToEvalAs(
    "(def x (list 2 3)) (backquote (1 (unquote-splicing x) 4 5))",
    List.of(1, 2, 3, 4, 5)
  )
}
```

Data l'utilità di questa macro, così come per *quote*, il linguaggio utilizza lo zucchero sintattico `` `expr `` in luogo di `(backquote expr)`, `,expr` per `(unquote expr)`, e `,@expr` per `(unquote-splicing expr)`.
Adesso possiamo riscrivere molto più comodamente la macro citata precedentemente in questo modo:

```
(defmacro defun (name params &rest body)
  `(def ,name (lambda ,params ,@body)))
```

che ha una forma molto più dichiarativa rispetto alla precedente implementazione. Per completezza, vediamo quindi come implementare la macro `backquote`. Essendo una dei primi costrutti implementati la leggibilità è molto scarsa, proprio perché agisce da fondamenta per definire in maniera più ergonomica gli altri.

```
; Funzioni usate:
; (list? lst) ritorna true quando lst èuna lista
; (first lst) ritorna il primo elemento di lst se presente (altrimenti nil)
; (second lst) èanalogo
; (map lst f) ritorna una nuova lista in cui f èapplicata a ogni elemento
    di lst
; es. (map (lambda (x) (+ 1 x)) '(10 20 30)) ritorna '(11 21 31)
; (concat x ... z) concatena le liste x, ..., z
; es (concat '(1 2) '(3 4) '(5 6)) ritorna '(1 2 3 4 5 6)

(defun backquote-helper (nested)
  (if (list? nested)
    (if (eq? (first nested) 'unquote)
      (list 'list (second nested))
      (if (eq? (first nested) 'unquote-splicing)
        (second nested)
        (list 'list (list 'backquote nested))))
    (list 'list (list 'quote nested))))

(defmacro backquote (body)
  (if (list? body)
    (if (eq? (first body) 'unquote)
      (second body)
      (cons 'concat (map body backquote-helper)))
    (list 'quote body)))
```

In altre parole, un'espressione del genere:

```
`(a ,b ,@c)
```

Diventa, a tempo di compilazione, la seguente espressione:

```
(concat (list 'a) (list b) c)
```

Che, a sua volta, viene valutato come

```
; assumendo che `b` valga "b", e che `c` valga '("c1" "c2")
('a "b" "c1" "c2")
```

1.4 Modello della concorrenza

Per quanto riguarda il calcolo concorrente, il linguaggio è dotato di sei primitive: `fork*`, `kill`, `alive?`, `self`, `send` e `receive`.
`fork*` riceve una lambda, che viene chiamata in un nuovo thread. Come valore di ritorno restituisce un intero che rappresenta il `pid` del processo creato. Il thread figlio muore appena muore il thread da cui è stato creato. Una proprietà importante è che la funzione passata a `fork*` può accedere sia alle variabili globali sia allo scope lessicale. Ad esempio:

```
(fork* (lambda ()
  (sleep 500)
  (println "inside fork")))

(println "inside main thread")
(sleep 1000)
```

Questo programma stampa prima il messaggio *inside main thread*, poi *inside fork*. Il programma ha un durata di esecuzione di circa 1000 millisecondi, non 1000 + 500. Per comodità d'ora in avanti useremo una macro al posto della primitiva `fork*`:

```
(defmacro fork (&rest body)
  `(fork* (lambda () ,@body)))
```

Una volta creato un processo, la funzione `alive?` restituisce `true` sse il processo non è ancora terminato. `kill` termina il processo e ritorna `false` se il processo era già terminato.

```
(def fork-pid
  (fork
    (sleep 500)
    (println "done")))

(alive? fork-pid) ; => true
(kill fork-pid) ; => true
(kill fork-pid) ; => false
(alive? fork-pid) ; => false
```

`self` ritorna il pid del processo corrente.
`send` e `receive` agiscono da primitive di sincronizzazione: ogni processo ha una propria *mailbox* di messaggi, ovvero una coda (*FIFO*) di valori. Chiamare `receive` blocca il processo fino a quando un messaggio non è disponibile nella mailbox del proprio processo - rimuovendolo quando arriva. `send` invece prende come argomenti un pid di un processo e un valore (che non abbia tipo *function*) e spedisce quel valore alla mailbox del processo che ha quel pid, bloccando finché non viene ricevuta dal processo. Per esempio:

```
(def main-pid (self))

(defun process-loop ()
  (println `("first value from main" ,(receive)))
  (println `("second value from main" ,(receive))))

(def fork-pid
  (fork (process-loop)))

(send 'a) ; ("first value from main" a)
(send 'b) ; ("second value from main" b)
```

Possiamo vedere come questo sistema ci permette di modellare qualcosa di simile al concetto di stato mutabile:

```
(defun counter-loop (value)
  (let1 (received-value (receive))
    (case (first received-value)
      'get (do
             (send (second received-value) value)
             (counter-loop value))
      'set (counter-loop (second received-value)))))

(defun create-state (initial-state)
  (fork (counter-loop initial-state)))

(defun set (pid value)
  (send pid `(put ,value)))

(defun get (pid)
  (send pid `(get ,(self)))
  (receive))

; esempio
(def counter (create-state 100))
(get counter) ; => 100
(set counter 99)
(get counter) ; => 99
```

Capitolo 2

Stack machine

L'esecuzione runtime avviene tramite una macchina virtuale, ovvero un'astrazione che simula delle istruzioni macchina primitive, implementata in un linguaggio ad alto livello. Questo approccio è più complesso rispetto ad un tree-walk interpreter (un programma che naviga l'albero della sintassi astratta e lo interpreta) ma permette maggiore margine di ottimizzazioni e maggiore flessibilità (ad esempio è possibile implementare la ricorsione di coda con allocazione costante dello stack delle chiamate).
I componenti principali dell'implementazione considerata sono:

- Una struttura dati a stack (LIFO) di valori runtime
- Uno stack di *frames* che contengono la lista di istruzioni da eseguire e un instruction pointer che indica l'indice dell'istruzione corrente.

Le istruzioni (che chiameremo *OpCode*) sono un insieme di operazioni primitive che possono agire sullo stack, sull'instruction pointer, allocare o deallocare frames o effettuare altre operazioni primitive (es. print).
Ad esempio il seguente è un programma che esegue la somma di 42 e 100:

```
Push 42 # Push di `42` sullo stack
Push 100 # Push di `100` sullo stack
Add # Pop dei due numeri, e push della loro somma
```

La codifica di queste istruzioni viene tipicamente fatta in bytecode (come ad esempio in Cpython). Ogni opcode viene codificato come un certo byte e gli argomenti sono i bytes successivi (tanti quanti ne servono a codificare quel dato). I valori letterali (nell'esempio di sopra, 42 e 100) vengono serializzati in un array di costanti, e il bytecode si riferisce all'indice di questo array.
Nella seguente implementazione *non* è stata adottata questa codifica. Per semplicità di esposizione infatti, invece di implementare con un linguaggio a basso livello, è stato scelto il linguaggio Scala (un linguaggio funzionale che compila come java bytecode). Una referenza della funzionalità principali, usata negli esempi, è contenuto alla fine della tesi.

Come prima cosa, bisogna modellare una rappresentazione runtime di un valore del linguaggio (per ora, senza funzioni).

```
// main/value/Value.scala
package value
```

```
sealed trait Value

case class Number(value: Float) extends Value

case class String(value: java.lang.String) extends Value

case class Symbol(value: java.lang.String) extends Value

case class List(value: scala.List[Value] = Nil) extends Value
```

A causa dell'omonimia tra le classi `String` (la classe che rappresenta le stringhe in java, e la classe che rappresenta le stringhe nel linguaggio che stiamo modellando), verrà sempre usato il namespace completo `java.lang.String`. Analogo per `scala.List`.

Definiamo anche i primi opcodes, il minimo necessario per definire un test unitario:

```
// main/vm/opcode/OpCode.scala
package vm.opcode

import value._

sealed trait OpCode

case class Push(value: Value) extends OpCode

case object Add extends OpCode
```

E il relativo test:

```
// test/VmSpec.scala
behavior of "Add opcode"
it should "sum the two topmost values on the stack" in {
  val instructions: Array[OpCode] = Array(
    Push(10),
    Push(20),
    Add,
  )

  new Vm().run(instructions) should be(Number(30))
}
```

Un lettore attento potrebbe aver notato che al costruttore `Push` viene passato un numero, mentre come argomento si aspetta un `Value` (come `Number(10)`). Non si tratta di un errore, ma di una funzionalità di scala (*implicits*) che permette di definire una conversione da un tipo A a un tipo B nel caso in cui, laddove sia necessario un valore di tipo A, ne venga passato uno di tipo B. Sono definiti degli implicits per le conversioni *Boolean* $\rightarrow$ *Value*, *Value* $\rightarrow$ *Boolean*, *Number* $\rightarrow$ *Value*, *String* $\rightarrow$ *Value*. Prima di entrare nel merito dell'implementazione, abbiamo bisogno di creare una struttura dati che agisca da stack. Per brevità l'implementazione è omessa, ma si tratta di un piccolo wrapper (chiamiamolo `ArrayStack`) attorno all'`array` di java, inizializzato a una dimensione fissa (1024), che contiene un intero, il quale a sua

volta definisce l'indice dell'elemento corrente (inizialmente -1). Le operazioni `push` e `pop` accrescono e decrementano l'indice. In questo modo è possibile avere accesso randomico agli elementi dello stack (requisito che verrà usato più in avanti).

Iniziamo quindi l'implementazione:

```
// main/vm/Vm.scala
package vm

class Vm {
  def run(instructions: Array[OpCode]): Value = {
    val loop = new VmLoop(instructions)
    loop.run()
  }

  // Stack inizializzato come vuoto
  private val stack = new ArrayStack[Value]()

  /**
    Lo stack di frames èinizializzato con un solo
    frame che contiene le istruzioni correnti
  */
  private val frames = ArrayStack(
    new Frame(
      /*
          implementazione omessa per ora
          un frame contiene
            * un indice `ip` (instructionPointer)
            * un array di opcodes `instructions`
      */
    )
  )

  private def execute(opCode: OpCode): Unit = opCode match {
    // implementazione omessa per ora
  }

  // Classe annidata
  private class VmLoop(private val instructions: Array[OpCode]) {
    @tailrec
    final def run(): Value = {
      val currentFrame = frames.peek()
      if (
        currentFrame.ip < currentFrame.instructions.length
      ) {
        // fetch dell'istruzione corrente
        val opCode = currentFrame.instructions(currentFrame.ip)
        // incremento instruction pointer
        currentFrame.ip += 1
        // esecuzione
        execute(opCode)
        // ricomincia loop
        run()
      } else {
```

```
        stack.peek()
      }
    }
  }
}
```

In altre parole, eseguiamo l'istruzione all'indice corrente finché non sono terminate le istruzioni.

2.1 Operazioni primitive

A questo punto implementare le istruzioni necessarie a rendere il test verde è abbastanza semplice. All'interno del metodo `execute`, che fa pattern matching sull'OpCode fetchato, definiamo le clausole necessarie:

```
// main/vm/Vm.scala
private def execOp2(f: (Value, Value) => Value): Unit = {
  val y = stack.pop()
  val x = stack.pop()
  val result = f(x, y)
  stack.push(result)
}

private def execute(opCode: OpCode): Unit = opCode match {
  case Push(value) => stack.push(value)
  case Add => execOp2((x, y) => (x, y) match {
    case (Number(na), Number(nb)) => na + nb
    case _ => // gestione errore
  })
}
```

Questo codice è sufficiente per far passare il primo test. L'implementazione per tutte le altre operazioni logico/matematiche è analoga e verrà omessa per brevità. Analoga anche l'implementazione di varie primitive (predicati di tipi, operazioni su lista, e così via). Per completezza, la seguente è l'implementazione di `Pop`:

```
// main/vm/opcode/OpCode.scala
case class Pop extends OpCode

// main/vm/Vm.scala
private def execute(opCode: OpCode): Unit = opCode match {
  //...
  case Pop => stack.pop()
}
```

2.2 Salti

Definiamo adesso l'operazione `Jump`, che riceve un indirizzo *assoluto* ed esegue un salto non condizionato:

```
// main/vm/opcode/OpCode.scala
case class Jump(target: Int) extends OpCode
```

```
// test/VmSpec.scala
behavior of "Jump"
it should "set the instruction pointer" in {
  val instructions: Array[OpCode] = Array(
    /* 0 */ Push(0),
    /* 1 */ Jump(3),
    /* 2 */ Push(1),
    /* 3 */
  )

  new Vm().run(instructions) should be(Number(0))
}
```

L'implementazione è abbastanza semplice:

```
// main/vm/Vm.scala
private def execute(opCode: OpCode): Unit = opCode match {
  // ...

  case Jump(target) =>
    frames.peek().ip = target
}
```

Analogamente definiamo `JumpIfNot`, che esegue un salto se e solo se il valore in cima allo stack è falso:

```
// main/vm/opcode/OpCode.scala
case class JumpIfNot(target: Int) extends OpCode

// test/VmSpec.scala
behavior of "JumpIfNot"
it should "set the instruction pointer when the topmost stack value is
    falsy" in {
  val instructions: Array[OpCode] = Array(
    /* 0 */ Push(0),
    /* 1 */ Push(false),
    /* 2 */ JumpIfNot(4),
    /* 3 */ Push(1),
    /* 4 */
  )

  new Vm().run(instructions) should be(Number(0))
}

it should "not set the instruction pointer when the topmost stack value is
    truthy" in {
  val instructions: Array[OpCode] = Array(
    /* 0 */ Push(0),
    /* 1 */ Push(true),
    /* 2 */ JumpIfNot(4),
    /* 3 */ Push(1),
    /* 4 */
  )
```

```
  new Vm().run(instructions) should be(Number(1))
}
```

L'implementazione è simile a prima:

```
// main/vm/Vm.scala
private def execute(opCode: OpCode): Unit = opCode match {
  // ...

  case JumpIfNot(target) =>
    val value = stack.pop()
    if (!value) {
      frames.peek().ip = target
    }
}
```

2.3 Variabili globali

Anziché identificare le variabili con il loro nome, assegnamo ad ogni variabile un indice crescente. In questo modo - come definito nella semantica operazionale - la definizione di nuove variabili con lo stesso nome fa shadowing rispetto alle precedenti anziché mutarle.

```
// main/vm/opcode/OpCode.scala
case class SetGlobal(name: Int) extends OpCode
case class GetGlobal(name: Int) extends OpCode

// test/VmSpec.scala
behavior of "SetGlobal/GetGlobal opcodes"
they should "set the variable as the topmost stack value/get the variable"
    in {
  val instructions: Array[OpCode] = Array(
    Push(42),
    SetGlobal(0),
    Pop,
    GetGlobal(0)
  )
  new Vm().run(instructions) should be(Number(42))
}

// main/vm/Vm.scala
class Vm {
  // persisting globals across repl loops
  private val globals = mutable.HashMap[Int, Value]()

  private class VmLoop {
    private def execute(opCode: OpCode): Unit = opCode match {
      // ...

      case SetGlobal(ident) =>
        val value = stack.pop()
        globals.put(ident, value)
        stack.push(Nil)
```

```
      case GetGlobal(ident) => globals.get(ident) match {
        // Receiving `None` means that (def) was called inside a lambda not
            yet called
        case None => stack.push(Nil)
        case Some(value) => stack.push(value)
      }
    }
  }
}
```

2.4 Funzioni

Prima di cominciare a definire specifiche e implementazioni delle funzioni, c'è un piccolo refactor da implementare. Intanto, dobbiamo ancora modellare un valore che rappresenta una funzione:

```
// main/value/Value.scala

// Aggiunto parametro di tipo a Value
sealed trait Value[+Op]

case class ArgumentsArity(
  // numero degli argomenti posizionali
  required: Int = 0,

  // true quando sono presenti degli argomenti variadici (&rest)
  rest: Boolean = false,

  // numero degli argomenti opzionali (&opt)
  optionals: Int = 0,
)

case class Function[Op](
  instructions: Array[Op],
  arity: ArgumentsArity = ArgumentsArity(),
) extends Value[Op]
```

Per non avere una dipendenza ciclica (il package `value` che usa `OpCode` e il package `vm.opcode` che usa `Value`), abbiamo aggiunto a `Value` un parametro di tipo che rappresenta l'opcode, in modo da non doverlo importare.

A questo punto possiamo definire gli OpCodes necessari:

```
// main/vm/opcode/OpCode.scala

// alloca frame per la chiamata di funzione
case class Call(passedArgs: Int) extends OpCode

// ritorna al chiamante
case object Return extends OpCode

// all'interno di una funzione, push del parametro i-esimo
case class GetLocal(ident: Int) extends OpCode
```

E i relativi tests:

```
// test/VmSpec.scala
behavior of "Call"
it should "Execute a function without arguments" in {
  /*
    opcodes relativi allo pseudocodice:

    let f = () => 142
    f()
  */


  val fn = Function[OpCode](instructions = Array(
    Push(142),
    Return,
  ))

  val instructions: Array[OpCode] = Array(
    Push(fn),
    Call(0),
  )

  new Vm().run(instructions) should be(Number(142))
}

it should "Execute a two-arguments function" in {
  /*
    opcodes relativi allo pseudocodice:

    let f = (a, b) => a + b
    f(a, b)
  */

  val fn = Function[OpCode](
    arity = ArgumentsArity(required = 2),
    instructions = Array(
      GetLocal(0),
      GetLocal(1),
      Add,
      Return,
    ))

  val instructions = Array[OpCode](
    Push(100),
    Push(200),
    Push(fn),
    Call(2),
  )

  new Vm().run(instructions) should be(Number(300))
}
```

Abbiamo definito il necessario per poter formulare dei test, ma prima di iniziare l'implementazione dobbiamo fare un piccolo passo indietro e tornare alla definizione di `Frame`, che era stata omessa.

```
// main/vm/Vm.scala
private class Frame(
  val fn: Function[OpCode],
  val basePointer: Int,
) {
  var ip = 0

  def instructions: Array[OpCode] = fn.instructions
}

class Vm {
  // ...
  private class VmLoop(private val instructions: Array[OpCode]) {
    private val frames = ArrayStack(
      new Frame(
        fn = Function(instructions),
        basePointer = 0,
      )
    )
  }
}
```

Dove `basePointer` è l'indice dello stack nel momento in cui la funzione viene chiamata. In questo modo gli argomenti passati alla funzione vengono allocati negli slot dello stack *successivi* al `basePointer`. Di conseguenza l'argomento i-esimo si può accedere in *O(1)* all'indice $basePointer + i$.
Adesso finalmente possiamo procedere con l'implementazione degli opcodes definiti:

```
// main/vm/Vm.scala
class Vm {
  // ...
  private class VmLoop(private val instructions: Array[OpCode]) {
    private def execute(opCode: OpCode): Unit = opCode match {
      // ...

      case GetLocal(ident) =>
        val index = frames.peek().basePointer + ident
        val retValue = stack.get(index)
        stack.push(retValue)

      case Return =>
        val retValue = stack.pop()
        val numLocals = stack.length() - frames.peek().basePointer
        for (_ <- 0 until numLocals) {
          stack.pop()
        }
        stack.push(retValue)
        frames.pop()

      case Call(argsGivenNumber) =>
```

```
    // omesso: controlla che fn abbia tipo Function (altrimenti lancia
        eccezione)
    val fn = stack.pop()
    // lista degli ultimi `argsGivenNumber` valori dello stack
    val givenArgs = (0 until argsGivenNumber)
          .map(_ => stack.pop())
          .reverse
          .toList
    callFunction(fn, givenArgs)
}

private def callFunction(
  fn: Function[OpCode],
  givenArgs: scala.List[Value[OpCode]]
): Unit =
  /*
    Il metodo `parse` della classe ArgumentsArity ritorna
    una struttura dati (ParsedArguments) che contiene la lista
    degli argomenti obbligatori, degli argomenti opzionali,
    e l'eventuale lista di argomenti variadici.
    L'implementazione èomessa.
  */
  fn.arity.parse(givenArgs) match {
    // ... gestione errori
    case Right(parsedArgs) =>
      val basePointer = stack.length()
      handleCallPush(parsedArgs)

      frames.push(new Frame(
        fn = fn,
        basePointer = basePointer
      ))
  }

// TODO spiegare meglio, forse omettere implementazione
private def handleCallPush(
  parsedArgs: ParsedArguments[Value[OpCode]]
): Unit = {
  for (arg <- parsedArgs.required) {
    stack.push(arg)
  }

  val (optionalPassed, optionalsNotGiven) = parsedArgs.optionals
  for (arg <- optionalPassed) {
    stack.push(arg)
  }
  for (_ <- 0 until optionalsNotGiven) {
    stack.push(Nil)
  }

  for (restArgs <- parsedArgs.rest) {
    stack.push(List(restArgs))
  }
```

```
    }
  }
}
```

Riepilogando, abbiamo implementato la possibilità di definire e chiamare funzioni, come first-class citizen, con tanto di argomenti opzionali e variadici. Ma, cosa più importante: il codice appena descritto permette la ricorsione, sfruttando la definizione di `SetGlobal`. Ecco il relativo test unitario:

```
it should "handle recursion" in {
  /*
    (defun f (n)
      (if (> n 4)
        n
        (f (+ 1 n))))

    (f 0)
  */

  val LIM = 4.toFloat

  val fn = Function[OpCode](
    arity = ArgumentsArity(required = 1),
    instructions = Array(
      /* 00 */ GetLocal(0),
      /* 01 */ Push(LIM),
      /* 02 */ GreaterThan,
      /* 03 */ JumpIfNot(6),
      /* 04 */ GetLocal(0),
      /* 05 */ Jump(11),
      /* 06 */ Push(1), // <- 03
      /* 07 */ GetLocal(0),
      /* 08 */ Add,
      /* 09 */ GetGlobal(0),
      /* 10 */ Call(1),
      /* 11 */ Return,
    )
  )

  val instructions = Array[OpCode](
    Push(fn),
    SetGlobal(0),
    Pop,
    Push(0),
    GetGlobal(0),
    Call(1),
  )

  new Vm().run(instructions) should be(Number(LIM + 1))
}
```

Sfortunatamente, se il numero di frames allocato è maggiore della capacità dello stack, il programma dovrà lanciare eccezione per stack overflow. Vedremo in seguito come risolvere questo problema.

2.5 Chiusure

La limitazione dell'attuale implementazione delle funzioni è che non sono supportate le chiusure. Cominciamo quindi a definirne il relativo tipo:

```
// main/value/Value.scala

case class Closure[Op](
    freeVariables: Array[Value[Op]],
    fn: Function[Op],
) extends Value[Op]
```

gli OpCode necessario:

```
// main/vm/opcode/OpCode.scala
case class PushClosure(
    freeVariables: Int,
    fn: Function[OpCode]
) extends OpCode

case class GetFree(ident: Int) extends OpCode
```

e, come al solito, il test unitario:

```
// test/VmSpec.scala

it should "execute PushClosure" in {
  // a => b => a + b

  val inner = Function[OpCode](
    arity = ArgumentsArity(required = 1),
    instructions = Array(
      GetFree(0),
      GetLocal(0),
      Add,
      Return,
    )
  )

  val outer = Function[OpCode](
    arity = ArgumentsArity(required = 1),
    instructions = Array(
      GetLocal(0),
      PushClosure(1, inner),
      Return
    )
  )

  val instructions = Array[OpCode](
    Push(100),
    Push(200),
    Push(outer),
    Call(1),
    Call(1),
  )
```

```
  new Vm().run(instructions) should be(Number(300))
}
```

Di nuovo, prima di implementare serve realizzare un piccolo refactor: la classe `Frame`, non prende più come argomento un valore di tipo `Function`. L'argomento dovrà invece avere tipo `Closure`. Il refactor non introduce regressione, poiché data una `Function` si può sempre costruire una `Closure` (ma non viceversa).
Ecco che possiamo quindi implementare i relativi OpCodes:

```
// main/vm/Vm.scala
private def execute(opCode: OpCode): Unit = opCode match {
  // ...

  case PushClosure(freeVariablesNum, fn) =>
    val freeVariables =
        (0 until freeVariablesNum)
          .map(_ => stack.pop())
          .toArray

    stack.push(Closure(freeVariables, fn))

  case GetFree(ident) =>
    val value = frames.peek().closure.freeVariables(ident)
    stack.push(value)
}
```

2.6 Primitive della concorrenza

La primitiva `fork` corrisponde alla creazione di un daemon thread java (ovvero un thread che termina quando termina il main thread). La primitiva `self` restituisce l'id del thread java corrente e le primitive di sincronizzazione (`send` e `receive`) utilizzano la classe `java.util.concurrent.LinkedTransferQueue` come coda di messaggi.

2.7 Ottimizzazione della ricorsione di coda

Per come è stata implementata, la vm presenta un difetto critico: la ricorsione - che è l'unico meccanismo di looping presente nel linguaggio - alloca un nuovo frame per ogni chiamata, e di conseguenza il programma termina per *StackOverflow* (rispetto allo stack dei frame di chiamata) se la ricorsione è troppo profonda. Un'ottimizzazione a cui possiamo ricorrere è quella di evitare di allocare nuovi frames nel caso della ricorsione di coda. La ricorsione di coda è una forma di ricorsione che si verifica quando la chiamata ricorsiva è l'ultimo step della funzione. Ad esempio la seguente funzione ricorsiva *non* di coda:

```
; somma gli elementi della lista lst
(defun sum-list (lst)
  (if (nil? lst)
    0
```

```
    (+ (first lst) (sum-lst (rest lst)))))
```

Può essere trasformata in ricorsione di coda, nel seguente modo (equivalente):

```
(defun sum-list-helper (lst acc)
  (if (nil? lst)
    acc
    (sum-lst-helper (rest lst) (+ 1 (first lst)))))

(defun sum-list (lst)
  (sum-list-helper 0))
```

Mentre nella prima formulazione, *dopo* la chiamata ricorsiva, viene effettuata l'operazione `+`, nella seconda viene restituito direttamente il valore della chiamata ricorsiva (o il caso base).

Per semplicità, applicheremo un'ottimizzazione runtime, che ha come costo un piccolo overhead, ma evita complesse analisi del codice statico (che, data la dinamicità del linguaggio, rischiano di essere unsafe). L'euristica usata è di attivare tale ottimizzazione quando valgono le seguenti condizioni:

- La funzione chiamante è uguale alla funzione relativa al frame corrente
- L'istruzione attuale è `Return` o un salto non condizionato a un'istruzione `Return`

L'implementazione è quindi, con qualche semplificazione:

```
// ...
@tailrec
private def isTailCall(closure: Closure[OpCode], ip: Int): Boolean =
  closure.fn.instructions(ip) match {
    case Return => true
    case Jump(target) => isTailCall(closure, target)
    case _ => false
  }

private def callFunction(
  closure: Closure[OpCode],
  givenArgs: scala.List[Value[OpCode]]
): Unit = {
  if (
    frames.peek().closure.fn == closure.fn &&
    isTailCall(closure, frames.peek().ip)
  ) {
    // ottimizzazione per la ricorsione di coda

    /*
      Omesso: push degli argomenti come nella
      normale chiamata di funzione, ma invece di
      farlo su un nuovo frame, viene fatto a partire
      dal `basePointer`, sovrascrivendo i vecchi argomenti.
    */

    frames.peek().ip = 0
```

```
  } else {
    // stesso codice di prima
  }
}
```

Capitolo 3

Parsing e compilazione

Abbiamo definito come eseguire una sequenza di OpCodes. Quello che introdurremo in questa sezione è la funzionalità di trasformare un programma scritto nella sintassi che abbiamo definito inizialmente, nell' input che la virtual machine riceve. In primo luogo, serve eseguire il parsing del linguaggio in una struttura dati adeguata. L'aspetto interessante è che non c'è bisogno di creare una struttura dati ad hoc che rappresenti l'AST del programma. Possiamo rappresentare l'AST tramite lo stesso tipo `Value` che abbiamo usato per rappresentare i valori runtime.

Non entreremo particolarmente nel dettaglio per quanto riguarda il parsing, di cui trascrivo solamente l'API esterna:

```
// main/value/parser/Parser.scala
package value.parser

object Parser {
  // ParseResult rappresenta i possibili risultati (parsing riuscito o
      meno)
  def run(input: CharSequence): ParseResult[scala.List[Value[Nothing]]] =
    // implementazione omessa
}
```

Internamente l'approccio usato è quello dei cosiddetti parsing combinators. Se non ci sono stati errori sintattici, abbiamo a disposizione una lista di `Value[Nothing]` [1]
La fase di compilazione consiste nel trasformare l'AST ricevuto nella lista di `OpCode` in modo che la virtual machine li possa eseguire.
La seguente è la struttura delle classi principali:

```
// main/compiler/Compiler.scala
package compiler

class Compiler {
  def compile(values: scala.List[Value[OpCode]]): Array[OpCode] = {
    // Interpreta la lista di valori come se fosse un blocco "do"
    val block = List[OpCode](
      Symbol("do") :: values
    )
```

[1] Il parametro di tipo per definire gli opcode stavolta è `Nothing` (un tipo non abitato) anziché `OpCode`. Il motivo è che la fase di parsing non dovrebbe ritornare valori di tipo `Function` o `Closure`.

```
    val compiler = new CompilerLoop()
    compiler.compile(block)
    compiler.collect()
  }

  private class CompilerLoop {
    /*
      Emitter èun buffer di OpCodes. L'implementazione èomessa.
    */
    private val emitter = new Emitter()

    def collect(): Array[OpCode] = emitter.collect

    def compile(value: Value[OpCode]): Unit = value match {
      // implementazione omessa, per ora
    }
  }
}
```

3.1 Do special form e valori letterali

Quanto scritto ci basta per poter formulare dei primi test unitari (riportati qui in ordine sparso):

```
// Numbers, Strings, i simboli "true" e "false" dovrebbero compilare come
    costanti
"42".shouldCompileAs(
  Push(42)
)

"(do)".shouldCompileAs(
  Push(Nil)
)

"(do 42)".shouldCompileAs(
  Push(42),
)

"(do 1 2 3)".shouldCompileAs(
  Push(1),
  Pop,
  Push(2),
  Pop,
  Push(3),
)
```

Ed ecco la logica necessaria per renderli verdi:

```
// main/compiler/Compiler.scala

class Compiler {
  // ...
```

```
  private class CompilerLoop {
    def compile(value: Value[OpCode]): Unit = value match {
      case Number(_) | String(_) | Symbol("true") | Symbol("false") =>
        emitter.emit(Push(value))

      // ...

      case List(forms) => forms match {
          case Nil => emitter.emit(Push(value))

          case Symbol("do") :: block => compileBlock(block)
      }
  }

  private def compileBlock(block: scala.List[Value[OpCode]]): Unit =
    block match {
      case Nil => emitter.emit(Push(Nil))
      case value :: Nil => compile(value)
      case _ => for ((value, index) <- block.zipWithIndex) {
        if (index != 0) {
          emitter.emit(Pop)
        }

        compile(value)
      }
    }
}
```

3.2 Quote special form

La compilazione della special form `quote` è particolarmente semplice. Definiamo intanto cosa ci aspettiamo che sia l'output:

```
"(quote abc)".shouldCompileAs(
  Push(Symbol("abc"))
)

"(quote (1 2 3))".shouldCompileAs(
  Push(List.of(1, 2, 3))
)
```

E la logica è semplicemente la seguente:

```
// main/compiler/Compiler.scala
class Compiler {
  // ...
  class CompilerLoop {
    // ...
    def compile(value: Value[OpCode]): Unit = value match {
      // ...
      case List(forms) => forms match {
        // ...
```

```
        case Symbol("quote") :: args => args match {
          case value :: Nil => emitter.emit(Push(value))
          case _ => // gestione errori
        }
      }
    }
  }
}
```

3.3 Operazioni primitive

Vediamo ora come compilare le operazioni primitive. Prendiamo come esempio l'opcode `Add`.

```
"(builtin/add 10 20)".shouldCompileAs(
  Push(10),
  Push(20),
  Add,
)
```

Nb. `builtin/add` è un normale simbolo, non si tratta di sintassi speciale. Nella standard library del linguaggio verrà poi definita la funzione

```
(defun + (a b)
  (builtin/add a b))
```

In modo da poter utilizzare la somma come valore (ad esempio `(list +)`, la lista che contiene la funzione `+`)
Anche stavolta l'implementazione è abbastanza semplice:

```
// main/compiler/Compiler.scala
class Compiler {
  // ...
  class CompilerLoop {
    // ...
    def compile(value: Value[OpCode]): Unit = value match {
        // ...
        case List(forms) => forms match {
          // ...
          case Symbol("builtin/add") :: args => compileOp2(Add, args)
        }
      }
    }

    private def compileOp2(
      op: OpCode,
      args: scala.List[Value[OpCode]],
    ): Unit = args match {
      case x :: y :: Nil =>
        compile(x)
        compile(y)
        emitter.emit(op)
      case _ => // gestione errori
```

```
    }
  }
}
```

L'implementazione è analoga per le altre operazioni primitive, comprese quelle di arità differente.

3.4 If special form

Le espressioni `if` vengono compilate tramite i salti.

```
"(if true \"when true\" \"when false\")".shouldCompileAs(
  /* 0 */ Push(true),
  /* 1 */ JumpIfNot(4),
  /* 2 */ Push("when true"),
  /* 3 */ Jump(5),
  /* 4 */ Push("when false"), // <- 1
  /* 5 */ // <- 3
)

"""
  (if (builtin/greater-than 100 200)
    (builtin/add 10 20)
    (builtin/not ()))
""".shouldCompileAs(
  /* 00 */ Push(100),
  /* 01 */ Push(200),
  /* 02 */ GreaterThan,
  /* 03 */ JumpIfNot(8),
  /* 04 */ Push(10), // if branch
  /* 05 */ Push(20),
  /* 06 */ Add,
  /* 07 */ Jump(10),
  /* 08 */ Push(Nil), // else branch <- 3
  /* 09 */ Not,
  /* 10 */ // <- 7
)

// main/compiler/Compiler.scala
class Compiler {
  class CompilerLoop {
    def compile(value: Value[OpCode]): Unit = value match {
      case List(forms) => forms match {
        case Symbol("if") :: args => args match {
          case cond :: a :: b :: Nil => compileIf(cond, a, b)
          case _ => // gestione errori
        }
      }
    }

    private def compileIf(
      cond: Value[OpCode],
      branchTrue: Value[OpCode] = Nil,
```

```
      branchFalse: Value[OpCode] = Nil,
    ): Unit = {
      compile(cond)

      /*
        Lascia uno spazio per un opcode da "riempire" successivamente
      */
      val beginBranchTrue = emitter.placeholder()
      compile(branchTrue)

      val beginBranchFalse = emitter.placeholder()

      /*
        eta reduction di
        beginBranchTrue.fill(index => JumpIfNot(index))
        dove `index` èil numero dell'istruzione corrente.
      */
      beginBranchTrue.fill(JumpIfNot)
      compile(branchFalse)
      beginBranchFalse.fill(Jump)
    }
  }
}
```

3.5 Def special form e bindings globali

Come già accennato, identifichiamo le variabili con degli interi. Per gestire il corretto scoping, e corrispondenza rispetto al nome originario, useremo una classe `SymbolTable`, la cui implementazione è omessa per semplicità.

Ci aspettiamo un output del genere:

```
"(def x 10)".shouldCompileAs(
  Push(10),
  SetGlobal(0),
)
```

Bisogna però implementare anche la parte di lookup di un binding globale. Il risultato atteso è di ottenere questo:

```
"(def x 10) (def y 20) (builtin/add x y)".shouldCompileAs(
  Push(10),
  SetGlobal(0),
  Pop,

  Push(20),
  SetGlobal(1),
  Pop,

  GetGlobal(0),
  GetGlobal(1),
  Add,
)
```

Per farlo, serve innanzitutto far mantenere alla classe `Compiler` un'istanza della `SymbolTable`, affinché anche all'interno della repl o iterando più espressioni, si possano conservare i bindings globali salvati. Attenzione: in fase di compilazione non stiamo salvando il *valore* del lookup (quella è una logica runtime), ma l'identificatore a cui è stato associato un binding, e il suo tipo (es. globale, lessicale).

```
class Compiler {
  private val symbolTable = new SymbolTable()

  private class CompilerLoop {

    def compile(value: Value[OpCode]): Unit = value match {
      // ...
      case Symbol(name) => symbolTable.resolve(name) match {
        case None => throw new Error(s"Binding not found: $name")
        case Some(sym) => sym.scope match {
          case Global => emitter.emit(GetGlobal(sym.index))
          // vedremo in seguito gli altri tipi di bindings
        }
      }

      case List(forms) => forms match {
        case Symbol(Compiler.DEF) :: args => args match {
          case Symbol(name) :: args2 => args2 match {
            case value :: Nil => compileDef(name, value)
            case _ => // gestione errori
          }
      }
    }

    private def compileDef(
      name: java.lang.String,
      value: Value[OpCode] = Nil,
    ): Unit = {
      compile(value)

      val symbol = symbolTable.define(name, isGlobal = true)
      emitter.emit(SetGlobal(symbol.index))
    }
  }
}
```

3.6 Lambda special form e bindings locali

Mentre la compilazione del body non è particolarmente complessa (basta creare un'istanza di `Function` che contenga il valore compilato del body, terminante con l'opcode `Return`), la gestione dei binding locali richiede un poco più di attenzione. Così come quelli globali, sono identificati da un intero crescente (l'ordine in cui appare tra i parametri della lambda). Ecco un esempio:

```
"(def x 42) (lambda* (y) (builtin/add x y))".shouldCompileAs(
```

```
  Push(42),
  SetGlobal(0),

  Pop,

  Push(Function(
    arity = ArgumentsArity(required = 1),
    instructions = Array(
      GetGlobal(0), // x
      GetLocal(0), // y
      Add,
      Return
    )
  )),
)
```

Eviteremo di entrare nel merito dei dettagli implementativi. In sintesi, quello che viene fatto è di, dopo aver compilato i parametri (cioè, trasformato la lista di parametri, che potrebbero contenere paramtri speciali come `&rest` o `&opt`), definire il relativo simbolo nella lookup table.
Durante il lookup del simbolo (visto nella sezione precedente), basta aggiungere il caso che l'identificatore abbia scope locale anziché globale ed emettere il rispettivo opcode (`GetLocal`).

Nella standard library definiamo poi la macro `lambda`, che è un semplice zucchero sintattico che permette di avere multiple forms come body [2]

```
(defmacro lambda (params &rest body)
  `(lambda* ,params (do ,@body)))
```

La compilazione dei form della forma `(f x ... z)`, invece, è abbastanza semplice:

```
for (arg <- args) {
  compile(arg)
}
compile(f)
emitter.emit(Call(args.length))
```

3.6.1 Chiusure

L'implementazione corrente non ci permette di usare lambda che siano chiusure. Vediamo allora innanzitutto *cosa* vogliamo ottenere:

```
val innerFunction = Function(
  arity = ArgumentsArity(required = 1),
  instructions = Array(
    GetFree(0), // x
    Return,
  )
)

val outerFunction = Function(
```

[2] Poiché l'espressione `(do expr)` viene compilata esattamente come expr, non ci sono penalità runtime ad usare la macro `lambda` definita in questo modo

```
  arity = ArgumentsArity(required = 1),
  instructions = Array(
    GetLocal(0), // x
    PushClosure(1, outerFunction),
    Return,
  )
)

"(lambda* (x) (lambda* (y) x))".shouldCompileAs(
  Push(outerFunction),
)
```

Anche in questo caso, la classe `SymbolTable` si occupa di stabilire se un binding è stato dichiarato in uno scope locale più esterno. Il compilatore itera quindi sulle variabili free della lambda da generare, e genera le istruzioni per metterle sullo stack (`GetGlobal`, `GetLocal`, o `GetFree` a seconda del loro scope). Infine genera l'istruzione `PushClosure(n, fn)` dove n è il numero di variabili free.

3.6.2 Shadowing

Possiamo osservare come la corrente implementazione di bindings globali e lambda, faccia shadowing delle variabili globali in maniera corretta:

```
(def x 0)
(defun f-1 () x)

(def x 1)
(defun f-2 () x)

(f-1) ; => 0
(f-2) ; => 1
```

3.7 Macro

Come preannunciato, le macro di LISP hanno piena capacità computazionale (hanno accesso a qualunque costrutto a cui ha accesso il resto linguaggio). Ne consegue che la classe `Compiler` ha accesso a un'istanza di `Vm`! Non ad un'istanza qualunque, ma alla stessa che sta eseguendo il codice runtime. Può sembrare una strana decisione di design non mantenere due strati completamente separati (è quello che succede ad esempio in elixir) ma è una scelta necessaria a fare in modo che i valori globali definiti precedentemente possano essere usati all'interno di `defmacro`.

Inoltre, il compilatore ha bisogno di tenere traccia delle macro registrate con `defmacro`. Le macro sono rappresentate internamente come `Function`, e identificate, a differenza delle variabili, con il loro nome. Infatti, non serve più implementare una semantica di shadowing come con le variabili globali: le macro vengono espanse a tempo di compilazione. In termini di codice, il tutto diventa:

```
class Compiler(vm: Vm = new Vm()) {
  private val macros =
    new mutable.HashMap[java.lang.String, Function[OpCode]]()
}
```

Per semplicità, il resto del codice è omesso, ma una sintesi è la seguente:
Il costrutto `defmacro` compila il body come funzione, lo aggiunge (o sovrascrive) all'hashmap di macro, ed emette solamente l'opcode `Push(Nil)`. I forms della forma `(f x ... z)`, nel caso f sia una macro, vengono espansi:

```
/*
`instructions` ha la forma:
  Push(x)
  ...
  Push(z)
  Push(macroFunction) // il valore della macro f
  Call(args.length)
*/
val result = vm.run(instructions)
// viene compilato _il risultato_ ottenuto dalla macro
compile(result)
```

Capitolo 4

Referenza sintassi Scala

A seguire una breve descrizioni della sintassi di Scala usata nel corso della tesi

4.1 Class

La keyword *class* crea una class (viene compilata come una classe java).

```
// Una classe con i seguenti campi
// name, che ha tipo int ed èpubblico e immutabile
// age, che ha tipo string ed èpubblico e mutabile
// password, che ha tipo string ed èprivato e mutabile
class Person(
      val name: String,
      var age: Int,
      private var password: String
) {

  // metodo privato
  private def incrementAge(): Unit = {
    this.age += 1
  }

  // metodo pubblico
  def checkPassword(triedPassword: String): Boolean =
    triedPassword == password
}

Non esistono metodi statici
```

4.2 Object

La keyword *object* crea un singleton (classe con uno e un solo abitante)

```
object Config {
  val USERNAME = "alescandone"
  private val PASSWORD = "password"
```

```
  // Un metodo di un singleton èun concetto equivalente a un metodo
      statico di una classe
  def login(): Unit {
    loginToService(USERNAME, PASSWORD)
  }
}
```

4.3 Inner class

Una *inner class* è una classe membro di un'altra classe. Di conseguenza, può essere istanziata solo in relazione a un'istanza della classe esterna.

```
class C(x: Int) {
  class Inner(y: Int) {
    def m() = {
      // accesso a `x` e `y`
    }
  }
}
```

In questo esempio, se `c1` e `c2` sono due istanze distinte di `C`, allora il tipo `c1.Inner` ha tipo diverso da `c2.Inner`.

4.4 Case class

La keyword *case* crea una classe immutabile, con implementazioni default dei metodi `toString()` e `equals()`

```
// Definizione
case class Point(x: Int, y: Int)

// Utilizzo
val origin = Point(0, 0)

// Si può accedere alla notazione tramite la notazione usuale
origin.x

// O tramite pattern matching (esempio successivo)

// Può anche avere degli argomenti di default
case class X(arg: Int = 0)
X().arg // => 0
```

4.5 Trait

Un trait di scala è simile a un'interfaccia java. Un trait può anche essere definito come vuoto (senza nessun metodo da implementare).

```
// trait senza nessun metodo necessario
trait T
```

```
class C() extends T
```

4.6 Sealed classes

Una sealed class è una classe i cui sottotipi possono essere dichiarati solo nello stesso file. Questo permette di conoscerli a tempo di compilazione e di poterne fare pattern matching, segnalando come errore di compilazione un matching non esaustivo.

```
sealed trait PeanoNumber

object Zero extends PeanoNumber

case class Succ(n: PeanoNumber) extends PeanoNumber

def peanoToInt(p: PeanoNumber): Int = p match {
  case Zero => 0
  case Succ(prev) => 1 + peanoToInt(prev)
}
```

4.7 Standard library

List

La classe `List` è una linked list immutabile. I due costruttori sono `Nil` e `::`. Ad esempio:

```
// equivalente a List(), la lista vuota
Nil

// equivalente a List(1), la lista che contiene il numero "1"
1 :: Nil

// equivalente a List(1, 2, 3)
1 :: 2 :: 3 :: Nil
```

Option

Rappresenta un valore che potrebbe essere presente o meno, i due costruttori sono `None` (se il tipo è assente) o `Some` (se il valore è presente).

Capitolo 5

Conclusione

Abbiamo visto nel dettaglio ogni passo necessario per realizzare un interprete funzionante. Abbiamo stabilito un set di operazioni di basso livello, architettato una semplice virtual machine per eseguirli e progettato un compilatore per trasformare le istruzioni ad alto livello nel suddetto insieme di operazioni. Abbiamo infine realizzato un sistema di macro per ottenere nuovi costrutti a partire dai pochi implementati.

Il progetto realizzato offre diversi spunti per degli sviluppi futuri:
1) riscrivere l'interprete qui realizzato in un linguaggio a basso livello (come C o rust). Codificare gli opcodes espressi tramite costrutti ad alto livello, in vere e proprie istruzioni bytecode. Implementare manualmente la garbage collection (invece di basarsi su quella del linguaggio host). L'algoritmo di garbage collection *mark-and-sweep* ha origine proprio nel LISP.
2) grazie al minimalismo del linguaggio realizzato, implementare ottimizzazioni nel codice compilato. Ad esempio un'ottimizzazione della ricorsione di coda come meccanismo compile time anziché runtime. In generale realizzare un generazione più efficiente degli opcode prodotti.

Bibliografia

[1] John McCarthy *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, 1960.

[2] Paul Graham, *What made LISP different*, 2002.

[3] Paul Graham, *The Roots of Lisp*, 2002.

[4] *Clojure Rationale.*

[5] *Metaprogrammazione in Julia.*

[6] *Metaprogrammazione in Elixir.*

[7] Saša Juric *Elixir in action*, 2019.

[8] Thorsten Ball, *Writing A Compiler In Go*, 2018.

[9] Thorsten Ball, *Writing An Interpreter In Go*, 2018.

[10] Robert Nystrom, *Crafting Interpreters*, 2021.

[11] *The Common Lisp HyperSpec*, 1996.