Jelly: Java embeddable lisp intepreter

Biggie Dickus

August 18, 2023

Contents

pardon per la *libertà artistica* nella scelta e validità dell'acronimo

1 Idea Generale

Lo scopo è quello di creare un interpreter per lisp che possa essere utilizzato per i cazzi sua o invocato da un app java, l'idea è quella di poter utilizzare questo, tramite una qualche `FFI` (`Foreign Function Interface`), per offrire funzionalità di scripting all'interno dell'app java, in modo non dissimile a quanto si possa fare con un python o un clojure. Questo, si ritiene, non sarebbe niente male come strumento per la configurazione, senza bisogno

di re-build, di sistemi complessi (si pensi alla configurazione di un server, o a uno script-fu stile gimp, a un actionscript, ...), o per permettere un utilizzo interattivo stile REPL del sistema, strumento molto utile durante fasi di prototipazione, e utilizzabile anche per un testing preliminare (considerando che tra lo scriverlo e il testarlo con una shell passano 10 secondi). Quest'ultimo, in quanto per essere fatto bene richiederebbe anche un minimo di integrazione, o api per integrazione, con editor/ide, non sarà fatto bene.

Il design è vagamente influenzato da ecl, per quanto Jelly non contenga un compiler da lisp a java, visto che non sono ancora a quei livelli.

2 Componenti

2.1 Lisp objects

In lisp tutto ha un valore di ritorno, nell'ambito di `Jelly` questo sarà un oggetto che implementa l'interfaccia `LispObject` (o `LispValue`, devo decidermi sul nome), visto che in lisp *le variabili non hanno un tipo, e sono invece i valori che hanno un tipo*[1] le implementazioni di `LispObject` terranno qualche attributo di tipo.

2.1.1 Tipizzazione

Questo è un po' un problema al momento, non ho ancora visto come farlo.

2.2 Funzioni

Da vedere poi se rappresentare le funzioni come oggetti che implementano `LispCallable` comunque se lo rappresenti con un `LispCallable` poi vedi se vuoi mettere il lexenv (che grazialcazzo che lo facciamo con scope lessicale) all'interno del callable o meno.

Vedere inoltre per quanto riguarda le funzioni builtin, si potrebbe avere un `isBuiltin` e un `callBuiltin` e magari quelli non implementano il callable, mentre lo potrebbero implementare funzioni user defined perchè onestamente re-implementare callable per

- `cons`
- `car`
- `cdr`

[1] per citare Paul Graham alla cazzo di cane più totale

- (probabile) `or`
- (probabile) `and`
- (probabile) `if`
- et cetera res

mi sembra pessimo design.

2.3 Runtime

Per quanto, almeno inizialmente, molte funzionalità di runtime saranno lasciate al runtime di java[2], sarà comunque necessario rappresentare in qualche modo lo stato della "lisp machine" embedded, con questo si intende l'ambiente di variabili/valori/funzioni presenti, e lo stato della computazione[3] in corso. Il runtime sarà inoltre utilizzabile come "handle" al sistema lisp interno per chiamare le funzionalità di questo all'interno dell'app java.

2.3.1 NON

- Non è previsto, per adesso, un meccanismo di multithreading.
- Non è previsto, per adesso, un meccanismo di gestione delle eccezioni. (poi ovvio che il codice java avrà delle eccezioni)
- Non è ancora deciso se implementare il runtime come singleton[4] o meno.

3 Verso una qualche idea di interfaccia

Lavorare verso l'implementare una qualche interfaccia prestabilita funziona spesso meglio dell lavorare verso un'idea generica di funzionamento, specie nell'ambito della programmazione a oggetti, (si vada a vedere ad esempio la sezione goals di gson)

[2] che ci cazzo si mette a scrivere garbage collector per 6 crediti
[3] daje che si esagera coi termini qui
[4] come con il rutime di java, che è unico all'interno dell'applicazione

3.1 Load, Eval, forse Apply

Il Runtime esporrà i seguenti metodi come pubblici

`LispObject load()` accetta un file/filepath e valuta(intepreta) il file/espressione data, modificando lo stato interno del runtime, aggiungendo ad esempio definizioni al vocabolario/stato interno del sistema, o modificando eventuali flag di errore (cosa che potrebbe o non potrebbe essere disgiunta dal modificare il vocabolario del sistema). Ritorna il valore dell'ultima form valutata nel toplevel del file

`LispObject eval()` valuta una form data, modificando eventualmente lo stato del sistema, ritorna il valore ritornato dalla valutazione della form, probabilmente si farà anche l'overload per accettare stringhe che poi internametne sarà un

```
import somepackage.Form;

public class LispRuntime {
    /* ... */
    public LispObject eval(LispForm lf) {
        /* ... */
    }
    public LispObject eval(String s) {
        this.eval(Form.parseFromString(s));
    }
    /* ... */
}
```

`LispObject apply(LispCallable lc, arglist)`[5] chiama il `LispCallable` (funzoine) `lc` con argomenti `arglist`, ritorna il valore ritornato dall'applicazione

3.1.1 Particolari per file/batch

Tutto coperto da `load()` per adesso, `eval()` se proprio si vuole esagerare

3.1.2 Particolari per REPL

Probabilmente invocato attraverso un `Runtime.repl()`, anche se facilmente spostabile a un qualche altro oggetto che comunicherà comunque col runtime, il repl agirà nel seguente modo

[5] da decidere se inserirlo o meno

read
- accetta caratteri finchè non si ha una form completa (bilanciamento (o assenza) di parentesi alla fine di un token)
- traduce i caratteri ricevuti in form per essere valutata

eval valuta la form letta dal passo `read`, la cosa può anche produrre side effect quali stampa di caratteri, apertura/creazione/modifica di file, et al

print stampa il valore ottenuto dalla valutazione

loop torna al passo `read` (a meno che la valutazoine della form non implichi anche l'arresto del loop)

4 Struttura dell'interpreter

L'interpreter ha il ruolo di passare da una rappresentazione testuale del programma all'aver fatto qualcosa, questo sarà diviso in due componenti la struttura sotto data si aspetta una qualche rappresentazione testuale, se questa poi arriva da una stringa[6] o dall'apertura di un file, cazzi altrui.

preprocessor si occupa[8] di rimuovere i commenti dal testo ricevuto, vedere poi se si vuole aggiungere un meccanismo di macro o meno.

parser traduce la rappresentazione testuale del programma in una lista a `cons`

evaluator esegue quanto rappresentato nella lista a `cons`

La rappresentazione interna è una lista a `cons`[9], che costituisce sia la struttura sia dati fondamentale, che la rappresentazione intemedia, del nostro codice. Questa rappresentazione è stata scelta in quanto questa struttura dati, e questa equivalenza codice/dati, costituiscono una precentuale anche troppo alta del dna di lisp.

[6] stile `python -c "print(max($1, $2))"` buttato lì in uno script perchè nessuno[7] sa usare l'algebra in bash

[7] me incluso

[8] per ora

[9] nome tra il lispone e il fru fru per dire un albero binario

4.1 Parser

l'ultima volta fare questo è stato un aborto perchè non avevo idea di cosa fosse un pda o una grammatica libera dal contesto. questa volta, se non riutilizzo il codice già fatto, essendo che so cos'è un pda o una grammatica libera dal contesto, questo sarà una psicosi post parto

4.2 Evaluator

la struttura dell'evaluator è gratuitamente "ispirata" a quelle presenti in The Lisp1.5 Programmer's Manual, nell'introduzoine al linguaggio e in Structure and Intepretation of Computer Programs[10], nel capitolo `4`, specie la sezione `4.1, The Metacircular Evaluator`. Queste ovviamente adattate a un ambiente più imperativo e object oriented, visto che tradurre direttamente il codice lisp in codice java porterebbe ad abominii contro la madre quel poveraccio di un Guy Steele.[11].

[10] da qui in poi anche detto `SICP`.
[11] se questo per oltraggi a lisp[12] o java[13], lo decida il lettore
[12] di cui ha scritto lo standard(almeno per common lisp e scheme)
[13] di cui ha scritto lo standard