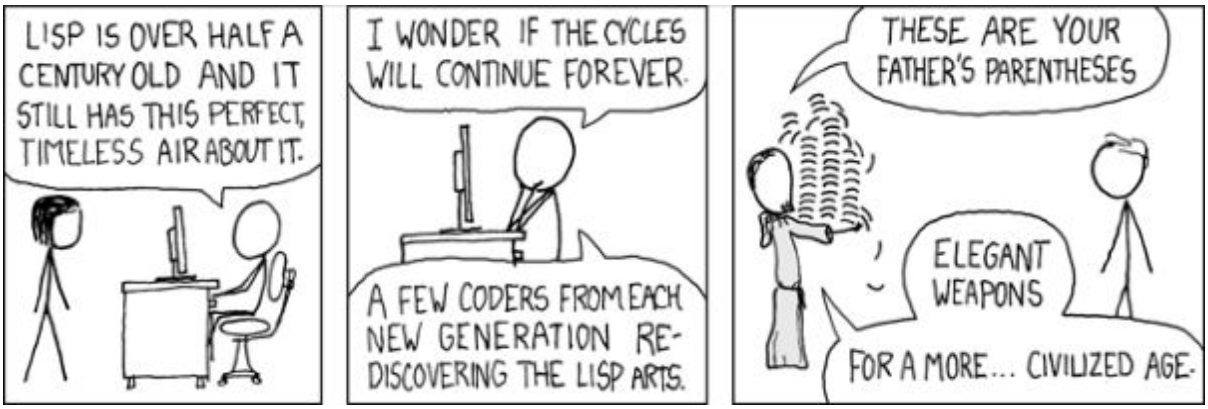
Notes sobre el llenguatge Lisp

(i sobre la implementació que heu de fer)



© XKCD Comic #297

El llenguatge Lisp (de LISt Processor) va ser inventat per John McCarthy a finals del 50s i és un dels llenguatges de programació més antics que encara s’utiltzen (bé, en alguna de les seves múltiples re-encarnacions modernes, com Clojure, CommonLisp o Racket).

Introducció

A diferència dels llenguatges de programació als que esteu acostumats, que estan orientats a instruccions (statements), el llenguatge Lisp està orientat a expressions.

Això vol dir que tota expressió que podem formar en el llenguatge, retorna sempre un valor. Simplificant, executar un programa en Lisp vol dir calcular el valor de l’expressió que forma aquest programa.

Un símil són les expressions matemàtiques a les que tant acostumats esteu:

```
(5 + 4) * (4 + 3)
```

és una expressió aritmètica i, seguint les regles de l’àlgebra dels nombres enters, el seu valor és `63`.

No tan sols sabem escriure expressions numèriques sinó que també som capaços d’avaluar expressions booleanes com per exemple:

```
(true ∧ false) ∨ false ∨ true
```

avalua a `true`.

La principal diferència entre les expressions de Lisp i les que estem acostumats és el lloc on col·loquem els operadors (o funcions) i com agrupem les coses entre parèntesis:

`(5 + 4)` s’escriu `(add 5 4)`

i, per tant;

`(5 + 4) * (4 + 3)` s’escriu `(mult (add 5 4) (add 4 3))`

Un avantatge d’aquesta notació és que no hem de recordar les regles de precedència dels operadors, ja que les expressions sempre estan parentitzades:

`5 + 4 * 3` s’escriu `(add 5 (mult 4 3))`

Un altre és que, a diferència dels operadors habituals que solen tenir un o dos paràmetres, en Lisp podem escriure expressions amb operadors que rebin tants paràmetres com volem, per exemple:

```
(add (mult 3 4) 5 6 (mult 7 8 9))
```

Expressions permeses

En el nostre cas només permetrem tres tipus d’expressions:

- Enters: 1, 2, -4, ...
- Símbols: a, add, if, ...
- Llistes: (), (1 a), (add 1 (mult 2 4)), ...

Com podeu veure les llistes són heterogènies, és a dir, no tots els elements de la llista han de ser del mateix tipus. A més, una llista pot contenir com elements altres llistes, que poden contenir altres llistes, ...

Per tant el tipus llista és eminentment recursiu i, per tant, de forma natural les funcions que les manipulen s’expressaran de forma natural de forma recursiva.

Lisp <3 Recursivitat

La sintaxi de Lisp no té res més, ja està explicada completament.

Avaluació d’una expressió

Abans hem dit que executar un programa en Lisp és avaluar l’expressió que el descriu. Per tant, ara que ja sabem escriure expressions en Lisp, és raonable que ens preguntem: com avaluem una expressió en Lisp? quines regles hem de seguir?

Anem a veure uns quants exemples que ens ajudaran a entendre-ho:

Enters

Quin valor us imagineu que té l’expressió 1 en Lisp? Quin hauria de ser el seu valor?

Si us digués que 42 pensaríeu que no té massa sentit.

El valor de l’expressió 1 hauria de ser 1. I així ho és.

El valor d’un número enter és ell mateix. Fàcil, oi?

Símbols

En matemàtiques esteu acostumats a coses com:

Sent `a = 2` i `b = 4`, el valor de a^2+b^2 és `20`.

i ens sembla normal fer servir els noms a i b per a referir-nos, en aquest cas, als nombres enters 2 i 4.

En Lisp passa el mateix: el valor d’un símbol és el valor al qual es refereix.

Més endavant veurem:

- com associar a un símbol un valor (l’equivalent de a = 2)
- alguns símbols estaran predefinits (add està predefinit com la funció suma)

Llistes

L’avaluació de les llistes és la part més complicada, ja que en Lisp una llista s’avalua com l’aplicació d’una funció (o una forma especial, com veurem més endavant) a una sèrie d’arguments.

Per exemple, considereu la expressió:

```
(add 2 3)
```

L’avaluació succeeix de la següent manera (fixeu-vos que l’avaluació també és un procés recursiu):

- S’avalua el primer element de la llista
    - Com és un símbol, el seu valor és el valor que té associat, en aquest cas, la funció suma
- Com el primer símbol és una funció, cal avaluar la resta d’elements (que formaran la llista d’arguments de la funció cridada)
    - El primer argument és un enter i s’avalua a sí mateix, en aquest cas 2
    - El segon argument també és un enter i s’avalua també a sí mateix, en aquest cas 3
- S’aplica la funció suma a la llista d’arguments `(2 3)`, obtenint un altre enter que és 5

Fins ara hem vist només exemples amb funcions que treballen amb números, però en Lisp tenim també funcions que treballen amb llistes. Anem a veure un exemple.

La funció car (es diu així per motius històrics), donada una llista amb almenys un element, retorna el primer element de la llista.

Seria vàlida aquesta expressió per a calcular el primer element de la llista (1 2 3)?

```
(car (1 2 3))
```

Anem a veure com funcionaria l’avaluació:

- Avaluem el primer element de la llista

    - Com és un símbol, el seu valor és el valor que té associat, en aquest cas, la funció que calcula el primer element
- Com el primer símbol representa una funció, caldrà avaluar la resta d’elements (que formaran la llista d’arguments de la funció cridada)
    - El primer element és una llista i, per tant hem de seguir el mecanisme d’avaluació de les llistes
        - S’avalua el primer element que és un enter
        - Però, podem aplicar un enter a una llista d’arguments????

Quin és el problema?

El problema és que Lisp interpreta la llista com una crida a una funció, no com la llista en sí mateixa.

Si volem poder fer servir llistes en les nostres expressions haurem de poder indicar al Lisp: *ep!, això és una llista (o una expressió) que no vull que avaluïs, sinó que la tractis literalment*.

És aquí on apareix la necessitat de definir elements amb “mecanismes especials d’avaluació” que no segueixin la regla general de les llistes (que és aplicar una funció a una llista d’arguments). A aquests elements se’ls anomena formes especials.

En aquest cas concret, la forma especial s’anomena quote i la seva avaluació simplement retorna de forma literal l’argument que se li passa.

Per tant, la forma correcta de l’expressió que calcula el primer element de la llista `(1 2 3)` és:

```
(car (quote (1 2 3)))
```

Anem a veure, de nou, com s’avalua:

- Avaluem el primer element de la llista
    - És el símbol car i avalua a la funció que calcula el primer element
- Com el primer símbol representa una funció, caldrà avaluar la resta d’elements (que formaran la llista d’arguments de la funció cridada)
    - El primer element és una llista i, per tant hem de seguir el mecanisme d’avaluació de les llistes
        i. S’avalua el primer element, que és un símbol i el seu valor és la forma especial que tracta una expressió com un literal
    - Per tant el valor és la llista (sense avaluar) que se li passa com argument, és a dir, la llista (1 2 3)
- S’aplica la funció que calcula el primer element sobre la llista a la llista de paràmetres ((1 2 3)) i el resultat l’enter 1.

A més de quote n’hi ha d’altres formes especials que ja s’explicaran més endavant.

Com l’ús de quote és molt habitual en Lisp, la part de l’intèrpret que llegeix les expressions i les analitza (que us donem feta) permet la substitució de

```
(quote (1 2 3))
```

per la utilització del caràcter *cometa simple* i, per tant, podem escriure:

```
‘(1 2 3)
```

Aquest mecanisme, anomenat tècnicament *macro de lectura*, és completament transparent per l’intèrpret: en tots dos casos l’intèrpret rep l’expressió `(quote (1 2 3))` per avaluar.

Elements predefinits

Quan s’implementa un intèrpret de Lisp, a més de les estructures de dades bàsiques, de les que parlarem més endavant, caldrà crear un conjunt mínim d’elements que seran utilitzats per poder crear-ne de nous.

A més, com veurem, part d’aquests elements predefinits també s’utilitzaran dins dels propi intèrpret.

Símbols

Existeixen dos símbols predefinits

- `t`
  - el seu valor és ell mateix
  - representa el valor lògic true
- `nil`
  - el seu valor és ell mateix
  - representa el valor lògic fals
  - representa la llista buida

Funcions

En el Lisp simplificat que veurem, n’hi haurà unes quantes funcions predefinides. Recordeu que a l’avaluar llistes que les contenen com a primer element, s’han d’avaluar la resta d’elements de la llista per a saber la llista d’arguments.

Un punt importat a destacar és que cap funció modifica el valor dels seus arguments.

Totes les funcions llencen l’excepció `EvaluationError` si no reben el nombre d’arguments correcte o bé si incompleixen alguna de les condicions addicionals descrites per cadascuna d’elles:

- `(add arg1 arg2 ....)`
  - retorna la suma de tots els enters que se li passen[1]

[1] Recordeu: els arguments són expressions que s’avaluen i és el seu valor els que es considera per a calcular, en aquest cas, la suma.

        - (add) avalua a 0
        - (add 1) avalua a 1
    - si algun dels arguments no avalua a un enter és un error
- `(mult arg1 arg2 ...)`
    - retorna el producte de tots els enters que se li passen
        - (mult) avalua a 1
        - (mult 2) avalua a 2
    - si algun dels arguments no avalua a un enter és un error
- `(car arg)`
    - retorna el primer element de la llista que se li passa com argument
    - si l’argument no és una llista amb almenys un element és un error
        - `(car (quote (1 2 3)))` avalua a 1
- `(cdr arg)`
    - retorna la llista amb la resta (tots menys el primer) d’elements de la llista que se li passa com argument
    - si l’argument no és una llista amb almenys un element és un error
        - `(cdr (quote (1 2 3)))` avalua a (2 3)
        - `(cdr (quote (1)))` avalua a nil
- `(cons arg1 arg2)`
    - retorna una llista que té com a primer element el primer argument i com a resta d’elements el segon argument
    - si el segon argument no és una llista o nil és un error[2]
        - `(cons 1 (quote (2 3)))` avalua a `(1 2 3)`
        - `(cons 1 nil)` avalua a `(1)`
        - `(cons (quote (1 2)) (quote (3 4)))` avalua a `((1 2) 3 4)`
- `(list arg1 arg2 ...)`
    - retorna la llista formada pels arguments que se li passen
        - `(list)` avalua a `nil`
        - `(list 1 2 3)` avalua a `(1 2 3)`
        - `(list 1 (cons 2 (list)) 3)` avalua a `(1 (2) 3)`
- `(eq arg1 arg2)`
    - retorna t si els dos arguments són iguals, nil altrament
        - `(eq (quote (2 3)) (cons 2 (cons 3 nil)))` avalua a t
        - `(eq 1 nil)` avalua a nil
- `(eval expr)`
    - retorna l’avaluació de l’expressió que se li passa com únic argument
        - `(eval (quote (add 1 2 3)))` avalua a 6
        - `(eval (quote (if (eq 0 1) 2 3)))` avalua a 3
- `(apply fun args)`
    - retorna el resultat d’aplicad la funció fun a la llista d’arguments args. Si fun no es una funció o bé si args no és una llista es tracta d’un error.
        - `(apply add (quote (1 2 3)))` avalua a 6
        - `(apply (lambda (n) (add n 1)) (quote (2)))` avalua a 3

[2] En el nostre Lisp simplificat considerarem només el que s’anomenen llistes pròpies.

Formes especials

Cada forma especial ha d’indicar com s’avaluen (o no) els diferents arguments que rep.

Totes les formes especials llencen l’excepció `EvaluationError` si no reben el nombre d’arguments correcte o bé si incompleixen alguna de les condicions addicionals descrites per cadascuna d’elles:

- `(quote arg)`
  - Retorna arg sense avaluar
  - Si se li passa més d’un argument és un error
    - (quote (1 2 3)) avalua a (1 2 3)
    - (quote 2) avalua a 2
    - (quote (quote 2)) avalua a (quote 2)
- `(if cond-expr then-expr else-expr)`
  - Avalua primer l’expressió cond-expr
    - Si el valor no és nil, el valor retornat és el d’avaluar then-expr
    - Altrament el valor retornat és el d’avaluar else-expr
  - Fixeu-vos que, com no pot ser de cap altra manera, només s’avalua una des les dues subexpressions then-expr o else-expr
  - Si se li pasa un nombre d’arguments diferent de 3 és un error
    - (if (eq 1 2) 3 4) retorna 4
    - (if 1 2 3) retorna 2
- `(define sym expr)`
  - Vincula globalment[3] el símbol sym (que no s’avalua) amb el valor obtingut d’avaluar expr
  - Obligatòriament té dos arguments i el primer d’ells ha de ser un símbol
  - Retorna nil
    - (define a (add 1 2)) vincula globalment a amb 3 i retorna nil de manera que si ara avaluem (add 1 a) el resultat serà 4.
- `(lambda params body)`
  - Retorna una funció anònima que té com a paràmetres la llista params i com a cos l’expressió body.
  - Obligatòriament té dos paràmetres, el primer dels quals ha de ser una llista de símbols.
  - Quan es “cridi” a la funció, els arguments que es passin s’hauran de vincular a aquests paràmetres
  - El resultat de la crida a la funció serà el d’avaluar body dins de l’entorn que conté la vinculació dels símbols de la llista params amb els valors dels arguments passats a la crida.

Com això de la definició i avaluació de funcions i formes especials és una mica complex, li dedicarem la propera subsecció.

[3] Més endavant veurem que en Lisp hi ha símbols locals a una funció, els seus paràmetres, i d’altres que són globals.

Definició i ús de noves funcions i formes especials

Anem a veure uns quants exemples de definició de funcions.

Molt sovint, de cara a després poder usar la funció, definirem una funció usant lambda dins d’una expressió define que li doni un nom.

Per exemple, anem a definir una funció tal que, donat un número, l’incrementa en 1:

```
(define succ (lambda (n) (add n 1)))
```

Què podem fer ara? Podem usar la nova funció.

Com? Cridant-la.

```
(succ 4)
```

Com s’avalua (succ 4)?

- És una llista, per tant hem d’avaluar el seu primer element
- succ és un àtom i el valor que té vinculat és el d’una funció
    - Hem d’avaluar els arguments, que en aquest cas és 4, per tant la llista d’arguments és (4)
- Hem d’aplicar la funció que té una llista de paràmetres (n) i un cos (add n 1) a la llista d’arguments (4) i una referència a l’entorn d’avaluació on es va crear la lambda que, en aquest cas, és l’entorn global:
    - hem de crear un nou entorn d’avaluació
        - partint de l’entorn que hi havia quan es va crear la lambda[4]
        - estenent-lo amb els vincles provinents del pas de paràmetres, que en aquest cas, és n vinculat a 4
    - hem d’avaluar (add n 1) en aquest nou entorn d’avaluació. En aquest cas, l’avaluació és 5.

Per exemple també podem definir funcions recursives com el factorial:

```
(define factorial
      (lambda (n)
            (if (eq n 0)
                1
                (mult n (factorial (add n -1))))))
```

Podem usar funcions sense necessitat de donar-lis un nom.

Per exemple, és perfectament vàlid fer:

```
((lambda (l) (car (cdr l))) (quote (1 2 3)))
```

[4] Aquesta combinació rep el nom tècnic de clausura (en anglès, closure).

que avalua a 2.

Estructures de dades

Anem a veure les estructures de dades que haureu d’implementar.

Les dividirem en dues parts:

- les que representen expressions (i, per tant, valors)
- i les que representen entorns d’avaluació.

SExpressions

Si repassem els valors que podrem manegar, seran els següents:

- enters
- símbols
- llistes
- funcions
- formes especials

i el nostre intèrpret de Lisp haurà de ser capaç de representar-los i manipular-los de forma convenient.

El tipus que representa a qualsevol d’aquestos valors s’anomena SExpression (s-expression o symbolic expression).

La forma més natural de representar-ho en Java seria:

```
public interface SExpression {
      SExpression eval(Environment env);
}
```

Totes les classes que implementin SExpression seran **immutables** i, a més dels mètodes definits a SExpression, implementaran de forma correcta:

- equals
- hashCode
- toString

El mètode eval és el que, donada una expressió, troba el seu valor.

Per a fer-ho requereix del paràmetre env, que és l’anomenat entorn d’avaluació i que serveix per guardar els valors que tenen els símbols (p.e. per saber que add està lligat a la funció que suma, que nil val nil, etc, etc). Parlarem més endavant dels entorns d’avaluació.

Enters

Podem definir els enters com una classe que guarda un valor enter que se li passa en el constructor.

El marquem com a final per a que mai no es pugui modificar i, sent així, podem estalviar-nos el fet de crear un getter per ell definint-lo com a public[5].

```
public class Integer implements SExpression {
      public final int value;
      ....
}
```

El mètode toString retorna el valor de l’enter en forma de String.

Símbols

En aquest cas el que guardem és el nom, de tipus String que se’ns passa al constructor.

```
public class Symbol implements  SExpression {

      public static final NIL = new Symbol(“nil”);

      public final String name;
            ...
}
```

Com el símbol nil s’utilitza a molts llocs, l’hem definit com una constant i així podrem escriure Symbol.NIL quan el necessitem.

El mètode toString retorna el nom del símbol.

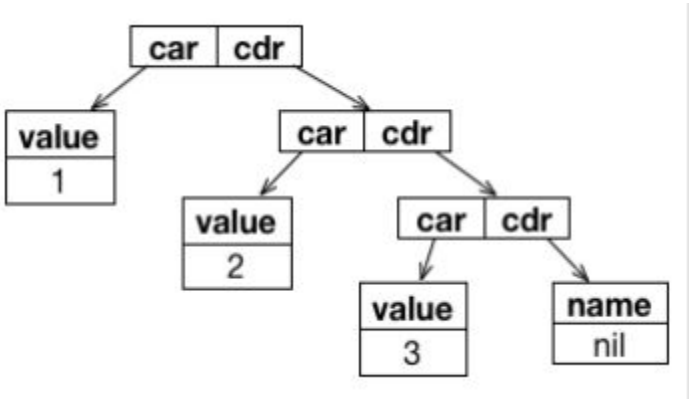
ConsCells

Són els elements que ens permeten construir llistes. Primer posem la seva definició:

```
public class ConsCell implements SExpression {
      public final SExpression car;
      public final SExpression cdr;
      ...
}
```

Amb aquesta representació, la llista (1 2 3) vindrà representada per:

[5] Si us molesta que sigui public el definiu com a privat i implementeu els getters convenients.



On 1, 2 i 3 representen nodes de tipus Integer amb els valors respectius, i nil representa un node de tipus Symbol amb nom nil.

El mètode toString escriu els elements de la llista (i els parèntesis inicial i finals). En el cas mostrat, el toString retornaria “(1 2 3)”.

Recordeu, la llista buida està representada per nil, no per una estructura de cel·les.

Function i Lambda

Totes les funcions avaluen a sí mateixes.

El que diferenciarà una funció de les altres és com s’aplica als valors que li passen com arguments.

Per això la classe Function es defineix com abstracta de la forma següent:

```
public abstract class Function implements SExpression {

      @Override
      public SExpression eval(Environment env) { ??? }

      public abstract SExpression apply(SExpression evargs,
                                        Environment env);

      @Override
      public String toString() {
            return String.format("<function-%x>", hashCode());
      }
}
```

Una qüestió tècnica és que per les funcions, el mètode equals i hashCode que heretem de la classe Object ja ens van bé i, per tant, no els modifiquem.
El toString de Function retorna “<function-23643>” on el número és el hashCode de l’objecte.

En el cas de les funcions el mètode apply rep la llista amb els arguments avaluats.

- En el cas d’una funció predefinida, aquest codi estarà a la implementació del mètode apply de la subclasse de Function que la implementa.
- En el cas d’una funció definida en Lisp (usant lambda) podeu definir una subclasse de Function anomenada Lambda que implementi, de forma genèrica, aquesta avaluació.
- Una instància de Lambda haurà de guardar:
    - la llista de paràmetres
    - la llista d’expressions del seu cos
    - una referència a l’entorn d’avaluació on va ser creada. Aquest entorn d’avaluació servirà com a partida, prèvia extensió amb el pas de paràmetres, per crear l’entorn on avaluar el cos de la funció.

Aquest mecanisme d’extensió d’entorns d’avaluació permet coses com:

```
(define incrementor
      (lambda (x)
            (lambda (y)
                  (add x y))))

(define add3 (incrementor 3))

(add3 24)
```

que avalua a 27.

Special

Les formes especials també avaluen a sí mateixes.

El que diferenciarà a una funció de les altres és com s’aplica a les expressions que li passen com arguments.

En el cas de les formes especials, el mètode `applySpecial`, rebrà la llista dels arguments sense avaluar (ja que cada forma especial ho farà de forma diferent).

Per les mateixes raons que abans, els mètodes equals i hashCode que heretem de la classe Object ja ens van bé i, per tant, no els modifiquem.

El toString de Special retorna “<special-23643>” on el número és el hashCode de l’objecte.

Per tant, de forma similar a la classe Function, podem definir la classe Special com:

```
public abstract class Special implements SExpression {
      @Override
```

```
        public SExpression eval(Environment env) { ??? }

        public abstract SExpression applySpecial(SExpression args,
                                                 Environment env);

        @Override
        public String toString() {
                    return String.format("<special-%x>", hashCode());
        }
}
```

Entorns d’avaluació

Aquests entorns guarden les associacions entre els símbols i els valors al que es refereixen.

Serveixen tant per a guardar les associacions globals (entre elles les dels elements predefinits del llenguatge) com la dels entorns locals (que es creen al cridar a les funcions lambda).

Els entorns d’avaluació tenen la següent interfície:

```
public interface Environment {
      SExpression find(Symbol symbol);
      void bindGlobal(Symbol symbol, SExpression value);
      void bind(Symbol symbol, SExpression value);
      Environment extend();
}
```

El que haureu de fer és crear una implementació d’aquesta interfície usant HashMap, és a dir:

```
public class NestedMap implements Environment {
      ....
}
```

Com funcionen aquests entorns?

La forma més fàcil d’entendre-ho és amb un exemple[6]:

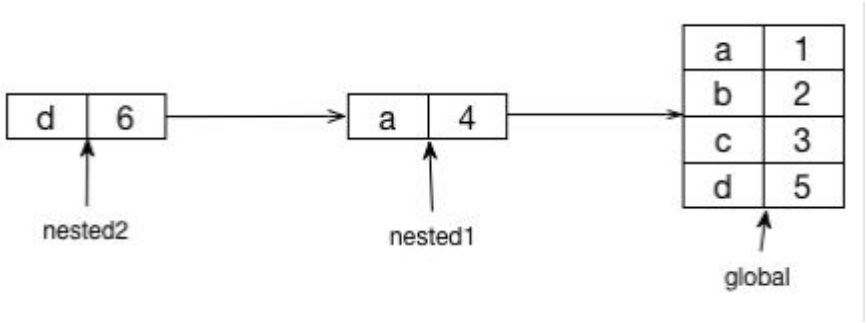
```
Environment global = new NestedMap();
global.bindGlobal(“a”, 1);
global.bind(“b”, 2);
Environment nested1 = global.extend();
nested1.bindGlobal(“c”, 3);
```

[6] Per simplificar, faré servir cadenes i enters directament, però haurien de ser símbols i expressions.

```
nested1.bind(“a”, 4);
Environment nested2 = nested1.extend();
nested2.bindGlobal(“d”, 5);
nested2.bind(“d”, 6);

global.find(“a”) -> 1
nested1.find(“a”) -> 4
nested2.find(“a”) -> 4
nested1.find(“d”) -> 5
```

En forma de diagrama:



Si busquem el valor associat a un símbol que no té un valor associat, es llença l’excepció EvaluationError (que, com l’hem definit com a no comprovada, no cal declarar-la a la capçalera del mètode).

Inicialment només necessitareu les implementacions de bindGlobal i find i, quan pugueu definir funcions en Lisp (lambda), necessitareu implementar també bind i extend.

La classe ListOps

Pot ser convenient tenir una classe d’utilitat que només contingui una sèrie de mètodes estàtics que poden ser d’interès, o simplificar, d’altres parts del codi.

Per exemple, aquesta classe conté la següent funció:

```
public static SExpression cons(SExpression car, SExpression cdr) {
            return new ConsCell(car, cdr);
}

public static SExpression car(SExpression sexpr) {
            return ((ConsCell) sexpr).car;
}

public static SExpression cdr(SExpression sexpr) {
            return ((ConsCell) sexpr).cdr;
}
```

D’aquesta manera el codi en java quedarà molt més clar.

Una altra funció que pot ser útil és la que calcula la mida d’una llista:

```
public static int length(SExpression sexpr) { ??? }
```

o una que us retorni l’element que ocupa la pos dins de la llista:

```
public static SExpression nth(SExpression sexpr, int pos) { ??? }
```

Per exemple, aquestes funcions (entre d’altres) poden ser molt útils per comprovar, quan implementeu les funcions i formes especials predefinit, el nombre de paràmetres que es passen i accedir a cadascun d’ells.

Totes aquestes funcions assumeixen que es criden amb les expressions adequades (p.e. no intentem calcular la mida d’una SExpression que és un Integer).

- Com disposem de prou tests com per a comprovar que l’ús **intern** que farem és el correcte no és massa arriscat fer-ho així.
- Les expressions que construïm usant les funcions predefinides (p.e. cons), ja comprovaran que el resultat sigui una llista ben construïda.
- O bé, si volem tenir més seguretat, podem afegir-hi la comprovació d’aquestes condicions a la implementació i llençar EvaluationError en cas de que la crida no les satisfaci.

Analitzador

És la part de l’intèrpret que transforma la cadena llegida en una SExpression.

Us el donem ja fet i el seu ús el podem veure al codi del repl:

```
try {
     Parser parser = new Parser(new StringLexer(input));
     SExpression sexpr = parser.sexpr();
     SExpression result = sexpr.eval(environment);
     System.out.printf(">>>>>>> %s%n", result);
} catch (LexerError | ParserError | EvaluationError ex) {
     System.out.printf("~~~~~~~ %s%n", ex.getMessage());
}
```

Errors

Les classes d’error us les donem fetes. Per simplificar, tots els errors es defineixen com a `RuntimeError` i, per tant, no cal declarar les excepcions a les funcions que les poden llençar.

Obligatòriament heu de passar un paràmetre de tipus String al seu constructor per indicar el missatge d’error.

N’hi ha tres classes d’excepció:

- `LexerError`: es llença quan l’analitzador lèxic ha trobat algun error.
- `ParserError`: es llença quan l’analitzador sintàctica ha trobat algun error.
- `EvaluationError`: l’heu de llençar quan a l’hora d’avaluar una excepció trobeu un error (p.e. si detecteu que algun dels arguments de la funció `add` no és un enter).

Codi que us donem

- Estructura de paquets
- Analitzador sencer
- REPL sencer (main)
- Esquelet de la classe Primitives on es definiran les primitives
- Totes les interfícies a completar
- Totes les classes de prova
- Esquelets mínims d’algunes classes (per a que el projecte es pugui compilar sense errors).
    - El codi de la classe llença UnsuportedOperationException i l’anireu substituïnt pel vostre codi a mesura que aneu completant les etapes.

Maven

Per tal de que pugueu usar l’entorn de desenvolupament que més us agradi, en comptes de crear un projecte Netbeans (amb el seu sotsdirectori nbproject ...), o Intellij (amb els seu sotsdirectori .idea, ...), el projecte està per configurar una eina anomenada maven.

Tant Netbeans[7] com Intellij són capaços de treballar directament amb un projecte de maven (sense necessitat d’instal·lar cap cosa addicional). La única cosa estranya que veureu és que al directori arrel hi ha un fitxer anomenat pom.xml, que conté la descripció del projecte.

El codi de projecte “penja de dos directoris”:

- Producció: `src/main/java/`
- Proves: `src/test/java/`

Quan vulgueu executar tots els tests del projecte, en Netbeans podeu fer: des del menú Run l’opció Test Project.

Si voleu executar el REPL: des del menú Run trieu l’opció Run Project (us sortirà un popup suggerint-vos la classe que conté el main) i a la finestar output teniu l’intèrpret.

[7] Provat amb Netbeans 8.1 i Intellij IDEA 15

També dins del menú Run teniu l’opció Build Project, la qual compilarà el projecte i, si el codi passa el joc de proves, crearà al sotsdirectori target un fitxer .jar amb la compilació del projecte. Des de la línia de comandes, podeu fer:

```
java -jar target/butterp-1.0-SNAPSHOT.jar
```

i s’executarà el REPL.

Una de les coses que fa que maven sigui molt útil és que gestiona les dependències del nostre projecte. Com veurem a la propera secció, de cara a fer les proves del projecte, necessitem la biblioteca JUnit. Doncs simplement declarant aquesta dependència al fitxer pom.xml, maven és capaç de descarregar-la i configurar-la dins del nostre projecte[8].

JUnit

Un projecte d’aquesta mida és gairebé impossible de fer sense tenir una infraestructura que ens permeti anar provant el codi a mesura que l’anem fent. En java hi ha una biblioteca anomenada JUnit que conté classes i mètodes que ens permeten escriure fàcilment proves del nostre codi.

A més, com és una biblioteca molt utilitzada, totes els entorns de desenvolupament, a més de l’eina maven de gestió del projecte, són capaces d’utilitzar-la.

A l’assignatura de tercer, Enginyeria del Programari, descrivim aquesta eina. Així que el que faré és adjuntar-vos la part dels apunts que la descriuen (veureu que és molt simple i, a més, teniu gairebé 100 exemples de com usar-la dins del codi que us hem proporcionat).

Part de la sessió de dilluns la dedicarem a explicar una mica aquesta biblioteca.

Seqüència de realització de la solució

A l’hora de fer la vostra solució **és obligatori** que completeu les etapes en l’ordre que es detalla a continuació i no passeu a l’etapa següent fins assegurar-vos (en la mesura del possible) de que el codi que heu afegit funciona correctament.

El joc de proves que se us proporciona us serà de molta d’ajuda.

Val a dir que un joc de proves, que es va anar fent a mesura que la solució del projecte anava creixent, i amb una seqüència de passes diferent, de vegades no pugui ajudar al 100%.

En aquests casos podeu definir testos addicionals que us ajudin, o si ja us funciona el REPL, fer servir l’intèrpret interactiu per provar.

1. class Integer

[8] La primera vegada que useu maven dóna la sensació de que es descarrega tot Internet. No patiu, internet és molt més gran.

    a. constructor
    b. getter (o accés públic al camp value que és final)
    c. equals
    d. hashCode
    e. toString
2. class Symbol
    a. constructor
    b. getter (o accés públic al camp name que és final)
    c. equals
    d. hashCode
    e. toString
3. class ConsCell
    a. constructor
    b. getters (o accés públic als camps car i cdr que són finals)
    c. equals
    d. hashCode
    e. toString
4. class NestedMaps
    a. bindGlobal
    b. find
5. class LIstOps
    a. cons
    b. car
    c. cdr
    d. list (dues versions)
    e. length
    f. nth
6. eval per Integer
7. eval per Symbol
8. class Function
9. predefined add (mireu la classe Primitives que és on es defineixen tots els elements predefinits)
10. eval per ConsCell
11. predefined mult
12. class Special
13. predefined define
14. predefined quote
15. predefined cons
16. predefined car
17. predefined cdr
18. predefined list
19. predefined eq
20. predefined if
21. class NestedMaps
    a. extend
    b. bind

    c. bindGlobal
22. predefined lambda
23. predefined eval
24. predefined apply

Comentaris:

- a partir de tenir LispOps, el codi de l’analitzador hauria de funcionar ja que depèn només de:
    - constructor de Symbol
    - constructor de Integer
    - ListOps.list(SExpression... elems)
    - ListOps.list(java.util.List<SExpression> elems)[9]
- el REPL depèn només de tenir
    - l’analitzador funcionant
    - que funcioni eval per les SExpressions que voleu avaluar
    - que funcioni toString pels resultats d’aquestes avaluacions
    - que estiguin els apply o applySpecial de les funcions o formes especials que esteu utilitzant.

A entregar

- El projecte que us donem ampliat amb el codi que heu realitzat.
- Ompliu el fitxer RESUM.md que hi trobareu a l’arrel del projecte amb la informació que es demana.

NOTA: al campus virtual (sakai) hem obert l’espai **Concurs Programació "Josep Mª Ribó"** on hi afegirem els equips inscrits, hi haurà tot el **material addicional**, podreu fer consultes **utiltzant l’eina debat**, ... , per tant, el **lliurament** el fareu com si d’una activitat d’una assignatura es tractés.

Bibliografia

- John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine”, Part I", *Communications of the ACM*, 1960.
- Hal Abelson's, Jerry Sussman's and Julie Sussman's. *"Structure and Interpretation of Computer Programs"* (MIT Press), 1984.
- Lisp (programming language) Article de la Wikipedia.
- John McCarthy Article de la Wikipedia.

[9] No us confongueu: el paràmetre no és una llista de Lisp sinó una llista de Java. Per remarcar això, he fet servir el nom completament qualificat de la interfície.