Week 12: Interpreting Lisp

We now discuss an interpreter for Lisp.

This is useful for two reasons.

1. Many computer systems have a small language inside, which often is interpreted.
2. The interpreter tells us in a precise way how Lisp programs in particular, and functional programs in general, are evaluated.

Interpreter Design

The interpreter takes as input an expression of type *Data*, which is represented by the data structure presented last week:

```
abstract class Data;
case class NUM(x: Int)                  extends Data;
case class STR(x: String)               extends Data;
case class SYM(name: String)            extends Data;
case class NIL                          extends Data;
case class CONS(car: Data, cdr: Data)   extends Data;
```

The interpreter returns another *Data* expression as output, which represents a Lisp value.

For instance, when applied to the input expression

```
CONS(SYM("*"), CONS(NUM(2), CONS(NUM(7), NIL)))
```

the interpreter should return the output expression

```
NUM(14)
```


But now a question poses itself: When the interpreter input is a symbol, such as *SYM("x")*, what should the output be?

It depends whether the name *x* is defined at the point of evaluating the symbol, and, if yes, to what value it is bound.

The interpreter needs to keep track of defined names in an *environment*.

Environments

An environment is a data structure that maps names to Lisp values.

The two fundamental operations on an environment are:

- *lookup*: Given a name, yield the value associated with the name.
- *extend*: Given a name/value binding, extend the environment with the binding, yielding a new environment.

This leads to the following class design:

```
class Environment with {
  def lookup(n: String): Data = ...
  def extend(name: String, v: Data) = ...
}
```

We also need to define a value *EmptyEnvironment*, which represents the empty environment.

Various data structures can be used for implementing environments, with different performance tradeoffs. (Examples?)

We use here a direct approach, without any auxiliary data structure.

```
abstract class Environment with {
  abstract def lookup(n: String): Data;
  def extend(name: String, v: Data) = new Environment with {
    def lookup(n: String): Data = if (n == name) v else outer.lookup(n);
  }
}
val EmptyEnvironment = new Environment with {
  def lookup(n: String): Data = error("undefined: " + n);
}
```

This is short and easy, but the *lookup* method takes time proportional to the number of bindings in an environment.

Predefined Functions

There also needs to be some way to interpret "standard" symbols such as $*$.

We can use an initial environment to record the values of such symbols.

But what should be the value of $*$?

We need to create a new subclass of *Data*, which represents for functions on *Lisp* expressions:

```
case class FUN(f: (List[Data])Data) extends Data;
```

For instance, the $*$ operation would be bound to the value:

```
FUN(case [NUM(arg1), NUM(arg2)] => NUM(arg1 * arg2))
```


Aside: the example shows the power of pattern matching – in a single expression, we can specify that

- the argument list must have length two, and that
- both arguments must be instances of *NUM*,
- and we make the elements of both arguments accessible under names *arg1* and *arg2*.

Note that a ***case*** expression is itself a function, which matches *FUN*'s argument type *(List[Data])Data*.

The Evaluator

We now present the Lisp evaluator.

It takes as input internal presentations of Lisp expressions. These are:

- Atoms: symbols, numbers, strings.
- Combinations, which themselves can be special forms or applications.

We simplify Lisp's special form for name bindings (e.g. *define*, **let**, *letrec*, *lambda*, ...) to just ***val*** (with the Scala meaning) and *lambda*.

So, instead of Lisp's

```
(define (add m n) (+ m n)) <rest of code>
```

we would write

```
(val add (lambda (m n)(+ m n)) <rest of code>)
```

Later on, we will add to this Scala's ***def***, but only for parameterless functions (since there is already *lambda* to express parameterization).

The Evaluator Function

```
def eval(x: Data, env: Environment): Data = x match {
  case NUM(_), STR(_), FUN(_) => x
  case SYM(name) =>
    env lookup name
  case CONS(SYM("val"), CONS(SYM(name),
                        CONS(body, CONS(rest, NIL)))) =>
    eval(rest, env.extend(name, eval(body, env)))
  case CONS(SYM("if"), CONS(cond,
                        CONS(thenpart, CONS(elsepart, NIL)))) =>
    if (toBoolean(eval(cond, env))) eval(thenpart, env)
    else eval(elsepart, env)
  case CONS(SYM("quote"), CONS(expr, NIL)) =>
    expr
  case CONS(SYM("lambda"), CONS(params, CONS(body, NIL))) =>
    mkLambda(params, body, env)
  case CONS(operator, operands) =>
    apply(eval(operator, env), toList(operands) map (x => eval(x, env))) }
```


Explanations

- If the input expression *x* is a number, string or function, we return the expression itself.
- Otherwise, if the expression is a symbol, we return the result of looking up the symbol in the current environment.
- Otherwise, if the expression is a special form

  (**val** <name> <body> <rest>)

  we evaluate the <rest> expression in an environment extended by the binding of <name> to the result of evaluating <body>.
- Otherwise, if the expression is a special form

  (**if** <cond> <thenpart> <elsepart>)

  we evaluate <cond>. If the result is the number *0*, we continue with the evaluation of <thenpart>; otherwise with the evaluation of <elsepart>. The test uses the function *toBoolean*, which is defined as follows.


```
def toBoolean(x: Data) = x match {
  case NUM(0) => False  case _ => True
}
```

- Otherwise, if the expression is a special form

  (quote <expr>)

  we return *expr* as the result of the evaluation.
- Otherwise, if the expression is a special form

  (lambda <params> expr)

  we create a new function using the *mkLambda* operation (see below).
- Finally, if the expression is a combination which is none of the special forms above, it must be a function application.

  In this case, we evaluate the application's operator and all its operands, and then *apply* the operator value to the operand values.

The *apply* function checks that the operator is indeed a *FUN* node and then applies its associated operation to the argument lists.

```
def apply(fn: Data, args: List[Data]): Data = fn match {
  case FUN(f) => f(args);
  case _ => error("application of non-function: " + fn);
}
```

The *toList* function converts a Lisp list which is built from *NIL* and *CONS* to a Scala list. It is defined as follows.

```
def toList(x: Data): List[Data] = x match {
  case NIL => []
  case CONS(y, ys) => y :: toList(ys)
  case _ => error("malformed list: " + x);
}
```

That's all, except that we still need to define how a *LAMBDA* expression gets converted to a function that can be applied.

Environments for Lambdas

Note that a *mkLambda* operation keeps a reference to the environment that was current when the function was built.

It then uses this environment in an application.

Here is an example of the evolution of the environments during a computation of the program

```
def f(x: Int) = g(y ⇒ x + y)
def g(x: (Int)Int) = x(2);
f(1)
```

(see blackboard).

It remains to define the function *extendEnv* which extends the environment by a list of bindings of formal parameter names to argument values.

This function is defined as follows.

```
def extendEnv(env: Environment,
                    ps: List[String], args: List[Data]): Environment =
  (ps, args) match {
    case ([], []) ⇒
      env
    case (p :: ps1, arg :: args1) ⇒
      extendEnv(env.extend(p, arg), ps1, args1)
    case _ ⇒
      error("wrong number of arguments")
  }
```

The function checks that the two lists of parameter names and argument values have the same length.

For each corresponding name/value pair, it applies the environment's *extend* method.

- That is, an application of the function causes the *body* expression to be evaluated in an environment which is extended by bindings that map formal parameter names to corresponding actual argument values.
- Note the analogy to the substitution model, where formal parameters where *replaced* in *body* by actual arguments *args*.
- We can think of an environment as a deferred substitution: Rather than doing the replacement immediately, we perform it when a symbol is looked up in the environment.
- It would be also possible to write an interpreter with substitutions instead of environments; but this interpreter would be more complicated and less efficient.

Making Functions

The conversion from Lisp expression to Scala function uses the auxiliary function *mkLambda*.

The call *mkLambda(params, body, env)* yields a *FUN* object with a (Scala-)function.

That function maps argument lists that correspond to the formal parameter list *params* to the result of evaluating *body* in *env*.

It works as follows:

```
def mkLambda(params: Data, body: Data, env: Environment): Data = {
  val ps: List[String] = toList(params) map {
    case SYM(name) ⇒ name
    case _ ⇒ error("illegal parameter list");
  }
  FUN(args ⇒ eval(body, extendEnv(env, ps, args)))
}
```

Dynamic Scoping in Original Lisp

The original version of Lisp used instead a global stack for environments.

When entering a function, bindings for parameters and local variables were pushed.

When leaving a function, the stack was cleared again.

But this makes higher-order function behave very strangely!

For instance, the result of evaluating $f(1)$ in the program above would result in a run-time error, because the anonymous function $y \Rightarrow x + y$ accesses the latest value of x on the stack, which is a function, not a value.

This problem with this particular program could be solved by renaming so that all parameter names are unique.

For instance, if we rename to:

```
def f(x: Int) = g(y ⇒ x + y)
def g(z: (Int)Int) = z(2);
f(1)
```

the program returns 3, as expected.

But the following program shows that one cannot always avoid problems by renaming.

```
def iterate(n: Int, f: (Int)Int) =
  if (n == 0) (x ⇒ x)
  else nTimes(n−1, (x ⇒ f(x)))
```

That function is supposed compose its parameter f n-times with itself.

Question: What is the effect of evaluating this function with original Lisp's stack-environment?

Another problem with original Lisp's scheme is that it is very dangerous to return functions. Consider:

```
def incrementer(x: Int) = y ⇒ y + x
```

Question: What is the effect of *incrementer*(2)(3)?

The implementation scheme of original Lisp is called dynamic scoping.

This means that the association of an identifier with its value is determined dynamically – it depends on the shape of the environment at the time the value is referenced.

Dynamic scoping offers interesting possibilities, but it severely impairs the usability of higher-order functions.

Therefore, more modern versions of Lisp such as Scheme have static scoping, just like Scala.

Older versions, including elisp still have dynamic scoping.

Common Lisp, of course, has both.

Handling Recursion

We have seen that original Lisp's environment scheme ran into problems.

But ours does as well, when we add recursion!

Recursion is introduced by adding to our interpreter a special form (***def*** *<name> <expr> <rest>*).

def is like ***val*** except that it defines a function *<name>* which can call itself recursively.

Recursion by Self Application

This technique demonstrates a deep connection in programming: recursion can be modelled by self-application.

In fact, all recursion is ultimately treated as self-application in lambda calculus, the theory underlying functional programming.

To demonstrate, without further explanations, here is a version of *faculty* that uses neither recursion nor loops!

```
(lambda (n)
  ((lambda (fact)
      (fact fact n))
    (lambda (ft k)
      (if (= k 1)
          1
          (* k (ft ft (- k 1)))))))
```


Environments Refined

Here is the new definition of environments.

```
abstract class Environment with {
  abstract def lookup(n: String): Data;
  def extendRec(name: String, expr: (Environment)Data) =
    new Environment with {
      def lookup(n: String): Data =
        if (n == name) expr(this) else outer.lookup(n);
  }
  def extend(name: String, v: Data) = extendRec(name, (env1 => v));
}
```

Note that *extendRec* is now the principal operation for extending an environment; *extend* is defined in terms of it.

Note also that *lookup* enables recursion by passing the current environment to the function that was found.

Evaluation of *def*

Here is how ***def*** can then be treated:

```
case CONS(SYM("def"), CONS(SYM(name),
                 CONS(body, CONS(rest, NIL)))) =>
  eval(rest, env.extendRec(name, (env1 => eval(body, env1))))
```

- This solves the problem of premature evaluation, because we now extend the environment with a function that evaluates *body*, not the result of the evaluation of *body*.
- Furthermore, we make recursion possible by changing the *lookup* of an environment.

The method for handling ***val*** in the interpreter,

```
case CONS(SYM("val"), CONS(SYM(name),
                 CONS(body, CONS(rest, NIL)))) =>
  eval(rest, env.extend(name, eval(body, env)))
```

does not work for ***def***, for two reasons:

- Body is evaluated too early; it should be evaluated only when *name* is accessed.
- *name* does not form part of the environment visible to *body*, so recursion is impossible.

We solve these problems by adding a new method *extendRec* which extends an environment with a binding containing a possible recursive computation.

Using Lisp from Scala

Here's a usage scenario how we could use the Scala interpreter to evaluate Lisp:

First, define a Lisp function as a string:

```
> def facultyDef =
      "def faculty (lambda (n)" +
      "  (if (= n 0)" +
      "      1" +
      "      (* n (faculty (- n 1)))))"
```

This function could then be applied as follows:

```
> evaluate("(" + facultyDef + "(faculty 4))")
24
```


The Top-level Interpreter Function

Here is the main function of the interpreter:

```
def evaluate(x: Data): Data = eval(x, globalEnv);
```

It evaluates a Lisp program in the global environment and returns the resulting data object.

To make usage easier for the programmer who types lisp expressions on the command line, we also include another version of *evaluate*, which takes Lisp expressions as strings and returns them as strings.

```
def evaluate(s: String): String = evaluate(string2lisp(s)).asString;
```



```
val globalEnv = EmptyEnvironment
  .extend("=",     FUN(case [NUM(arg1), NUM(arg2)] ⇒
                          NUM(if (arg1 == arg2) 1 else 0)
                        case [STR(arg1), STR(arg2)] ⇒
                          NUM(if (arg1 == arg2) 1 else 0)))
  .extend("+",     FUN(case [NUM(arg1), NUM(arg2)] ⇒
                          NUM(arg1 + arg2)
                        case [STR(arg1), STR(arg2)] ⇒
                          STR(arg1 + arg2)))
  .extend("-",     FUN(case [NUM(arg1), NUM(arg2)] ⇒
                          NUM(arg1 - arg2)))
  .extend("*",     FUN(case [NUM(arg1), NUM(arg2)] ⇒
                          NUM(arg1 * arg2)))
  .extend("/",     FUN(case [NUM(arg1), NUM(arg2)] ⇒
                          NUM(arg1 / arg2)))
  .extend("car",   FUN(case [CONS(x, xs)] ⇒ x))
  .extend("cdr",   FUN(case [CONS(x, xs)] ⇒ xs))
  .extend("null?", FUN(case [NIL] ⇒ NUM(1) case _ ⇒ NUM(0)))
  .extend("cons",  FUN(case [x, y] ⇒ CONS(x, y)));
```


The Global Environment

We evaluate Lisp expressions in an initial (global) environment, which contains definitions for commonly used operations such as + or *cons*.

Here is a minimally useful version of such an environment.

Summary

We have seen how functional programs are evaluated by writing an interpreter for Lisp.

The central auxiliary data structure was the environment, which represents all bindings known at the point of computation.

Environments replace substitutions in the formal model of evaluation.

Extended Exercise

Augment the Lisp interpreter so that it has its own interpreter loop, which accepts individual definitions and expressions.

Given an input definition, the interpreter would add the defined bindings to the global environment.

Given an input expression, it would evaluate the expression and print the result.