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 a Scheme– expression (of type *Data*) which is one a the following,

- a number,
- a string,
- a symbol
- a list of expressions

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

For instance, when applied to the input expression

```
List('*, 2, 7)
```

the interpreter should return the output expression

```
14
```

But now a question poses itself: When the interpreter input is a symbol, such as `'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 {
  def lookup(n: String): Data = ...
  def extend(name: String, v: Data): Environement = ...
}
```

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 {
  def lookup(n: String): Data;
  def extend(name: String, v: Data) = new Environment {
    def lookup(n: String): Data =
      if (n == name) v else Environment.this.lookup(n);
  }
}
val EmptyEnvironment = new Environment {
  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.

Note: The construct *Environment.**this*** refers to the enclosing self-object which is an instance of the *Environment* class (as opposed to the object created from the enclosing anonymous class).

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 $*$? It is neither a number, nor a string, symbol or list!

We need to create a data type, which represents functions on *Lisp* expressions:

$$\textbf{\textit{case class}}\ Lambda(f\colon List[Data] \Rightarrow Data);$$

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

$$Lambda\ \{\ \textbf{\textit{case}}\ List(arg1\colon int,\ arg2\colon int) \Rightarrow arg1 * arg2\ \}$$

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 integers,
- 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 *Lambda*'s argument type $List[Data] \Rightarrow Data$.

The Evaluator

We now present the Lisp evaluator.

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

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

In a first step, 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 y: String ⇒ x
  case y: int ⇒ x
  case Lambda(_) ⇒ x
  case Symbol(name) ⇒
    env lookup name
  case 'val :: Symbol(name) :: expr :: rest :: Nil ⇒
    eval(rest, env.extend(name, eval(expr, env)))
  case 'if :: cond :: thenpart :: elsepart :: Nil ⇒
    if (asBoolean(eval(cond, env))) eval(thenpart, env)
    else eval(elsepart, env)
  case 'quote :: y :: Nil ⇒ y
  case 'lambda :: params :: body :: Nil ⇒
    mkLambda(params, body, env)
  case operator :: operands ⇒
    apply(eval(operator, env), 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> <expr> <rest>)
  ```

  we evaluate the <*rest*> expression in an environment extended by the binding of <*name*> to the result of evaluating <*expr*>.
- 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 *asBoolean*, which is defined as follows.


```
def asBoolean(x: Data): boolean =
  if (x == 0) false else 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> <body>)
  ```

  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 *Lambda* node and then applies its associated operation to the argument lists.

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

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

Making Functions

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

The call *mkLambda*(*params*, *body*, *env*) yields a *Lambda* object which contains 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] = asList(params) map {
    case Symbol(name) ⇒ name
    case _ ⇒ error("illegal parameter list");
  }
  Lambda(args ⇒ eval(expr, extendEnv(env, ps, args))) }
```

- 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.

Note: The *asList* function casts an arbitrary object to a list. It is defined as follows:

```
def asList(x: Data): List[Data] =
  if (x is List[Data]) x as List[Data]
  else lispError("malformed list: " + x);
```

is and *as* are Scala's constructs for type tests and type casts.

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 =
  Pair(ps, args) match {
    case Pair(List(), List()) ⇒
      env
    case Pair(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.

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).

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:

$$\begin{array}{l}\textbf{def}\, f(x\colon \mathit{Int}) = g(y \Rightarrow x + y)\\ \textbf{def}\, g(z\colon \mathit{Int} \Rightarrow \mathit{Int}) = z(2);\\ f(1)\end{array}$$

the program returns 3, as expected.

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

$$\begin{array}{l}\textbf{def}\, \mathit{iterate}(n\colon \mathit{Int},\, f\colon \mathit{Int} \Rightarrow \mathit{Int}) =\\ \quad \textbf{if}\ (n == 0)\ x \Rightarrow x\\ \quad \textbf{else}\ \mathit{nTimes}(n-1,\, x \Rightarrow f(x))\end{array}$$

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:

$$\textbf{def}\ \mathit{incrementer}\,(x\colon \mathit{Int}\,) = y \Rightarrow y + x$$

Question: What is the effect of $\mathit{incrementer}\,(2)\,(3)$?

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.

The method for handling `val` in the interpreter,

case *'**val*** :: *Symbol*(*name*) :: *expr* :: *rest* :: *Nil* ⇒
eval(*rest*, *env.extend*(*name*, *eval*(*expr*, *env*)))

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

- `expr` is evaluated too early; it should be evaluated only when `name` is accessed.
- `name` does not form part of the environment visible to `expr`, 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.

Evaluation of `def`

Here is how `def` can then be treated:

case *'**def*** :: *Symbol*(*name*) :: *expr* :: *rest* :: *Nil* ⇒
eval(*rest*, *env.extendRec*(*name*, (*env1* ⇒ *eval*(*expr*, *env1*))))

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

Environments Refined

Here is the new definition of environments.

```
abstract class Environment {
  def lookup(n: String): Data;
  def extendRec(name: String, expr: Environment ⇒ Data) =
    new Environment {
      def lookup(n: String): Data =
        if (n == name) expr(this) else Environment.this.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.

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)))))))
```

The Global Environment

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

Here is a minimally useful version of such an environment.

```
val globalEnv = EmptyEnvironment
  .extend("=", Lambda{
    case List(arg1, arg2) ⇒ if(arg1 == arg2) 1 else 0} )
  .extend("+", Lambda{
    case List(arg1: int, arg2: int) ⇒ arg1 + arg2
    case List(arg1: String, arg2: String) ⇒ arg1 + arg2} )
  .extend("−", Lambda{
    case List(arg1: int, arg2: int) ⇒ arg1 − arg2} )
```



```
  .extend("∗", Lambda{
    case List(arg1: int, arg2: int) ⇒ arg1 ∗ arg2} )
  .extend("/", Lambda{
    case List(arg1: int, arg2: int) ⇒ arg1 / arg2} )
  .extend("nil", Nil)
  .extend("cons", Lambda{
    case List(arg1, arg2) ⇒ arg1 :: asList(arg2)} )
  .extend("car", Lambda{
    case List(x :: xs) ⇒ x} )
  .extend("cdr", Lambda{
    case List(x :: xs) ⇒ xs} )
  .extend("null?", Lambda{
    case List(Nil) ⇒ 1
    case _ ⇒ 0} );
```

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 add the derived special forms discussed last week such as `and`, `or`, or `cond`, we would modify this to:

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

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 = lisp2string(evaluate(string2lisp(s)));
```


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
```

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.

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.