Interpreting Scheme--

A Scheme-- interpreter

We will now present an interpreter for Scheme--.

This is useful for two reasons:

1. Many software systems include a small language, which is often interpreted.
2. The interpreter tells us in a precise way how Scheme-- programs in particular, and functional programs in general, are evaluated.

Layout of the interpreter

The interpreter accepts a Scheme-- expression (of type `Data`) as its input, that is, either

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

The interpreter returns another `Data` expressions representing a Scheme-- value as its output.

For example, when applied to the input expression

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

the interpreter should return the output expression

```
14
```

Layout of the interpreter (cont)

This brings up the following question:

What should the interpreter return when it is given a symbol, such as `x`, as its input?

This depends on whether the name `x` is defined when the symbol is evaluated, and if yes, to what value it is bound.

The interpreter has to store defined names in an *environment*.

Environments

An environment is a data structure that associates Scheme-- values with names.

The two basic operations on an environment are

- `lookup`: given a name, returns the value associated with that name.
- `extend`: given a name-value binding, returns a new environment extended with that binding.

This leads to the following class layout:

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

Environments (cont)

We also need to define a value `EmptyEnvironment` that represents the empty environment.

Environments (cont)

Different data structures can be used to implement environments, with different performance trade-offs. (Examples?)

We will use a direct approach without any auxiliary data structures.

```
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)
}
```

Environments (cont)

This is short and simple, but the `lookup` method takes time proportional to the number of bindings in the environment.

Note: The construct `Environment.this` refers to the current enclosing object that is an instance of the class `Environment` (as opposed to the current instance of the anonymous inner class).

Predefined functions

We also need a way to interpret "standard" symbols such as $*$.

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

But what should the value of $*$ be? It is neither a number, nor a string, nor a list!

We need to introduce a data type to represent functions acting on Scheme-- expressions:

```
case class Lambda(f: List[Data] => Data)
```

For example, the value associated with the $*$ operation will be:

```
Lambda { case List(arg1: Int, arg2: Int) => arg1 * arg2 }
```

Predefined functions (cont)

Note: the example illustrates the power of pattern matching – in a single expression, we can specify

- that the argument list must be of length two,
- that the two arguments must be integers,
- and we make the values of the two arguments available through the names `arg1` and `arg2`.

Note also that the `case` expression is a (partial) function of type `List[Data] => Data` that pattern-matches the arguments.

The evaluator

We now present our Scheme-- evaluator.

It takes the internal representation of a Scheme-- expression as its input, that is, either

- an atom: symbols, numbers, strings, or
- a combination, which in turn can be a special form or an application.

The evaluation function

```
def eval(x: Data, env: Environment): Data = x match {
  case _: String => x
  case _: Int => 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 (eval(cond, env) != 0) eval(thenpart, env)
    else eval(elsepart, env)
  case 'quote :: y :: Nil => y
  case 'lambda :: params :: body :: Nil =>
    mkLambda(asList(params) map paramName, body, env)
  case operator :: operands =>
    apply(eval(operator, env), operands map (x => eval(x, env)))
```

Explanations

- If the input expression `x` is a number, a string or a function, we return the expression itself.
- If the expression is a symbol, we look up the symbol in the current environment and return the result.
- If the expression is a special form

  ```
  (val <name> <expr> <rest>)
  ```

  we evaluate the expression `<rest>` in an environment extended by the binding of `<name>` to the result of the evaluation of `<expr>`.
- If the expression is a special form

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

  we first evaluate `<cond>`. If the result is the number `0`, we continue by evaluating `<thenpart>` ; otherwise by evaluating `<elsepart>`.

Explanations (cont)

- If the expression is a special form

  `(quote <expr>)`

  we return `expr` as the result of the evaluation.
- If the expression is a special form

  `(lambda <params> <body>)`

  we create a new function by using the operation `mkLambda` (see below).
- Finally, if the expression is a combination that corresponds to none of the special forms above, it must be a function application.
  In this case, we first evaluate the operator of the application, as well as all its operands, and then *apply* the value of the operator to the values of the operands.

Explanations (cont)

The function `apply` first verifies that the operator is indeed a `Lamba` node and then applies the associated operation to the argument list.

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

That’s it, except, we still need to define how a `Lambda` expression is converted into a function that can be applied to a `Data` expression.

Constructing functions

The conversion from a Scheme-- expression in to a Scala function uses the helper function `mkLambda`.

A call to `mkLambda(params, body, env)` returns a `Lambda` object that contains a (Scala) function.

This function associates a list of arguments, which correspond to the list of formal parameters `params`, with the result of the evaluation of `body` in `env`.

Here is how this works:

```scala
def mkLambda(ps: List[String], body: Data, env: Environment) =
  Lambda { args => eval(body, env.extendMulti(ps, args)) }
```

Constructing functions (cont)

- A call to the function will trigger the evaluation of `body` in an environment extended by the bindings that associate the names of the formal parameters with the actual values of the corresponding arguments.
- Note the analogy with the substitution model, where the formal parameters were *replaced* in `body` by the actual arguments `args`.
- We can regard an environment as a deferred substitution: rather than performing the replacement immediately, we do it once the symbol is looked up in the environment.
- It would also be possible to write an interpreter based on substitutions rather than environments; but such an interpreter would turn out to be more complicated and less efficient.

Constructing functions (cont)

Note: the function `asList` allows us to promote an arbitrary object to a list. It is defined as follows:

```
def asList(x: Data): List[Data] = x match {
  case xs: List[_] => xs
  case _ => error(”malformed list : ” + x)
}
```

`List[_]` is a type pattern, where the parameter of the type `List` is unspecified.

Constructing functions (cont)

Finally, we need to define the method `extendMulti`, which extends an environment with a list of bindings between formal parameters and argument values.

This method is defined as follows:

```
abstract class Environment {
  // ...
  def extendMulti(ps: List[String], vs: List[Data]): Environment =
    (ps, vs) match {
      case (List(), List()) =>
        this
      case (p :: ps1, arg :: args1) =>
        extend(p, arg).extendMulti(ps1, args1)
      case _ =>
        error("wrong number of arguments")
    }
```

Constructing functions (cont)

- The method verifies that the two lists containing the parameters and the argument values, respectively, have the same length.
- For each name-value pair, it calls the `extend` method.

Environments for lambdas

Note that functions returned by the operation `mkLambda` retain a reference to the environment that was the current environment when the function value was created.

This environment is used during applications of the function.

Here is an example of the evolution of the environments during the execution 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 first version of Lisp used a global stack to store environments.

Upon entering a function, the bindings for its parameters and local variables were pushed to the stack.

When exiting the function, the stack was reset.

As a consequence, higher-order functions behaved very strangely!

For example, the evaluation of `f(1)` in the above program causes a run-time error, because the anonymous function `y => x + y` accesses the last value of `x` on the stack, which is a function, not an integer value.

In this particular case, the problem could be resolved by renaming the parameters to make their names unique.

Dynamic scoping in original Lisp (cont)

For example, after renaming

```
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 it is not always possible to work around the problem using only renaming:

```
def fact(n: Int, f: () => Int): Int =
  if (n == 0) f()
  else fact(n - 1, () => n * f())
```

Question: What effect has the evaluation of `fact(7, () => 1)` with the stack-based environment of original Lisp?

Dynamic scoping in original Lisp (cont)

Another problem with original Lisp is that returning functions is very dangerous. Consider the following:

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

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

The technique used to implement environments in original Lisp is called *dynamic scoping*.

It means that the binding between an identifier and its value is determined dynamically – it depends on the form of the environment at the moment when the value is referenced.

Dynamic scoping offers some interesting possibilities, but it interferes heavily with the use of higher-order functions.

Therefore, most modern versions of Lisp, such as Scheme, use static scoping, just like Scala.

Dynamic scoping in original Lisp (cont)

Older versions, including elisp, still use dynamic scoping.

Common Lisp, of course, uses both.

Taking recursion into account

We have seen that the way of implementing environments employed by original Lisp leads to problems.

But so does ours once we add recursion!

Recursion can be introduced by adding a special form

```
(def <name> <expr> <rest>)
```

to our interpreter.

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

Taking recursion into account (cont)

The method we use to handle `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:

- The body `expr` is evaluated too early; it should not be evaluated until we access `name`.
- `name` is not part of the environment that is visible to `expr`, hence the recursion is impossible.

We can resolve these problems by adding a new method `extendRec` that extends the environment with a binding containing a potentially recursive computation.

Evaluation of def

We can handle def as follows:

```
case 'def :: Symbol(name) :: expr :: rest :: Nil =>
  eval(rest, env.extendRec(name, env1 => eval(expr, env1)))
```

- This solves the problem of premature evaluation as we now extend the environment with a function that will evaluate expr once it is called, instead of the result of the evaluation of expr.
- Next, we need to enable recursion by changing the lookup method of environments.

New environments

Here is the new definition for 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 main operation used to extend an environment; the method `extend` is defined in terms of `extendRec`.

Note also that `lookup` now enables recursion by passing the current environment to the function it retrieves.

Recursion through self-application

This technique reveals a profound connection in programming: we can model recursion through self-application.

In fact, any recursion is ultimately handled by self-application in the lambda calculus, the underlying theory of functional programming.

To illustrate this without further explanation, consider the following version of the factorial function 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 Scheme-- expressions in an (global) initial environment that contains the definitions of the operations and constants that are currently in use, such as +, `cons`, or `nil`.

Here is a minimal 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})
```

The global environment (cont)

```
.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 main interpreter function

The main function of the interpreter looks as follows:

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

It evaluates a Scheme-- program in the global environment and returns the result.

In order to add the derived special forms we saw last time, such as `and`, `or`, or `cond`, we need to modify it as follows:

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

To simplify the life of programmers who want to enter Scheme-- expressions on the command line, we also include another version of `evaluate` that accepts and returns Scheme-- expressions in string format:

```
def evaluate(s: String): String = lisp2string(evaluate(string2lisp(s)))
```

Using Scheme-- from Scala

Here is a use case for evaluating Scheme-- code in the Scala REPL:

We first define a Scheme-- function as a string:

```
> def factDef =
    """def factorial (lambda (n)"
         (if (= n 0) +
           1 +
           (* n (factorial (- n 1)))))
    """
```

We can then call this function as follows:

```
> evaluate("(" + factDef + "(factorial 4))")
24
```

Exercise

Extend the Scheme-- interpreter with a proper REPL that accepts definitions and individual expressions.

When given a definition as its input, the REPL should add the corresponding binding to the global environment.

When given an expression as its input, the REPL should evaluate it and print the result.

Summary

We saw how functional programs are evaluated by writing an interpreter for a Lisp dialect.

The main auxiliary data structure used was the environment, which represents the currently known bindings at the time of the computation.

Environments replace the substitutions presented in the formal evaluation model.