An Example

Here is the definition and a use case of the `map` function in Scheme:

```
(define (map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs)) (map f (cdr xs)))))
(map (lambda (x) (* x x)) '(1 2 3))
```

What does this expression evaluate to?

Why Scheme interpreter?

- We will implement a complete programming language that is capable of expressing any algorithm (Turing complete).
- This language will have an external syntax that resembles Lisp, not Scala. This syntax will be translated into an internal Scala data structure through a parser.

The implementation consists of three steps.

1. Define an internal representation for Scheme-- programs.
2. Define a translator from strings to the internal representation.
3. Define an interpreter capable of evaluating the internal representation.

Internal representation for Scheme--

Our internal representation for Scheme-- strictly follows the data structures used in Scheme. It also reuses the corresponding Scala constructs whenever possible.

We define a type `Data`, representing data in Scheme, as an alias of the type `scala.Any`.

```
type Data = Any
```

- Numbers and strings are represented as values of the Scala types `Int` and `String`,
- lists are represented by Scala lists,
- symbols are represented as instances of the Scala class `Symbol` (see below).

Scala Symbols

- Symbols in Scala are values that represent identifiers.
- There is a simplified syntax for such symbols: if `name` is an identifier, then

  `'name`

  is a shorthand for `Symbol("name")`.

The standard class `Symbol` is defined as follows in the package `scala`:

```
case class Symbol(name: String) {
    override def toString() = "'" + name
}
```

Example: a Scheme-- program defining and using the factorial

```
(def factorial
     (lambda (n)
       (if (= n 0)
           1
           (* n (factorial (- n 1)))))
  (factorial 5))
```

Its internal representation can be constructed as follows:

```
List('def, 'factorial,
     List('lambda, List('n),
          List('if, List('=, 'n, 0),
               1,
               List('*, 'n, List('factorial, List('-, 'n, 1))))),
     List('factorial, 5))
```

From text to internal representation

We are now going to write a parser that constructs an internal representation for a given string.

This construction consists of two steps:

1. from a string to a sequence of words (called lexemes or tokens)
2. from a sequence of words to a `Data` tree.

Here, a token can be

- an opening parenthesis "`(`" or a closing parenthesis "`)`",
- a sequence of characters that does not contain any white space characters or parentheses.

Tokens that are not parentheses must be separated by white space, that is, spaces, new line characters or tabs.

The tokenizer

We represent the sequence of words that make up a Lisp program using an iterator class, defined as follows:

```
class LispTokenizer(s: String) extends Iterator[String] {
    private var i = 0
    private def isDelimiter(ch: Char) = ch <= ' ' || ch == '(' || ch == ')'
    def hasNext: Boolean = {
      while (i < s.length() && s.charAt(i) <= ' ') { i = i + 1 }
      i < s.length()
    }
    ...
```

Remarks:

- The iterator holds a private variable `i` that denotes the index of the next character to be read.

The tokenizer (cont)

- The function `hasNext` skips over all the white space preceding the next token (if it exists). It uses the method `charAt` from the Java class `String` to access the characters of a string.

The tokenizer (cont)

```
  ...
  def next: String =
    if (hasNext) {
      val start = i
      var ch = s.charAt(i); i = i + 1
      if (ch == '(') "("
      else if (ch == ')') ")"
      else {
        while (i < s.length() && !isDelimiter(s.charAt(i))) { i = i + 1 }
        s.substring(start, i)
      }
    } else error("more input expected")
}
```

- The function `next` returns the next token. It uses the method `substring` of the class `String`.

Parsing and construction of the tree

Given the simplicity of the Lisp syntax, it is possible to parse it without the use of advanced parsing techniques.

This is how it works:

```
def string2lisp(s: String): Data = {
  val it = new LispTokenizer(s)
  def parseExpr(token: String): Data = {
    if (token == "(") parseList
    else if (token == ")") error("unmatched parenthesis")
    else if (token.charAt(0).isDigit) token.toInt
    else if (token.charAt(0) == '\"'
             && token.charAt(token.length()-1)=='\"')
      token.substring(1,token.length - 1)
    else Symbol(token)
  }
```

Parsing and construction of the tree (cont)

```
  def parseList: List[Data] = {
    val token = it.next
    if (token == ")") Nil else parseExpr(token) :: parseList
  }
  parseExpr(it.next)
}
```

Remarks:

- The function `string2lisp` converts a string in a tree expressions of type `Data`.
- It starts by defining a `LispTokenizer` iterator called `it`.
- This iterator is used by the two mutually recursive functions `parseExpr` and `parseList` (recursive descent parsing).
- `parseExpr` parses simple expressions.
- `parseList` parses a list of expressions enclosed by parentheses.

Parsing and construction of the tree (cont)

Now, if we write the following in the Scala REPL:

```
string2lisp("(lambda (x) (+ (* x x) 1))")
```

we obtain (without the indentation)

```
List('lambda, List('x),
  List('+,
    List('*,'x,'x),
    1))
```

Exercise

Write a function `lisp2string(x: Data)` that prints Lisp expression in Lisp format. For example

```
lisp2string(string2lisp("(lambda (x) (+ (* x x) 1))"))
```

should return

```
(lambda (x) (+ (* x x) 1))
```

Special forms

Our Scheme-- interpreter will be capable of evaluating a single expression only.

This expression can have one of the following special forms:

- `(val x expr rest)`
  evaluates `expr`, binds the result to `x` and then evaluates `rest`. Analogous to `val x = expr; rest` in Scala.
- `(def x expr rest)`
  binds `expr` to `x` and then evaluates `rest`. `expr` is evaluated every time `x` is used. Analogous to `def x = expr; rest` in Scala.

The real Scheme contains a variety of binders called `define`, for the top level, and `let`, `let*` and `letrec` used inside combinations.

Scheme's `define` and `letrec` correspond roughly to `def`, while Scheme's `let` corresponds roughly to `val`, but their syntax is more complicated.

Special forms (cont)

- `(lambda (`$p_1 \ldots p_n$`) expr)`
  defines an anonymous function with parameters $p_1 \ldots p_n$ and body `expr`
- `(quote expr)`
  returns `expr` without evaluating it.
- `(if cond then-expr else-expr)`
  is the usual conditional.

Syntactic sugar

Some other forms can be converted into the above through transformations on the internal representation.

We can write a function `normalize` that eliminates these other special forms from the tree.

For example, Lisp supports the special forms

```
(and x y)
(or x y)
```

for the short-circuited logic relations `and` and `or`. The `normalize` function can convert them to `if` expressions as follows:

Syntactic sugar (cont)

```
def normalize(expr: Data): Data = expr match {
  case 'and :: x :: y :: Nil =>
    normalize('if :: x :: y :: 0 :: Nil)

  case 'or :: x :: y :: Nil =>
    normalize('if :: x :: 1 :: y :: Nil)

  // other simplifications...
}
```

Derived forms

Our normalization function accepts the following derived forms:

```
(and x y)                          => (if x y 0)
(or x y)                           => (if x 1 y)

(def (name args_1 ... args_n)      => (def name
   body                                 (lambda (args_1 ... args_n) body)
 expr)                                expr)

(cond (test_1 expr_1) ...          => (if test_1 expr_1
      (test_n expr_n)                    (...
      (else expr'))                        (if test_n expr_n expr')...))
```