Interpreting a Language

Matt Neary

August 2013

The most important revelation in learning the art of programming is that the language in which you work is completely arbitrary. More specifically, the language in which you express concepts was defined in terms of another language at some point. We have already made clear this concept in our definition of our symbolic language. We now turn to the interpretation of one language within another.

0.1 Lambda Calculus

The language which we will interpret is one with which we are already familiar, Lambda Calculus. The Lambda Calculus has very simple syntax and will thus not be too hard to interpret. Recall the syntax, which is composed of the following expressions.

- A variable reference.
- A function definition of the form λab where a is a variable reference and b is an expression.
- A function application of the form $(a)b$ where a is an expression, as is b.

Note that the generality of the third form, function application, is what gives this syntax its description as the Untyped Lambda Calculus. Since no qualification is given to the expression which will be passed argument, this language is without types.

0.2 Expression of Lambda Calculus in S-Expressions

In expression the Lambda Calculus in S-Expressions, we will utilize the 'quote' function as well as the structure inherent of parenthetical expressions in these expressions. Hence an example of an expression which could be evaluated is the following.

$$'(lam\ x\ lam\ y\ (x)\ y)$$

0.3 An Evaluator

We define our evaluator pretty easily. Note that we will begin by defining an 'apply' function. This function accepts a function and list of arguments, and then applies each of these arguments to a lambda one by one.

```
(define (apply-set fn args)
  (if
    (null? args)
    fn
    (apply-set (fn (car args)) (cdr args))))
```

Just like our definition of the syntax, our evaluator handles variable reference, lambda definition, and function application.

Variable reference is a problem very easily solved. Atoms are considered variable references, and hence serve as keys in accessing values from the environment hash.

Function definition is achieved by returning a lambda of a single variable for expressions of the necessary form. Within the lambda, the passed argument is appended to the environment with the argument name as its key. The function body is then evaluated.

Function application is the default case, thus we match against the antecedent ‘t‘. The applier ‘apply-set‘ is then called with the evaluated form of the first argument and the evaluated arguments. This architecture is an explicit choice *not* to opt for a lazy method of evaluation.

```
(evallam (lambda (evallam expr env)
  (cond (((atom? expr) (assoc expr env))
         ((equal? (car expr) 'lam)
           (lambda (x)
             (evallam
               (cddr expr)
               (set (cadr expr) x env))))
         ((null? (cdr expr))
          (evallam (car expr) env))
         (t
           (apply-set
             (evallam (car expr) env)
             (map
               (lambda (expr) (evallam expr env))
               (cdr expr)))))))))
```

0.4 Evaluation of Forms

An example of a form which could be evaluated is the following.

$$'(lam\ x\ lam\ y\ ((x)\ y)\ 1)$$

0.5 Flat-Input Lambda Calculus

In the prior implementation of an interpreter, we took advantage of the structure inherent to a nested S-Expression. This approach was sufficient for our initial purposes; however, to separate our interpreter from the details of its use within our Symbolic Language, we will now allow its interpretation to apply to a flat list of atoms. In order to represent the expression previously expressed by nested S-Expressions, we will now utilize some symbols which will represent parenthetical expressions. The following example of this new flat structure.

$$'(lam\ x\ lam\ y\ <\ x\ >\ y)$$

Of course, this use of '¡¿' symbols would extend to any instance of parentheses in our prior method.

With our new, less inherently structured approach, we will need to provide an additional layer of parsing. Parsing will provide this missing aspect of structure. Parsing parentheses is actually our most complex algorithm yet attempted. We will take this algorithm's implementation as an opportunity to experiment with the second style of programming we have yet to investigate, imperative programming.

0.6 The Two Styles of Programming

There are two basic approaches to programming, derived from the two original theories of computation. We have talked far more about functional programming tactics in prior sections of this book, leaving imperative programming on the sidelines. However, the problem at hand is a great case study in the relation between imperative and functional languages. We will begin with an imperative implementation, and then port the code over to our current language of choice.

0.7 Imperative Constructs

In our exploration of imperative programming, we will encounter a few new operators, and utilize some new idioms. In later chapters, we will provide implementation of these operators. For now, however, these operators will be viewed as hypothetical constructs, fulfilling the utility with which we associate them.

0.8 Mutators

The main difference between imperative and *purely* functional programming is the presence of mutability. In functional programs, a value can be defined but not mutated; however, when taking the imperative approach, values will often be set to a new value after their definition. The following is an example of this behavior.

$(define\ x\ 5)$
$(set!\ x\ (*\ 2\ x))$
$(equal?\ x\ 5)$
$;;\ =>\ f$
$(equal?\ x\ 10)$
$;;\ =>\ t$
$=>\ x\ =\ 10$

The ‘define‘ operator serves to allocate a variable and initiate it with a value. This variable, ‘x‘, can then be accessed throughout the procedure, and even mutated to equal a new value. In the example, it was initiated as 5, but ‘set¡ to 10.

The *scoping* of a variable is specified by the define operator; hence the following is another example of this behavior.

$(define\ x\ 5)$
$((lambda\ (y)$
$\quad (set!\ x\ y))\ 12)$
$(equal?\ x\ 12)$
$;;\ t$

Of note is the fact that the ‘define‘ occurred separate from any function. This means that the defined variable will now take on the *global* scope, being accessible and mutable from within any function. If the ‘define‘ instead occurred within a function definition, as in the following, it would only be accessible from within that function, or other functions defined within it.

$(define\ scope\ (lambda\ (x)$
$\quad (define\ y\ x)))$
$(scope\ 5)$
y
$;;\ The\ written\ variable,\ `y`,\ will\ be\ inaccessible.$

In the above we demonstrate definition with a single-function scope. Thus the ‘define‘ is fulfilling the same role as ‘let‘ did in prior programs.

However, since ‘define‘ does not accept an expression which it will govern, the example definition is of no effect. In the following section we display a means of making use of this sort of ‘define‘ statement.

0.9 Multiple Expression Procedures

In our earlier, purely-functional programs, a procedure consisting of multiple expressions would have been no use. Without side-effects, only the final expression could bear any form of result. However, now investigating an imperative approach, a procedure may utilize multiple expressions, each contributing its own mutation to a final effect. The following is an example of this in practice; the syntax is simply a chain of expressions where an individual would have previously existed.

$$
\begin{array}{l}
(define\ incr\ (lambda\ (x) \\
\quad (define\ y\ (+\ x\ 1)) \\
\quad y))
\end{array}
$$

Obviously, this example is of no utility. The desired function could be just as easily achieved with a single expression. Useful examples, however, will present themselves in the following sections.

0.10 Loop Constructs

You will often see imperative programming avoiding use of recursion. Rather, these programs will often iterate, mutating the environment in each step. For convenience in utilization of this approach, we define a function for constructing a range over which to iterate.

$$
\begin{array}{l}
(define\ range\ (lambda\ (x) \\
\quad (if\ (equal?\ x\ 0) \\
\quad\quad nil \\
\quad\quad (cons\ (-\ x\ 1)\ (range\ (-\ x\ 1))))))
\end{array}
$$

The above definition is pretty straight-forward, much like earlier function definitions. Note that the ranges are of the form 0, 1, ..., n-1. Here's an example of this function being used to calculate a factorial.

```
(define fact (lambda (x)
  (define ans 1)
  (map (range x) (lambda (n)
    (set! ans (* and (+ 1 n)))))
  ans))
```

The starting value of the answer is 1, just like the sort of inductive definitions we provided earlier in the book. The final answer is then achieved by repeated multiplication performed on the previous 'ans'. In the case of 5, for example, the accumulator 'ans' takes on the following values.

$$
\begin{aligned}
&1 \\
&\implies 1 * 1 \implies 1 \\
&\implies 1 * 2 \implies 2 \\
&\implies 2 * 3 \implies 6 \\
&\implies 4 * 6 \implies 24 \\
&\implies 5 * 24 \implies 120
\end{aligned}
$$

0.11 An Imperative Solution

Now we will jump right in to the non-trivial problem at hand, restated below.

Given a string of nested angle-bracket delimited groups, return a nested list containing the contents of these groups. For example, given the list of characters $'(a¡bc¿d)$ return $'(a(bc)d)$.

Since we are taking an imperative approach, think, "What is the easily defined iterative process underlying this problem?" The answer is clearly navigation of the string, and so we begin with a 'range'-based loop that will cycle through each character of the string in order.

```
(define parse (lambda (expr)
  (map (range (length expr)) (lambda (i)
    (define read (get expr i))
    // ...
  )))
```

Now we will need to describe a slightly more specific strategy in performing the desired process.

- A parenthetical will be split from the string, with a segment, although possibly an empty one, before and after it.
- Once a parenthetical has been removed, we will need to recurse on these segments, i.e., the parenthetical and the portion after it.

To make our way toward this implementation, we will define a variable 'before' that will hold the segment of the string occurring prior to any parenthetical; a variable 'accum' that will hold characters that have been read in but whose destination has yet to be determined, in this way serving as a cache; 'paren' which will hold a separated out parenthetical; and 'found' which will be true if and only if a parenthetical has been parsed.

```
(define parse (lambda (expr)
  (define before)
  (define accum nil)
  (define paren)
  (define found f)
  (map (range (length expr)) (lambda (i)
    (define read (get expr i))
    // ...
  )))
```

In order to parse out the parenthetical, however, we will need an additional variable. This variable will aid us in parsing nested parentheses to separate out the top-level parenthetical.

We will need to handle three obvious classes of characters in our parsing of the parentheses:

- An opening parenthesis.
- A closing parenthesis.
- Any other character.

Additionally, the class of a character may be disregarded if we have already parsed a top-level parenthetical. Its parsing will be handled when we are ready to recurse. Given these additions of case-handling, we insert 'if ... else' statements as in the following.

```
(define parse (lambda (expr)
  (define before)
  (define accum nil)
  (define paren)
  (define found f)
  (define nested 0)
  (map (range (length expr)) (lambda (i)
    (define read (get expr i))
    (if (and (equal?' < read) (not found))
      (..."1. an opening parenthesis"...)
      (if (and (equal?' > read) (not found))
        (..."2. a closing parenthesis"...)
        (..."3. any other character"...))))))
```

Of course, we will need to combine any separated out parenthetical with the components occurring before and after it to form the designated response. Hence we provide the following 'return' statement.

```
(define parse (lambda (expr)
  ..."variables"...
  (map (range (length expr)) (lambda (i)
    (..."parse"...))
  (if paren
    (concat (push before paren) (parse accum))
    expr)))
```

Now we implement our nesting logic and the final algorithm. Nesting will be handled based on one of the following occurrences.

- A once nested expression was just opened.

- An expression was just closed to be un-nested.

Parentheses occurred within a nested expression.

The first and second cases are handled under the conditionals for their respective character classes, and in either class under another nesting case, the third will be handled.

The last components missing from our implementation are the building up of an accumulator and the setting of the various components to the accumulator. We will implement these portions in the following code.

- When the parenthetical is closed, it is recursively ‘parsed‘ and set to the ‘paren‘ variable.
- When a parenthetical is open, ‘before‘ receives the accumulator value.

```
(define parse (lambda (expr)
  (define before)
  (define accum nil)
  (define paren)
  (define found f)
  (define nested 0)
  (map (range (length expr)) (lambda (i)
    (define read (get expr i))
    (if (and (equal? '< read) (not found))
      ((set! nested (+ 1 nested))
       (if (equal? nested 1)
         ((set! before accum)
          (set! accum nil))
         (set accum (push accum read)))
       (if (and (equal? '> read) (not found))
         ((set! nested (- 1 nested))
          (if (equal? nested 0)
            ((set! found t)
             (set! paren (parse accum))
             (set! accum nil))
            (set accum (push accum read))))
         (set accum (push accum read)))))))
  (if paren
    (concat (push before paren) (parse accum))
    expr)))
```

0.12 From Imperative to Functional

From the above final implementation of our program we can derive a functional version. The differences will be based on the following principles of functional programming:

- Values shall not be mutated.

- Control-flow shall not be explicit.
- Recursion is dope.

Let's begin by abiding to the second rule, inspired by the third. The first thing you will notice is that all variables were made function arguments. This is because in a pure function, the only state is provided by the arguments. Hence when recursing, we will need to pass all required data to the function as argument.

Also of note is the fact that rather than maintain an index of the list on which we are operating, we pass as argument to the recursive call only subsequent characters, i.e., those which have yet to be read. This is both logical in that our progress in navigating the list is maintained, and idiomatic as you have seen in prior programs written in our Symbolic Language.

```
(define funparse (lambda (expr nested before paren accum found)
  (if (null? expr)
    (if (not (null? paren))
      (concat (push before paren) (funparse_ accum))
      expr)
    ((let read (get expr 0)
       (if (and (equal? read '<) (not found)))
         ((set! nested (+ 1 nested))
          (if (equal? 1 nested)
            ((set! before accum)
             (set! accum nil))
            (set accum (push accum read))))
         (if (and (equal? read '>) (not found)))
          ((set! nested (- 1 nested))
           (if (equal? 0 nested)
             ((set! paren (funparse_ accum))
              (set! found t)
             (set! accum nil))
               (set accum (push accum read))))
       (set accum (push accum read)))
     (funparse (cdr expr) nested before paren accum found)))))
(define funparse_ (lambda (expr)
  (funparse expr 0 '() '() '() f)))
```

The final portion of our program includes a definition of 'funparse_'. This was merely for convenience, as 'funparse_' provides all of the initialization values as argument to 'funparse'.

We now remove mutation to achieve implementation of the final principle we listed. Our means of achieving this is by allowing all values to be function arguments or expressions operating on arguments.

```
(define funparse (lambda (expr nested before paren accum found)
  (if (null? expr)
    (if (not (null? paren))
      (concat (push before paren) (funparse_ accum))
      expr)
    (if (and (equal? '< (get expr 0)) (not found))
      (if (equal? nested 0)
        (funparse (cdr expr) (+ nested 1) accum paren nil found)
        (funparse (cdr expr) (+ nested 1) before paren (push accum (car expr)) found))
      (if (and (equal? '> (get expr 0)) (not found))
        (if (equal? nested 1)
          (funparse (cdr expr) (- nested 1) before (funparse_ accum) nil t)
          (funparse (cdr expr) (- nested 1) before paren (push accum (car expr)) found))
        (funparse (cdr expr) nested before paren (push accum (car expr)) found))))))
(define funparse_ (lambda (expr)
  (funparse expr 0 '() '() '() f)))
```

You should begin to see how our rewrite of this algorithm reads much more as an inductive definition than as a description of a process. In the following section we will make this even more evident.

0.13 Adopting a Few Conventions

There are a few vestiges of our initial, imperative implementation which we will now remove. Of note is the prior ‘define‘ keyword that was appropriately substituted by ‘letrec‘, with ‘funparse_‘ then being another definition within the ‘letrec‘ procedure.

```
(letrec funparse (lambda (funparse expr nested before paren accum found)
  (let
    funparse_
    (lambda (expr) (funparse expr 0 '() '() '() f))
    (if (null? expr)
      (if (null? paren)
        expr
        (concat (push before paren) (funparse_ accum)))
      (if (and (equal? '< (car expr)) (not found))
        (if (equal? nested 0)
          (funparse (cdr expr) (+ nested 1) accum paren nil found)
          (funparse (cdr expr) (+ nested 1) before paren (push accum (car expr)) found))
        (if (and (equal? '> (car expr)) (not found))
          (if (equal? nested 1)
            (funparse (cdr expr) (- nested 1) before (funparse_ accum) nil t)
            (funparse (cdr expr) (- nested 1) before paren (push accum (car expr)) found))
          (funparse (cdr expr) nested before paren (push accum (car expr)) found))))))) ...)
```

0.14 The Parser

The parser now works as in the following example.

```
(letrec parse (lambda (...) ...)
  (parse '(< a >< b < c >>< d >)))
 ⟹ ((a) (b (c)) (d))
```

0.15 Evaluation

Combining the prior evaluator with the new addition of the parser, we have the behavior you would have expected.