On Interpretation & Bootstrapping

Matt Neary

August 25, 2013

Abstract

The interpretation of languages is the key to modern Computer Science. It is through interpretation that we are able to add layers of abstraction to computational procedures, concealing the details of implementation in favor of more expressive forms. In the following article, a language will be iteratively defined and interpreted using exclusively Mathematical constructs. Through this approach, we will remove all prerequisites from the introduction of language interpretation, and we will introduce the factors to consider when choosing a language for use in computation. From there, we will design a new foundation for our interpretation which we will bootstrap rather easily.

1 An Interpreter By Means of Algebra

1.1 Encoding a Language

In order for a language to be interpreted, it needs to be rendered in a format which is easily parsed. For example, in the case of natural language, we have arrived at a set of easily discerned phonemes of speech, and a specific set of letters for written language. Another example was first introduced by Gödel in his proof of the Incompleteness Theorem; his means of encoding was a numbering system known as Gödel Numbering. A Gödel Numbering system consists of a function which assigns to each symbol of a language a Natural Number. The following is an example of his numbering system in action.

$$\begin{aligned} & 2 \ + \ 3 \\ \implies & (22, \ 01, \ 23) \\ \implies & 2^{22} \ + \ 3^{1} \ + \ 5^{23} \\ \implies & \approx 1.19 \times 10^{16} \end{aligned}$$

As you can see, Gödel made use of a prime-factorization-dependent method for his encoding. That is to say, his encoding allowed for the original form to be recognized by means of factoring. This is a very robust means of encoding from a theoretical stand-point, an arbitrarily large number can be a token of the language to be encoded; however, as you have seen through our example, the encoded forms grow extremely rapidly in magnitude.

It is for this reason that we will opt to avoid his definition in our examples, instead opting for a limit to be placed on the magnitude of our tokens; for now we allow that limit to be 79. Our dictionary for the encoding of symbols follows.

$$\begin{aligned} + &\implies 01 \\ &\dots \\ 0 &\implies 20 \\ 1 &\implies 21 \\ &\dots \\ 79 &\implies 99 \end{aligned}$$

Note that we begin with a range dedicated to symbols, of which we have only defined one, +. Then, we translate each number to a number greater by 20, as to avoid the range of symbols. A string of these tokens will then be represented as a single number, composed of two-digit sequences corresponding to each token.

1.2 Evaluating an Expression

The example we will now choose to define will consist of addition of numbers and nothing else. There will be no sort of grouping or complex expressions, but merely the evaluation of addition represented by a three-token encoding.

We begin by defining some elementary functions to aid in the interpretation of our encoding. The naming scheme of *car*, *cdr*, *caddr*, *cddr*, and *cadr* is derived from Lisp; for the sake of this article, consider *car* to mean *head*, *cdr* to mean *tail*, and all derived forms to be the composition of these primitives suggested by their lettering.

$$\begin{aligned} cdr(x) &= \lfloor x/100 \rfloor^{1} \\ car(x) &= x - (100)cdr(x) \\ caddr(x) &= car \cdot cdr \cdot cdr \\ cddr(x) &= cdr \cdot cdr \\ cadr(x) &= car \cdot cdr \end{aligned}$$

Note that the *car* function gets what is considered the head value, otherwise the first two digits from the decimal point. Hence, we access them by subtracting the rounded-off higher values.

Now, we will begin by separating out the functionality of our evaluator into two componenets: reduction of an encoded numeral, and evaluation of an addition form.

$$\begin{aligned} number(x) &= x - 20 \\ eval(x) &= number(car(x)) + number(caddr(x)) \end{aligned}$$

The above should seem pretty straightforward, assuming that you recall the meaning of *car* and *caddr*. Simply put, the above defition of *eval* renders an addition form as the sum of its first token and its third token. For example, let's encode and evaluate the form $2 + 3$.

[1] This is known as the *floor* function or *greatest integer* function. It serves to reduce a fractional number to the next highest integer, e.g., $5.2 \implies 5$.

$$
\begin{aligned}
& eval(2+3) \\
\Longrightarrow\ & eval((22,\ 01,\ 23)) \\
\Longrightarrow\ & eval(230122) \\
\Longrightarrow\ & number(car(230122)) + number(caddr(230122)) \\
\Longrightarrow\ & number(22) + number(car(cdr(2301))) \\
\Longrightarrow\ & 2 + number(car(23)) \\
\Longrightarrow\ & 2 + number(23) \\
\Longrightarrow\ & 2 + 3 \\
\Longrightarrow\ & 5
\end{aligned}
$$

Of course, we achieved the desired result. However, we would now like to streamline the architecture of our evaluator. To do this, we will consolidate the *eval* and *number* function into one to allow for a more extensible function. We will be making use of a piece-wise function[2].

$$
eval(x) = \begin{cases} cdr(x) = 0, & x - 20 \\ cadr(x) < 20, & eval(car(x)) + eval(caddr(x)) \end{cases}
$$

1.3 A Language with Composition

Earlier, we defined an evaluator of a language with a single expression type: binary sums. However, we now wish to define an evaluator of nested addition. Rather than incorporate parentheses, we will make use of Reverse-Polish Notation (RPN) in our language. The following is an example of how this structure translates into traditional Mathematic notation.

$$
\begin{aligned}
& 2\ 3\ +\ 4\ +\ 1\ 6\ +\ + \\
\Longrightarrow\ & (((2+3)+4)+(1+6))
\end{aligned}
$$

In order to evaluate expressions of this form, we will introduce a second parameter to the function. Our addition will allow for the evaluator to manage a stack of values which it will render with addition. However, first we will define a function[3] for the creation of token-chains called *cons*.

$$
cons(x, y) = x\ +\ y(100)
$$

Note the relationships between *cons* and the *car* and *cdr* functions.

[2]Some different tactics were used in this translation. For example, rather than check the token type by a simple < 20, we checked whether there was a single token passed by means of $cdr(x) = 0$.

[3]This may be your first time seeing a function of multiple variables. Note that the mechanics remain the same, the primary difference being its rendering as a graph.

$$
\begin{aligned}
car(cons(x, y)) &= x \\
cadr(cons(x, y)) &= y \\
cdr(cons(x, cons(y, z))) &= cons(y, z) \\
cons(car(cons(x, y)), cdr(cons(x, y))) &= cons(x, y)
\end{aligned}
$$

Now we will define our new *eval* function accepting two arguments. The first argument remains the expression, but the second is a stack of values.

$$
eval(x, s) = \begin{cases} x = 0, & car(s) \\ car(x) < 20, & eval(cdr(x), cons(car(s) + cadr(s),\ cddr(s))) \\ car(x) \geq 20, & eval(cdr(x), cons(car(x) - 20,\ s)) \end{cases}
$$

Notice that once the entire expression has been read, the first value in the stack is returned. Otherwise, if the character is the addition operator, the rest of the expression is evaluated with the first two values on the stack added. Lastly, if a numeric value is read, the function recurses upon the rest of the expression with the number pushed to the stack.

1.4 A Language with Multiple Forms

We have already expanded out language to allow for arbitrary nesting of expressions. Now, we will set out to add the additional operators of Mathematics. First, we expand our dictionary of encoding.

$$
\begin{aligned}
+ &\implies 01 \\
\cdot &\implies 02 \\
- &\implies 03 \\
/ &\implies 04
\end{aligned}
$$

Now, we expand our piece-wise evaluator to account for these operators. The evaluator will be based on an *op* function which we define as follows.

$$
op(o, a, b) = \begin{cases} o = 1, & a + b \\ o = 2, & a \cdot b \\ o = 3, & a - b \\ o = 4, & a/b \end{cases}
$$

Now we incorporate *op* into our evaluator.

$$
eval(x, s) = \begin{cases} x = 0, & car(s) \\ car(x) < 20, & eval(cdr(x), cons(op(car(x), cadr(s), car(s)),\ cddr(s))) \\ car(x) \geq 20, & eval(cdr(x), cons(car(x) - 20,\ s)) \end{cases}
$$

With the evaluator defined, we will walk through another example.

$$
\begin{aligned}
& 2\,3\ \cdot\ 2\ +\ 2\ - \\
\Longrightarrow\ & eval((22,23,02,22,01,22,03),0) \\
\Longrightarrow\ & eval(03220122022322,0) \\
\Longrightarrow\ & eval(032201220223,02) \\
\Longrightarrow\ & eval(0322012202,0203) \\
\Longrightarrow\ & eval(03220122,06) \\
\Longrightarrow\ & eval(032201,0602) \\
\Longrightarrow\ & eval(0322,08) \\
\Longrightarrow\ & eval(03,0802) \\
\Longrightarrow\ & eval(0,06) \\
\Longrightarrow\ & 6
\end{aligned}
$$

We arrived at the correct result, and hopefully the methodologies of the *eval* function were made clearer.

2 Toward Bootstrapping

2.1 A More Accommodating Foundation

Notably, our means of encoding was not very natural and not very extensible. If we were to design our own foundation for the interpretation of languages, we would modify its structure. The main issue we would want to address would be our data-structure.

When operating within simple[4] Mathematics, we were forced to make use of a number for our data storage. However, in our own choice of foundations, we could instead create a primitive list type of our own. The list would again be interacted with by means of *car* and *cdr*, and constructable by means of *cons*; however, the elements of the list could be of any form, rather than a two-digit token. For example a list could be formed consisting of various lists.

Additionally, we could choose to form a language exhibiting *homoiconicity* to make it as easy as possible to create a *bootstrapped* interpreter. Homoiconicity is the trait of a language whose primary data-type is similar to its syntax. So for example, if we were to make lists the primary data-type, as we suggested earlier, we would want a syntax that was very list-like as well.

The language we have described in our prescription of traits is *Lisp*. In Lisp, every expression is a Symbolic Expression. A Symbolic Expression is of the following form.

[4] I emphasize that this issue was mainly present because of our dealing exclusively with simple Math. As you would expect, there exist the facilities for complex data-structures in Mathematics, but we chose to avoid them.

$$(1\ 2\ 3\ 4)$$

Nesting of Symbolic Expressions would then look like this:

$$(1\ (2\ (3)\ 4))$$

As you can see, this is a very clean way of defining complex trees of values, very much suited for the expression of *Syntax Trees* which will be interpreted. Hence, we would have the following clarity in our expression of the prior-defined langauge[5].

$$(-\ (+\ (\cdot\ 2\ 3)\ 2)\ 2)$$

With a few primitives at hand, we would be able to interpret this structure from within the language.

2.2 Language Primitives

We accept as primitive the following functions, with booleans, nil, and atoms available as atomic values.

1. *eq* - determine whether two values are equal.
2. *atom* - determine whether a value is atomic.
3. *cons* - form a list with a given head and tail.
4. *car* - access the head of a list.
5. *cdr* - access the tail of a list.
6. *cond* - evaluate a series of conditionals.
7. *lambda* - form a function accepting a list of parameters.
8. *label* - name a function.
9. *quote* - treat a Symbolic Expression as a literal.
10. $\#t$ - true.
11. *nil* or $\#f$ - nil.
12. *null* - determine whether a value is nil.

We begin by defining some perliminary functions in this language which we have just described. Note that an application of any one of these primitive functions is a Symbolic Expression led by their name. For example, to invoke *cons* upon 1 and 2, one would use the following expression.

$$(cons\ 1\ 2)$$

[5]I chose to represent the prior language now in *Polish* or Prefix Notation to foreshadow the language upon which we will later settle.

2.3 Prerequisite Functions

Now, we return to our definition of a prelude. The functions which we wish to expose to the evaluator are *set*, *apply*, *assoc*, and cond-list, with all other definitions being dependencies of these functions.

```
(label pair (lambda (e k)
  (if
    (null e)
    #f
    (if
      (eq (caar e) k)
      (car e)
      (pair (cdr e) k)))))

(label update (lambda (k v e)
  (if
    (null e)
    nil
    (if
      (eq k (caar e))
      (cons (cons k (cons v nil)) (cdr e))
      (cons (car e) (update k v (cdr e)))))))

(label set (lambda (k v e)
  (if
    (pair k e)
    (update k v e)
    (push (cons k (cons v nil)) e))))

(label assoc (lambda (e k)
  (if
    (pair e k)
    (cadr (pair e k))
    nil)))

(label apply (lambda (f args)
  (if
    (null args)
    f
    (apply (f (car args)) (cdr args)))))

(label cond-list (lambda (conds env)
  (if
    (null? conds)
    #f
    (if
      (eval (caar conds) env)
```

```
      (eval (cadar conds) env)
      (cond-list (cdr conds) env)))))
```

2.4 An Evaluator

With this prelude of functions defined, we now are able to define an evaluator of Lisp, from within Lisp.

```
(label eval (lambda (x env)
  (cond (((atom x) (assoc env x))
         ((eq (car x) (quote lambda))
          (if
            (null (cadr x))
            (eval (caddr x) env)
            (lambda (x)
              (eval
                (cons
                  (quote lambda)
                  (cons
                    (cdadr x)
                    (cons (caddr x) nil)))
                (set (caadr x) x env)))))
         ((eq (car x) (quote cond))
          (cond-list (cadr x) env))
         ((eq (car x) (quote label))
          (set (cadr x) (eval (caddr x) env) env))
         ((eq (car x) (quote quote))
          (cadr x))
         (#t (apply (eval (car x) env) (cdr x)))))))
```

The above defined the core constructs of the language, like *lambda* and *label*; however, it neglected the simpler functions, like *car*. In the following, we provide a function which will expose these aspects of the language by means of the environment value.

Note that in the following, we simply form an environment constituent of the various functions we wish to expose to the programmer from the beginning, like *car* and *cdr*, and then provide it to the *eval* function already defined.

```
(label ev (lambda (x)
  (eval x
    (cons (cons (quote car) (cons car nil))
          (cons (cons (quote cdr) (cons cdr nil))
                (cons (cons (quote cons) (cons cons nil))
                      (cons (cons (quote atom) (cons atom nil))
                            (cons (cons (quote eq) (cons eq nil))
                                  nil)))))))))
```

Finally, we are ready to evaluate Lisp expressions from within Lisp. The following is an example.

$$\begin{aligned} & (ev\ (quote\ ((lambda\ (x)\ (car\ x))\ (cons\ \#t\ nil)))) \\ \Longrightarrow\ & \#t \end{aligned}$$

2.5 The Value of Bootstrapping

Bootstrapping can seem an extremely esoteric and useless matter; however, once a language has been bootstrapped, features may be added to the language from within. To put that more strongly, the syntax of a language can be completely changed or simply expanded from within that same language.

Even better, this process can be repeated iteratively. Of course, this practice would be better suited for a bootstrapped compiler[6], but regardless, you could form an interpreter adding a feature to your language, and then interpret a new interpreter building upon that feature.

3 Last Words & Lambda Calculus

We began by defining a basic interpreter in Algebra alone. Through this project, we learned about the encoding of language, recursive nature of evaluators, and the need for appropriate data-structures. With these things in mind we went on to define a Bootstrapped Interpreter of Lisp. However, these lessons would be applicable in Interpreting or Bootstrapping any language. For example, if you wish to Bootstrap the Lambda Calculus, you will need an appropriate data-structure, most likely a list; and a means of encoding.

Reflecting briefly on these ideas, we would have some of the primitives that follow, along with the named encodings denoted by a quotation-marked character.

$$\begin{aligned} 0 &= \lambda f \lambda x x \\ succ &= \lambda n \lambda f \lambda x (f)((n)f)x \\ 1 &= (succ)0 \\ &\cdots \\ ten &= (succ)9 \end{aligned}$$

[6] A compiler translates a program written in a given language to a machine understandable procedure. Usually, this takes the form of binary encodings of processor instructions.

$$
\begin{aligned}
+ &= \lambda a \lambda b((a)succ)b \\
* &= \lambda a \lambda b((a)(add)b)0 \\
\# &= \lambda a \lambda b((+)((*)ten)a)b \\
\lambda\text{"} &= ((\#)0)1 \\
)\text{"} &= ((\#)0)2 \\
(\text{"} &= ((\#)0)3 \\
a\text{"} &= ((\#)1)0 \\
b\text{"} &= ((\#)1)1 \\
&\dots \\
z\text{"} &= ((\#)3)6 \\
cons &= \lambda a \lambda b \lambda f((f)a)b \\
car &= \lambda p(p) \lambda a \lambda b a \\
cdr &= \lambda p(p) \lambda a \lambda b b
\end{aligned}
$$

The design decisions involved in the *atom* and *eq* aspects are not necessary in understanding how our prior topics would translate into a boostrapping of the Lambda Calculus; we have successfully devised a scheme for encoding tokens and forming chains of these tokens. From here, we need only parse the parenthetical expressions, which is an entirely separate task, and then evaluate based on the simple rules of the Lambda Calculus.

Note that once the Lambda Calculus has been bootstrapped, its syntax may be expanded. This is especially enticing considering apparent density of its syntax. For example, one could pretty easily implement Lisp style lists in the Lambda Calculus once it has been boostrapped. This would then make writing an interpreter far easier. As was described earlier, this process of iterative feature addition is the foundation of modern Computer Science.