On Interpretation

Matt Neary

August 11, 2014

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.

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.

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

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.

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

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 $2 + 3$.

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

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.

[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., $5.2 \implies 5$.

[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 $cdr(x) = 0$.

$$\begin{gathered} 2\;3\;+\;4\;+\;1\;6\;+\;+ \\ \implies (((2+3)+4)+(1+6)) \end{gathered}$$

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*.

$$\mathit{cons}(x,y) = x\ +\ y(100)$$

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

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

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.

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

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.

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

[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.

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. Finally, we will now prove that the interpreter does its job: That it accurately evaluates all expressions of the language's grammar.

1.5 Verification

Our first step is to define a variation on *eval*, called $eval'$. This altered form will instead bear the full stack when finished evaluating.

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

We will begin with a related proof. We wish to show that for all expressions e in the grammar, $eval'$ bears a stack with the evaluation of e at the top.

Proof. To begin, we wish to show that for all stacks s, $n \in [0, 79]$ and e such that e is the expression representation of n, $eval'(e, s)$ bears a stack with n at the top. We know that $e = n + 20$, and thus have that $eval'(e, s) = eval'(n+20, s)$. Now, since $n+20 \geq 20$, we have that $eval'(e, s) = eval'(0, cons(n+20-20, s)) = cons(n, s)$.

Now, given two expressions e_1 and e_2 such that their evaluations bear a stack with their values at the top, we wish to show that for all stacks$s$, $eval'(cons(e_1, eval'(e_2, o)), s) = op(o, car(eval'(e_1, s)), car(eval'(e_2, s)))$. This follows trivially from the inductive hypothesis and definition of $eval'$.

We now conclude that for all expressions e in the language and stacks s, $eval'(e, s)$ bears a stack with the value of e at the top.

□

Using the above proof, we declare that for all expressions e in the language and stacks s, $eval(e, s)$ bears the value of e.