A Small Interpreted Language

What would you need to build a small computing language based on mathematical principles? The language should be simple, Turing equivalent (i.e.: it can compute anything that any other language can compute) and relatively easy to use. Assume the computing hardware is constrained to vonNeumann processes, with memory, an ALU, and appropriate registers. We will also assume that we know about formal mathematical languages and the necessary mathematical pieces: representation, recognizer, constructor, accessor, invariants/facts, functions, and induction/recursion.

Base *Representation* of Atoms

First, the *alphabet* of a language is simply a collection of unique identifiers, called *atoms*. The essential memory management trick is to divide each memory cell into two parts, an address part (call it **First**) and a contents part (call it **Rest**). Addresses are also called *pointers*. We begin with an array of empty cells, each having some empty representation in both the **First** and the **Rest** parts. This is the *free list* of memory cells.

The ground: We need an atom which means nothing, the null atom. Call it **nil**.

The symbol table: This table consists of a collection of non-empty memory cells, one cell for each atom in the language. The **First** part of an atom cell contains nil. The actual literal representation of the atom is in the **Rest** of the cell. The symbol table is a dynamic array.

Constructor of Compound Expressions

We need to construct compound expressions. Consider an expression which uses two atoms, say FOO BAR. The symbol table contains each atom, so all we need is a way to connect them. This can be done simply by building another memory cell which contains the two addresses of FOO and BAR. We put all atom addresses in the **First** part of a cell (see cell 005 below) and connecting addresses in the **Rest** part. The instruction to build connecting cells is called **Cons**. The end of an expression has **nil** in the **Rest**.

If we build the expression (TRUE BAR TRUE FOO) in cell 007, memory would look like this:

Address	First	Rest	
000	nil	nil	symbol table
001	nil	FOO	
002	nil	BAR	
003	nil	BAZ	
004	nil	TRUE	end of symbol table
005	001	006	the expression (FOO BAZ)
006	003	000	end of expression
007	004	008	the expression (TRUE BAR TRUE FOO)
008	002	009	
009	004	010	
010	001	000	end of expression
011	empty	empty	begin free list
...			

To construct an expression, we **Cons** smaller pieces together. For instance:

```
Cons JOHN (TRUE BAR TRUE FOO) ==> (JOHN TRUE BAR TRUE FOO)
```

The operational memory changes are:

```
011         nil         JOHN        the atom JOHN
012         011         007         connect JOHN to (TRUE BAR TRUE FOO)
013         empty       empty
```

Consider **Cons**ing two compound expressions together:

```
Cons (FOO BAZ) (TRUE BAR TRUE FOO) ==> (FOO BAZ TRUE BAR TRUE FOO)
```

This operation is slightly more complex. For the entire expression to begin in cell 012, we need memory to end up as

```
011         003         007               (BAZ TRUE BAR TRUE FOO)
012         001         011               (FOO BAZ TRUE BAR TRUE FOO)
013         empty       empty
```

Several design decisions are involved with this result. Technically, we have used *structure sharing* for (TRUE BAR TRUE FOO) since both the original four atom expression and the final six atom expression use some of the same memory cells. However, the front of the expression, (FOO BAZ) is not engaged in structure sharing, and this may seem a little unsymmetrical. As it is, (TRUE BAR TRUE FOO) is confounded with **Rest** (**Rest** (FOO BAZ TRUE BAR TRUE FOO)).

An alternative which would allow us to continue to refer to the original would be to duplicate the four atom expression entirely in constructing the six atom expression.

Note also that the construction is slightly different, rather than adding a symbol cell, as in the case of JOHN, we have added a *cons cell*. To acknowledge these differences, we might consider **Cons** of two compound expressions to be a different operation. Call it **Append**. Now the first object in a **Cons** operation is restricted to be an atom. **Append** is used when the first object is compound. To keep the language simple, we would want to be able to build new operations out of the existing ones. For this, we use a recursive definition:

```
Append <obj1> <obj2> =def=
  If Isa-atom <obj1>
    then ERROR
    else if Is-empty <obj1>
      then <obj2>
      else Cons (First <obj1>)(Append (Rest <obj1>) <obj2>)
```

This recursive definition first does a *type-check* on <obj1>. It then tests the *base case*, that <obj1> is **nil**. **Append**ing nothing onto <obj2> results in <obj2>. Otherwise we proceed one piece at a time. The recursion bottoms-out when **Rest** <obj1> is **nil**. For this to be the case, <obj1> must have only one atom, as in (BAZ), which is **Cons**ed onto <obj2>. At that time, BAZ is the **First** of <obj1>. Just prior to this case, <obj2> is actually (BAZ TRUE BAR TRUE FOO), since we have **Cons**ed BAZ to (TRUE BAR TRUE FOO). <Obj1> is (FOO BAZ), and we are about to **Cons First** <obj1>, i.e. FOO, onto (BAZ TRUE BAR TRUE FOO).

This description has backed up from the end to the beginning. Tracing the events in memory:

```
Append nil (TRUE BAR TRUE FOO) ==> (TRUE BAR TRUE FOO)

011          000          007          Append nil
012          empty        empty        begin free list
```

By definition, cell 011 is the same as 007, so operationally this step is not necessary to take. We leave 011 free, treating **Append**ing **nil** as a *no-op*.

```
Cons BAZ (TRUE BAR TRUE FOO) ==> (BAZ TRUE BAR TRUE FOO)

011          003          007

Cons FOO (BAZ TRUE BAR TRUE FOO) ==> (FOO BAZ TRUE BAR TRUE FOO)

012          001          011
013          empty        empty
```

What we have done here is to specify exactly the sequence of operations on memory that result in the action of **Append**ing. And we have used the single construction tool of **Cons**.

This example illustrates the close connection between a software program, the attendant changes in memory, and the hardware architecture which unites both.

Recognizer of Atoms

The *recognizer* of each atom is a function which looks in the symbol table for the memory cell which contains that atom. For instance, the predicate **Isa-atom** is true if its argument can be found in the **Rest** portion of the symbol table. At this point, we have three separate memory areas (or uses): *free cells*, *atom cells*, and *cons cells*.

Isa-atom: Atom cells are recognized by having **nil** in the **First** part.
Is-empty: Empty expressions can be uniquely recognized because they have **nil** in the **Rest** part.
Equal: Tests if two atoms are the same atom.
Isa-expression: **Cons** cells are recognized as those cells having two addresses. An expression ends with **nil** in the **Rest** part.

The above are close to operational definitions. Here are some slightly more elaborated operational definitions. We will assume that each part of a memory cell (address, first, rest) has eight bits.

Is-empty <obj>:

Assign **nil** a special binary code, 00000000, and put it in address 00000000.
An object is empty, that is, it is equal to **nil**, if the **Rest** part is equal to the code of **nil**.
To distinguish **nil** from an empty cell on the free list, we could put a special code in free list cells, perhaps 11111111. A better approach is to use only seven bits of the address for address information, and use the eighth bit for marking if a cell is free. This is the basis for many *garbage collection* algorithms.

```
Is-empty <obj> =def= Equal (Rest <obj>) 00000000
```

Isa-atom <obj>:

Test the encoding of <obj> against all the encodings in the **Rest** part of memory which also have **nil** in the **First** part.

```
Isa-atom <obj> =def= for some memory cell
  (Equal (First <obj>) nil) and (Equal (Rest <obj>) <obj>)
```

Here is another *design choice*: is **nil** an atom or not? If it is not an atom, we will have to have special tests for atoms vs **nil**. For simplicity, let's say it is an atom:

```
(Isa-atom nil) is True
```

This design choice is our first *fact*, or invariant.

More generally:

```
Isa-atom (Is-empty <obj>) =def=
      True iff (Is-empty <obj>) is True
```

Recognizer of Expressions

We can use the instructions **First** and **Rest** to access and decompose all expressions. (**First** <obj>) looks at the first part of memory for the specific object, (**Rest** <obj>) looks at the rest part.

To recognize compound expressions, we test to see if each part of that expression is in the memory table, and the linking structure of the expression matches the rules for constructing that expression. Operationally:

```
Isa-expression <obj> =def=
  (Isa-expression (First <obj>)) and
  (Isa-expression (Rest <obj>))
```

Since we know that decomposing an expression will end in either atoms or nil, we will have to add those rules:

```
Isa-expression (Isa-atom <obj>) =def=
  True iff (Isa-atom <obj>) is True

(Isa-expression nil) is True.
```

This is another application of recursive decomposition. The rules specify the base cases, while the definition specifies the general recursive case. The two together specify a program.

The definition above is another example of *pseudo-code*, that is, machine specific instructions written in a mathematical style that is independent of the specifics of any programming language, yet specific enough to be implemented in any language. Of course, a *high level programming language* accepts something very close to pseudo-code specification as valid input. Another strong advantage of pseudo-code is that it can be proven to be correct using the Induction Principle.

The primary reasons that current programming languages appear to be very different than pseudo-code are

1. Many programming tasks lack a formal model (i.e. they are hacks).

2. Many programming languages lack mathematical structure (i.e. they are machine architecture specific.)

Accessors of Atoms and Expressions

First and **Rest** are the accessors. They let us take apart an expression. In this implementation, **First** and **Rest** have simple mappings onto the idealized physical structure of memory.

All objects except **nil** are constructed by **Cons**. Since **Cons** uses two objects as arguments, this means that all **First** and **Rest** parts are also objects. Eventually all objects end in **nil**, so **nil** is also an object, although a very special kind.

Cons is related to **First** and **Rest** by the following invariant, or rule:

```
<obj> = Cons (First <obj>) (Rest <obj>)
```

This says that all valid objects have been constructed by **Cons** to have a **First** part and a **Rest** part in memory. Alternatively, all objects in memory can be accessed through their **First** and **Rest** parts. The essential mathematical condition is that all valid objects are decomposable into unique subcomponents which bottom-out at the base cases. This is simply to say that all compound expressions are defined recursively.

Although recursive composition and decomposition are necessary to define data structures and algorithms, the more important aspect of recursive definition is to provide access to proof through the Induction Principle. Procedural languages do not provide this capability; they are thus immature. Declarative, functional, and mathematical programming languages all provide the capability of abstract proof (minimally in pseudo-code).

Note that recognizing, constructing, and accessing an expression involve almost the same steps. The difference is in the initial goal and the final result.

	GOAL	PROCESS	RESULT
Constructor:	build a pattern	rearrange memory	the pattern is in memory
Recognizer:	test a pattern	access memory	true if the pattern is accessible
Accessor:	get a pattern	access memory	return the pattern if found

SUMMARY of the ABSTRACT DATA STRUCTURE FUNCTIONS

First <obj> returns the expression indicated by the **First** of the <obj>

Rest <obj> returns the expression indicated by the **Rest** of the <obj>

Is-empty <obj> returns True if the cell containing <obj> has **nil** in **Rest**.

Isa-atom <obj> returns True if the <obj> is in the **Rest** part of a cell and **nil** is in the **First** part.

Isa-expression <obj> returns True if the <obj> has either **nil** or any address in the **First** part.

Equal <obj1> <obj2> In the case of atoms, returns True if both objects are in the **Rest** of the same symbol cell. In the case of compound expressions, returns True if following the addresses in the **First** leads to the same set of **Rest** symbol cells.

Cons <obj1> <obj2> builds an expression by adding <obj1> to the front of <obj2>

Invariants

The *equality invariant* (also called the Uniqueness Axiom) assures that each object is unambiguous. That is, objects are the same object when they are equal; equal objects are constructed and deconstructed in exactly the same way. This is a physical kind of equality, *structural equality*, in that the structure of memory is the same for two objects. It is not necessary that the same memory cells are used for both objects (structure-sharing), just that the contents of memory for both objects are the same. Recursively,

```
Equal <obj1> <obj2> =def=
  (Equal (First <obj1>) (First <obj2>)) and
  (Equal (Rest <obj1>) (Rest <obj2>))
```

We need to support this definition with base cases. For instance,

```
(Equal nil nil) is True
```

This is also an example of the *Induction Principle* at work in our implementation. To implement an equality test for expressions, the computation will test for identical structure over all memory cells of both objects. The Induction Principle is the only guarantee that this recursive process will end. The only end point is (**Equal nil nil**), all other cases are failures.

Note that equality for atoms is also covered in the above definition. What happens, though, when we have two atoms which have the same encodings, but each is in a different memory cell? This is an inconvenience for an implementation, since testing each object would require looking through the entire symbol table. A better approach is to insist that each atom is unique and occurs only once in the symbol table. This is why Equality and Uniqueness are the same ideas.

The uniqueness of atoms is implemented by having each new atom *register* itself in the symbol table. In the background, when an unrecognized, new atom is entered, the implementation verifies that it is new, and then puts it in the symbol table. To do this is to *intern* the atom. If the atom already exists, then the address of the cell which contains that object is associated with the new input.

A different kind of equality refers to equality under transformation. The actual expressions may be different, but transformation rules allow us to say that the meaning of the different expressions is the same. This is *semantic equality*, also called *algebraic equality* and *mathematical equality*. Only defined transformations are allowed; all transformations (with the exception of **Cons**) are required to keep meaning consistent. It takes a special *symbolic architecture* to implement mathematical equality, mainly because transformations refer to sets or classes of objects rather than to specific objects. In the above, we have designed a *literal architecture*, as yet it has no capacity for dealing with sets of objects.

Now on to the functional part of the language. We will elect to use *lambda calculus* as the mathematical model.

Functions and *Recursions*

A function is an expression with the function name first and then the arguments. (The order of operators and arguments is somewhat arbitrary, just so long as it is consistent and unambiguous.) For example:

```
+ 3 4
```

The Arithmetic Logic Unit (ALU) can process logical and arithmetic operators when applied to atoms. Internally, both arithmetic and logic are encoded by binary sequences, so it is the responsibility of the operator, or of a *type test*, to make sure that expressions meant to represent numbers are channeled to the arithmetic units and expressions intended to represent logic are channeled to the logic units.

One way to implement the difference between logic and arithmetic is to assign another single bit in the memory cell that records the type of object in that cell. Note that silicon gates process only logic. Thus arithmetic objects must be encoded into a logical form for processing. In computation, *logic is fundamental*, arithmetic is derivative.

All logic functions can be defined in terms of a single function, so we need only one primitive logic function. Let's use **IfThenElse** (**Nand** and **Nor** are alternatives).

```
IfThenElse <obj1> <obj2> <obj3>
```

IfThenElse will evaluate <obj1> and then either evaluate <obj2> (if <obj1> is True) or evaluate <obj3> (if <obj1> is False). Here we have another function which uses different types of objects (the first example was **Cons**). In particular, <obj1> must be a logical type, returning either True or False.

Function composition permits complex sequences of operations. A function expression can be put in any place that an atom can be put, since all functions will reduce to single atoms. To separate composed functions, we can use parentheses to contain each function expression. We will choose to evaluate all inner arguments first, then use these results to evaluate outer

functions. Lambda calculus permits another order of evaluation, outermost first. This choice is a design decision, and is based on mathematical characteristics of each form of evaluation.

For example, an innermost evaluation:

```
(* (+ 3 4) (+ 1 2)) or ((3 + 4) * (1 + 2))
```

means that expressions with atoms as arguments are evaluated, or *reduced*, first.

The memory for this object would look like this:

Address	First	Rest	
000	nil	nil	symbol table
001	nil	1	
002	nil	2	
003	nil	3	
004	nil	4	
005	nil	+	
006	nil	*	end of symbol table
007	006	008	expression ((3 + 4) * (1 + 2))
008	005	009	
009	003	010	
010	004	011	
011	005	012	
012	001	013	
013	002	000	end of expression
014	empty	empty	begin free list
...			

There are several things to note about the above memory configuration.

Operators and numbers are not distinguished in memory, they are distinguished by what happens when they are handed to the ALU.

Each operator has two arguments, but we have no way to have two references in one memory cell. The solution is to order the expression so that operators are followed by their arguments. When an operator is fetched for evaluation, the machine code recognizes that that operator requires two more fetches. Should a fetch return another operator, then the first operator waits until the second operator converts its two arguments into one result.

Fetches occur by following the addresses in sequence. This is efficient since the address register (the register which keeps track of what to fetch next) need only be decremented by one to find the next memory cell.

It is possible to turn all functions into one argument functions (the technique is called *currying*). This is effectively what has happened by storing the expression in the *operator first* form (also known as reverse Polish notation).

Finally, consider how close the syntax of many programming languages is to what actually happens at the *register transfer level* of the computer. We are still at the very early stages of development of computing languages, since the syntax reflects low level data shuffling rather than high level task requirements. Progress means moving our profession toward human capabilities, and moving away from low level machine details.

We need a way to define arbitrary functions and a way to bind the variables of functions to values for the ALU to process. For example

Square <obj> =def= (* <obj> <obj>)

so that **Square** 4 => (* 4 4) => 16

First consider variables, names which stand for any valid object. We have been using the names <obj1>, <obj2>, etc. as variables names. The angle brackets notate that the name in question is not the name of a single thing, but rather it is the name of a class, or *set*, of things, all of which are of a particular *type*.

Variables (or *parameters*, when the names are arguments of a function) are atoms also, so they are simply added to the symbol table. To assign a value to a variable symbol, we can put a reference to the location of the value we wish to associate with the variable in the **First** part of the memory cell for the variable. Thus variables are distinguished from objects representing a specific value because their **First** part is not **nil**. It is an error to access a *variable* which has **nil** as the **First** part. Objects which do have **nil** in the **First** part are called *ground objects*.

The function which assigns ground objects to variable objects is called **Let**.

We can use this same mechanism to store the definitions of functions. The memory cell which contains the name of the function in the **Rest** part can contain the address of the definition of the function in its **First** part. Consider the memory configuration for the above definition of Square:

Address	First	Rest	
000	nil	nil	symbol table
010	nil	OBJECT	
011	nil	*	end of symbol table
012	013	SQUARE	function definition
013	011	014	
014	010	015	
015	010	000	end of function definition

When the call **Square** 4 is added to memory we get:

016	nil	4	symbol table
017	012	018	function call (Square 4)
018	016	000	

To bind OBJECT to the value 4, we use the call **Let** object 4:

019	nil	LET	symbol table
020	019	021	function call (Let object 4)
021	010	022	
022	016	000	

Finally we need to get the processor to actually evaluate the function call. Let's call this **Eval**. We can actually make **Eval** the default. Whenever a new expression is added it can be automatically evaluated. This just shifts the issue to needing an instruction to stop evaluation. Let's call this evaluation stopper **Quote**.

What the above memory configuration contains is **Quote** (**Square** 4), which simply puts the *data structure* **Square** 4 into memory. If we write the *function* **Square** 4, then evaluation will happen automatically. This process consists of changing the value of object from **nil** to 4, and following the sequence until a single atom is returned. That is, the function **Let** says to the processor: go to the symbol which immediately follows **Let** and put the address of the second symbol which follows **Let** (i.e. 4) in its **First** part. This results in

```
010          016          OBJECT
```

Now the definition of **Square** will find the value of OBJECT and use it rather than using the symbolic variable "OBJECT". And, of course, symbolic variables are the only items in the symbol table which can contain something other than **nil**.

There is a slight problem here because the symbol "OBJECT" might be used in more than one function. This can be handled in one of two ways:

1) make sure all of the symbols are unique, or

2) divide the symbol table into subtables which associate and isolate each function with its own variables.

Finally, we simply use *recursion* directly as repeated actions of the same sort, since nothing in the above structuring stops this from working.

The *Function* Eval

In the above description, evaluation is an implicit action of the ALU. By claiming evaluation is automatic, we are committed to wiring the ALU in a specific way. However the above mechanism for handling memory can be made flexible by *defining **Eval** in the programming language itself*. This process is called *meta-circular evaluation,* cause it uses a language itself to define how that language should behave. All we have to do is to define the evaluation function by telling the system what to do when an expression is typed in. The function **Eval** takes two arguments, the expression to be evaluated and the *binding environment*, that is, an address of the memory array which contains all of the primitive functions and atoms (and any other symbols which we may have added) in the language. The binding environment contains the definitions of all user defined functions, and the values of each of the variables (function arguments).

Since the binding environment does not change in this example, (i.e. we have not designed the language to establish separate environments for each function call), we will treat the token **Eval** to mean "Eval-in-environment"

The definition of **Eval** which follows uses only primitive functions introduced above. Some of the syntax has been changed to make it more readable.

This **Eval** function recognizes seven operators:

First	Rest	Cons	
IfThenElse	Equal	Quote	Let

In addition, **Eval** uses built-in tests to determine the types of objects, as operationalized above.

Is-empty	Isa-atom	Isa-expression

```
Eval exp  =def=

If Isa-atom exp
 Then                                                    ;process atom
   If ((Is-empty exp) or (Equal exp (Quote True)))
    Then                                                 ;return the SYMBOL
      exp
    Else                                                 ; or its VALUE
      Get-value-in-env exp
 Else                                                    ;process expression
   If Isa-atom (First exp)
    Then                                                 ;process Atom in First*
      Let token (First exp)                              ;naming the atom
        If Equal token (Quote Quote)
         Then                                            ;return what follows
           Second exp
         Else                                              ;other operators
           If Equal token (Quote IfThenElse)
            Then                                         ;process logic operator
              EvalLogic (Rest exp)
            Else                                           ;other operators
              If Equal token (Quote First)
               Then                                      ;First of Eval of Rest
                 First (Eval (Second exp))
               Else                                        ;other operators
                 If Equal token (Quote Rest)
                  Then                                   ;Rest of Eval of Rest
                    Rest (Eval (Second exp))
                  Else                                     ;other operators
                    If Equal token (Quote Isa-atom)
                     Then                                ;Isa-atom Eval of Rest
                       Isa-atom (Eval (Second exp))
                     Else                                  ;other operators
                       If Equal token (Quote Cons)
                        Then                             ;Cons Eval of Rest**
                          Cons (Eval (Second exp))
                               (Eval (Third exp))
                        Else                               ;other operators
                          If Equal token (Quote Equal)
                           Then                          ;Equal Eval of args
                             Equal (Eval (Second exp))
                                   (Eval (Third exp))
                           Else                          ;replace the token with
                             Eval (Cons                  ;its value
                                    (Get-value-in-env token) (Rest exp))
    Else                                                   ;compound First
      If Isa-expression (First exp)
       Then                                              ;process expression
         EvalExp exp
       Else ERROR
```

```
EvalLogic exp  =def=

If Equal (Eval (First exp)) (Quote True)          ;if First is TRUE
 Then                                             ;Eval second argument
   Eval (Second exp)
 Else                                             ;Eval third argument
   Eval (Third exp))


EvalExp exp   =def=

If Is-empty exp                                   ;if at the end
 Then                                             ;return ground
   nil
 Else                                             ;Eval the parts
  Cons (Eval (First exp)) (Eval (Rest exp))       ; and put them together
```

Notes:

* process Atom in First: Here we have defined a syntax for parsing. Every expression begins with an atom or is an atom. If an expression begins with an atom, that atom is taken by the processor to be an operator, and thus a processing instruction. The operator **Quote** is the no-op.

** Cons Eval of Rest: This is again a syntax constraint. Once we have removed the beginning operator of an expression, what follows is either an atom, or another expression which itself begins with an atom operator.

The syntax of the language is thus:

```
Expression ::= Atom | (Atom Expression*)
```

The Kleene star means that an operator atom can have any number of following arguments. Note that this BNF specification is one of a *regular language*.

Finally, note that a meta-circular language can evaluate its own definition of **Eval**, since the above definition is self-consistent.

The Punch Line

The above programming language actually exists, it is one of the very few oldest programming languages still in active use. Its name is *LISP*.

In 1955, John McCarthy followed a similar line of reasoning in developing LISP. Currently LISP stands uniquely among programming languages in that it is rigorous, efficient, largely machine independent, and permits simulation of all other programming language models (such as procedural, functional, object-oriented, logical, and mathematical). As well, when the function **Eval** is processing input, LISP is *interpreted*, responding dynamically to new inputs and definitions, and requiring no compilation or linking. It thus provides a powerful interactive programming environment which supports real-time debugging and symbolic proof of correctness.