Project Naurk: Lecture 1

Ramkumar Ramachandra

February 7, 2013

The objective of this course is to teach students how to think precisely. We will now answer the question in plain English, and then go on to successively replace the English constructs with *expressions* that a machine can *evaluate*

- A number whose only factors are one and itself
- A natural number that is not a multiple of any natural number except 1 and the number itself
- A `natural number` that is `<not a multiple of>` any `natural number` except 1 and `the number itself`
- (`natural number` `<not a multiple of>` any `natural number` except 1 and `the number itself`)

Problem 1: Assert if 27 is a prime number. Start off by specializing the previous expression to fit this problem statement

- (natural number <not a multiple of> any natural number except 1 and the number itself)

- (27 <not a multiple of> any natural number except 1 and 27)

Let us formalize our talk, and call the item above an *expression*. The first property of an expression is that it can be *evaluated*.

Notice how the following expressions are similar. All of them can be evaluated to produce a value.

- (something <not a multiple of> something else)

- (3 + 5) evaluates to the value 8

- (<square> 5) evaluates to the value 25

Consider the expression (m is a chocolate). This can evaluate to either **True** or **False**, depending on whether or not m is a chocolate. Set our initial problem aside for a moment and look at the following expressions:

1. (m is a chocolate), where m could be any item from a mixed bag of goodies
2. (any (m is a chocolate)), where m is an item from a mixed bag of goodies
3. (all (m is a chocolate)), where m is an item from a mixed bag of goodies

What each of these mean to a machine:

1. Pick any (random) item from the bag, and assert that it's a chocolate
2. Exhaustively pick items from the bag, and assert that any of them is a chocolate
3. Exhaustively pick items from the bag, and assert that all of them are chocolates

What we're doing: To find out the nature of the items in the bag, we have designed a random experiment where we pick up and test items from the bag. To be able to conclusively state that all the items in the bag are chocolates, we keep repeating procedure 1 until we've exhausted all the items. Hence 3 is just a restatement of 1, when we want a conclusion. 2 addresses a different problem.

Back to our original problem. To assert if 27 is prime. Notice the remarkable similarity between these two statements. We have actually expressed our algorithm as a random experiment!

1. (m is a chocolate), where m could be any item from a mixed bag of goodies

2. (27 <not a multiple of> any natural number except 1 and 27)

To instruct the machine more concretely, let us rephrase it to get a conclusive result. After all, we do conclusively want to know whether or not 27 is prime.

1. (all (m is a chocolate)), where m is an item from a mixed bag of goodies

2. (all (27 <not a multiple of> m)), where m is a natural number not 1, not 27

(all (27 <not a multiple of> m)), where <mark>m is a natural number not 1, not 27</mark>

Now observe the following expressions:

- (27 <not a multiple of> 6)
- (27 <not a multiple of> m)

In the first expression, we have used the *literals* 27 and 6 to represent their corresponding integer values. In the second, we have used the *symbol* m to represent an arbitrary value.

What do I mean when I say "value"? It could be a single number, a *set* of decimal numbers, a *list* of complex numbers, or just about anything else. However, an expression like (27 <not a multiple of> [complex number]) doesn't make much sense.

- m is a *symbol* used to represent *one value* in a *range* of integer values. We can rewrite it as m ∈ [1, 2, 3 ..] - [1, 27] = [2, 3 .. 26, 28 ..]

  (all (27 <not a multiple of> m); <mark>m ∈ [2, 3 .. 26, 28 ..]</mark>)

Every programming language needs to provide features for data representation. Data can be represented by literals, or equivalently, bound symbols. Simple data types provided by common programming languages include integers, fixed point decimal numbers, floating point (decimal numbers), fractions, complex numbers, and characters. Many come with composite data types like lists, strings, hashtables, and vectors.

(all (27 <not a multiple of> m) ; m ∈ [2, 3 .. 26, 28 ..])

Here, 27 and m clearly represent integers. If m were *bound* to a decimal number instead, this expression wouldn't make much sense, and the machine would throw up. We will see the mechanism for this in the next slide.

(all (27 <not a multiple of> m); $m \in [2, 3 .. 26, 28 ..]$)

What does 35*x mean to a machine? Several definitions need to be in order:

- * is an *operator* defined to act on two *operands*
- The symbol x could be *bound* to any value. To bind, we can use another operator $=$, and say (x $=$ 4) to bind x to 4.
- More restrictions are required to be imposed on *: It is defined only when both its operands are *numbers* (integers, decimals or complex). So, an expression like (3*"ram") is meaningless[1]

<not a multiple of> is a *function*[2]

(**all** (27 <**not a multiple of**> m); m $\in$ [2, 3 .. 26, 28 ..])

---

[1] unless ofcourse we define such operations explicitly
[2] we have conveniently omitted the discussion of $\in$ and **all** for the moment

(**all** (27 <**not a multiple of**> m); m $\in$ [2, 3 .. 26, 28 ..])

Time to observe. Notice how we've progressively stripped out the English in favor of expressions. Now look at each one individually:

1. (27 <**not a multiple of**> m)
2. (**all** [expression in 1])
3. [expression in 2]; m $\in$ [2, 3 .. 26, 28 ..]

Observe the general forms:

1. ([value] [function[3]] [value]). <**not a multiple of**> is clearly an function of two *parameters*[4]
2. ([function] [expression]) => ([function] [value]) after evaluation of the expression
3. [expression] along with some additional data

---

[3]the term function encapsulates operators as well
[4]operators:operands = functions:parameters

(all (27 <not a multiple of> m) ; m ∈ [2, 3 .. 26, 28 ..])

From the observations on the previous slide, there are functions of two parameters like <not a multiple of>, as well as functions of one parameter like <square>. More generally there are functions of n parameters[5]. Therefore, this is a more consistent way of writing expressions:

([function] [parameter 1] [parameter 2] ..)

Voila! Formally, an expression written in this form is called an *s-expression*. Let's rewrite our expression as an s-expression now:

(all (<not a multiple of> 27 m) ; m ∈ [2, 3 .. 26, 28 ..])

---

[5]imagine a function that adds n numbers by repeatedly using the + operator, which adds two numbers together

(all (<not a multiple of> 27 m) ; m ∈ [2, 3 .. 26, 28 ..])

Now, we simply have to design the function <not a multiple of>. First, think about what the function does in specific cases.

(<not a multiple of> 3 2)

It means that when 3 is divided by 2, it leaves a nonzero remainder, provided that 2 is less than 3. Rewriting it in this form, we get

(<not equal to zero> (<remainder after division> 3 2))

Many languages already provide the <remainder after division> operator. However, <not equal to zero> seems to be nonstandard, but languages already provide operators for comparison.

(<not equal to> (<remainder after division> 3 2) 0)

For the sake of brevity, let us represent these operators by symbols.

(/= (% 3 2) 0)

Finally, when we write the final expression, we have to make sure that m is less than 27.

(all (/= (% 27 m) 0) ; m ∈ [2, 3 .. 26])

Enter: Functions

(all (/= (% 27 m) 0); m ∈ [2, 3 .. 26])

Notice how functions and operators are also represented by symbols. Here, we're assuming that all these symbols are bound to functions provided by the programming language. Let us now attempt to write a custom function to square a given number for example.

(λ x (* x x))

Our function is ready. It accepts a parameter (the symbol x), and returns the square of that number. Carefully observe the above expression: λ itself is a in-built function of one parameter. All the expressions we've seen so far evaluate to a value (data); the one above doesn't- it evaluates to a function. Now let us use the function to square the number 3.

((λ x (* x x)) 3)

We could have bound some symbol to this function, say ν. The above expression would then look like this

(ν 3)

Pretty similar to writing (+ 2 4). Now, imagine that the programming language provides me the function + to add two numbers. And imagine that I want to add three: 1, 2 and x. How would I do it?

(+ 2 (+ 1 x))

The inner expression doesn't evaluate to a value until x is passed to it- it remains a function and is passed as-it-is to the outer expression.

(all (/= (% 27 m) 0); m $\in$ [2, 3 .. 26])

Now to eliminate the one unexplained symbol: $\in$. To do this, we first have to formalize **all**

- ► **all** is a function of two parameters: A (function of one parameter) and a list. The function here is the *predicate* that each member of the list must satisfy, and the parameter is the list itself.

- ► What **all** does: It picks items from the list and checks if <u>all</u> of them satisfy the predicate, and accordingly return **True** or **False**. The predicate itself is a function of one parameter that returns a boolean.

In our problem, the list is the list of numbers from 2 to 26, and the predicate is to check that 27 is <not a multiple of> the given number. Let us now rewrite our expression eliminating the need for the mysterious $\in$:

(all (λ m (/= (% 27 m) 0)) [2, 3 .. 26])

(all (λ m (/= (% 27 m) 0)) [2, 3 .. 26])

We are now ready to get rid of our s-expression crutches and implement this program in a real-world programming language[6]. Most programming languages use parenthesis just as separators, to explicitly define operator precedence, for example.

1. 3 * 4 - 6
2. 3 * (4 - 6)

So finally, in Haskell, the final program is written as

all (\m -> 27 'mod' m /= 0) [2, 3 .. 26]

Yes, the \ is supposed to look like a λ if you squint hard enough.

---

[6]In a Lisp like Common Lisp, Scheme, or Clojure, the s-expression is the final form