Scheme, or indeed any fixed programming language, may not be sufficient for our needs.

We must constantly turn to new languages in order to express our ideas more effectively.

Establishing new languages is a powerful strategy for controlling complexity in engineering design.

We have built abstractions for procedure and abstractions for data.

One can also solving complex problems by creating a new language or vocabulary to better understand the problem space.
Metalinguistic abstraction refers to the creation of languages — abstracting description.

We have built abstractions for procedure and abstractions for data.

One can also solving complex problems by creating a new language or vocabulary to better understand the problem space.
Metalinguistic abstraction refers to the creation of languages — abstracting description.

An **evaluator** (/interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

The evaluator/interpreter, which determines the meaning of expressions in a programming language, is just another program.

Any program can be regarded as an the evaluator for some language possibly with restricted functionality.

For eg., the polynomial manipulation system captures the rules of polynomial arithmetic and implements them in terms of operations on list-structured data.

Any program can be regarded as an the evaluator for some language possibly with restricted functionality.

For eg., the polynomial manipulation system captures the rules of polynomial arithmetic and implements them in terms of operations on list-structured data.

If we augment this system with procedures to read and print polynomial expressions, we have the core of a special-purpose language for dealing with problems in symbolic mathematics, like wolfram mathematica.

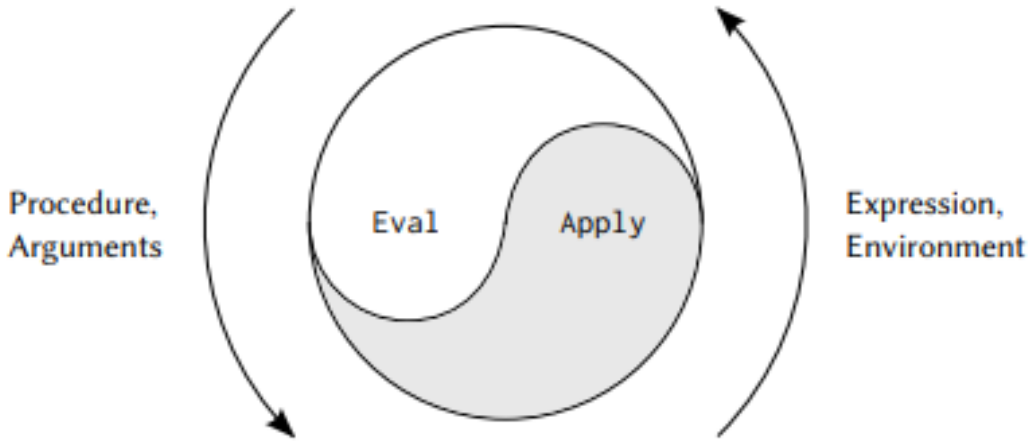
An evaluator that is written in the same language that it evaluates is said to be **metacircular**.

The metacircular evaluator is essentially a Scheme formulation of the environment model of evaluation.

An evaluator that is written in the same language that it evaluates is said to be **metacircular**.

The metacircular evaluator is essentially a Scheme formulation of the environment model of evaluation.

These two rules describe the essence of the evaluation process. This evaluation cycle will be embodied by the interplay between the two critical procedures in the evaluator, **eval** and **apply**.



This model has two basic parts:

1. To evaluate a combination (a compound expression other than a special form),
   - 1.1 Evaluate the subexpressions
   - 1.2 Apply the value of the operator subexpression to the values of the operand subexpressions
2. To apply a compound procedure to a set of arguments,
   - 2.1 Evaluate the body of the procedure in a new environment.
   - 2.2 Construct this environment by extending the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the arguments to which the procedure is applied.

The implementation of the evaluator will depend upon procedures that define the *syntax* of the expressions to be evaluated.

We will use data abstraction to make the evaluator independent of the representation of the language.

For example, rather than committing to a choice that an assignment is to be represented by a list beginning with the *symbol set!* we use an abstract predicate *assignment?* to test for an assignment.

Eval takes as arguments an *expression* and an *environment*.
(eval $< exp >$ $< env >$)
Each type of expression has a predicate that tests for it and an abstract means for selecting its parts.

- For self-evaluating expressions, such as numbers/strings
  eval returns the expression itself.
  For example,
  In environment e_0, (eval "Bye Bye" e_0), $\rightarrow$ "Bye Bye"

- For variables,
  eval returns the values of the variables, by looking up the variables in the environment.
  In environment e_1, (eval x e_1) $\rightarrow$ value of x in e_1

- Quoted expressions:-
  (eval (quote exp) e_0) $\rightarrow$ exp.

- Assignment of (or a definition of) variable must recursively call eval to compute the new value to be associated with the variable. The environment must also be modified to change (or create) the binding of the variable.

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

The procedures return 'ok', just to help us differentiate it from the in-built define.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp)
                                       (lambda-body exp)
                                       env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((switch? exp) (eval (switch->if  exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type: EVAL" exp))))
```

For sake of brevity, we do not look into the procedures' workings. One can implement using basic procedures like cons, car.

Apply takes two arguments, a *procedure* and a list of *arguments* to which the procedure should be applied.
(apply <proc> <args>)

apply classifies procedures into two kinds:

1. It calls apply-primitive-procedure to apply primitives.
2. It applies compound procedures by sequentially evaluating the expressions that make up the body of the procedure.
   The environment for the evaluation of the body of a compound procedure is constructed by extending the base environment carried by the procedure.

```
(define (apply proc args)
(cond ((primitive-procedure? proc)
        (apply-primitive-procedure proc args))
      ((compound-procedure? proc)
        (eval-sequence
          (procedure-body proc)
          (extend-environment
            (procedure-parameters proc)
            args
            (procedure-environment proc))))
      (else
        (error
          "Unknown procedure type: APPLY" procedure))))
```

Notice how eval and apply make use of each other to perform their respective tasks.
When we encounter compound-procedures, we create a bundle for corresponding block of code known as the closure.

Sample procedure and its output

$$
\begin{array}{ll}
(((\lambda(x) & \\
\quad (\lambda(y) & \\
\quad\quad (+\ x\ y)\ )\ )\ 3)\ 4) & \\
\rightarrow & ((\lambda(y) \\
\quad\quad (+\ 3\ y)\ )\ 4) \\
\rightarrow & (+\ 3\ 4) \\
\rightarrow & 7
\end{array}
$$

Let ℓ_{inner} be $(\lambda(y)\ (+\ x\ y))$
Let ℓ_{outer} be $(\lambda(x)\ \ell_{inner})$
Let ℓ be $((\ell_{outer}\ 3\ )\ 4\ )$

Let's look at an example of evaluating an expression in an environment e_0.
e_0 has the definitions for $+$,*, car, cdr, cons etc..

(eval ℓ e_0)
(eval ((ℓ_{outer} 3) 4) e_0)

ℓ does not fit into any of the cases in the cond of eval, except the second last case.

We realize that we have encountered the case where the expression is an application of an operator to its operands.
In this case, 4 happens to the operand and the rest of the expression [i.e (ℓ_{outer} 3)] preceeding 4 happens to be the operator.

```
((application? exp)
 (apply (eval (operator exp) env)
        (list-of-values (operands exp) env)))
```

$\rightarrow$ (apply (eval (ℓ_{outer} 3) e_0)
(list-of-values (4) e_0))
$\rightarrow$ (apply (eval (($\lambda(x)$ ℓ_{inner} 3) e_0)
'(4)))

```
((application? exp)
 (apply (eval (operator exp) env)
        (list-of-values (operands exp) env)))
```

$\rightarrow$ (apply (eval (ℓ_{outer} 3) e_0)
(list-of-values (4) e_0))
$\rightarrow$ (apply (eval (($\lambda(x)$ ℓ_{inner} 3) e_0)
'(4)))

Now we deal with the eval inside the apply procedure.
$\rightarrow$ (apply (apply (eval ($\lambda(x)$ ℓ_{inner} e_0)
(list-of-values (3) e_0))
'(4)))

Now the inside eval is corresponds the case where we encounter the lambda? case which creates a procedure object.

This procedure object is captured by the closure. The closure binds x to the internals of the body.

$\rightarrow$ (apply (apply (closure('(x) ℓ_{inner} e_0) '(3))
'(4))

A new environment e_1 is made at this point which has e_0 as its parent.e_1 has access to everything which e_0 has access to and it also binds x to 3.

$\rightarrow$ (apply (eval ℓ_{inner} e_1)
'(4))
$\rightarrow$ (apply (eval ($\lambda(y)$ (+ x y)) e_1)
'(4))
$\rightarrow$ (apply '(closure ('(y) (+ x y)) e_1)

A new environment e_2 is made at this point which has e_1 as its parent. e_2 has access to everything which e_1 has access to and it also binds y to 4.

$\rightarrow$ (eval (+ x y) e_2)
$\rightarrow$ (apply (eval + e_2)
(list-of-values '(x y)) e_2)
$\rightarrow$ (apply + '(3 4))
$\rightarrow$ 7

One operational view of the meaning of a program is that a program is a description of an abstract (perhaps infinitely large) machine.

We can regard the evaluator as a very special machine that takes as input a description of a machine and configures itself to emulate the machine described.
From this perspective, our evaluator is seen to be a universal machine. It mimics other machines when these are described as Scheme programs.

Alan Turing first captured this the notion of "what can in principle be computed" (ignoring practicalities of time and memory required) as being independent of the language or the computer, and instead reflecting an underlying notion of *computability*.

Alan Turing captured this notion using Turing machines. Turing machines were akin to procedures and the Universal Turing machine was akin to the evaluator.

The evaluator acts as a bridge between

- the data objects that are manipulated by our programming language and
- the programming language itself.

From the perspective of the user,
an input expression such as $(* \; x \; x)$ is an expression in the programming language, which the evaluator should execute.
From the perspective of the evaluator,
the expression is simply a list (in this case, a list of three symbols: $*$, x, and x) that is to be manipulated according to a well-defined set of rules.

Thus, the user's programs are the evaluator's data.