New languages are often invented by first writing an evaluator that embeds the new language within an existing high-level language.

For example, if one wishes to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, then he/she can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications.

The high-level implementation base make it easier to test and debug the evaluator.

Later if the new aspect seems worthy, the designer goes to the trouble of building a complete implementation in a low-level language or in hardware.

The preliminary interpreter we studied/designed in the previous lecture did not have the capability to deal with functions which can possibly have any arbitary number of elements.

If we had to add three numbers in the current version of the interpreter we would have to express in the following way
(+ (+ 3 4) 5)

The preliminary interpreter we studied/designed in the previous lecture did not have the capability to deal with functions which can possibly have any arbitary number of elements.

If we had to add three numbers in the current version of the interpreter we would have to express in the following way
(+ (+ 3 4) 5)

But certain functions cannot operate as desired unless all the arbitrary number of operands are provided simultaneously.
For example,
The function which takes all its arbitary number of arguments, squares them, sums up these squares and then finds the square root of the sum.

We need to come with syntatic specification to notate the additional arguments with **unknown cardinality**.

Then we need to specify how to interpret that notation correctly.

We need to come with syntatic specification to notate the additional arguments with **unknown cardinality**.

Then we need to specify how to interpret that notation correctly.

Let there be a procedure consisting of two sets (in the mathematical sense) of arguments X and Y.
Suppose $X = x_1 \ x_2 \dots x_i$ is of fixed size i, whereas Y can be of any size and possibly zero.

If $i = 3$, denoting it as $\lambda(x_1 \ x_2 \ x_3 \ Y)$ would not work as expected, as the procedure would try to map Y to some argument, which may not be present.

If the procedure has three required arguments and rest are optional, one could **create a new notation** as
$\lambda(x_1\ x_2\ x_3 \cdot Y)$

The dot indicates that the thing after the dot is a list of the all the rest of the arguments.

This is one possible syntatic specification. This dotted representation is what is being used to represent the underlying cons of pairs.

Now, we need to modify the metacircular interpreter to **correctly interpret the new notation**.

We need to create a internal procedure that pairs-up the formal parameters with the arguments that are passed.

Now, we need to modify the metacircular interpreter to **correctly interpret the new notation**.

We need to create a internal procedure that pairs-up the formal parameters with the arguments that are passed.

1. If the list of formal parameters and the arguments are both empty, we do not face a problem, otherwise if only the list of formal parameters is empty, then we have supplied too many arguments and that causes an error.
2. Else, If the list of formal parameters is non-empty, then we pair up the variable with its value, otherwise it implies that the variable has not been assigned a value and that causes an error.

Then we move on to the next pair of formal parameters and arguments to process them recursively.

Now, we need to modify the metacircular interpreter to **correctly interpret the new notation**.

We need to create a internal procedure that pairs-up the formal parameters with the arguments that are passed.

1. If the list of formal parameters and the arguments are both empty, we do not face a problem, otherwise if only the list of formal parameters is empty, then we have supplied too many arguments and that causes an error.
2. Else, If the list of formal parameters is non-empty, then we pair up the variable with its value, otherwise it implies that the variable has not been assigned a value and that causes an error.

Then we move on to the next pair of formal parameters and arguments to process them recursively.

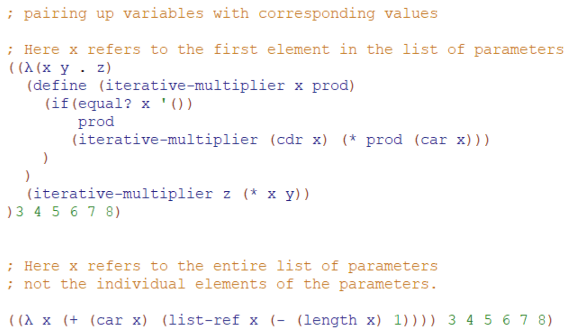
This is the **general** case

There is a special case which one also needs to keep in mind.

What if the dot is missing and there is only one formal parameter and there are several arbitrary number of arguments provided?

There is a special case which one also needs to keep in mind.

What if the dot is missing and there is only one formal parameter and there are several arbitrary number of arguments provided?



```
; pairing up variables with corresponding values

; Here x refers to the first element in the list of parameters
((λ(x y . z)
  (define (iterative-multiplier x prod)
    (if(equal? x '())
        prod
       (iterative-multiplier (cdr x) (* prod (car x)))
    )
  )
  (iterative-multiplier z (* x y))
)3 4 5 6 7 8)


; Here x refers to the entire list of parameters
; not the individual elements of the parameters.

((λ x (+ (car x) (list-ref x (- (length x) 1)))) 3 4 5 6 7 8)
```

It is possible to interpret the entire set of args as a single argument. While modifying the meta-circular evaluator, one needs to be careful

Static and Dynamic scoping refer to the mechanism using which free variables get their values.

Static Scoping:- Free variable(s) in the procedure gets their value(s) from the environment in which the procedure is defined.

This means that the binding of a variable in a program is determined by the static structure of the program, not by its run-time behavior.

Thus, the scope of a variable can be determined at compile-time before the program runs just by examining the source code.

- In static scoping, an occurrence of a variable in an expression always refers to the innermost lexically apparent binding of that variable. For this reason, static binding is also called lexical scoping.

- Most modern languages are statically scoped.

- Static scoping helps programmers to read source code and determine what values names are bound to without need to run the code.

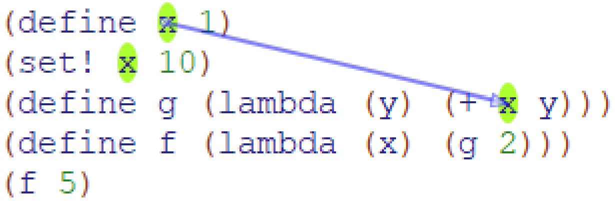
  In other words, a compiler can determine which definition each variable refers to—but it may not be able to determine the values of each variable.

Under dynamic scoping, a variable is bound to the most recent value assigned to that variable, i.e., the most recent assignment during the program's execution.

In technical terms, this means that each variable has a global stack of bindings and the occurrence of an variable is searched in the most recent binding.

The original version of Lisp had dynamic scoping.

Scheme currently uses static scoping.



```
(define x 1)
(set! x 10)
(define g (lambda (y) (+ x y)))
(define f (lambda (x) (g 2)))
(f 5)
```

Due to lexical scoping, the compiler tracks down the *x* being referred from within the body of *g* is called, *x* is given the value of 10.

Thus, in Scheme, the output of (*f* 5) would be 12.

Reason:- Before $(f\ 5)$ is called, x was bound to 1 by the first define.
Later x is set to 10.
When $(f\ 5)$ is called, the 5 is bound to x in the lambda expression for f.
Then $(g\ 2)$ is called, and the 2 is bound y in the lambda expression for g.
So at this point, there are three bound variables:

- x bound to 10,
- x bound to 5, and
- y bound to 2. (no conflict)

In g's body expression $(+\ x\ y)$, what value should be used for x?

Since Scheme is statically scoped, it decides on bindings before the code runs, which means it must use the x bound to 10.

However, in a dynamically scoped language
(notably many versions of Lisp before Scheme),
the most recently bound value of x is used.

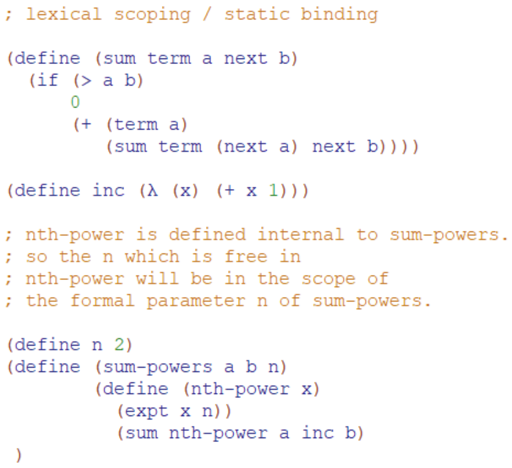
If Scheme were dynamically scoped, x would be assigned the value of 5, then (f 5) would return 7.

Since Scheme is statically scoped, it decides on bindings before the code runs, which means it must use the x bound to 10.

However, in a dynamically scoped language
(notably many versions of Lisp before Scheme),
the most recently bound value of x is used.

If Scheme were dynamically scoped, x would be assigned the value of 5, then (f 5) would return 7.

Why would someone need dynamic scoping?



```
; lexical scoping / static binding

(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))

(define inc (λ (x) (+ x 1)))

; nth-power is defined internal to sum-powers.
; so the n which is free in
; nth-power will be in the scope of
; the formal parameter n of sum-powers.

(define n 2)
(define (sum-powers a b n)
        (define (nth-power x)
          (expt x n))
          (sum nth-power a inc b)
 )
```

nth-power is an internal procedure which cannot be accessed by other procedures.

Internal definitions allow us to notate a procedure with more parametric control than just arguments.

We then have two paths to influence its computation. We can communicate with a procedure by passing arguments through formal parameters, and we can bind its free variables.

Internal definitions allow us to notate a procedure with more parametric control than just arguments.

We then have two paths to influence its computation. We can communicate with a procedure by passing arguments through formal parameters, and we can bind its free variables.

But what if the internal procedure is used by several other procedures.
It would not be unreasonable to define the internal procedure globally.

```
(define n 3)

(define (sum-powers a b n)
  (sum nth-power a inc b))

(define (product-powers a b n)
  (product nth-power a inc b))

; since nth-power is used in several places,
; we would like to define it external to the
; procedures that use them

(define (nth-power x)
  (expt x n))

; In this case, the third parameter in
; sum-powers description is
; never used as the nth-powers procedure
; makes use of the globally defined n.

(sum-powers 1 5 10)
(sum-powers 1 5 2)
```

Here the nth-power has been defined globally, but it needs to assign value to the free variable *n*.

Initial Lisp systems have been implemented so that variables are bound dynamically rather than statically.

In a language with dynamic binding, free variables in a procedure get their values
from the environment from which the procedure is called
rather than from the environment in which the procedure is defined.

Thus, for example, the free *n* in nth-power would get whatever value *n* had when sum called it.

Thus, the test case (sum-powers 1 5 10), would use the value of *n* to be 10 and ~~not 3~~.

In this situation dynamic binding would be preferred over static binding.

If nth-power represents a common pattern of usage, its definition will be repeated as a subdefinition in many contexts.

It would be desirable to abstract the definition of nth-power to a more global context where it can be shared among many procedures.

In this situation dynamic binding would be preferred over static binding.

If nth-power represents a common pattern of usage, its definition will be repeated as a subdefinition in many contexts.

It would be desirable to abstract the definition of nth-power to a more global context where it can be shared among many procedures.

For example, in a dynamically bound Scheme, if we have both sum and product accumulator procedures, we can define both sum-powers and product-powers to share the same nth-power routine:

To make this work, early scientists were motivated towards the development of dynamic binding disciplines in traditional Lisp dialects.

Unfortunately, dynamic binding must, of necessity, have the problem of symbol conflicts where higher-order procedures capture free variables in procedures passed as arguments.

Dynamic binding violates the principle that changing the name of a parameter throughout a procedure definition (in this example, changing say inc to *n* in sum) should not change the behavior of the procedure.

This is an important modularity problem because then the user of a procedure which takes a procedural parameter must know that the author of the procedure he is using did not name any of his bound variables in a way wich might conflict with the free variables occurring in the procedures being passed as arguments.

To make our evaluator use dynamic binding rather than static binding requires only a tiny change.

To make our evaluator use dynamic binding rather than static binding requires only a tiny change.

- When apply builds the environment for executing the body of a compound procedure,
  it extends the evaluation environment of the combination that called for the procedure application,
  rather than extending the environment of the ~~procedure's definition~~.

To make our evaluator use dynamic binding rather than static binding requires only a tiny change.

- When apply builds the environment for executing the body of a compound procedure,
  it extends the evaluation environment of the combination that called for the procedure application,
  rather than extending the environment of the ~~procedure's definition~~.
- This environment must therefore be passed from eval to apply as an additional argument.

To make our evaluator use dynamic binding rather than static binding requires only a tiny change.

- When apply builds the environment for executing the body of a compound procedure,
  it extends the evaluation environment of the combination that called for the procedure application,
  rather than extending the environment of the ~~procedure's definition~~.
- This environment must therefore be passed from eval to apply as an additional argument.

Also, in a dynamically scoped Scheme, it is unnecessary for make-procedure to attach the definition environment to a procedure, since this is never used.

A solution for lexically scoped evaluator for the feature we desire through dynamic scoping.

```
; pgen now abstracts the exponentiation
; and still works for lexical scoping.

(define  pgen
         (λ(n)
            (λ(x) (expt x n))))

(define sum-powers
       (λ(a b n)
          (sum (pgen n) a inc b)))

(define prod-powers
       (λ(a b n)
          (product (pgen n) a inc b)))
```

We create a parameterized procedure pgen. pgen depends upon explicitly, the lexically apparent value of *n*.

We achieve the desired feature (abstracting (expt *x* *n*)) by naming the term generation procedures with an *n* in them.