Rules of expression evaluation

• Numbers evaluate to “themselves”
  – a character string representing a number evaluates to its base 10 numeric representation (unless a different base is explicitly indicated)

• For example:
  – Scheme
    > 100 if you ask Scheme to evaluate a string of (decimal) digits
    100 you get the corresponding decimal number as a result
    > #o100 if you ask Scheme to evaluate a string of octal digits
    64 you get the corresponding decimal number as a result
  – This character string to numeric representation happens in other languages as well; consider Java:
    ```java
    int x = 100;
    int y = 0100;
    System.out.println("x is "+x+, but y is "+y);
    ```
  – prints: x is 100, but y is 64
Evaluating names

- A name is evaluated by looking it up. Lookup starts in the current environment, and continues along the chain of statically-linked environments until either a binding is found (in which case the corresponding value is returned) or it isn’t (in which case an error occurs).
- For example (assuming no binding for x exists yet):

  > x
  Error: reference to undefined identifier: x
  > (define x 12)
  > x
  12
  >
lambda forms

• A lambda form (lambda abstraction) defines a function in Scheme.

• Informally, the syntax is:

  \[ \text{lambda (parameters) body} \]

• When a lambda form is evaluated a closure results (which is printed as \#<procedure:name>).
  
  – Aside - comparison of proposals for adding closures to Java: www.artima.com/weblogs/viewpost.jsp?thread=202004

• For example:

  \[
  \text{> (define addOne (lambda (p) (+ p 1)))}
  \text{> addOne}
  \text{#<procedure:addOne>}
  \text{>}
  \]
Primitives

- Primitives, such as + and -, are ordinary names with name-value bindings established at start-up in the *primitive environment*, the base environment for evaluation (base in the sense that its *static link* is null).
- We can use + wherever we need a function.
- For example (defining a function which takes a function as argument):
  ```scheme
  > (define applyOperator (lambda (op) (op 3 4)))
  > (applyOperator +)
  7
  > (applyOperator -)
  -1
  ```
- We can also rebind the names of primitives (not a good idea in general, but this shows that primitives are not treated differently from other names in the language).
  ```scheme
  > (+ 3 4) name “+” refers to addition function
  7
  > (define + -) name “+” now refers to subtraction function
  > (+ 3 4)
  -1
  ```
Function applications

• A function application is evaluated by first evaluating each expression inside the application, then applying the function to its arguments.
• This is accomplished by first creating a new environment in which the function’s parameters are bound to its arguments, and then evaluating the function’s body with respect to the new environment (which is now the current environment).
Evaluating \((\text{addOne} \ 4)\)
Local bindings

• In a language like C or Java we can create local bindings inside a function/method by defining local variables:

```c
int mystery(int x, int y, int z) {
    int a = func1(x,y);
    int b = func2(y,z);
    <body>
}
```

• Local bindings can be created using a let-form:

```lisp
(define mystery (lambda (x y z)
    (let ((a (func1 x y)) (b (func2 y z))) <body>)))
```

• Let-form is just syntactic sugar for a procedure application; the above let-form is equivalent to:

```lisp
((lambda (a b) <body>) (func1 x y) (func2 y z))
```
Evaluating

\(\text{(define foo (let ((a 1) (b 2)) (lambda (c) (+ a b c)))})\)