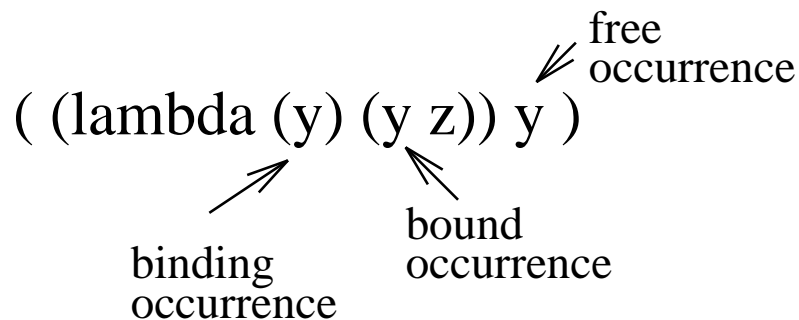


## Review: Free and bound variables

---

Function  $(\text{lambda } (x) e)$  “binds” variable  $x$  in “body”  $e$ . You can think of this as a declaration of variable  $x$  with scope  $e$ .



### Lexical (static) scoping:

The occurrence of a variable matches the lexically closest binding occurrence. An occurrence of a variable without a matching binding occurrence is called *free*.

A variable can occur *free* and *bound* in an expression.

⇒ Conceptually, we only substitute free occurrences of the formal arguments in the function body when computing a function application!

# Environments and Closures

---

## Environments

- Defer substitution by recording the bindings for the variables we would substitute in a data structure called an **environment**. If we need the value that a variable denotes, we just look it up in the environment.

*An environment is a finite map from variables to values*

$$\rho \in Env = Variables \rightarrow Values$$

## Closures

This is what you get if you define a function value in our scm scheme interpreter:

```
> (lambda (x) (+ x a))
  #<CLOSURE (x) (+ x a)>
> (define test (lambda (x) (+ x a)))
  #<unspecified>
> (test 1)
  ERROR: unbound variable:  a
```

## Closures (cont.)

---

- Pair the environment with a function (lambda abstraction). The environment must contain values for all free variables of the function. The function can only be evaluated in its attached environment, making capturing (i.e., replacing the “wrong” parameter) impossible.

Such a pairing is called a **closure**.

*A closure is a pair consisting of an environment and a lambda abstraction*

$$cl \in Closure = \{ \langle \lambda, \rho \rangle \mid FreeVar(\lambda) \subseteq DOM(\rho) \}$$

Closures can be used to implement lexical scoping. They represent lexically scoped function values.

## How To Apply a Closure?

---

How to apply a closure value to actual argument values?

1. Let  $c_v$  be the closure value  $\langle (\text{lambda}(\mathbf{x}) \ \mathbf{e}), \rho \rangle$ .
2. Apply  $c_v$  to a value  $a_v$  as follows:  
 Evaluate the body  $\mathbf{e}$  of the function in the environment  $\rho$  of the closure **extended** by the mapping of the formal parameter  $\mathbf{x}$  to the actual value  $a_v$  ( $\rho[x \rightarrow a_v]$ ).

$((\text{lambda}(\mathbf{x})$   
 $((\text{lambda}(\mathbf{z}) ((\text{lambda}(\mathbf{x})(\mathbf{z} \ \mathbf{x})) \ 3)) (\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})))) \ 1)$

closure interpreter	{ }
$((\text{lambda}(\mathbf{z})$ $((\text{lambda}(\mathbf{x})(\mathbf{z} \ \mathbf{x})) \ 3))$ $(\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})))$	$\{\mathbf{x} \rightarrow 1\}$
$((\text{lambda}(\mathbf{x})$ $(\mathbf{z} \ \mathbf{x})) \ 3)$	$\{\mathbf{x} \rightarrow 1,$ $\mathbf{z} \rightarrow \langle (\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})), \{\mathbf{x} \rightarrow 1\} \rangle\}$
$(\mathbf{z} \ \mathbf{x})$	$\{\mathbf{x} \rightarrow 3,$ $\mathbf{z} \rightarrow \langle (\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})), \{\mathbf{x} \rightarrow 1\} \rangle\}$
$(+ \ \mathbf{x} \ \mathbf{y})$	$\{\mathbf{x} \rightarrow 1, \mathbf{y} \rightarrow 3\}$

4

# Environments - How to implement them?

## procedural (functional) representation

```
(define extend
  (lambda (env x v)
    (lambda (y)
      (if (eq? x y)
          v
          (lookup env y)))))
```

```
(define lookup
  (lambda (env y) (env y)))
```

```
(define empty-env
  (lambda (y)
    (error "unbound variable")))
```

## data representation?

```
(define extend
  (lambda (env x v) . . .
```

```
(define lookup
  (lambda (env y) . . .
```

```
(define empty-env . . .
```

# Data Types

---

**Representation:** organization of values that corresponds to some meaningful entity.

**Type:** set of values + valid operations on them

E.g.,

Integers: + - \* *div* < ≤ = ≥ > ...

Booleans: ∧ ∨ ¬ ...

Sets: Union, Intersection, Difference....

Strings: Concatenate, Reverse ...

## Data Types

---

- Enable programmers to think in terms of modelling reality with different kinds of values
  - Program semantics are embedded in types
- Enable additional compile- and run-time checks beyond syntax checking
  - variable decls
  - operator/operand compatibility
- Determine layout and allocation of data

# Type Systems

---

- Primitive types
  - E.g., C: `int`, `float`, `char`, `double`, ...
  - E.g., Pascal: **`char`**, **`integer`**, **`real`**, **`boolean`** ...
- Constructors for structured types
  - E.g., C: `*`, `&`, `[]`, **`struct`**, ...
  - E.g., Pascal: `↑`, `[]`, **`record`**, ...
  - E.g., Java: **`class`**, ...

# Determining Types

---

## Explicit Typing:

- Function and variable types are determined by declarations
- Type (usually) invariant throughout execution
- Supports static type determination

E.g., Pascal, Algol, C, C++, Java

# Determining Types

---

## Implicit Typing:

- Function and variable types are determined by use  
E.g., Prolog, Scheme, Smalltalk

## Mixture:

- Implicit by default, but allows explicit declarations  
E.g., Miranda, Haskell, ML

## Rules for Determining Types of Expressions

General Rule: If  $f$  has type  $S \rightarrow T$ , and  $x$  has type  $S$  then  $f(x)$  has type  $T$ :

type of `3 div 2` is `int`

type of `round(3.5)` is `int`

Special Rules: Needed to handle automatic type conversion:

type of `2 + 3.3` is `float`

**Type Errors:** Some expressions cannot be given a type, and are rejected:

reject `round('Nancy')`

reject `3.5 div 2.5`

## Type Checking

---

Is each operator/function supplied with the correct type of arguments?

**Type Error:** Applying a function of type  $S \rightarrow T$  to an argument not of type  $S$

### Static or compile-time checking:

Use declaration information or static usage to infer type

### Dynamic or run-time checking:

At execution check type of objects before performing operations on them  
(e.g., use type tags to record types of values)

## Type Safety and Strong Typing

---

**Type-Safe Program:** A program that executes without type errors on all possible inputs

**Strongly Typed:** Programming language definition forces all type-checked programs to be type-safe, i.e., no type error remains undetected.

**(Statically) Strongly Typed:** Compiler allows only programs that it can statically determine to be type safe

**(Dynamic) Strongly Typed:** All operations include code to check the types of the operands at run time, if the type is unknown at compile time