Functional Programming Wrap-Up

Robert Keller
February 2010

Definitions

- A definition in Scheme extends the base environment with a new binding of a **variable** (the thing being defined) to a **value** (the value of the right-hand-side).

Two Ways to Achieve Definitions

- Purely functional way:
  - At each iteration of the read-eval-print loop, an environment is returned for use at the next step.

- Side-effect way:
  - When a definition is encoutered, the environment is modified to accommodate the new binding:
    - A variable defined for the first time adds a binding.
    - A variable re-defined can either add a binding (shadowing previous ones), or could modify an existing one.

Purely-Functional REPL

Previous environment

```
(define (read-eval-print env)
 (begin
  (prompt)
  (let (
       (expression (read))
       )
   (if (eof-object? expression)
       expression                                  ;; EOF
        (read-eval-print (top-level expression env))  ;; Normal
        )
   )
  ))
```

Returns new environment

Previous environment

top-level returns environment

```
(define (top-level expression env)
 (cond
  ((non-empty-list? expression) (handle-composite expression env))
  (else (handle-evaluation expression env))))

(define (handle-composite expression env)
 (if (equal? (first expression) definition-symbol)
    (handle-definition (rest expression) env)
    (handle-evaluation expression env)))

(define (handle-evaluation expression env)
 (let (
      (value (Eval expression env))
      )
  (begin
   (print (render value))
   env)))
```

Assumes evaluation doesn't need to change environment.

'(define *variable expression*)
in user input
creates new environment from old

```
(define (handle-definition definition env)
  (if (and (length2? definition) (variable-symbol? (first definition)))
     (let* (
          (variable (first definition))
          (result (Eval (second definition) env))
          (newenv   (cons (list variable result) env))
          )
       (begin
         (display variable)
         (display " is ")
         (display (render result))
         newenv))
     (Eval-error "ill-formed definition" definition)))
```

new env't

returned value

Side-Effect Method

```
(define (handle-definition definition)
  (if (and (length2? definition) (variable-symbol? (first definition)))
      (let (
            (variable (first definition))
            (result (Eval (second definition) global-environment))
            )
        (begin
          (set! global-environment
              (cons (list variable result) global-environment))
          (display variable)
          (display " is ")
          (display (render result))
          result))
      (Eval-error "ill-formed definition" definition)))
```

Modifies environment destructively.

Trade-off

- If, for some reason, the REPL were stopped, the environment would be lost with the purely-functional version,
- unless stopping also entailed saving the environment, which might not always be feasible, e.g. if an exception is thrown, and which would not be purely functional.

Implementation of Functions in an Interpreter

- **Suppose we want to extend our interpreter to include one of the following Scheme ideas:**
  - (define (fun … args …) ___ body ___) ; Named functions
  - (lambda (… args …) ___ body ___) ; Anonymous functions
- We first note that if we can do the second and can have definitions, we can do the first, since the first is equivalent to:
  - (define fun (lambda (… args …) ___ body ___))

Implementing lambda is Sufficient

- In order to be applied following its creation, a function (defined by a lambda expression), must contain:
  - A list of variables (called **formal** variables, to distinguish them from actual)
  - The **body** expression, for evaluation.
  - Note that the body expression generally contains occurences of the formal variables.

Application of a Function

- To apply a function, the interpreter must:
  - **Evaluate** the **actual** values (which could be specified by general expressions).
  - **Create a new environment** in which the formals are bound to the corresponding actual values.
  - **Evaluate the body** in the environment thus created.

Example

- Function application:

  ((lambda (x y) (+ x (* 3 y))) 4 5)

  formals body actuals

New environment:

((x 4) (y 5) …)

old environment
(in case of globals)

Example

- Function application with argument evaluation:

((lambda (x y) (+ x (* 3 y))) (+ 4 5) (+ 6 7))

formals

body

actuals (evaluated in old env)

New environment:

((x 9) (y 13) ...)

another environment (to be discussed)

Implementing Imported Variables

- When the body contains imported variables, there are special considerations.

- The imports should take on whatever meaning they had at the time the function is created from the lambda expression.

Examples with Imports

```
(let (
        (b 99)
          )
        (let (
           (f (lambda (x) (+ b x)))
                )
            (f 1)
        )
)
```

The value should be 100.

Examples with Imports

```
(let (
        (b 99)
         )
        (let (
          (f (lambda (x) (+ b x)))
                )
                (let (
                      (b 1)
                     )
                (f 1)
                )
        )
)
```

The value should again be 100, not 2.

Static vs. Dynamic Binding

- The prescribed behavior is an example of **static binding**: A function should not change its value based on shadowed bindings.

- If it were to (e.g. give value 2 in the previous example), that would be dynamic binding.

Implementation of Static Binding

- In addition to the components of a function already mentioned:
  formals and *body*
  a function, as a data item, must also carry the **static environment** for the imports
- The formal-actual bindings are added to the top of the static environment.

Closures

- "Closure" is the standard term for a data structure representing a function. It contains:
  - Formal argument list
  - Static environment
  - Body expression

Closures as Extensions of Lambda

- We'll represent a closure as a 3-component lambda expression, to distinguish it from the normal 2-component one:

New!

```
(lambda (…formals…) ___body ___ env)
```

For example

```
(lambda (x y)   (+ x (* b y))    (… (b 99) …))
```

Evaluation of a Lambda Expression

- To evaluate a lambda expression:
  - Create a **closure** (3-component lambda), by added the current environment to the formals and body

Application of a Function

- Evaluate the thing being applied.
- Evaluate the actual argument expressions.
- The result should be either a closure or a built-in.
- Assuming it is a closure:
  - Bind the formals to the actuals
  - Add the bindings to the top of the environment in the closure.
  - Evaluate the body in the resulting environment.

New Code

```
(define (Eval-operator operator actuals env)
  (case operator
    ('not (Eval-not actuals env))
    ('and (Eval-and actuals env))
    ('or  (Eval-or  actuals env))
    ('let (Eval-let  actuals env))
    ('lambda (Eval-lambda  actuals env))
    (else
     (let (
           (closure-value (Eval operator env))
           )
      (Eval-closure-application closure-value actuals env)))))
```

Eval lambda exp

Eval application

Evaluating lambda to closure

```
(define (Eval-lambda actuals env)
  (if (length2? actuals)
     (let* (
           (formals (first actuals))
          (body (second actuals))
          )
      (list 'lambda formals body env)  ;; capture the environment
      )
     (Eval-error "error in lambda expression" (list 'lambda formals actuals)
```

Applying Closure

```
(define (Eval-closure-application closure actuals env)
 (let* (
      (actual-values (map (lambda (x) (Eval x env)) actuals))
      (formals (second closure))
      (body (third closure))
      (static-environment (fourth closure))
      (body-environment
                (add-bindings formals actuals static-environment))
      )
  (Eval body body-environment)))
```

Example with Environments

```
(let (
        (b 99)
         )
        (let (
          (f (lambda (x) (+ b x)))
              )
                      (let (
                           (b 1)
                          )
                      (f 1)
              )
        )
)
```

(b 99)

(f (lambda (x) (+ b x))) ((b 99))
(b 99)

(b 1)
(f (lambda (x) (+ b x))) ((b 99))
(b 99)

The value should again be 100.

Tests using Logic Evaluator

```
(test (Eval '(lambda (x y) (or x y)) ())          '(lambda (x y) (or x y) ()))

(test (Eval '(lambda (x y) (or x y)) '((b 1))) '(lambda (x y) (or x y) ((b 1))))

(test (Eval '(let ((f (lambda (x y) (or x y)))) (f 0 1)) ())          1)

(test (Eval '(let ((b 1)) (let ((f (lambda (x) (or b x)))) (f 0))) ())   1)
```

The Case of Globals

- If the imported variable is defined in the global environment using a define, the environment for the import will reflect the current value of the global is (it may be changed using set!).
- The environment *structure* does not change, but the value may.
- Implementation: Don't carry the global environment around. Instead, look up in it if variable not found in the static environment.

Bindings to Global Environment in Scheme

```
(define b 1)

(define f (lambda (x) (+ b x)))

(f 99)
```

value is 100

```
(set! b 100)

(f 99)
```

value is 109

New Topic: McCarthy's Transformation

- Every imperative program (program based on assignment statements) can be converted to a functional program.

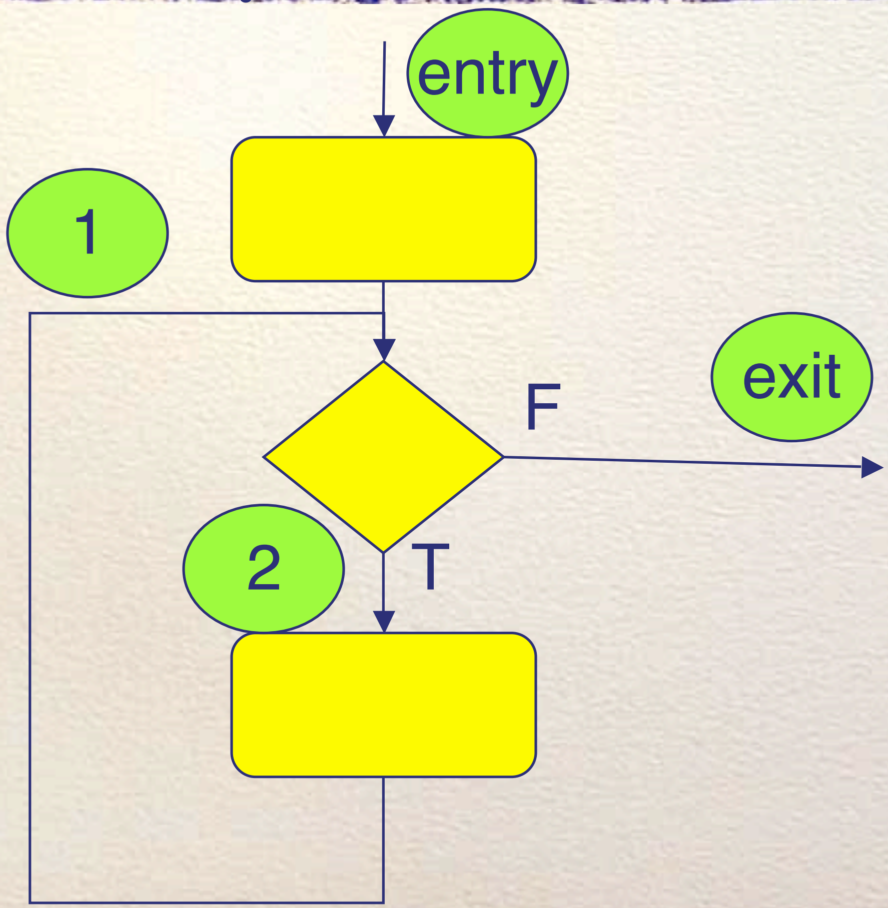
Idea

- The **state** of an imperative program is a mapping from variable names to values.
- A **place** in an imperative program is the point between two commands.
- For each place, construct a mapping from state to **final value** returned by the program.

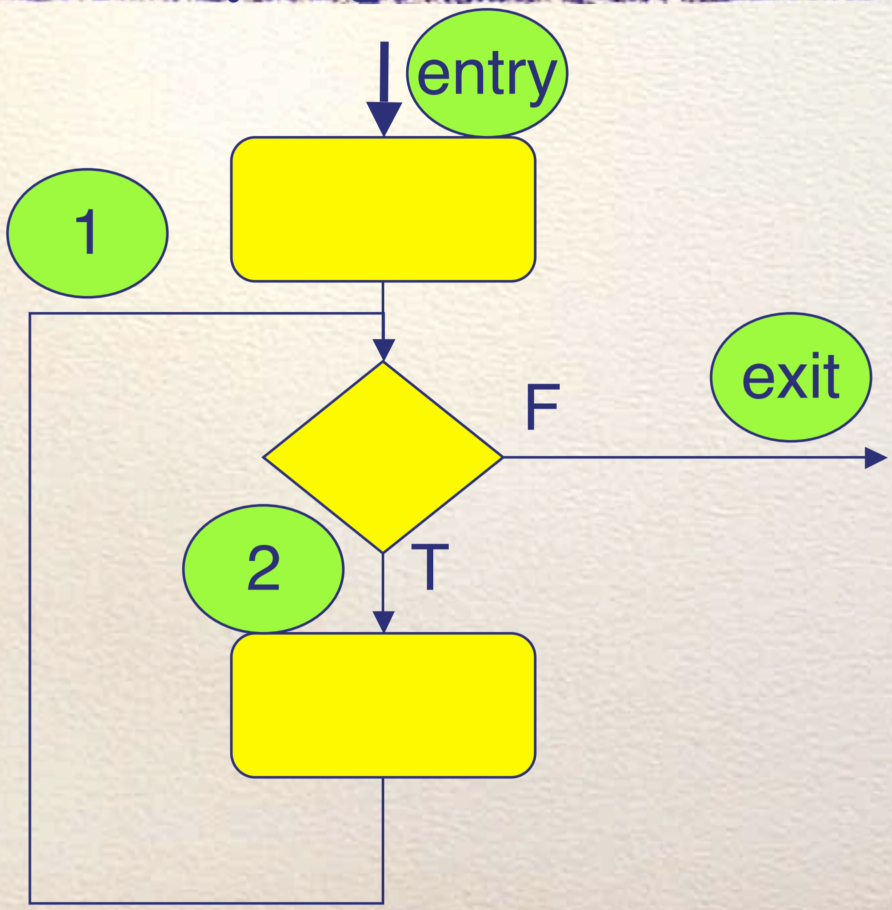
Imperative Program Basis

- Assume a flow chart
- Two kinds of boxes:
  - Assignment
  - Test

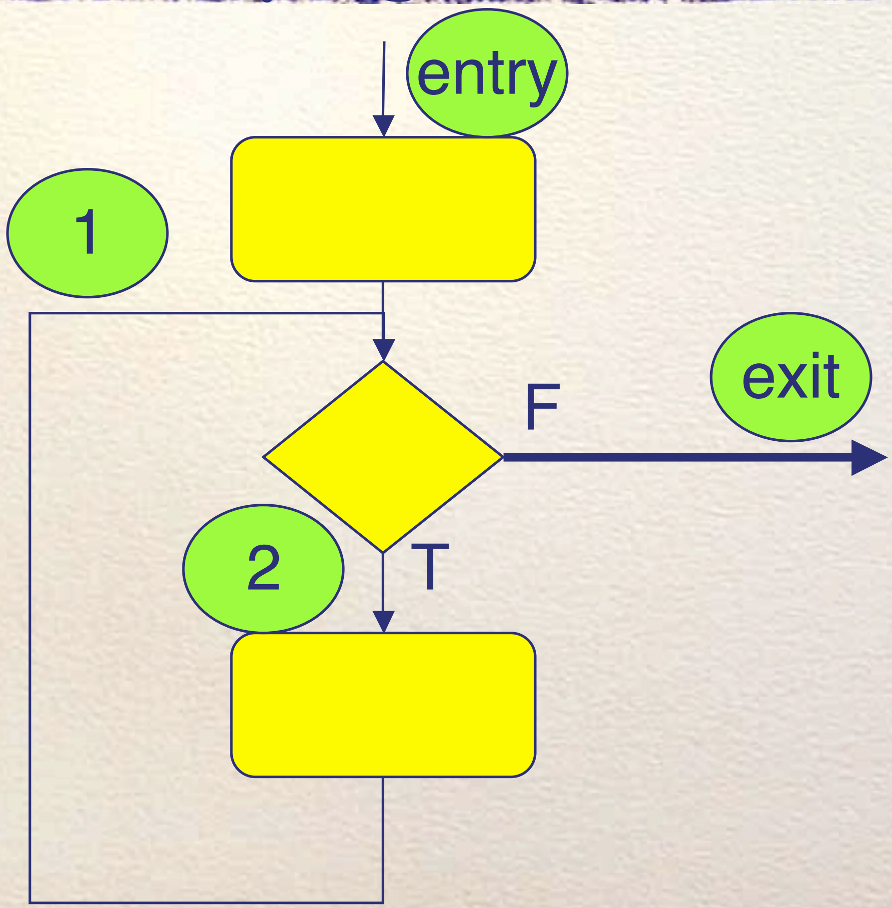
Example Flowchart



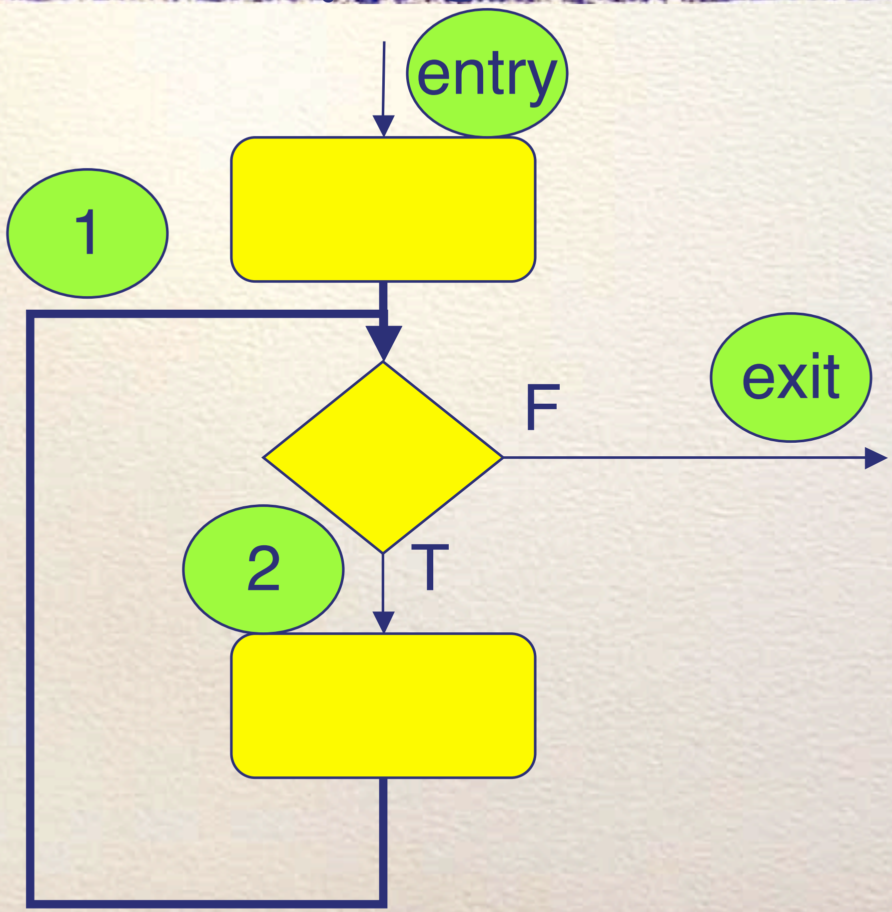
Identifying Place entry



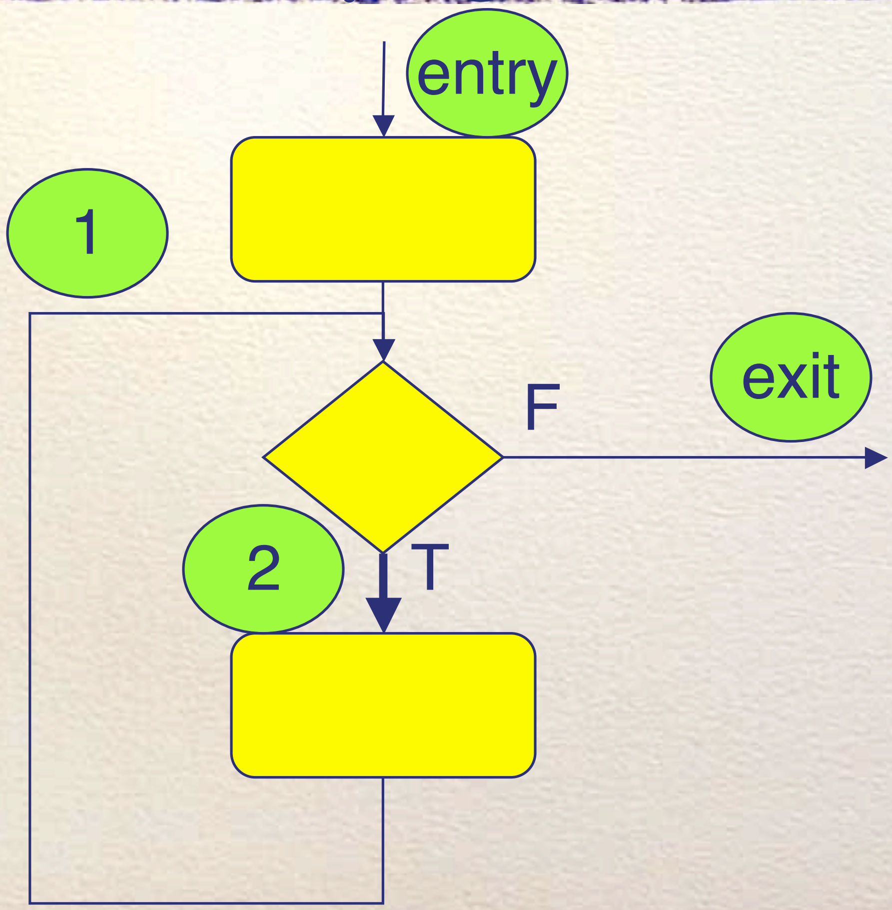
Identifying Place exit



Identifying Place 1



Identifying Place2



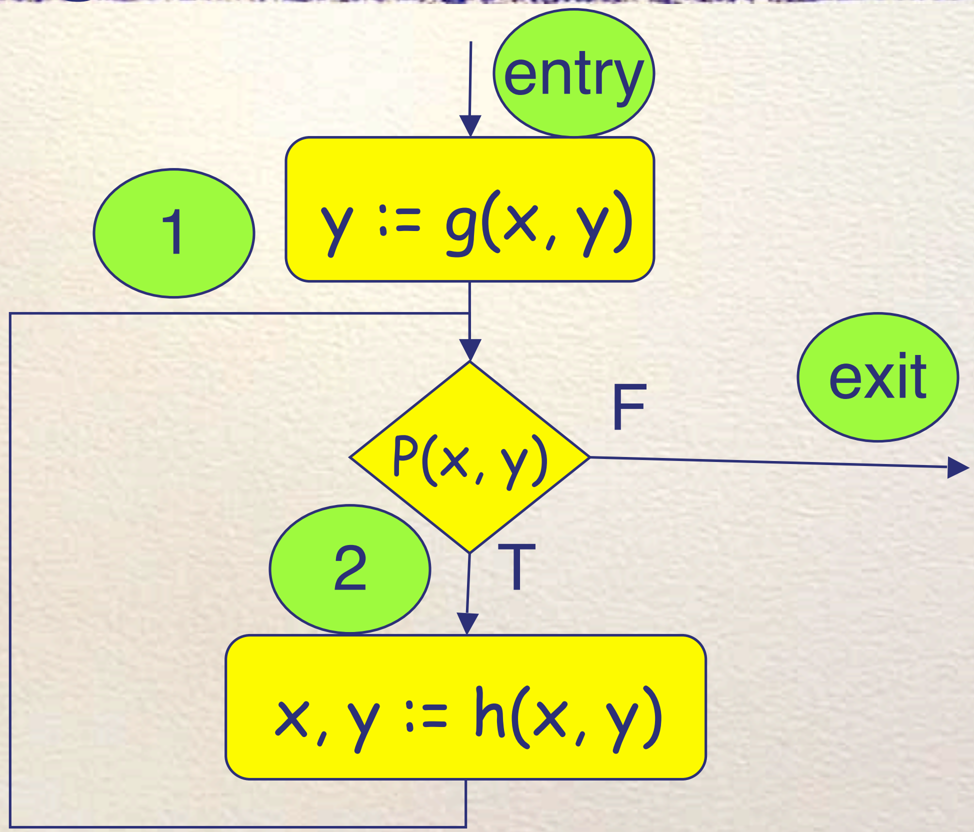
Construction

- For sake of illustration, suppose there are two variables x, y.
- Suppose the result is the value of y.

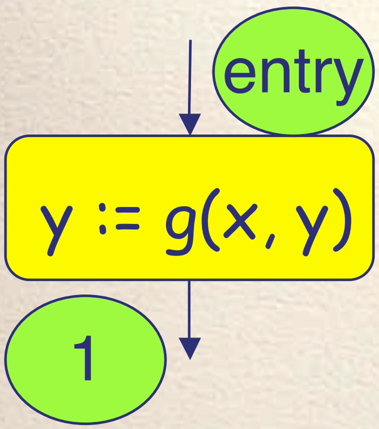
Functions at Each Place

- $f_{exit}(x, y) = y$, since y is the result.
- $f_{entry}(x, y)$ is **called with** the initial values of x and y, x_0, y_0. It is still to be defined.
- f_1 , f_2 still to be defined.

Filling in boxes, for example



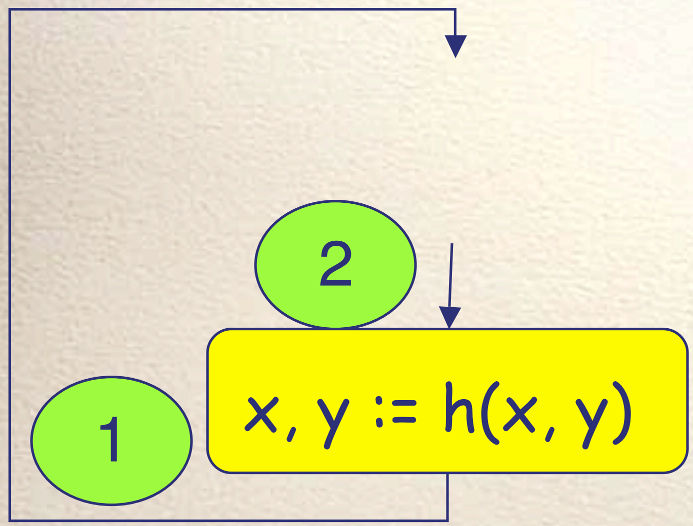
Construction of One Function for Each Place



$$f_{entry}(x, y) = f_1(x, g(x, y))$$

Interpretation: The place functions give the output if the computation were started at that place with specify values of x and y.

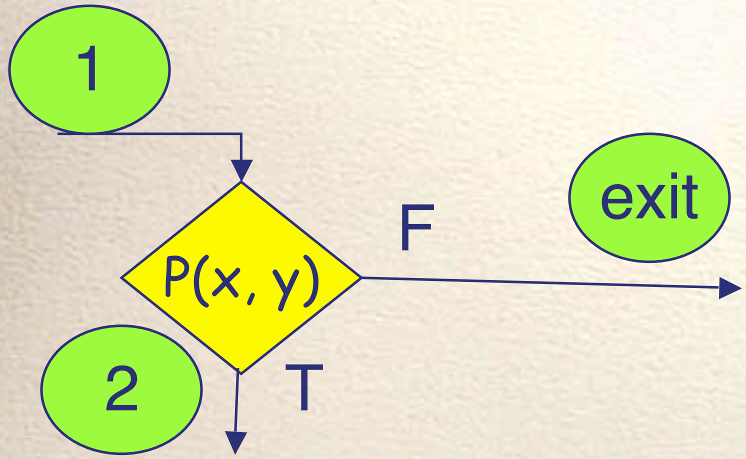
Construction



$$f_2(x, y) = f_1(h(x, y))$$

Interpretation: The place functions give the output if the computation were started at that place with specify values of x and y.

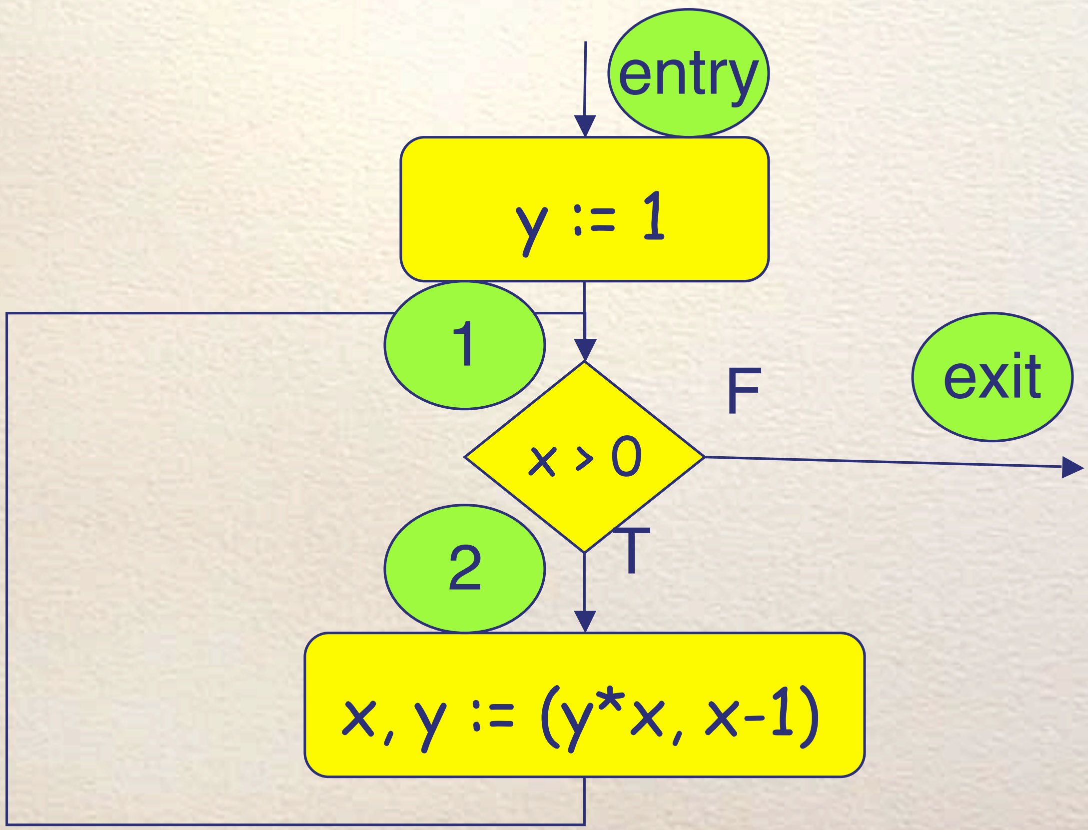
Construction



$$f_1(x, y) = \text{if } P(x, y) \text{ then } f_2(x, y) \text{ else } f_{exit}(x, y)$$

Interpretation: The place functions give the output if the computation were started at that place with specify values of x and y.

Specific Example



Specific Example

$f_{entry}(x) = f_1(x, 1)$

$f_1(x, y) = \text{if } x > 0 \text{ then } f_2(x, y) \text{ else } f_{exit}(x, y)$

$f_2(x, y) = f_1(x-1, y*x)$

$f_{exit}(x, y) = y$

Scheme Transcription

```
(define (fentry x) (f1 x 1))

(define (f1 x y) (if (> x 0) (f2 x y) (exit x y)))

(define (f2 x y) (f1 (- x 1) (* y x)))

(define (fexit x y) y)
```

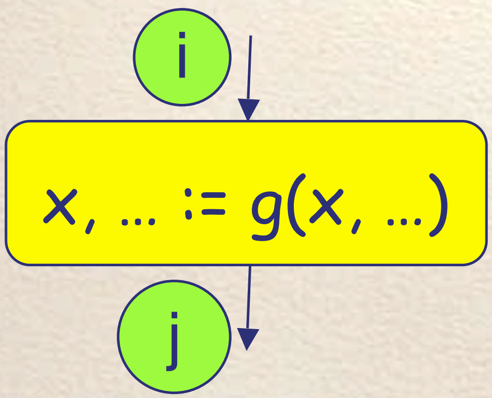
What function is computed by fentry?

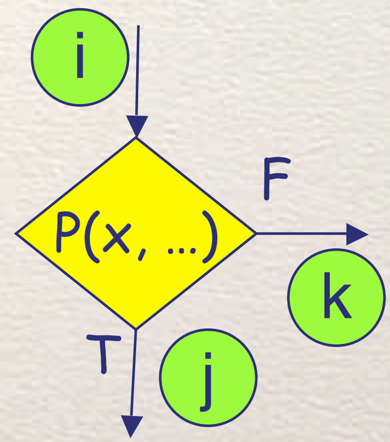
Notes

- Functions are mutually recursive.
- Functions are tail-recursive.
- Executing the function by rewriting simulates the execution of the imperative program.

Summary: Rules for McCarthy Construction

- $f_{entry}(x, \ldots)$ is **called with** the initial values
- $f_{exit}(x, \ldots)$ returns the final result





$f_i(x, \ldots) = f_j(g(x, \ldots))$

$$f_i(x, \ldots) = \text{if } P(x, \ldots) \text{ then } f_j(x, \ldots) \text{ else } f_k(x, \ldots)$$

McCarthy's Transformation

- transforms programming into pure mathematics

- Reference: McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Part I. Comm. ACM 3, 184–204 (1960)

Continuations
(allusion to a more advanced topic)

- The functions produced in McCarthy's transformation can be considered a special case of "continuations".

- A continuation is a function that represents what happens for the rest of the program, acting like a "receiver" of the the local result value.

- Programs, functional and otherwise, can be re-cast into a style known as continuation-passing style (CPS), in which **each function takes a continuation function** as an argument.

- The result can be obscure, but this is what many compilers do.

CPS Example

```
(define (factorial n)
 (if (= n 0)
     1
     (* n (factorial (- n 1)))))
```

continuation argument for factorial

```
(define (factorial n k)
  (= n 0
       (lambda (b)
       (if b
         (k 1)
         (- n 1 (lambda (nm1)
                 (factorial nm1 (lambda (f)
                                 (* n f k)))))))))
```

continuation argument for =

Scheme and Continuations

- Scheme was the first language to support CPS by providing **call/cc** (call with current continuation).
- This enables fancy tricks such as coroutines.
- For more information, take CS 131.

Handling Arrays in McCarthy's Transformation

- Array assignment has to be treated as if a new array were created, differing from the original only in the element modified.

a[i] := E

is like

a := a'

where a' is like a, except that
element i is the value of E

Arrays in Functional Programming

- Arrays themselves may be viewed as functions.
- The main issue is efficiency.
- Creating a new array at each step is expensive.
- Clever data structures (e.g. certain trees) can simulate array creation with less cost.

Iterating over Lists

- Although functions such as list-ref are available, they should not be used at each iteration when iterating over a list.

- It is better to plan the iteration so that combinations of first/rest are used.

Good Example

- Sum elements of a list.

- ```
  (define (sum L Acc)
      (if (null? L)
            Acc
            (sum (rest L) (+ (first L) Acc))))
  ```

Time: O(N) where N is the length of L
  ```

Bad Example

- Sum elements of a list using (sum L 0 0)

- ```
  (define (sum L index Acc)
      (if (= index (length L))
          Acc
          (sum L (+ 1 index)
               (+ (list-ref L index) Acc))))
  ```

Time: $O(N^2)$ where N is the length of L,
due to repeated use of: (a) length, (b) list-ref
  ```

Why is Array Access Faster then List Referencing?

- Also, when is fast array accessing a kind of fiction?