

Lecture 3

Functional Programming



T. METIN SEZGIN

Announcements



1. Etutor assignment due on Sunday midnight
2. Reading SICP 1.2 (pages 31-50)
3. Labs (PSes) started

Lecture 2

Functional Programming & Scheme



T. METIN SEZGIN

Main programming paradigms



<u>Paradigm</u>	Description	Main traits	Related paradigm(s)	Examples
<u>Imperative</u>	Programs as <u>statements</u> that <i>directly</i> change computed <u>state</u> (<u>datafields</u>)	Direct <u>assignments</u> , common <u>data structures</u> , <u>global variables</u>		<u>C</u> , <u>C++</u> , <u>Java</u> , <u>Kotlin</u> , <u>PHP</u> , <u>Python</u> , <u>Ruby</u>
<u>Procedural</u>	Derived from structured programming, based on the concept of <u>modular programming</u> or the <i>procedure call</i>	<u>Local variables</u> , sequence, selection, <u>iteration</u> , and <u>modularization</u>	Structured, imperative	<u>C</u> , <u>C++</u> , <u>Lisp</u> , <u>PHP</u> , <u>Python</u>
<u>Functional</u>	Treats <u>computation</u> as the evaluation of <u>mathematical functions</u> avoiding <u>state</u> and <u>mutable</u> data	<u>Lambda calculus</u> , <u>compositionality</u> , <u>formula</u> , <u>recursion</u> , <u>referential transparency</u> , no <u>side effects</u>	Declarative	<u>C++</u> , ^[1] <u>C#</u> , ^[2] <u>circular reference</u> <u>Clojure</u> , <u>CoffeeScript</u> , ^[3] <u>Elixir</u> , <u>Erlang</u> , <u>F#</u> , <u>Haskell</u> , <u>Java</u> (since version 8), <u>Kotlin</u> , <u>Lisp</u> , <u>Python</u> , <u>R</u> , ^[4] <u>Ruby</u> , <u>Scala</u> , <u>SequenceL</u> , <u>Standard ML</u> , <u>JavaScript</u> , <u>Elm</u>
<u>Object-oriented</u>	Treats <u>datafields</u> as <i>objects</i> manipulated through predefined <u>methods</u> only	<u>Objects</u> , methods, <u>message passing</u> , <u>information hiding</u> , <u>data abstraction</u> , <u>encapsulation</u> , <u>polymorphism</u> , <u>inheritance</u> , <u>serialization-marshalling</u>	Procedural	<u>Common Lisp</u> , <u>C++</u> , <u>C#</u> , <u>Eiffel</u> , <u>Java</u> , <u>Kotlin</u> , <u>PHP</u> , <u>Python</u> , <u>Ruby</u> , <u>Scala</u> , <u>JavaScript</u> ^[5]
<u>Declarative</u>	Defines program logic, but not detailed <u>control flow</u>	<u>Fourth-generation languages</u> , <u>spreadsheets</u> , <u>report program generators</u>		<u>SQL</u> , <u>regular expressions</u> , <u>Prolog</u> , <u>OWL</u> , <u>SPARQL</u> , <u>Datalog</u> , <u>XSLT</u>

Write a function for factorial



Kinds of Language Constructs

- Primitives
- Means of combination
- Means of abstraction

Kinds of Language Constructs

- Primitives (integers, float, + - as. primitive procedure)
- Means of Combination (tic = tic + 1)
- Means of Abstraction (-Naming)

```
def create_adder(x):  
    global tic  
    tic = x  
  
    def adder():  
        global tic  
        tic = tic + 1  
        return tic  
  
    return adder  
  
fun_a = create_adder(0)  
fun_b = create_adder(0)  
  
print(fun_a(), fun_b(), fun_a(), fun_b())
```

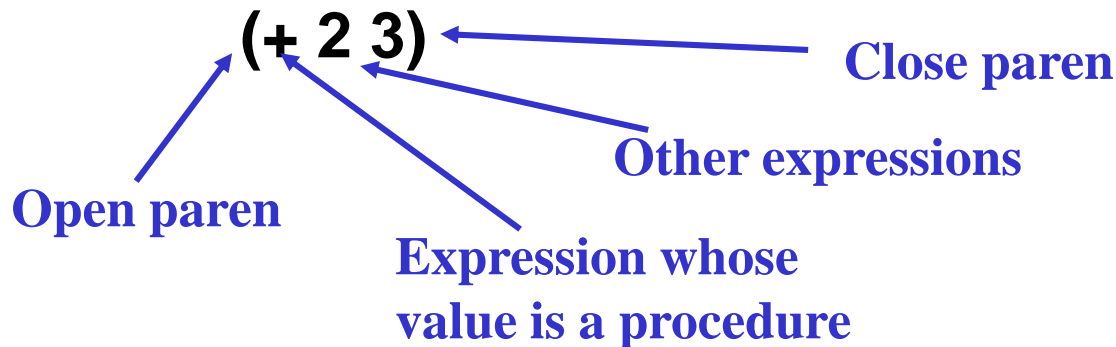
Omer Veysel Cagatan

Language elements – primitives

- Names for built-in procedures
 - $+$, $*$, $-$, $/$, $=$, ...
 - What is the value of such an expression?
 - $+$ \rightarrow [#procedure ...]
 - Evaluate by looking up value associated with name in a special table

Language elements – combinations

- How do we create expressions using these procedures?



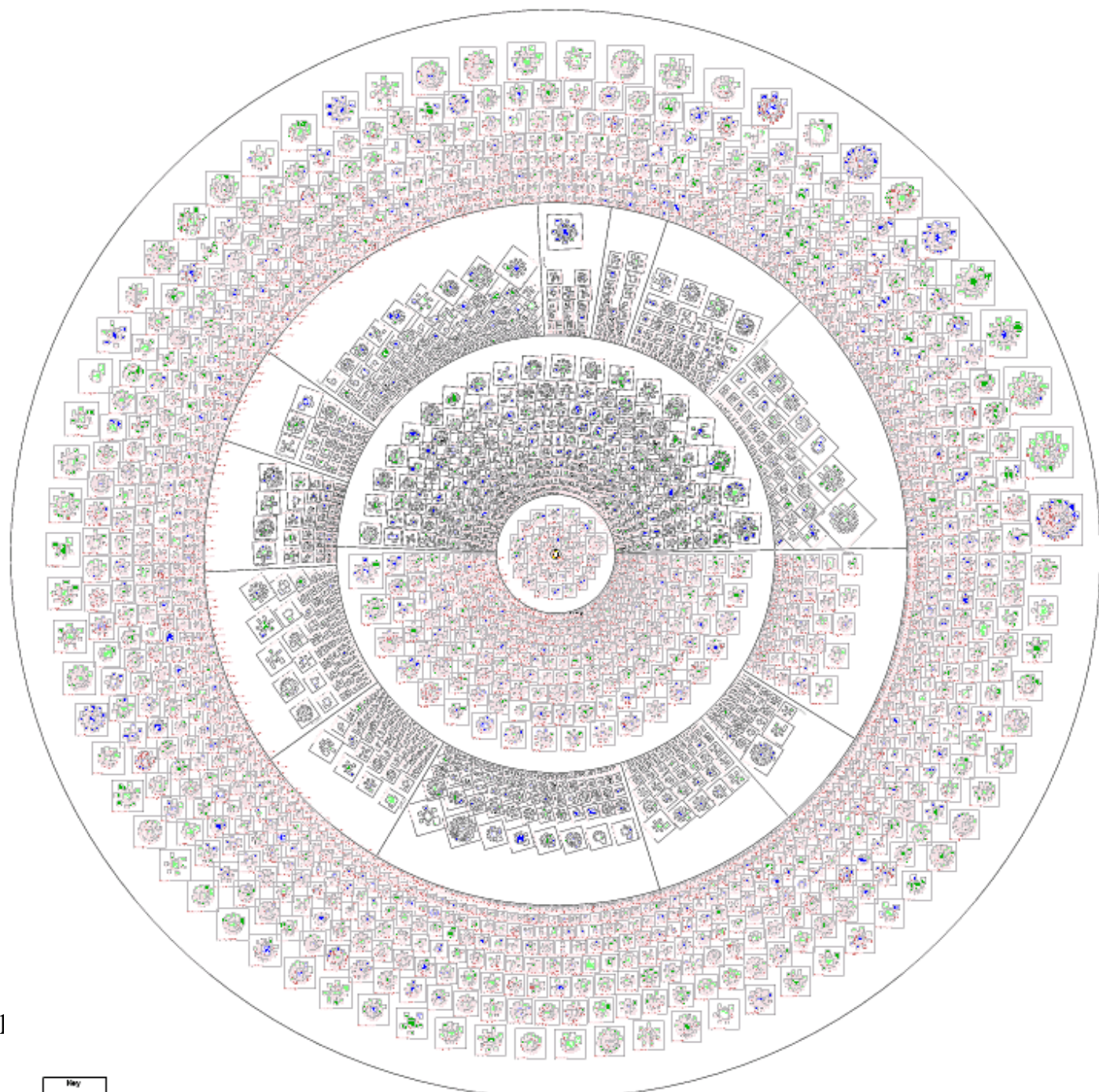
- Evaluate by getting values of sub-expressions, then applying operator to values of arguments

Language elements -- abstractions

- In order to abstract an expression, need way to give it a name

(define score 23)

- This is a special form
 - Does not evaluate second expression
 - Rather, it pairs name with value of the third expression
- Return value is unspecified



Nugget

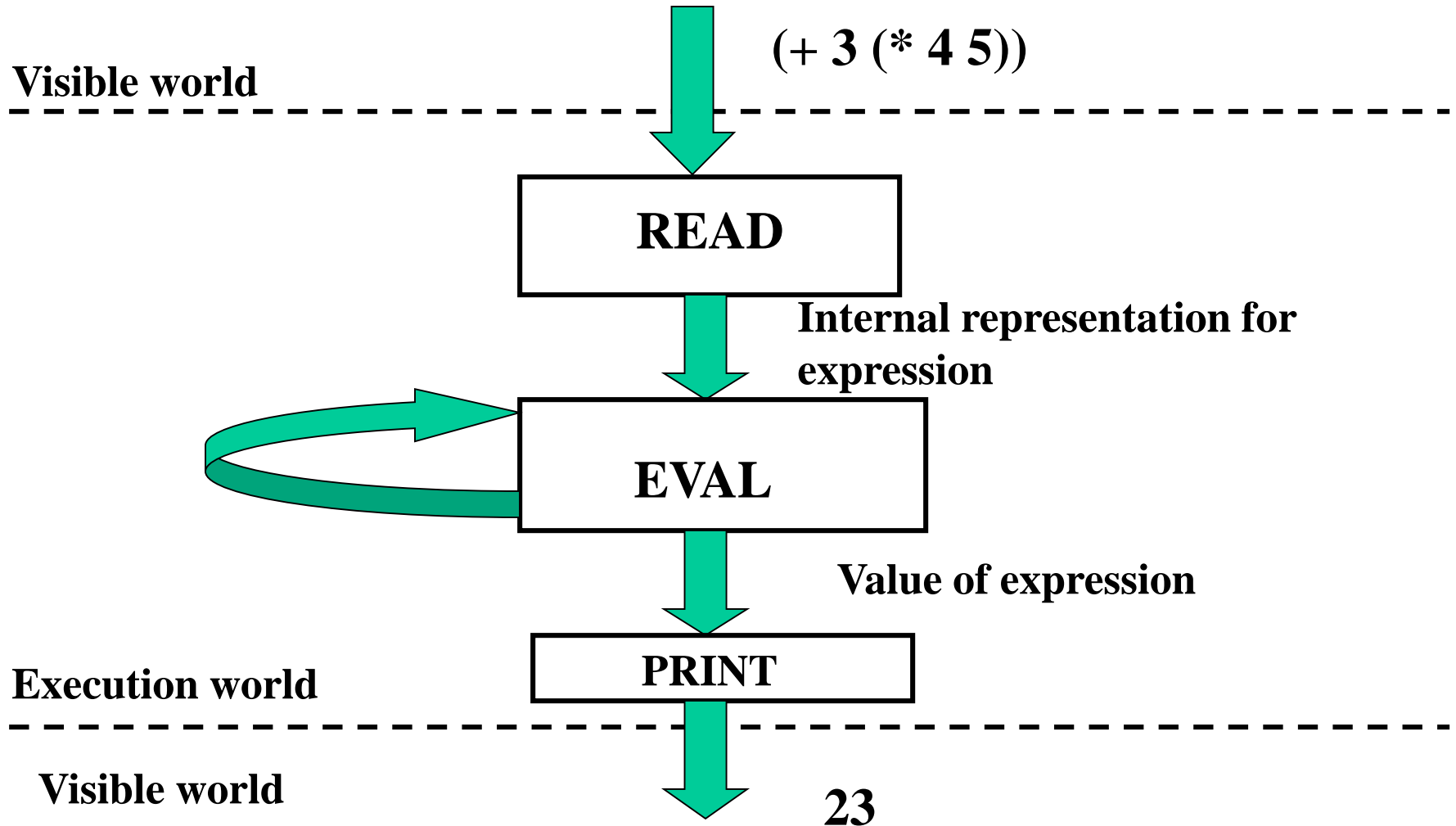


Functions are first class citizens

Scheme Basics

- Rules for evaluation
 1. If **self-evaluating**, return value.
 2. If a **name**, return value associated with name in environment.
 3. If a **special form**, do something special.
 4. If a **combination**, then
 - a. *Evaluate* all of the subexpressions of combination (in any order)
 - b. *apply* the operator to the values of the operands (arguments) and return result
- Rules for application
 1. If procedure is **primitive procedure**, just do it.
 2. If procedure is a **compound procedure**, then:
evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

Read-Eval-Print



Lecture 3

Functional Programming



T. METIN SEZGIN

Lecture Nuggets



- Lambda expressions create procedures
 - Formal parameters
 - Body
 - Procedures allow creating abstractions
- We can solve problems by creating functions
- The substitution model is a good mental model of an interpreter

Hold your breath



Language elements -- abstractions

- Need to capture ways of doing things – use procedures

(lambda (x) (* x x))

The diagram shows the lambda expression **(lambda (x) (* x x))** with several annotations. A red arrow points from the word **parameters** to the opening parenthesis of the parameter list **(x)**. Another red arrow points from the word **body** to the opening parenthesis of the body **(* x x)**. Two blue arrows point upwards: one from the word **To process** to the **lambda** keyword, and another from the word **something** to the parameter **x**. A third blue arrow points from the phrase **multiply it by itself** to the first **x** in the body **(* x x)**.

- Special form – creates a procedure and returns it as value

Nugget



Lambda expressions creates
procedures

Language elements -- abstractions

- Use this anywhere you would use a procedure

((lambda (x) (* x x)) 5)

(* 5 5)

25

- Can give it a name

(define square (lambda (x) (* x x)))

(square 5) → 25

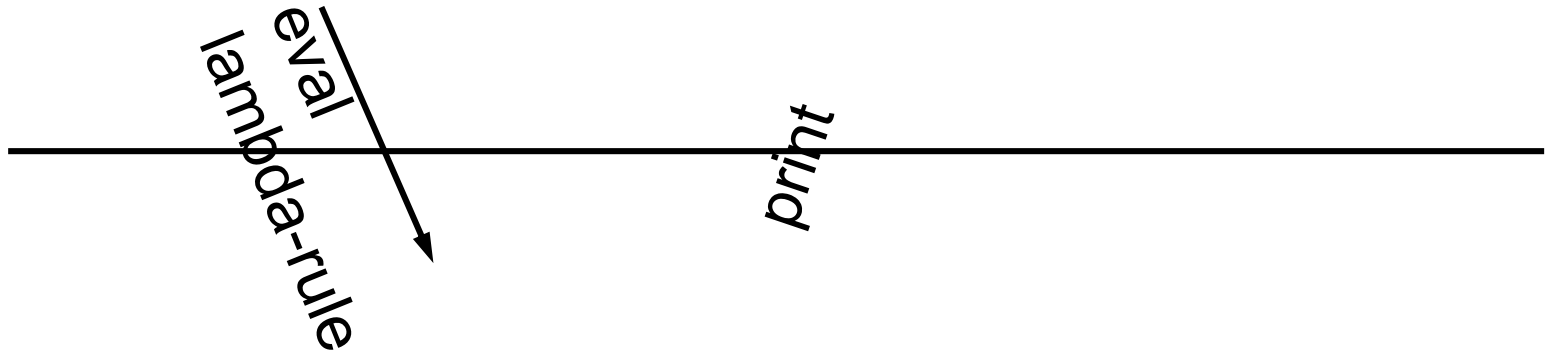
Lambda: making new procedures

expression

printed representation of value

`(lambda (x) (* x x))`

`#[compound-procedure 9]`



A compound proc
that squares its
argument

value

Interaction of define and lambda

1. `(lambda (x) (* x x))`
 `==> #[compound-procedure 9]`
2. `(define square (lambda (x) (* x x)))`
 `==> undef`
3. `(square 4)` `==> 16`
4. `((lambda (x) (* x x)) 4)` `==> 16`
5. `(define (square x) (* x x))` `==> undef`

This is a convenient shorthand (called “syntactic sugar”) for 2 above – this is a use of lambda!

Lambda special form

- lambda syntax `(lambda (x y) (/ (+ x y) 2))`
- 1st operand position: the **parameter list** `(x y)`
 - a list of names (perhaps empty)
 - determines the number of operands required
- 2nd operand position: the **body** `(/ (+ x y) 2)`
 - may be any expression
 - not evaluated when the lambda is evaluated
 - evaluated when the procedure is applied
- semantics of lambda:

**THE VALUE OF
A LAMBDA EXPRESSION
IS
A PROCEDURE**

Nugget



We can solve problems by creating
functions

Procedures allow abstraction

- Breaking computation into modules that capture commonality
 - Enables reuse in other places (e.g. square)
- Isolates details of computation within a procedure from use of the procedure
- May be many ways to divide up

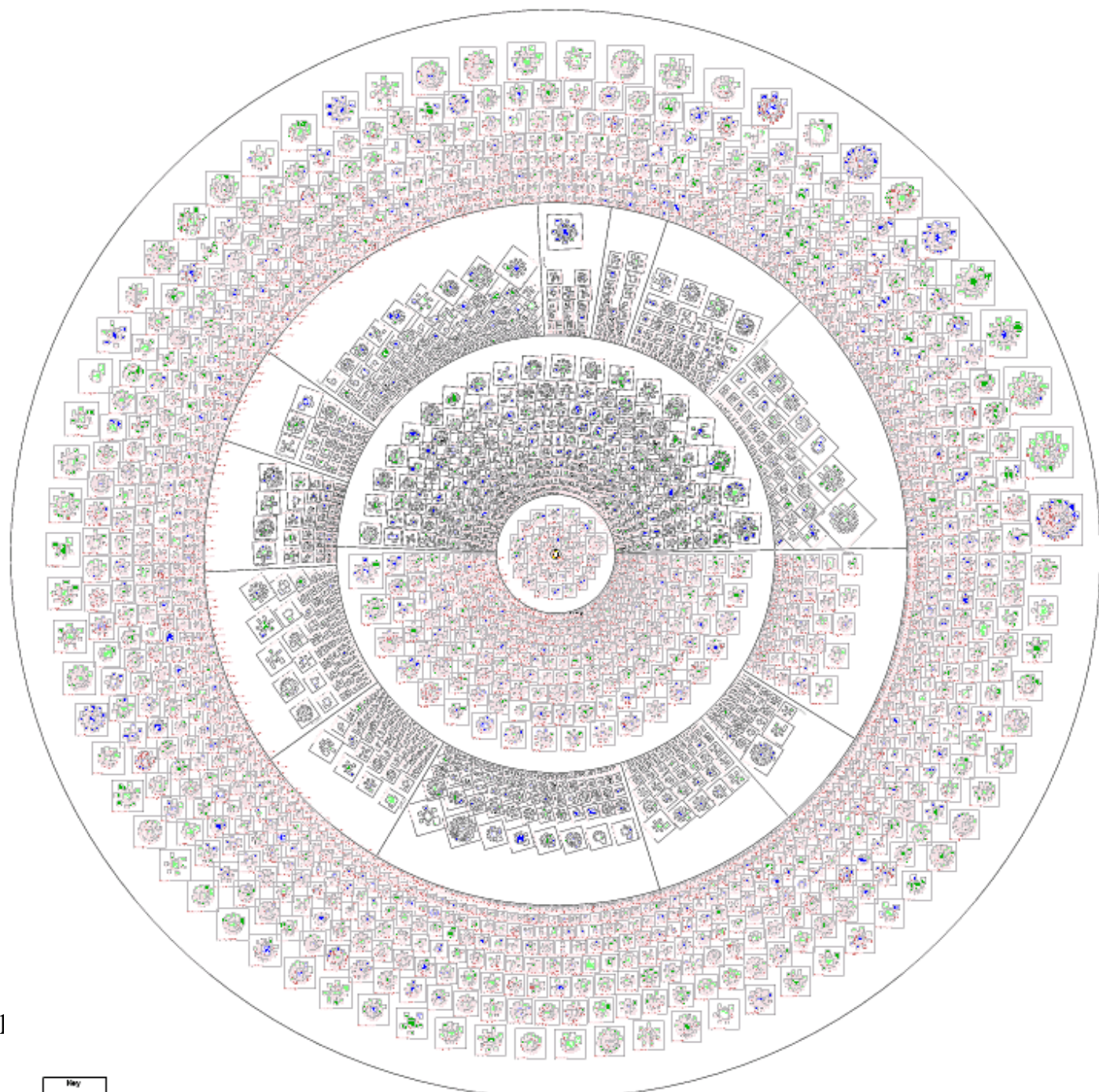
```
(define square (lambda (x) (* x x)))
```

```
(define sum-squares
```

```
  (lambda (x y) (+ (square x) (square y))))
```

```
(define pythagoras
```

```
  (lambda (y x) (sqrt (sum-squares y x))))
```



Abstracting the process

- Stages in capturing common patterns of computation
 - Identify modules or stages of process
 - Capture each module within a procedural abstraction
 - Construct a procedure to control the interactions between the modules
 - Repeat the process within each module as necessary

A more complex example

- Remember our method for finding sqrts
 - To find the square root of X
 - Make a guess, called G
 - If G is close enough, stop
 - Else make a new guess by averaging G and X/G

Imperative Knowledge

- “How to” knowledge

To find an approximation of square root of x :

- Make a guess G
- Improve the guess by averaging G and x/G
- Keep improving the guess until it is good enough

Example : \sqrt{x} for $x = 2$.

$X = 2$	$G = 1$
$X/G = 2$	$G = \frac{1}{2} (1 + 2) = 1.5$
$X/G = 4/3$	$G = \frac{1}{2} (3/2 + 4/3) = 17/12 = 1.416666$
$X/G = 24/17$	$G = \frac{1}{2} (17/12 + 24/17) = 577/408 = 1.4142156$

The stages of “SQRT”

- When is something “close enough”
- How do we create a new guess
- How to we control the process of using the new guess in place of the old one

Procedural abstractions

For “close enough”:

```
(define close-enuf?  
  (lambda (guess x)  
    (< (abs (- (square guess) x)) 0.001)))
```



Note use of procedural abstraction!

Procedural abstractions

For “improve”:

```
(define average  
  (lambda (a b) (/ (+ a b) 2)))  
  
(define improve  
  (lambda (guess x)  
    (average guess (/ x guess)))))
```


Why this modularity?

- “Average” is something we are likely to want in other computations, so only need to create once
- Abstraction lets us separate implementation details from use
 - E.g. could redefine as

```
(define average  
  (lambda (x y) (* (+ x y) 0.5)))
```

- No other changes needed to procedures that use **average**
- Also note that variables (or parameters) are internal to procedure – cannot be referred to by name outside of scope of lambda

Controlling the process

- Basic idea:
 - Given X , G , want **(improve G X)** as new guess
 - Need to make a decision – for this need a new *special form*

(if <predicate> <consequence> <alternative>)

The IF special form

(if <predicate> <consequence> <alternative>)

- Evaluator first evaluates the <predicate> expression.
- If it evaluates to a TRUE value, then the evaluator evaluates and returns the value of the <consequence> expression.
- Otherwise, it evaluates and returns the value of the <alternative> expression.
- Why must this be a special form?

Controlling the process

- Basic idea:
 - Given X, G, want **(improve G X)** as new guess
 - Need to make a decision – for this need a new *special form*
(if <predicate> <consequence> <alternative>)
 - So heart of process should be:

```
(if (close-enuf? G X)
    G
    (improve G X))
```

- But somehow we want to use the value returned by “improving” things as the new guess, and repeat the process

Controlling the process

- Basic idea:
 - Given X, G, want **(improve G X)** as new guess
 - Need to make a decision – for this need a new *special form*
(if <predicate> <consequence> <alternative>)
 - So heart of process should be:
(define sqrt-loop (lambda (G X)
 (if (close-enuf? G X)
 G
 (sqrt-loop (improve G X) X) **)**
 - But somehow we want to use the value returned by “improving” things as the new guess, and repeat the process
 - Call process **sqrt-loop** and reuse it!

Putting it together

- Then we can create our procedure, by simply starting with some initial guess:

```
(define sqrt  
  (lambda (x)  
    (sqrt-loop 1.0 x)))
```

Checking that it does the “right thing”

- Next lecture, we will see a formal way of tracing evolution of evaluation process
- For now, just walk through basic steps

– **(sqrt 2)**

- **(sqrt-loop 1.0 2)**
- **(if (close-enuf? 1.0 2))**
- **(sqrt-loop (improve 1.0 2) 2)**

This is just like a normal combination

- **(sqrt-loop 1.5 2)**
- **(if (close-enuf? 1.5 2))**
- **(sqrt-loop 1.4166666 2)**

- **And so on...**

Nugget



The substitution model is a good
mental model of an interpreter

Remainder of this lecture

- Substitution model
- An example using the substitution model
- Designing recursive procedures
- Designing iterative procedures



Substitution model

- a way to figure out what happens during evaluation
 - not really what happens in the computer
- to apply a compound procedure:
 - evaluate the body of the procedure, with each parameter replaced by the corresponding operand
- to apply a primitive procedure: just do it

```
(define square (lambda (x) (* x x)))
```

```
1.      (square 4)
2.      (* 4 4)
3.      16
```

Substitution model details

```
(define square (lambda (x) (* x x)))  
(define average (lambda (x y) (/ (+ x y) 2)))
```

```
(average 5 (square 3))
```

```
(average 5 (* 3 3))
```

```
(average 5 9)
```

first evaluate operands,
then substitute (applicative order)

```
(/ (+ 5 9) 2)
```

```
(/ 14 2)
```

if operator is a primitive procedure,
replace by result of operation

7

End of part 1

- how to use substitution model to trace evaluation

A less trivial procedure: factorial

- Compute n factorial, defined as $n! = n(n-1)(n-2)(n-3)\dots 1$
- Notice that $n! = n * [(n-1)(n-2)\dots] = n * (n-1)! \quad \text{if } n > 1$

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
```

- predicate = tests numerical equality

`(= 4 4) ==> #t` (true)

`(= 4 5) ==> #f` (false)

- if special form

`(if (= 4 4) 2 3) ==> 2`

`(if (= 4 5) 2 3) ==> 3`


predicate consequent alternative

```
(define fact(lambda (n)
  (if (= n 1) 1 (* n (fact (- n 1))))))
```

```
(fact 3)
```

```
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
```

```
(if #f 1 (* 3 (fact (- 3 1))))
```

```
(* 3 (fact (- 3 1)))
```

```
(* 3 (fact 2))
```

```
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
```

```
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
```

```
(* 3 (* 2 (fact (- 2 1))))
```

```
(* 3 (* 2 (fact 1)))
```

```
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
```

```
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1))))))
```

```
(* 3 (* 2 1))
```

```
(* 3 2)
```

The fact procedure is a recursive algorithm

- A recursive algorithm:
 - In the substitution model, the expression keeps growing

```
(fact 3)  
(* 3 (fact 2))  
(* 3 (* 2 (fact 1)))
```
 - Other ways to identify will be described next time

End of part 2

- how to use substitution model to trace evaluation
- how to recognize a recursive procedure in the trace

How to design recursive algorithms

- follow the general pattern:
 1. wishful thinking
 2. decompose the problem
 3. identify non-decomposable (smallest) problems

1. Wishful thinking

- Assume the desired procedure exists.
- want to implement fact? OK, assume it exists.
- BUT, only solves a smaller version of the problem.

2. Decompose the problem

- Solve a problem by
 1. solve a smaller instance (using wishful thinking)
 2. convert that solution to the desired solution
- Step 2 requires creativity!
 - Must design the strategy before coding.
 - $n! = n(n-1)(n-2)\dots = n[(n-1)(n-2)\dots] = n * (n-1)!$
 - solve the smaller instance, multiply it by n to get solution

```
(define fact  
  (lambda (n) (* n (fact (- n 1)))))
```

3. Identify non-decomposable problems

- Decomposing not enough by itself
- Must identify the "smallest" problems and solve directly
- Define $1! = 1$

```
(define fact
  (lambda (n)
    (if (= n 1) 1
        (* n (fact (- n 1))))))
```

General form of recursive algorithms

- test, base case, recursive case

```
(define fact
  (lambda (n)
    (if (= n 1)          ; test for base case
        1                ; base case
        (* n (fact (- n 1)) ; recursive case
    )))
```

- base case: smallest (non-decomposable) problem
- recursive case: larger (decomposable) problem

End of part 3

- Design a recursive algorithm by
 1. wishful thinking
 2. decompose the problem
 3. identify non-decomposable (smallest) problems
- Recursive algorithms have
 1. test
 2. recursive case
 3. base case

Iterative algorithms

- In a recursive algorithm, bigger operands => more space

```
(define fact (lambda (n)
  (if (= n 1) 1
      (* n (fact (- n 1))))))

(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
...
24
```

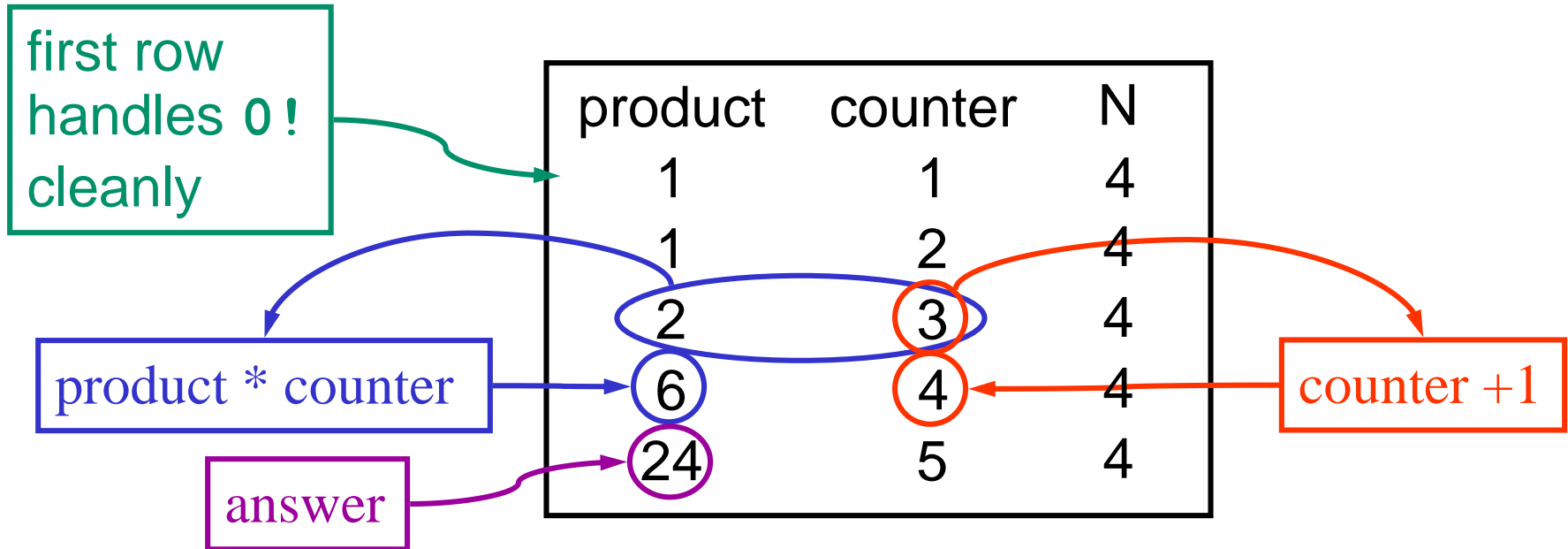
- An iterative algorithm uses **constant space**

Intuition for iterative factorial

- same as you would do if calculating $4!$ by hand:
 1. multiply 4 by 3 gives 12
 2. multiply 12 by 2 gives 24
 3. multiply 24 by 1 gives 24
- At each step, only need to remember:
previous product, next multiplier
- Therefore, constant space
- Because multiplication is associative and commutative:
 1. multiply 1 by 2 gives 2
 2. multiply 2 by 3 gives 6
 3. multiply 6 by 4 gives 24

Iterative algorithm to compute 4! as a table

- In this table:
 - One column for each piece of information used
 - One row for each step



- The last row is the one where counter > n
- The answer is in the product column of the last row

Iterative factorial in scheme

- (define ifact (lambda (n) (ifact-helper 1 1 n)))

initial
row of table

(define ifact-helper (lambda (product counter n)

(if (> counter n)

product

compute next row of table

(ifact-helper (* product counter) (+ counter 1) n))))

answer is in product column of last row

at last row when counter > n

Partial trace for (ifact 4)

```
(define ifact-helper (lambda (product count n)
  (if (> count n) product
      (ifact-helper (* product count)
                     (+ count 1) n))))
```

```
(ifact 4)
(ifact-helper 1 1 4)
(if (> 1 4) 1 (ifact-helper (* 1 1) (+ 1 1) 4))
(ifact-helper 1 2 4)
(if (> 2 4) 1 (ifact-helper (* 1 2) (+ 2 1) 4))
(ifact-helper 2 3 4)
(if (> 3 4) 2 (ifact-helper (* 2 3) (+ 3 1) 4))
(ifact-helper 6 4 4)
(if (> 4 4) 6 (ifact-helper (* 6 4) (+ 4 1) 4))
(ifact-helper 24 5 4)
(if (> 5 4) 24 (ifact-helper (* 24 5) (+ 5 1) 4))
24
```

Iterative = no pending operations when procedure calls itself

- Recursive factorial:

```
(define fact (lambda (n)
  (if (= n 1) 1
      (* n (fact (- n 1)) )
  )))
```



pending operation

- ```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
```

- Pending ops make the expression grow continuously

# Iterative = no pending operations

- Iterative factorial:

```
(define ifact-helper (lambda (product count n)
 (if (> count n) product
 (ifact-helper (* product count)
 (+ count 1) n))))
```

- ```
(ifact-helper 1 1 4)
```



```
(ifact-helper 1 2 4)
```

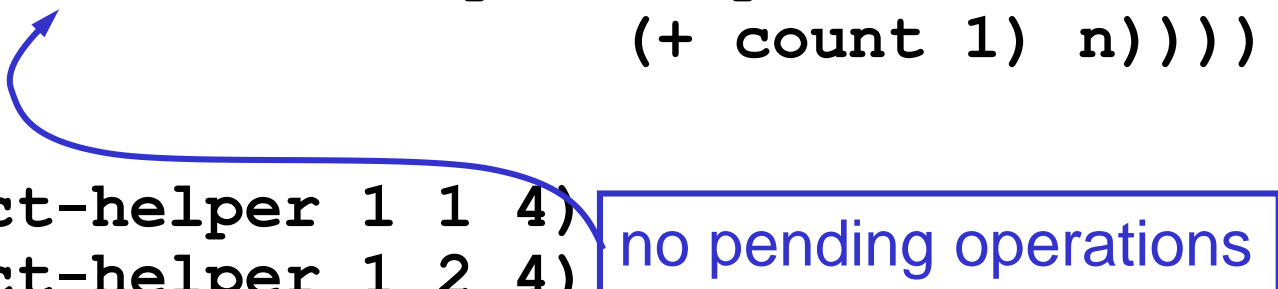


```
(ifact-helper 2 3 4)
```



```
(ifact-helper 6 4 4)
```



```
(ifact-helper 24 5 4)
```
- no pending operations
- 

- Fixed size because no pending operations

End of part 4

- Iterative algorithms have constant space
- How to develop an iterative algorithm
 - figure out a way to accumulate partial answers
 - write out a table to analyze precisely:
 - initialization of first row
 - update rules for other rows
 - how to know when to stop
 - translate rules into scheme code
- Iterative algorithms have no pending operations when the procedure calls itself

Announcements

1. Reading SICP 1.2 (pages 31-50)
2. Etutor assignment due Sunday midnight
3. Labs (PSes) started already