Lecture 3
Functional Programming

T. METIN SEZGIN

1

Announcements

1. Assignment due on Friday
2. Reading SICP 1.2 (pages 31-50)
3. Etutor assignment due Friday 8th
4. Labs (PSes) start this week

2

2

Lecture 2
Functional Programming & Scheme

T. METIN SEZGIN

3

Main programming paradigms

Paradigm	Description	Main traits	Related paradigm(s)	Examples
Imperative	Programs as statements that *directly* change computed state (datafields)	Direct assignments, common data structures, global variables		C, C++, Java, Kotlin, PHP, Python, Ruby
Procedural	Derived from structured programming, based on the concept of modular programming or the *procedure call*	Local variables, sequence, selection, iteration, and modularization	Structured, imperative	C, C++, Lisp, PHP, Python
Functional	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Lambda calculus, compositionality, formula, recursion, referential transparency, no side effects	Declarative	C++,[1] C#,[2][*circular reference*] Clojure, CoffeeScript,[3] Elixir, Erlang, F#, Haskell, Java (since version 8), Kotlin, Lisp, Python, R,[4] Ruby, Scala, SequenceL, Standard ML, JavaScript, Elm
Object-oriented	Treats datafields as *objects* manipulated through predefined methods only	Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance, serialization-marshalling	Procedural	Common Lisp, C++, C#, Eiffel, Java, Kotlin, PHP, Python, Ruby, Scala, JavaScript[8][9]
Declarative	Defines program logic, but not detailed control flow	Fourth-generation languages, spreadsheets, report program generators		SQL, regular expressions, Prolog, OWL, SPARQL, Datalog, XSLT

Source: Wikipedia

4

Write a function for factorial

5

Kinds of Language Constructs

- Primitives
- Means of combination
- Means of abstraction

```python
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())
```


6

Language elements – primitives

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

Language elements – combinations

- How do we create expressions using these procedures?

(+ 2 3)

Close paren

Other expressions

Open paren

Expression whose value is a procedure

- 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

Hold your breath

11

Language elements -- abstractions

- Need to capture ways of doing things – use procedures

parameters

(lambda (x) (* x x)) body

To process something multiply it by itself

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

12

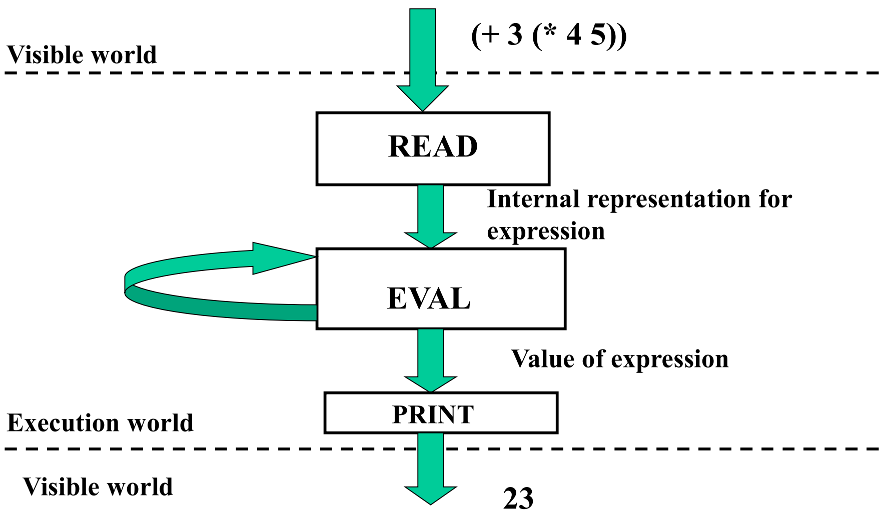
Scheme Basics

- 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.

13

Read-Eval-Print



14

Lecture 3
Functional Programming

T. METIN SEZGIN

15

Lecture Nuggets

- Lambda expressions creates 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

16

Nugget

Lambda expressions creates procedures

17

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

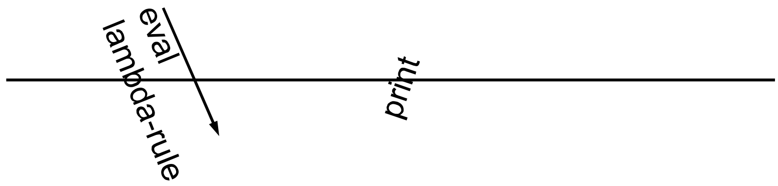
18

Lambda: making new procedures

expression

printed representation of value

```
(lambda (x) (* x x))      #[compound-procedure 9]
```



A compound proc that squares its argument

value

19

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!

20

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

23

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))))
```


24

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 = ½ (1+ 2) = 1.5
X/G = 4/3	G = ½ (3/2 + 4/3) = 17/12 = 1.416666
X/G = 24/17	G = ½ (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

31

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>)
```


32

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)
7
```

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

COMP 301 SICP 41

41

End of part 1

- how to use substitution model to trace evaluation

COMP 301 SICP 42

42

A less trivial procedure: factorial

- Compute n factorial, defined as n! = n(n-1)(n-2)(n-3)...1

•Notice that n! = n * [(n-1)(n-2)...] = n * (n-1)! 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

43

43

```
(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)
6
```

COMP 301 SICP

44

44

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

45

End of part 2

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

46

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)... = n[(n-1)(n-2)...] = 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))))))
```

COMP 301 SICP

49

49

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

COMP 301 SICP

50

50

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

COMP 301 SICP 53

53

Iterative algorithm to compute 4! as a table

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

first row handles 0! cleanly

product * counter

answer

counter +1

product	counter	N
1	1	4
1	2	4
2	3	4
6	4	4
24	5	4

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

COMP 301 SICP 54

54

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)

compute next row of table

product

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

answer is in product column of last row

at last row when counter > n

COMP 301 SICP 55

55

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
```

COMP 301 SICP 56

56

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 continuosly

COMP 301 SICP

57

57

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

COMP 301 SICP

58

58

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. Assignment due on Friday
2. Reading SICP 1.2 (pages 31-50)
3. Etutor assignment due Friday 8th
4. Labs (PSes) start this week