CS 4
Fundamentals of Computer Programming

Lecture 2: January 7, 2015

The substitution model

Last time

- Basics of the Scheme programming language
  - syntax
  - expressions
  - functions (`lambda` expressions)
  - definitions

Today

- An explicit model for how expressions are evaluated: the *substitution model*
- Booleans and conditional expressions

Precision

- Human (natural) language is imprecise
  - full of ambiguity
- Computer languages must be precise
  - only one meaning
  - ambiguity is unacceptable

Model

- Today, we'll determine the meaning of Scheme expressions
  - by developing a precise model for interpreting them
- This model is called the Substitution Model
- *Goal:* To have a model so precise, we could write a computer program to implement it!

Scheme combinations

- Recall:

```
(operator operand1 operand2 ...)
```

  - delimited by parentheses
  - first element is the **operator**
  - rest are **operands**

Some simple combinations

- `(+ 2 3)`
- `(- 4 1)`
- `(+ 2 3 4)`
- `(abs -7)`
- `(abs (+ (- x1 x2) (- y1 y2)))`

Meaning?

- What does a Scheme expression *mean*?
- In other words:
- How do we know what *value* will be calculated by a Scheme expression?

Substitution Model

The basic rule:

To evaluate a Scheme expression:

1. Evaluate its **operands**
2. Evaluate the **operator**
3. Apply the operator to the evaluated operands

Warning! Boredom ahead!

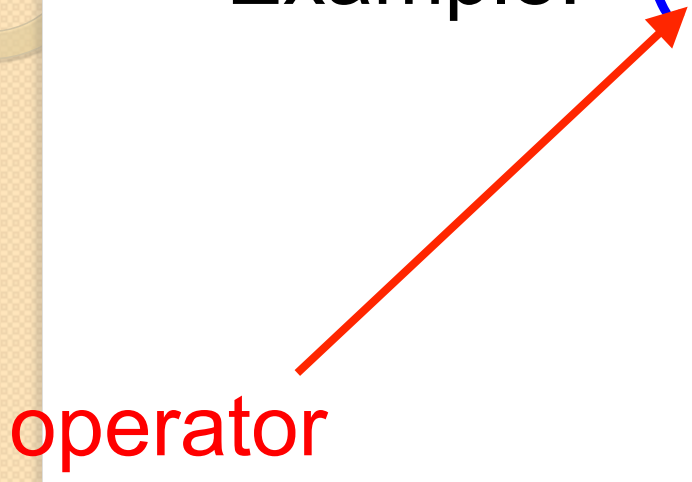
- We will walk through some evaluations
  - in painful, excruciating detail
- *Not* the most exciting part of course
- But necessary to give a *precise* model
  - which is what the computer needs in order to be able to compute anything

Example expression evaluation

- Example: `(+ 3 (* 4 5))`

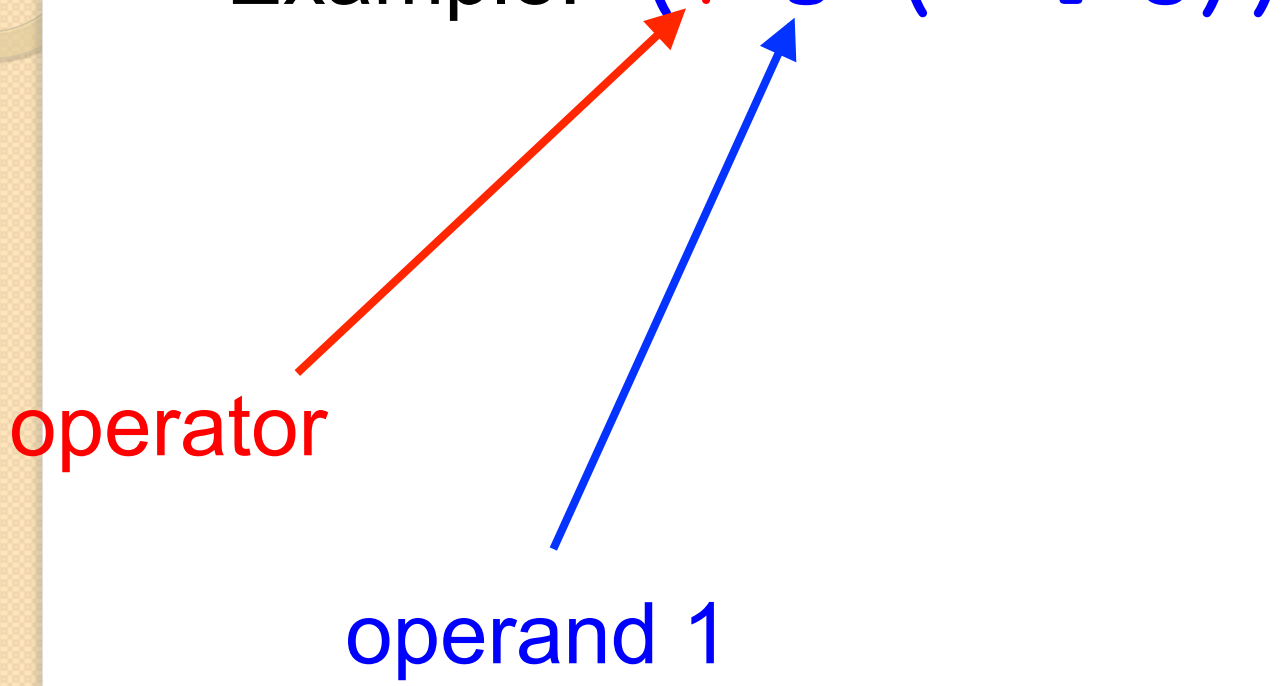
Example expression evaluation

- Example: `(+ 3 (* 4 5))`



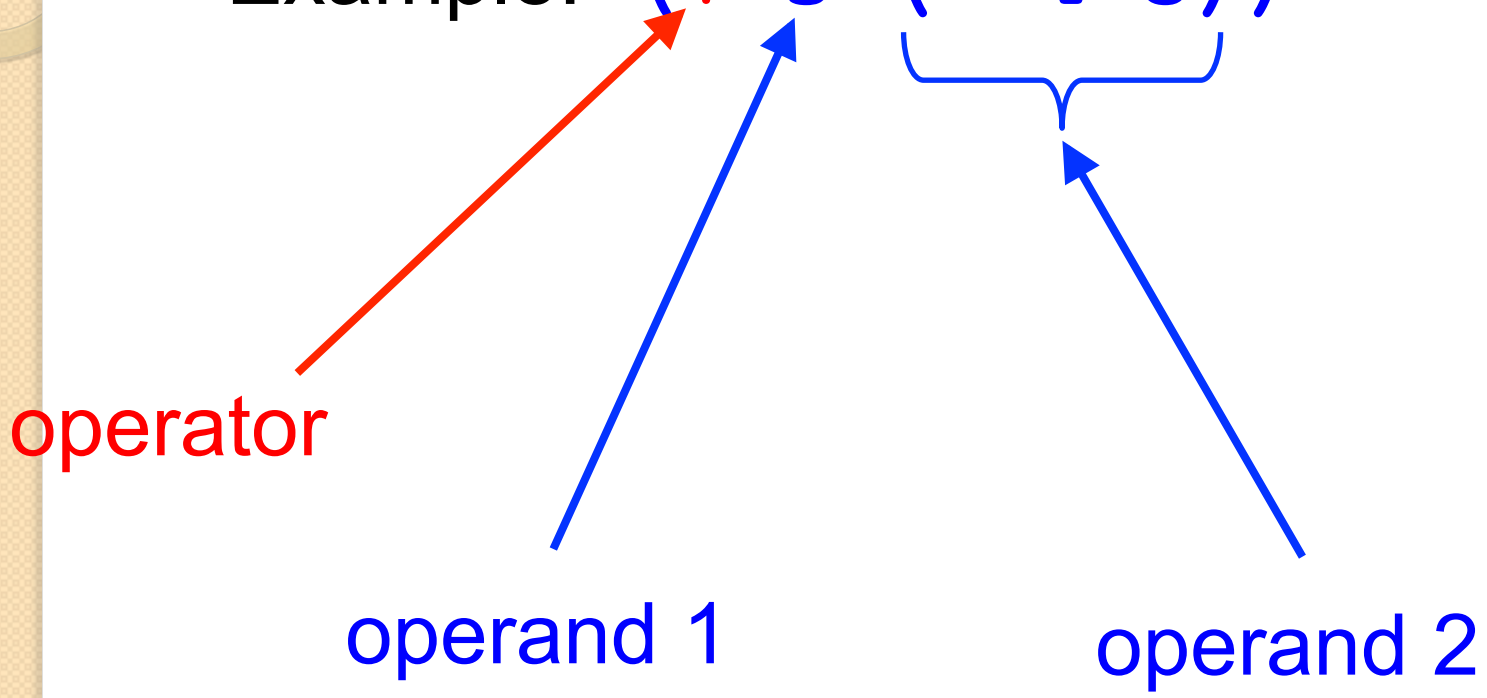
Example expression evaluation

- Example: `(+ 3 (* 4 5))`



Example expression evaluation

- Example: `(+ 3 (* 4 5))`



Example expression evaluation

- Note:
  - an operand can be a combination
  - or a number
- Here:
  - first operand: `3` (number)
  - second operand: `(* 4 5)` (combination)

Example expression evaluation

- Evaluate: `(+ 3 (* 4 5))`
  - evaluate `3`
  - evaluate `(* 4 5)`
    - evaluate `4`
    - evaluate `5`
    - evaluate `*`
    - apply `*` to `4`, `5` → `20`
  - evaluate `+`
  - apply `+` to `3`, `20` → `23`

Primitive expressions

- Numbers evaluate to themselves
  - `105` → `105`
- Primitive procedures
  - `+`, `-`, `*`, `/`, `abs` …
  - evaluate to the corresponding internal procedure
  - *e.g.* "`+`" evaluates to the procedure that adds numbers together
- Can write this as: `+` → `+`
  - or as: `+` → `[primitive procedure +]`
  - We'll write this out explicitly sometimes

Another example

- `(+ 3 (- 4 (* 5 (expt 2 2))))`
  - evaluate `3` → `3`
  - evaluate `(- 4 (* 5 (expt 2 2)))`
    - evaluate `4` → `4`
    - evaluate `(* 5 (expt 2 2))`
      - evaluate `5` → `5`
      - evaluate `(expt 2 2)`
        - evaluate `2` → `2`, evaluate `2` → `2`, `expt` → `expt`
        - apply `expt` to `2`, `2` → `4`
          (`expt` means "to the power of")
      - evaluate `*` → `*`
      - apply `*` to `5`, `4` → `20`
    - evaluate `-` → `-`
    - apply `-` to `4`, `20` → `-16`
  - evaluate `+` → `+`
  - apply `+` to `3`, `-16` → `-13`

Evaluation with variables

- An assignment creates an association between a symbol (variable) and its value

```
(define x 3)
```

  - Here `x` is the variable, `3` is the value
- To evaluate a variable:
  - find the value associated with the variable
  - and replace the variable with its value

Variable evaluation example

- `(define x 3)`
- Then evaluate `x`
  - look up value of `x`
  - `x` has value `3` (due to `define`)
  - result: `3`
- A bit vague right now; later we will be more explicit about how this works

Simple expression evaluation

- Assignment and evaluation

```
(define x 3)
(define y 4)
```

  - evaluate `(+ x y)`
    - evaluate `x` → `3`
    - evaluate `y` → `4`
    - evaluate `+` → `[primitive procedure +]`
    - apply `+` to `3`, `4` → `7`

Special forms

- ***N.B.*** There are a few special forms which do ***not*** evaluate in the way we've described.
- **`define`** is one of them

```
(define x 3)
```

  - We do **not** evaluate **`x`** before applying **`define`** to **`x`** and **`3`**
  - Instead, we
    - evaluate the *second* operand (**`3`** → **`3`**)
    - make an association between it and the first operand (the unevaluated symbol **`x`**)

Special forms

- Another example: `(define x (+ 2 3))`
- Evaluate `(+ 2 3)`
  - evaluate `2` → `2`
  - evaluate `3` → `3`
  - evaluate `+` → `[primitive procedure +]`
  - apply `+` to `2`, `3` → `5`
- Make association between `x` and `5`
  - details of how association is made aren't important (for now)

Special forms vs. syntax

- *All* programming languages have special forms to do fundamental language tasks
- *Most* languages have special syntax for each special form (*e.g.* `def`, `if`, `for` in Python)
- Scheme uses same basic syntax for everything
  - but some forms like `define` are "special" because they have their own special evaluation rules

`lambda`

- **`lambda`** is also a special form:
  - result of a **`lambda`** expression is always a function
    - also known as a "procedure"
  - We do ***not*** evaluate its contents
    - ***none*** of the operands get evaluated
    - just “save them for later”

Scheme function definition

- Translate: a(r)= pi * r^2

which performs the operation

- `(define a (lambda (r) (* pi (expt r 2))))`

define **a** to be a function

of one variable, r

Evaluating lambda

```
(define a (lambda (r) (* pi (expt r 2))))
```

- Evaluate: `(lambda (r) (* pi (expt r 2)))`
  - create procedure with one *argument* `r`
    and *body* `(* pi (expt r 2))`
  - details aren't important (for now ☺)
  - can write as
    `(lambda (r) (* pi (expt r 2)))` →
    `(lambda (r) (* pi (expt r 2)))`
  - (or just: `"itself"`)
- Make association between `a` and new procedure
  - again, details aren't important, but we will revisit this later

Evaluating a function call

The basic rule again:

1. Evaluate the operands (arguments)
2. Evaluate the operator (function)
3. Apply the operator (function) to the (evaluated) operands (arguments)

To apply a function to its arguments:

1. Substitute the function argument *variables* with the *values* given in the call everywhere they occur in the function *body* (not the entire function)
2. Evaluate the resulting expression

Example 1

```
(define f
  (lambda (x) (+ x 1)))
```

- Evaluate the **`define`**, then...
- Evaluate `(f 2)`
  - evaluate `2` → `2`
  - evaluate `f` → `(lambda (x) (+ x 1))`
  - apply `(lambda (x) (+ x 1))` to `2`
    - substitute `2` for `x` in the expression `(+ x 1)` → `(+ 2 1)`
    - evaluate `(+ 2 1)` → (skip obvious steps) → `3`

Example 2

- `(define f`
  `(lambda (x y)`
  `(+ (* 3 x) (* -4 y) 2)))`
- Evaluate `(f 3 2)`
  - evaluate `3` → `3`
  - evaluate `2` → `2`
  - evaluate `f` → `(lambda (x y)`
    `(+ (* 3 x) (* -4 y) 2))`
  - apply `(lambda (x y) ...)` to `3`, `2`
    - substitute `3` for `x`, `2` for `y` in body
      → `(+ (* 3 3) (* -4 2) 2)`
      - evaluate → [skip obvious steps] → `3`

Syntactic sugar

- We can write:

```
(define f (lambda (x) (+ x 1))
```

- as:

```
(define (f x) (+ x 1))
```

- Simply an alternate syntax
  - allows us not to write `lambda` everywhere
  - feels more natural (to me, anyway)
  - means the **exact same** thing

Quote:

"Syntactic sugar causes cancer of the semicolon."

– Alan Perlis

Evaluating `define` with functions

To evaluate: `(define (f x) (+ x 1))`

1. "Desugar" it into `lambda` form:
   - `(define f (lambda (x) (+ x 1)))`
2. Now evaluate like any `define`:
   - create the function `(lambda (x) (+ x 1))`
   - create an association between the name `f` and the function

A longer example

- `(define sq (lambda (x) (* x x)))`
- `(define d`
  `(lambda (x y) (+ (sq x) (sq y))))`
- Evaluate: `(d 3 4)`
  - evaluate `3` → `3`
  - evaluate `4` → `4`
  - evaluate `d` →
    `(lambda (x y) (+ (sq x) (sq y)))`
  - apply `(lambda (x y) ...)` to `3`, `4`

Example cont'd

- Apply `(lambda (x y) (+ (sq x) (sq y)))` to `3`, `4`
  - Substitute `3` for `x`, `4` for `y` in `(+ (sq x) (sq y))`
  - Evaluate `(+ (sq 3) (sq 4))`
    - evaluate `(sq 3)`
      - evaluate `3` → `3`
      - evaluate `sq` → `(lambda (x) (* x x))`
      - apply `(lambda (x) (* x x))` to `3`
        - substitute 3 for `x` in `(* x x)`
        - evaluate `(* 3 3)`
        - evaluate `3` → `3`, `3` → `3`, `*` → `*`
        - apply `*` to `3`, `3` → `9`

Example cont'd (2)

- Apply `(lambda (x y) (+ (sq x) (sq y)))` to `3`, `4`
  - Substitute `3` for `x`, `4` for `y` in `(+ (sq x) (sq y))`
  - Evaluate `(+ (sq 3) (sq 4))`
    - evaluate `(sq 3)` → [many steps, previous slide] → `9`
    - evaluate `(sq 4)`
      - evaluate `4` → `4`
      - evaluate `sq` → `(lambda (x) (* x x))`
      - apply `(lambda (x) (* x x))` to `4`
        - substitute `4` for `x` in `(* x x)`
        - evaluate `(* 4 4)`
        - evaluate `4` → `4`, `4` → `4`, `*` → `*`
        - apply `*` to `4`, `4` → `16`

Example cont'd (3)

- Apply `(lambda (x y) (+ (sq x) (sq y)))` to `3`, `4`
  - Substitute `3` for `x`, `4` for `y` in `(+ (sq x) (sq y))`
  - Evaluate `(+ (sq 3) (sq 4))`
    - evaluate `(sq 3)` → [many steps, 2 slides back] → `9`
    - evaluate `(sq 4)` → [many steps, previous slide] → `16`
    - evaluate `+` → `[primitive procedure +]`
    - apply `+` to `9` and `16` → `25`
- which is the final result
- `(d 3 4)` → `25`

Substitution Model

- Gives a *precise* model for evaluation
- Can be carried out mechanically
  - by you or by a computer
- We will expand and revise this model later

Video clip

- Introducing recursion…

So far

- You now know Scheme syntax
- You can write any procedure that deals with a fixed amount of data

Question:

- How do we deal with computations that require *unbounded* size / number of operations?

…e.g. “sum the numbers from 1 to n”?

sum 1-to-*n*

- Easy when *n* constant:
  - *e.g.* sum numbers from 1 to 10

```
(define sum-1-to-10
    (+ 1 2 3 4 5 6 7 8 9 10))
```

- but want to do this for <u>any</u> value of *n*
  - *i.e.* no upper bound on size of *n*
  - same procedure must work for any positive *n*

Note

- To deal with this, we'll first have to deal with some other issues:
  - *Types*
  - *Booleans*
  - *Conditionals*
- Preview: the solution will involve *recursion*

Types

- Last time, all of our expressions evaluated to either
  - *numbers,* or
  - *procedures (functions)*
- Convenient to have other *kinds* of things
- **Type:** term we use to classify different kinds of values

Types we'll use

- Numbers:
  - ... `-1`, `0`, `1`, `2`, `3`, `4`, `5`, ... `42`, ... `65536`, ...
  - … also rationals: `3/2`, `4/5`, `7/16`
  - ... and real numbers: `1.50`, `3.1415`, `6.022e+23`
- Procedures/functions:
  - `(lambda (x) (+ x 1))`
- Booleans: (new)
  - `#t` means "true"
  - `#f` means "false"
- … and others we'll encounter along our way

Functions evaluating to booleans

- We've seen functions that evaluate to numbers:

```
(define (circle-area r)
  (* pi (expt r 2)))
(circle-area 1) ;; evaluates to 3.1415...
```

- Some functions evaluate to booleans:

```
(> 3 4)   ;; evaluates to #f
(< 3 4)   ;; evaluates to #t
(= 2 3)   ;; etc.
```

  - still prefix notation!

Functions with boolean operands

- We've already seen functions that expect numeric operands:

```
(define (circle-area r)
    (* pi (expt r 2)))
```

- There are also functions that expect boolean operands:
  - built-in ones:

```
(and #t #f)  ;; evaluates to #f
(or #t #f)   ;; evaluates to #t
(not #t)     ;; evaluates to #f
```

  - or your own…

n.b. **and** and **or** are really special forms

`and` and `or`

- **`and`** and **`or`** evaluate only as many operands as needed

  `(and #f <anything>)` → **`#f`**

  `(or #t <anything>)` → **`#t`**

- If 1st operand of **`and`** is **`#f`**, don't evaluate 2nd
- If 1st operand of **`or`** is **`#t`**, don't evaluate 2nd
- **`and`** and **`or`** are thus special forms, not functions
  - don't use standard evaluation rule
  - could be functions, but would be less efficient
- **`and`**/**`or`** can also have more than 2 operands

  `(and #t #t #t #f)` → **`#f`**

A new function on booleans

```
(define (at-least-two-true? a b c)
  (or (and a b)
      (and b c)
      (and a c)))
```

- Convention: functions returning booleans have names ending in `?`
- Often referred to as “predicates”
  - (`and`, `or`, `not` don’t follow this convention)

Using booleans

- Booleans are not much good by themselves
- But we can use them (or functions returning them) to make decisions
- Which leads us to...

The `if` conditional expression

- A new *special form*
- Idea: based on the value of one expression, evaluate one of two other expressions
- Form:

```
(if <boolean-expression>
    <true-case-expression>
    <false-case-expression>)
```

How does `if` evaluate?

```
(if <boolean-expression>
    <true-case-expression>
    <false-case-expression>)
```

1. Evaluate the <*boolean-expression*>.
2. Does it evaluate to `#t`? If so, evaluate the ***`<true-case-expression>`***.
3. Does it evaluate to `#f`? If so, evaluate the ***`<false-case-expression>`***.
4. The whole `if`-expression then evaluates to the result of either 2 or 3.

   Critical: Only **ONE** of operands 2 and 3 is evaluated.

Examples

```
(define (fn1 a b) (if (> a b) a b)) ;; does what?
(fn1 5 3) ;; evaluates to?
```

→ 5

```
(define (fn2 a) (if (< a 0) (- a) a)) ;; does what?
(fn2 50)  ;; evaluates to?
```

→ 50

```
(fn2 -23) ;; evaluates to?
```

→ 23

```
(fn2 #t)  ;; evaluates to?
expects argument of type <real number>;
  given #t
```

Type errors

- This is a type error
- Program expected one type of data, given another (here, boolean instead of number)
- Many computer languages check types in advance before running any code
- Scheme *doesn't* do this
  - more flexible, but
  - can get type errors at run time

Substitution model with `if`

- Unchanged!
- Previous substitution model works exactly the same as before
  - as long as we know how to evaluate `if` expression itself (special form)

Substitution model with `if`

- `(define (max a b) (if (> a b) a b))`
- Evaluate `(max 3 2)`
  - evaluate `3`… evaluate `2`... yadda yadda...
  - evaluate `max` → `(lambda (a b) (if (> a b) a b))`
  - apply `(lambda (a b) (if (> a b) a b))` to `3`, `2`
    - substitute `3` for `a`, `2` for `b` in `(if (> a b) a b)`
    - evaluate `(if (> 3 2) 3 2)` [special form]
      - evaluate `(> 3 2)` … `#t`
      - `(if #t 3 2)` ... evaluate true case ...
      - evaluate `3` → `3`
- result: `3`

`if` is an expression

- ***N.B.*** **`if`**, like everything else we've seen (except for **`define`**), is an expression, so…
- It "returns a value"
  - or, "evaluates to a value"

    (this is different from many programming languages)
- You can use that value:
- *e.g.*

```
(/ (if (> a b) a b)
   (if (> a b) b a))
```

`if` as expression: `scale`

```
(define (scale a b)
  (/ (if (> a b) a b) (if (> a b) b a)))
```

- Evaluate: **`(scale 4 2)`**
  - eval `4` → `4`, `2` → `2`
  - eval `scale` →

```
(lambda (a b)
  (/ (if (> a b) a b) (if (> a b) b a)))
```

`if` as expression: `scale`

- Apply
```
(lambda (a b)
        (/ (if (> a b) a b)
           (if (> a b) b a)))
```
to `4`, `2`
  - Substitute `4` for `a`, `2` for `b`

```
(/ (if (> 4 2) 4 2)
   (if (> 4 2) 2 4))
→ (/ (if  #t 4 2) (if #t 2 4))
→ (/ 4 2)
→ 2
```

Next time

- Recursion!
  - see "recursion"