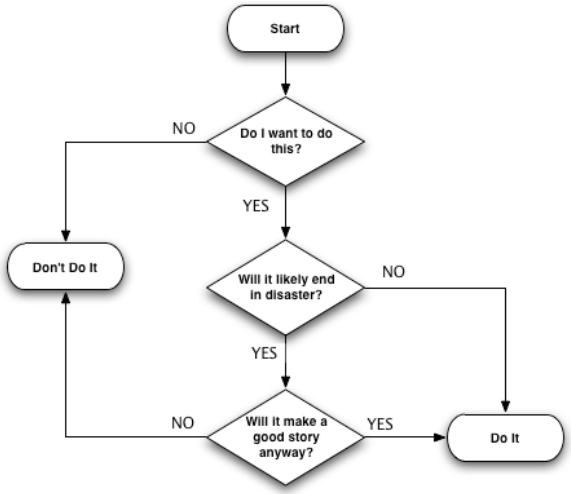
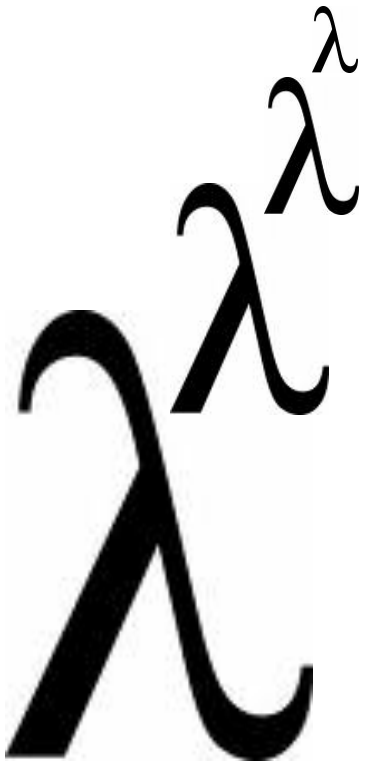
CS 4
Fundamentals of Computer Programming

Lecture 1: January 5, 2015

Introduction





Today

- Administrative details
- What class is about (course philosophy)
- Review mathematical notation
  - translate to Scheme programming language

Administrative details

People

- Lectures: Mike Vanier (mvanier@cms)
- TAs:
  - Dzhelil Rufat (drufat@caltech)
  - Chung Eun Kim (ckkim@caltech)
  - Aman Agarwal (aaagarwa@caltech)
  - More TBA

Home page

- The CS 4 home page is on the Caltech Moodle site: http://moodle.caltech.edu
- You can enroll in the site using the "enrolment key", which is typedracket
- All lectures, assignments, exams, course policies, grading schemes etc. will be posted there

Prerequisites

- A passing grade in CS 1
- This is not a course for absolute beginners!
  - too fast, too general, too abstract
- It's a great *second* course in programming
  - can be taken concurrently with CS 2
- Also need to know basic Unix/Linux
  - at the level used in CS 1

Course organization

- Lectures: (2 or 3) x 50 min per week
- Lab sections in the CMS computer lab
  - at least two per week
  - come to either or both (attendance is optional)
  - probably in the evening

Grading system

- *Assignments* (AKA "labs"): 7 labs, worth 3 points each, for a total of 21 points
  - usually out on Wednesday, due one week later
- *Midterm*: 6 points
- *Final*: 18 points
- Maximum grade: 45 points
- Passing grade: 28 points
- Course is PASS/FAIL only!

Lab grading

- Each lab is given an integer grade between 0 and 3:
- 0 means the lab is completely inadequate
- 1 means the lab has some serious flaws in one or more sections
- 2 means the lab is acceptable
- 3 means the lab is extremely good, with no significant flaws

Lab grading

- Each lab will have multiple sections (usually 2 or 3)
- Each section will get an integer grade from 0 to 3 as described previously
- The overall grade of the lab is the *minimum* of the section scores
- This might appear to be excessively harsh, if it weren't for:

Rework

- Each lab can be reworked for one full week, starting from the time the original version is graded and returned
- This is a good way of bringing lab grades up
  - often one section is a 1 while other sections are 3s, so rework can bring entire lab up to a 3 grade

The computer lab

- The computer lab is located in Annenberg room 104
- The lab contains a large number of computers, running the Linux operating system
  - these computers are called "the CMS cluster"
- You are encouraged to work on your labs there, especially during lab sections
- It's also OK to work on the labs on your own computer, in the lab or elsewhere

The computer lab

- To use the computers in the computer lab, you will need to apply for a CMS cluster account
  - details are on the CS 4 web site
- It usually takes 2-3 days to get an account, so don't do it the day before a lab is due
- Your password will be mailed to you, so don't delete the email or forget it
  - you'll change it after logging in for the first time
- May need to re-activate existing accounts

csman

- Grades will be submitted and graded using the csman grading program, created by Donnie Pinkston
- Lab source code files will be uploaded to the csman web site: http://csman.cms.caltech.edu
- Grades will be available from the same site

csman

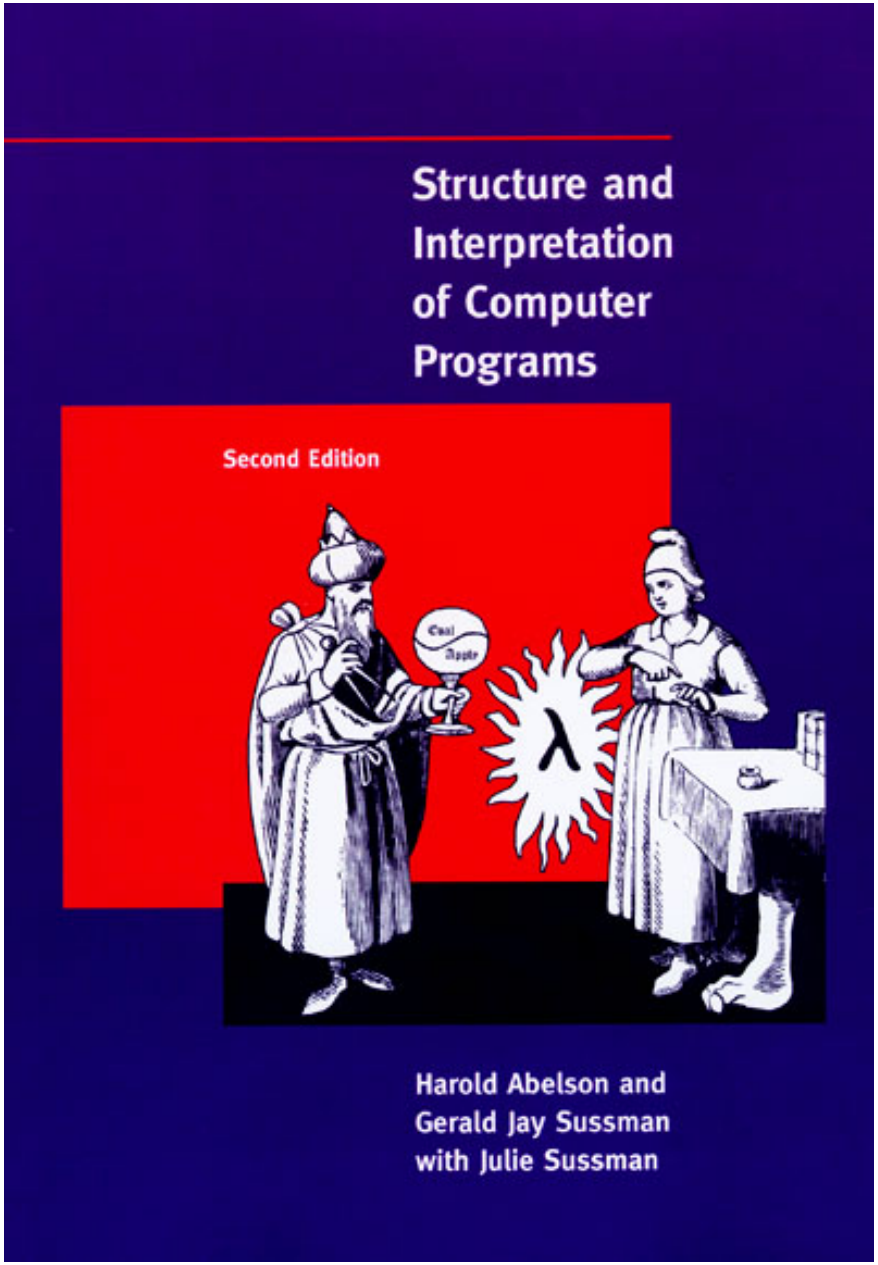
- You will need to have a csman account in order to submit assignments
- csman uses the same username/password as the CMS cluster itself, so you can log in using that password
  - However, you still need to be assigned to the CMS 4 csman page in order to submit homework
- csman "signup sheet" will be posted to CS 4 moodle web site (like we did for CS 1)

Time on assignments

- Taking excessive time on a problem is a **symptom** that you missed an important concept
  - It's possible to do things the **wrong** way (missing what we're teaching you)
    - …but it probably will take you a lot more time
- Time hint on assignments
  - suggested upper bound
  - feedback to you → you should see a TA
  - Don't spend hours alone on a problem!

Textbook

- This course uses a non-optional textbook: Structure and Interpretation of Computer Programs (*SICP*), by Hal Abelson and Gerry Sussman
- Available for free online at http://mitpress.mit.edu/sicp
- Also available at MIT press web site or at amazon.com
  - see links on CS 4 home page


Structure and
Interpretation
of Computer
Programs
Second Edition
λ
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

Your action items

- Log in to CS 4 home page and register there
  - read administrative items
- Check out the textbook online, and order a paper copy if you want
  - read through section 1.1.5
- Apply for a CMS cluster account
  - details on the web site

Course overview and philosophy

What this course is not

- This course is not a "practical" course in computer programming
  - We expect you to have some practical experience already
  - Take CS 2, CS 3, CS 11 for more practical stuff
  - The knowledge you will get from CS 4 *will* be practical, but not as immediately as a course in how-to-draw-pictures-in-language-X might be
- We won't try to give you cookbook recipes to follow in order to accomplish some computational task you're interested in

What this course is

- This course is a *conceptual* course in programming
  - *i.e.* understanding the "how" and "why" of programming, not just the "what"
- We will stress the *big ideas* in programming and programming languages
  - abstraction, time/space complexity, types, design
- We will discuss different *paradigms* of programming
  - functional, imperative, object-oriented

What this course is

- We will emphasize the notion of a programming *model* in order to give us a rigorous understanding of what happens when a program executes
- We will refine this model as the course progresses so that we can understand more and more complex kinds of code

What this course is

- We will show you ways of programming you probably didn't realize were possible
  - higher-order functions
  - generic functions
  - the Y combinator
  - structural macros
  - continuations and continuation-passing style
- Some of this material is fairly advanced, brain-melting stuff, but it's fun and very interesting (and increasingly useful)

What this course is

- We will start with a very small number of language features and build up from there
- We will implement some language features ourselves (*e.g.* object-oriented programming) which will allow us to understand the design space of those features

What this course is

- Once you've finished this course, you should be able to pick up new languages very easily
  - because you will understand how to relate the language's features to what the space of possibilities is
  - You will be able to distinguish surface differences in languages vs. more fundamental ones

Quote (about the textbook)

- Peter Norvig (director of research at Google) said this about SICP:

To use an analogy, if SICP were about automobiles, it would be for the person who wants to know how cars work, how they are built, and how one might design fuel-efficient, safe, reliable vehicles for the 21st century. The people who hate SICP are the ones who just want to know how to drive their car on the highway, just like everyone else.

Quote

- Peter Norvig (director of research at Google) said this about SICP:

Those who hate SICP think it doesn't deliver enough tips and tricks for the amount of time it takes to read. But if you're like me, you're not looking for one more trick, rather you're looking for a way of synthesizing what you already know, and building a rich framework onto which you can add new learning over a career. That's what SICP has done for me. I read a draft version of the book around 1982 and it changed the way I think about my profession. If you're a thoughtful computer scientist (or want to be one), it will change your life too.

Languages we'll use

- We'll learn three languages in CS 4:
  - Scheme (Racket dialect)
  - Typed Racket
  - Ocaml (Objective CAML)
- We will not cover all aspects of these languages by a long shot!
- We will use them to illustrate key concepts in programming languages
  - not because they're particularly "practical languages" (though they can be *e.g.* Ocaml is used a lot in the financial sector)

Languages we'll use

- SICP uses a very small subset of Scheme, so we'll start with that
- We will also cover some cool features of Scheme not discussed in SICP
- We will transition to Ocaml at the end of the course, to talk more about types and static typing

Scheme / Racket

- The "dialect" of Scheme we will be using is called *Racket*
- Technically this is a new language, independent of Scheme
- But actually, Racket is mostly a strict superset of Scheme with tons of additional features, so it's fine for us to use here
- I'll point out the areas where Racket differs significantly from Scheme when we encounter them

Typed Racket FTW

- The Racket language has a very cool feature: multiple Scheme dialects can be used within one language environment
- The most interesting of these (to me) is *Typed Racket*, which extends Scheme with facilities for compile-time type checking
- We will go back and forth between (untyped) Racket and Typed Racket as we go along
- Typed Racket shines for larger programs!

Another reason

- There's another reason we will learn Scheme/Racket (and Ocaml):

> "Because it lets us do things we can't do in other languages."
>
> -- Paul Graham

Example 1

- Sum all the numbers from 1 to n
  - for some number n you specify
  - easy to do in Scheme or C, or Java, or Python
  - or any other programming language

Example 2

- Write a function that takes a number n and returns a *function* which adds n to its argument
- In Scheme:

```
(define (addn n) (lambda (x) (+ x n)))
```

- In Ocaml:

```
let addn n = fun x -> x + n
```

- In C or Java:
  - "You just can't write it."

The point

- "Programming languages teach you not to want what they don't provide."
- We want you to know what features programming languages *can* provide
  - even if the language you're using doesn't provide them
- Want you to be educated consumers of languages
  - or maybe even producers...

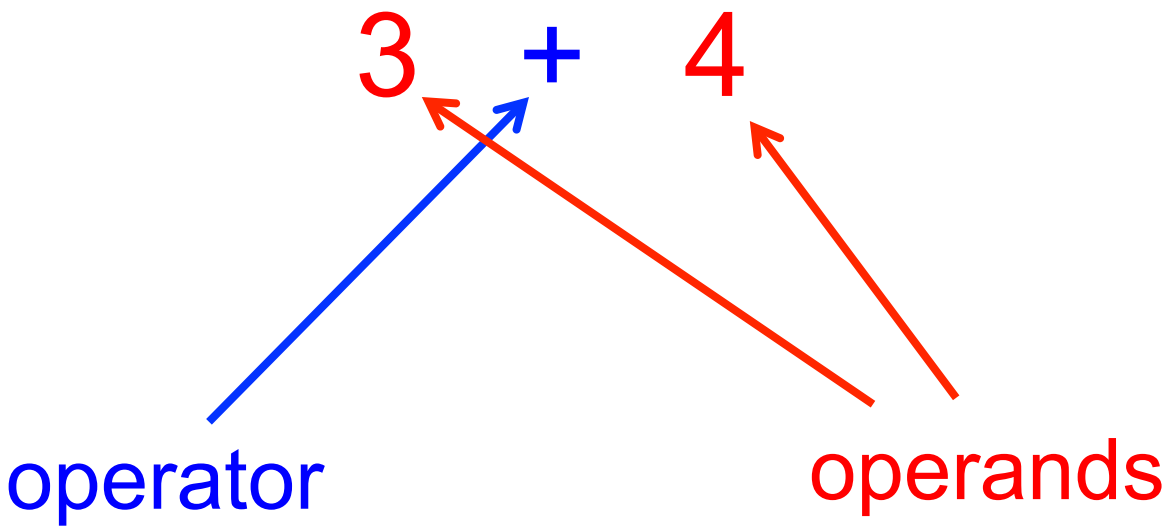
Hidden (?) agenda

- Scheme and Ocaml both support a very interesting kind of programming called functional programming
  - Example 2 required this
- This course will introduce you to functional programming, along with other ways of programming
- It will also give you the background you need to start learning more advanced functional languages like Haskell

Scheme in 20 minutes
(start of course material)

Infix notation

- Infix: *operators* come between *operands*

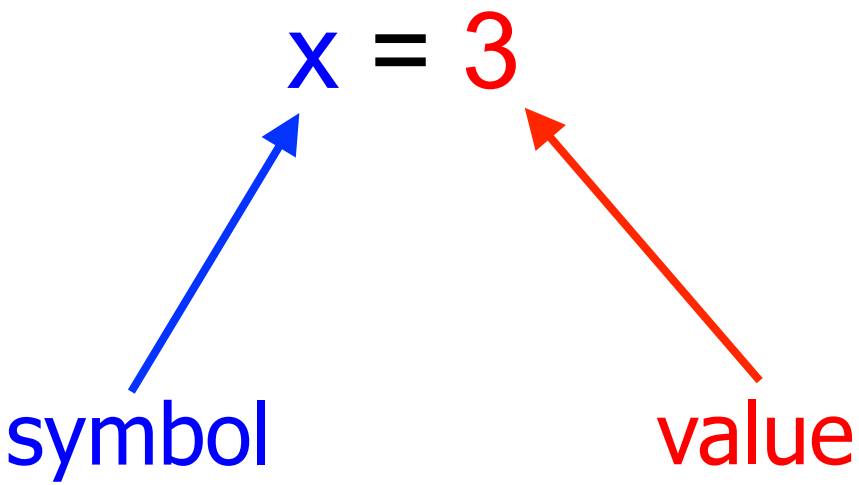


Infix notation

- Examples:
  - 3 + 4
  - 3 + 4 * 5
  - 3 + 4 – 5 * 2^2
- We sometimes use parentheses to specify order of evaluation
  - (3 + 4) * 5
  - (3 + ((4 - 5) * 2))^2

Assignment

- Assignment gives a symbol a particular value



Assignment

- Examples:

  y = 4

  z = x + y

  w = z + 2

Evaluation

- x = 3
- y = 4
- z = x + y
  - How do we evaluate this to get z?
  - We *substitute* a symbol's value for the symbol in the right-hand side of the equation

  → z = 3 + 4 = 7
- w = z + 2
  - w = 7 + 2 = 9

Functions

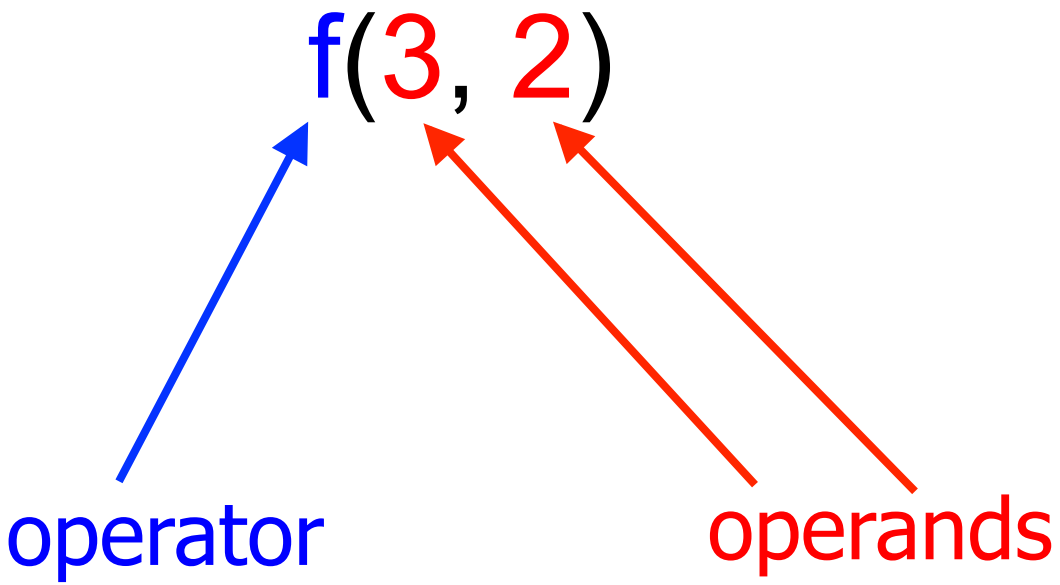
- Function definition:
  - $f(x) = x^2 - 2*x + 4$
  - $f(x,y) = 3*x - 4*y + 2$
  - $a(r) = pi * r^2$

Function evaluation

- *Substitute* numbers for arguments, simplify
- $f(x) = x^2 - 2*x + 4$
  - $f(3) = 3^2 - 2*3 + 4 = 9 - 6 + 4 = 7$
- $f(x,y) = 3*x - 4*y + 2$
  - $f(3,2) = 3*3 - 4*2 + 2 = 3$
- $a(r) = pi * r^2$
  - $a(1) = pi * 1^2 = 3.1415926\ldots$

Prefix notation

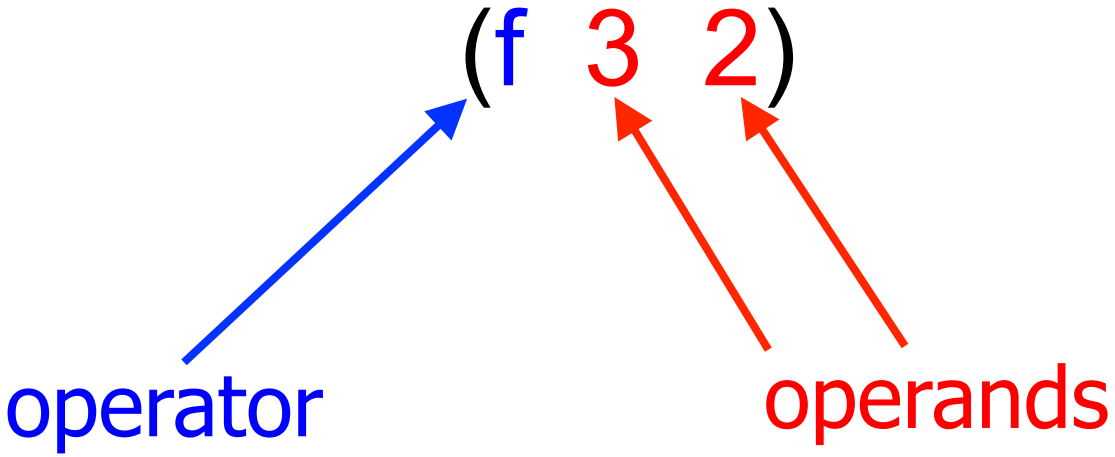
- Prefix: operator precedes the operands
- Familiar from function usage



Prefix notation

- Examples:
  - f(x) operator: f operand: x
  - f(3) operator: f operand: 3
  - f(x,y) operator: f operands: x, y
  - f(3,2) etc.
  - a(r)
  - a(1)

Alternative prefix notation



- means the same thing
- parentheses on the outside
- separate with spaces, not commas
  - space is significant!

Scheme

- abandons infix; only uses (2nd) prefix notation

math	Scheme
3 + 4	`(+ 3 4)`
3 + 4*5	`(+ 3 (* 4 5))`
3 + 4 - 5*2^2	`(+ 3 (- 4 (* 5 (expt 2 2))))`
f(3)	`(f 3)`
f(x,y)	`(f x y)`
f(3,2)	`(f 3 2)`

Yes, it's weird

- This notation takes getting used to
- Lots of parentheses
  - DrRacket editor will help you with these
- but completely consistent
  - no special rules to remember
  - *all* operators are prefix
  - no "operator precedence"
- Usually takes 2-3 weeks before comfortable

Scheme combinations

- Scheme "combinations" look like this:

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

- Delimited by parentheses
- First element is the **operator**
- Rest are **operands**
- In general, there can be an arbitrary number of operands
  - but will depend on operator

Scheme combinations

- Examples:

```
(+ 2 3)
(+ 2 3 4)
(abs -7)
```

- Combinations may be nested:

```
(abs (+ (- x1 x2) (- y1 y2)))
```

Parentheses

- ***NOTE:*** Parentheses are *not* optional
  - Everything has parentheses
  - They have a very specific meaning
    - first item in parenthesized set is the operator….
    - rest are operands
  - *Can't* just wrap more parentheses around things
    - changes the meaning!
- This is *different* from
  - mathematical use
  - use in most other programming languages

Parentheses

- Scheme is a dialect of the programming language “Lisp”
- Lisp means either
  - LISt Processing
  - Lots of Irritating Stupid Parentheses

…depending on who you ask

Parentheses

- Scheme even uses parentheses for function definitions, control structures (conditionals, loops), etc.
- This makes the syntax extremely simple and regular
  - less to remember!

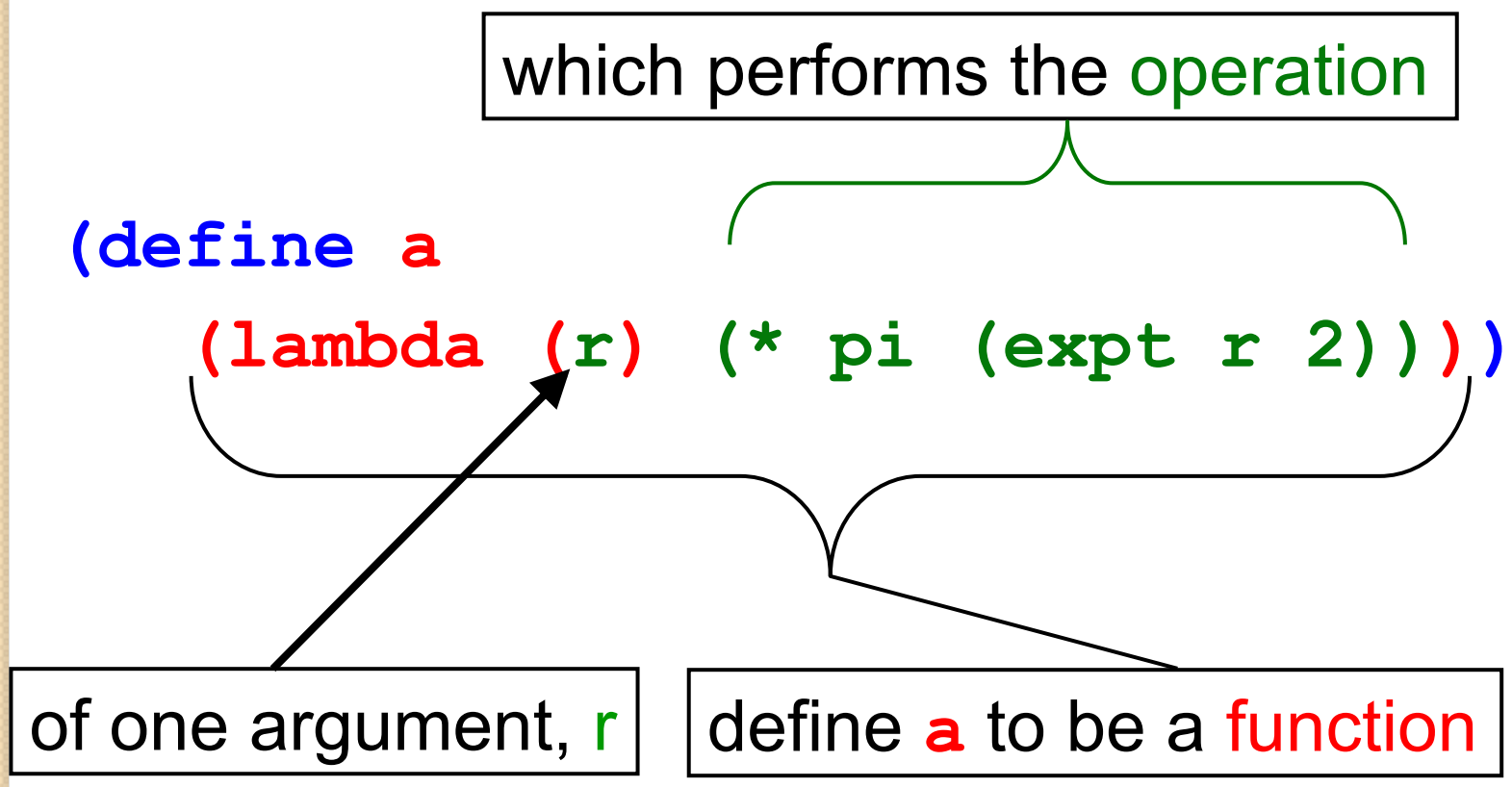
Assignment in Scheme

- Assignment is also a prefix operation

math	Scheme
x = 3	`(define x 3)`
y = 4	`(define y 4)`
z = x + y	`(define z (+ x y))`
w = z + 2	`(define w (+ z 2))`

Scheme function definition

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



lambda???

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

- **lambda** means “this combination represents a procedure (function)”
- from the Greek letter lambda (λ)
  - we'll see why later
- Somewhat cumbersome; we’ll see a shortcut next lecture

More function definitions

- f(x) = x^2 - 2*x + 4
  - ```
    (define f
      (lambda (x)
        (+ (expt x 2) (* -2 x) 4)))
    ```
- f(x,y) = 3*x - 4*y + 2
  - ```
    (define f
      (lambda (x y)
        (+ (* 3 x) (* -4 y) 2)))
    ```
- *N.B.* some operators (like **+**) can have a variable number of operands

Comments

- Often useful to leave notes for
  - your future self
  - others who may read/modify your code
- Can do this by using *comments*
  - words and symbols you tell the machine to ignore
- In Scheme:
  - comments start with a semicolon (usually two) and go to the end of the line

```
;; This is a comment.
```

Scheme as a calculator

- Now you know enough to use a Scheme interpreter as a (programmable) calculator
  - Assignment 1 will walk you through how to invoke the Scheme interpreter (DrRacket)
- More importantly:
  - Now you know (almost) all the syntax in Scheme!
  - So we won't have to spend much time on syntax from now on

Next time

- The substitution model of evaluation