What does it take to write a program?

- Find out (or get told) what the program should do
- You have a working executable

Vocabulary for PLs

Excerpt from the R6RS standard

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple **semantics** and few different ways to form expressions. A wide variety of **programming paradigms**, including **functional**, **imperative**, and message passing styles, find convenient expression in Scheme.

Reading this should give you a good overview of what Scheme is about. But first we have to learn what the terms mean!

(Scheme demo)

Programming Language Paradigms

Most popular PLs fall into at least one of six classes:

- Imperative/procedural
  C, Fortran, Cobol
- Functional
  Lisp, Scheme, ML, Haskell
- Object-oriented
  C++, Java, Smalltalk
- Scripting
  Perl, PHP, Javascript
- Logic Programming (Prolog *et al*)
- Esoteric Languages (brainfuck, INTERCAL, befunge, Chef)

Imperative Programming Languages

Consider the following code fragment from C++:

```
int x = 0;
x = 3;
x = x + 1;
```

- Each statement is a command.
- Code specifies actions and a specific ordering.
- Expressions may produce values (these do),
  but *side effects* are often more important.

Functional Programming

Functional programming is *declarative*: the output is a mathematical function of the input.
Emphasizes describing *what* is computed rather than *how*.

Key features:

- **Referential transparency**
  The value of an expression does not depend on its context.
- **Functions are first-class**
  Functions can be passed as arguments, created on-the-fly, and returned from other functions. Functions are data!
- **Types are first-class**
  This is not true in Scheme (there are no types), but is in other functional PLs.

Other common properties of functional PLs

- Garbage collection
- Built-in list types and operators
- Interpreters rather than compilers
- Extensive polymporphism
  (again, not applicable to Scheme)

Skill outcomes of SI 413

There are other goals on the course policy, but here's what **you will be able to do** in a few months:

1. Choose a programming language well-suited for a particular task.
2. Learn a new programming language quickly and with relative ease.
3. Understand the inner workings of compilers and interpreters and become a better user of both.

Major Course Components

Labs: 12 of 'em, worth 30%

- Will be done **in pairs** (which must change)
- Due most Wednesday nights
- **Do not expect to complete during lab time!**

Homeworks: 12 of 'em, worth 10%

- Due most Monday mornings
- Collaborate! You will have to **take notes** and **read**!

Project: 17% (next page. . .)

Scheme Practicum: 8% (in lab Thursday, 14 September)

Midterm Exam: 10% (in class Friday, 27 October)

Final Exam: 25%

Course Project

The course project will involve you learning different programming languages (**in pairs**), writing some programs and becoming mini-experts on the language.

Part 0 (due Aug. 23): Choose partners & languages

Part 1 (20%; due Sept. 20): Very simple program

Part 2 (50%; due Nov. 1): More involved program

Part 3 (30%; last day of class): In-class presentations

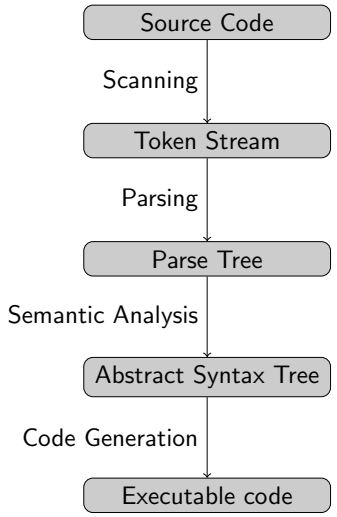
Phases of Programming

What does programming actually involve?

- Write a program
- Execute the program

Note: an **interpreter** essentially does compilation and execution simultaneously, on-the-fly.

Steps of Compilation



Example: English Language

Example: Python Code

```
x = int(input("Enter a number:"))
print("Three more is", x+3)
```


Compiled vs Interpreted

- Common compiled languages:
- Common interpreted languages:

In-between options

- Just-In-Time compilation
- Bytecode compilation

Unit Review

You should know:

- What this class is all about
- The major programming language paradigms
- Why we have developed so many different languages
- The basic steps of compilation
- The difference between language syntax and semantics