

## CS101: Programming Languages

- Course outline
  - Administration
  - Topics



---

---

---

---

---

---

---

---

## Administrivia

- Instructor: Jason Hickey, [jyh@cs.caltech.edu](mailto:jyh@cs.caltech.edu)
  - Office hours: MW 3-4pm
- Course times
  - MW, 2-3pm, 7AJRG



---

---

---

---

---

---

---

---

## Online info

- Course web site
  - <http://www.cs.caltech.edu/~jyh/cs101/>
- Mailing lists
  - General discussions
    - [cs101-class@metaprl.org](mailto:cs101-class@metaprl.org)
    - To subscribe, send mail to
      - [cs101-class-subscribe@metaprl.org](mailto:cs101-class-subscribe@metaprl.org)
  - Private correspondence
    - [cs101-admin@metaprl.org](mailto:cs101-admin@metaprl.org)



---

---

---

---

---

---

---

---

## Course info

- Prerequisites
  - CS20
- Grading
  - There are no exams
  - 9 Homeworks, weighted equally
  - Collaboration is allowed
- Course text
  - Pierce, "Types and Programming Languages", MIT Press, 2002



---

---

---

---

---

---

---

---

---

---

## Topics

- This is not a survey course on various programming languages
  - C, Java, Erlang, Forth, C++, OCaml, Python, sh, Perl, Javascript, Eiffel, Modula, Smalltalk, Scheme, Visual Basic, C#, FORTRAN, Ada, Pascal, sed, Oberon, Type theory, gas, Simula, Algol, Prolog, PCF, Excel, Unlambda, VHDL, Flavors, ...



---

---

---

---

---

---

---

---

---

---

## Programming language semantics

- Programming language *semantics* is the study of program *interpretations*
  - It is primarily concerned with building mathematical models, although translations between languages are also common
- Purpose
  - Understanding and reasoning about how programs behave
  - Help in designing programming languages
  - Design automation: automated tools usually use programming language models



---

---

---

---

---

---

---

---

---

---

## Rough outline

- Logic
- Semantics
  - Operational
  - Denotational
  - Axiomatic
- Type systems
  - System F
  - Dependent types
  - Type inference
  - Higher-order type systems



---

---

---

---

---

---

---

---

## Syntax

- A *grammar* gives a set of definitions for the well-formed syntax a programming language
- Usually expressed as a set of productions (like a context-free language)

$i ::= 0 \mid 1 \mid 2 \mid \dots$

$e ::= i \mid e + e \mid e - e \mid e * e \mid e / e \mid (e)$

$1 + (7/0) * (21 - 7)$



---

---

---

---

---

---

---

---

## Functions

- Most programming languages have functions, and function application

```
let rec fact i =  
  if i = 0 then  
    1  
  else  
    i * (fact (i - 1))
```

```
int fact(int i)  
{  
  int k = 1;  
  while(i)  
    k *= i--;  
  return k;  
}
```



---

---

---

---

---

---

---

---

## What is a function?

### Set theory

A function is a set of ordered pairs.

(0, 0)  
(1, 3)  
(2, 6)  
(3, 9)  
(4, 12)  
⋮

### Programming

A function is a *rule* for computing a value from an argument.

$\lambda i. i * 3$



---

---

---

---

---

---

---

---

---

---

## Functions

- If we view a function as a rule for computing a value, we have to describe how to compute the value
- Operational semantics
  - *The study of computation and evaluation*



---

---

---

---

---

---

---

---

---

---

## Lambda calculus

- A starting point for reasoning about functions
- The foundation of most functional programming languages, including the Lisp and ML languages (and the CS134 compilers)

$v$  ranges over a countable number of variables

$e ::= v$  (variables)  
|  $e_1 e_2$  (function application)  
|  $\lambda v. e$  (function abstraction)



---

---

---

---

---

---

---

---

---

---

## Lambda calculus

- The lambda calculus is Turing-complete
  - It can compute all computable functions (if we believe Church's thesis)
- We'll usually include numbers
- We'll ignore parentheses
  - Function application binds tighter than function abstraction

$\lambda f.f\ 1$   
 $(\lambda f.f)\ 1$   
 $(\lambda f.f\ 1)\ (\lambda x.x + 1)$   
 $(\lambda x.x\ x)\ (\lambda y.y\ y)$



---

---

---

---

---

---

---

---

## Evaluation

$\lambda f.f\ 1 \rightarrow \lambda f.f\ 1$   
 $(\lambda f.f)\ 1 \rightarrow 1$   
 $(\lambda f.f\ 1)\ (\lambda x.x + 1) \rightarrow (\lambda x.x + 1)\ 1$   
 $\rightarrow 1 + 1$   
 $\rightarrow 2$   
 $(\lambda x.x\ x)\ (\lambda y.y\ y) \rightarrow (\lambda y.y\ y)\ (\lambda y.y\ y)$   
 $\rightarrow (\lambda y.y\ y)\ (\lambda y.y\ y)$   
 $\rightarrow (\lambda y.y\ y)\ (\lambda y.y\ y)$   
 $\vdots$



---

---

---

---

---

---

---

---

## Reduction

- Program evaluation uses a substitution semantics.
- "To evaluate an application  $(\lambda v.e_1)\ e_2$ , substitute  $e_2$  for  $v$  in  $e_1$ ."
- Substitution:  $e_1[e_2/v]$  means:
  - Substitute  $e_2$  for  $v$  in  $e_1$
  - Replace all occurrences of  $v$  in  $e_1$  with  $e_2$
- Sometimes written  $e_1[v := e_2]$ .



---

---

---

---

---

---

---

---

## Single-step evaluation

- A single-step reduction is just a substitution
- Called “beta-reduction”

$$(\lambda v. e_1) e_2 \rightarrow_{\beta} e_1[e_2/v]$$



---

---

---

---

---

---

---

---

## Variable renaming

- It doesn't really matter what the parameters to a function are named
- This is usually ignored, but it can be really tricky

$$\begin{aligned} \lambda x. x + x &=_{\alpha} \lambda y. y + y \\ \lambda x. \lambda y. x &=_{\alpha} \lambda y. \lambda x. y \\ \lambda x. x + y &=_{\alpha} \lambda z. z + y \\ \lambda x. x + y &\neq_{\alpha} \lambda y. y + y \end{aligned}$$



---

---

---

---

---

---

---

---

## Defining substitution

- Define substitution  $e_1[e_2/v]$  by induction on the structure of  $e_1$ 
  - $e_1$  is a variable  $v'$  ( $v'$  may or may not be the same variable as  $v$ )
  - $e_1$  is a function application  $e_3 e_4$
  - $e_1$  is a function abstraction  $\lambda v'. e_3$  ( $v'$  may or may not be the same variable as  $v$ )



---

---

---

---

---

---

---

---

## Defining substitution

- $v'[e_2/v] = \begin{cases} e_2 & \text{if } v = v' \\ v' & \text{otherwise} \end{cases}$
- $(e_3 e_4)[e_2/v] = (e_3[e_2/v] e_4[e_2/v])$
- $(\lambda v'.e_3)[e_2/v] = \begin{cases} \lambda v'.e_3 & \text{if } v = v' \\ \lambda v'.e_3[e_2/v] & \text{otherwise (not really)} \end{cases}$



---

---

---

---

---

---

---

---

## Substitution and capture

- A variable is "captured" when it becomes bound during substitution
- Capture:  $(\lambda y.x)[y/x] = \lambda y.y$
- No capture:  $(\lambda z.x)[y/x] = \lambda z.y$



---

---

---

---

---

---

---

---

## Capture-avoiding substitution

- $v'[e_2/v] = \begin{cases} e_2 & \text{if } v = v' \\ v' & \text{otherwise} \end{cases}$
- $(e_3 e_4)[e_2/v] = (e_3[e_2/v] e_4[e_2/v])$
- $(\lambda v'.e_3)[e_2/v] = \begin{cases} \lambda v'.e_3 & \text{if } v = v' \\ \lambda v'.e_3[e_2/v] & \text{if } v' \notin FV(e_2) \\ \lambda v''.e_3[v''/v'] [e_2/v] & \text{otherwise, new } v'' \end{cases}$



---

---

---

---

---

---

---

---

## Variables

- An occurrence of a variable is a “binding occurrence” if it is defined by a lambda
- An occurrence is “bound” if it is in the scope of a binding occurrence with the same name
- Other occurrences are “free”



---

---

---

---

---

---

---

---

## Free variables

Free variables are defined by induction (as usual):

$$\begin{aligned}FV(v) &= \{v\} \\FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\FV(\lambda v.e) &= FV(e) - \{v\} \\FV(\lambda x.\lambda y.x) &= \{\} \\FV(\lambda x.x + y) &= \{y\} \\FV(\lambda x.(\lambda y.x) (\lambda z.y)) &= \{y\}\end{aligned}$$



---

---

---

---

---

---

---

---

## Some sensible properties

- Any two alpha-equivalent terms have the same free variables
- Free variables only decrease during evaluation
- Beta-reduction works:

If  $e_1 =_{\alpha} e_2$  and  $e_1 \rightarrow_{\beta} e_3$ , then  $e_2 \rightarrow_{\beta} e_4$  and  $e_3 =_{\alpha} e_4$ .



---

---

---

---

---

---

---

---