Lecture 15

CIS 341: COMPILERS

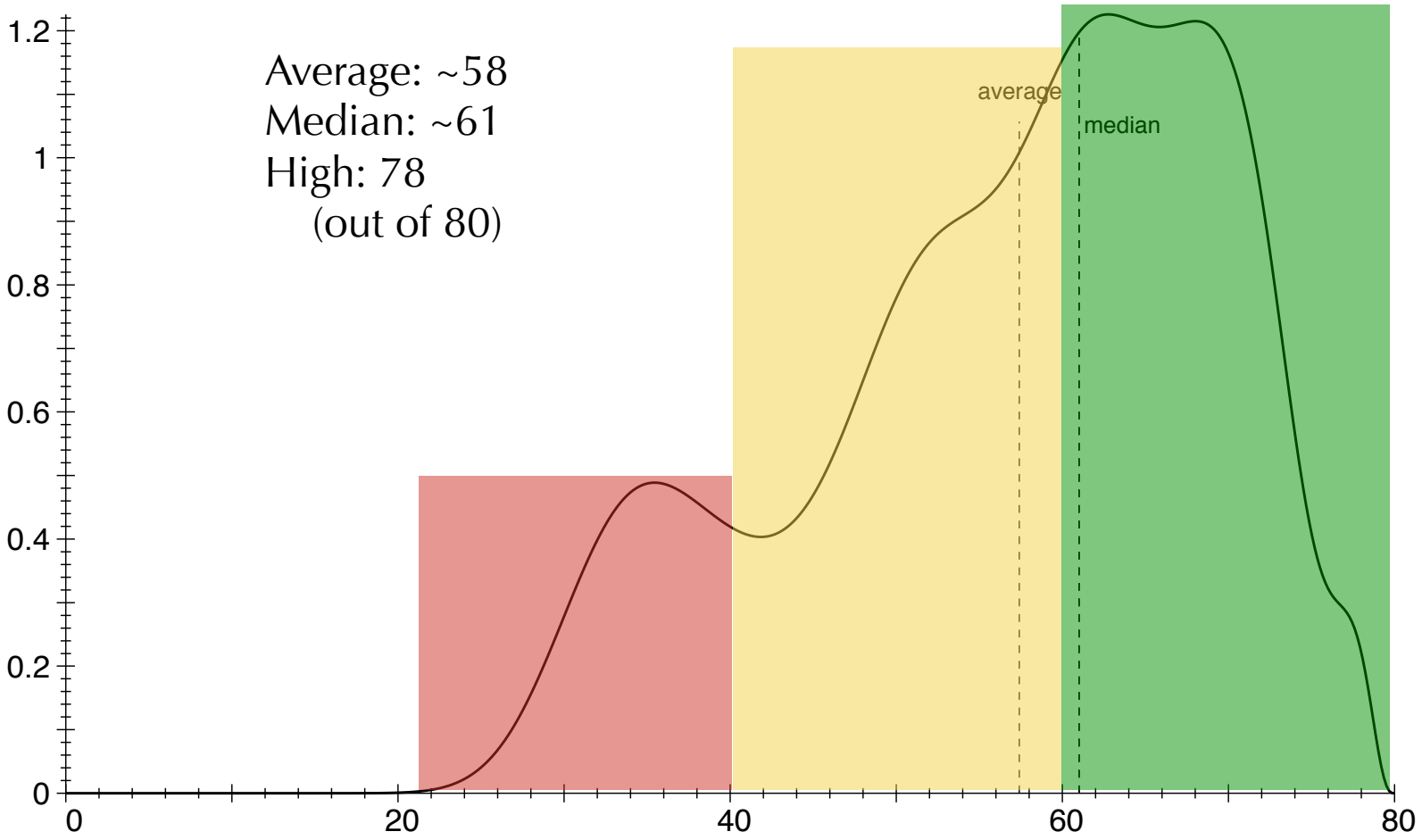
Announcements

- Reminder: HW4 Compiling OAT v.1
- DUE: Thursday, March 26th
- ***START TODAY! (IF YOU HAVEN'T ALREADY)***

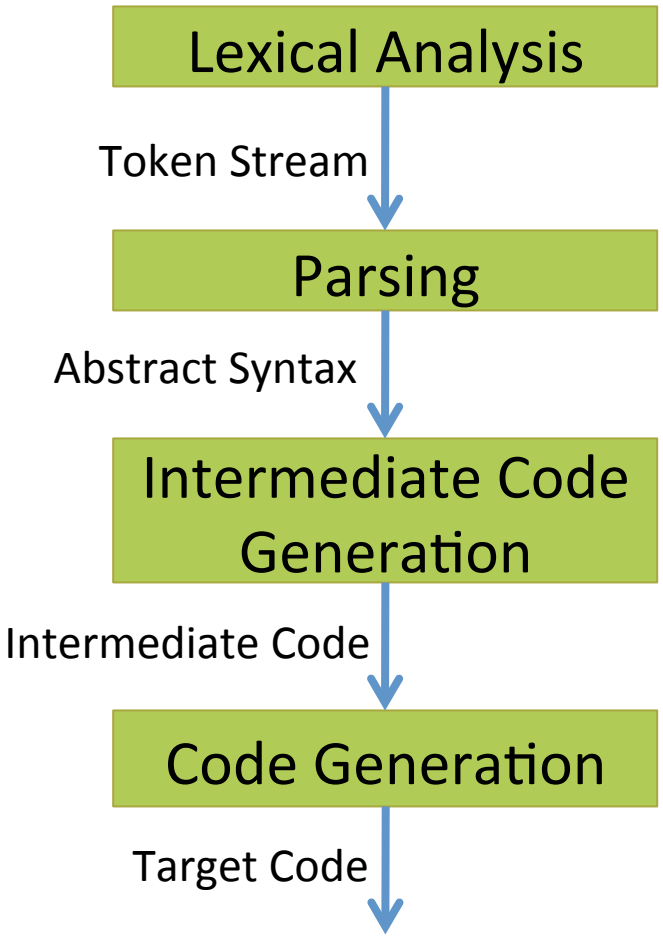
- Midterm has been graded

- My office hours today: 4:00 – 5:30 instead of 3:30 – 5:00

Midterm Statistics



The Story So Far



- As of HW4:
  - See how to compile a C-like language to x86 assembly by way of the LLVM IR
- Main idea 1:
  - Translation by way of a series of languages, each with well-defined semantics
- Main idea 2:
  - Structure of the semantics (e.g. scoping and/or type-checking rules) guides the structure of the translation

What's next?

- Source language features:
  - First-class functions
  - Objects & Classes
  - Polymorphism
  - Modules

  ⇒ How do we define their semantics? How do we compile them?

- Performance / Optimization:
  - How can we improve the quality of the generated code?
  - What information do we need to do the optimization?

  ⇒ Static analyses

Untyped lambda calculus
Substitution
Evaluation

FIRST-CLASS FUNCTIONS

"Functional" languages

- Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?

Free Variables and Scoping

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.
- We say that the variable `x` is *free* in `fun y -> x + y`
  - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by "`fun y`" and its scope is the body "`x + y`" in the expression `fun y -> x + y`

- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: λ x. e
- It has variables, functions, and function application.
  - That's it!
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for "functional" languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =
  | Var of var          (* variables                 *)
  | Fun of var * exp    (* functions: fun x -> e *)
  | App of exp * exp    (* function application   *)
```

Concrete syntax:

exp ::=

| x	*variables*
| `fun` x -> exp	*functions*
| exp_1 exp_2	*function application*
| (exp)	*parentheses*

Values and Substitution

- The only values of the lambda calculus are (closed) functions:

```
val ::=
      | fun x -> exp      functions are values
```

- To *substitute* a (closed) value `v` for some variable `x` in an expression `e`
  - Replace all *free occurrences* of `x` in `e` by `v`.
  - In OCaml: written `subst v x e`
  - In Math: written e{v/x}

- Function application is interpreted by *substitution*:

```
  (fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

Lambda Calculus Operational Semantics

- Substitution function (in Math):

$x\{v/x\}$	$= v$	*(replace the free x by v)*
$y\{v/x\}$	$= y$	*(assuming $y \neq x$)*
$(\text{fun x -> exp})\{v/x\}$	$= (\text{fun x -> exp})$	*(x is bound in exp)*
$(\text{fun y -> exp})\{v/x\}$	$= (\text{fun y -> exp}\{v/x\})$	*(assuming $y \neq x$)*
$(e_1\ e_2)\{v/x\}$	$= (e_1\{v/x\}\ e_2\{v/x\})$	*(substitute everywhere)*

- Examples:

x y {(fun z ->z)/y} ⇒ x (fun z -> z)

(fun x -> x y){(fun z -> z) / y} ⇒ (fun x -> x (fun z -> z))

(fun x -> x){(fun z -> z) / x} ⇒ (fun x -> x) // x is not free!

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : var list =
  begin match e with
    | Var x         -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- A lambda expression `e` is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$
\begin{aligned}
fv(x) &= \{x\} \\
fv(\text{fun } x \text{ -> } exp) &= fv(exp) \setminus \{x\} \quad \textit{(`x' is a bound in exp)} \\
fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2)
\end{aligned}
$$

Operational Semantics

- Specified using just two inference rules with judgments of the form exp ⇓ val
  - Read this notation a as "program exp evaluates to value val"
  - This is *call-by-value* semantics: function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \text{ -> } exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

See fun.ml

IMPLEMENTING THE INTERPRETER

Adding Integers to Lambda Calculus

exp ::=

…
n — *constant integers*
exp_1 + exp_2 — *binary arithmetic operation*

val ::=

`fun x -> exp` — *functions are values*
n — *integers are values*

n{v/x}	= n	*constants have no free vars.*
$(e_1 + e_2)\{v/x\}$	$= (e_1\{v/x\} + e_2\{v/x\})$	*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \Downarrow (n1 \; [\![+]\!] \; n2)}$$

Object-level '+' Meta-level '+'