• HW4: OAT v. 1.0
  – Parsing & basic code generation
  – **Due:** Tuesday, November 12\textsuperscript{th} at 23:59
See HW4
• Simple C-like Imperative Language
  – supports 64-bit integers, arrays, strings
  – top-level, mutually recursive procedures
  – scoped local, imperative variables

• See examples in HW 4

• How to design/specify such a language?
  – Grammatical constructs
  – Semantic constructs
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:
If
  Eq
  Assn
b	0	a	1

Intermediate code:
11:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:

Analysis & Transformation

Lexical Analysis

Parsing

Backend

Assembly Code
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2: ...

...
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

```ml
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?
  - In an interpreter? In a compiled language?
(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language
  - Note: we’re writing (fun x -> e) lambda-calculus notation: \( \lambda x. e \)
- 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

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

exp ::= 
  | x variables
  | fun x -> exp functions
  | exp1 exp2 function application
  | ( exp ) parentheses
```

Zhendong Su    Compiler Design
Values and Substitution

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

\[
\text{val ::= } \\
| \text{fun } x \rightarrow \text{exp} \quad \text{functions are values}
\]

• To \textit{substitute} a (closed) value \(v\) for some variable \(x\) in an expression \(e\)
  – Replace all \textit{free occurrences} of \(x\) in \(e\) by \(v\)
  – In OCaml: written \texttt{subst } \(v\) \(x\) \(e\)
  – In Math: written \(e\{v/x\}\)

• Function application is interpreted by \textit{substitution}

\[
\text{(fun } x \rightarrow \text{fun } y \rightarrow x + y) \ 1 \\
= \text{subst } 1 \ x \ (\text{fun } y \rightarrow x + y) \\
= (\text{fun } y \rightarrow 1 + y)
\]
Lambda Calculus Operational Semantics

• Substitution function (in Math)

\[ x\{v/x\} = v \quad \text{(replace the free } x \text{ by } v) \]
\[ y\{v/x\} = y \quad \text{(assuming } y \neq x) \]
\[ \text{(fun } x \rightarrow \text{exp)}\{v/x\} = \text{(fun } x \rightarrow \text{exp)} \quad \text{(x is bound in exp)} \]
\[ \text{(fun } y \rightarrow \text{exp)}\{v/x\} = \text{(fun } y \rightarrow \text{exp}\{v/x\}) \quad \text{(assuming } y \neq x) \]
\[ (e_1 \ e_2)\{v/x\} = (e_1\{v/x\} \ e_2\{v/x\}) \quad \text{(substitute everywhere)} \]

• Examples:

\[ x \ y \ ((\text{fun } z \rightarrow z)/y) \Rightarrow x \ (\text{fun } z \rightarrow z) \]
\[ (\text{fun } x \rightarrow x \ y)((\text{fun } z \rightarrow z) / y) \Rightarrow (\text{fun } x \rightarrow x \ (\text{fun } z \rightarrow z)) \]
\[ (\text{fun } x \rightarrow x)((\text{fun } z \rightarrow z) / x) \Rightarrow (\text{fun } x \rightarrow x) \quad // x \text{ is not free!} \]
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
Free Variable Calculation

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

```ocaml
let rec free_vars (e:exp) : VarSet.t =
  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

\[
\begin{align*}
fv(x) &= \{x\} \\
fv(\text{fun } x \to \text{ exp}) &= fv(\text{exp}) \setminus \{x\} \quad (\text{"x" is a bound in exp}) \\
fv(\text{exp}_1 \ \text{exp}_2) &= fv(\text{exp}_1) \cup fv(\text{exp}_2)
\end{align*}
\]
Variable Capture

• Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables

\[
\text{(fun x -} \to (x \ y)) \ \{\text{(fun z -} \to x) / y\} \quad \text{Note: x is free in (fun z -} \to x) \\
= \quad \text{fun x -} \to (x \ \text{(fun z -} \to x)) \quad \text{free x is } \text{captured}!!
\]

• Usually not the desired behavior
  – This property is sometimes called "dynamic scoping"
    The meaning of "x" is determined by where it is bound dynamically, not where it is bound statically
  – Some languages (e.g. Emacs Lisp) are implemented with this as a "feature"
  – But, leads to hard to debug scoping issues
Alpha Equivalence

• Note that the names of bound variables don't matter
  – i.e. it doesn't matter which variable names you use, as long as we use them consistently

  (fun x -> y x) is the "same" as (fun z -> y z)

  the choice of "x" or "z" is arbitrary, as long as we consistently rename them

  – Two terms that differ only by consistent renaming of bound variables are called alpha equivalent

• The names of free variables do matter

  (fun x -> y x) is not the "same" as (fun x -> z x)

  Intuitively: y an z may refer to different things from some outer scope
Fixing Substitution

- Consider the substitution operation

  \[ e_1{e_2/x} \]

- To avoid capture, we define substitution to pick an alpha equivalent version of \( e_1 \) such that the bound names of \( e_1 \) don't mention the free names of \( e_2 \)
  - Then do the "naïve" substitution

For example: \((\text{fun } x \to (x \ y))\) \{\((\text{fun } z \to x) \ / \ y\)\}

  \[ = (\text{fun } x' \to (x' (\text{fun } z \to x))) \quad \text{rename } x \text{ to } x' \]
Operational Semantics

- Specified using just two inference rules with judgments of the form exp $\Downarrow$ val
  - Read this notation as “program exp evaluates to value val”
  - This is call-by-value semantics: function arguments are evaluated before substitution

\[ v \Downarrow v \]

“Values evaluate to themselves”

\[
\begin{align*}
\text{exp}_1 & \Downarrow \text{(fun } x \rightarrow \text{exp}_3) & \text{exp}_2 & \Downarrow v & \text{exp}_3\{v/x\} & \Downarrow w \\
\hline
\text{exp}_1 \text{ exp}_2 & \Downarrow w
\end{align*}
\]

“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