Announcements

• **Midterm next **Thursday (Feb 13) **in class**
  • Includes this week’s topics: Polymorphism and Type Inference
  • Practice questions at bottom of Schedule webpage

• **HW#4 due Friday Feb 21**
  • Will be posted later today
  • Problem 1: Lexer/Parser for NanoML
  • Problem 2: Evaluator for NanoML
Structure of a PL Implementation

- Lexer
- Parser
- Evaluator (a.k.a. Interpreter)
- Type Checker
- Other Static Analysis
- Native Code Generator

HW#4 Problem 1

HW#4 Problem 2

Perhaps later assignment

cse131 and cse231
Announcements

• Midterm next **Thursday (Feb 13)** in class
  • Includes this week’s topics: Polymorphism and Type Inference
  • Practice questions at bottom of Schedule webpage

• HW#4 due Friday Feb 21
  • Will be posted later today
  • Problem 1: Lexer/Parser for NanoML
  • Problem 2: Evaluator for NanoML

• **Work on Problem 2 before midterm**
  • Will help clarify environments and closures
Review Session?

(a) Sunday evening
(b) Monday evening
(c) Tuesday evening
(d) Tuesday in class (instead of lexing/parsing for HW#4)
(e) Never
Values of Functions = “Closures”

Two questions:

What is the value:

1. ... of a function?

Closure =

Code of Fun. (formal x + body e)
+ Environment at Fun. Definition
Two questions:

What is the value:

1. ... of a function?

2. ... of a function “application” (call)?
Free vs. Bound Variables

```ocaml
let a = 20;;
let f x =
  let y = 1 in
  let g z = y + z in
  a + (g x)
;;
f 0;;
```

Environment **frozen** inside function definition

Used to evaluate function application \((e1 \ e2)\)

Which vars are needed from frozen env?
Free vs. Bound Variables

let a = 20;;
let f x =
  let y = 1 in
  let g z = y + z in
  a + (g x)
;;
f 0;;

Inside a function,
A “bound” occurrence:
1. Formal variable
2. Variable bound in let-in
x, y, g are “bound” inside f

A “free” occurrence:
• An unbound variable
a is “free” inside f

Frozen Environment
needed for values of free vars
Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
  a + (g x)
;;

f 0;;
```

**Bound** values determined when function is evaluated (“called”)
- Arguments
- Local variable bindings
Values of Function Application

Value of a function “application” (call) \((e_1 \ e_2)\)

1. Eval \(e_1\) in current-env \(E_0\) to a closure
   \[= \text{code (formal } x + \text{ body } e) + \text{ closure-env } E\]

2. Eval \(e_2\) in current-env \(E_0\) to get (argument) \(v_2\)

3. Evaluate body \(e\) in env \(E\) extended with \(x := v_2\)
let f g =
  let x = 0 in
  g 2
;;

let x = 100;;

let h y = x + y;;

let res = f h;;

Q: What is the value of `res`?

(a) Syntax Error
(b) 102
(c) Type Error
(d) 2
(e) 100
Immutability: The Colbert Principle

“A function behaves the same way on Wednesday as it behaved on Monday, no matter what happened on Tuesday!”
Static/Lexical Scoping

• For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment

• Static/Lexical: Determined from the program text
  - Without executing the program

• Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence
What type to assign to ... ?

# let id x = x ;;

These types are too specific:

- `int -> int`
- `bool -> bool`
- `int * bool -> int * bool`
- `(int -> bool) -> (int -> bool)`
- ...
What type to assign to ... ?

```
# let id x = x ;;;
val id : 'a -> 'a = <fun>
```

'

'a -> 'a

Read “For every type ’a, id has type ’a -> ’a ”

Type variables prefixed with single quote
What type to assign to ... ?

```ocaml
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

\[ 'a \rightarrow 'a \approx \forall 'a. 'a \rightarrow 'a \]

Implicit quantification over every type \( 'a \)
What type to assign to ... ?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>
# let snd x y = y ;;
```

\[
\forall \alpha, \beta. \ \alpha \to \beta \to \alpha
\]

Implicit quantification over every type \( \alpha \) and \( \beta \)
What type to assign to ... ?

```ocaml
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>
```
Polymorphism

- Poly = *many*, morph = *kind*
- Also called “forall types”
- Also called “parametric polymorphism”
- Type vars can be instantiated with any type
  - ML infers instantiations automatically!

```
# id 5 ;;
- : int = 5
≈ id[int] 5

# id (fun x -> x+1) ;;
- : int -> int = <fun>
≈ id[int->int] ...
```
Polymorphism

\[ \text{id} :: \forall a. \ a \rightarrow a \]

\[ \text{id[int]} :: \text{int} \rightarrow \text{int} \]

\[ \text{id[int->int]} :: (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \]

Instantiation is always implicit, OCaml doesn’t allow explicit type arguments

# id 5 ;;
- : int = 5

# id (fun x -> x+1) ;;
- : int -> int = <fun>

≈ id[int] 5

≈ id[int->int] ...
What does res1 evaluate to?

(a) Type Error
(b) 'b -> 'a = <fun>
(c) 'b -> int = <fun>
(d) 'b -> 'b = <fun>
(e) 'b -> int = <fun>

```ocaml
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>

# let res1 = fst 17 ;;
???
```
What does res1 evaluate to?

(a) Type Error
(b) 'b -> 'a = <fun>

(c) 'b -> int = <fun>
(d) 'b -> 'b = <fun>
(e) 'b -> int = <fun>
What does res2 evaluate to?

(a) Type Error
(b) 'b -> 'a = <fun>
(c) 'b -> int = <fun>
(d) 'b -> 'b = <fun>
(e) 'b -> int = <fun>
What does res2 evaluate to?

(a) Type Error
(b) 'b -> 'a = <fun>
(c) 'b -> int = <fun>
(d) 'b -> 'b = <fun>
(e) 'b -> int = <fun>

# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>

# let res2 = snd 17 ;;
???
• What’s with underscores in type vars ???

• We won’t go into it in this course...

  If you’re really curious: http://mlton.org/ValueRestriction

• For our purposes, assume: \( '_a \approx 'a \)

```ocaml
# let res1 = fst 17 ;;
val res1 : '_a -> int = <fun>

# let res2 = snd 17 ;;
val res2 : '_a -> '_a = <fun>
```
Where did the ‘b variables go???

For all types are equivalent up to renaming:

\[ \forall a, b, c. \quad \text{type} \quad a \rightarrow \text{int} \quad \approx \quad b \rightarrow \text{int} \quad \approx \quad c \rightarrow \text{int} \]

\[ \forall a, b, c. \quad \text{type} \quad a \rightarrow a \quad \approx \quad b \rightarrow b \quad \approx \quad c \rightarrow c \]

```ml
# let res1 = fst 17 ;; val res1 : 'a -> int = <fun>

# let res2 = snd 17 ;; val res2 : 'a -> 'a = <fun>
```
• Where did the ’b variables go ???

• For all types are equivalent up to renaming

\[ \begin{align*} 
'a \to int & \approx 'b \to int \approx 'c \to int \\
'a \to 'a & \approx 'b \to 'b \approx 'c \to 'c \\
'a \to 'b \to 'a & \approx 'b \to 'a \to 'b 
\end{align*} \]

• ML infers type variable names from left to right, starting with ’a, ’b, ’c, ...
Polymorphism Enables Reuse

- If function (algorithm) is independent of type, can reuse generic code for all types!

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev</td>
<td>'a list → 'a list</td>
</tr>
<tr>
<td>len</td>
<td>'a list → int</td>
</tr>
<tr>
<td>swap</td>
<td>'a * 'b → 'b * 'a</td>
</tr>
<tr>
<td>sort</td>
<td>('a → 'a → bool) → 'a list → 'a list</td>
</tr>
<tr>
<td>filter</td>
<td>('a → bool) → 'a list → 'a list</td>
</tr>
<tr>
<td>map</td>
<td>('a → 'b) → 'a list → 'b list</td>
</tr>
<tr>
<td>foldl</td>
<td>('a → 'b → 'a) → 'a → 'b list → 'a</td>
</tr>
<tr>
<td>foldr</td>
<td>('a → 'b → 'b) → 'a list → 'b → 'b</td>
</tr>
</tbody>
</table>
Polymorphic Data Types

- Data types are also polymorphic!

  ```plaintext
  type 'a list =
  | Nil
  | Cons of 'a * 'a list
  ```

- Type variable is instantiated for each use:

  Cons(1,Cons(2,Nil)) : int list
  Cons("a",Cons("b",Nil)) : string list
  Cons((1,2),Cons((3,4),Nil)) : (int*int) list
  Nil : 'a list
Option Types

• Consider the following type:

```haskell
type 'a option =
| None
| Some of 'a
```

• How could this be useful?
• Remember this function?
  \[
  \text{val assoc: } 'b \rightarrow 'a \rightarrow ('a*'b) \text{ list} \rightarrow 'b
  \]

• We had to pass in a “default” value (Yuck!)

• Instead, we could return an option
  - \textbf{None} denotes “failure”
  - \textbf{Some} denotes “success”

\[
\text{val assoc: } 'a \rightarrow ('a*'b) \text{ list} \rightarrow 'b \text{ option}
\]
Exercise: Simple Calculator

Let’s write a function

```ocaml
type expr =
| Num of int
| Add of expr * expr
| Div of expr * expr
```

val eval : expr -> int
Exercise: Simple Calculator

Let's write a function

```
type expr =
  | Num of int
  | Add of expr * expr
  | Div of expr * expr
```

```
val safeEval : expr -> int option

... that returns None if a div-by-zero occurs
Syntax for datatype with multiple type variables

type ('a, 'b) tree =
    Leaf
  | Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree
Q: What is the type of \texttt{x}?

```ocaml
type ('a, 'b) tree =
    Leaf
  | Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree

let \texttt{x} = Node ("alice", 5, Leaf, Leaf)
```

(a) (int, string) tree
(b) ('a,'b) tree
(c) int tree
(d) string tree
(e) (string, int) tree
Multiple Type Variables

- Data types can have multiple type vars

```haskell
type ('a,'b) tree =
| Leaf
| Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree
```

- Type variables instantiated for each use:

```
Node("alice",2,Leaf,Leaf) : (string, int) tree
Node("charlie",3,Leaf,Leaf) : (string, int) tree
Node("bob",13,
    Node("alice",2,Leaf,Leaf),
    Node("charlie",3,Leaf,Leaf)) : (string, int) tree
```
Multiple Type Variables

- Data types can have multiple type vars

```haskell
type ('a, 'b) tree =
  | Leaf
  | Node of 'a * 'b * ('a, 'b) tree * ('a, 'b) tree
```

- Type variables instantiated for each use:

```haskell
Node("alice", 2, Leaf, Leaf) : (string, int) tree
Node("charlie", 3, Leaf, Leaf) : (string, int) tree
Node("bob", 13,
    Node("alice", 2, Leaf, Leaf),
    Node(3, "charlie", Leaf, Leaf))
```

`TYPE ERROR`
Binary Search Trees

\[
\text{type } (\texttt{\textquoteleft a,\textquoteleft b}) \text{ tree } = \\
| \texttt{Leaf} \\
| \texttt{Node of } \texttt{\textquoteleft a } \ast \texttt{\textquoteleft b } \ast (\texttt{\textquoteleft a,\textquoteleft b}) \text{ tree } \ast (\texttt{\textquoteleft a,\textquoteleft b}) \text{ tree}
\]

**Node** \((\texttt{key, value, left, right})\)

**BST Property:**
\[
\text{keys in left } < \texttt{key} < \text{keys in right}
\]
BST Property: keys in left < key < keys in right

Node("bob", 13,
    Node("alice", 2, Leaf, Leaf),
    Node("charlie", 3, Leaf, Leaf))
Exercise: BST Lookup

BST Property: keys in left < key < keys in right

```ocaml
type ('a,'b) tree =
  | Leaf
  | Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree
```

Write a function to lookup keys...

```ocaml
val lookup : 'a -> ('a,'b) tree -> 'b option
```
type ('a, 'b) weirdlist =
    Nil
  | Cons 'a * ('b, 'a) weirdlist

Pick a well-typed expression from:

(a) Cons(1, Cons("a", Cons(3.14, Nil)))
(b) Cons(1, Cons("a", Cons(1, Nil)))
(c) Cons(1, Cons("a", Cons("a", Nil)))
(d) Cons(1, Cons(1, Cons("a", Nil)))
(e) Cons(1, Cons(1, Cons(1, Nil)))
Polymorphic Data Structures

- **Container** data structures independent of type!
- Appropriate type is **instantiated** at each use

\[
\begin{align*}
\text{'a list} \\
\text{('a, 'b) tree} \\
\text{('a, 'b) hashtbl} \\
\text{...}
\end{align*}
\]

- Static type checking catches errors early
  - Cannot add `int` key to `string` hashtable

- **Generics**: in Java, C#, VB (borrowed from ML)
What does foo evaluate to?

```ocaml
# let foo x =
  if x > 0 then
    x * x
  else
    failwith "x is negative" ;;

(a) Type Error
(b) int -> string = <fun>
(c) int -> int    = <fun>
(d) int -> 'a     = <fun>
(e) 'a  -> 'b     = <fun>
```
**failwith**

\[
\text{failwith :: } \forall \alpha. \text{ string } \rightarrow '\alpha
\]

- Call to `failwith` can be instantiated to produce any type.

```ocaml
# let foo x =
  if x > 0 then
    x * x
  else
    failwith "x is negative" ;;
val foo : int -> int = <fun>
```

- Can be used to “escape with fatal error”
- Very useful for building up skeleton code
  - Can use `failwith "..."` for sub-expressions not yet written, but can still type check overall function!
OCaml has a more general exception handling mechanism

- raise e
- try e with pat1 -> e1 | ... | patn -> en

But we won’t study this any further...
What does bar evaluate to?

# let bar x = failwith "!!!" ;;

(a) Type Error
(b) 'a -> string = <fun>
(c) 'a -> 'a = <fun>
(d) 'a -> 'b = <fun>
Recall our trusty friend, fold

“fold-right”
**a.k.a.** List.fold_right

```ocaml
let rec foldr f acc xs =
  match xs with
  | []   -> acc
  | x::xs' -> f x (foldr f acc xs')
```

“fold-left”
**a.k.a.** List.fold_left

```ocaml
let foldl f initAcc xs =
  let rec helper acc xs =
    match xs with
    | []   -> acc
    | x::xs' -> helper (f acc x) xs'
  in
  helper initAcc xs
```

\[ (\forall a, b. (a \to b \to b)) \to (a \to b \to b) \]

\[ \approx \]

\[ \forall a, b. (a \to b \to b) \to a\text{ list} \to b \to b \]
Recall our trusty friend, fold

“fold-right”  
**a.k.a.** List.fold_right

```ml
let rec foldr f acc xs =  
  match xs with  
  | []         -> acc  
  | x::xs'     -> f x (foldr f acc xs')
```

```
('a → 'b → 'b) → 'a list → 'b → 'b
```

“fold-left”  
**a.k.a.** List.fold_left

```ml
let foldl f initAcc xs =  
  let rec helper acc xs =  
    match xs with  
    | []         -> acc  
    | x::xs'     -> helper (f acc x) xs'  
  in  
  helper initAcc xs
```

```
('a → 'b → 'a) → 'a → 'b list → 'a
```

- A bit annoying that the interfaces differ...
- Let’s write a version of fold-left of type:

```
('a → 'b → 'b) → 'a list → 'b → 'b
```