CS 3110: Functional Programming and Data Structures

Course Notes

Chirag Bharadwaj
cb625@cornell.edu

18 July 2015

Lecture 1

- 15 weeks left to finish computer programming education (computer SCIENCE $\gg$ just programming)
- Programming well = hard problem (but is it NP-complete?)
- High variance in industry professionals' productivity due to lack of streamlined education (10x)
- CS 3110 goals: hard work, patience, open mind, BETTER PROGRAMMING
- How the $*$110 series of CS classes work: (*breadth* in PL)
  - CS 1110: Coding for your professor
  - CS 2110: Coding for yourself
  - CS 2112: Coding for yourself/your classmates (i.e. `Some` others, will laugh later…)
    - Honors 2110, more work for sure
  - CS 3110: Coding for others
    - Emphasis: *design*, *performance*, *correctness*
    - Collaboration tools and techniques
    - Writing code for/with other people
  - CS 4110: Coding for mathematicians
  - CS 6110: Coding for purists (CS 4110 on steroids) with naiveté
  - CS 7110: Coding for purists in the modern, concurrent world
    - Now called CS 6112 ("Honors CS 6110"? No. See why:)
- About the 611$*$ series of CS classes: (*depth* in PL)
  - CS 6110: Advanced Programming Languages and Logics
  - CS 6112: Foundations of Concurrency
  - CS 6113: Language-Based Security
  - CS 6114: Network Programming Languages
  - CS 6115: Certified Software Systems
  - CS 6116: CS 6110 Practicum
  - CS 6117: Category Theory for Computer Scientists
  - CS 6118: Types and Semantics
  - CS 6111 and CS 6119 TBD (stay tuned through 2016)
  - NO other school offers such a deep series in PL (not Princeton, not MIT, etc.)
  - You can't take them all, unfortunately (not enough time!)
  - Do a MS/PhD…

- Back to CS 3110 goals. How to achieve? We need methodologies/structure:
  1. Functional programming (OCaml)
     - Challenge: Think OUTSIDE of Java/Python/C *imperative* family of languages
     - Programming transcends the PL of choice
       - All languages have same features: *syntax*, *semantics*, *idioms*, *tools*
  2. Data structures and modern programming paradigms
     - Challenge: Think about *abstraction*
     - Rigorously analyze performance and correctness (e.g. pre-CS 4820)
     - Learn how to write *concurrent* programs
     - Learn how to write *scalable* programs
  3. Software engineering
     - Experience with modular design, specification, integrated testing, source control, code reviews
       - i.e. CS 2112 part II
     - Exposure to tools used in the real world (e.g. Linux, `git`)
- Roadblocks:
  - Programming $\neq$ Java
    - Let go of Java!
    - Paradigm won't help here... we'll talk about OOP towards the end
  - Programming $\neq$ hacking until desired functionality achieved
    - Professional: disciplined work habits
    - THINK first – there are an infinite number of incorrect programs!
    - All code must be: maintainable, reliable, provable, efficient, readable, testable (MR. PERT)
- What is OCaml?
  - Functional programming language (FPL)
  - Objective Caml (Categorically-Abstract ML)
  - ML is a family of languages, originally "meta-language" for tools
  - Take CS 6117 for more on categories
- Why OCaml?
  - Immutable programming
    - Variables cannot be altered – easier to reason about programs
  - Algebraic datatypes, pattern matching
    - Defining/manipulating complex data structures is easy to express
  - First-class functions
    - Functions passed around as values, like variables
  - Static type-checking
    - No run-time errors (only *exceptions*, like in Java, C, etc.)
  - Automatic type inference
    - No type-based burden (unlike in Java, C, etc.)
  - Parametric polymorphism
    - Enables abstraction to be used across many data types
  - Garbage collection
    - Automated memory management – no leaks, `valgrind`, etc.
- What probably struck out to you was "immutable abstraction"

- Why immutability?
  - Imperative = mutable programming
    - Commands specify computation by destructively changing *state*

      ```
      x = x + 1; // Modifies x
      a[i] = 42; // Destroys the previous value
      p.next = p.next.next; // Original Node lost
      ```
    - Functions/methods have side effects

      ```
      int wheels(Vehicle v) {
          v.size++; // Modifies v.size
          return v.numWheels;
      }
      ```
    - Need complex "patterns" like Builder model, AbstractFactory pattern, etc. to avoid changing state
    - Claims loose coupling, actually tightly coupled modules relative to FP
  - Mutable programming not well-suited to modern computing standards!
    - Fantasy of mutability:
      - ☐ There is only one state
      - ☐ The computer does one thing at a time
    - Reality of world:
      - ☐ There is no single state
      - ☐ Programs have many threads, spread across many cores, spread across many processors, spread across many computers, spread across many networks... each with its own view of memory (technically not true – computers use *time-sharing* of resources, see CS 3410 for more details)
      - ☐ There is no single program (robust applications invoke 3-4 programs simultaneously)
  - Functional = immutable programming
    - Expressions specify computations without destroying state
      - ☐ Variables never change value
      - ☐ Functions never have side effects
    - Effects on reality of world:
      - ☐ No need to even think about state
      - ☐ Powerful way to build concurrent programs
- FPL = higher level of abstraction, easier to develop robust pieces of software (you'll see soon)
- FP predicts the future! These concepts were once dismissed as "too pedantic in nature":
  - Garbage collection
    - Java (1995), LISP (1958)
  - Generics/parametric polymorphism
    - Java 5 (2004), ML (1990)
  - Higher-order functions
    - Java 8 (2014), LISP (1958)
  - Type inference
    - Java 8 (2014), ML (1990)
- FPL matters in the real world:
  - F# (Microsoft framework)
  - Scala (Twitter, LinkedIn, FourSquare, etc.)
  - Java 8 (being adopted everywhere)
  - Haskell (small companies, mathematical programmers)
  - Erlang (Facebook chat and other distributed systems – see CS 3410/4410)

  - OCaml (Jane Street)

- Let's examine samples of Java vs. Ocaml in standard contexts (these are PREVIEWS, syntax isn't obvious)

- The sum of squares of integers:

  - A mutable version in Java:

```
// Yields ∑_{i=1}^n i^2
int sumSquares(int n) {
    int sum = 0;
    for (int k = 1; k <= n; k++) {
        sum = sum + k*k; // Destroys state
    }
    return sum;
}
```

  - An immutable version (contrived) in Java:

```
// Yields ∑_{i=1}^n i^2
int sumSquares(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n*n + sumSquares(n-1);
    }
    // No state to destroy, i.e. all recursive calls
}
```

  - The OCaml version is like the "inefficient" Java one except cleaner, more compact, and reads like math:

```
(* Yields ∑_{i=1}^n i^2 *)
let rec sum_squares n =
    if n = 0 then 0
    else n*n + sum_squares (n-1)
```

- Reversing a linked-list:

  - Classic overly-verbose Java:

```
// Reverses a linked-list given the head Node to the linked-list
Node<T> reverse(Node<T> list) {
    Node<T> reversedList = null;
    while (list != null) {
        Node<T> temp = list.next;
        list.next = reversedList;
        reversedList = list;
        list = temp;
    }
    return reversedList;
}
```

  - Utilizing pattern-matching in OCaml (e.g. see Java 8):

```
(* Reversing a list with elegance and recursion *)
let rec reverse list =
    match list with
    | [] -> []
    | head :: body -> List.append (reverse body) head
```

- Implementing quicksort:

  - A fairly efficient version in Java:

```
/** This algorithm assumes that T implements Comparable. */

/** Sorts list[l...r) */
```

```
T[] quicksort(T[] list, int l, int r) {
    if (l == r-1) return; // Base case: already sorted
    int index = partition(list, l, r);
    quicksort(list, l, index);
    quicksort(list, index, r);
    return list;
}

/**
 * Partition list into list[l...index) and list[index...r), where
 * l < index < r, and all elements in list[l...index) are less than
 * or equal to all elements in list[index...r).
 *
 * Requires: 0 ≤ l, r ≤ list.length, and r − l ≥ 2.
 */
int partition(T[] list, int l, int r) {
    T pivot = list[l]; // Better: swap list[l] with random element first
    int i = l, j = r;
    do j--; while (list[j].compareTo(pivot) > 0);
    while (i < j) {
        swap(list[i], list[j]); // See Java 8 for this
        do i++; while (list[i].compareTo(pivot) < 0);
        do j--; while (list[j].compareTo(pivot) > 0);
    }
    return j+1;
}
```

- As expected, a very quick way to do it in OCaml:

```
(* Returns list, sorted according to the partial order ≤.
 * Poor pivot choice, as in the Java example *)
let rec quicksort list =
    match list with
    | [] -> []
    | pivot :: rest ->
        let (left, right) = partition ((<=) pivot) rest in
        List.append (List.append (quicksort left) [pivot]) (quicksort right)
```

- Generally interesting things to note:
  - OCaml seems to have good type inference
  - OCaml handles polymorphism well/almost invisibly
  - Pattern-matching is useful and quick
  - There is a mathematical structure of organization, shows in syntax
  - No pesky objects and hierarchies getting in the way of calling functions
    - Everything is a function, no loops?!
    - Some JavaScript-esque appearance/behavior
  - OCaml syntax is mostly succinct/brief
- We will learn more about this syntax/way of thinking next time…

Recitation 1

- OCaml = FPL
- Functional vs. imperative… differences?
  - Different *execution model*

  - Imperative languages (e.g. C, Java) based on commands that change machine state
  - Functional languages (e.g. OCaml, Haskell) based on evaluating expressions to produce values

- Might seem like a stretch in real world… real programs **do** things, not **compute** things
- Later we will learn about *side-effects*, i.e. "evaluate-then-do/display" model
- Thus OCaml is not a PURE functional language, unlike Haskell, which does NOT allow for side-effects
- OCaml is better-suited than OOP to build large-scale, correct, and understandable expressions
- We will focus on understanding expressions for this recitation
- Assumption: already have OCaml properly installed on a Linux/Unix shell
  - OCaml *toplevel* is the system to interact with the compiler directly (*read-evaluate-print loop*, or REPL)
  - Toplevel allows for some syntactic sugar
  - User types in an expression, compiler evaluates and tells resulting value as well as resulting type
  - How to access: Type in `ocaml` at terminal
  - CS 3110 specialized VM has enhanced version of `ocaml` instead: `utop` (use this)
  - How to use: Type in expression (may span several lines with line breaks), and type `;;` to compile/evaluate
- Use standard Unix approaches: `CTRL+C` to clear prompt, `CTRL+L` to clear shell, etc. (these are called *interrupts* and will reset the prompt state–see CS 3410, CS 4410 for more information), `cd`, `pwd`, `ls`, …
- Can use the psuedo-C approach: Put OCaml code into a file (do NOT use `;;`, this is not part of language syntax… just needed for compiler) and type `#use "file";;` to essentially copy the code onto the shell and evaluate the expression or several (compounded) expressions at once
  - If in a different directory, specify the relative/absolute path and use standard Unix syntax: `.../file.ml`
- We will introduce the notion of *expressions* in OCaml by presenting it with a context-free BNF specification (Backus-Naur form)… should be familiar with this from CS 2112, Assignment 4/5
- BNF (counterpart: EBNF, or *extended*-BNF) allows for simple recursive definitions of types/grammar of PL
- BNF = metalanguage; syntax is a set of *derivation rules* (i.e. inductive definitions):
  - All definitions are of the form `<definition> ::= <another_definition> | command`
  - `<*>` refers to a *non-terminal* definition, i.e. must keep parsing until reach a terminal definition
  - `|` refers to a choice, i.e. select ONE of the options ("pipe")
  - Compare this with EBNF: ***`definition`*** $\rightarrow$ ***`another_definition`*** `| commmand`
  - EBNF is easier to read (***`non-terminal`*** vs. `terminal`), easier to encode, harder to support in compiler
  - We will mix and match syntax with BNF, EBNF, regexes–i.e. HBNF, or *half-extended*-BNF
- We will slowly build up the OCaml syntax by starting with fundamental building blocks (just a BROAD overview today… some of these topics will be covered again in more depth soon!)
  - We know about primitive types like integers, floating point numbers, booleans, etc.
  - We know we can chain primitive types together to make other types
  - We know that there are certain expressions in every language that signify control flow (`if-else`), etc.
  - We know we need some sort of declaration/assignment procedure
  - We know we need some functions or some sort of procedure-binding
  - We know we need some sense of exception/error handling
  - Do we need anything else? Are there other features…? Yes! *"There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know."*

Rudimentary Encodings (part of every language)

Challenge: Encode HBNF in HBNF.

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |
w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U
| V | W | X | Y | Z
special ::= ! | @ | # | $ | % | ^ | & | * | ( | ) | [ | ] | { | } | : | ; | ‘ | ’ | " | , | . | ?
| / | < | > | - | _ | + | = | | | \ | ~ | ␣
token ::= digit | letter | special
```

Primitive Types

```
int ::= digit digit*
float ::= (int int*).int*
boolean ::= true | false
char ::= ‘token’
string ::= "token*"
```

Operator Types

```
neg ::= - | not
op ::= + | - | * | / | +. | -. | *. | /. | mod | < | > | <= | >= | = | ^
```

Commands (this will change/be added to as time goes on)

```
value ::= int | float | boolean | char | string
expr ::= value | neg expr | expr1 op expr2 | (expr) | if expr1 then expr2 else expr3
```

- Type-checking is an important feature of OCaml
- Let us review the types of each of these operators and their respective operands
- Can view operators as “functions” that take in LHS, return RHS (important way to think)

Table 1: Unary operators

neg	Type
-	int -> int
	float -> float
not	boolean -> boolean

Table 2: Binary operators

op	Type
+, -, *, /	int -> int -> int
+., -., *., /.	float -> float -> float
mod	int -> int -> int
<, >, <=, >=, =	int -> int -> bool
	float -> float -> bool
^	string -> string -> string

- The binary operators (`op`) seem to have an unusual type, even though they are functions that take in two inputs and return a single output
  - There are separate operators/functions for `int`s vs. `float`s
  - Unary operators are *prefix* operators, while binary operators are *infix* operators
  - `=` is for checking equality, NOT assignment (there is no such thing in FP)
- Functions can actually only take in one argument, as we will learn in *lambda calculus*… this is a *curried* form

- More on that later! Way to view this correctly is to consider *operator associativity* for `->`
  - All operators are either *unassociative* (cannot be chained), *associative* (can be grouped any way), *left-associative* (must fold/group to the left), or *right-associative* (must fold/group to the right)
  - For example, addition is associative (e.g. $3 + 4 + 5 = (3 + 4) + 5 = 3 + (4 + 5)$)
  - No common operator has *conditional* associativity, i.e. addition, subtraction, multiplication, division are all fully associative and left/right-folding is irrelevant
  - But `->` is right-associative, so must be grouped to the right
  - Thus `type1 -> type2 -> type3` is actually `type1 -> (type2 -> type3)`, and it takes in `type1` and returns a new function that takes in `type2` and returns `type3` (i.e. functions take in at most ONE input)
  - Syntactic sugar: `(type1, type2) -> type3`
  - We will talk about this more in a few lectures (there's a special reason!)
- Now we can determine the types of expressions; main rule is: **The type of the expression is the type of the result.**
  - Style of *type annotations*: `expr:type` (this can be encoded in OCaml as well, and is a GOOD IDEA)
  - e.g. `function arg1 arg2` vs. `function ((arg1:type1) (arg2:type2)):return_type` (more explcit – otherwise OCaml does *type inference* and could cause compile-time errors)
  - Let's outline the rules for type inference:
  - If `neg:(type1 -> type2)` and `expr:type1`, then `(neg expr):type2`
  - If `op:(type1 -> type2 -> type3)`, `expr1:type1`, and `expr2:type2`, then `(expr1 op expr2):type3`.
  - If `expr1:boolean`, `expr2:a'`, and `expr3:a'`, then `(if expr1 then expr2 else expr3):a'`
    - Here, `a'` stands for a *generic*, or *polymorphic* type (can be anything)
    - Think about why we need `expr2` and `expr3` to have same type: Either branch could be taken, instruction set architecture needs to verify that program could still continue in all possible cases regardless of which branch is actually taken (see CS 3410 notes)
    - Unlike Java or C, `if-then-else` is an EXPRESSION, like the `?:` *tertiary* operator in those languages
  - If an expression does NOT satisfy these three rules, then that expression does not have an inferrable type, so a compile-time error occurs
  - i.e. inconsistently-typed arguments = invalid program = compile-time error (no run-time errors, i.e. OCaml is considered a *strongly-typed* language with *static inference* capabilities/semantics)
  - Note: invalid program $\neq$ generic type… generic type can still be inferred to be such
  - Best practice is to ALWAYS type-annotate arguments to avoid clashes (enforces a strong type check)
- We still need to discuss how to declare state, i.e. bind values to names (variables)
- A *declaration* is of the form `let name = expr`, so we can refer to expressions by `name` as well
  - Scope: Lasts until another declaration with same name occurs and *shadows* the first one
  - Can also achieve local scope by doing local-binding, i.e. declarations of the form `let decl in expr`
  - Caveat: `let`-bindings done at toplevel, or REPL, last for rest of program, i.e. it is essentially equivalent to declaring something of the form `let name = expr in (* rest of program *)`
  - Here, the declaration `decl` is evaluated, then the expression `expr` is evaluated with the bound `decl`
  - Of course, the type of `let decl in expr` is the type of `expr`, because **the type of the expression is the type of the result**
  - This is just the active way of rewriting passive mathematics: `evaluate expr(x), where x = ...`
- To reuse expressions (abstraction; more on this later), we need to factor out common behavior into a *function*
- Functions are of the form `let name args... = expr` as well! Here, `expr` is called the *body* of the function
- What does this reveal? In OCaml and FP, functions are *first-class citizens*, i.e. they are declared the same way values are declared and can be passed just like values are… FUNCTIONS ARE VALUES in FP

- Proper type annotation: `let name((x1:type1)(x2:type2)...(xn:typen)):return_type = expr`
- An example: `let square x = x*x`, or `let square (x:int) : int = x*x` (looks like math, not Java garbage)
- The type of `square` is `int -> int` (takes in an integer, returns an integer)
- In general, a function `let name((x1:type1)(x2:type2)...(xn:typen)):return_type` has a compound type: `(type1 * type2 * ... * typen -> return_type`
- There are also *recursive* functions that call themselves... need an extra `rec` declaration
- e.g. `let rec name((x1:type1)(x2:type2)...(xn:typen)):return_type` MUST call itself at least once
- How to check to see if a function application `f((x1:t1)(x2:t2)...(xm:tm)):t` is done correctly? Need to satisfy some rules, in order:
  1. Type-checking: `f` has type `(t1 * ... * tn) -> t`
  2. Partial-application: $m \leq n$ (we will learn about this later; for now, let's use the stronger condition $m = n$)
- So in our example: `square 10` will compile, but `square true` will not, and neither will `square 10 20`, etc.
- Last thing for today: A small introduction to locally-bound functions (i.e. *inner* functions, which is not the same thing as *anonymous* functions, or *lambdas* – more later, as usual)
  - Possible in Pascal, Haskell, etc., but NOT in Java, C, etc.

```
(* Cleaner version of the code *)
let fourth y =
    let square x = x*x in square(square y)

(* Excessive, but obvious that it type-checks *)
let fourth (y:int) : int =
    let square (x:int) : int = x*x in square(square y)
```

  - In either case, notice that there is a CONSERVATIVE use of parentheses, unlike C-family of languages
  - We are building bigger functions from smaller ones
  - Calling `square` outside of the scope of `fourth` gives a compile-time error due to scope
  - This is abstraction, since we are REUSING building blocks
  - This is 3110! (i.e what FP is all about – abstraction and reusing modules)
- Let's label the extensions to the expression syntax in our HBNF notation (`fun_name` is function's name)

Commands (will be updated during next lecture...)

```
value ::= int | float | boolean | char | string
decl ::= let fun_name = expr | let fun_name arg1...argn = expr | let rec name arg1...argn = expr
expr ::= value | neg expr | expr1 op expr2 | (expr) | if expr1 then expr2 else expr3 | name | let
decl in expr | fun_name expr1...exprm
```

Lecture 2

- We know some OCaml syntax; let's talk about the semantics and what these PL terms mean in general
- Five key aspects of learning ANY PL:
  1. *Syntax*: How to write compilable language constructs?
  2. *Semantics*: What does program mean? (Type-checking, evaluation rules)
  3. *Idioms*: What are typical language patterns/features used for expression?
  4. *Libraries*: What facilities/utilities are part of "standard" package?
  5. *Tools*: What things exist to make life easier? (toplevel, GUI IDE, debugger)

- Breaking down learning into 5 steps (in this order specifically) makes learning PLs easier
- CS 3110 focuses on *semantics* and *idioms–syntax* is boring, need to learn every time
- Don't complain about syntax, won't change very likely and is part of original designer's vision
- Semantics = metatool: facilitates learning other languages
- Idioms = expressing oneself: facilitates becoming a better programmer
- *Libraries*, *tools* = crucical, but keep udpating so no point getting fixated (except maybe LaTeX)
- Recall: Expressions can be arbitrarily large since subexpressions can contain subexpressions, etc. (rec. def.)
- Every expression has both syntax and semantics (see below for more on semantics):
  - Type-checking rules: Produce a type or fail with a message (invalid program)
  - Evaluation rules: Produce a value or exception or infinite loop (ONLY if expression already type-checks)
- Notice we already talked about some of this last time (how to know function application is valid, etc.)
- Values are *terminal* expressions, i.e. they don't need more evaluation (e.g. `34:int`, `42.:float`, `false:boolean`)
- Note: all values are expressions, not all expressions are values (types are not expressions though)
- Types ≠ values! e.g. `int` is not a value, `34` is – in HBNF, `value ::= <type>` means INSTANCE of type
- Notice that `string` is not really a primitive type... achieved by chaining `chars` (HBNF: `token` vs. `token*`)
- So: `"CS " ^ "3110"` is an expression, but not a value (see evaluation below)
- `expr1 op expr2 --> value1 ^ value2 --> string1 ^ string2 --> "CS " ^ "3110" --> "CS 3110"`
- Alternatively: `expr1 op expr2 --> "CS 3110"`
- First version: *small-step semantics*; second version: *big-step semantics* (`-->` is approximately "evaluates to")
- `OCaml utop` displays big-step semantics, underlying evaluation is done with small-step semantics (we use small-step semantics exclusively in CS 3110 when we show the computations)
- Challenge: Tracing expression back up the *parse tree* to get definition (i.e `"CS 3110" <-- expr1 op expr2`)
- Technically the evaluation above traces an *abstract-syntax tree*, doesn't show everything: `token`/`digit`, etc.
- We will discuss lexing, parsing, and grammar trees later (if not, see CS 2112/CS 4120 notes)
- `OCaml`'s execution model is based on evaluation reduction (i.e. small-step semantics)
- We already discussed evluation of most commands/primitives and what types they give, except for **`let`**-bindings
- Consider generalized snippet of `OCaml` code: `let x = expr1:t1 in expr2:t2` (types shown explicitly here)
- From now on, for simplification, we will use `t=type`, `d=decl`, `v=value`, `e=expr` (all others explicitly named)
- Easier to read: `let x = e1:t1 in e2:t2` (carries same meaning, simply substitute aforementioned names)
  - Type-checking: this entire expression evaluates to the type `t2` (recall the main rule from recitation)
  - Evaluation: Multi-step process (see outline below for a thorough analysis that covers the tricky parts)
    1. Evaluate `e1 --> v1` (to a terminal value)
    2. Substitute `v1` for `e1` in `e1` and name the result `e2'`
    3. Evaluate `e2' --> v` (to another terminal value)
    4. Result: "return" `v` (in `OCaml`, `return` call is optional: result of expression always returned)
    5. Note: Type of expression is type of `v`, which is type of `e2'`, which is type of `e2`, which is `t2`
  - Chaining the process above helps evaluate complex sub-expressioned expressions (`e2' --> e3 --> e3'`)
- Bad idiom: Multiple variable bindings of the same name ("shadowing")
```
let x = 6 in
    ((let x = 5 in x) + x)
```
- This becomes: `let x = 6 in (5 + x) --> (5 + 6) --> 11` (we'll talk about this in a few lectures)

Recitation 2

- Need to cover some minor features we may have missed, some details on evaluation and substitution model
- Note for below: we use `def ::= (* definition *)` to make a definition, `def := <type>` to specify its type
- Some infix binary operators we missed:
  1. Exponentiation: `** := float -> float -> float` (can customize a function to extend to `int`s as well)
  2. Logical `and`: `&& := boolean -> boolean -> boolean`
     - In `a && b`, if `a` is `false`, then `a && b` short-circuits to `false`
  3. Logical `or`: `|| := boolean -> boolean -> boolean`
     - In `c || d`, if `c` is `true`, then `c || d` short-circuits to `true`
  4. Bitwise `and`: `land := int -> int -> int`
  5. Bitwise `or`: `lor := int -> int -> int`
  6. Bitwise `xor`: `lxor := int -> int -> int`
  7. Bitwise `not`: `lnot := int -> int`
     - This is included for completion; it is actually a unary prefix operator
  8. Logical left shift: `lsl := int -> int -> int`
     - The arithmetic left shift is the same as the logical one (see CS 3410 notes for more on digital logic)
  9. Logical right shift: `lsr := int -> int -> int`
  10. Arithmetic right shift: `asr := int -> int -> int`
- Distinction between *true equality* and *Leibniz equality*: Former refers to same mathematical object while latter refers to structural equality, i.e. two objects are indiscernable property-wise but are distinct in allocation
- In practice: true equality is equality of memory address, while structural equailty is equality of properties/fields
- OCaml: There are separate equal/unequal operators for checking true vs. structural equality (Java-esque)
  - True equality: `==` (equal), `!=` (unequal)
  - Structural equality: `=` (equal), `<>` (unequal)
- The other thing we missed were some simple floating-point mathematical functions that make life easier:
  1. Absolute value: `abs := float -> float`
  2. Arc-cosine: `acos := float -> float`
  3. Arc-sine: `asin := float -> float`
  4. Arc-tangent: `atan := float -> float`
  5. Cosine: `cos := float -> float`
  6. Hyperbolic cosine: `cosh := float -> float`
  7. Exponential (e): `exp := float -> float`
  8. Natural logarithm: `log := float -> float`
  9. Common logarithm: `log10 := float -> float`
  10. Sine: `sin := float -> float`
  11. Hyperbolic sine: `sinh := float -> float`
  12. Square root: `sqrt := float -> float`
  13. Tangent: `tan := float -> float`
  14. Hyperbolic tangent: `tanh := float -> float`
- Challenge: Use math skills to derive all other needed functions and implement them into your own library
- Now to explore the *substitution model* we've been using all along in our evaluations

- Quick notation simplification before we start using it: `e{v/x}` means "evaluate `e` with `v` plugged in for `x`", or more precisely "`let x = v in e`" (this is the better intuition for evaluation)
- A quick example of how we evaluated using small-step semantics in our substitution model:
  ```
  let x = 3 in (x + x) --> (3 + 3) --> 6
  ```
- There are generalized rules we must use, in case expressions themselves define new variables:
  1. `n{v'/x} ::= n`
  2. `x{v/x} ::= v`
  3. `y{v/x} ::= y` if `y` is not equal to `x` (equality becomes Case 2)
  4. `(e1 op e2) {v/x} ::= e1{v/x} op e2{v/x}`
  5. `(let x = e1 in e2) {v/x} ::= let x = e1{v/x} in e2`
  6. `(let y = e1 in e2) {v/x} ::= let y = e1{v/x} in e2{v/x}` if `x` is not equal to `y`
- Let's move onto OCaml evaluation rules; recall that every OCaml program is an expression:
  1. Double expression operated reduction:
     ```
     e1 op e2 --> e1' op e2
        if e1 --> e1'
     ```
  2. Single expression operated reduction:
     ```
     v1 op e2 --> v1 op e2'
        if e2 --> e2'
     ```
  3. Numerical operated reduction:
     ```
     n1 op n2 --> n3
            where n3 is the binop result of n1 and n2
     ```
  4. Triple expression bound reduction:
     ```
     if e1 then e2 else e3 --> if e1' then e2 else e3
                     if e1 --> e1'
     ```
  5. True branched reduction:
     ```
     if true  then e2 else e3 --> e2
     ```
  6. False branched reduction:
     ```
     if false then e2 else e3 --> e3
     ```
  7. Double expression bound reduction:
     ```
     let x = e1 in e2 --> let x = e1' in e2
                if e1 --> e1'
     ```
  8. Double expression reduction:
     ```
     e1 e2 --> e1' e2
     if e1 --> e1'
     ```
  9. Single expression reduction:
     ```
      v e2 --> v e2'
     if e2 --> e2'
     ```
- It may seem like there are other ones, but they can mostly be derived from these with the substitution model and expression-level chaining
- Once the entire program is fully-reduced to a terminal value, execution is completed (and the resulting displayed type on the toplevel, or REPL, may very well be extremely compounded and complex)
- We need to extend our HBNF definitions table now, since we've added a lot of extra terms (seems like it will never be complete)
- Will modify the HBNF definitions list after next lecture, once we cover more (i.e. alternative declarations, functions, etc.)
- Need to remember key point: Substitution model and evaluation rules are used in conjunction to analyze and reduce a program to a final value with a clear, discernable type (note, again, that this STILL allows for generic types, which have been determined to be such)

Lecture 3

- Need to develop more on the theory of functions (e.g. functions as values, functions as data, etc.)