Midterm Review!

- Variable Bindings, Shadowing, Let Expressions
- Boolean, Comparison and Arithmetic Operations
  - Equality Types
- Types, Datatypes, Type synonyms
  - Tuples, Records and Lists
- Case statement, Pattern Matching
- Functions, Anonymous Functions, Higher Order Functions
  - Actually Taking in Tuples, Function Closures
  - Tail Recursion
  - Currying
  - Filter, Map, Fold
Midterm Review!

• Lexical Scope vs Dynamic Scope
• Type Inference, Polymorphic Types and Type Generality
• Modules
• Equivalence
Variable Bindings

• SML evaluation creates bindings in the environments (static and dynamic) rather than change values store in variables.

• Repeated Variable names?
  • Shadowing

• Let Expression allows us to create bindings in a smaller Scope
Boolean, Comparison and Arithmetic Operations

• Boolean Operators
  • `andalso`, `orelse` evaluates for booleans only, they are not functions (you cannot do partial evaluation with them)
• `not` is a function
  • `- op not;
  • `val it = fn : bool -> bool`
  • `- List.map not [true, true, false];`
  • `val it = [false,false,true] : bool list`
Boolean, Comparison and Arithmetic Operations

• Comparison and Arithmetic Operators
  • =, <>, equality types
  • >, <, >=, <=, +, -, *, must take the same type on both sides
  • ‘div’ for integers, ‘/’ for reals
  • You cannot divide on integer by a real or vice versa
  • Because these operators are all functions!
Types, Datatypes, Type synonyms

• Built-in types
  • String, int, real, bool, records, lists
  • What about tuples?
    • They are just syntactic sugar for records

• `datatype` keyword
  • Allows you to create types by yourself
  • “one of type” and recursive type

• `type` keyword
  • “each of type”, just renaming the existing types
Case statement, Pattern Matching

\[
\text{case } e0 \text{ of } \\
\quad p1 \Rightarrow e1 \\
\quad | \quad p2 \Rightarrow e2 \\
\quad | \quad \ldots \\
\quad | \quad pn \Rightarrow en
\]

• Values and variables form patterns
• SML is essentially creating variable bindings of the variable with the actual value in \( e0 \).
• It is not checking if the value stored in the variable equals to what’s in the current environment
Functions, Anonymous Functions, Higher Order Functions

• Functions actually takes in a **pattern**, for example, 
  
  \((x : \text{int}, y : \text{bool})\).

• By pattern matching, it creates bindings of variables and values. Then the environment is **bound**

• The **bounded environment** along with the **code** creates **function closure**.
Functions, Anonymous Functions, Higher Order Functions

• Anonymous Functions use keyword `fn` rather than `fun`, which cannot be recursive

• Tail Recursion
  • You are not doing any more operation after getting returned value from your recursive call
Functions, Anonymous Functions, Higher Order Functions

• Currying is taking a function with “several arguments” and make it into nested functions, which takes *one argument at a time*

• Partial evaluation: since curried functions are just nested functions, we can pass in one argument at a time *in order*

• We can take in functions as arguments
  • Higher order functions are just those functions that return or take in functions
Functions, Anonymous Functions, Higher Order Functions

• Classic higher order functions

  • List.filter
  • List.map
  • List.foldl
  • List.foldr

• What do they do?
Lexical Scope vs Dynamic Scope

• Lexical scope: use environment where function is defined
• Our Function Closure so far is in lexical scope
• Dynamic scope: use environment where function is called
Type Inference, Polymorphic Types and Type Generality

• Polymorphic types means it can be any type

• So `a list * `a list -> `a list is more general than int list * int list -> int list

• But not more general than

  int list * string list -> int list

• Polymorphic type can be any type

• More general means you can substitute one type by another \textit{consistently}
Modules

• You can hide a function by using signatures

structure MyModule = struct bindings end

signature SIGNAME =
sig types-for-bindings end

structure MyModule => SIGNAME =
struct bindings end
Modules

• Remember from lecture you can ensure constraints on values

```plaintext
structure Rational3 =
  struct
  type rational = int * int
exception BadFrac

fun make_frac (x,y) = ...
fun Whole i = (i,1) (* needed for RATIONAL_C *)
fun add ((a,b)(c,d)) = (a*d+b*c,b*d)
fun toString r = ... (* reduce at last minute *)
end
```
Equivalence

• Given equivalent arguments, two equivalent pieces of code:
  • Produce equivalent results
  • Have the same (non-)termination behavior
  • Mutate (non-local) memory in the same way
  • Do the same input/output
  • Raise the same exceptions

• Look for function closures, dynamic and static environments and side effects like print
Good Luck!