Today’s Agenda

• Testing
• Lists, Let-Expression (Review)
• Options
• Type synonyms
• Type generality
• Equality types
• Syntactic sugar

Reminder: Check out the CSE341 style guide as you work on HW!
Also check out the style guides in section 1 slide!
Testing

• You should still test your code!
• We will assign points to your testing file
• Just do something like this:

```
val test1 = ((4 div 4) = 1);
```

“Is expected output = actual output”
Section Learning Objectives

- Review building/accessing new types (e.g. datatypes)
- Recognize type synonyms as “convenient” feature
- Be able to generalize specific types with polymorphism (e.g. int list into ‘a list) and equality types
- Practice using pattern-matching with case expressions
Lists

- Lots of new types: For any type t, the type t list describes lists where all elements have type t
  - Examples: int list, bool list, int list list, (int * int) list, (int list * int) list
- So [] can have type t list for any type t
  - SML uses type 'a list to indicate this ("tick a" or "alpha")
- For e1 : e2 to type-check, we need a t such that e1 has type t and e2 has type t list. Then the result type is t list
  - null: 'a list -> bool
  - hd: 'a list -> 'a
  - tl: 'a list -> 'a list
Let-Expression

- Syntax: `let b1 b2 ... bn in e end`
  - Each `bi` is any binding and `e` is any expression
  - Type-checking: Type-check each `bi` and `e` in a static environment that includes the previous bindings.
  - Type of whole let-expression is the type of `e`.
  - Evaluation: Evaluate each `bi` and `e` in a dynamic environment that includes the previous bindings.

Result of whole let-expression is result of evaluating `e`.
**Options**

\( t \) option is a type for any type \( t \)
- (much like \( t \) list, but a different type, not a list)

Building:
- **NONE** has type \( 'a \) option (much like [] has type 'a list)
- **SOME e** has type \( t \) option if \( e \) has type \( t \) (much like \( e::[] \))

Accessing:
- **isSome** has type \( 'a \) option \( \rightarrow \) bool
- **valOf** has type \( 'a \) option \( \rightarrow \) 'a (exception if given NONE)
Type Synonyms

• What does \texttt{int * int * int} represent?
• In HW1 we called it a date
• Wouldn’t it be nice to reflect this representation in the source code itself?

\texttt{type date} = \texttt{int * int * int}
Datatypes

• What if we want something \textbf{unique}? A \textbf{new} type?
• We can’t just use type synonyms because they can only be built from existing types.
• \textbf{Datatypes} give us the ability to define \textbf{custom types}.

\begin{verbatim}
datatype foo = bar | baz of int | qux of bool
\end{verbatim}
**type VS datatype**

- **datatype** introduces a new type name, distinct from all existing types

```plaintext
datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace
    | Num of int
```

- **type** is just another name

```plaintext
type card = suit * rank
```
Type Synonyms

Why?

• For now, just for convenience
• It doesn’t let us do anything new

Later in the course we will see another use related to modularity.
Type Generality

Write a function that appends two string lists...
Type Generality

• We would expect
  
  \[\text{string list} \times \text{string list} \rightarrow \text{string list}\]

• But the type checker found
  
  \[\text{\textsc{'a} list} \times \text{\textsc{'a} list} \rightarrow \text{\textsc{'a} list}\]

• \textsc{'a} are called Polymorphic Types
• Why is this OK?
More General Types

• The type

\[\text{'a list} \times \text{'a list} \to \text{'a list}\]

is more general than the type

\[\text{string list} \times \text{string list} \to \text{string list}\]

and “can be used” as any less general type, such as

\[\text{int list} \times \text{int list} \to \text{int list}\]

• But it is not more general than the type

\[\text{int list} \times \text{string list} \to \text{int list}\]
The Type Generality Rule

The “more general” rule

A type $t_1$ is more general than the type $t_2$ if you can take $t_1$, replace its type variables consistently, and get $t_2$

What does consistently mean?
Equality Types

Write a list “contains” function...
Equality Types

• The double quoted variable arises from use of the = operator
  • We can use = on most types like int, bool, string, tuples (that contain only “equality types”)
  • Functions and real are not “equality types”
• Generality rules work the same, except substitution must be some type which can be compared with =
• You can ignore warnings about “calling polyEqual”
More Syntactic Sugar

• Tuples are just records

• If-then-else is implemented as syntactic sugar for a case statement
If-then-else

• We’ve just covered case statements
• How could we implement if-then-else

```plaintext
case x of
  true => "apple"
| false => "banana"
```

```plaintext
if x then "apple" else "banana"
```
val-Pattern Matching

Remember our unit test?

(* Neat trick for creating hard-fail tests: *)

val true = ((4 div 4) = 1);

Just a pattern match!

“Match the left hand side against the value ‘template’ true, binding any variables (there aren’t any)”
Adventures in pattern matching

• Shape example
• Function-pattern syntax if we get to it