Basics

- You should be familiar with set notation, relations, and sequences.
- Ordered sets:
  - A binary relation $R$ on a set $S$ is:
    - reflexive if $(s,s) \in R$
    - symmetric if $(s,t) \in R \land (t,s) \in R$
    - transitive if $(s,t), (t,u) \in R \land (s,u) \in R$
    - antisymmetric if $(s,t), (t,s) \in R \land s = t$
  - A reflexive and transitive relation on $S$ is a called a preorder on $S$
  - A reflexive, transitive, antisymmetric relation on $S$ is called a partial order ($\sqsubseteq$) on $S$
    - A partial order $\sqsubseteq$ on a set $S$ is a total order if for each $s,t \in S$, either $s \sqsubseteq t$ or $t \sqsubseteq s$.
  - A reflexive, transitive, symmetric relation on $S$ is called an equivalence relation on $S$. 
Basics

- Suppose \( \sqsubseteq \) is a partial order on \( S \) and \( s,t \in S \).
  - \( j = s \sqcup t \in S \) is the join or least upper bound of \( s \) and \( t \) if
  - \( s \sqsubseteq j \) and \( t \sqsubseteq j \)
    - \( \forall k \in S \) s.t. \( s \sqsubseteq k \) and \( t \sqsubseteq k \), \( j \sqsubseteq k \).
  - \( m = s \sqcap t \in S \) is the meet or greatest lower bound of \( s \) and \( t \) if
    - \( m \sqsubseteq s \) and \( m \sqsubseteq t \)
    - \( \forall n \in S \) s.t. \( n \sqsubseteq s \) and \( n \sqsubseteq t \), \( n \sqsubseteq m \)
- Exercise: Give an example of a partial and equivalence order.
Basics

- Suppose $R$ is a binary relation on $S$.
  - The reflexive closure of $R$ is the smallest reflexive relation $R'$ that contains $R$.
  - The transitive closure of $R$ is the smallest transitive relation $R'$ that contains $R$.

- Let $R$ be a binary relation on $S$. Define $R'$ as:
  \[ R' = R \cup \{(s,s) \mid s \in S\} \]
  - Show that $R'$ is the reflexive closure of $R$.
    - Need to show that $R'$ is a reflexive relation on $R$.
    - Need to show that $R'$ is the smallest such relation.
Basics

- Let $S$ have preorder $\sqsubseteq$. We say $\sqsubseteq$ is well-founded if it contains no infinite decreasing chains. A preorder is a relation that is reflexive and transitive.
  - The preorder defining the natural numbers is well-founded.
  - The preorder on the integers is not.
Induction

- Principle of ordinary induction on natural numbers:
  - Suppose that $P$ is a predicate on $\mathbb{N}$. Then if $P(0)$ holds, and for all $i$, $P(i) \to P(i+1)$,
  - $P(n)$ holds for all $n$.

- Example:
  - Theorem: $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$ for all $n$. 
Proof

By induction on n.

- Base case (n = 0):
  \[2^0 = 2^1 - 1\]

- Inductive case (n = i + 1):
  \[2^0 + 2^1 + \ldots + 2^{i+1} = (2^0 + 2^1 + \ldots + 2^i) + 2^{i+1}\]
  \[2^{i+1} - 1 + 2^{i+1} \quad \text{(induction hypothesis)}\]
  \[2 \times 2^{i+1} - 1 = 2^{i+2} - 1\]
Many of the examples in the text use functional programming notation

- Programs expressed in OCaml
  - The examples in this lecture are given in SML which is closely related to OCaml
- It is sufficient to just consider the “core” language
  - Don’t need to understand objects or modules

Functional programming
- immutable data structures
- recursion and functional call as the primary control structure
- heavy use of higher-order functions
- Contrast with imperative languages
Expressions

- Programs are expressions. The meaning of a program is the value of the expression:

  # 16 + 18 ;
  val it = 34 : int

  # 2 * 8 + 3 * 6;
  val it = 34 : int
Names

Introduce bindings using let:

```ocaml
# let val v = 200 in v + 1 end;
val it = 201 : int

# let val y = it * 3 in y end;
val it = 603 : int
```

Bindings cannot refer to themselves in definitions:

```ocaml
# let val z = z + 1 in z end;;  (* illegal *)
```
Functions

# val double = fn (x:int) => x + x
val double = fn: int -> int

# double 3;
val it = 6: int

- x is the parameter of the function. The expression x * x is the body.
- The expression “double 3” is the application of the function
- The type of the function is “int -> int”
Functions

Functions with multiple arguments:

```
# val sumsquare = fn x => fn y => x*x + y*y
- val sumsquare = fn: int -> int -> int
# sumsquare 3 4;
- val it = 25 : int
# sumsquare 3;
- val it = fn: int -> int
# fun sumsquare x y = x * x + y * y
- val sumsquare = fn: int -> int -> int
# fun sumsquare (x,y) = x * x + y * y
- val sumsquare = fn: int * int -> int
```
Booleans and Conditionals

There are only two values of type Boolean: true and false

# 1 = 2;
val it = false : bool

# not (5 <= 10);
val it = false : bool

Conditional expressions defined as usual:

# if 3 < 4 then 7 else 100;
val it = 7 : int

# if false then (3 + 3) else 10;
val it = 10 : int

# if true then 10 else false;
Error: types of rules don’t agree
Inductive and Recursive Definitions

```ml
# fun sum 0 = 0
    | sum n = n + sum(n-1)
- val sum = fn: int -> int
# sum 3;
- val it = 6 : int

# fun sum(n) = if n = 0
    then 0
    else n + sum(n-1)
- val sum = fn: int -> int
# sum 3;
- val it = 6 : int
```
Lists

- Lists store a collection of homogeneous data values.
- The empty list is written as [ ]; hd and tl used to deconstruct lists

```plaintext
# [1,2,3];
- val it = [1,2,3] : int list
# let val l = [1,2,3] in [hd(l),hd(tl(l))] end;
- val it = [1,2] : int list
# let val l = [1,2,3] in [hd(l),tl(l)] end;
- error
```

- Lists can be built from any of type
- Can build list of lists:

```
# [ [1,2], [1,2,3] ]
- val it = [ [1,2], [1,2,3]] : int list list
```
Lists

- **Can construct lists using cons:**

```plaintext
# 1 :: [2,3]
- val it = [1,2,3] : int list
```

- **Functions to build lists:**

```plaintext
# fun map(f, []: int list) = []
  | map(f:int->int, x::l) = f(x)::map(f,l)
- val map = fn: (int -> int) * int list -> int list
# map(fn x => x + 1, [1,2,3])
- val it = [2,3,4]: int list
```

- **What would happen if we removed the type declarations on f and []?**
Example

Reverse a list:

```haskell
# fun snoc(x:int,y) = if y = []
  then [x]
  els hd(y)::snoc(x,tl(y))
- val snoc = fn: int * int list -> int list

# fun reverse(l: int list) =
  if l = []
    then []
    else snoc( hd(l), reverse(tl l))
```

Why is this inefficient?
Example

```ocaml
# fun reverse l: int list =
    let fun aux([], result) = result
    | aux(l',result) =
        aux(tl(l'),hd(l')::result)
    in aux l []
end

- val reverse = fn: int list -> int list

Why is this better?
```
Tail Recursion

Express loops using tail recursion: A tail recursive function is one in which the result of every control-path is defined in terms of another function call.

```ml
# fun fact 0 = 1
    | fact n = n * fact(n-1)
This is not in tail form.
# fun fact n =
    let fun aux(0,r) = r
        | aux(n,r) = aux(n-1,r*n)
    in aux(n,1)
end
This is in tail form. Why is this better?
```
Type Inference & Polymorphism

```haskell
# fun fact 0 = 1
| fact n = n * fact (n - 1)
- val fact = fn: int -> int

Compiler deduces that n must be an integer, and that fact is a function
over integers

fun map(f,[]) = []
| map(f, x::l) = f(x)::map(f,l)
- val map = fn: ('a -> 'b) * 'a list -> 'b list

'a and 'b are type variables

The inferred type tells us that f operates over something of type 'a and
returns something of type 'b, and that l is a list of type 'a and map
returns a list of type 'b
```
Data Types

- Can create new datatypes
- A powerful programming tool that is useful in preventing errors
  - Example: we have circles and squares, both represented in terms of coordinates:
    - circles: center and radius
    - square: bottom left corner and width
    - both represented as a triple of floats
    - how do we prevent accidentally mistaking a square for a circle?
Datatypes

# datatype shape =
    Circle of real * real * real |
    Square of real * real * real

Creates two constructors of type shape named Circle and Square

# Square(1.0,2.0,3.0);
- val it = Square(1.0,2.0,3.0) : shape
Datatypes

Use pattern matching to extract elements of a datatype:

# fun areaOfSquare s =
    case s of
        Square(_,_,z) => z * z
    | _ => raise Fail "not a square"
# fun areaOfSquare(Square(_,_,z)) = z * z

Constructors behave as both patterns and functions.
Identified by being capitalized.
Datatypes

# fun areaOf s =
    (case s of
        Square(_,_,z) => z * z
    | Circle(x,y,r) = 3.14 * r*r)
- val areaOf = fn: shape -> real
Recursive types

Consider the language of arithmetic expressions:

\[ \text{exp} :: \text{number} \]

\[ \mid (\text{exp} + \text{exp}) \]

\[ \mid (\text{exp} - \text{exp}) \]

\[ \mid (\text{exp} \times \text{exp}) \]
Recursive types

- We can translate this grammar directly into a datatype definition

```ml
datatype ast = Num of int
             | Plus of ast * ast
             | Minus of ast * ast
             | Times of ast * ast
```

Like the grammar, this datatype is recursive.

Surface details about the syntax (e.g., brackets) have been omitted.

Exercise: write an ML function to evaluate the language of simple arithmetic expressions:

defunction eval : ast -> int
Reading

- Read Chapter 1 and 2 of TAPL
- Homework:
  - 2.2.6
  - 2.2.7
- Next time:
  - Untyped arithmetic expressions
  - Introduction to operational semantics