Where are we?

- We have an informal idea of the semantics of uScheme
- Plan: study the formal semantics and an interpreter
- But first we need a reading knowledge of Standard ML

Overview of ML

- Like Scheme:
  - A functional language with imperative features
  - First-class functions, recursion
  - Assignment, loops, sequencing
- Unlike Scheme: many things, including
  - Static type checking and type inference
  - Polymorphic type system

Running it

- Put it on your path, and

```plaintext
> mosml
Moscow ML version 2.01 (January 2004)
Enter `quit();' to quit.
- load "List.uo";
  > val it = () : unit
  - open List;
  > datatype 'a list =
     ('a list,{con 'a nil : 'a list, con 'a :: : 'a * 'a list -> 'a list})
  val 'a tabulate = fn : int * (int -> 'a) -> 'a list
  val 'a rev = fn : 'a list -> 'a list
...
```
Basics of ML interaction

- Sample interaction
  
  mosml
  Moscow ML version 2.01 (January 2004)
  Enter `quit();' to quit.
  - hd [1,2,3];
    > val it = 1 : int
  - tl [1,2,3];
    > val it = [2, 3] : int list

Novel features

- Things to notice
  - hd [1,2,3];
    > val it = 1 : int
  - Function application is notated by juxtaposition
    f x
  - The compiler (yes!) infers the type and prints it
  - The special identifier it is bound to the most recent result

Built-in types

- Integers, reals, strings, tuples, lists, records
  1, ~1, .5, "hello", (1,2,3), [1,2,3], 1::2::3::nil,
  {color="red",length=5}

Patterns

- Used to take apart compound values
  - val record = {color="red",length=5};
    > val record = {color = "red", length = 5} : {color : string, length : int}
  - val {color=x, length=y} = record;
    > val x = "red" : string
    val y = 5 : int
  - val {length = y, color=x} = record;
    > val y = 5 : int
    val x = "red" : string
More patterns

- Taking apart a list
  
  - val xs = ["heads", "or", "tails"];  
  > val xs = ["heads", "or", "tails"] : string list
  
  - val y::ys = xs;  
  > val y = "heads" : string  
  val ys = ["or", "tails"] : string list

Wild cards

- If we don't care about part of the structure
  
  - val _::ys = xs;  
  > val ys = ["or", "tails"] : string list

Deep patterns

- We can match "into" a structure:
  
  - val x = ([1,2],"hi");  
  > val x = ([1, 2], "hi") : int list*
    string
  - val (y::ys,s) = x;  
  > val y = 1 : int  
    val ys = [2] : int list  
    val s = "hi" : string

Layered patterns

- ...and we can bind identifiers to parts of a structure:
  
  - val x = ([1,2],"hi");  
  - val (l as y::ys, s) = x;  
  > val l = [1, 2] : int list  
    val y = 1 : int  
    val ys = [2] : int list  
    val s = "hi" : string
Defining functions

- Keyword "fun"

```haskell
fun fact x = 
  if x=0 then 1 else x*fact (x-1);
> val fact = fn : int -> int
```

- Binds identifier just as `val` does
- Note the function type!

Defining functions by pattern matching

- Our old friend "append"

```haskell
- fun append (nil,l) = l 
    | append (x::xs, l) = ...
> val 'a append = fn :
     'a list * 'a list -> 'a list
- append ([1,2],[3,4,5]);
> val it = [1, 2, 3, 4, 5] : int list
```

Parametric polymorphism

- What's that 'a thing?

  'a list * 'a list -> 'a list
- A type variable (often pronounced "alpha")
- Our append function is **polymorphic**.

  ```haskell
  - append ("a","b","c"],["d","e"]);
  > val it = ["a", "b", "c", "d", "e"] : string list
  ```

Beware!

- "Parametric" means (roughly) that we must be able to substitute some type for 'a everywhere:

  ```haskell
  - append ("a","b","c"],[1,2]);
  ! Toplevel input:
  ! append ("a","b","c"],[1,2]);
  ! Type clash: expression of type
  !   int
  ! cannot have type
  !   string
  ```
Local functions

- Much like Scheme

```ocaml
fun reverse xs = 
  let fun revapp (nil,zs) = zs
      | revapp (y::ys,zs) = 
        revapp(ys,y::zs)
  in 
  revapp(xs,nil)
end;
> val 'a reverse = fn : 'a list -> 'a list
- reverse [1,2,3];
> val it = [3, 2, 1] : int list
```

Higher-order functions

- You've seen it all before, but with different syntax
- Example:

```ocaml
- fun curry f = fn x => fn y => f(x,y);
  > val ('a, 'b, 'c) curry = fn : ('a * 'b -> 'c)
      -> 'a -> 'b -> 'c
  > val pos = curry op < 0;
  > val pos = fn : int -> bool
  > filter pos [1, ~1, ~2, 3];
  > val it = [1, 3] : int list
```

Defining data types

- Enumeration types

```ocaml
- datatype color = Yellow | Magenta | Cyan;
  > New type names: =color
datatype color =
  (color,{con Cyan : color, con Magenta : color, con Yellow : color})
  con Cyan = Cyan : color
  con Magenta = Magenta : color
  con Yellow = Yellow : color
```
- The "con"s are constructors, or in this case, since they have no parameters, constants

Parameters of constructors

```ocaml
- datatype money = nomoney | Coin of int | Bill of int;
  > New type names: =money
datatype money =
  (money,
   {con Bill : int -> money, con Coin : int -> money, 
   con nomoney : money})
  con Bill = fn : int -> money
  con Coin = fn : int -> money
  con nomoney = nomoney : money
  > val dime = Coin 10;
  > val dime = Coin 10 : money
  > val dollar = Bill 1;
  > val dollar = Bill 1 : money
  > val it = nomoney : money
```
Recursive datatypes

- datatype 'a bintree = Leaf of 'a
  | Tree of 'a * 'a bintree * 'a bintree;

> New type names: =bintree

datatype 'a bintree =
  (con 'a Leaf : 'a -> 'a bintree
   con 'a Tree : 'a * 'a bintree * 'a bintree -> 'a bintree)

con 'a Leaf = fn : 'a -> 'a bintree
con 'a Tree = fn : 'a * 'a bintree * 'a bintree -> 'a bintree

val t = Tree(5,Tree(2,Leaf 1,Leaf 4),Tree(9,
  Tree(7,Leaf 6,Leaf 8),Leaf 10));

Functions over datatypes

fun inord (Leaf x) = [x]
  | inord (Tree(x,left,right)) =
      (inord left)@x::(inord right);

> val 'a inord = fn : 'a bintree -> 'a list

- inord t;
> val it = [1, 2, 4, 5, 6, 7, 8, 9, 10] : int list

Note the infix "append" notation @
Abstract syntax: top level

datatype toplevel = EXP of exp  
| DEFINE of name * lambda  
| VAL of name * exp  
| USE of name

Abstract syntax: expressions

datatype exp = LITERAL of value
  | VAR of name  
  | SET of name * exp  
  | IFX of exp * exp * exp  
  | WHILEX of exp * exp  
  | BEGIN of exp list  
  | LETX of let_kind *  
  | (name * exp) list * exp  
  | LAMBDA of lambda  
  | APPLY of exp * exp list

and let_kind = LET | LETREC | LETSTAR
Abstract syntax: values

and value =
  NIL
  | BOOL of bool
  | NUM of int
  | SYM of name
  | PAIR of value * value
  | CLOSURE of lambda * value ref env
  | PRIMITIVE of primitive

withtype primitive = value list -> value
and lambda = name list * exp

Semantics of uScheme

- Top-level judgment:
  \( \langle t, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle \)
- Expression evaluation judgment:
  \( \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \)

What a rule means

- Operationally we read a rule as having inputs, possibly some subgoals, and outputs
- Inputs: initial state of abstract machine
- Subgoals: what the machine must do
- Outputs: final state of abstract machine
- Note that metavariables x and x' and x₁ are all different!

Variables and assignment

- Variable lookup
  \[ x \in \text{dom} \rho \quad \rho(x) \in \text{dom} \sigma \]
  \[ \langle \text{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle \]

- Assignment
  \[ x \in \text{dom} \rho \quad \rho(x) = l \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \]
  \[ \langle \text{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'[l \mapsto v] \rangle \]
Let-binding

- Simultaneous binding

\[
\begin{aligned}
  l_1, \ldots, l_n \notin \text{dom } \sigma \\
  \langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\
  \vdots \\
  \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\
  \langle e, \rho(x_1 \mapsto l_1, \ldots, x_n \mapsto l_n), \sigma_1(l_1 \mapsto v_1, \ldots, l_n \mapsto v_n) \rangle \Downarrow \langle v, \sigma' \rangle \\
  \langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle
\end{aligned}
\]

Let* binding

- Sequential binding

\[
\begin{aligned}
  l_1, \ldots, l_n \notin \text{dom } \sigma \\
  \langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \\
  \rho_1 = \rho[x_1 \mapsto l_1] \\
  \sigma_1 = \sigma'[l_1 \mapsto v_1] \\
  \vdots \\
  \langle e_n, \rho_{n-1}, \sigma'_{n-1} \rangle \Downarrow \langle v_n, \sigma'_{n-1} \rangle \\
  \rho_n = \rho_{n-1}[x_n \mapsto l_n] \\
  \sigma_n = \sigma'_{n-1}[l_n \mapsto v_n] \\
  \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle \\
  \langle \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle
\end{aligned}
\]

Letrec binding

- Left as an exercise for the reader

Functions

- Lambdas evaluate to closures

\[
\begin{aligned}
  x_1, \ldots, x_n \text{ all distinct} \\
  \langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \{\text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \}, \sigma \rangle
\end{aligned}
\]
Functions

- Function applications

\[
\begin{align*}
& l_1, \ldots, l_n \notin \text{dom } \sigma \\
& \langle e, \rho, \sigma \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle, \sigma_0 \rangle \\
& \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\
& \vdots \\
& \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\
& \langle e, \rho \{ x_1 \mapsto l_1, \ldots, x_n \mapsto l_n \}, \sigma \{ l_1 \mapsto v_1, \ldots, l_n \mapsto v_n \} \rangle \Downarrow \langle v, \sigma' \rangle \\
& \text{APPLY}(e, e_1, \ldots, e_n, \rho, \sigma) \Downarrow \langle v, \sigma' \rangle
\end{align*}
\]

Global variables

- Note the two rules:
  - one for the case where a global is bound already (in which case we do an assignment),
  - one for the case where the global is new (in which case we extend the environment)

Impcore-like

- Literals
- Control flow
- Primitives

Top-level functions

- "define" is just syntactic sugar for a val binding to a lambda expression
An interpreter in ML

Environment and store

- ML types for generic environments
  ```ml
  type name = string
  type 'a env = (name * 'a) list
  ```
- Instantiate to get a mapping from names to locations:
  ```ml
  value ref env
  ```
- We're using the ML store to represent the uScheme store!

Structure of interpreter

- Create initial environment binding the primitives and initial basis
- Enter a read-eval-print loop
  ```ml
  repeatedly read and eval top-level item
  ```
- Evaluation code is structured just like the operational semantics

Assignment

- Read Ramsey & Kamin, Chapter 5
- Do exercise 5.9 on page 196. Note that this requires a significant (in the sense of understanding, not lines of code!) change to the interpreter; also note that you must answer the question as well as implementing the change and testing it thoroughly