Recap

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   - Of the same “type” found in step 2

Expressions (Syntax)  Values (Semantics)

Types

Expressions (Syntax)

<table>
<thead>
<tr>
<th>Compile-time “Static”</th>
<th>Types</th>
</tr>
</thead>
</table>

Values (Semantics)

Types

If-then-else expressions

if (1 < 2) then 5 else 10

• then-subexp, else-subexp must have same type!
  - ...which is the type of resulting expression

if (1 < 2) then [1;2] else [5]

• Then-subexp, Else-subexp must have same type!
  - Equals type of resulting expression

if (1 < 2) then ["ab","cd"] else ["a"]

• Tuples, Records: #i
  - Fixed number of values, of different types
• Lists: ::, @, hd, tl, null
  - Unbounded number of values, of same type

If-then-else expressions

if (1 < 2) then 5 else 10
if (1 < 2) then ["ab","cd"] else ["a"]

if (1 < 2) then [1;2] else [5]

if false then [1;2] else [5]
**Style Exercise**

Java

```java
int foo(i, b, c, d) {
    if (b) {
        i++;
    }
    if (c) {
        return i + 2;
    }
    if (d) {
        i = i + 3;
    }
    else {
        return i + 4;
    }
    return i;
}
```

OCaml

```ocaml
let i = 0 in
if b then i <- i + 1
if c then i <- i + 2
if d then i <- i + 3
else i <- i + 4
i
```

**Next: Variables**

**Variables and Bindings**

Q: How to use variables in ML?

Q: How to “assign” to a variable?

```ocaml
# let x = 2+2;;
val x : int = 4

let x = e;;
“Bind the value of expression e to the variable x”
```

**Variables and Bindings**

```ocaml
# let x = 2+2;;
val x : int = 4

# let y = x * x * x;;
val y : int = 64

# let z = [x;y;x+y];;
val z : int list = [4;64;68]
```

Later declared expressions can use `x`
- **Most recent** “bound” value used for evaluation

Sounds like C/Java?

**NO!**

**Environments (“Phone Book”)**

How ML deals with variables
- Variables = “names”
- Values = “phone number”

```
x: int
y: int
```

**Environments and Evaluation**

ML begins in a “top-level” environment
- Some names bound

```ocaml
let x = e;;
```

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. **Evaluate** `expr e` in current env to get value `v : t`
2. **Extend** env to bind `x` to `v : t`
(Repeat with next binding)
Environments

“Phone book”
• Variables = “names”
• Values = “phone number”

1. Evaluate:
Find and use most recent value of variable

2. Extend:
Add new binding at end of “phone book”

Example

```ocaml
let x = 2+2;;
val x : int = 4

let y = x * x * x;;
val y : int = 64

let z = [x;y;x+y];;
val z : int list = [4;64;68]
```

How is this different from C/Java’s “store”?

```ocaml
let f = fun y -> x + y;;
val f : int -> int = fn

let x = x + x ;;
val x : int = 8;
# f 0;;
val it : int = 4
```

1. Evaluate: Use most recent bound value of var
2. Extend: Add new binding at end

How is this different from C/Java’s “store”? 

```ocaml
let x = 2+2;;
val x : int = 4

let f = fun y -> x + y;;
val f : int -> int = fn

let x = x + x ;;
val x : int = 8;
# f 0;
val it : int = 4
```

New binding!

Environment at fun declaration frozen inside fun “value”

• Frozen env used to evaluate application (# _)

Q: Why is this a good thing?

```ocaml
let x = 2+2;;
val x : int = 4

let f = fun y -> x + y;;
val f : int -> int = fn

let x = x + x ;;
val x : int = 8;
# f 0;
val it : int = 4
```

Cannot change the world

Cannot “assign” to variables
• Can extend the env by adding a fresh binding
• Does not affect previous uses of variable

Environment at fun declaration frozen inside fun “value”

• Frozen env used to evaluate application (# _)

Q: Why is this a good thing?
Cannot change the world

Q: Why is this a good thing?
A: Function behavior frozen at declaration

- Nothing entered afterwards affects function
- Same inputs always produce same outputs
  - Localizes debugging
  - Localizes reasoning about the program
- No “sharing” means no evil aliasing

Examples of no sharing

Remember: No addresses, no sharing.
- Each variable is bound to a “fresh instance” of a value
  - Tuples, Lists ...
- Efficient implementation without sharing?
- There is sharing and pointers but hidden from you

- Compiler’s job is to optimize code
- Efficiently implement these “no-sharing” semantics
- Your job is to use the simplified semantics
- Write correct, cleaner, readable, extendable systems

Function bindings

Functions are values, can bind using `val`

```ocaml
let fname = fun x -> e ;;
```

Problem: Can’t define recursive functions!
- `fname` is bound after computing rhs value
- no (or “old”) binding for occurrences of `fname` inside `e`

```ocaml
let rec fname x = e ;;
```

Occurrences of `fname` inside `e` bound to “this” definition

```ocaml
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```

Recap: Environments

“Phone book”

- Variables = “names”
- Values = “phone number”

1. Evaluate:
   - Find and use most recent value of variable
2. Extend: `let x = e ;;`
   - Add new binding at end of “phone book”

What about more complex data?

- We’ve seen some base types and values:
  - Integers, Floats, Bool, String etc.
- Some ways to build up types:
  - Products (tuples), records, “lists”
  - Functions
- Design Principle: Orthogonality
  - Don’t clutter core language with stuff
  - Few, powerful orthogonal building techniques
  - Put “derived” types, values, functions in libraries

What about more complex data?

- Values = “phone number”
- Variables = “names”
- Expressions = “phone book”

Many kinds of expressions:

1. Simple
2. Variables
3. Functions
Next: Building datatypes

Three key ways to build complex types/values

1. “Each-of” types
   Value of T contains value of T1 and a value of T2

2. “One-of” types
   Value of T contains value of T1 or a value of T2

3. “Recursive”
   Value of T contains (sub)-value of same type T

Suppose I wanted ...

... a program that processed lists of attributes
- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (float)
- Alive (boolean)
- Phone (int-int)
- Email (string)

Many kinds of attributes:
- too many to put in a record
- can have multiple names, addresses, phones, emails etc.

Want to store them in a list. Can I?

Constructing Datatypes

\[
\text{type } t = C_1 \text{ of } t_1 | C_2 \text{ of } t_2 | \ldots | C_n \text{ of } t_n
\]

\text{t is a new datatype.}
A value of type \( t \) is either:
- a value of type \( t_1 \) placed in a box labeled \( C_1 \)
- or a value of type \( t_2 \) placed in a box labeled \( C_2 \)
- or \( \ldots \)
- or a value of type \( t_n \) placed in a box labeled \( C_n \)

Suppose I wanted ...

Attributes:
- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- email (string)

\[
\text{type } \text{attrib} =
\begin{align*}
\text{Name of string} \\
\text{Age of int} \\
\text{DOB of int*int*int} \\
\text{Address of string} \\
\text{Height of real} \\
\text{Alive of bool} \\
\text{Phone of int*int} \\
\text{Email of string} \\
\end{align*}
\]

Creating Values

How to create values of type \text{attrib}?

\[
\begin{align*}
\text{let } a_1 = \text{Name}\ "John"; \\
\text{val } x : \text{attrib} = \text{Name}\ "John" \\
\text{let } a_2 = \text{Height} 5.83; \\
\text{val } a_2 : \text{attrib} = \text{Height} 5.83 \\
\text{let } year = 1977 ;; \\
\text{val } \text{year} : \text{int} = 1977 \\
\text{let } a_3 = \text{DOB} (9,8,\text{year}) ;; \\
\text{val } a_3 : \text{attrib} = \text{DOB} (9,8,1977) \\
\text{let } a_1:a_2:a_3 ;; \\
\text{val } a_3 : \text{attrib list} = ...
\end{align*}
\]

One-of types

- We’ve defined a “one-of” type named \text{attrib}
- Elements are one of:
  - string,
  - int,
  - int*int*int,
  - float,
  - bool ...

- Can create uniform \text{attrib} lists
- Suppose I want a function to print \text{attribs}...
How to tell what's in the box?

Pattern-match expression: check if \( e \) is of the form ...
- On match:
  - value in box bound to pattern variable
  - matching result expression is evaluated
- Simultaneously test and extract contents of box

**match-with** is an Expression

- Value in box bound to pattern variable
- Matching result expression is evaluated
- Simultaneously test and extract contents of box

**Benefits of match-with**

1. Simultaneous test-extract-bind
2. Compile-time checks for:
   - missed cases: ML warns if you miss a \( t \) value
   - redundant cases: ML warns if a case never matches

**What about “Recursive” types?**

- Type rules:
  - \( e_1, e_2, \ldots, e_n \) must have same type
  - Which is type of whole expression

- Type rules:
  - \( e_1, e_2, \ldots, e_n \) must have same type
  - Which is type of whole expression

**What about “Recursive” types?**

- Type \( t \) =
  - \( C_1 \) of \( t_1 \)
  - \( C_2 \) of \( t_2 \)
  - \( \ldots \)
  - \( C_n \) of \( t_n \)

- Type \( \) int_list =
  - Nil
  - Cons of int * int_list

Think about this! What are values of \( \) int_list ?
- Cons(1,Cons(2,Cons(3,Nil)))
- Cons(2,Cons(1,Nil))
- Cons(3,Nil)
- Nil

- Lists aren’t built-in!

Lists are a derived type: built using elegant core!
1. Each-of
2. One-of
3. Recursive

:: is just a pretty way to say “Cons”
[] is just a pretty way to say “Nil”
Some functions on Lists: Length

```ocaml
let rec len l =
  match l with
  Nil -> 0
| Cons(_,t) -> 1 + (len t)
```

Base pattern
Ind pattern

Base Expression
Inductive Expression

Matches everything, no binding
Pattern-matching in order
- Must match with Nil

null, hd, tl are all functions ...

Bad ML style: More than aesthetics!
Pattern-matching better than test-extract:
- ML checks all cases covered
- ML checks no redundant cases
- ...at compile-time:
  - fewer errors (crashes) during execution
  - get the bugs out ASAP!

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:
- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) = -14.766$

Q: What's a ML datatype for such expressions?

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:
- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) = -14.766$

Whats a ML function for evaluating such expressions?
Functions

Two questions about function values:

What is the value:
1. ... of a function?
2. ... of a function "application" (call)?

Values

"Body" expression not evaluated until application
- but type-checking takes place at compile time
- i.e. when function is defined

Function value =
- <code + environment at definition>
- "closure"

Values of function application

Application: fancy word for "call"

• "apply" the argument e2 to the (function) e1

Application Value:
1. Evaluate e1 in current env to get (function) v1
   - v1 is code + env
   - code is (formal x + body e1), env is E
2. Evaluate e2 in current env to get (argument) v2
3. Evaluate body e in env E extended by binding x to v2

Example 1

let x = 1;
let f y = x + y;
let x = 2;
let y = ;
(f (x + y));
Example 1

```ocaml
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 4;;
f (x + y);;
```

Example 2

```ocaml
let x = 1;;
let f y =
let x = 2 in
fun z -> x + y + z
;;
let x = 100;;
let g = f 4;;
let y = 100;;
g 1;;
```

Example 3

```ocaml
let f g =
let x = 0 in
g 2
;;
let x = 100;;
let h y = x + y;;
f h;;
```

Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment
- Static/Lexical: Determined from the program text
  - Without executing the program
- Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence