Scheme problems

Unsolved:

- **Printf debugging**

Solved:

- **Need switch or similar**
  (Clausal definition, case expression)
- **Only cons for data**
  (Define as many forms as you like: datatype)
- **Wrong number of arguments (typecheck)**
- **car or cdr of non-list (typecheck)**
- **car or cdr of *empty* list (pattern match)**

New vocabulary

Data:

- Constructed data
- Value constructor

Code:

- Pattern
- Pattern matching
- Clausal definition
- Clause

Types:

- Type variable ('a)

Today's plan: programming with types

1. Mechanisms to know:
   - Define a type
   - Create value
   - Observe a value
2. Making types work for you: from types, code

Mechanisms to know

Three programming tasks:

- **Define a type**
- **Create value ("introduction")**
- **Observe a value ("elimination")**

  For functions,

  All you can do with a function is apply it

  For constructed data,

  "How were you made & from what parts?"

Datatype definitions

```
datatype  suit     = HEARTS | DIAMONDS | CLUBS | SPADES

datatype 'a list   = nil              (* copy me NOT! *)
                   | op :: of 'a * 'a list

datatype 'a heap   = EHEAP
                   | HEAP of 'a * 'a heap * 'a heap

type suit           val HEARTS : suit, ...
type 'a list        val nil    : forall 'a . 'a list
                    val op :: : forall 'a .
                                'a * 'a list -> 'a list
type 'a heap
val EHEAP: forall 'a.                          'a heap
val HEAP : forall 'a.'a * 'a heap * 'a heap -> 'a heap
```

Structure of algebraic types

An algebraic data type is a collection of alternatives

- Each alternative must have a name

The thing named is the value constructor

(Also called “datatype constructor”)

Your turn: Define a type

An ordinary S-expression is one of

- A symbol (string)
- A number (int)
- A Boolean (bool)
- A list of ordinary S-expressions

Two steps:

1. For each form, choose a value constructor
2. Write the datatype definition

Your turn: Define a type

```
datatype sx
  = SYMBOL of string
  | NUMBER of int
  | BOOL   of bool
  | SXLIST of sx list
```

Other constructed data: Tuples

Always only one way to form

- **Expressions** $(e_1, e_2, \ldots, e_n)$
- **Patterns** $(p_1, p_2, \ldots, p_n)$

Example:

```
let val (left, right) = splitList xs
in  if abs (length left - length right) < 1
    then
      NONE
    else
      SOME "not nearly equal"
end
```

”Eliminate” values of algebraic types

New language construct case (an expression)

```
fun length xs =
  case xs
    of []       => 0
     | (x::xs) => 1 + length xs
```

Clausal definition is preferred
(sugar for val rec, fn, case)

`case` works for any datatype

```
fun toStr t =
    case t
      of EHEAP => "empty heap"
       | HEAP (v, left, right) =>
                    "nonempty heap"
```

But often a clausal definition is better style:

```
fun toStr' EHEAP = "empty heap"
  | toStr' (HEAP (v,left,right)) =
                    "nonempty heap"
```

Bonus content

The rest of this slide deck is “bonus content” (intended primarily for those without books)

Define algebraic data types for SX_1 and SX_2, where

$$SX_1 = ATOM \cup LIST(SX_1)$$

$$SX_2 = ATOM \cup \{\texttt{(cons}\ \ v_1\ \ v_2\texttt{)} \mid v_1 \in SX_2, v_2 \in SX_2\}$$

(take $ATOM$, with ML type `atom` as given)

Wait for it . . .

Exercise answers

```
datatype sx1 = ATOM1 of atom
             | LIST1 of sx1 list

datatype sx2 = ATOM2 of atom
             | PAIR2 of sx2 * sx2
```

Exception handling in action

```
loop (evaldef (reader (), rho, echo))
handle EOF                 => finish ()
  | Div                    => continue "Division by zero"
  | Overflow               => continue "Arith overflow"
  | RuntimeError msg       => continue ("error: " ^ msg)
  | IO.Io {name, ...}      => continue ("I/O error: " ^
                                        name)
  | SyntaxError msg        => continue ("error: " ^ msg)
  | NotFound n             => continue (n ^ "not found")
```

ML Traps and pitfalls

Order of clauses matters

```
fun take n (x::xs) = x :: take (n-1) xs
  | take 0 xs      = []
  | take n []      = []

(* what goes wrong? *)
```

Gotcha — overloading

```
- fun plus x y = x + y;
> val plus = fn : int -> int -> int
- fun plus x y = x + y : real;
> val plus = fn : real -> real -> real
```

Gotcha — equality types

```
- (fn (x, y) => x = y);
> val it = fn : ∀ ''a . ''a * ''a -> bool
```

Tyvar `''a` is "equality type variable":

- **values must "admit equality"**
- **(functions don't admit equality)**

Gotcha — parentheses

Put parentheses around anything with |

- `case`, `handle`, `fn`

Function application has higher precedence than any infix operator

Syntactic sugar for lists

```
- 1 :: 2 :: 3 :: 4 :: nil; (* :: associates to the right *)
> val it = [1, 2, 3, 4] : int list

- "the" :: "ML" :: "follies" :: [];
> val it = ["the", "ML", "follies"] : string list

> concat it;
val it = "theMLfollies" : string
```

ML from 10,000 feet

The value environment

Names bound to immutable values

- **Immutable `ref` and `array` values point to mutable locations**

ML has no binding-changing assignment

Definitions add new bindings (hide old ones):

- `val` ***pattern*** = ***exp***
- `val rec` ***pattern*** = ***exp***
- `fun` ***ident patterns*** = ***exp***
- `datatype ... = ...`

Nesting environments

At top level, definitions

Definitions contain expressions:

def* ::= `val` *pattern* = *exp

Expressions contain definitions:

***exp* ::= `let` *defs* `in` *exp* `end`**

Sequence of *defs* has let-star semantics

What is a pattern?

```
pattern ::= variable
          | wildcard
          | value-constructor [pattern]
          | tuple-pattern
          | record-pattern
          | integer-literal
          | list-pattern
```

Design bug: no lexical distinction between

- **VALUE CONSTRUCTORS**
- **variables**

Workaround: programming convention

Function pecularities: 1 argument

Each function takes 1 argument, returns 1 result

For "multiple arguments," use tuples!

```
fun factorial n =
  let fun f (i, prod) =
        if i > n then prod else f (i+1, i*prod)
  in  f (1, 1)
  end

fun factorial n =  (* you can also Curry *)
  let fun f i prod =
        if i > n then prod else f (i+1) (i*prod)
  in  f 1 1
  end
```

Mutual recursion

Let-star semantics will not do.

Use `and` (different from `andalso`)!

```
fun a x = ... b (x-1) ...
and b y = ... a (y-1) ...
```

Syntax of ML types

Abstract syntax for types:

$ty \Rightarrow$	TYVAR of string	type variable
|	TYCON of string * ty list	apply type constructor

Each tycon takes fixed number of arguments.

nullary	`int`, `bool`, `string`, ...
unary	`list`, `option`, ...
binary	`->`
n-ary	tuples (infix `*`)

Syntax of ML types

Concrete syntax is baroque:

ty ⇒ *tyvar*	type variable
| *tycon*	(nullary) type constructor
| *ty tycon*	(unary) type constructor
| `(`*ty*`, ...,` *ty*`)` *tycon*	(n-ary) type constructor
| *ty* `*` ... `*` *ty*	tuple type
| *ty* `->` *ty*	arrow (function) type
| `(`*ty*`)`	

tyvar ⇒ *'identifier* `'a, 'b, 'c, ...`

tycon ⇒ *identifier* `list, int, bool, ...`

Polymorphic types

Abstract syntax of type scheme σ:

$\sigma \Rightarrow$ **FORALL of tyvar list * ty**

Bad decision: $\forall$ left out of concrete syntax

```
(fn (f,g) => fn x => f (g x))
   : ∀ 'a, 'b, 'c .
      ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)
```

Key idea: subtitute for quantified type variables

Old and new friends

```
op o    : ∀ 'a, 'b, 'c .
          ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
length : ∀ 'a . 'a list -> int
map     : ∀ 'a, 'b .
          ('a -> 'b) -> ('a list -> 'b list)
curry   : ∀ 'a, 'b, 'c .
          ('a * 'b -> 'c) -> 'a -> 'b -> 'c
id      : ∀ 'a . 'a -> 'a
```