Some Recursive Functions

Function `fact` computes factorial:

```
fun fact(n) =
  if n = 0 then 1 else n*fact(n-1);
```

Function `len` computes the number of elements in a list:

```
fun len(x) =
  if null(x) then 0 else 1 + len(tl(x));
```

Function `fib` computes the Fibonacci sequence 1, 1, 2, 3, 5, 8:

```
fun fib(n) =
  if n = 0 orelse n = 1 then 1
  else fib(n-1) + fib(n-2);
```


Programming in ML - Recursive Functions

Idea:
  Define the base case
  Define a way to solve a smaller problem recursively.

Function `append` which concatenates two lists (works like `@`).

Ex. `append ([1, 2], [2, 3, 4]) = [1, 2, 2, 3, 4]`

Recursive definition:

```
append([], z) = z
append(a::y, z) = a ::  append(y,z)
```

Finally, the function definition:

```
fun append(x, z) = if null(x) then z
    else hd(x) ::  append(tl(x), z)
```


Function Declaration by Cases

Motivation: this is a notation for declaring functions to manipulate recursive data structures such as lists. Two cases- empty and non-empty list. Empty list matches with [], non-empty - with `a::y`, where `a` is some head element, and `y` is the tail.

Idea: examine the structure of a list implicitly, without the explicit use of `null`, `hd` and `tl`

Examples:

```
fun append([], z) = z
|   append(a::y, z) = a :: append(y,z)
fun length([]) = 0
|   length(a::y) = 1 + length(y)
```

Both definitions of `length` and `append` work the same way. However, the `if` statement explicitly specifies the order in which the cases are examined.

How does this work?

Answer: Pattern matching.

_ matches with anything
$< var >$ matches with anything
[] matches with empty list
a::y matches with a list [a,...]
(x,y) matches with a tuple with two elements - a pair

Examples:

```
fun length([]) = 0
|   length(_::y) = 1 + length(y);
fun first(x,y) = x;
fun first(x,_) = x;
fun second(x,y) = y;
```

Case Analysis, Cont'd

ML gives errors if the cases in a function declaration are not exhaustive.

```
fun head(a::y) = a;
  Warning: match not exhaustive
    a :: y => ...
  val head = fn : 'a list -> 'a
```

It also gives errors if cases are redundant.

```
fun f(nul) = 0
|  f(a::y) = a+f(y);
  Warning: redundant patterns in match
    nul => ...
  --> a :: y => ...
  val f = fn : 'a list -> int
```


Case Analysis(Cont'd)

Cannot use a formal parameter more than once in a pattern.

Example: function strip(a,x) that removes leading a's from list z:

```
strip(1, [1,1,3,2,1,2,1,1]) = [3,2,1,2,1,1]
```

Recursive definition:

```
strip(a,[]) = []
strip(a, a::y) = strip(a,y)
strip(a, b::y) = b::y, if b ≠ a
```

The function definition:

```
fun strip(a, []) = []
|  strip(a, a::y) = strip(a,y)
|  strip(a, b::y) = b::y;
  Error: duplicate variable in pattern(s): a
```

Correct definition:

```
fun strip(a, []) = []
|  strip(a, b::y) =
     if a=b then strip(a, y) else b::y;
```


Functions that involve more than two cases

Example - remove even numbers from the list.

```
remove-even [1,4,3,6,5,3,8,7] = [1,3,5,3,7]
```

Recursive definition:

```
remove-even([]) = []
remove-even(a::y) = remove-even(y),
     if even(a)
remove-even(a::y) = a::remove-even(y)
```

Function definition:

```
fun remove-even([]) = []
|  remove-even([a::y]) = if not even(a)
        then a::remove-even(y)
        else remove-even(y)
```


Datatypes in ML

- Can define new data types and define functions over these data types using pattern-matching.
- Scheme - all data types (lists, records, etc) were basically defined over a single data type, lists. Scheme has dynamic typing.
- ML - static typing, so we need to introduce separate data types. Saw lists and tuples (records). What about tree data types with different node types in a tree? (In Scheme these would still be lists, possibly with the car of the list is a symbol tagging the node type and the remainder of the list included any subtrees of that node.)
- Feature exists in all modern functional languages (Hope, Miranda, Haskell, etc).

Type Definition in ML

1) A type name is an abbreviation for a type expression:

```
type complex = {re:real, im:real};
  type complex = {re:real, im:real}
fun add (x:complex) (y:complex) :  complex
= {re = (# re x) + (# re y), im = (# im x)
+ (# im y)} ;
  val add = fn : complex → complex → complex
add { re=4.5, im=1.0 } { re=2.3, im=8.9 }
  val it = {re=6.8, im=9.9} : complex
```

Note:
- There is no data abstraction - the `add` procedure accesses both of its arguments as records, even though they are declared as `complex`
- Arguments to `add` do not have to be declared `complex`, as long as they have the same structure as `complex`. (ML uses `structural equivalence`.)

Type Definition in ML (Cont'd)

2) Can define enumerated types, which creates new type constructors and new data constructors:

```
datatype days = sun | mon | tue | wed
    | thu | fri | sat ;
  type days
  con sun : days
  ...
  con sat : days
```

Type constructor is `days` and data constructors (new constants that have been dynamically added to the program by the declaration) are `sun`, `mon`, ... `sat`.

Can use pattern matching to operate on these datatypes.

Type Definition - Example

```
fun what_to_do sun = "rest"
| what_to_do mon = "program"
| what_to_do tue = "program"
| what_to_do wed = "program"
| what_to_do thu = "program"
| what_to_do fri = "party"
| what_to_do sat = "shop"
  val what_to_do = fn : days → string
```

This version has a lot of tiresome repetition. Can get avoid it by supplying a default case:

```
fun what_to_do sun = "rest"
| what_to_do fri = "party"
| what_to_do sat = "shop"
| what_to_do _ = "program"
  val what_to_do = fn : days → string
```


Type Definition in ML - Patterns

Datatype declarations can declare more than just enumerated types, and patterns can be very elaborate. We can define new types in terms of existing types:

```
datatype int_pair = pair of int * int
  type int_pair
    con pair : int * int → int_pair

fun proj1 (pair p) = (#1 p);
  val proj1 = fn : int_pair → int
```

Notes:
- Unlike the `type` declaration, this introduces a new type constructor `int_pair` which only matches with itself. This is *name equivalence*.
- Data constructors can be parameterized by values, and applications can also be patterns.

Notes on Patterns, Cont'd

- However, procedure being applied in a pattern application must be a data constructor name.
- Patterns can contain variables, but these are *defining occurrences*. In definition of `proj1`, the variable `p` is declared in the pattern and used on the right hand side.
- Variables can occur at most once in a pattern.

Patterns are used to define functions that can extract parts of any *observable* value. A pattern is composed of data constructors (possibly applied to patterns), variables and the special underscore character.

Patterns, Cont'd

Patterns can be used to extract components of recursive data structures.

```
val (x, y) = (17, true);
  val x = 17 : int
  val y = true : bool
fun proj1 (x, _) = x;
  val proj1 = fn: 'a * 'b → 'a
fun car ( a ::  _) = a
| car nil = ...?
  car = fn : 'a list → 'a
fun length ( _::  x) = 1 + (length x)
| length nil = 0;
  val length = fn :  'a list → int
```

Can also have "deep" patterns:

```
fun cadr (_ ::  ( a ::  _) ) = a
| cadr [ _ ] = ...?
| cadr nil = ...?
  val cadr = fn : 'a list → 'a
```


How ML Does Pattern-Matching

A pattern can be represented by a tree.
Example: Pattern `(x::y::zs, w)` is represented as

Now, let's try to match it with expression ([1,2,3,4], 5), represented as

Pattern-Matching (Cont'd)

To match a pattern and expression, overlay their trees starting by matching the roots and continuing recursively by matching the children.

If a match is not possible, the pattern-matching fails.
Example: matching `x::xs` with `nil`.

Pattern-Matching (Cont'd)

Example: matching `x::y::zs` with `[1]`.

When matching is done successfully, then identifiers at the leaves of the pattern tree match nodes that represent subexpressions. These become the values associated with those identifiers.
Example:

Here, `x` corresponds to 1, `y` corresponds to 2, `zs` corresponds to `3::4::nil` or `[3,4]`, and `w` corresponds to `5`.

Pattern-Matching (Cont'd)

Subtle Pattern Bug - suppose we want special meaning when the list is empty (`nil`), but misspell it slightly:

```
fun reverse(niil) = nil
|   reverse(x::xs) = reverse(xs) @ [x];
```

```
Warning:  redundant patterns in match
    niil ⇒ ...
  → x::xs ⇒ ...
val reverse = fn:  'a list → 'a list
```

```
reverse([1,2,3]);
  val it = []:  int list
```

What happened? `niil` is matched with any argument, and so the pattern on the second line will never be reached.

Variant Records

```
datatype money = nomoney | coin of int |
    note of int | check of string * int;
    type money
      nomoney :  money
      coin :  int → money
      note :  int → money
      check:  string * int → money
```

```
fun amount nomoney = 0
  | amount coin(cents) = cents
  | amount note(dollars) = 100 * dollars
  | amount check(bank, pence) = pence;
  val amount = fn :  money → int
```


Recursive Datatypes - Example

Datatype P *bitree*

```
datatype bitree = leaf
| nonleaf of bitree * bitree
```

Here we have two value constructors, *leaf* and *nonleaf*.

Value constructors can appear within expressions as constants if they are parameterless and as functions if they take parameters.

What do the following expressions correspond to?

```
  leaf
  nonleaf (leaf, leaf)
  nonleaf (leaf, nonleaf (leaf, leaf))
  nonleaf (nonleaf (nonleaf (leaf, leaf), leaf),
leaf)
```

Example (Cont'd)

Let's define functions on binary trees (containing nothing).

Count leaves:

```
fun leafcount(leaf) = 1
|   leafcount(nonleaf(s, t)) =
      leafcount(s) + leafcount(t);
```

Is the tree a leaf?

```
fun isleaf(leaf) = true
|   isleaf(nonleaf(_)) = false;
```

Extract right subtree

```
fun right(nonleaf(s, t)) = t;
```

What happens if tree is `leaf`?

Polymorphism: Parameterized Types

ML is known for its support for *polymorphic* functions, i.e., functions that can be applied to parameters of more than one type. (More later)

For all lists, the function `hd` returns the head of a list:

```
hd [1,2,3];
  val it = 1 : int
hd ["a", "b", "c"];
  val it = "a" : string
```

So, what is type of `hd`?

```
hd;
  val it = fn : 'a list -> 'a
```

ML uses a leading quote to identify a type parameter. The type of `hd` can be read as "function from a list of any type `'a` to type `'a`".

Type Variables (new in ML)

We have already seen type constants (such as `string`, `int`) and type operators (such as `list`, =>). *Type constructor* takes a collection of type arguments and returns a valid type. For example, the following are all valid types:

```
string list
(int list) list
int * (int list)
```

Tuples are n-ary type constructors, while the arrow type is a binary type constructor. In addition to type constructors, ML has *type variables*, denoted by quoted identifiers:

`'a, 'b, ...`

Do not confuse ML type variables with Scheme symbols!

Program variables - "placeholders" for values at runtime

Type variables - placeholders for types.

Polymorphic Data Structures

Type variables allow us to define polymorphic data structures:

```
datatype ('a, 'b) pair = pair of 'a * 'b
  type pair
  con pair : 'a * 'b → ('a, 'b) pair

fun proj1 ( pair (x, _) ) = x;
  val proj1 = fn : ('a, 'b) pair → 'a
```

Note that here we use the pairing notation to signify a type constructor of two type arguments

Recursive Datatypes

```
datatype 'a mylist = mynil | mycons of 'a *
('a mylist)
```

type mylist

cons mynil : 'a mylist

con mycons : 'a * ('a mylist) → 'a mylist

```
length (mycons ( _, l)) = 1 + (length l)
| length mynil = 0;
```

val length = fn : 'a mylist → int

- Allow to have user-defined structures of unbounded size
- Any datatype which does not contain arrow types is types is also an *eqtype*, i.e., based on the structure of a datatype definition, the compiler can automatically build an equality predicate it.

More Recursive Datatypes - "Real" Binary Trees

```
datatype 'label btree =
   Empty |
   Node of 'label * 'label btree * 'label btree;
```

```
datatype 'label btree
con Empty:  'label btree
con Node:  'label * 'label btree * 'label btree
```

→ 'label btree

Example:

corresponds to

```
Node("ML",
   Node("as",
      Node("a",Empty,Empty),
      Node("in",Empty,Empty)
   ),
   Node("types",Empty,Empty)
)
```


Mutually-Recursive Datatypes

An *even tree* is a binary tree in which each path from the root to a node with one or two empty subtrees has an even number of nodes. An `odd tree` is the one in which each path from root to a node with nonzero empty subtrees has an odd number of nodes.

```
datatype
  'label evenTree = Empty |
     Enode of 'label * 'label oddTree
     * 'label oddTree
and
  'label oddTree =
     Onode of 'label * 'label evenTree
     * 'label evenTree;
```

```
datatype 'label evenTree
con Empty: 'label evenTree
con Enode: 'label * 'label oddTree * 'label oddTree
           -> 'label evenTree
datatype 'label oddTree
con Onode: 'label * 'label evenTree * 'label evenTree
           -> 'label oddTree
```


Mutually Recursive Datatypes (Cont'd)

```
fun buildEven(x,T1,T2) = Enode(x,T1,T2);
fun buildOdd(x,T1,T2) = Onode(x,T1,T2);

val t1 = buildOdd(1,Empty,Empty);
```

val t1 = Onode(1,Empty,Empty) : int oddTree

```
val t2 = buildOdd(2,Empty,Empty);
```

val t2 = Onode(2,Empty,Empty) : int oddTree

```
val t3 = buildEven(3,t1,t2);
```

val t3 = Enode(3, Onode(1,Empty,Empty), Onode(2,Empty,Empty)): int evenTree

```
val t4 = buildOdd(4,t3,Empty);
```

val t4 = Onode(4, Enode(3, Onode #, Onode #), Empty): int oddTree

```
val t5 = buildEven(5,t4,t4)
```

val t5 = Enode(5, Onode(4, Enode #, Empty), Onode(4, Enode #, Empty)): int evenTree

Mutually-Recursive Datatypes (Cont'd)

```
Enode(5,
  Onode(4,
    Enode(3,
      Onode(1,Empty,Empty),
      Onode(2,Empty,Empty)),
    Empty
  ),
  Onode(4,
    Enode(3,
      Onode(1,Empty,Empty),
      Onode(2,Empty,Empty)),
    Empty
  )
)
```

Picture:

Type Variables (cont'd)

Example:

```
fun Id x = x;
  Id = fn : 'a -> 'a
(Id 3, Id true, Id "hello")
  val it = (3, true, "hello") : int * bool * string
```

`Id` is the identity function. But not enough type information for the compiler to figure out the domain and range type, but it knows that both types should be the same. Compiler infers a *minimally constrained type* for `Id`, with `'a` as a placeholder.

In previous example, `Id` is used in three different places with different types (`int`, `bool` and `string`). Definition of `Id` makes no assumption about the structure of its argument, and so the type of its formal parameter could be filled in with any type whatsoever.

Parametric Polymorphism

The different occurrences of `Id` are filled in differently. The definition of `Id` is implicitly parameterized by the type of its argument, with this type information filled in at the points where `Id` is called.

This is an example of *parametric polymorphism*.

Implicit type parameters are not just placeholders. They also constrain the calling patterns of polymorphic procedures. For `Id`, its return type is the same as the type of its actual argument.

Parametric Polymorphism (Cont'd)

Another example:

```
fun apply P A = P A ;
  val apply = fn : ('a -> 'b) -> 'a -> b
```

The type `'a -> 'b` of the procedure parameter is an example of a very general phenomenon - building type expressions involving type variables which give partial structure to the types of procedure's arguments and result.

Other examples of type expressions:

```
- 'a * 'b
- list
```

`apply` is defined over any procedure value.

Parametric Polymorphism - Drawbacks

1. A polymorphic procedure cannot be passed as an argument to another procedure:

```
fun applyTwice P = (P true, P 3);
  std_inL 0.0-0 Error: operator and operand don't
  agree (tycon mismatch)
    operator domain: bool
    operand: int
    in expression: P 3
```

What is wrong? The compiler will not allow the formal parameter P to be polymorphic. From the first usage of `P` it determines that `P`'s argument should be `int`, but this conflicts with the second usage of `P`.

Another example:

```
let fun Id x in (Id true, Id 3)
  val it = (true, 3) : bool * int
```

Here, ML "knows" what the polymorphic procedure `P` is and can check that the intended type of its usage should be, whereas in the former case we do not have enough information about `P` to type-check its usage.

This can be regarded as a weakness of the fact that ML is *implicitly typed*. More powerful languages (Pebble, Russell, Quest are proposed in research community) are based on explicit typing.

Parametric Polymorphism - Drawbacks (Cont'd)

2. ML does not really handle overloading.

Consider the following:

```
4 + 8;
  val it = 12 : int
5.6 + 9.31 ;
  val it = 14.91 : real
```

The + operation is overloaded to operate over both integers and reals. So, is + a polymorphic operation?

The difference between this and parametric polymorphism is that these are really two + operations, one for integers and the other for reals, with the compiler deciding on the basis of the type information which + operation to insert into the code. On the other hand, there is really only a single `length` operation, which executes the same code at each call site.

Ad-hoc Polymorphism

Overloading is sometimes referred to as *ad-hoc polymorphism*. Ad-hoc and parametric polymorphisms do not interact well with each other:

```
fun double x = x + x ;
  std_in:10.18 Error: overloaded variable cannot be
  resolved: +
```

We want to say that `double` is defined over any type for which a + operation is defined. There is no way to say this in ML.

Exceptions - equality (=) and inequality (<>):

```
fun equal x y = ( x = y) ;
  val equal = fn : '' a → ''a → bool
equal [1, 2, 3] [1, 2, 4] ; ⇒ false
equal (true, "hello") (true, "hello"); ⇒ true
```

Note that domain type here is `''a` rather than `'a` - types for which equality is defined (*eq-types*).

Eqtypes

An eqtype is

1. integer, real, boolean and string
2. a product of equality types (for tuples)
3. a list whose elements are of equality type.

Examples:

```
int * int
real list * string
```

Comparing for equality:

```
val x = (1,2);
val y = (2,3);
x = y;          => false
x = (1,2);      => true
val L = [1,2,3];
val M = [2,3];
L <> M;         => true
L = 1::M;       => true
```


Eqtypes (Cont'd)

But functions cannot be compared:

```
fun identity(x) = x;
  val identity = fn: 'a → 'a
identity = identity
  Error: operator and operand don't agree
      (equality type required)
  operator domain: ''Z * ''Z
  operand: ('Y → 'Y) * ('X → 'X)
  in expression: = (identity, identity)
```

Note that different invocations of identity need not apply to the same type, so `'X` $\neq$ `'Y`.

Another Example

```
fun reverse1(L) =
     if L = nil then nil
     else reverse1(tl(L)) @ [hd(L)];
  val reverse = fn: ''a list → ''a list

fun reverse2(nil) = nil
|   reverse2(x::xs) = reverse2(xs) @ [x]
  val reverse = fn : 'a list → 'a list

reverse2([identity, identity])
  val it = [identity, identity] : ('a → 'a) list
reverse1([identity, identity])
  Error: operand and operator don't agree
       (equality type required)
  ...
```


Eqtypes (Cont'd)

Built-in function `null` allows checking lists for emptiness without requiring that the list is an equality type.

```
fun reverse3(L) =
     if L = nil then nil
     else reverse3(tl(L)) @ [hd(L)];
  val reverse = fn: 'a list → 'a list

reverse3([identity, identity])
  val it = [identity, identity] : ('a → 'a) list
```

Higher-Order Programming

Recall: a function is *higher order* if it takes a function as input, or returns a function as output. Otherwise, it is *first order*

```
map square [1,2,3,4] => [1,4,9,16]
map hd [[1,2],[3,4],[5,6]] => [1,3,5]
map tl [[1,2,3],[4,5,6]] => [[2,3],[5,6]]
```

Anonymous (unnamed) functions are like lambda functions in Scheme:

Scheme: `(lambda(x)(* x x))`

ML: `fn x => x*x`

```
map (fn x=>x*x)[1,2,3,4] => [1,4,9,16] :int list
map (fn x=>x+1)[1,2,3,4] => [2,3,4,5] :int list
map (fn l => hd(tl l)) [[1,2,3],[4,5,6]]
        =>[2,5] : int list
```


Higher-Order Programming (Cont'd)

Returning functions as values:

```
fun compose(F,G) = (fn x => F(G(x)));
compose(hd,tl) = (fn x => hd(tl(x)));
compose(hd,tl)[1,2,3,4] => 2
map(compose(hd,tl)) [[1,2,3],[4,5,6]] => [2,5]
```


Functions - Cont'd

ML takes functions of just one parameter(!!)

What about `fun add x (y:int) = x + y;`

val add = fn: add → int → int

ML transforms functions of several parameters to a composition of one-parameter functions, using *currying*:

```
fun add x y =
  fn y => (fn x => (x + y))
```

So, we don't have to provide all parameters:

```
add 3;
```

val it = fn : int → int

```
it 4;
```

val it = 7 : int

Functions (Cont'd)

We have seen curried functions or functions with tuple as a parameter. They are "almost" equivalent, but have different types:

```
fun exponent1(x,0) = 1.0
|  exponent1(x,y) = x * exponent1(x,y-1);
```

val exponent1 = fn: real*int → real

```
fun exponent2 x 0 = 1.0
|  exponent2 x y = x * exponent2 x (y-1);
```

val exponent2 = fn: real → int → real

Can easily convert between curried and uncurried notations:

```
fun curry F (x1,x2,x3) = F x1 x2 x3;
```

val curry = fn: ('a → 'b → 'c → 'd) → 'a * 'b * 'c → 'd

```
fun uncurry F x1 x2 x3 = F(x1,x2,x3);
```

val uncurry = fn : ('a * 'b * 'c → 'd) → 'a → 'b → 'c → 'd

Exception Handling

Just as in C++, exceptions in ML are used for handling special cases or failures that occur during the execution of a program. Most usual use - to handle cases for which functions are not defined.
For example, function `hd`:

```
hd [ ];
```

uncaught exception Hd

In Lisp or Scheme, programmers usually return an empty list; in C - return 0. Here, failures can be made explicit.

Exceptions can be defined and raised.

(* exception definition *)

```
exception Nomatch;
```

exception Nomatch : exn

(* exception raising *)

`raise` <*exception-name*>

Exception Handling - Cont'd

Example:

```
fun member a [] = raise Nomatch
|  member a (b::y) = if a = b then y
       else member a y;
member(3, [1,2,3,1,2,3]);
```

val it = [3,1,2,3] : int list

```
member(4, []);
```

uncaught exception Nomatch

So, what happened? We need to not only raise the exception, we also need to handle it.

How to Handle Exceptions

Syntax:
<*expression*>$_1$ `handle` <*exception-name*> => <*expression*>$_2$
to mean that this will evaluate <*expression*>$_1$, but if <*exception-name*> is raised, it will evaluate <*expression*>$_2$ and return its value.

Example:

```
exception Oops; exception Other;
```

exception Oops : exn
exception Other : exn

```
2*2 handle Oops => 0;
```

val it = 5 : int

```
(if 2<3 then raise Oops) handle Oops => 0;
```

val it = 0 : int

```
(raise Oops) handle Other => 0;
```

uncaught exception Other

Dynamic Exception Handling

Unlike type checking, exception handling is done at run-time, using *dynamic scoping*, e.g., if f calls g, g calls h, and h raises an exception, then we look for handlers along the call chain, h, g, f.

Example:

```
exception Neg;
```

exception Neg : exn

```
fun s(m,n) :  int =
  if m >= n then m-n else raise Neg;
```

val s = fn : ((int * int) → int)

```
function subtract(m,n) =
  s(m,n) handle Neg => 0;
```

val subtract = fn : ((int * int) → int)

```
subtract(5,3);
```

val it = 2 : int

```
subtract(5,15);
```

val it = 0 : int