CSE 505: Programming Languages

"But if **thought corrupts language**, **language can also corrupt thought**. A bad usage can spread by tradition and imitation even among people who should and do know better."
George Orwell, *Politics and the English Language,* 1946

"If you cannot be the master of your language, you must be its slave."
Richard Mitchell

"A different language is a different vision of life."
Federico Fellini

"The language we use ... determines the way in which we view and think about the world around us."
The Sapir-Whorf hypothesis

CSE 505: Programming Languages

Instructor: Craig Chambers
TAs: Keunwoo Lee, Michael Ringenburg

Goals:
- study major concepts & design principles in programming languages
- get practical experience using languages embodying concepts & principles
- gain reading-level understanding of formal semantics
- be exposed to some current research

Why?
- understand the capabilities of modern programming language technology
- understand how to exploit this technology in service of more reliable, safer, more flexible systems and more productive humans

Course outline

Functional languages (e.g. **ML**, Scheme, Haskell)
- side-effect-free programming
- recursive first-class functions, recursive data structures
- algebraic data types, pattern-matching
- polymorphic static type systems & type inference

Formal semantics
- lambda calculus & extensions
- static & dynamic (operational) semantics
- key theorems, some proofs

Object-oriented languages (e.g. Smalltalk, Self, Cecil/**Diesel**)
- inheritance, subtype polymorphism
- various models of dynamic dispatching
- polymorphic static type systems

Coursework

Functional & OO sections:
- 1-2 homeworks each
- 1-2 programming projects each
- 1 exam each

Semantics section:
- 1-2 homeworks
- 1 exam

Final exam

Language design goals

Some end goals:

- be easy to learn
- support rapid initial development
- support easy maintenance, evolution
- encourage/guarantee reliability, safety
- encourage/guarantee portability
- allow/encourage efficiency

Some means to these goals:

- readability
- writability
- simplicity [but what does "simple" mean?]
- expressiveness [but what does this mean?]
- fully-specified, platform-independent, safe semantics

Many goals in conflict
⇒ language design is an engineering & artistic activity
⇒ need to consider target audience's needs

Some target audiences

Scientific, numerical computing

- Fortran, APL, ZPL

Systems programming

- C, C++, Modula-3, ...

Applications/symbolic programming

- Java, C#, Lisp, Scheme, ML, Smalltalk, Cecil, Diesel, ...

Scripting, macro languages

- csh, Perl, Python, Tcl, Excel macros, ...

Specialized languages

- SQL, LaTeX, PostScript, Unix regular expressions, ...

Some good language design principles

Strive for a simple, regular, **orthogonal** model

- for evaluation
- for data reference
- for memory management

E.g., be expression-oriented, reference-oriented

Include sophisticated **abstraction** mechanisms,
to define and name abstractions once, use many times

- for control structures, data structures, types, ...

Include polymorphic static type checking

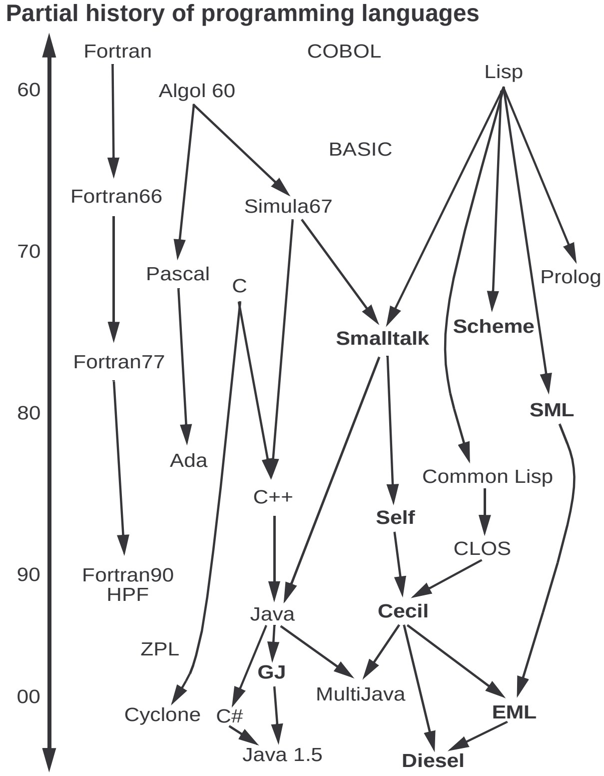
E.g., with universal and existential subtype-bounded quantification

Have a complete & precise language specification

- full run-time error checking for cases not detected statically

Domain-specific languages can exploit domain **restrictions**
for better checking, expressiveness, performance

Partial history of programming languages



ML

Salient features:

- functional
  - functions are first-class values
  - largely side-effect free
- strongly, statically typed
  - polymorphic type system
  - automatic type inference
- expression-oriented, recursion-oriented
- garbage-collected heap
- pattern matching
- exceptions
- advanced module system
- **highly regular and expressive**

Designed as a **M**eta **L**anguage for automatic theorem proving system in mid 70's by Milner et al.

Standard ML: 1986

SML'97: 1997

Caml: a French version of ML, mid 80's

O'Caml: an object-oriented extension of Caml, late 90's

EML: a locally-developed OO extension of ML, 2002

Interpreter interface

Read-eval-print loop

- **read** input expression
  - reading ends with semi-colon (not needed in files)
  - = prompt indicates continuing expression on next line
- **eval**uate expression
- **print** result
- repeat

```
- 3 + 4;
val it = 7 : int
- it + 5;
val it = 12 : int
- it + 5;
val it = 17 : int
```

`it` variable (re)bound to last evaluated value, in case you want to use it again

An interpreter is particularly useful during initial learning and debugging

Basic ML data types and operations

ML is organized around types

- each type defines some set of values of that type
- each type defines a set of operations on values of that type

`int`

- ~, +, -, *, `div`, `mod`; =, <>, <, >, <=, >=; `real`, `chr`

`real`

- ~, +, -, *, /; <, >, <=, >= (no equality); `floor`, `ceil`, `trunc`, `round`

`bool`: different from `int`

- `true`, `false`; =, <>; `orelse`, `andalso`

`string`

- e.g. `"I said \"hi\"\tin dir C:\\stuff\\dir\n"`
- =, <>, ^

`char`

- e.g. `#"a"`, `#"\n"`
- =, <>; `ord`, `str`

Variables and binding

Variables declared and initialized with a `val` binding:

```
- val x:int = 6;
val x = 6 : int
- val y:int = x * x;
val y = 36 : int
```

Variable bindings cannot be changed!

- unlike assignment in C
- like equality in math

Variables can be bound again, but this **shadows** the previous definition

- e.g. `it`

Variable types can be omitted

- they will be **inferred** by ML based on the type of the r.h.s.

```
- val z = x * y + 5;
val z = 221 : int
```

Strong, static typing

ML is **statically typed**: it will check for type errors statically (i.e., when programs are entered, not when they're run)

- opposite extreme: dynamically typed
- blends also possible

ML is **strongly typed**: it catches all type errors (a.k.a. **type safe**) [but which errors are classified as type errors?]

- if not strongly typed, then weakly typed

Examples of other combinations?

	static ←→ **dynamic**
strong	ML
weak	

Type errors

Warning: type errors can look weird, since they use ML jargon:

```
- asd;
Error: unbound variable or constructor: asd
- 3 + 4.5;
Error: operator and operand don't agree
  operator domain: int * int
  operand:         int * real
  in expression:
    3 + 4.5
- 3 / 4;
Error: overloaded variable not defined at type
  symbol: /
  type: int
```


Records

ML records are like C structs

- allow heterogeneous field types, but fixed number of fields

A record type: `{name:string, age:int}`

- field order doesn't matter

Unlike C, can write down a record value directly:

```
{name="Bob Smith", age=20}
```

Unlike C, can construct record values that have run-time expressions specifying the field values

```
{name = "Bob " ^ "Smith",
 age  = 18+num_years_in_college}
```

As with any other value, can bind record values to variables

```
- val bob = {name="Bob " ^ "Smith", age=...};
val bob = {age=20,name="Bob Smith"}
        : {age:int,name:string}
```

Orthogonality in action...

More on records

Can extract record fields using `#fieldname` function (like C's `->` operator, but a regular function)

```
- val bob' = {name= #name(bob),
=             age= #age(bob)+1};
val bob' = {age=21,name="Bob Smith"} : {...}
```

(But wait for pattern-matching, a better way to access components of records)

Cannot assign to a record's fields

- an immutable data structure

Tuples

Like records, but fields ordered by position, not label

Useful for pairs, triples, etc.

A tuple type: `string * int`

- order **does** matter

A tuple value: `("Joe Stevens", 45)`

A tuple expr: `("Joe " ^ "Stevens", 25+num_jobs*10)`

Binding a name to a tuple:

```
- val joe = ("Joe "^"Stevens", 25+num_jobs*10);
val joe = ("Joe Stevens",45) : string * int
```

Can extract tuple fields using #*n* functions
  (but wait for pattern-matching for a better way)

```
- val joe' = (#1(joe), #2(joe)+1);
val joe' = ("Joe Stevens",46) : string * int
```

Cannot assign to a tuple's components

- another immutable data structure

Lists

ML has built-in support for singly-linked lists

- unlike records, require homogeneous element types, but allow variable number of elements

A list type: `int list`

- in general: *T* `list`, for any type *T*

A list value: `[3, 4, 5]`

- `[]` (or `nil`) is the empty list

An expression constructing a list:

```
[1+2, 8 div 2, #age(bob)-15]
```

Binding a name to a list:

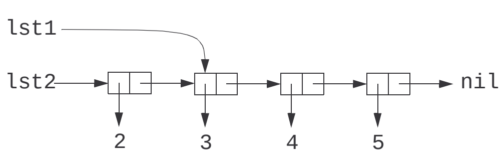
```
- val lst = [1+2, 8 div 2, #age(bob)-15];
val lst = [3,4,5] : int list
```


Basic operations on lists

Add to front of list, non-destructively: `::` (an infix operator)

```
- val lst1 = 3::(4::(5::nil));
val lst1 = [3,4,5] : int list
- val lst2 = 2::lst1;
val lst2 = [2,3,4,5] : int list
```

Adding to the front allocates a new link;
  the original list is unchanged and still available




```
- lst1;
val it = [3,4,5] : int list
```


More on lists

Lists can be nested:

```
- (3 :: nil) :: (4 :: 5 :: nil) :: nil;
val it = [[3],[4,5]]: int list list
```

Lists are homogeneous:

```
- [3, "hi there"];
Error: operator and operand don't agree
  operator domain: int * int list
  operand:         int * string list
  in expression:
    (3 : int) :: "hi there" :: nil
```

Manipulating lists

Test whether a list is empty: `null`

```
- null([]);
val it = true : bool
```

Extract the first ("head") element of the list: `hd`

```
- hd(l1) + hd(l2);
val it = 5 : int
```

Extract the rest ("tail") of the list: `tl`

```
- val lst3 = tl(lst1);
val lst3 = [4,5] : int list
- val lst4 = tl(tl(lst3));
val lst4 = [] : int list
- tl(lst4);  (* or hd(lst4) *)
uncaught exception Empty
```

(Pattern-matching offers alternative ways)

Cannot assign to a list's elements

- another immutable data structure

First-class values

All of ML's data values are **first-class**

- there are no restrictions on how they can be created, used, passed around, bound to names, stored in other data structures, ...

One consequence:
can nest records, tuples, lists arbitrarily

A legal value, and its type:

```
{foo=(3, 5.6, "seattle"),
 bar=[[3,4], [5,6,7,8], [], [1,2]]}
: {bar:int list list, foo:int*real*string}
```

Another consequence:
can create initialized, anonymous values as expressions, instead of using a sequence of statements to first declare (allocate named space) and then assign to initialize

- name-binding is **orthogonal** to value creation

A further consequence:
all data values are fully initialized upon creation

- no safety issues about accessing uninitialized data

Reference data model

A variable **refers to** a value (of whatever type), uniformly

A record, tuple, or list **refers to** its element values, uniformly

- all values are implicitly referred to by pointer
  (even scalars like ints, bools, & chars can be viewed this way, although they're likely implemented more efficiently)

A variable expression evaluates to
a reference to the value that the variable was bound to

A variable binding makes the l.h.s. variable
refer to its r.h.s. value

No implicit copying upon binding, parameter passing, returning from a function, storing in a data structure

- like Java, Scheme, Smalltalk, ...; all high-level languages
- unlike C, where non-pointer values are copied
  - C arrays?

No restrictions on where values may be passed, stored
$\Rightarrow$ values have potentially unlimited lifetime

- implementation allocates all (non-scalar) values in the heap

Garbage collection

ML provides several ways to **allocate** & initialize new values:
`(...)`, `{...}`, `[...]`, `::`

But ML provides no way to **deallocate**/free values that are no longer being used

Instead, ML provides **automatic garbage collection**:
when there are no more references to a value (either from variables or from other objects), it is deemed garbage, and the system will automatically deallocate the value

Evaluation of automatic garbage collection

+ dangling pointers impossible
  (could not guarantee type safety without this!)
+ storage leaks "impossible"
+ simpler programming
+ can be more efficient!
– less ability to carefully manage memory use & reuse

(Automatic GCs exist even for C & C++, as free libraries)

Functions

Some function definitions:

```
- fun square(x:int):int = x * x;
val square = fn : int -> int
- fun swap(a:int, b:string):string*int = (b,a);
val swap = fn : int*string -> string*int
```

A function has a type of the form T_{arg} `->` T_{result}

- if want multiple arguments, use tuple type for T_{arg}
  - `*` binds tighter than `->`
- can use tuple type for T_{result}, too!

Some function calls:

```
- square(3);
val it = 9 : int
- swap(3 * 4, "billy" ^ "bob");
val it = ("billybob",12) : string * int
```


Function call syntax

Since all functions take one argument,
parentheses aren't part of the call syntax:

```
- square 3;
val it = 9 : int
- (square 3) + (square 4);
val it = 25 : int
```

Juxtaposition binds tighter than infix operators:

```
- square 3 + square 4;
val it = 25 : int
- square (3 + square 4);
val it = 361 : int
```

Parentheses common if argument is a tuple expression:

```
- swap (3 * 4, "billy" ^ "bob");
val it = ("billybob",12) : string * int
```


Expression-orientation

Function body is a single expression

```
fun square(x:int):int = x * x
```

- not a statement list
- no `return` keyword

Like equality in math

- a call to a function is equivalent to its body,
  after substituting its formals for the actuals in the call

`(square 3)` $\Leftrightarrow$ (x*x)[x$\rightarrow$3] $\Leftrightarrow$ `3*3`

There are no statements in ML, only expressions

- what would be statements in other languages
  are recast as expressions in ML

If expression

General form:

if *test* **then** *e1* **else** *e2*

- return value of either *e1* or *e2*,
  based on whether *test* is `true` or `false`
- cannot omit `else` part

```
- fun max(x:int, y:int):int =
=     if x >= y then x else y;
val max = fn : int * int -> int
```

Like *test* **?** *e1* **:** *e2* expression in C

- don't need a distinct `if` statement

Static typechecking of if expression

What are the rules for typechecking an `if` expression?

What's the type of the result of `if`?

Some basic principles of typechecking:

- values are members of types
- the type of an expression must include all the values that might possibly result from evaluating that expression at run-time

Requirements on each `if` expression:

- the type of the *`test`* expression must be `bool`
- the type of the result of the `if` must include whatever values might be returned from the `if`
  - the `if` might return the result of either *`e1`* or *`e2`*
  - ML's solution: *`e1`* and *`e2`* must have the same type, and that type is the type of the result of the `if` expression (other languages have more general solutions)

Let expression

An expression that introduces a new nested scope with local variable declarations

- unlike `{ ... }` statements in C, which don't compute results

General form:

$$\begin{array}{l}\textbf{let val } id_1 : type_1 = e_1 \\ \quad \ldots \\ \quad \textbf{val } id_n : type_n = e_n \\ \textbf{in} \\ \quad e_{body} \\ \textbf{end}\end{array}$$

- $type_i$ are optional; they'll be inferred from the e_i

Evaluates each e_i and binds it to id_i, in turn

- each e_i can refer to the previous $id_1..id_{i-1}$ bindings
- each id_i shadows any earlier/enclosing bindings of the same name

Evaluates e_{body} and returns its result as result of `let` expr

- e_{body} can refer to all the $id_1..id_n$ bindings

The id_i bindings (not values) disappear after e_{body} is evaluated

Example scopes

```
- val x = 3;
val x = 3 : int
- fun f(y:int):int =
=   let
=     val z = x + y
=     val x = 4
=   in
=     (let
=         val y = z + x
=      in
=         x + y + z
=      end)
=     + x + y + z
=   end;
val f = fn : int -> int
- val x = 5;
val x = 5 : int
- f x;
val it = 41 : int
```


"Statements"

For expressions that have no useful result, return empty tuple, of type `unit`:

```
- print "hi\n";
hi
val it = () : unit
```

Expression sequence operator: `;` (infix operator)

- evaluates both "arguments", returns second one
  - like C's comma operator

```
- val z = (print "hi "; print "there\n"; 3);
hi there
val z = 3 : int
```

Type inference for functions

Declaration of function result types can be omitted

- **infer** function result type from body expression result type

```
- fun max(x:int, y:int) =
=     if x >= y then x else y;
val max = fn : int * int -> int
```

Can even omit declarations of formal argument types

- infer based on how arguments are used in body
- constraint-based algorithm to do type inference

```
- fun max(x, y) =
=     if x >= y then x else y;
val max = fn : int * int -> int
```


Functions with many possible types

Some functions could be used on arguments of different types

Some examples:

`null`: can test an `int list`, or a `string list`, or ....

- in general, work on a list of any type *T*:
  `null: T list -> bool`

`hd`: similarly works on a list of any type *T*, and returns an element of that type:
  `hd: T list -> T`

`swap`: takes a pair of an *A* and a *B*, returns a pair of a *B* and an *A*:
  `swap: A * B -> B * A`

How to define such functions in a statically-typed language?

- in C: can't (or have to use casts)
- in C++: can use templates (but can't check separately)
- in Java, C#: use generic `Object` type, plus downcasts
- in ML: allow functions to have **polymorphic types**

Polymorphic types

A polymorphic type contains one or more **type variables**

- an identifier prefixed with a quote

E.g.

```
'a list
'a * 'b * 'a * 'c
{x:'a, y:'b} list * 'a -> 'b
```

A polymorphic type describes a set of possible types, where each type variable is replaced with some actual type

- each occurrence of a type variable must be replaced with the same type

```
'a * 'b * 'a * 'c
 ['a → int, 'b → string, 'c → real->real]
⇒
 int * string * int * (real->real)
```


Polymorphic functions

Functions can have polymorphic types:

```
null   : 'a list -> bool
hd     : 'a list -> 'a
tl     : 'a list -> 'a list
(op ::): 'a * 'a list -> 'a list
swap   : 'a * 'b -> 'b * 'a
```

To call a polymorphic function, must first **instantiate** the polymorphic type into some regular function type

- caller knows types of arguments
- can compute how to replace type variables so that the replaced function type matches the argument types
- derive type of result of call
- each call of a function instantiated independently

E.g. `hd [3,4,5]`

- actual argument type: `int list`
- polymorphic type of `hd`: `'a list -> 'a`
- replace `'a` with `int` (to make `'a list` match `int list`)
- instantiated type of `hd` for this call: `int list -> int`
- type of result of call: `int`

Polymorphic values

Non-functions can have polymorphic types, too:

```
nil: 'a list
```

Each reference to a polymorphic value finds the right instantiation for that use, separately from other references

E.g.

```
(3 :: 4 :: nil) :: (5 :: nil) :: nil
```


Polymorphism versus overloading

Polymorphic function:
same function usable for many different argument types, with uniform behavior

```
- fun swap(a,b) = (b,a);
val swap = fn : 'a * 'b -> 'b * 'a
```

Overloaded function:
different functions with same name but (possibly) unrelated behavior

Resolve overloading to particular function,
based on static argument types in ML

```
- 3 + 4;
val it = 7 : int

- 3.0 + 4.5;
val it = 7.5 : real

- (op +);  (* which +? default to int version *)
val it = fn : int*int -> int

- (op +):real*real->real;
val it = fn : real*real -> real
```


An awkward special case: equality types

The built-in = function tests for "structural" or value equality (not identity)

The = function is polymorphic over all types that "admit equality"

- any type except those containing reals or functions
- use `''a`, `''b`, etc. to stand for these equality types

```
- fun is_same(x, y) =
    if x = y then "yes" else "no";
val is_same = fn : ''a * ''a -> string
- is_same(3, 4);
val it = "no" : string
- is_same({l=[3,4,5],h=("a","b"),w=nil},
          {l=[3,4,5],h=("a","b"),w=nil});
val it = "yes" : string
- is_same(3.4, 3.4);
Error: operator and operand don't agree
[equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    is_same (3.4,3.4)
```


Loops, using recursion

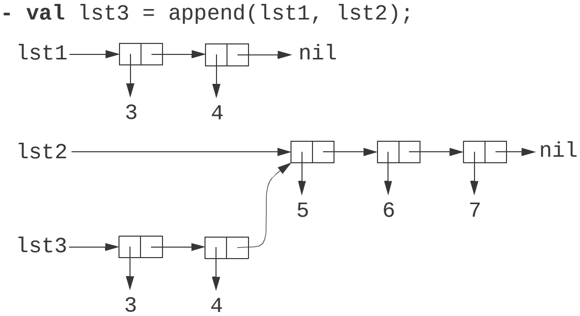
ML has no loop statement or expression

Instead, use recursion to compute a result

E.g., appending one list onto the front of another one
(non-destructively, since lists are immutable)

```
fun append(l1, l2) =
  if null l1
  then l2
  else hd l1::append(tl l1, l2)

- val lst1 = [3, 4];
- val lst2 = [5, 6, 7];
- val lst3 = append(lst1, lst2);
```



Tail recursion

Tail recursion: recursive call is last operation before returning

- can be implemented just as efficiently as iteration,
  in both time and space,
  since tail-caller isn't needed after callee returns

Some tail-recursive functions:

```
fun last(lst) =
  let val tail = tl lst in
    if null tail then
      hd lst
    else
      last tail
  end

fun includes(lst, x) =
  if null lst then
      false
  else if hd lst = x then
      true
  else
      includes(tl lst, x)
```

Is `append` tail-recursive?

Converting to tail-recursive form

Can often rewrite a non-tail-recursive function tail-recursively

- introduce a helper function
- the helper function has an extra accumulator argument
- the accumulator holds the partial result computed so far
- accumulator returned as full result when base case reached

This isn't tail-recursive:

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

This is:

```
fun fact(n) =
  let fun fact_helper(n, res) =
    if n <= 1 then
      res
    else
      fact_helper(n - 1, res * n)
  in
    fact_helper(n, 1)
  end
```


Pattern matching

Pattern-matching: a convenient syntax for
extracting components of compound values
(tuple, record, or list)

A pattern looks like an expression to build a compound value,
but with variable names in some places

- cannot use the same variable name more than once

Can use pattern in place of variable on l.h.s. of `val` binding

- binds any variable names in pattern to the corresponding
  subparts of the value on the r.h.s.

```
- val x = (false,17);
val x = (false,17) : bool*int

- val (a,b) = x;
val a = false : bool
val b = 17 : int

- val (root1, root2) = quad_roots(3.0,4.0,5.0);
val root1 = 0.786299647847 : real
val root2 = ~2.11963298118 : real
```


More patterns

```
- val [x,y] = 3::4::nil;
val x = 3 : int
val y = 4 : int

- val (x::y::zs) = [3,4,5,6,7];
val x = 3 : int
val y = 4 : int
val zs = [5,6,7] : int list
```

Constants (ints, bools, strings, chars, nil) can be patterns:

```
- val (x,true,3,"x",z) =
          (5.5,true,3,"x",[3,4]);
val x = 5.5 : real
val z = [3,4] : int list
```

If don't care about some component, can use a wildcard: _

```
- val (_::_::zs) = [3,4,5,6,7];
val zs = [5,6,7] : int list
```

Patterns can be nested, too

- orthogonality

Function argument patterns

Formal parameter of a `fun` declaration can be a pattern

```
- fun swap (a, b) = (b, a);
val swap = fn : 'a*'b -> 'b*'a
- fun swap2 x = (#1 x, #2 x);
val swap2 = fn : 'a*'b -> 'b*'a
- fun swap3 x =
    let val (a,b) = x in (b,a) end;
val swap3 = fn : 'a*'b -> 'b*'a

- fun best_friend
    {student={name=n,age=_},
     grades=_,
     best_friends={name=f,age=_}::_} =
              n ^ "'s best friend is " ^ f;
val best_friend = fn
  : {best_friends:{age:'a, name:string} list,
     grades:'b,
     student:{age:'c, name:string}}
```

Patterns allowed wherever **binding** occurs, orthogonally

Multiple cases

Often a function's implementation can be broken down into several different cases, based on the argument value

ML allows a single function to be declared via several cases

Each case identified using pattern-matching

- cases checked in order, until first matching case

```
- fun fib 0 = 0
    | fib 1 = 1
    | fib n = fib(n-1) + fib(n-2);
val fib = fn : int -> int

- fun null nil    = true
    | null (_::_) = false;
val null = fn : 'a list -> bool

- fun append(nil,  lst) = lst
    | append(x::xs,lst) = x :: append(xs,lst);
val append = fn : 'a list * 'a list -> 'a list
```

The function has a single type

⇒ all cases must have same argument and result types

Missing cases

What if we don't provide enough cases?

- ML gives a warning message "match nonexhaustive" when function is declared (**statically**)
- ML raises an exception "nonexhaustive match failure" if invoked and no existing case applies (**dynamically**)

```
- fun first_elem (x::xs) = x;
Warning: match nonexhaustive
   x :: xs => ...
val first_elem = fn : 'a list -> 'a

- first_elem [3,4,5];
val it = 3 : int

- first_elem [];
uncaught exception nonexhaustive match failure
```

How would you provide an implementation of this missing case?

- Unlike C, ML has no catch-all NULL pointer that could be returned

Exceptions

If get in a situation where you can't produce a normal value of the right type, then can raise an exception

- aborts out of normal execution
- can be handled by some caller
- reported as a top-level "uncaught exception" if not handled

Step 1: declare an `exception` that can be raised

```
- exception EmptyList;
exception EmptyList
```

Step 2: use the `raise` expression where desired

```
- fun first_elem (x::xs) = x
    | first_elem nil = raise EmptyList;
val first_elem = fn : 'a list -> 'a

- first_elem [3,4,5];
val it = 3 : int

- first_elem [];
uncaught exception EmptyList
```

Handling exceptions

Add handler clause to expressions to handle (some) exceptions raised in that expression

Syntax:

```
expr handle exn_name1 => expr1
          | exn_name2 => expr2
            ...
          | _ => exprn
```

- this is an expression;
  each $expr_i$ must return same type as *expr*

```
- fun second_elem l = first_elem (tl l);
val second_elem = fn : 'a list -> 'a

- (second_elem [3]
     handle EmptyList => ~1) + 5;
val it = 4 : int
```


Exceptions with arguments

Can have exceptions with arguments

```
- exception IOError of int;
exception IOError of int;

- (... raise IOError(-3) ...)
     handle IOError(code) => ... code ...;
```