Some thoughts on language

- “But if thought corrupts language, language can also corrupt thought.”
  
  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

Why study programming languages?

- Knowing many languages broadens thought
- Better ways to organize software
- In both existing and new languages
- Better ways to divide responsibilities among tools and humans
- To understand issues underlying language designs, debates, etc.
- Language design impacts software engineering, software quality, compilers & optimizations
- Some language tools can aid other systems
  - E.g., extensible/open but safe systems

Course overview (1/2)

- Part 1: functional languages
  - A practical example: ML
  - Other exposure: Scheme, Haskell
  - Theoretical foundations: lambda calculi, operational semantics, type theory
  - Project: a Scheme interpreter & type inferencer, implemented in ML

Course overview (2/2)

- Part 2: object-oriented languages
  - A practical example: Cecil
  - Other exposure: Self, Java/C#, EML
  - Theoretical foundations
  - Project: a Self interpreter & type checker, implemented in Cecil (maybe)

Course work

- Readings
- Weekly homework
  - Some moderate programming
  - Some paper exercises
- Midterm
- Final
Language Design Overview

Some language design goals
- Be easy to learn
- Support rapid (initial) development
- Support easy maintenance, evolution
- Foster reliable, safe software
- Foster portable software
- Support efficient software

Some means to those goals
- Simplicity
  - But what does “simple” mean?
- Readability
- Writability
- Expressiveness
- Well-defined, platform-independent, safe semantics

The problem
- 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 programming
  - Java, C#, Lisp, Scheme, ML, Smalltalk, Cecil, ...
- Scripting, macro languages
  - Sh, Perl, Python, Tcl, Excel macros, ...
- Specialized languages
  - SQL, LaTeX, PostScript, Unix regular expressions, ...

Main PL concepts (1/2)
- Separation of syntax, semantics, and pragmatics
- EBNF to specify syntax precisely
- Semantics is more important than syntax
- Pragmatics: programming style, intended use, performance model
- Control structures
  - Iteration, conditionals; exceptions
  - Procedures, functions; recursion
  - Message passing
  - Backtracking
  - Parallelism
Main PL concepts (2/2)

- Data structures, types
  - Atomic types: numbers, chars, bools
  - Type constructors: records, tuples, lists, arrays, functions, ...
  - User-defined abstract data types (ADTs); classes
  - Polymorphic/parameterized types
  - Explicit memory management vs. garbage collection
- Type checking
  - Static vs. dynamic typing
  - Strong vs. weak typing
  - Type inference
- Lexical vs. dynamic scoping
- Eager vs. lazy evaluation

Some good language design principles

- Strive for a simple, regular, orthogonal model
  - In evaluation, data reference, memory management, ...
  - E.g. be expression-oriented, reference-oriented
- Include sophisticated abstraction mechanisms
  - Define and name abstractions once then use many times
  - For control, data, types, ...
- Include polymorphic static type checking
- Have a complete & precise language specification
  - Full run-time error checking for cases not detected statically

Partial history of programming languages

History

- Designed as a Meta Language 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

Main features

- Expression-oriented
- List-oriented, garbage-collected heap-based
- Functional
  - Functions are first-class values
  - Largely side-effect free
- Strongly, statically typed
  - Polymorphic type system
  - Automatic type inference
- Pattern matching
- Exceptions
- Modules
- Highly regular and expressive
Interpreter interface

- **Read-eval-print loop**
  - **Read** input expression
  - Reading ends with semicolon (not needed in files)
  - Prompt indicates continuing expression on next line
  - **Evaluate** expression
  - \( \text{it} \) (re)bound to result, in case you want to use it again
  - **Print** result

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

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**: \( \sim, +, -, \times, / \)
  - **floor**, **ceil**, **trunc**, **round**
  - **true**, **false**, \( \wedge \), \( \vee \), **orelse**, **andalso**
  - **string**: \( "\text{I said } \backslash \text{hi}\backslash\text{in dir C:stuff\dir\n}" \)
  - **char**: \#"a", \#"\n"

- **Variables and binding**

```
val x = 6 : int
val y = x * x;
val y = 36 : int
val y = y + 1;
val y = 37 : int
```

- **Variable bindings cannot be changed!**
  - Variables can be bound again, but this shadows the previous definition

```
val y = y + 1;
val y = 37 : int (* a new, different y *)
```

- **Variables declared and initialized with a **val** binding**
  - val \( x : \text{int} = 6 \)
  - val \( x = 6 : \text{int} \)
  - val \( y : \text{int} = x * x \)
  - val \( y = 36 : \text{int} \)

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

```
val y = y + 1;
val y = 37 : int (* a new, different y *)
```

- **Variable types can be omitted**
  - They will be inferred by ML based on the type of the r.h.s.

```
val \( x : \text{int} * y : \text{int} \)
val \( x = 227: \text{int} \)
```

- **Strong, static typing**
  - **ML is statically typed**: it will check for type errors statically
  - When programs are entered, not when they're run

- **ML is strongly typed**: it will catch all type errors (a.k.a. it's type-safe)

  - But which errors are type errors?
  - Can have weakly, statically typed languages, and strongly, dynamically typed languages

Type errors

- **Type errors can look weird, given ML's fancy type system**

```
Error: unbound variable or constructor: \text{and}
Error: operator and operand don't agree
  operator domain: \text{int} * \text{int}
  operand:
  \text{int} * \text{real}
in expression:
  \text{y} * \text{4.5}
```

- **Records**

- **ML records are like C structs**
  - Allow heterogeneous element types, but fixed \# of elements

- **A record type**: \{ name: \text{string}, age: \text{int} \}
  - Field order doesn't matter

- **A record value**: \{ name="Bob Smith", age=20 \}
  - Can construct record values from expressions for field values

- **As with any value, can bind record values to variables**

```
val bob = \{ \text{name}=\"Bob Smith\", \text{age}=20 \};
val bob = \{ \text{name}=\"Bob Smith\"; \}
```

```
\{ \text{name}=\text{age=20}, \text{name}=\text{Bob Smith}; \}
\text{age=20}\text{.name}=\text{Bob Smith};
```

```
\text{age=20}\text{.name}\text{.string} = \text{Bob Smith};
```

Accessing parts of 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"} : {...}`
- Cannot assign/change 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)
  - Can construct tuple values from expressions for elements
    - as with any value, can bind tuple values to variables
      - `val joe = ("Joe Stevens", 25*num_jobs+10);`
      - `val joe = ("Joe Stevens",15) : string * int`

Accessing parts of tuples

- Can extract tuple fields using `#n` function
  - `val joe' = (#1(joe), #2(joe)+1);`
  - `val joe' = ("Joe Stevens",46) : string * int`
- Cannot assign/change a tuple’s components
  ⇒ another immutable data structure

Lists

- ML lists are built-in, singly-linked lists
  - homogeneous element types, but variable # of elements
- A list type: `int list`
  - in general: `T list`, for any type `T`
- A list value: `[3, 4, 5]`
- Empty list: `[]`
  - `null(lst): tests if lst is nil`
- Can create a list value using the `...` notation
  - `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`

Basic operations on lists

- Adding to the front allocates a new link; the original list is unchanged and still available
  - `lst1:
    - `val it = [3,4,5] : int list`
    - `lst2;
    - `val it = [2,3,4,5] : int list`
More on lists

- Lists can be nested:
  ```ml
  (3 :: nil) :: (4 :: 5 :: nil) :: nil;
  val it = [[3], [4, 5]] : int list list
  ```

- Lists must be homogeneous:
  ```ml
  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

- Look up the first ("head") element: `hd`
  ```ml
  hd(let1) = hd(let2);
  val it = 5 : int
  ```

- Extract the rest ("tail") of the list: `tl`
  ```ml
  - val let1 = tl(let1);  val let3 = [4, 5] : int list
  - val let4 = tl(let3);
  - tl(let4); (* or hd(let4) *)
    uncaught exception Empty
  ```

- Cannot assign/change 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
  - an example of orthogonal design
    ```ml
    {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 directly, as expressions
  - instead of using a sequence of statements to first declare (allocate named space) and then assign to initialize

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

- 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-style?

- Reference-oriented values are heap-allocated (logically)
  - scalar values like ints, reals, chars, bools, nil optimized

Garbage collection

- ML provides several ways to allocate & initialize new values
  - `{...}`...

- But it provides no way to deallocate/free values that are no longer being used

- Instead, it 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
  - dangling pointers impossible
  - storage leaks impossible
  - simpler programming
  - can be more efficient!
  - less ability to carefully manage memory use & reuse

- GCs exist even for C & C++, as free libraries

Functions

- Some function definitions:
  ```ml
  - 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
  ```

- Functions are values with types of the form
  ```ml
  Targ -> Tresult
  ```

- use tuple type for multiple arguments
- use tuple type for multiple results (orthogonality!)
  - binds tighter than ->

- Some function calls:
  ```ml
  - square(3); (* parens not needed! *)
  val it = 9 : int
  - swap(3,4, "billy", "bob"); (*parens needed*)
  val it = ("bobbilly",12) : string * int
  ```
Expression-orientation

- Function body is a single expression
  \[ \text{fun} \ square(x:\text{int}):\text{int} = x \times x \]
- not a statement list
- no return keyword
- Like equality in math
  - a call to a function is equivalent to its body, after substituting the actuals in the call for its formals

\[ \text{square}(3) \iff (x \times x)[x \to 3] \iff 3 \times 3 \]

- There are no statements in ML, only expressions
  - simplicity, regularity, and orthogonality in action
  - 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

\[ \text{fun} \ \text{max}(x:\text{int}, y:\text{int}):\text{int} = \]
\[ = \text{if} \ x \geq y \text{ then } x \text{ else } y; \]
\[ \text{val} \ \text{max} = \text{fn} : \text{int} \times \text{int} \to \text{int} \]

- Like ?:: operator 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 an 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

- A solution: at and ec must have the same type, and that type is the type of the result of the if expression

Let expression

- let: an expression that introduces a new nested scope with local variable declarations
- unlike \{ \ldots \} statements in C, which don't compute results
- General form:
  - let val id1 : type1 = e1...
  - val idn : typen = en in ebody end

- typei are optional; they'll be inferred from the ei
- Evaluates each ei and binds it to id1, in turn
- each ei can refer to the previous id1..idi-1 bindings
- Evaluates ebody and returns it as the result of the let expression
- The idi bindings disappear after ebody is evaluated
  - they're in a nested, local scope

Example scopes

- \text{val} x = 3;
- \text{val} x = 3 : \text{int}
- \text{fun} f(y:\text{int}) : \text{int} =
  \text{let} val z = x + y 
  = val x = 4 
  = in (let val y = z + x 
  = in x + y = z end) 
  = x + y = z 
  = end; 
- \text{val} f = \text{fn} : \text{int} \to \text{int}
- \text{val} x = 5;
- \text{val} x = 5 : \text{int}
- \text{f}(x);

Statements"

- For expressions that have no useful result, return empty tuple, of type unit
  - \text{print}("hi\n")
  - \text{val} x = 3 : \text{int}

- Expression sequence operator: ;
  - (an infix operator, like C's comma operator)
  - evaluates both "arguments", returns second one
  - \text{val} x = \text{print}("hi "); \text{print}("there\n"); 3; 
  - \text{hi there}
  - \text{val} x = 3 : \text{int}
Type inference for functions

- Declaration of function result type can be omitted
  - infer function result type from body expression result type
    
    \[
    \text{fun } \text{max}(\text{x:int, y:int}) = \\
    \begin{cases}
    \text{x} & \text{if } x \geq y \\
    \text{y} & \text{otherwise}
    \end{cases}
    \]
    
    \text{val max = fn : int * int -> int}

- Can even omit formal argument type declarations
  - infer all types based on how arguments are used in body

constraint-based algorithm to do type inference

\[
\text{fun } \text{max}(\text{x, y}) = \\
\begin{cases}
\text{x} & \text{if } x \geq y \\
\text{y} & \text{otherwise}
\end{cases}
\]

\text{val max = fn : int * int -> int}

Functions with many possible types

- Some functions could be used on arguments of different types
  - Some examples:
    - \text{null} can test an \text{int} list, or a \text{string} list, or... 
    - in general, work on a list of any type \( T \): 
      - \text{null : list -> bool} 
      - \text{hd:} similarly works on a list of any type \( T \), and returns an element of that type: 
        - \text{hd : list \to T} 
      - \text{swap:} takes a pair of an \( a \) and a \( b \), returns a pair of a \( b \) and an \( a \): 
        - \text{swap : A * B \to 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 ML: allow functions to have \text{polymorphic types}

Polymorphic types

- A polymorphic type contains one or more type variables
  - an identifier starting with a quote
    - \( 'a \text{ list} \)
    - \( 'a * 'b * 'a \)
    - \( (x:'a, y:'b) \text{ list} * 'a \to 'b \)
- A polymorphic type describes a set of possible types, where each type variable is replaced with some type
  - each occurrence of a type variable must be replaced with the same type
    - \( ['a * 'b * 'a * 'c] \text{ list} * 'a \to 'b \text{ list} \)
    - \( ['a * 'b * 'a * 'c] \to ['a \to int] \text{ list} * ['b \to int] * ['c \to (real \to real)] \)

Polymorphic functions

- Functions can have polymorphic types:
  - \text{null : 'a list \to bool}
  - \text{hd : 'a list \to 'a}
  - \text{tl : 'a list \to 'a list}
  - \text{(op ::) : 'a * 'a list \to 'a list}
  - \text{swap : 'a * 'b \to 'b * 'a}

Calling polymorphic functions

- When calling a polymorphic function, need to find the instantiation of the polymorphic type into a regular type that’s appropriate for the actual arguments
  - caller knows types of actual arguments
  - can compute how to replace type variables so that the replaced function type matches the argument types
  - derive type of result of call

Example: \( \text{hd}([3,4,5]) \)
  - type of argument: \text{int list}
  - type of function: \text{'a list \to 'a}
  - replace \( 'a \) with \text{int} to make a match
  - instantiated type of \text{hd} for this call: \text{int list \to int}
  - type of result of this call: \text{int}

Polymorphic values

- Regular values can polymorphic, too
  - \text{nil : 'a list}

  - Each reference to \text{nil} finds the right instantiation for that use, separately from other references
    - \text{nil \to 'a list}
    - \text{3 :: 4 :: nil \to ('a list) \to nil \to nil}
Polymorphism versus overloading

- **Polymorphic** function: same function usable for many different types
  - Fun `swap(i, j) = (j, i);`
  - Val `swap = fn : 'a * 'b -> 'b * 'a`

- **Overloaded** function: several different functions, but with same name
  - The name is overloaded
  - A function of type `int*int->int`
  - A function of type `real*real->real`

- **Resolve** overloading to particular function based on:
  - Static argument types (in ML)
  - Dynamic argument classes (in object-oriented languages)

Example of overload resolution

- `3 + 4;`
  - Val it = 7 : int
- `3.0 + 4.5;`
  - Val it = 7.5 : real
- `(op +); (* which? default to int *)`
  - Val it = fn : int*int -> int
- `(op +):real*real->real;`
  - Val it = fn : real*real -> real

Equality types

- **Built-in** = is polymorphic over all types that "admit equality"
  - I.e., 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 `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 looping statement or expression
- Instead, use **recursion** to compute a result

  - Fun `append(lst1, lst2) =`
    - If `null(lst1)` then `lst2`
    - Else `hd(lst1) :: append(tl(lst1), lst2)`

  - 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)`

- `append?`

Converting to tail-recursive form

- Can often rewrite a recursive function into a tail-recursive one
  - Introduce a helper function (usually nested)
  - 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 `fact(n-1) * n`

  - This is:
    - Fun `fact_helper(m, res) =`
      - If `m = 1` then `res`
      - Else `fact_helper(m-1, res * m)`
      - In `fact_helper(0, 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 to be bound in some places.
- Cannot use the same variable name more than once.
- Use pattern in place of variable on l.h.s. of `val` binding.
- Use pattern in place of variable on r.h.s. of `val` binding.
- Anywhere `val` can appear: either at top-level or in `let` binding orthogonality & regularity.
- List 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.
- Function argument patterns
  - Formal parameter of a `fun` declaration can be a pattern
    - `fun swap (i, j) = (j, i);`
    - `val swap = fn : ('a * 'b) -> ('b * 'a)`
    - `fun swap2 p = (#2 p, #1 p);`
    - `val swap2 = fn : ('a * 'b) -> ('b * 'a)`
    - `fun swap3 p = let val (a,b) = p 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='b} list, grades='c, student:{age='d, name='e} list} -> 'b`.
- Multiple cases
  - Often a function's implementation can be broken down into several different cases, based on the argument value.
  - Each case identified using pattern-matching.
  - Cases checked in order, until first matching case.
  - Functions have a single type.
  - List patterns:
    - `val [x,y] = 3::4::nil;`
    - `val x = 3 : int`
    - `val y = 4 : int`
  - 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.
- 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`
      - `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 for `nil`?
      - `fun first_elem nil = _ | first_elem xs = case`
      - `| first_elem nil = ???`
- 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
      - `fun first_elem (x::xs) = x | first_elem nil = raise EmptyList;`
    - Step 2: use the raise expression where desired.
      - `fun first_elem (x::xs) = x | first_elem nil = raise EmptyList;`
      - `val 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
  
  ```ml
  expr handle exn_name1 => expr1
  | exn_name2 => expr2
  ... 
  | exn_name_n => expr_n
  ```

  - if `expr` raises `exn_name`, then evaluate and return `expr` instead

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

Exceptions with arguments

- Can have exceptions with arguments
  ```ml
  exception IOError of int;
  ```

```ml
raise IOError(-3) ...
```

Type synonyms

- Can give a name to a type, for convenience
  - name and type are equivalent, interchangeable
  ```ml
  type person = {name:string, age:int};
  ```

```ml
val p:person = {name="Bob", age=18};
val p2 = p;
```

Polymorphic type synonyms

- Can define polymorphic synonyms
  ```ml
  type 'a stack = 'a list;
  ```

```ml
val emptyStack = \[
```

Datatypes

- Users can define their own (polymorphic) data structures
  - a new type, unlike type synonyms

```ml
datatype sign = Positive | Zero | Negative;
```
Datatypes and pattern-matching

- Constructor values can be used in patterns, too
  - fun signum(Positive) = 1
  - | signum(Negative) = -1;
  val signum = fn : sign -> int

Datatypes with data

- Each constructor can have data of particular type stored with it
- constructors with data are functions that allocate & initialize new values with that "tag"
  - datatype LiteralExpr = Nil | Integer of int | String of string;
  datatype LiteralExpr = Nil | Integer | String
  val it = Nil : LiteralExpr
  - val it = Integer 3 : LiteralExpr
  - val it = String "xyz" : LiteralExpr

Pattern-matching on datatypes

- The only way to access components of a value of a datatype is via pattern-matching
- Constructor "calls" can be used in patterns to test for and take apart values with that "tag"
  - fun toString(Nil) = "nil"
  - | toString(Integer(i)) = Int.toString(i)
  - | toString(String(s)) = "" ^ s ^ ""
  val toString = fn : LiteralExpr -> string

Recursive datatypes

- Many datatypes are recursive: one or more constructors are defined in terms of the datatype itself
  - datatype Expr = Nil | Integer of int | String of string | Variable of string | Tuple of Expr list | BinOpExpr of {arg1:Expr, operator:string, arg2:Expr} | FnCall of {function:string, arg:Expr};
  datatype Expr = ...
  val e1 = Tuple [Integer 3, String "hi"] : Expr
  val e2 = FnCall { function="f", arg=Tuple [BinOpExpr {arg1=Integer(3), operator="+", arg2=Variable "x"}, String "hi"] } : Expr

Another example Expr value

(* If x = "hi" *)
- val e2 =
  - fn("",&"",&"", &"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&"",&")

Recursive functions over recursive datatypes

- Often manipulate recursive datatypes with recursive functions
  - pattern of recursion in function matches pattern of recursion in datatype
  - fun toString(Nil) = "nil"
  - | toString(Integer(i)) = Int.toString(i)
  - | toString(String(s)) = "" ^ s ^ ""
  - | toString(Variable(name)) = name
  - | toString(Tuple(elems)) = "(" ^ listToString(elems) ^ ")"
  - val toString = fn : Expr -> string
Mutually recursive functions and datatypes

- If two or more functions are defined in terms of each other, recursively, then must be declared together, and linked with and
  
  ```
  fun toString(\ldots) = \ldots listToString \ldots
  and listToString(\ldots) = \ldots
  ```

- If two or more mutually recursive datatypes, then declare them together, linked by and
  
  ```
  datatype Stmt = \ldots Expr \ldots
  and Expr = \ldots Stmt \ldots
  ```

A convenience: record pattern syntactic sugar

- Instead of writing \{a=a, b=b, c=c\} as a pattern, can write \{a,b,c\}
  
  ```
  E.g.
  ...
  BinOpExpr\{arg1,operator,arg2\} ...
  ```

- is short-hand for
  
  ```
  ...
  BinOpExpr\{arg1=arg1, operator=operator, arg2=arg2\} ...
  ```

Polymorphic datatypes

- Datatypes can be polymorphic
  
  ```
  datatype 'a List = Nil
  | Cons of 'a * 'a List;
  datatype 'a List = Cons of 'a * 'a List |
  val lst = Cons(3, Cons(4, Nil));
  val list = fn \ldots
  ```

- Other definitions of empty, store, fetch, etc. don’t clash
- Common names can be reused by different structures

Modules for name-space management

- A file full of types and functions can be cumbersome to manage
- Would like some hierarchical organization to names
- Modules allow grouping declarations to achieve a hierarchical name-space
- ML structure declarations create modules
  
  ```
  structure Assoc_List = struct
  type ('a,'b) assoc_list = ('a*'b) list
  val empty = nil
  fun store(alist, key, value) = ...
  val fetch(alist, key) = ...
  end
  ```

- Other definitions of empty, store, fetch, etc. don’t clash
- Common names can be reused by different structures

Using structures

- To access declarations in a structure, can use dot notation
  
  ```
  val league = Assoc_List.store(league, "Mariners", \{..\});
  ```

- Other definitions of empty, store, fetch, etc. don’t clash
- Common names can be reused by different structures

The open declaration

- To avoid typing a lot of structure names, can use the open struct_name declaration to introduce local synonyms for all the declarations in a structure
- Usually in a let, local, or within some other structure
Modules for encapsulation

- Want to hide details of data structure implementations from clients, i.e., data abstraction
- simplify interface to clients
- allow implementation to change without affecting clients
- In C++ and Java, use public/private annotations
- In ML:
  - define a signature that specifies the desired interface
  - specify the signature with the structure declaration
  - E.g., a signature that hides the implementation of assoc_list:
    ```ml
    signature ASSOC_LIST = sig
    type ('k,'v) assoc_list   (* no rhs! *)
    val empty : ('k,'v) assoc_list
    val store : ('k,'v) assoc_list * 'k * 'v -> ('k,'v) assoc_list
    val fetch : ('k,'v) assoc_list * 'k -> 'v
    end;
    signature ASSOC_LIST = sig ... end
    ```

Specifying the signatures of structures

- Specify desired signature of structure when declaring it:
  ```ml
  structure Assoc_List :> ASSOC_LIST = struct
  type ('k,'v) assoc_list = ('k*'v) list
  val empty = nil
  fun store(alist, key, value) = ...
  fun fetch(alist, key) = ...
  fun helper(...) = ...
  end;
  ```
- The structure's interface is the given one, not the default interface that exposes everything

Hidden implementation

- Now clients can't see implementation, nor guess it
  ```ml
  val teams = Assoc_List.empty;
  val teams = "Mariners"::"Yankees"::teams;
  Error: operator and operand don't agree
  operator: string * string list
  operand: string * (''Z,'Y) Assoc_List.assoc_list
  - Assoc_List.helper(...);
  Error: unbound variable helper in path Assoc_List.helper
  - type Records = (string,...) Assoc_List.assoc_list;
    type Records = (string,...) Assoc_List.assoc_list
    fun sortStandings(nil:Records):Records = nil
    | sortStandings(pivot::rest) = ...;
    Error: pattern and constraint don't agree
    pattern: 'Z list
    constraint: Records
    in pattern nil : Records
  ```