Part I

Introduction
ML is another functional programming language that evolved from the LISP community, adding several important features.

- **Prettier concrete syntax** — ML programs support infix operators and statement separators like in C/C++, eliminating Scheme/LISP’s heavy use of full parenthesization. However, this eliminates Scheme/LISP’s advantage of naturally processing programs as data.
- **Strong typing** — type checking is a very helpful safety feature in software engineering, like in C/C++.
- **Type inference** — usually, you can omit the declaration of the type of a variable, because ML can automatically infer what the type should be from the context of the expression the variable occurs in. So the convenience and ease-of-use of ML is closer to Scheme/LISP where you never have to declare types of variables. (cf. C++0x auto)
- **Patterns** — built-in pattern matching allows a declarative style of programming that can be very clean.
Standard ML or SML is a dialect of ML that resulted from an international community standardization process. Consequently, there are many implementations of SML. A few other dialects still exist. Of particular note, CAML and its object-oriented extension O’Caml are very efficient implementations rivalling C++, that have a significant following.
SML (Standard Meta Language)

SML (/usr/local/sml/bin/sml) supports:

- **Higher-Order Functions**: composite functions in math. e.g. \( f(g(h(x))) \)
- **Abstract Data Types**: type \( = \) data + functions (as in OOP).
- **Polymorphism**: functions, data types (c.f. template in C++).
- **Strong Typing**: Every expression has a type which can be determined at compile time. (c.f. C++ is not. e.g. virtual function)
- **Static Scope**: All identifier references resolved at compile time.
- **Rule-Based Programming**: Actions are selected through if-then pattern-matching rules (c.f. AI language like Prolog).
- **Type-Safe Exception Mechanism**: to handle unusual situations arising at run-time. e.g. division-by-zero.
- **Modules**: an ML program \( = \) a set of interdependent modules glued together using functors.
Part II

Types, Values, Patterns
<table>
<thead>
<tr>
<th>TYPE</th>
<th>SYMBOL</th>
<th>EXAMPLE</th>
<th>OPERATIONS/TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
<td>()</td>
<td>—</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
<td>true, false</td>
<td>not, andalso, orelse</td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
<td>2, 0, 87</td>
<td>∼, +, -, ∗, div, mod</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>1.3, 3E2</td>
<td>∼, +, -, ∗, /</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>&quot;hello&quot;</td>
<td>∧</td>
</tr>
<tr>
<td>tuple</td>
<td>(...)</td>
<td>(1, &quot;yes&quot;, 2.5)</td>
<td>int<em>string</em>real</td>
</tr>
<tr>
<td>list</td>
<td>[...]</td>
<td>[3, 8, 1, 9]</td>
<td>int list</td>
</tr>
<tr>
<td>record</td>
<td>{...}</td>
<td>{name=&quot;Eagle&quot;, dead=true}</td>
<td>{name:string,dead:bool}</td>
</tr>
</tbody>
</table>
- **unit** is similar to **void** in C. It is used
  - whenever an expression has no interesting value.
  - when a function is to have no arguments.

- Negative **int** or **real** values are denoted using the unary operator \( \sim \) instead of the usual minus sign.

- Integer division uses **div** and **mod**, and real number division uses \(/\).

- **NO** implicit coercion!

- The boolean operators **andalso** and **orelse** perform short-circuit evaluations: i.e.
  - \( E_1 \text{ andalso } E_2 \Rightarrow \) will NOT evaluate \( E_2 \) if \( E_1 \) is **false**.
  - \( E_1 \text{ orelse } E_2 \Rightarrow \) will NOT evaluate \( E_2 \) if \( E_1 \) is **true**.
Example: int/real

- ();
  val it = () : unit

- 5 + 13;
  val it = 18 : int

- ~5 + 13;
  val it = 8 : int

- floor(123.6);
  val it = 123 : int

- floor(~123.6);
  val it = ~124 : int
Example: String

- "Hong"^" "^"Kong";
val it = "Hong Kong" : string

- size "Hong Kong";
val it = 9 : int

- size "Hong"^" "^"Kong";
stdIn:69.1-69.23 Error:
   operator and operand don’t agree [tycon mismatch]
   operator domain: string * string
   operand:       int * string
   in expression: size "Hong" ^ " "

- size("Hong"^" "^"Kong");
val it = 9 : int
Example: Type Checking

- \(5/6;\)
  stdIn:50.2 Error: overloaded variable not defined at type symbol: /
  type: int

- \(\text{real}(5)/6;\)
  stdIn:1.1-49.6 Error: operator and operand don’t agree
  operator domain: real * real
  operand: real * int
  in expression:
    real 5 / 6

- \(\text{real}(5)/\text{real}(6);\)
  val it = 0.8333333333333 : real
Example: Boolean Expression

- if 2=3 then "don’t worry" else "be happy";
  val it = "be happy" : string

- if "don’t worry"="be happy" then 1 else 2;
  val it = 2 : int

- if 2=3 then "don’t worry" else 4;
  stdIn:1.1-61.3 Error: types of rules don’t agree [literal]
  earlier rule(s): bool -> string
  this rule:        bool -> int
  in rule:         false => 4

- “if<bool-exp> then<then-exp> else<else-exp>” always come together; and its value is that of <then-exp> if <bool-exp> is true, otherwise that of <else-exp>.

- <then-exp> and <else-exp> must match in their types.
- (4, true, "cat");
val it = (4,true,"cat") : int * bool * string
- (if 3=8 then "X" else "Y", 9.5/0.5, 5 div 2);
val it = ("Y",19.0,2) : string * real * int
- (14 mod 3, not false) = (1+1, true);
val it = true : bool

- #2("for", "your", "info");
val it = "your" : string

- Ordered $n$-tuple: $(e_1, e_2, \ldots, e_n)$.
- Like vector in Scheme.
- The $n$ expressions may be of mixed types.
- 2 $n$-tuples are equal if their corresponding components are equal.
- “#” is the item selection operator.
List

- Empty list: \texttt{nil} or [ ];
- \texttt{nil} : ‘a list \Rightarrow a polymorphic object.
- \([e_1, e_2, \ldots, e_n]\) is an abbreviation for \(e_1::e_2::\ldots::e_n::\texttt{nil}\).
- \(::\) is the list constructor pronounced as “cons”.
- \(::\) is an infix operator which is right associative.
- \(<\texttt{new-list}> = <\texttt{item}>::<\texttt{list}>>.

\[
1::2::3::\texttt{nil} = 1::(2::(3::\texttt{nil})) \\
= 1::(2::[3]) \\
= 1::[2,3] \\
= [1,2,3]
\]

- Equality on 2 lists is item-by-item.
List Operators

- **cons operator:** :: : ‘a item * ‘a list → ‘a list
- **head operator:** hd() : ‘a list → ‘a item
- **tail operator:** tl() : ‘a list → ‘a list
- **append operator:** @ : ‘a list * ‘a list → ‘a list
List Examples

- `hd([1,2,3,4]);`
  val it = 1 : int

- `tl([1,2,3,4]);`
  val it = [2,3,4] : int list

- `hd([1,2,3,4]):tl([1,2,3,4]);`
  val it = [1,2,3,4] : int list

- `[5,6]@tl([1,2,3,4]);`
  val it = [5,6,2,3,4] : int list
- c.f. **struct** in C.
- Syntax: \{ label_1 = E_1, label_2 = E_2, \ldots \}
- Order does NOT matter since the fields are labelled.
- Tuples are actually short-hands for records.
  \((E_1, E_2, E_3) = \{ 1=E_1, 2=E_2, 3=E_3 \}\)

- \{name="bird", age=5, dead=true\};
  val it = \{age=5,dead=true,name="bird"\}
    : \{age:int, dead:bool, name:string\}

- \{name="bird", age=5, dead=true\}
  = \{age=5, dead=true,name="bird"\};
  val it = true : bool
Identifiers

BNF for alphanumerical identifiers:

<Id> ::=<First_Char><Other_Chars>
<First_Char> ::= [A-Z]|[a-z]|’
<Other_Chars> ::= <empty>|<Other_Char><Other_Chars>
<Other_Char> ::= [A-Z]|[a-z]|[0-9]|[_]

BNF for symbolic identifiers:

<Id> ::= <S_Char>|<S_Char><Id>
<S_Char> ::= [+-/*<>]=!@#$%^‘~\$?:]

- Disallow mixing the 20 symbols with alphanumerical characters.
- ’<Other_Char> are alpha variables ONLY used for data types.
- Symbolic identifiers should be used for user-defined operators.
Identifiers: Value Binding

Syntax: \texttt{val < identifier > } \texttt{=} \texttt{< expression >};

- \texttt{val a\_df} = 3+2; (* c.f. const int a\_df = 3+2; in C++ *)
\texttt{val a\_df} = 5 : int

- \texttt{val a'a} = "Albert"\textasciitilde" "\textasciitilde"Einstein";
\texttt{val a'a} = "Albert Einstein" : string

- \texttt{val a1b2} = 2;
\texttt{val a1b2} = 2 : int

- \texttt{val +++$$} = 9*3; (* may hold integral value *)
\texttt{val +++$$} = 27 : int

- \texttt{+++$$} + \texttt{+++$$}; (* Though you don’t want to do that *)
\texttt{val it} = 54 : int
Pattern Matching

- Pattern matching with **tuples**

  - val (left, right) = ("Einstein", 4);
    val left = "Einstein" : string
    val right = 4 : int

- Pattern matching with **lists**

  - val x::y = [5,6,7,8];  (* [5,6,7,8] = 5::[6,7,8] *)
    val x = 5 : int
    val y = [6,7,8] : int list

- Pattern matching with **records**

  - val {flag=y,count=x} = {count=2,flag=true};
    val x = 2 : int
    val y = true : bool
The **wildcard pattern** «_» (underscore symbol) may be used for terms that you don’t care in pattern matching.

- `val (left,_) = ("Einstein", 4);`
  
  val left = "Einstein" : string

- `val _::a = [1,2,3];`
  
  val a = [2,3] : int list

- `val x::_::z = [[1,2],[3,4],[7,9],[0,0]];`
  
  val x = [1,2] : int list
  
  val z = [[7,9],[0,0]] : int list list
Identifiers cannot duplicate in various parts of a pattern.

- `val (x, x::y) = (3, [3,4,5]);`  
  `stdIn:1.1-287.4 Error: duplicate variable in pattern(s): x`

- `val (x, x) = (3,3);`  
  `stdIn:1.1-279.7 Error: duplicate variable in pattern(s): x`
Part III

SML Functions
Functions: It is “fun”

- **Syntax:** `fun <identifier> (<parameter-list>) = <expression>;

- Parameter passing method: **Call-By-Value.**

```plaintext
- fun square(x) = x*x;
val square = fn : int -> int

- fun square x = x*x; (* parentheses are optional *)
val square = fn : int -> int

- square 4;
val it = 16 : int

- fun first (x,y) = x;  first (3, "foo");
val first = fn : 'a * 'b -> 'a
val it = 3 : int
```
Each identifier, variable or function, has a type.

**Function**: $<\text{domain type}> \rightarrow <\text{range type}>$

- Argument type may be explicitly specified with :$<\text{type}>$.
  - e.g. A function whose input is a **real** number and which returns a **real** number:

  - fun f_square(x: real) = x*x;
  val f_square = fn : real -> real

  - fun f_square(x):real = x*x; (* Another way *)

- A function whose domain type is a tuple ('a type, 'b type) and whose range type is 'a.

  - fun first (x,y) = x;
  val first = fn : 'a * 'b -> 'a
More Complex Functions

- Defined with boolean expressions.

  - fun greater(x,y) = if x > y then x else y;

  - fun factorial x = if x = 0
    = then 1 (* Initial ‘‘=’’ is continuation symbol *)
    = else x*factorial(x-1);

- Defined by enumerating ALL cases with pattern matching (⇒ more readable).

  - fun factorial 0 = 1
    | factorial x = x * factorial(x-1);
Functions: Bug

When functions are defined by case analysis, SML issues a warning or an error if

- Not all cases are covered.

  - fun myhead(head::tail) = head;
  stdIn:266.1-266.30 Warning: match nonexhaustive
  head :: tail => ...
  val myhead = fn : 'a list -> 'a

- A case is redundant because of earlier cases.

  - fun nonsense(_ _) = 3 | nonsense(0) = 5;
  stdIn:275.1-275.47 Error: match redundant
    _ => ... 
    --> 0 => ...
- **Type System**: for a language is a set of rules for associating a type with an expression in the language.
- **Type Inference**: to deduce the type of an expression
- **Basic Rule**: if \( f : A \rightarrow B \), and \( a \) has type \( A \) then \( f(a) \) must be of type \( B \).

Whenever possible, ML **automatically** infers the type of an expression.
Types of operands and results of arithmetic operators must agree.

\[ 4.8/(a - b) \]

Types of operands of comparison operators must agree.

\[ x = 1; \quad x < y \]

For the **if-then-else** expression (not statement!), then-expression and else-expression must be of the same type.

\[ \text{if } x > 1 \ \text{then } y \ \text{else } z \]

Types of actual and formal parameters of a function must agree.

\[ \text{fun } g(x) = 2 \times x; \quad g(5); \]
Functions **taking** functions as arguments:

- fun square x = x*x; fun twice x = 2*x;

- fun apply5 f = f 5; apply5 square;
  val apply5 = fn : (int -> 'a) -> 'a
  val it = 25 : int

- fun apply f x = f(twice(x)); apply square 3;
  val apply = fn : (int -> 'a) -> int -> 'a
  val it = 36 : int

- fun first x y = x; first 2 "foo";
  val first = fn : 'a -> 'b -> 'a
  val it = 2 : int
Function application is left-associative.

Thus, \((\text{first } x \ y) = ((\text{first } x) \ y)\).

Operator \(\rightarrow\) is right-associative.

Thus, \('a \rightarrow 'b \rightarrow 'a = 'a \rightarrow ('b \rightarrow 'a)'\).

i.e. first() has domain type = 'a, range = 'b \(\rightarrow\) 'a.

i.e. first() takes an 'a value and returns another function which takes a 'b value and returns an 'a value.
Functions **returning** function:

- fun sq_or_twice x = if x > 0 then square else twice;
  val sq_or_twice = fn : int -> int

- (sq_or_twice 2) 5;
  val it = 25 : int

- sq_or_twice 2;
  val it = fn : int -> int
fun H f x = f x

\[
\begin{align*}
H f &= g \\
g x &= y \\
f x &= y
\end{align*}
\]

\[
\begin{align*}
type(H) &= type(f) \rightarrow type(g) \\
type(g) &= type(x) \rightarrow type(y) \\
type(f) &= type(x) \rightarrow type(y)
\end{align*}
\]

Let type(x) = 'a and type(y) = 'b, then

\[
\begin{align*}
type(g) &= type(f) = 'a \rightarrow 'b \\
type(H) &= (\ 'a \rightarrow 'b \ ) \rightarrow (\ 'a \rightarrow 'b )
\end{align*}
\]
fun H f x = G(f x) where type(G) = 'a \rightarrow 'b.

\[
\begin{align*}
H f &= g \\
g x &= y \\
f x &= z \\
G z &= y
\end{align*}
\]

\[
\begin{align*}
type(H) &= type(f) \rightarrow type(g) \\
type(g) &= type(x) \rightarrow type(y) \\
type(f) &= type(x) \rightarrow type(z) \\
type(G) &= type(z) \rightarrow type(y)
\end{align*}
\]

\[
\equiv 'a \rightarrow 'b
\]

Let \(type(x) = 'c\), then

\[
\begin{align*}
type(f) &= 'c \rightarrow 'a \\
type(g) &= 'c \rightarrow 'b \\
type(H) &= (‘c \rightarrow ‘a) \rightarrow (‘c \rightarrow ‘b)
\end{align*}
\]
In general, a function on list must deal with the 2 cases:

- [] or nil
- head::tail

- fun len([]) = 0 | len(x::tail) = 1 + len(tail);

- fun sum([]) = 0 | sum(x::tail) = x + sum(tail);

- fun mean L = sum L div len L;

- mean [1,2,3];
val it = 2 : int

- fun append([], L2) = L2
  | append(x::tail, L2) = x::append(tail, L2);

- append([3,5], [9,8,7]);
val it = [3,5,9,8,7] : int list
The built-in `map()` has 2 arguments: a function `f()` and a list. It applies function `f()` to each element of the list.

```
fun map f [ ] = [ ]
| map f (head::tail) = (f head)::(map f tail);
```

- Type of list: `'a list`
- Type of `f`: `'a → 'b`
- Type of `map`: `('a → 'b) → 'a list → 'b list`
- fun odd x = (x mod 2) = 1;
val odd = fn : int -> bool

- map odd [1,2,3];
val it = [true,false,true] : bool list

- map odd;
val it = fn : int list -> bool list

- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
List Function: filter

- filter applies a boolean test function to each element of a list, removing the element should the test fail.

```haskell
fun filter f [] = []
  | filter f (head::tail) = if (f head)
      then head::(filter f tail)
      else (filter f tail);
```

- filter odd [1,2,3,4,5];
  val it = [1,3,5] : int list

- filter;
  val it = fn : ('a -> bool) -> 'a list -> 'a list

- filter odd;
  val it = fn : int list -> int list
List Function: reduce

- reduce accumulates a result from a list.

```haskell
fun reduce f [ ] v = v |
   reduce f (head::tail) v = f head (reduce f tail v);
```

- reduce add [1,2,3,4,5] 0;
  val it = 15 : int

- reduce;
  val it = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b

- reduce add;
  val it = fn : int list -> int -> int

- reduce add [1,2,3,4,5];
  val it = fn : int -> int
- fun reverse_([], L2) = L2
  | reverse_(x::tail, L2) = reverse_(tail, x::L2);
- fun reverse L = reverse_(L, []);

- reverse ["D","O","G"]; val it = ["G","O","D"] : string list

- rev: 'a list → 'a list, is SML’s built-in operator to do that.

- rev ["D","O","G"]; val it = ["G","O","D"] : string list
Anonymous Functions

Syntax: \textbf{fn} \langle\text{formal parameter}\rangle \Rightarrow \langle\text{body}\rangle

- An \textit{anonymous function} is a function without a name.
- Used when only a locally defined function is needed.

\begin{itemize}
  \item map (fn x => x*x) [2,3,4];
  val it = [4,9,16] : int list

  \item map (fn (x,_) => x) [(1,2), (3,4), (5,6)];
  val it = [1,3,5] : int list
\end{itemize}
Functions as Values

Functions are the first-class objects in SML, they can be input as arguments, returned as return-values, and also created as values.

- \texttt{val square = fn x => x*x; square 4;}
  \texttt{val square = fn : int -> int}
  \texttt{val it = 16 : int}

- \texttt{val f = square; f 4;}
  \texttt{val f = fn : int -> int}
  \texttt{val it = 16 : int}

- \texttt{val g = map square;}
  \texttt{val g = fn : int list -> int list}

- \texttt{g [1,2,3,4];}
  \texttt{val it = [1,4,9,16] : int list}
Composite Functions

Given: \( f: 'b \rightarrow 'c \) and \( g: 'a \rightarrow 'b \).

Define a new function: \( h(x) = f \circ g(x) \equiv f(g(x)) : 'a \rightarrow 'c \).

i.e first apply function \( g() \) to an input \( x \) of \( 'a \) type, returning a value of \( 'b \) type, which is then piped into function \( f() \) to give the final result of \( 'c \) type.

- fun square x = x*x;       fun twice x = 2*x;
val square = fn : int -> int
val twice = fn : int -> int

- val sq_twice = square o twice; (* Use val NOT fun *)
val sq_twice = fn : int -> int

- sq_twice 3;
val it = 36 : int
fun f(x) = if x = 1 then 1 else x*f(x-1);

f(1+1) = f(2)
  = if 2 = 1 then 1 else 2 * f(2-1)
  = 2 * f(1)
  = 2 * { if 1 = 1 then 1 else 1 * f(1-1) }
  = 2 * 1
  = 2

Actual parameters are evaluated before they are passed to functions ⇒ Call-By-Value.
f(1+1) = if (1+1) = 1 then 1 else (1+1) * f((1+1)-1)
= if 2 = 1 then 1 else (1+1) * f((1+1)-1)
= (1+1) * f((1+1)-1)
= 2 * f((1+1)-1)
= 2 * { if (((1+1)-1) = 1 then 1 else
        ((1+1)-1) * f(((1+1)-1) - 1) }
= 2 * 1
= 2

Actual parameters are evaluated only when they are needed.
Eager vs. Lazy Evaluation

- Give **same** result if the execution **terminates**.
- But consider the following 2 examples:

  ```c
  if X = 0 or Y/X > 5 then ... else ...;
  
  X + (Y == 0 ? 2 : 4/Y);
  ```
For function application:

- **eager evaluation** of actual parameters

For boolean expression:

- **short-circuit evaluation** = **lazy evaluation**

$E_1 \textbf{ or } E_2$ actually is a function $\textbf{or}(E_1, E_2)$. ML’s eager evaluation of actual parameters may not give the same result as required by short-circuit evaluation.

$\Rightarrow$ a new operator **orelse**. (same for **andalso**)

Prof. Dekai Wu, HKUST (dekai@cs.ust.hk)  COMP3031 (Fall 2012)
Creating New Infix Operators

Left-associative: `infix <precedence-level> <operator id>.
Right-associative: `infixr <precedence-level> <operator id>.

- If omitted, `<precedence-level> = 0 — the min. level.
- The highest precedence level is 9 in our SML.

<table>
<thead>
<tr>
<th>PRECEDENCE</th>
<th>OPERATORS</th>
<th>ASSOCIATIVITY</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>o</td>
<td>—</td>
<td>function composition</td>
</tr>
<tr>
<td></td>
<td>:=</td>
<td>—</td>
<td>assignment</td>
</tr>
<tr>
<td>4</td>
<td>=, &lt;&gt;, &lt;, &gt;, ≤, ≥</td>
<td>left</td>
<td>relational operators</td>
</tr>
<tr>
<td>5</td>
<td>::</td>
<td>right</td>
<td>list constructor</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>right</td>
<td>list concatenation</td>
</tr>
<tr>
<td>6</td>
<td>+, -</td>
<td>left</td>
<td>add/subtract</td>
</tr>
<tr>
<td></td>
<td>∧</td>
<td>left</td>
<td>string concatenation</td>
</tr>
<tr>
<td>7</td>
<td>*, /, div, mod</td>
<td>left</td>
<td>multiply/divide</td>
</tr>
</tbody>
</table>
New Operator..

(* First create the function *)
- fun **(a,0) = 1 | **(a,b) = a * **(a,b-1);
val ** = fn : int * int -> int

- **(2,5);
val it = 32 : int

- infix 7 **;  (* Make ** left-associative *)
infix 7 **

- 4 + 2**5 - 6;    2**3**2;
val it = 30 : int
val it = 64 : int

- infixr 7 **;  (* Make ** right-associative *)
infixr 7 **

- 2**3**2;
val it = 512 : int
Functions are called as **prefix** operators.

Built-in operators like `+`, `−`, `∗`, `/` are called as **infix** operators.

Internally, infix operators are actually functions. To use them as functions: `op` operator-symbol.

- `op+(2,3); op*(4,5);`
  - `val it = 5 : int`
  - `val it = 20 : int`

- `reduce op+ [2,3,4] 0; reduce op* [2,3,4] 1;`
  - `val it = 9 : int`
  - `val it = 24 : int`

- `op+;`
  - `val it = fn : int * int -> int`
Part IV

Static Scope: let Expression
let Expression

let

\[\text{val } \langle 1\text{st-identifier} \rangle = \langle E_1 \rangle;\]
\[\text{val } \langle 2\text{nd-identifier} \rangle = \langle E_2 \rangle;\]
\[
\ldots
\]

in

\[\langle \text{expression} \rangle\]

end

- The semicolons at the end of each \texttt{val} statements is \textit{optional}.
- c.f. Declaration of local variables in C++
- val z = 
  let
    val x = 3;
    val y = 5;
  in
    x*x + 3*y
  end;
val z = 24 : int

• As spaces are immaterial, the statement may as well be written all in one single line as follows:

  val z = let val x = 3 val y = 5 in x*x + 3*y end;

• To avoid too many val statements in the let-part, one may use tuples to group all identifiers as follows:

  val z = let val (x, y) = (3, 5) in x*x + 3*y end;
Nested let Example

- let val x = 3.0 val y = 5.0 in
  let val a = x+y val b = x-y in
    let val f = a*b*x val g = a/b/y in f/g end
  end
end

Quiz: What is the output?
Let’s rewrite the function reverse() with a locally defined function, rev_().

```ml
fun reverse L = let fun rev_([], L2) = L2 |
               rev_(x::tail, L2) = rev_(tail, x::L2)
in  rev_(L, []) end;
```

Identifiers with the same names are resolved using the static lexical scope rule.

```ml
fun weird(x: real) = let val x = x*x
                        val x = x*x
                        in  x*x*x end;
- weird 2.0;    (* What is the result? *)
```
Part V

New Datatypes
Syntax: `datatype <type-name>`

= `<1st-constructor>` | `<2nd-constructor>` | ... 

- A simple example:

```
datatype Primary_Lights = red | green | blue;
- red;
val it = red : Primary_Lights
```

- c.f. `enumeration` in C++

```
enum Primary_Lights = { red, green, blue };
```
More complex objects can be constructed too. e.g.

```
datatype Money = nomoney fun amount nomoney = 0
  | coin of int          | amount(coin(x)) = x
  | note10 of int        | amount(note10(x)) = 10*x
  | note100 of int       | amount(note100(x)) = 100*x
  | check of string*int; | amount(check(bank,x)) = x;
- amount (note100(2));
val it = 200 : int
```

Money has 5 constructors: nomoney as a constant constructor, coin(int), note10(int), note100(int), and check(string, int).

Any function on Money should deal with have 5 cases, one for each constructor.
Recursive Datatype: Differentiation Example

- datatype expr = constant of int
  | variable of string
  | sum of expr * expr
  | product of expr * expr;
- val zero = constant 0; val one = constant 1;

- fun D x (constant _) = zero
  | D x (variable z) = if x = z then one else zero
  | D x (sum(e1, e2)) = sum(D x e1, D x e2)
  | D x (product(e1, e2)) =
    let val term1 = product(D x e1, e2)
        val term2 = product(e1, D x e2)
    in sum(term1, term2) end;
val D = fn : string -> expr -> expr
• \texttt{expr} has 4 constructors: \texttt{constant(int)}, \texttt{variable(string)}, \texttt{sum(expr, expr)}, \texttt{product(expr, expr)}.

• Declarations of “zero” and “one” is necessary in order to have an output type of \texttt{expr}; you can’t use integers 0 and 1.

In order to use the new datatype \texttt{expr} and the differentiation function \texttt{D}, one has to convert a mathematical expression to \texttt{expr}. For example, to differentiate “\(x^2 + 5x\)”: 
- Compiler.Control.Print.printDepth := 10; 
val it = () : unit

- val term = sum(product(variable "x", variable "x"),
  product(constant 5, variable "x"));
val it =
  sum (product (variable "x",variable "x"),
    product (constant 5,variable "x")) : expr

- D "x" term;
val it = sum (sum (product (constant 1,variable "x"),
  product (variable "x",constant 1)),
  sum (product (constant 0,variable "x"),
    product (constant 5,constant 1))) : expr
Polymorphic Datatype: Binary Tree Example

datatype 'a tree =
    empty_tree | leaf of 'a | node of 'a tree*'a tree;

The 'a tree has 3 constructors: empty_tree (constant constructor), leaf('a tree), and node('a tree, 'a tree).

- fun leafcount(empty_tree) = 0
  | leafcount(leaf(x)) = 1
  | leafcount(node(L,R)) = leafcount(L) + leafcount(R);
val leafcount = fn : 'a tree -> int

- val x = node(node(leaf(1), leaf(2)), leaf(3));
val x = node (node (leaf #,leaf #),leaf 3) : int tree

- leafcount x;
val it = 3 : int


```ml
abstype 'a stack = stack of 'a list

with

    val emptystack = stack [];  
    fun SK_empty(stack y) = y = nil;  
    fun SK_push(x, stack y) = stack (x::y);  
    fun SK_pop(stack y) = (hd(y), stack(tl(y))));  
    fun SK_list(stack y) = y;  

end;

val x = emptystack;  
val y = SK_push(3, SK_push(4,x));  
val z = SK_pop y;  
SK_list x; SK_list y; SK_list (#2(z));  
SK_pop(#2(SK_pop(#2(SK_pop y))));
```
Part VI

Misc: Value Binding, Exception
Reference variable points to a value (c.f. indirect addressing):
\[
\text{val} \ <\text{identifier}\> = \text{ref} \ <\text{expression}\>.
\]
Assignment: \(<\text{identifier}\> := <\text{expression}>\)

Dereference: \(!<\text{identifier}>\)

- val x = ref(2+3);
  val x = ref 5 : int ref
- x := 9;
  val it = () : unit
- x;
  val it = ref 9 : int ref
- !x;
  val it = ref 9 : int ref

- val y = ref 9;
  val y = ref 9 : int ref
- !x = !y;
  val it = true : bool
- x = y;
  val it = false : bool

- val it = 9 : int
The phrase: “\( \text{val } x = 17 \)” is called a value binding; the variable \( x \) is bound to the value 17.

When an identifier is declared by a value binding, a new identifier is “created” — it has nothing whatever to do with any previously declared identifier of the same name.

Once an identifier is bound to a value, there is no way to change that value.

Environment: the current set of ordered pairs (identifier, value) that are visible.
Environment: Example

- val x = 17;
  val x = 17 : int

- val y = x;
  val y = 17 : int
  y = 17
  x = 17

- val x = true;
  val x = true : bool
   x = true
   y = 17

- val z = x;
  val z = true : bool
   z = true
   x = true
   y = 17
   x = 17
The assignment \( x := 9 \) produces the side-effect such that not only \( x \)'s dereferenced value is changed, but also \( y \)'s.
Alias: When a data object is visible through more than one name in a single referencing environment, each name is termed an alias.

Examples: passed parameters by reference in a function, several pointers to the same object.

Pitfall: programs are harder to understand.

Side Effects: An operation has side effects if it makes changes which persist after it returns.

Examples: A function changes its parameters or modifies global variables (through assignments); printouts.

Pitfall: programs are harder to understand, evaluation order of expressions becomes important.
```c
int x = 2, y = 5;
int Bad(int m) { return x+=m; }

void Swap(int* a, int* b)
{
    int temp = *a; *a = *b; *b = temp;
    x = 4;
}

int main()
{
    int* z = &x;
    int k = x * Bad(7) + x;

    printf("k = %d\n", k);
    Swap(&x, &y);
    printf("(x, y) = (%d,%d)\n", x, y);
}
```
Assignment and Value Binding: Example

\begin{verbatim}
val x = ref 0;

fun F y =
    let
        val w = 5;
    in
        y + 3*w + !x
    end;

F 1;
x := 10;
F 1;
val x = 999;
F 1;
val x = ref 999;
F 1;
\end{verbatim}

• What are the values after each "F 1;" expressions?
Keywords: *exception*, *raise*, *handle* =>

- $8 + 9 \text{ div } 0$;
  uncaught exception divide by zero

- exception DBZ;
 exception DBZ

- fun //\((a,b) = \text{ if } b = 0 \text{ then raise DBZ else } a \text{ div } b;\)
  val // = fn : int * int \to int

- infix 7 //;
  infix 7 //
- fun g (x,y,z) = x + y // z;
val g = fn : int * int * int -> int
- g(8,9,3);
val it = 11 : int

- g(8,9,0);
uncaught exception DBZ
  raised at: stdin:30.3-30.6

- fun f(x,y,z) = x + y // z handle DBZ => ~999;
val f = fn : int * int * int -> int

- f(8,9,0);
val it = ~999 : int
To load an SML program file which may contain value and function definitions, use “use”

- use "full-filename";

To load lib, use “open”

- open Int;
- open Real;
- open Math;
- open String;
- open List;
- open IO;
- open TextIO;
Part VII

Summary
✓ A task is achieved through applications of functions.
✓ No pointers!
✓ No coercion!
✓ No side-effects!
✓ Assignment is replaced by value binding.
✓ Implicit type inference.
✓ Implicit memory management: Objects are allocated as needed, and deallocated when they become inaccessible.
✓ Pattern matching ⇒ program by examples.
✓ Allow recursive definition of polymorphic datatypes.
✓ Simple exception handling.
### Summary: FP vs. IP

<table>
<thead>
<tr>
<th>IP:</th>
<th>Since IP languages are based on the von Neumann architecture, programmers must deal with the management of variables, assignment of values to them, memory locations, and sometimes even memory allocations.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Adv:</strong> efficient computation</td>
</tr>
<tr>
<td></td>
<td><strong>Disadv:</strong> laborious construction of programs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FP:</th>
<th>Do not manipulate memory directly; no variables, no assignments. Instead they work on values that are independent of an underlying machine.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Adv:</strong> compact language, simple syntax, higher level of programming</td>
</tr>
<tr>
<td></td>
<td><strong>Disadv:</strong> efficiency is sacrificed</td>
</tr>
<tr>
<td><strong>IP:</strong></td>
<td>Due to aliases and side effects, the effects of a subprogram or a block cannot be determined in isolation from the entire program.</td>
</tr>
<tr>
<td><strong>FP:</strong></td>
<td>Since they only manipulate values, there are no aliases nor side effects.</td>
</tr>
</tbody>
</table>

| **IP:** | Explicit memory management. |
| **FP:** | Storage is allocated as necessary; and storage that becomes inaccessible is automatically deallocated and reclaimed during garbage collection. |

| **IP:** | The power comes from mimicking operations on the underlying computer architecture with assignments, loops, and jumps. |
| **FP:** | The power comes from recursion and treating functions as “first-class” values. |