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** — often, the declaration of the type of a variable can be omitted, because ML can automatically infer what the type should be from the context of the expression the variable occurs in. Standard C++ still doesn’t support this. This brings the convenience and ease-of-use of ML a bit closer to Scheme/LISP where you never have to declare types of variables.

- **Patterns** — built-in pattern matching allows a declarative
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) 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
### 5 Basic Types, 3 Composite Types

<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>
Basic Types

- **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 ∼ 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

- 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

- real(5)/real(6);
  val it = 0.833333333333 : real
- 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.
Composite Type: Tuple

- (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.
Empty list: **nil** or [ ];

**nil** : ‘a list ⇒ a **polymorphic** object.

\([e_1, e_2, \ldots, e_n] \) is an abbreviation for \(e_1::e_2::\ldots::e_n::\text{nil}.\)

:: is the list **constructor** pronounced as “cons”.

:: is an infix operator which is **right associative**.

\(<\text{new-list}> = <\text{item}>::<\text{list}>.\)

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

Equality on 2 lists is item-by-item.
• **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 **alphanumeric identifiers**:

\[
<Id> ::= <First_Char><Other_Char><Other_Chars>
<First_Char> ::= [A-Z]|[a-z]|'
<Other_Char> ::= [A-Z]| [a-z]| [0-9]| [']
\]

BNF for **symbolic identifiers**:

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

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

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

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

- \( \texttt{val a'\_a} = "Albert"\^" "\^"Einstein"; \)
  \( \texttt{val a'\_a} = "Albert Einstein" : \texttt{string} \)

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

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

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

- 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, "man");
  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;

- Defined by enumerating **ALL** cases with pattern matching (⇒ more readable).
  - fun factorial 0 = 1
    | factorial x = x * factorial(x-1);
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 & Inference**

- **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 \rightarrow 'a) \rightarrow 'a
  val it = 25 : int

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

- fun first x y = x; first 2 "mak";
  val first = fn : 'a \rightarrow 'b \rightarrow '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. \text{first}() has domain type = \('a\), range = \('b \rightarrow 'a\).

- i.e. \text{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 -> 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

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

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

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 → 'b.

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

\[
\begin{align*}
type(H) &= type(f) → type(g) \\
type(g) &= type(x) → type(y) \\
type(f) &= type(x) → type(z) \\
type(G) &= type(z) → type(y) \equiv 'a → 'b
\end{align*}
\]

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

\[
\begin{align*}
type(f) &= 'c → 'a \\
type(g) &= 'c → 'b \\
type(H) &= ('c → 'a) → ('c → 'b)
\end{align*}
\]
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 \texttt{map()} has 2 arguments: a function \( f() \) and a list. It applies function \( f() \) to each element of the list.

\begin{verbatim}
fun map f [] = []
| map f (head::tail) = (f head)::(map f tail);
\end{verbatim}

- Type of list: \( 'a \) list
- Type of \( f \): \( 'a \rightarrow 'b \)
- Type of \texttt{map}: \( ('a \rightarrow 'b) \rightarrow 'a \) list \rightarrow 'b \) list
map: Examples

- 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.

```ocaml
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: \texttt{reduce}

- \texttt{reduce} accumulates a result from a list.

\begin{verbatim}
fun reduce f [ ] v = v
| reduce f (head::tail) v = f (head, reduce f tail v);
\end{verbatim}

\begin{itemize}
  \item reduce add [1,2,3,4,5] 0;
  \end{itemize}
  \begin{verbatim}
  val it = 15 : int
  \end{verbatim}

\begin{itemize}
  \item reduce;
  \end{itemize}
  \begin{verbatim}
  val it = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b
  \end{verbatim}

\begin{itemize}
  \item reduce add;
  \end{itemize}
  \begin{verbatim}
  val it = fn : int list -> int -> int
  \end{verbatim}

\begin{itemize}
  \item reduce add [1,2,3,4,5];
  \end{itemize}
  \begin{verbatim}
  val it = fn : int -> int
  \end{verbatim}
List Function: Example

- 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:** `fn <formal parameter> ⇒ <body>`

- An *anonymous function* is a function without a name.
- Used when only a locally defined function is needed.

- `map (fn x => x*x) [2,3,4];`
  `val it = [4,9,16] : int list`

- `map (fn (x,_) => x) [(1,2), (3,4), (5,6)];`
  `val it = [1,3,5] : int list`
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.

- val square = fn x => x*x; square 4;
  val square = fn : int -> int
  val it = 16 : int

- val f = square; f 4;
  val f = fn : int -> int
  val it = 16 : int

- val g = map square;
  val g = fn : int list -> int list

- g [1,2,3,4];
  val it = [1,4,9,16] : int list
Given: \( f: 'b \to 'c \) and \( g: 'a \to 'b \).

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

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.
Lazy (Outermost) Evaluation

\[ f(1+1) = \text{if } (1+1) = 1 \text{ then } 1 \text{ else } (1+1) * f((1+1)-1) \]
\[ = \text{if } 2 = 1 \text{ then } 1 \text{ else } (1+1) * f((1+1)-1) \]
\[ = (1+1) * f((1+1)-1) \]
\[ = 2 * f((1+1)-1) \]
\[ = 2 * \{ \text{if } ((1+1)-1) = 1 \text{ then } 1 \text{ else } \]
\[ \quad ((1+1)-1) * f(((1+1)-1) - 1) \} \]
\[ = 2 * 1 \]
\[ = 2 \]

Actual parameters are evaluated **only** when they are needed.
Give *same* result if the execution *terminates*.

But consider the following 2 examples:

```
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**)
Creating New Infix Operators

Left-associative: \texttt{infix} \texttt{<precedence-level>} \texttt{<operator id>}.  

Right-associative: \texttt{infixr} \texttt{<precedence-level>} \texttt{<operator id>}.  

- If omitted, \texttt{<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;, \leq, \geq</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>\land</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
Operators as Functions

- 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

val <1st-identifier> = < E_1 >;
val <2nd-identifier> = < E_2 >;
...

in

<expression>

end

- The semicolons at the end of each `val` statements is optional.
- c.f. Declaration of local variables in C++
let: val Example

- 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;
- 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 };
```
Constructors of Datatype

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
Recursive Datatype: Differentiation Example

- `expr` has 4 constructors: `constant(int)`, `variable(string)`, `sum(expr, expr)`, `product(expr, expr)`.

- Declarations of “zero” and “one” is necessary in order to have an output type of `expr`; you can’t use integers 0 and 1.

In order to use the new datatype `expr` and the differentiation function `D`, one has to convert a mathematical expression to `expr`. For example, to differentiate “`x*x + 5*x`”: 
Recursive Datatype: Differentiation Example ...

- 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
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
Abstract Data Types

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
Impure FP: Ref-Variables, Assignments

- **Reference variable** points to a value (c.f. indirect addressing):
  \[
  \text{val } \langle\text{identifier}\rangle = \text{ref } \langle\text{expression}\rangle.
  \]
- **Assignment**: \langle\text{identifier}\rangle := \langle\text{expression}\rangle
- **Dereference**: \!\langle\text{identifier}\rangle

\[
- \text{val } x = \text{ref}(2+3);
\]
\[
\text{val } x = \text{ref } 5 : \text{int ref}
\]
\[
- x := 9;
\]
\[
\text{val } \text{it} = () : \text{unit}
\]
\[
- x;
\]
\[
\text{val } \text{it} = \text{ref } 9 : \text{int ref}
\]
\[
- \!x;
\]
\[
\text{val } \text{it} = 9 : \text{int}
\]

- \text{val } y = \text{ref } 9;
- \text{val } y = \text{ref } 9 : \text{int ref}
- \!x = \!y;
- \text{val } \text{it} = \text{true} : \text{bool}
- \text{val } \text{it} = \text{false} : \text{bool}
- \text{val } \text{it} = \text{true} : \text{bool}
- \text{val } \text{it} = \text{false} : \text{bool}
The phrase: “\texttt{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.

\textbf{Environment}: the current set of ordered pairs (identifier, value) that are visible.
- val x = 17;
val x = 17 : int

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

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

- val z = x;
val z = true : bool

env:

x = 17
y = 17
x = true
z = true

Prof. Dekai Wu, HKUST (dekai@cs.ust.hk)
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);
}
```
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;

• What are the values after each “F 1;” expressions?
Keywords: exception, raise, handle

- \( 8 + 9 \div 0; \)
  uncaught exception divide by zero

- exception DBZ;
exception DBZ

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

- 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 $\Rightarrow$ program by examples.
- Allow recursive definition of polymorphic datatypes.
- Simple exception handling.
## Summary: FP vs. IP

| **IP:** | 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.  
| **Adv:** efficient computation  
| **Disadv:** laborious construction of programs |  
| **FP:** | Do not manipulate memory directly; no variables, no assignments. Instead they work on values that are independent of an underlying machine.  
| **Adv:** compact language, simple syntax, higher level of programming  
<p>| <strong>Disadv:</strong> efficiency is sacrificed |</p>
<table>
<thead>
<tr>
<th><strong>IP</strong></th>
<th>Due to aliases and side effects, the effects of a subprogram or a block cannot be determined in isolation from the entire program.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FP</strong></td>
<td>Since they only manipulate values, there are no aliases nor side effects.</td>
</tr>
<tr>
<td><strong>IP</strong></td>
<td>Explicit memory management.</td>
</tr>
<tr>
<td><strong>FP</strong></td>
<td>Storage is allocated as necessary; and storage that becomes inaccessible is automatically deallocated and reclaimed during garbage collection.</td>
</tr>
<tr>
<td><strong>IP</strong></td>
<td>The power comes from mimicking operations on the underlying computer architecture with assignments, loops, and jumps.</td>
</tr>
<tr>
<td><strong>FP</strong></td>
<td>The power comes from recursion and treating functions as “first-class” values.</td>
</tr>
</tbody>
</table>