Functional Paradigm

Pratt A.7
Robert Harper’s SML tutorial (Sec II)

Rafael Ramirez
Dep Tecnologia
Universitat Pompeu Fabra
Paradigm Shift

**Imperative Paradigm**
- State Machine (von Neumann architecture)
  - Series of statements to change state of memory
  - Multiple Assignments
  - Functions produce Side-Effects

**Functional Paradigm**
- Expression Rewrite Machine
  - One expression to evaluate
  - Single Assignment (For pure FP)
  - Functions do not produce Side-Effects (For pure FP)
Paradigm Shift - Machine

- Computational Model is **not** a Von Neumann state machine.
- It is an Expression Rewrite Machine.

Rules/Definitions

```plaintext
fun fac x = ...
fun try x y = ...
...
```

Repeat

1. Match Expression with Rules to Select correct rule.
2. **Rewrite the Expression**: Perform Textual Substitution with renaming of variables

Until Expression cannot be rewritten anymore

Starting Expression

```
fac 3
```

Look carefully, there’s no concept of memory state!!!
Another Characteristic of pure functional paradigm is the single ‘assignment’ nature. The motivation lies in math proofs.

Let’s say that in the process of proving, we say...

\[ A = \{ 2x \mid x \in \mathbb{Z} \} \]

Now, what if we want the set of odd integers? Do we ‘assign’ it back to A? No! If we want the set of odd integers, we ‘declare a new variable’

\[ B = \{ 2x+1 \mid x \in \mathbb{Z} \} \]

Note that the ‘=’ symbol is not assignment. It is more accurately interpreted as definition.

We do not read that A is assigned to the set of even integers, but rather A is defined to be the set of even integers.

We do not change definitions. If you want the odd integer set, then you define another set.

‘=’ is more accurately read as a DEFINITION, and DEFINITIONS DO NOT CHANGE.
Paradigm Shift – single ‘assignment’

- A consequence of single ‘assignment’ is a property called **referential transparency**.
- Also known as the ‘**equals can be substituted for equals**’ property.
- Since a variable is defined once, it can never be **changed**. Hence you can substitute it any where in the program. There is **no concept of time where states change**.
- This is just like in math: once A is a set of even integers, you can substitute A anywhere in your calculations: since the set once defined, will never change.
Paradigm Shift – Side-Effect Free

- Functions are treated as mathematical functions and mathematical functions do not have side-effects.
  - Output is solely dependent of input.

\[ \text{Pure function} \rightarrow \text{States} \rightarrow \text{Impure function} \]

\[ \text{without side-effects} \rightarrow \text{with side-effects} \]
Boolean Expressions

Scalars – Boolean Types
- Represented by true and false.

```
- true;
val it = true : bool
- false;
val it = false : bool
```

Operations on Booleans.
- Logical And:  andalso
- Logical Or:   orelse
- Logical Not:  not

- Evaluation, from left to right.

```
- false andalso true orelse not true;
val it = false : bool
```
Numerical Expressions

 Scalars – Numerical Types

 Integers

 Reals

 Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM&CMB]

 \[-2;\]
 val it = 2 : int

 \[-3.14;\]
 val it = 3.14 : real

 Negative Numbers use ‘~’ and not ‘–’

 \[-1;\]
 stdIn:1.1 Error: expression or pattern begins with infix identifier "-"
 stdIn:1.1-24.2 Error: operator and operand don’t agree [literal]
 \hspace{1em} operator domain: 'Z * 'Z
 \hspace{1em} operand: int
 \hspace{1em} in expression:
 \hspace{2.5em} - 1

 \[-~1;\]
 val it = ~1 : int
Numerical Expressions

Operations on Numerical Types

- Standard arithmetic: +,-,*, div (integer division), / (real division)

  - 2 + 3;
  val it = 5 : int
  - 2.5 * 7.4;
  val it = 18.5 : real

- Boolean operations:
  - Equality:  =
  - Inequality:  <>
  - Comparison:  <,>,<=,>=

  - 3 = 5;
    val it = false : bool
  - 3 <> 5;
    val it = true : bool
  - 3 <= 5 andalso 5 <> 6;
    val it = true : bool
Numerical Expressions

Operations on Numerical Types

- Overloading supported.
  
  - 5 * 3;
  
  val it = 15 : int
  
  - 5.0 * 3.0;
  
  val it = 15.0 : real

- Coercion **not** supported.
  
  - 2 + 3.4;
  
  stdin:27.1-27.8 Error: operator and operand don't agree

[literal]

  operator domain: int * int
  
  operand: int * real
  
  in expression:
  
  2 + 3.4
Summary

\[ \begin{align*}
\langle \text{Exp} \rangle & ::= \langle \text{Constants} \rangle; \\
& \quad \mid \langle \text{Exp}_1 \rangle \ \langle \text{op} \rangle \ \langle \text{Exp}_2 \rangle;
\end{align*} \]

\[ \begin{align*}
\langle \text{Constants} \rangle & ::= \text{all constants} \\
\langle \text{op} \rangle & ::= \text{all binary operators}
\end{align*} \]
If-Expression

\[ \text{<Exp>} ::= \text{<Constants>}; \]
\[ \quad | \text{<Exp}_1> \quad \text{<op>} \quad \text{<Exp}_2>; \]
\[ \quad | \text{if} \quad \text{<Exp}_0> \quad \text{then} \quad \text{<Exp}_1> \quad \text{else} \quad \text{<Exp}_2>; \]

\[ \text{<Constants>} ::= \text{all constants} \]
\[ \text{<op>} ::= \text{all binary operators} \]
If-Expression

- If-expression (and not if-statement).

\[ \text{if } <e_0> \ \text{then } <e_1> \ \text{else } <e_2> \]

- Example:

  ```
  - if (2 > 3 andalso 3.5 < 7.4) then 4 + 5 else 2 - 3;
  val it = ~1 : int
  ```

- Since all expressions evaluate to a value, the If-expression also evaluate to a value.

- Why can’t we leave out the ‘else’ part (just a if-then syntax)?
- See the difference between an if-expression and an if-statement?
Declarations - \texttt{val}

\[
\begin{align*}
<\text{Decl}> & ::= <\text{ValueDecl}> \mid \ldots \\
<\text{ValueDecl}> & ::= \texttt{val} <\text{pattern}> = <\text{Exp}> \\
<\text{pattern}> & ::= <\text{Identifier}> \mid \ldots \\
<\text{Exp}> & ::= <\text{Constant}>; \\
& \quad \mid <\text{Exp}_1> \; <\text{op}> \; <\text{Exp}_2>; \\
& \quad \mid \text{if} <\text{Exp}_0> \; \text{then} \; <\text{Exp}_1> \; \text{else} \; <\text{Exp}_2>; \\
<\text{Identifier}> & ::= \text{all Identifiers} \\
<\text{Constant}> & ::= \text{all constants} \\
<\text{op}> & ::= \text{all binary operators}
\end{align*}
\]
Declarations - `val`

Declaring an expression with a name enables easy reference.

- `val x = 3;`
- `val y = 4 + x;`
- `val z = 20 - y;`
- `x + y + z;`
- `val it = 23 : int`
Declarations - fun

<Decl> ::= <ValueDecl> | <FunctionDecl>
<ValueDecl> ::= val <pattern> = <Exp>
<FunctionDecl> ::= fun <var> <param> = <exp>
<param> ::= <pattern> { <pattern> }
<pattern> ::= <Identifier> | …

<Exp> ::= <Constant>;  
  | <Exp₁> <op> <Exp₂>;  
  | if <Exp₀> then <Exp₁> else <Exp₂>;  
  | <Exp₁> <Exp₂>;

<Identifier> ::= all Identifiers
<Constant> ::= all constants
<op> ::= all binary operators
Declarations - \textbf{fun}

\begin{align*}
\langle \text{Decl} \rangle & ::= \text{fun} \ \langle \text{var} \rangle \ \langle \text{param} \rangle = \langle \exp \rangle \\
\langle \text{param} \rangle & ::= \langle \text{pattern} \rangle \ \{ \langle \text{pattern} \rangle \} \\
\langle \text{pattern} \rangle & ::= \langle \text{identifier} \rangle \ | \ldots \\
\langle \exp \rangle & ::= \ldots \ | \ \langle \exp_1 \rangle \ \langle \exp_2 \rangle;
\end{align*}

\begin{align*}
\text{fun } \texttt{fac} \ x & = \text{if } (x = 0) \text{ then } 1 \\
& \quad \text{else } x \ * \ (\texttt{fac} \ (x-1));
\end{align*}

\begin{align*}
\text{val fac} & = \text{fn} : \text{int} \rightarrow \text{int} \\
\text{fac} \ 5; & \quad \text{val it} = 120 : \text{int}
\end{align*}

Iterations are accomplished via \textit{recursion}. 
Expression Evaluation

Reminder:
- Computation is via an Expression Rewrite Machine.

Rules/Definitions
- \texttt{fun fac x = ...}
- \texttt{fun try x y = ...}
- ...

Repeat
1. Match Expression with Rules to Select correct rule.
2. \textbf{Rewrite the Expression}:
   - Perform Textual Substitution with renaming of variables
   - Until Expression cannot be rewritten anymore

Look carefully, there’s no concept of memory state!!!
Expression Evaluation

**Evaluation of Expressions using an Expression Rewrite System (ERS).**

```plaintext
fac 3
→ if (3 = 0) then 1 else 3 * (fac (3 – 1))
→ 3 * (fac (3 – 1))
→ 3 * (fac 2)
→ 3 * (if (2 = 0) then 1 else 2 * (fac (2 – 1)))
→ 3 * (2 * (fac (2 – 1)))
→ 3 * (2 * (fac 1))
→ 3 * (2 * (if (1 = 0) then 1 else 1 * (fac (1 – 1))))
→ 3 * (2 * (1 * (fac (1 – 1))))
→ 3 * (2 * (1 * (fac 0)))
→ 3 * (2 * (1 * (if (0 = 0) then 1 else 0 * (fac (0 – 1)))))
→ 3 * (2 * (1 * 1))
→ 6
```
Equational Definitions

fun fac 0 = 1
| fac n = n * (fac (n-1));

Known as **equational definition**.

**Pattern-matching** proceeds in a *top-down, left-right* manner.
Equational Definitions

\[
\text{\textbf{Decl}} \quad ::= \quad \text{fun} \quad \langle \text{var} \rangle \quad \langle \text{param} \rangle \quad = \quad \langle \text{Exp} \rangle \\
\quad \quad \quad \{ \text{``|''} \quad \langle \text{var} \rangle \quad \langle \text{param} \rangle \quad = \quad \langle \text{Exp} \rangle \}
\]

\[
\text{\textbf{param}} \quad ::= \quad \langle \text{pattern} \rangle \quad \{ \langle \text{pattern} \rangle \}
\]

\[
\text{\textbf{pattern}} \quad ::= \quad \langle \text{identifier} \rangle \quad | \quad \langle \text{Constant} \rangle \quad | \quad \ldots
\]

\[
\text{\textbf{Exp}} \quad ::= \quad \ldots \quad | \quad \langle \text{Exp}_1 \rangle \quad \langle \text{Exp}_2 \rangle;
\]

\[
\text{fun} \quad \text{gcd} \quad 0 \quad n \quad = \quad n \\
| \quad \text{gcd} \quad n \quad 0 \quad = \quad n \\
| \quad \text{gcd} \quad m \quad n \quad = \quad \text{if} \quad (m \quad > \quad n) \quad \text{then} \quad \text{gcd} \quad (m-n) \quad n \\
\quad \quad \quad \text{else} \quad \text{gcd} \quad (n-m) \quad m
\]

Known as \textbf{equational definition}.

Pattern-matching proceeds in a \textbf{top-down}, \textbf{left-right} manner.
Mutual Recursion

Mutual Recursion can be specified as above

```
fun even(0)  = true  
| even(1)  = false 
| even(x)  = odd(x-1) 
and odd(0) = false 
| odd(1)  = true 
| odd(x)  = even(x-1);
```
Equational Definitions

\[
<\text{Decl}> ::= <\text{ValueDecl}> | <\text{FunctionDecl}>
\]
\[
<\text{ValueDecl}> ::= \text{val} <\text{pattern}> = <\text{Exp}>
\]
\[
<\text{FunctionDecl}> ::= \text{fun} <\text{var}> <\text{param}> = <\text{Exp}>
\]
\[
\{
\begin{array}{l}
\text{[‘|’ | and] } <\text{var}> <\text{param}> = <\text{Exp}>
\end{array}
\}
\]
\[
<\text{param}> ::= <\text{pattern}> \{ <\text{pattern}> \}
\]
\[
<\text{pattern}> ::= <\text{Identifier}> | <\text{Constant}> | ... 
\]
\[
<\text{Exp}> ::= <\text{Constant}>;
\]
\[
| <\text{Exp}_1> <\text{op}> <\text{Exp}_2>;
\]
\[
| \text{if } <\text{Exp}_0> \text{ then } <\text{Exp}_1> \text{ else } <\text{Exp}_2>;
\]
\[
| <\text{Exp}_1> <\text{Exp}_2>;
\]
\[
<\text{Identifier}> ::= \text{all Identifiers}
\]
\[
<\text{Constant}> ::= \text{all constants}
\]
\[
<\text{op}> ::= \text{all binary operators}
\]
Functions

What’s new about functions?

- Functions can be nameless (lambda abstraction).
- Functions can be passed in as arguments.
- Functions can be returned as results.
- Functions can be partially applied (related to the previous two).

The above attributes make the usage of functions **higher-order** in nature (in contrast to first-order).

People sometimes say that functions are ‘**first-class citizens**’ (i.e. no different from other datatypes – eg. integers can be passed in as arguments, returned as results, integer constants need not be given a name).
Lambda Abstraction

(1) Functions can be nameless (called a lambda abstraction)

Lambda abstraction: A nameless function

\[
\text{fn } \langle \text{param} \rangle \Rightarrow \langle \text{Exp} \rangle
\]

\[
(fn \ x \Rightarrow x + 2)
\]
val it = fn int -> int

\[
(fn \ x \Rightarrow x + 2) \ 3
\]
val it = 5 : int

Evaluation of a nameless function is again through call-by-name. Textual substitution of formals for actuals.
Lambda Abstraction

(1) Functions can be nameless (called a lambda abstraction)

Lambda abstraction: A nameless function

\[ \text{fn } <\text{param}> \Rightarrow <\text{Exp}> \]

\[
\text{<Exp>} ::= \text{<Constant>};
| \text{<Exp}_1 \ <\text{op}> \ <\text{Exp}_2>; \\
| \text{if} \ <\text{Exp}_0> \ \text{then} \ <\text{Exp}_1> \ \text{else} \ <\text{Exp}_2>; \\
| \ <\text{Exp}_1> \ <\text{Exp}_2>; \\
| \text{fn} \ <\text{param}> \Rightarrow <\text{Exp}> \quad \text{// lambda abstraction} \\
\]
Functions as arguments

(2) Functions can be passed in as arguments

```ml
fun apply f x = f x

apply fac 3
  ➔ fac 3
  ➔ ...
  ➔ 6

fun twice f x = apply f (apply f x)

twice fac 3
  ➔ ???
```
Functions as arguments

(2) Functions can be passed in as arguments

```ml
fun apply f x = f x

apply fac 3
  ➔ fac 3
  ➔ ...
  ➔ 6

fun twice f x = apply f (apply f x)

twice fac 3
  ➔ apply fac (apply fac 3)
  ➔ apply fac (fac 3)
  ➔ ...
  ➔ apply fac 6
  ➔ fac 6
  ➔ ...
  ➔ 720
```
(2) Functions can be passed in as arguments

\[(\text{fn } g \Rightarrow g \ 3) \ (\text{fn } x \Rightarrow x \ * \ 2)\]
(2) Functions can be passed in as arguments

\[(\text{fn } g \Rightarrow g \ 3) \ (\text{fn } x \Rightarrow x \ * \ 2)\]
Functions as results

(3) Functions can be returned as results

```martin
fun choose x f g = if x then f else g ;
val choose = fn : bool -> 'a -> 'a -> 'a
  choose (2=3) (fn x => x + 2) (fn x => x * 2)
  ⇒ if (2=3) then (fn x => x + 2)
      else (fn x => x * 2)
  ⇒ (fn x => x * 2)  // Function result

fun compose f g = (fn x => f (g x))
compose (fn x => x + 3) fac
  ⇒ (fn y => (fn x => x + 3) (fac y)) // function result
  // Note: rename variables to avoid confusion.
compose (fn x => x + 3) fac 4
  ⇒ (fn y => (fn x => x + 3) (fac y)) 4
  ⇒ (fn x => x + 3) (fac 4)
  ⇒ ... ⇒ 27
```
Functions as results

```sml
defun compose f g = (fn x => f (g x))
```

The above function composition is predefined in SML as an infix ‘o’ operator (same idea as in math)

And so

```sml
compose (fn x => x + 3) fac
```

can be re-written as:

```sml
(fn x => x + 3) o fac
```

And we can invoke it with an argument:

```sml
((fn x => x + 3) o fac) 4
```
Partially Applied Functions

(4) Functions can be partially applied

```ml
fun pappdemo x y z = x + 2*y + 3*z;
val it = fn int -> int -> int -> int

pappdemo 1 2 3;
val it = 14 : int

val newfun = pappdemo 1;  // partial application
val it = fn int -> int -> int

newfun 2 3;
val it = 14 : int

Note that Evaluation proceeds in the following manner:
newfun 2 3  ➔  pappdemo 1 2 3  ➔  ...  ➔  14
```
Let-Expressions (Local Declarations)

\[
<\text{Decl}> ::= <\text{ValueDecl}> \mid <\text{FunctionDecl}>
\]

\[
<\text{ValueDecl}> ::= \text{val} <\text{pattern}> = <\text{Exp}>
\]

\[
<\text{FunctionDecl}> ::= \text{fun} <\text{var}> <\text{param}> = <\text{Exp}>
\quad \{ '|' <\text{var}> <\text{param}> = <\text{Exp}> \}
\]

\[
<\text{param}> ::= <\text{pattern}> \{ <\text{pattern}> \}
\]

\[
<\text{pattern}> ::= <\text{Identifier}> \mid <\text{Constant}> \mid \ldots
\]

\[
<\text{Exp}> ::= <\text{Constant}>
\quad | <\text{Exp}_1> <\text{op}> <\text{Exp}_2>
\quad | \text{if} <\text{Exp}_0> \text{ then } <\text{Exp}_1> \text{ else } <\text{Exp}_2>
\quad | \text{let} \{ <\text{Decl}> \} \text{ in } <\text{Exp}> \text{ end}
\quad | <\text{Exp}_1> <\text{Exp}_2>
\quad | \text{fn} <\text{param}> = > <\text{Exp}>
\]

\[
<\text{Identifier}> ::= \text{all Identifiers}
\]

\[
<\text{Constant}> ::= \text{all constants}
\]

\[
<\text{op}> ::= \text{all binary operators}
\]
Let-Expressions (Local Declarations)

\[
\begin{align*}
<Exp> & ::= \ldots \\
& \mid \text{let } \{<Decl>\} \text{ in } <Exp> \text{ end;}
& \mid \ldots
\end{align*}
\]

let val x = 3
val y = 4
fun try c = c + x + y
in try 5
end;

val it = 12 : int
Let-Expressions (Local Declarations)

\(<\text{Exp}> ::= \ldots\)

\(| \ \text{let } \{\text{<Decl>}}\} \ \text{in} \ \text{<Exp>} \ \text{end};\)

\(| \ \ldots\)

\(<\text{Decl}>\)

\(<\text{Exp}>\)

\(<\text{Exp}>\)

let val y = let
  val a = 4
  val x = 5
in
  let
    val b = x + a
  in
    b + a
end
end;

val it = 13 : int

1. Nested Lets (BNF allows it)
2. Static Scoping Rules
The SML program

\[
\text{<SMLProgram>} ::= \{ \text{<Decl>} \} \text{<Exp>}
\]

Comment: Being an interpreted language, \{ <Decl> | <Exp> \} is allowed.

\[
\text{<Decl>} ::= \text{<ValueDecl>} | \text{<FunctionDecl>}
\]

\[
\text{<ValueDecl>} ::= \text{val} \text{<pattern>} = \text{<Exp>}
\]

\[
\text{<FunctionDecl>} ::= \text{fun} \text{<var> <param>} = \text{<Exp>}
\]

\[
\quad \{ [ \text{\textquotesingle} | \text{\textquotesingle} ] \text{<var> <param>} = \text{<Exp>} \}
\]

\[
\text{<param>} ::= \text{<pattern>} \{ \text{<pattern>} \}
\]

\[
\text{<pattern>} ::= \text{<Identifier>} | \text{<Constant>} | \ldots
\]

\[
\text{<Exp>} ::= \text{<Constant>}
\]

\[
\quad | \text{<Exp}_1 \text{<op> <Exp}_2>
\]

\[
\quad | \text{if} \text{<Exp}_0 \text{then} \text{<Exp}_1 \text{else} \text{<Exp}_2
\]

\[
\quad | \text{let} \{ \text{<Decl>} \} \text{in} \text{<Exp>} \text{end}
\]

\[
\quad | \text{<Exp}_1 \text{<Exp}_2
\]

\[
\quad | \text{fn} \text{<param>} \Rightarrow \text{<Exp>}
\]

\[
\text{<Identifier>} ::= \text{all Identifiers}
\]

\[
\text{<Constant>} ::= \text{all constants}
\]

\[
\text{<op>} ::= \text{all binary operators}
\]
Tuples and Lists

- Heterogeneous data stored using **TUPLES** (which are predefined).

- Homogeneous data stored using (singly-linked) **LISTS** (which are also predefined)
Tuples

- A tuple is a record without field names.
- Exactly the same as the mathematical understanding of n-tuples.

\[ <\text{Exp}> ::= \ldots \mid ( <\text{Exp}> , <\text{Exp}> \{ , <\text{Exp}>\} ) \]

- Note that n-tuples with \( n > 1 \) are supported. (pairs, triplets etc.)
- Tuple **Data Constructor**: ()
- Tuple **Type Constructor**: *
  - (Cartesian product of 2 sets)
- **Examples:**
  - \((1,2) : \text{int} \ast \text{int}\)
  - \((1,2.3,\text{True}) : \text{int} \ast \text{real} \ast \text{bool}\)
Tuples – Pattern Matching

```ocaml
let
    x = (2+3, 4-2, 10)
in
let
    (a, b, c) = x
in
    a + c
```

- Tuple **construction** is done via `( )`.
- Tuple **deconstruction** is done via pattern-matching.
Tuples – Pattern Matching

```
let
  x = (2+3, 4-2, 10)
in
  let
    (a, b) = x
    in
    a + c
```

- **Wrong deconstruction** (i.e. matched to wrong pattern) will cause run-time error
Tuples – Pattern Matching

```
let
  x = ((1,2),3,(4,5,6))
in
let
  (_,_,(_,y,_)) = x
in
  y
```

- **Patterns** can nest to arbitrary depth.
- Use `_` for anonymous (don’t care) patterns.
- Note that \((1,(2,3)) \neq ((1,2),3) \neq (1,2,3)\)
Tuples - BNF

\[ <\text{SMLProgram}> ::= \{ <\text{Decl}> \} <\text{Exp}> \]

\[ <\text{Decl}> ::= <\text{ValueDecl}> | <\text{FunctionDecl}> \]

\[ <\text{ValueDecl}> ::= \text{val} <\text{pattern}> = <\text{Exp}> \]

\[ <\text{FunctionDecl}> ::= \text{fun} <\text{var}> <\text{param}> = <\text{Exp}> \]

\[ \{ [' |' | \text{and}] <\text{var}> <\text{param}> = <\text{Exp}> \} \]

\[ <\text{param}> ::= <\text{pattern}> \{ <\text{pattern}> \} \]

\[ <\text{pattern}> ::= <\text{Identifier}> | <\text{Constant}> | <\text{Constructor}> \{<\text{pattern}>\} \]

\[ <\text{Exp}> ::= <\text{Constant}> \]

\[ | <\text{Exp}_1> \text{ op } <\text{Exp}_2> \]

\[ | \text{if} <\text{Exp}_0> \text{ then } <\text{Exp}_1> \text{ else } <\text{Exp}_2> \]

\[ | \text{let} \{ <\text{Decl}> \} \text{ in } <\text{Exp}> \text{ end} \]

\[ | <\text{Exp}_1> <\text{Exp}_2> \]

\[ | \text{fn} <\text{param}> => <\text{Exp}> \]

\[ | <\text{TupleExp}> \]

\[ <\text{TupleExp}> ::= ( <\text{Exp}> , <\text{Exp}> \{ , <\text{Exp}>\} ) \]

Each data constructor has its fixity modes. So adapt accordingly

Tuple Expressions
Lists

- Lists are predefined types in ML. List type-constructor: list
  - List of objects of type x denoted as: ‘x list

- Lists are constructed using data-constructors:
  - Cons: denoted by ::
  - Nil: denoted by [] or nil

...both constructors have a type.

The list constructor takes in an element of type ‘a and a list of elements of type ‘a and returns a list of elements of type ‘a

- op :: ;
- val it = fn : 'a * 'a list -> 'a list
- nil;
- val it = [] : 'a list
List Expressions

\[
<ListExp> ::= \text{nil} \mid <\text{Exp}> :: <\text{ListExp}>
\]

\[
:: \text{\textquoteleft} \left[ \langle \text{Exp} \rangle \{, <\text{Exp}> \} \right] \text{\textquoteright}
\]

- **nil**
- **1::nil**
- **1::(2::nil)**
- **1::(2::(3::nil))**

**Note**: A list must end with **nil**.

**Head** of a list     **Tail** of a list

**Terminals**

**EBNF Optional symbol**

**EBNF zero or more repetitions**
List Expressions

Example:
- Different types of lists
  1 :: (2 :: (3 :: nil)) : int list
  [3.14, 2.414] : real list
  [[1,2] , [3,4]] : int list list
List Expressions

Example:

Different ways of specifying the same list:

1 :: (2 :: (3 :: nil))

...is also the same as saying:

1 :: (2 :: (3 :: []))

...which is also the same as saying:

1 :: 2 :: 3 :: []

(since the :: is right associative.)

...which can be expressed as:

1 :: [2, 3]

...which can be expressed in a more readable way:

[1, 2, 3]
List Expressions

\[
\begin{align*}
\langle \text{SMLProgram} \rangle & ::= \{ \langle \text{Decl} \rangle \} \langle \text{Exp} \rangle \\
\langle \text{Decl} \rangle & ::= \langle \text{ValueDecl} \rangle \mid \langle \text{FunctionDecl} \rangle \\
\langle \text{ValueDecl} \rangle & ::= \text{val} \langle \text{pattern} \rangle = \langle \text{Exp} \rangle \\
\langle \text{FunctionDecl} \rangle & ::= \text{fun} \langle \text{var} \rangle \langle \text{param} \rangle = \langle \text{Exp} \rangle \\
& \quad \{[ \text{'} \mid \text{ and} ] \langle \text{var} \rangle \langle \text{param} \rangle = \langle \text{Exp} \rangle \} \\
\langle \text{param} \rangle & ::= \langle \text{pattern} \rangle \{ \langle \text{pattern} \rangle \} \\
\langle \text{pattern} \rangle & ::= \langle \text{Identifier} \rangle \mid \langle \text{Constant} \rangle \\
& \quad | \langle \text{Constructor} \rangle \{ \langle \text{pattern} \rangle \} \\
\langle \text{Exp} \rangle & ::= \langle \text{Constant} \rangle \\
& \quad | \langle \text{Exp}_1 \rangle \langle \text{op} \rangle \langle \text{Exp}_2 \rangle \\
& \quad | \text{if} \langle \text{Exp}_0 \rangle \text{ then } \langle \text{Exp}_1 \rangle \text{ else } \langle \text{Exp}_2 \rangle \\
& \quad | \text{let} \{ \langle \text{Decl} \rangle \} \text{ in } \langle \text{Exp} \rangle \text{ end} \\
& \quad | \langle \text{Exp}_1 \rangle \langle \text{Exp}_2 \rangle \\
& \quad | \text{fn} \langle \text{param} \rangle \Rightarrow \langle \text{Exp} \rangle \\
& \quad | \langle \text{TupleExp} \rangle \mid \langle \text{ListExp} \rangle \\
\end{align*}
\]

\[
\begin{align*}
\langle \text{ListExp} \rangle & ::= \text{nil} \mid \langle \text{Exp} \rangle :< \langle \text{ListExp} \rangle \\
& \quad \mid \text{'}[\text{' [<Exp> \{, <Exp}>]} \text{']}' \\
\langle \text{TupleExp} \rangle & ::= (\langle \text{Exp} \rangle , \langle \text{Exp} \rangle \{, \langle \text{Exp} \rangle \})
\end{align*}
\]
Pattern Matching with Lists

```
let
   val (x :: tail) = [1,2,3,4,5]
in
   tail
end;

val it = [2,3,4,5] : int list
```

- **List deconstruction** is done via **pattern-matching**
  - `<Exp> ::= ... let {<Decl>} in <Exp> end`
  - `<ValueDecl> ::= val <pattern> = <Exp>`
Pattern Matching with Lists

```
let
  val (x :: tail) = []
in
  tail
end;
```

stdIn:20.1-20.33 Warning: type vars not generalized because of value restriction are instantiated to dummy types (X1,X2,...)

- **Wrong deconstruction** (i.e. matched to wrong pattern) will cause run-time error.
Pattern Matching with Lists

```plaintext
let
  val ((x::xs) :: tail) = [[1,2,3],[4,5]]

  in
  xs
  end;

val it = ???
```

- Patterns can nest to arbitrary depth.
Pattern Matching with Lists

let
  val ((x::xs) :: tail) = [[1,2,3],[4,5]]
in
  xs
end;

val it = [2,3] : int list

Patterns can nest to arbitrary depth.
Pattern Matching with Lists

```ocaml
let
  val ((_::xs) :: _) = [[1,2,3],[4,5]]
in
  xs
end;

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

- You can specify a ‘nameless match’. 
Pattern Matching with Lists

fun length []      = 0
|   length (x::xs) = 1 + length(xs);

length([10,10,10])

1. Note how a list is being consumed from head to tail.
2. Length is a predefined function in SML.
Pattern Matching with Lists

fun length []      = 0 |
                length (x::xs) = 1 + length(xs) ;

length([10,10,10])
⇒ 1 + length([10,10])
⇒ 1 + 1 + length([10])
⇒ 1 + 1 + 1 + length([])
⇒ 1 + 1 + 1 + 0
⇒ 3

1. Note how a list is being consumed from head to tail.
2. Length is a predefined function in SML.
Pattern Matching with Lists

fun append []       ys = ys
| append (x::xs) ys = x :: (append xs ys);

append [1,2] [3,4]
⇒ ???

1. Note how a new list is being *produced* from head to tail.
2. Append is predefined in SML by an infix operator @. So the above code is the same as: [1,2] @ [3,4]
Pattern Matching with Lists

\[
\text{fun append \} \quad \text{ys} = \text{ys} \\
| \quad \text{append \} (\text{x::xs}) \quad \text{ys} = \text{x} :: (\text{append \} \text{xs} \quad \text{ys})
\]

append \{1,2\} \{3,4\}

\Rightarrow \text{append \} (1::2::nil) \{3,4\}
\Rightarrow 1::(\text{append \} (2::nil) \{3,4\})
\Rightarrow 1::(2::(\text{append \} [\] \{3,4\}))
\Rightarrow 1::(2::([3,4]))
\Rightarrow [1,2,3,4]

1. Note how a new list is being \textit{produced} from head to tail.
2. Append is predefined in SML by an infix operator \texttt{@}. So
the above code is the same as: \{1,2\} \texttt{@} \{3,4\}
fun rev(xs) = 
    let fun r [] ys = ys 
    | r (x::xs) ys = r xs (x::ys) 
    in r xs [] 
end

rev [1,2,3] 
⇒ ???
Pattern Matching with Lists

```ml
fun rev(xs) = 
  let fun r []      ys = ys 
        | r (x::xs) ys = r xs (x::ys) 
  in r xs [] 
end

rev [1,2,3] 
  ➔ r [1,2,3] [] 
  ➔ r [2,3]   (1::[]) 
  ➔ r [3]     (2::1::[]) 
  ➔ r []      (3::2::1::[]) 
  ➔ [3,2,1]
```

1. Note how lists are being *produced and consumed* concurrently.
2. rev is predefined in SML.
Common Operations on Lists

There are usually two common things you do on lists:

- Applying an operation on all the elements of the list => Mapping
- Collapsing the list => Folding

Both of these illustrate

- the use of higher-order functions
- the power and flexibility of functional language.
fun map f [] = []
| map f (x::xs) = (f x) :: (map f xs)

map is predefined in SML.

fun dble x = x * 2;

map dble [1,2,3]
⇒ ???
Mapping

\[
\text{fun} \quad \text{map} \ f \ [\] \quad = \ [\] \\
\| \quad \text{map} \ f \ (x::xs) \quad = \ (f \ x) :: (\text{map} \ f \ xs)
\]

\text{map is predefined in SML.}

\text{fun} \quad \text{dble} \ x \quad = \ x \times 2;

\text{map} \ \text{dble} \ [1,2,3]
\Rightarrow (\text{dble} \ 1)::(\text{map} \ \text{dble} \ [2,3])
\Rightarrow (\text{dble} \ 1)::(\text{dble} \ 2)::(\text{map} \ \text{dble} \ [3])
\Rightarrow (\text{dble} \ 1)::(\text{dble} \ 2)::(\text{dble} \ 3)::(\text{map} \ \text{dble} \ [\])
\Rightarrow (\text{dble} \ 1)::(\text{dble} \ 2)::(\text{dble} \ 3)::[\])
\Rightarrow ... \Rightarrow [2,4,6]
fun map f [] = []
| map f (x::xs) = (f x) :: (map f xs)

Law for Map:

\[ \text{map } f \ (\text{map } g \ \text{list}) = \text{map } (f \circ g) \ \text{list} \]

\[
\begin{align*}
\text{map } f \ (\text{map } g \ [a_1, \ldots, a_n]) & \\
& = \text{map } f \ [g \ a_1, \ldots, g \ a_n] \\
& = [f \ (g \ a_1), \ldots, f \ (g \ a_n)] \\
& = [(f \circ g) \ a_1, \ldots, (f \circ g) \ a_n] \\
& = \text{map } (f \circ g) \ [a_1, \ldots, a_n]
\end{align*}
\]
Folding

`fun foldr f z []     = z`
| `foldr f z (x::xs) = f (x, (foldr f z xs))`

`foldr` is predefined in SML.

\[
\text{foldr} \oplus z \ [a_1, \ldots, a_n] = \\
\quad a_1 \oplus (a_2 \oplus (\ldots \oplus (a_n \oplus z)))
\]
Folding – Example 1

```sml
fun foldr f z [] = z
| foldr f z (x::xs) = f (x, (foldr f z xs))
```

`foldr` is predefined in SML.

Example 1: Add up all the elements of the list.

The ‘standard’ way of doing it:

```sml
fun addup [] = 0
| addup (x:xs) = x + (addup xs);
```
Folding – Example 1

Foldr is predefined in SML.

Example 1: Add up all the elements of the list.

The ‘foldr’ way of doing it:

fun addup xs = foldr (op +) 0 xs

Note: ‘op’ tells the SML interpreter to use an operator in a prefix mode.
fun foldr f z []      = z
|  foldr f z (x::xs) = f (x,(foldr f z xs))

fun addup xs = foldr (op +) 0 xs

Example 1: Add up all the elements of the list.

Detailed Trace:
addup [1,2,3]
⇒ ???

Note: ‘op’ tells the SML interpreter to use an operator in a prefix mode.
Folding – Example 1

fun foldr f z [] = z
| foldr f z (x::xs) = f (x,(foldr f z xs))

fun addup xs = foldr (op +) 0 xs

Example 1: Add up all the elements of the list.

Detailed Trace:
addup [1,2,3]
→ foldr (op +) 0 [1,2,3]
→ (op +) 1 (foldr (op +) 0 [2,3])
→ (op +) 1 ((op +) 2 (foldr (op +) 0 [3]))
→ (op +) 1 ((op +) 2 ((op +) 3 (foldr (op +) 0 [])))
→ (op +) 1 ((op +) 2 ((op +) 3 0))
→ ... → 6
Example 2: Find the length of the list

The ‘standard’ way of doing it:

```haskell
fun len [] = 0
| len (x:xs) = 1 + (len xs);
```

```haskell
fun foldr f z [] = z
| foldr f z (x::xs) = f (x, (foldr f z xs))
```
Folding – Example 2

Example 2: Find the length of the list

Another way of doing it: Expressing `len` in terms of `foldr`

```
fun len xs = foldr (fn (x, y) => 1 + y) 0 xs
```

```
fun foldr f z [] = z
| foldr f z (x :: xs) = f (x, (foldr f z xs))
```
Folding – Example 2

Example 2: Find the length of the list

Another way of doing it: Expressing \texttt{len} in terms of \texttt{foldr}

\begin{verbatim}
fun foldr f z [] = z
  | foldr f z (x::xs) = f (x,(foldr f z xs))
\end{verbatim}

\begin{verbatim}
fun len xs = foldr (fn (x,y) => 1 + y) 0 xs
\end{verbatim}
fun foldr f z [] = z
| foldr f z (x::xs) = f (x, (foldr f z xs))

Example 2: Find the length of the list

Detailed Trace:

len [20,40,60]

⇒ foldr (fn (x,y)=>1+y) 0 [20,40,60]

⇒ (fn (x,y)=>1+y)) (20, (foldr (fn (x,y)=>1+y) 0 [40,60]))

⇒ 1 + (foldr (fn (x,y)=>1+y) 0 [40,60]))

    // Notice how the 20 is ignored and a count of 1 is used instead

⇒ 1 + ((fn (x,y)=>1+y) (40, (foldr (fn (x,y)=>1+y) 0 [60])))

⇒ 1 + (1 + (foldr (fn (x,y)=>1+y) 0 [60]))

⇒ ...


Folding – Example 3

Example 3: Reversing the list.

The ‘standard’ way of doing it:

\[
\begin{align*}
\text{fun } \text{foldr} \ f \ z \ [ ] & \quad = \ z \\
| \quad \text{foldr} \ f \ z \ (x::xs) & \quad = \ f \ (x,(\text{foldr} \ f \ z \ xs))
\end{align*}
\]

\[
\text{fun } \text{rev}(xs) = \quad \text{let fun } \ r \ [ ] \quad \text{ys} = \ ys \\
| \quad \quad (x::xs) \quad \text{ys} = \ r \ xs \ (x::ys) \quad \text{in} \\
| \quad \quad r \ xs \ [ ] \quad \text{end}
\]
Example 3: Reversing the list.

Another way of doing it: Expressing \( \text{rev} \) in terms of \( \text{foldr} \)

\[
\text{fun rev xs} = \text{foldr} \ (\text{fn} \ (x,y) \Rightarrow y@[x]) \ [] \ xs
\]
Example 3: Reversing the list.

The ‘cool’ way of doing it: Expressing \texttt{rev} in terms of \texttt{foldr}

\[
\text{fun } \texttt{rev } \texttt{xs} = \texttt{foldr } (\texttt{fn } (x,y) \Rightarrow y @ [x]) [ ] \texttt{xs}
\]

\[
\begin{array}{c}
20 \\
40 \\
60 [ ]
\end{array}
\] \quad \xrightarrow{\text{foldr}} \quad \begin{array}{c}
@ \\
[20]
\end{array}
\] \quad \xrightarrow{\text{foldr}} \quad \begin{array}{c}
@ \\
[40]
\end{array}
\] \quad \xrightarrow{\text{foldr}} \quad \begin{array}{c}
@ \\
[60]
\end{array}
\]
Folding – Example 3

```
fun foldr f z []      = z
| foldr f z (x::xs) = f (x, (foldr f z xs))
```

Example 3: Reversing the list.

Detailed Trace:
```
rev [20,40,60]
⇒ foldr (fn (x,y)=>y@[x]) [] [20,40,60]
⇒ (fn (x,y)=>y@[x])) (20, (foldr (fn (x,y)=>y@[x]) [] [40,60]))
⇒ (foldr (fn (x,y)=>y@[x]) [] [40,60]) @ [20]
      // Notice the switch
⇒ ...
```
Folding – Summary

\[
\text{fun } \text{foldr } f \ z \ [\ ] \ = \ z \\
| \ \text{foldr } f \ z \ (x::xs) \ = \ f \ (x,(\text{foldr } f \ z \ xs))
\]

Many things that you need to do to lists can be done using just \texttt{foldr} (one-line of code!!!).

Very important list construct. Must try to get used to it.
Functions as first-class citizens

- You can store functions in data structures (as much as you can store integers in data structures)

```ocaml
val flist = [ fac, fib, add3] ;

let val [f1,f2,f3] = flist
in (f1 2) + (f2 3) + (f3 4)
end ;
```

- You still must obey type rules. In the case of lists, all functions stored must be of the same type (since a list stores elements of the same type).
Summary

<Exp> ::= <Constant> \\
| <Exp₁> <op> <Exp₂> \\
| if <Exp₀> then <Exp₁> else <Exp₂> \\
| let {<Decl>} in <Exp> end \\
| <Exp₁> <Exp₂> \\
| fn <param> => <Exp> \\
| <TupleExp> | <ListExp>

<ListExp> ::= nil | <Exp>::<ListExp> \\
| '[' [Exp] , <Exp>]']'

<TupleExp> ::= ( <Exp> , <Exp> )
Conclusion

- A functional program consists of an expression, not a sequence of statements. The computation model is an expression rewrite system.
- Functions are first-class citizen in the language.
  - Functions can be nameless, passed in as arguments, returned as results, partially applied.
- List processing is convenient and expressive
  - Use of map, foldr with higher-order functions for descriptive power.
- Every expression must be well-typed.
  - Strongly Typed Language.