Functional Programming: Expressions

(Based on [Sethi 1996] and [Leroy *et al.* 2012; OCaml])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Functional Programming

- Characteristics of *pure* functional programming:
  - *Programming without assignments*.
    The value of an expression depends only on the values of its subexpressions, if any.
  - *Implicit storage management*.
    Storage is allocated as necessary by built-in operations on data. Storage that becomes inaccessible is automatically deallocated.
  - *Functions as first-class values*.
    Functions have the same status as any other values. A function can be the value of an expression, it can be passed as an argument, and it can be put in a data structure.
- Many functional languages also include imperative constructs, making them "impure."

Computing with Expressions

- Example expressions:

2	an integer constant
x	a variable (defined earlier)
$\log x$	function log applied to x
$2 + 3$	function $+$ applied to 2 and 3

- Expressions can also include conditionals and function definitions.

  - **if** $x \geq y$ **then** x **else** y
  - **let** $addone\ n = n + 1$ **in** $addone\ 3$

Quilts: Values and Operations

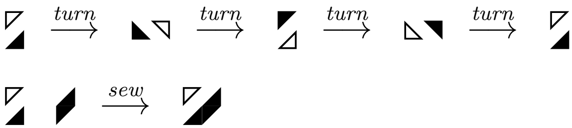
- We will consider, as a tiny functional language, *Little Quilt* for manipulating objects like the following:

- Below are the two primitive objects in *Little Quilt*:

They are actually *square* pieces whose lower left half is invisible.

Quilts: Values and Operations (cont.)

- Quilts can be *turned* and *sewed together*.
- Quilts and the operations on them are specified by the following rules:
  1. A quilt is one of the two primitive pieces, or
  2. it is formed by turning a quilt clockwise $90°$, or
  3. it is formed by sewing a quilt to the right of another quilt of equal height.
  4. Nothing else is a quilt.
- Examples:



Constants (in Little Quilt)

- Let the two primitive pieces be called a and b respectively.
  - So, we will be maniplating the quilts “symbolically.”
  - A layer of visualization will be added, when we implement Little Quilt in a real functional language.
- Let the two basic operations be called $turn$ and sew.

Expressions

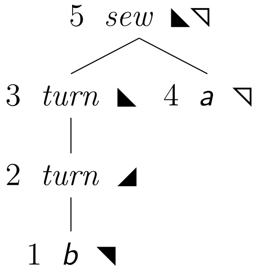
- The syntax of expressions in Little Quilt:

$$\begin{array}{rcl} \langle expr \rangle & ::= & a \\ & | & b \\ & | & (turn\ \langle expr \rangle) \\ & | & (sew\ \langle expr \rangle\ \langle expr \rangle) \end{array}$$

  The outermost pair of parentheses in an expression may be discarded.
- The *semantics* of expressions specifies the quilt denoted by an expression.
- Expressions will be extended by allowing functions from quilts to quilts and by allowing names for quilts.

Expressions (cont.)

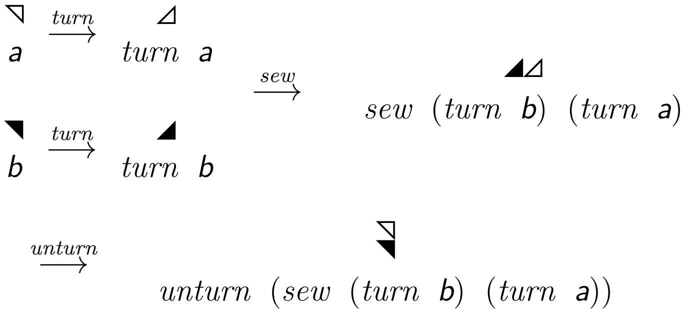
no.	operation	quilt
1	b	◥
2	$turn\ b$	◢
3	$turn\ (turn\ b)$	◣
4	a	◹
5	$sew\ (turn\ (turn\ b))\ a$	◣◹



User-Defined Functions

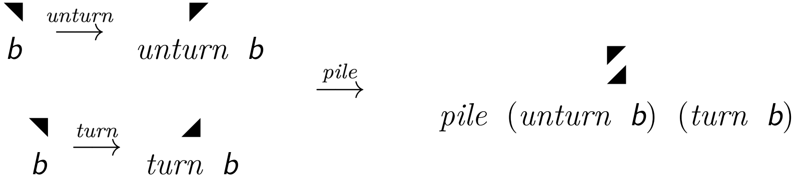
- Frequent operations, like "unturning" and "piling", can be programmed, but it would be convenient to give them names.
  - **let** $unturn\ x = turn\ (turn\ (turn\ x))$
  - **let** $pile\ x\ y\ = unturn\ (sew\ (turn\ y)\ (turn\ x))$

  Such expressons/declarations are called *let-expressions* or *let-bindings*.
- Visually, $pile$ works as follows:



User-Defined Functions (cont.)

- The named operations can then be used without having to think about how they are implemented.
- After these declarations, $unturn$ *E*, for any expression *E*, is equivalent to $turn$ ($turn$ ($turn$ *E*)); similarly for $pile$.



- Once declared, a function can be used to declare others.

Local Declarations

- User-defined functions may be made *local* to a particular expression.
- Let-expressions allow declarations to appear within expressions in the following form:

  $$\textbf{let}\ \langle \textit{declaration} \rangle\ \textbf{in}\ \langle \textit{expression} \rangle$$

  where ⟨*declaration*⟩ equates a user-defined name/function with its defining expression.
- An example:

  $\textbf{let}\ unturn\ x = turn\ (turn\ (turn\ x))\ \textbf{in}$
  $\textbf{let}\ pile\ x\ y = unturn\ (sew\ (turn\ y)\ (turn\ x))\ \textbf{in}$
  $pile\ (unturn\ b)\ (turn\ b)$

User-Defined Names for Values

- Frequently-used expressions/values can also be given names as follows.

$$\textbf{let}\ \langle \mathit{name} \rangle = \langle \mathit{expression} \rangle$$

- They may be seen as user-defined functions without parameters (a.k.a. constants).
- Examples:
  - **let** $x = \mathit{turn}\ b$
  - **let** $y = \mathit{sew}\ (\mathit{turn}\ a)\ (\mathit{turn}\ (\mathit{turn}\ b))$

User-Defined Names for Values (cont.)

IM NTU

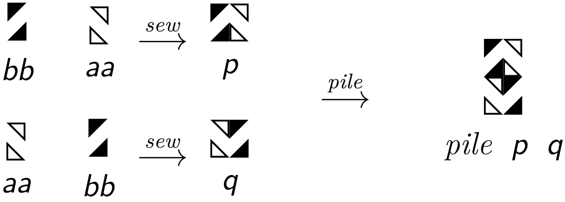
- Value declarations may also be made local.
- An expression of the form

$$\textbf{let } x = E_1 \textbf{ in } E_2$$

  means: occurrences of name x in E_2 represent the value of E_1. Any other name can be used instead of x without changing the meaning of the expression.
- The expression $\mathit{pile}\ (\mathit{unturn}\ b)\ (\mathit{turn}\ b)$ can be rewritten as
  $\textbf{let } bnw = \mathit{unturn}\ b \textbf{ in } \mathit{pile}\ bnw\ (\mathit{turn}\ b)$
  or as

  $\textbf{let } bnw = \mathit{unturn}\ b \textbf{ in}$
  $\textbf{let } bse = \mathit{turn}\ b \textbf{ in}$
  $\mathit{pile}\ bnw\ bse$

Specificatin of a Quilt



$$\textbf{let } \mathit{unturn}\ x = \mathit{turn}\ (\mathit{turn}\ (\mathit{turn}\ x))\ \textbf{in}$$
$$\textbf{let } \mathit{pile}\ x\ y = \mathit{unturn}\ (\mathit{sew}\ (\mathit{turn}\ y)(\mathit{turn}\ x))\ \textbf{in}$$
$$\textbf{let } aa = \mathit{pile}\ a\ (\mathit{turn}\ (\mathit{turn}\ a))\ \textbf{in}$$
$$\textbf{let } bb = \mathit{pile}\ (\mathit{unturn}\ b)\ (\mathit{turn}\ b)\ \textbf{in}$$
$$\textbf{let } p = \mathit{sew}\ bb\ aa\ \textbf{in}$$
$$\textbf{let } q = \mathit{sew}\ aa\ bb\ \textbf{in}$$
$$\mathit{pile}\ p\ q$$

CFG of Little Quilt

$$
\begin{array}{rcl}
\langle\text{expression}\rangle & ::= & a \mid b \\
\langle\text{expression}\rangle & ::= & (\mathit{turn}\ \langle\text{expression}\rangle) \mid \\
 & & (\mathit{sew}\ \langle\text{expression}\rangle\ \langle\text{expression}\rangle) \\
\langle\text{expression}\rangle & ::= & \textbf{let}\ \langle\text{declaration}\rangle\ \textbf{in}\ \langle\text{expression}\rangle \\
\langle\text{declaration}\rangle & ::= & \langle\text{name}\rangle = \langle\text{expression}\rangle \\
\langle\text{expression}\rangle & ::= & \langle\text{name}\rangle \\
\langle\text{declaration}\rangle & ::= & \langle\text{name}\rangle\ \langle\text{formals}\rangle = \langle\text{expression}\rangle \\
\langle\text{formals}\rangle & ::= & \langle\text{name}\rangle \mid \langle\text{name}\rangle\ \langle\text{formals}\rangle \\
\langle\text{expression}\rangle & ::= & \langle\text{name}\rangle\ \langle\text{actuals}\rangle \\
\langle\text{actuals}\rangle & ::= & \langle\text{expression}\rangle \mid \langle\text{expression}\rangle\ \langle\text{actuals}\rangle
\end{array}
$$

Types

- A *type* consists of a set of elements called *values* together with a set of functions called *operations*.
- Types are denoted by *type expressions*.
- We will consider methods for defining structured values such as products, lists, and functions. Structured values can be used freely in functional languages as basic values like integers and strings.
- Values in a functional language take advantage of the underlying machine, but are not tied to it.
- Common categories of types:
  - Basic types
  - Products of types
  - Lists of elements
  - Functions from a domain to a range

Type Expressions

$$
\begin{array}{rcl}
\langle \text{type-expr} \rangle & ::= & \langle \text{type-name} \rangle \\
& | & \langle \text{type-expr} \rangle \rightarrow \langle \text{type-expr} \rangle \\
& | & \langle \text{type-expr} \rangle \ * \ \langle \text{type-expr} \rangle \\
& | & \langle \text{type-expr} \rangle \ \textbf{list}
\end{array}
$$

Basic Types

- Values
  A type is *basic* if its values are atomic, i.e., if the values are treated as whole elements, with no internal structure.
  For example, the *boolean* values in the set $\{\textbf{true}, \textbf{false}\}$ are basic values.
- Operations
  Basic values have no internal structure, so the only operation defined for *all* basic types is a comparison of equality.
  For example, the equality $2 = 2$ is true and the inequality $2 \neq 2$ is false.

Basic Types of ML

The predeclared basic types of ML include **boolean**, **int**, **float**, **char**, and **string**.

type	*name*	*values*	*operations*
boolean	**bool**	`true`, `false`	=, <>, $\cdots$
integer	**int**	`-1`, `0`, `2`	=, <>, <, +, *, /, `mod`, $\cdots$
real	**float**	`0.`, `3.14`	=, <>, <, `+.`, `*.`, `/.`, $\cdots$
character	**char**	`'A'`, `'b'`	=, <>, $\cdots$
string	**string**	`"Abc"`	=, <>, $\cdots$

Products

- Values
  The *product* $A * B$ of two types A and B consists of *ordered pairs* written as (a, b), where a is a value of type A and b is a value of type B.
  A *product* of n types $A_1 * A_2 * \cdots * A_n$ consists of *tuples* written as $(a_1, a_2, \cdots, a_n)$, where a_i is a value of type A_i, for $1 \leq i \leq n$.
- Operations
  Associated with pairs are operations called *projection functions* to extract the first and second elements from a pair.
  They can be defined in ML as follows:
  let $\mathit{first}\ (x, y) = x$
  let $\mathit{second}\ (x, y) = y$

Lists

- Values
  A *list* is a finite-length sequence of elements.
  The type "*A* **list**" consists of all lists of elements, where each element belongs to type *A*. For example, **int list** consists of all lists of integers.
  In ML, list elements are written between brackets "[" and "]", and separated by semicolons ";". The *empty* list is written as [ ].
- Operations

List.hd x	The first or head element of list *x*.
List.tl x	The tail of list *x* after removing the first element.
$a :: x$	Construct a list with head *a* and tail *x*.

$[1; 2; 3] = 1 :: [2; 3] = 1 :: 2 :: [3] = 1 :: 2 :: 3 :: [\,]$
The cons operator :: is right associative; e.g., $1 :: 2 :: [3]$ is equivalent to $1 :: (2 :: [3])$.

Functions

- Values

  The type $A \rightarrow B$ consists of all functions from A to B.

  A function f in $A \rightarrow B$ is *total* if it is defined at each element of A. A is called the *domain* and B the *range* of f. Function f is said to *map* elements of its domain to elements of its range.

  A function f in $A \rightarrow B$ is *partial* if it need not be defined at each element of A.

- Operations

  A key operation associated with the set $A \rightarrow B$ is *application*, which takes a function f in $A \rightarrow B$ and an element a in A, and yields an element b of B.

  In ML, the application of f to a is written as $f\ a$.

  Parentheses do not affect the value of an expression, so $f\ a$ is equivalent to $f\,(a)$ and to $(f\ a)$.

  Application is left associative; $f\ a\ b$ is equivalent to $(f\ a)\ b$, the application of $f\ a$ to b.

Types in ML

- New basic types can be defined as needed by enumerating their elements in a **type** declaration. For example,
  `type direction = NE | SE | SW | NW;;`
  The names `NE`, `SE`, `SW`, and `NW` are called *value constructors*, or simply *constructors*, of type `direction`; they construct elements of `direction` out of nothing.
- Type constructors (in order of increasing precedence):

type	*constructor*	*notation*	*example*
function	`->`	infix	`int -> bool`
product	`*`	infix	`int*int`
list	`list`	postfix	`string list`

- A *type* declaration gives a name to a type. For example,
  `type intpair = int*int`
  makes `intpair` a synonym for `int*int`.

Quilts in ML

- A quilt is a list of rows.
- A row is a list of squares.
- A square has a texture and a direction.
- Call the textures *WTriangle* and *BTriangle*.
- Call the directions *NE*, *SE*, *SW*, and *NW*.
- This view leads to the following representation:
  - `type texture = WTriangle | BTriangle`
  - `type direction = NE | SE | SW | NW`
  - `type square = texture*direction`
  - `type row = square list`
  - `type quilt = row list`

Quilts in ML (cont.)

 `[[(WTriangle,NE)]]`

 `[[(BTriangle,NE)]]`

 `[[(WTriangle,NE); (BTriangle,NE)]]`

```
[[(WTriangle,NE); (BTriangle,NE)];
 [(BTriangle,SW); (WTriangle,SW)]]
```

Functions Declarations

- An expression is formed by applying a function or operation to subexpressions. Once a function is declared, it can be applied as an operator within expressions.
- A function declaration has three parts:
  1. The name of the declared function
  2. The parameters of the function
  3. A rule for computing a result from the parameters
- The basic syntax for function declaration is

  $$\textbf{let}\ \langle \textit{name} \rangle\ \langle \textit{formal-parameter} \rangle = \langle \textit{body} \rangle$$

  Example:

  ```
  # let successor n = n + 1;;
  val successor : int -> int = <fun>
  ```
- The syntax for function application is

  $$\langle \textit{name} \rangle\ \langle \textit{actual-parameter} \rangle$$

  Example: *successor* $(2 + 3)$

Recursive Functions

- A function f is *recursive* if its body contains an application of f. More generally, a function f is recursive if f can activate itself, possibly through other functions.
- Examples:

  let rec len $x =$
  if $x = [\ ]$ **then** 0 **else** $1 + len\ (List.tl\ x)$

  let rec fib $n =$
  if $n = 0 \ || \ n = 1$ **then** 1
  else $fib\ (n-2) + fib\ (n-1)$

Innermost Evaluation

- Under the *innermost-evaluation* rule, the evaluation of a function application

  $$\langle \textit{name} \rangle \ \langle \textit{actual-parameter} \rangle$$

  proceeds as follows:
  1. Evaluate the expression represented by $\langle \textit{actual-parameter} \rangle$.
  2. Substitute the result for the formal in the function body.
  3. Evaluate the body.
  4. Return its value as the answer.
- Each evaluation of a function body is called an activation of the function.
- The approach of evaluating arguments before the function body is also referred to as *call-by-value* evaluation. Call-by-value can be implemented efficiently, so it is widely used.
- Under call-by-value, all arguments are evaluated, whether their values are needed or not.

Selective Evaluation

- The ability to evaluate selectively some parts of an expression and ingore others is provided by the construct

$$\textbf{if}\ \langle \mathit{condition} \rangle\ \textbf{then}\ \langle \mathit{expr}_1 \rangle\ \textbf{else}\ \langle \mathit{expr}_2 \rangle$$

- Either $\langle \mathit{expr}_1 \rangle$ or $\langle \mathit{expr}_2 \rangle$ is evaluated, not both.

Outermost Evaluation

- Under the *outermost-evaluation* rule, the evaluation of a function application

$$\langle \mathit{name} \rangle \ \langle \mathit{actual\text{-}parameter} \rangle$$

  proceeds as follows:
  1. Substitute the actual (without evaluating it) for the formal in the function body.
  2. Evaluate the body.
  3. Return its value as the answer.
- Innermost and outermost evaluations produce the same result if both terminate with a result.
- The distinguishing difference between the evaluation methods is that actual parameters are evaluated as they are needed in outermost evaluation; they are not evaluated before substitution.
- OCaml uses call-by-value or innermost evaluation.

Short-Circuit Evaluation

- The operators && (andalso) and || (orelse) perform *short-circuit evaluation* of boolean expressions, in which the right operand is evaluated only if it has to be.
- Expression "E && F" is false if E is false; it is true if both E and F are true. The evaluation of "E && F" proceeds from left to right, with F being evaluated only if E is true.
- The evaluation of "E || F" is true if E evaluates to true. F is skipped if E is true.
- So, the evaluation of "**true** || F" always terminates even if F leads to a nonterminating computation.
- For a language using innermost evaluation, the operator || has to be provided by the language. It cannot be user-defined as part of a program.

Lexical Scope

- Bound occurrences of variables can be renamed without changing the meaning of a program. For example,
  let $successor\ x = x + 1$
  let $successor\ n = n + 1$
  This renaming principle is the basis for the *lexical scope rule* for determining the meanings of names in programs.
- When a function declaration refers to a name that is not a formal parameter, the value of that name has to be determined by some context.
- Lexical scope rules use the program text surrounding a function declaration to determine the context in which nonlocal names are evaluated. The program text is static in contrast to run-time execution, so such rules are also called static scope rules.

Let Bindings: Names

- The occurrence of x to the right of keyword **let** in

$$\textbf{let } x = E_1 \textbf{ in } E_2$$

  is called a *binding occurrence* or simply *binding of x*. All occurrences of x in E_2 are said to be within the *scope* of this binding; the scope of a binding includes itself.
- The occurrences of x within the scope of a binding are said to be *bound*. A binding of a name is said to be *visible* to all occurrences of the name in the scope of the binding.
- Occurrences of x in E_1 are not in the scope of this binding of x.

Let Bindings: Names (cont.)

- Determining the scopes of the two binding occurrences of x in the following expression may be challenging to a beginner:

$$\textbf{let } x = 2 \textbf{ in let } x = x + 1 \textbf{ in } x * x$$

- The value of an expression is left undisturbed if we replace all occurrences of a variable x within the scope of a binding of x by a fresh variable.

$$\textbf{let } x = 2 \textbf{ in let } y = x + 1 \textbf{ in } y * y$$

Let Bindings: Functions

- The occurrences of f and x to the right of **let** or **let rec** in

$$\textbf{let}\ f\,x = E_1\ \textbf{in}\ E_2$$

or

$$\textbf{let rec}\ f\,x = E_1\ \textbf{in}\ E_2$$

are *bindings* of f and x.
- The binding of the formal parameter x is visible only to the occurrences of x in E_1.
- The binding of the function name f is visible to the occurrences of f in E_2, and the **let rec** binding of f is *also* visible in E_1.
- Example: **let** $x = 2$ **in let** $f\,x = x + 1$ **in** $f\,x$

Simultaneous Bindings

- Mutually recursive functions require the simultaneous binding of more than one function name.
- In

  let rec $f_1\ x_1 = E_1$
  and $f_2\ x_2 = E_2$ **in**
  E

  the scope of both f_1 and f_2 includes E_1, E_2, and E. The scopes of the formal parameters x_1 and x_2 are, as usual, limited to the respective function bodies.

Simultaneous Bindings (cont.)

```
# let rec even x =
    if x=0 then true
    else if x=1 then false
    else odd (x-1)
  and odd x =
    if x=0 then false
    else if x=1 then true
    else even (x-1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>

# (even 2, odd 2);;
- : bool * bool = (true, false)
```

Type Checking

- Type distinctions between values carry over to expressions.
- A *type system* for a language is a set of rules for associating a type with expressions in the language. A type system *rejects* an expression if it does not associate a type with the expression.
- Wherever possible, ML infers the type of an expression. An error is reported if the type of the expression cannot be inferred.
- At the heart of all type systems is the following rule for function application:

  If f is a function of type $A \to B$, and a has type A, then $(f\ a)$ has type B.

Type Equivalence

- Two type expressions are *structurally equivalent* if and only if they are equivalent under the following rules:
  1. A type name is structurally equivalent to itself.
  2. Two type expressions are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types.
  3. After a type declaration, **type** $n = T$, the type name n is structurally equivalent to T.
- ML uses structural equivalence of types.

Type Equivalence (cont.)

```
# [[(WTriangle,NE)]];;
- : (texture * direction) list list =
[[(WTriangle, NE)]]
```

The type of this expression is structurally equivalent to the type name *quilt* declared as follows:

```
type square = texture*direction;;
type row = square list;;
type quilt = row list;;
```

Coercion: Implicit Type Conversion

- A *coercion* is a conversion from one type to another, inserted automatically by a programming language.

```
# 2 * 3.14;;
Characters 4-8:
  2 * 3.14;;
      ^^^^
Error: This expression has type float but
an expression was  expected of type int
```

- Type conversions must be specified explicitly in ML because the language does not coerce types.

```
# float(2);;
- : float = 2.
```

Polymorphism: Parameterized Types

- For all lists, the function *List.hd* returns the head or first element of a list:

```
# List.hd [1;2;3];;
- : int = 1
# List.hd ["a";"b";"c"];;
- : string = "a"
```

- What is the type of *List.hd*?

```
# List.hd;;
- : 'a list -> 'a = <fun>
```

- ML uses a leading quote, as in `'a`, to identify a type parameter.
- ML is known for its support for *polymorphic* functions, which can be applied to parameters of more than one type.