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 are 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.
COMPUTING WITH EXPRESSIONS

Example expressions:

2 An integer constant
x A variable
log n Function log applied to n
2+3 Function + applied to 2 and 3

Expressions can also include conditionals and function definitions.

if x ≥ y then x else y

A LITTLE LANGUAGE OF EXPRESSIONS

A quilt is:

– One of the primitive pieces , or (b) 
– It is formed by turning a quilt clockwise 90° or
– It is formed by sewing a quilt to the right of another quilt of equal height
– Nothing else is a quilt
OPERATIONS ON QUILTS

QUILTS MADE UP OF SIMPLER PIECES

(a)  

(b)
Let a be , and b be . Then, the BNF syntax for quilt expressions is as follows:

\[
\text{<expression>} ::= a \mid b \mid \text{turn (}<expression>\text{)} \mid \text{sew (}<expression> <expression>\text{)}
\]

The semantics of an quilt expression specifies the quilt denoted by the expression \( \text{turn (sew (turn (turn (a) ) , b) )} \equiv \)

---

**USER-DEFINED FUNCTIONS**

\[
\text{fun } \text{untrun}(x) = \text{turn (turn (turn (x) ))}
\]

\[
\text{fun } \text{pile}(x, y) = \text{turn (sew (untrun (x) , unturn (y) ))}
\]

Once declared, a function can be used to declare others.
**LOCAL DECLARATIONS**

```ml
let <declaration> in <expression> end

let fun unturn(x)=turn(turn(turn(x)))
  fun pile(x,y) = turn(sew(unturn(x),unturn(y)))
  in
    pile(unturn(b),turn(b))
  end
```

**USER-DEFINED NAMES FOR VALUES**

A value declaration
```
val <name> = <expression>
```
gives a name to a value. Value declarations are used together with let-bindings. An expression of the form
```
let val x = E_1 in E_2 end
```
Any other name can be used instead of x without changing the meaning of the expression.

```
let val bnw = unturn(b)
  val bse = turn(b)
  in
    pile(bnw,bse)
  end
```
SPECIFICATION OF A QUILT

let fun unturn (x) = turn (turn (turn (x)))
    fun pile (x, y) = unturn (sew (turn (y), turn (x)))
    val aa = pile (a, turn (turn (a)))
    val bb = pile (unturn (b), turn (b))
    val p = sew (bb, aa)
    val q = sew (aa, bb)
    in pile (p, q)
end

REVIEW: DESIGN OF A LITTLE QUILT

The specification of the language Little Quilt can be found in Fig. 8.6. Its main constructs are borrowed from ML.
**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.

```
integer = { ⋯ -2, -1, 0, 1, 2, ⋯ }
```

“2 is of type integer” means “2 ∈ { ⋯ -2, -1, 0, 1, 2, ⋯ }”

Conventionally, {false, true} is boolean type.

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.

Common categories of types:
- Basic Types
- Products of Types
- Lists of Elements
- Functions from a Domain to a Range

**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 `{true, 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≠2` is false.
PRODUCTS

Values

The product $A \times B$ 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 $A_1 \times A_2 \times \ldots \times A_n$ consists of tuples written as $(a_1, a_2, \ldots, a_n)$, 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:

```
fun first(x, y) = x;
fun 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 commas. The empty list is written as `[ ]` or as `nil`.

Operations

```
null(x) True if x is the empty list.
hd(x) The first or head element of list x.
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.
```
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 $b$.

PREDECLARED BASIC TYPES IN ML

The predeclared basic types of ML include boolean, int, real,
and string.
New basic types can be defined as needed by enumerating
their elements in a datatype declaration. For example,

datatype 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.

A type declaration gives a name to a type. For example,
type intpair = int*int;
makes intpair a synonym for int*int.
## TYPE CONSTRUCTORS IN ML

Type constructors (in order of increasing precedence):

<table>
<thead>
<tr>
<th>type</th>
<th>constructor</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>-&gt;</td>
<td>int -&gt; bool</td>
</tr>
<tr>
<td>product</td>
<td>*</td>
<td>int*int</td>
</tr>
<tr>
<td>list</td>
<td>list</td>
<td>string list</td>
</tr>
</tbody>
</table>

## 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 **arcs** and **bands**.
Call the directions **ne**, **se**, **sw**, and **nw**.

This view leads to the following representation:

```ml
datatype texture = arcs | bands;
datatype direction = ne | se | sw | nw;
type square = texture*direction;
type row = square list;
type quilt = row list;
```
FUNCTION 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:
- The name of the declared function
- The parameters of the function
- A rule for computing a result from the parameters

The basic syntax for function declaration is
```
fun <name><formal-parameter> = <body>;
```

Example:
```
fun successor n = n + 1;
```

An alternative form: `fun successor(n) = n + 1;`

The syntax for function application is
```
<name> <actual-parameter>
```

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:

\[
\text{fun } \text{len}(x) = \\
\text{ if } \text{null}(x) \text{ then } 0 \text{ else } 1 + \text{len(tl}(x));
\]

\[
\text{fun } \text{fib}(n) = \\
\text{ if } n=0 \text{ or else } n=1 \text{ then } 1 \\
\text{ else } \text{fib}(n-1) + \text{fib}(n-2);
\]

INNERMOST EVALUATION

Under the innermost-evaluation rule, the evaluation of a function application \(<\text{name}> <\text{actual-parameter}>\)
proceeds as follows:

– Evaluate the expression represented by \(<\text{actual-parameter}>\).
– Substitute the result for the formal in the function body.
– Evaluate the body.
– 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 ignore others is provided by the construct

\[
\text{if } \langle\text{condition}\rangle \text{ then } \langle\text{expression}_1\rangle \\
\text{else } \langle\text{expression}_2\rangle;
\]

OUTERMOST EVALUATION

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

\[
\langle\text{name}\rangle \langle\text{actual-parameter}\rangle
\]

proceeds as follows:

– Substitute the actual (without evaluating it) for the formal in the function body.
– Evaluate the body.
– Return its value as the answer.

Innermost and outermost evaluation 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.

Standard ML uses call-by-value or innermost evaluation.
SHORT-CIRCUIT EVALUATION

The operators andalso and orelse in ML perform short-circuit evaluation of boolean expressions, in which the right operand is evaluated only if it has to be.

E andalso F is false if E is false; it is true if both E and F are true. The evaluation of E andalso F proceeds from left to right, with F being evaluated only if E is true.

So E andalso F may terminate even if F does not.

The evaluation of E orelse F is true if E evaluates to true. F is skipped if E is true.

The 91-function

fun f(x)=
if x > 100 then x-10 else f(f(x+11))

Innermost Evaluation

f(100) = if 100>100 then 100-10 else f(f(100+11))
= f(f(100+11))
= f(f(111))
= f(if 111>100 then 111-10 else f(f(111+11)))
= f(111-10)
= f(101)
= if 101>100 then 101-10 else f(f(101+11))
= 101-10
= 91
The 91-function (cont.)

Outermost Evaluation

\[
\begin{align*}
f(100) &= \text{if } 100 > 100 \text{ then } 100 - 10 \text{ else } f(f(100+11)) \\
&= f(f(100+11)) \\
&= \text{if } f(100+11) > 100 \text{ then } f(100+11) - 10 \text{ else } f(f(f(100+11)+11))
\end{align*}
\]

For simplicity, the next few lines show only the evaluation of \((100+11)\):
\[
\begin{align*}
f(100+11) &= \text{if } 100 + 11 > 100 \text{ then } 100 + 11 - 10 \text{ else } f(f(100+11)+11)) \\
&= \text{if } 111 > 100 \text{ then } 100 + 11 - 10 \text{ else } f(f(100+11)+11)) \\
&= 100 + 11 - 10 \\
&= 111 - 10 \\
&= 101
\end{align*}
\]

Returning to the evaluation of \(f(100)\):
\[
\begin{align*}
f(100) &= \text{if } 100 > 100 \text{ then } f(100+11) - 10 \text{ else } f(f(f(100+11)+11)) \\
&= f(100+11) - 10 \\
&= \ldots \\
&= 91
\end{align*}
\]

OUTERMOST vs. INNERMOST

“Outermost” appears to do more work than “innermost”. “Outermost” can terminate where “innermost” fails.

\[
\text{fun } \mathsf{or}(x,y) = \\
\quad \text{if } x \text{ then } \mathsf{true} \text{ else } y;
\]

- Under innermost-evaluation, both subexpressions \(E\) and \(F\) in \(\mathsf{or}(E,F)\) are evaluated before they are substituted into the function body. So \(\mathsf{or}(\mathsf{true}, F)\) results in a nonterminating computation if the evaluation of \(F\) does not terminate.

- Under outermost-evaluation, \(\mathsf{or}(\mathsf{true}, F) = \text{if } \mathsf{true} \text{ then } \mathsf{true} \text{ else } F;\)

so the computation terminates regardless of the evaluation of \(F\) terminates or not.

Since ML uses innermost evaluation, the operator \(\mathsf{or}\) 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,

```plaintext
fun successor(x) = x + 1;
fun successor(n) = n + 1;
```

This renaming principle is the basis for the **lexical scope rule** for determining the meanings of names in programs.

```plaintext
fun addy(x) = x + y;
```

What is \( y \)? 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**.

---

VAL BINDINGS

The occurrence of \( x \) to the right of keyword **val** in

```plaintext
let val x = E_1 in E_2 end
```

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.
**VAL BINDINGS (cont.)**

```ml
let val x = 2 in let val y = x+1 in x*x end end
```

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.

```ml
let val x = 2 in
  let val y = x+1 in y*y end end
```

---

**FUN BINDINGS**

The occurrences of `f` and `x` to the right of keyword `fun` in

```ml
let fun f(x) = E₁ in E₂ end
```

are bindings of `f` and `x`.

This binding of the formal parameter `x` is visible only to the occurrences of `x` in `E₁`.

This binding of the function name `f` is visible to the occurrences of `f` in both `E₁` and `E₂`.

```ml
let x = 2 in
  let fun f(x) = x+1 in f(f(x)) end end
```
NESTED BINDINGS

Sequences of val and fun bindings are treated as nested bindings. Thus,

\[
\text{let val } x_1 = E_1 \\
\quad x_2 = E_2 \\
\text{in } E \text{ end}
\]

is treated as if the individual bindings were nested:

\[
\text{let val } x_1=E_1 \text{ in let val } x_2=E_2 \text{ in } E \text{ end end}
\]

This approach generalizes to any sequence of val and fun bindings.

SIMULTANEOUS BINDINGS

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

\[
\text{let fun } f_1(x_1) = E_1 \\
\quad \text{and } f_2(x_2) = E_2 \\
\text{in } E \text{ end}
\]

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.

\[
\text{let fun even(x)=} \\
\quad \text{if } x=0 \text{ then true else} \\
\quad \quad \text{if } x=1 \text{ then false else} \quad \text{odd(x-1)} \\
\quad \text{and odd(x)=} \\
\quad \quad \text{if } x=0 \text{ then false else} \\
\quad \quad \quad \text{if } x=1 \text{ then true else} \quad \text{even(x-1)} \\
\text{in (even(24), odd(24)) end}
\]
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.

Why we need a type system for each language?
To detect program errors as early as possible.

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 $A \rightarrow B$, 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:

- A type name is structurally equivalent to itself.
- Two type expressions are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types.
- After a type declaration, type $n=T$, the type name $n$ is structurally equivalent to $T$.

ML uses structural equivalence of types.

```
- [(arcs,ne)];
  val it = [(arcs,ne)] : (texture * direction) list list
```

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;
```
OVERLOADING

A symbol is **overloaded** if it has different meanings in different contexts. Familiar operator symbols like + and * are overloaded.

When ML cannot resolve overloading, it complains, as in

```
- fun add(x,y) = x+y;
  stdIn:10.17 Error: overloaded variable cannot be resolved: +
```

Explicit types can then be given to resolve overloading.

```
- fun add(x,y):int = x+y;
  val add = fn : int * int -> int
- fun add(x,y) = x+(y:int);
  val add = fn : int * int -> int
```

COERCION: IMPLICIT TYPE CONVERSION

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

```
- 2*3.142;
  stdIn:1.1 Error: expected integer type,
  found type: real
```

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

```
- real(2);
  val it = 2.0 : real
```
For all lists, the function $hd$ returns the head or first element of a list:

- $hd \ [1,2,3]$;
  
  $val \ it = 1 : \ int$

- $hd \ ["a","b","c"]$;
  
  $val \ it = "a" : \ string$

What is the type of $hd$?

- $hd$;
  
  $val \ it = fn : \ 'a \ list \ -> \ 'a$

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.