Principles of programming languages Lecture 3

http://few.vu.nl/~nsilvis/PPL/2007

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

vrije Universiteit *amsterdam*

Outline

- Part I. Language systems
- Part II. Functional programming. First look at ML.

Part I

Language systems

Imagine:

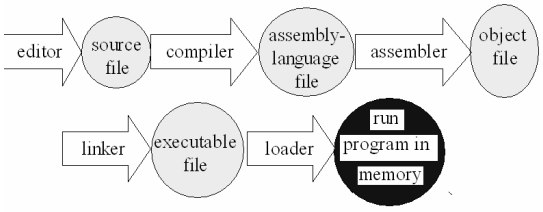
A new language have been designed and your task is to **implement** this language.

You have to build a **language system**

Language systems

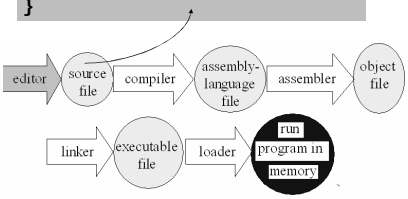
- The classical sequence
- Variations on the classical sequence
- Binding
- Debuggers
- Runtime support

The classical sequence



1. Creating

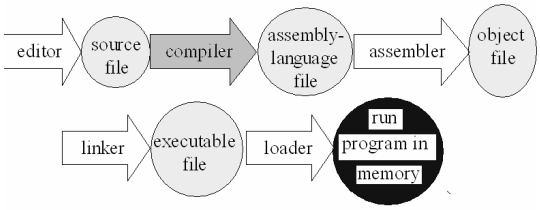
```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```



1. Creating

- The programmer uses an editor to create a text file containing the program
- A high-level language: machine independent

2. Compiling



2. Compiling

- Compiler translates to assembly language
- Machine-specific
- Each line represents either a piece of data, or a single machine-level instruction
- Programs used to be written directly in assembly language, before Fortran (1957)
- Now used directly only rarely

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

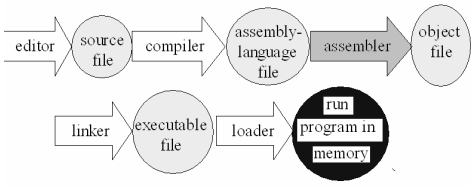
C

compiler

```
i:     data word 0
main:  move 1 to i
t1:    compare i with 100
       jump to t2 if greater
       push i
       call fred
       add 1 to i
       go to t1
t2:    return
```

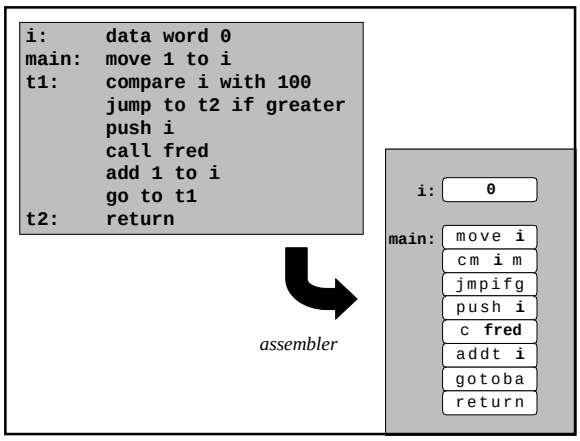
assembler

3. Assembling

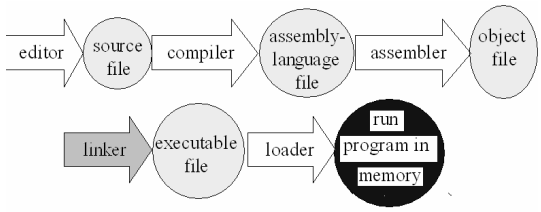


3. Assembling

- Assembly language is still not directly executable
  - Still text format, readable by people
  - Still has names, not memory addresses
- Assembler converts each assembly-language instruction into the machine's binary format: its *machine language*
- Resulting object file not readable by people

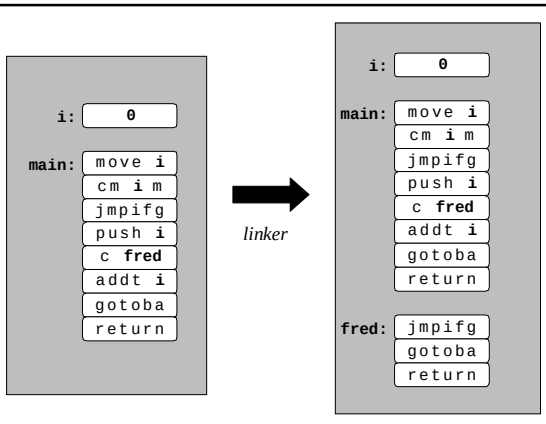


4. Linking

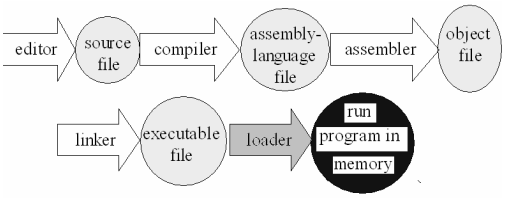


4. Linking

- Object file *still* not directly executable
  - Missing some parts
  - Still has some names
  - Mostly machine language, but not entirely
- Linker collects and combines all the different parts
- In our example, **`fred`** was compiled separately, and may even have been written in a different high-level language
- Result is the executable file

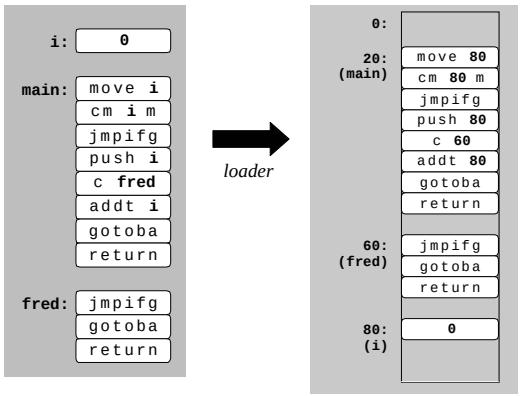


5. Loading

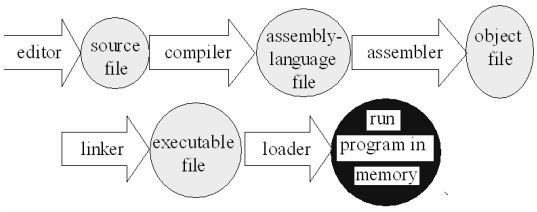


5. Loading

- "Executable" file *still* not directly executable
  - Still has some names
  - Mostly machine language, but not entirely
- Final step: when the program is run, the loader loads it into memory and replaces names with addresses



6. Running



6. Running

- After loading, the program is entirely machine language
- Processor begins executing its instructions, and the program runs

Optimization

- Usually after compilation step
- Code generated by a compiler is usually *optimized* to make it faster, smaller, or both
- First used for Fortran in 1957
- Other optimizations may be done by the assembler, linker, and/or loader
- A misnomer: the resulting code is better, but not guaranteed to be optimal

Example: loop invariant removal

- Original code:

```
int i = 0;
while (i < 100) {
  a[i++] = x*x*x;
}
```

- Improved code, with loop invariant moved:

```
int i = 0;
int temp = x*x*x;
while (i < 100) {
  a[i++] = temp;
}
```

```
#include <stdio.h>
double powern (double d, unsigned n)
{ double x = 1.0;
unsigned j;
for (j = 1; j <= n; j++) x *= d;
return x; }

int main (void)
{ double sum = 0.0;
unsigned i;
for (i = 1; i <= 100000000; i++)
 { sum += powern (i, i % 5); }
printf ("sum = %g\n", sum); return 0;
}
```

$ gcc -Wall -O0 test.c -lm	13.370 s
$ gcc -Wall -O1 test.c -lm	10.030 s
$ gcc -Wall -O2 test.c -lm	8.380 s
$ gcc -Wall -O3 test.c -lm	6.730 s
$ gcc -Wall -O3 -funroll-loops test.c -lm	5.390 s

Language systems

- The classical sequence
- **Variations on the classical sequence**

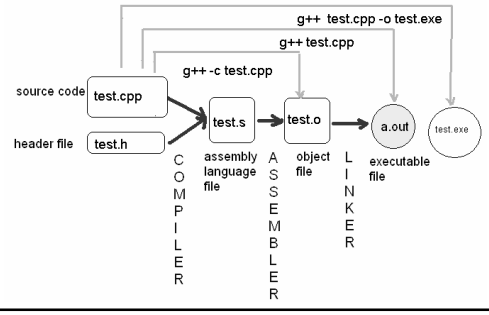
Variations on the classical sequence

- Hiding steps
- Interpreters
- Virtual machines (JVM=Java virtual machine)
- Delayed linking (.dll in Windows)
- Dynamic compilation (JIT=just in time compilation)

The gcc C++ example

- http://www.network-theory.co.uk/docs/gccintro/

Hiding steps example: the GNU C++ language system



Language systems

- The classical sequence
- Variations on the classical sequence
- **Binding**

What is binding?

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Binding

Association between two things (like a name and a property)

Example:

What set of values is associated with **int**?

What is the type of **fred**?

What is the address of the object code for **main**?

What is the value of **i**?

Binding Times

- Binding time = the time when a binding is created (or an implementation decision is made, or an answer to a question is given)
- Very important for a language semantics
- A binding can happen at different times:
  - Language design time
  - Language implementation time
  - Compile time
  - Link time
  - Load time
  - Runtime

1.Language Design Time

- Some properties are bound when the language is defined:
  Ex: Meanings of keywords: **`void`**, **`for`**

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

2. Language Implementation Time

- Some properties are bound when **the language system** is written:
  - range of values of type **`int`** in C (but in Java, these are part of the language definition)
  - implementation limitations: max identifier length, max number of array dimensions, etc

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

3. Compile Time

- Some properties are bound when the program **is compiled** or prepared for interpretation:
  - Types of variables, in languages like C and ML that use static typing
  - Declaration that goes with a given use of a variable, in languages that use static scoping (most languages)

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

4. Link Time

- Some properties are bound when separately-compiled program parts are combined into one executable file by the linker:
  - Object code for external function names

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

5. Load Time

- Some properties are bound when the program is loaded into the computer's memory, but before it runs:
  - Memory locations for code for functions
  - Memory locations for static variables

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

6. Run Time

- Some properties are bound only when the code in question is executed:
  - Values of variables
  - Types of variables, in languages like Lisp that use dynamic typing
- Binding at run time is called *late* or *dynamic* binding
- Binding before run time is called *early* or *static*

Practice (Exercise 3, Ch.4)

What is the binding time for each of the following in a Java program?

§The location in memory of a local variable in a method?

§The location in memory of a non-static field in a class?

§The meaning of the keyword while?

§The size in memory of a variable of type int?

§The type of a local variable in a function?

§The value assigned to a variable?

Practice (Exercise 3, Ch.4)

What is the binding time for each of the following in a Java program?

§The location in memory of a local variable in a method?	Runtime
§The location in memory of a non-static field in a class?	Runtime
§The meaning of the keyword while?	Language definition
§The size in memory of a variable of type int?	Language definition
§The type of a local variable in a function?	Compile time
§The value assigned to a variable?	Run time

Trade-off

- Early Binding vs. Late Binding
- Early: faster and more secure at runtime (less to do, less that can go wrong) (compilers)
- Late: flexible at runtime (as with types, dynamic loading, etc.) (interpreters)

Language systems

- The classical sequence
- Variations on the classical sequence
- Binding
- Debuggers
- **Runtime support**

Runtime Support

- Additional code the linker includes even if the program does not refer to it explicitly
  - Startup processing: initializing the machine state
  - Exception handling: reacting to exceptions
  - Memory management: allocating memory, reusing it when the program is finished with it
  - Operating system interface: communicating between running program and operating system for I/O, etc.
- An important hidden player in language systems

Part II

Functional languages: A first look to ML

Functional languages

- emphasize the definition and application of mathematical functions, in contrast to imperative programming, which emphasizes the execution of sequential commands
- well-defined semantics. Make much easier to formally prove the correctness of functional programs
- based on lambda λ calculus
- Examples: Lisp, ML, Scheme, Haskell, Miranda
- Read more about : http://www.cs.nott.ac.uk/~gmh//faq.html
- Discussions forum: `comp.lang.functional`

y = f(x) * f(x)

Can you say it is the same with:

z = f(x)

y = z*z ;

?

About ML

- ML=Meta Language used for a theorem prover
- Edinburgh, 1974, Robin Milner's group
- There are a number of dialects
- We are using **Standard ML** (installed on Linux machines)
- Good book: Ulmann, Elements of ML programming
- Another popular dialect is OCaml = Objective Caml (used by Inleiding in Theoretische Informatica)

Getting started

To run SML in interactive mode, type to the Unix prompt

```
$sml
```

Type an expression after **-** prompt; ML replies with value and type

Don't forget ; after the expression!

Variable **it** is a special variable that is bound to the value of the expression you type

```
Standard ML of New Jersey
- 1+2*3;
val it = 7 : int
```

Outline

- Constants
- Operators
- Variables
- Tuples and Lists
- Functions

Constants

```
- 1234;
val it = 1234 : int
- 123.4;
val it = 123.4 : real
- true;
val it = true : bool
- "fred";
val it = "fred" : string
- "H";
val it = "H" : string
- #"H";
val it = #"H" : char
```

int : Integer constants:

real : Real constants

bool: Boolean constants **true** and **false**

string: String constants: text inside double quotes

char: Character constants: put **#** before a 1-character string

ML is case-sensitive

Operators

```
-~ 1 + 2 - 3 * 4 div 5 mod 6;
val it = ~1 : int
- ~ 1.0 + 2.0 - 3.0 * 4.0 / 5.0;
val it = ~1.4 : real
-"bibity" ^ "bobity" ^ "boo";
val it = "bibitybobityboo" : string
```

Standard operators for integers, using **~** for unary negation and **-** for binary subtraction

Same operators for reals, but use **/** for division

String concatenation: **^** operator

Left associative, precedence is {**+**,**-**} < {*****,**/**,**div**,**mod**} < {**~**}.

Ordering operators

```
- 2 < 3;
val it = true : bool
- 1.0 <= 1.0;
val it = true : bool
- #"d" > #"c";
val it = true : bool
- "abce" >= "abd";
val it = false : bool
```

Ordering comparisons: **<**, **>**, **<=**, **>=**, apply to **string**, **char**, **int** and **real**

Order on strings and characters is lexicographic

Equality comparison operators

```
- 1 = 2;
val it = false : bool
- true <> false;
val it = true : bool
- 1.3 = 1.3;
Error: operator and operand don't agree
           [equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    1.3 = 1.3
```

Most types are equality testable: these are *equality types*

Type real is not an equality type

Boolean operators

```
- 1 < 2 orelse 3 > 4;
val it = true : bool
- 1 < 2 andalso not (3 < 4);
val it = false : bool
```

Boolean operators: **andalso**, **orelse**, **not**. (And we can also use **=** for equivalence and **<>** for exclusive or.)

Precedence so far: {**orelse**} < {**andalso**} < {**=**,**<>**,**<**,**>**,**<=**,**>=**} < {**+**,**-**,**^**} < {*****,**/**,**div**,**mod**} < {**~**,**not**}

Short circuiting operators

```
- true orelse 1 div 0 = 0;
val it = true : bool
```

andalso and **orelse** are **short-circuiting operators** =

if the first operand of **orelse** is true, the second is not evaluated;

likewise if the first operand of **andalso** is false

Conditional expressions

```
- if 1 < 2 then #"x" else #"y";
val it = #"x" : char
- if 1 > 2 then 34 else 56;
val it = 56 : int
- (if 1 < 2 then 34 else 56) + 1;
val it = 35 : int
```

Conditional expression (not statement) using **if** … **then** … **else** …

Similar to C's ternary operator: **(1<2) ? 'x' : 'y'**

Value of the expression is the value of the **then** part, if the test part is true, or the value of the **else** part otherwise

There is no **if** … **then** construct

Overloaded operators

```
- 1 * 2;
val it = 2 : int
- 1.0 * 2.0;
val it = 2.0 : real
- 1.0 * 2;
Error: operator and operand don't agree
[literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    1.0 * 2
```

The ***** operator, and others like **+** and **<**, are *overloaded* to have one meaning on pairs of integers, and another on pairs of reals

Type conversions

ML does not perform implicit type conversion

ML has builtin conversion functions: **real** (**int** to **real**), **floor** (**real** to **int**), **ceil** (**real** to **int**), **round** (**real** to **int**), **trunc** (**real** to **int**), **ord** (**char** to **int**), **chr** (**int** to **char**), **str** (**char** to **string**)

```
- real(123);
val it = 123.0 : real
- floor(3.6);
val it = 3 : int
- floor 3.6;
val it = 3 : int
- str #"a";
val it = "a" : string
```

A little about functions

```
- square 2+1;
val it = 5 : int
- square (2+1);
val it = 9 : int
```

- **Function application has higher precedence than any operator**
- **Function application is left-associative**

Defining variables

```
- val x = 1+2*3;
val x = 7 : int
- x;
val it = 7 : int
- val y = if x = 7 then 1.0 else 2.0;
val y = 1.0 : real
```

val creates a binding

val defines a new variable and binds it to a value.

ML has No assignment

val definition in ML is not the same as assignment in C or Java.

val defines a **new** variable and binds it to a value BUT it does not change the old definition. It adds a new definition op top of the other.

```
- val radius = 4.0;
val  radius = 4.0 : real
- radius;
val it = 4.0 : real
- val radius = 5.0;
val  radius = 5.0 : real
- radius;
val it = 5.0 : real
```

Functions have no side effects

```
-val x = 0 ;
val x=0; int
-fun inc(x) = x+1 ;
val inc = fn : int -> int
- inc x;
val it = 1 : int
- inc x;
val it = 1 : int
```

The value of x is not changed by the function = referential transparency

Tuples

Tuple = collection of values of different types (like **barney,point1**)

Tuples can contain other tuples

To get i'th element of a tuple x, use **#i x**

There is no such thing as a tuple of one

```
- val barney = (1+2, 3.0*4.0, "brown");
val barney = (3,12.0,"brown") : int * real * string
- val point1 = ("red", (300,200));
val point1 = ("red",(300,200)) : string * (int *
int)
- #2 barney;
val it = 12.0 : real
- #1 (#2 point1);
val it = 300 : int
```

Tuple Type Constructor

- ML gives the type of a tuple using * as a type constructor
- For example, **int * bool** is the type of pairs (x,y) where x is an **int** and y is a **bool**
- Note that parentheses have structural significance here: **int * (int * bool)** is not the same as **(int * int) * bool**, and neither is the same as **int * int * bool**

Lists

List = collection of values of the same type

```
-[1,2,3];
val it = [1,2,3] : int list
- [1.0,2.0];
val it = [1.0,2.0] : real list
- [(1,2),(1,3)];
val it = [(1,2),(1,3)] : (int * int) list
- [[1,2,3],[1,2]];
val it = [[1,2,3],[1,2]] : int list list
- [];
val it = [] : 'a list
- nil;
val it = [] : 'a list
```

Empty list is **[]** or **nil**. The type of the empty list is **'a list**

'a list means *a list of elements, type unknown*

The **null** test

```
- null [];
val it = true : bool
- null [1,2,3];
val it = false : bool
```

- **null** tests whether a given list is empty
- You could also use an equality test, as in
  x = []
- However, **null x** is preferred; we will see why in a moment

List Type Constructor

- ML gives the type of lists using **list** as a type constructor
- For example, **int list** is the type of lists of things, each of which is of type **int**
- A list is not a tuple

§The **@** operator concatenates 2 lists of the same type

```
- [1,2,3]@[4,5,6];
val it = [1,2,3,4,5,6] : int list
```

§List-builder (*cons*) operator is **::**

It takes an element of any type, and a list of elements of that same type, and produces a new list by putting the new element on the front of the old list

```
- val x = #"c"::[];
val x = [#"c"] : char list
- val y = #"b"::x;
val y = [#"b",#"c"] : char list
 - val z = 1::2::3::[];
val z = [1,2,3] : int list
```

The **::** operator is right-associative

The **hd** function gets the head of a list: the first element

The **tl** function gets the tail of a list: the whole list after the first element

The **explode** function converts a string to a list of characters, and the **implode** function does the reverse

```
- hd z;
val it = 1 : int
- tl z;
val it = [2,3] : int list
- tl(tl z);
val it = [3] : int list
- tl(tl(tl z));
val it = [] : int list
- explode "hello";
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
- implode [#"h",#"i"];
val it = "hi" : string
```

Defining functions

```
- fun firstChar s = hd (explode s);
val firstChar = fn : string -> char
- firstChar "abc";
val it = #"a" : char
```

fun defines a new function and binds it to a variable

Here **fn** means a function, the thing itself, considered separately from any name we've given it. The value of **firstChar** is a function whose type is **string -> char**

Function Definition Syntax

<fun-def> ::=
fun *<function-name>* *<parameter>* **=** *<expression>* **;**

- *<function-name>* can be any legal ML name
- The simplest *<parameter>* is just a single variable name: the formal parameter of the function
- The *<expression>* is any ML expression; its value is the value the function returns
- This is only a subset of ML function definition syntax;

Function Type Constructor

- ML gives the type of functions using **->** as a type constructor
- For example, **int -> real** is the type of a function that takes an **int** parameter (the *domain type*) and produces a **real** result (the *range type*)

```
- fun quot(a,b) = a div b;
val quot = fn : int * int -> int
- quot (6,2);
val it = 3 : int
- val pair = (6,2);
val pair = (6,2) : int * int
- quot pair;
val it = 3 : int
```

All ML functions take exactly one parameter

To pass more than one thing, you can pass a tuple

Example 1

```
- fun fact n =
=   if n = 0 then 1
=   else n * fact(n-1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int
```

Recursive factorial function

Example 2

```
- fun listsum x =
=   if null x then 0
=   else hd x + listsum(tl x);
val listsum = fn : int list -> int
- listsum [1,2,3,4,5];
val it = 15 : int
```

Recursive function to add up the elements of an **int list**

A common pattern: base case for **null x**, recursive call on **tl x**

Example 3

```
- fun length x =
=   if null x then 0
=   else 1 + length (tl x);
val length = fn : 'a list -> int
- length [true,false,true];
val it = 3 : int
- length [4.0,3.0,2.0,1.0];
val it = 4 : int
```

Recursive function to compute the length of a list

Note type: this works on any type of list. It is *polymorphic*.

```
- fun badlength x =
=   if x=[] then 0
=   else 1 + badlength (tl x);
val badlength = fn : ''a list -> int
- badlength [true,false,true];
val it = 3 : int
- badlength [4.0,3.0,2.0,1.0];
Error: operator and operand don't agree
  [equality type required]
```

Same as previous example, but with **x=[]** instead of **null x**

Type variables that begin with two apostrophes, like **''a**, are restricted to equality types. ML insists on that restriction because we compared **x** for equality with the empty list.

That's why you should use **null x** instead of **x=[]**. It avoids unnecessary type restrictions.

Example 4

```
- fun reverse L =
=   if null L then nil
=   else reverse(tl L) @ [hd L];
val reverse = fn : 'a list -> 'a list
- reverse [1,2,3];
val it = [3,2,1] : int list
```

Recursive function to reverse a list

That pattern again

Summary

- Constants and primitive types: **`int`**, **`real`**, **`bool`**, **`char`**, **`string`**
- Operators: **`~`**, **`+`**, **`-`**, **`*`**, **`div`**, **`mod`**, **`/`**, **`^`**, **`::`**, **`@`**, **`<`**, **`>`**, **`<=`**, **`>=`**, **`=`**, **`<>`**, **`not`**, **`andalso`**, **`orelse`**
- Conditional expression
- Function application
- Predefined functions: **`real`**, **`floor`**, **`ceil`**, **`round`**, **`trunc`**, **`ord`**, **`chr`**, **`str`**, **`hd`**, **`tl`**, **`explode`**, **`implode`**, and **`null`**

Summary, Continued

- Defining new variable bindings using **`val`**
- Tuple construction using **`(x,y,…,z)`** and selection using **`#n`**
- List construction using **`[x,y,…,z]`**
- Type constructors **`*`**, **`list`**, and **`->`**
- Function declaration using **`fun`**

It's easy; it's "fun"

All real programming in ML is conducted by definition of functions and application of these functions to arguments.

Functional languages are also called applicative languages

Practice

What is the value and ML type for each of these expressions?

```
1 * 2 + 3 * 4
"abc" ^ "def"
if (1 < 2) then 3.0 else 4.0
1 < 2 orelse (1 div 0) = 0
```

What is wrong with each of these expressions?

```
10 / 5
#"a" = #"b" or 1 = 2
1.0 = 1.0
if (1<2) then 3
```

Practice

Suppose we make these ML declarations:

```
val a = "123";
val b = "456";
val c = a ^ b ^ "789";
val a = 3 + 4;
```

Then what is the value and type of each of these expressions?

```
a
b
c
```

Practice

What are the values of these expressions?

```
#2(3,4,5)
hd(1::2::nil)
hd(tl(#2([1,2],[3,4])));
```

What is wrong with the following expressions?

```
1@2
hd(tl(tl [1,2]))
[1]::[2,3]
```