CSE 341
Lecture 1

Programming Languages; Intro to ML

Reading: Ullman 1.1; 2; 3 - 3.2

slides created by Marty Stepp

http://www.cs.washington.edu/341/

Programming languages

- **programming language**: A system of communication designed to express computations to be performed, presumably by a computer.
  - syntax, semantics, type system
  - libraries, specifications, implementations
  - idioms (how is the language typically used?)
  - user base, references

- Why learn general features vs. specific languages?
- What does learning, for example, ML teach us about Java (or about languages in general)?

Programming language timeline

- 1951 - Regional Assembly Lang
- 1952 - Autocode
- 1954 - FORTRAN
- 1958 - ALGOL
- 1958 - LISP
- 1959 - COBOL
- 1960 - ALGOL 60
- 1962 - APL
- 1964 - BASIC
- 1964 - PL/I
- 1970 - Pascal
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- **1973 - ML**
- **1975 - Scheme**
- 1978 - SQL
- 1980 - C++
- 1983 - Objective-C
- 1983 - Ada
- 1986 - Erlang
- 1987 - Perl
- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- **1993 - Ruby**
- 1993 - Lua
- 1995 - Java
- **1995 - JavaScript**
- 1995 - PHP
- 1999 - D
- 2001 - C#
- 2002 - F#
- 2003 - Scala
- 2007 - Clojure, Groovy
- 2009 - Go

http://en.wikipedia.org/wiki/History_of_programming_languages

Another timeline

category	1960s	1970s	1980s	1990s	2000s
scientific	Fortran			Matlab	
business	Cobol	DBMSes	SQL	VB	
functional	Lisp	ML, Scheme	Erlang	Haskell	F#
imperative/ procedural	Algol	Pascal, C, Smalltalk	Ada, C++	Java	C#
scripting	BASIC		Perl	Python, Ruby, PHP, JavaScript	
logical		Prolog	CLP(R)		

Functional programming

- **imperative/procedural programming**: views a program as a sequence of commands or *statements*

- **functional programming**: views a program as a sequence of *functions* that call each other as *expressions*
  - seen by some as an unintuitive or esoteric style
  - but many of its features are "assimilated" by other langs
    - functional constructs in F#, C#, .NET 3.0
    - closures, lambdas, generics, garbage collection in Java
    - MapReduce algorithm at Google

ML

- **ML (meta-language)**: A general-purpose functional programming language created in 1973 by Robin Milner et. al. from University of Edinburgh
  - created for developing advanced "lambda calculus" proofs
  - pioneered "statically typed" functional programming langs
  - known for clean syntax, elegant type system and design
  - criticized by some for being functionally "impure"
  - good textbook and supporting materials

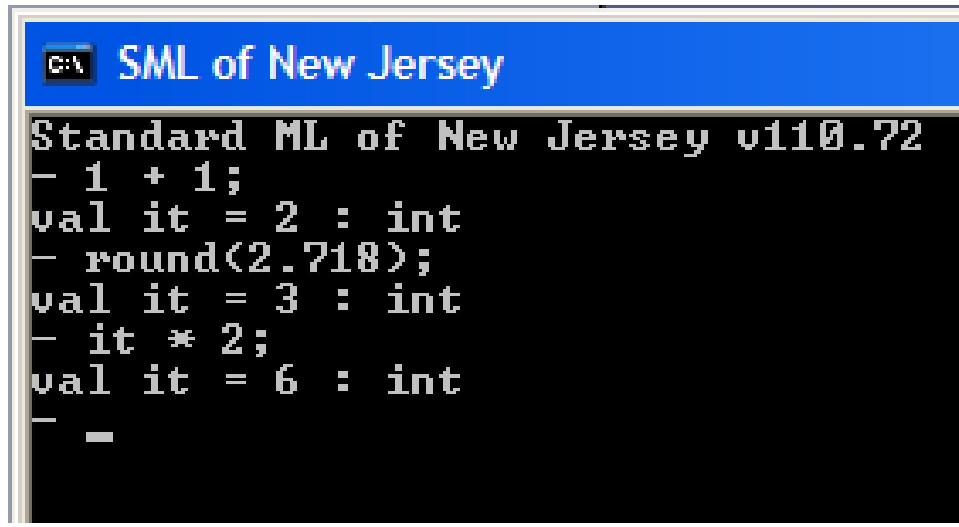
- dialects: SML, Caml/OCaml, LML, F# (Microsoft .NET)

Core features of ML

- functional
- heavily recursive
- higher-order functions
- static / strict type system
- rich abstract data types (ADTs)
- type inference
- polymorphic
- minimizing of *side effects*
  - makes code easier to parallelize
- rules and pattern matching
- garbage collection

The ML interpreter

- waits for you to type expressions, immediately evaluates them, and displays the result



- a read-evaluate-print loop ("REPL")
- similar to Interactions pane of jGRASP, DrJava, etc.
- useful for learning and practicing ML syntax, types

Using the interpreter

- type an expression at the `-` prompt; its result appears:

```
- 1 + 2 + 3;
```
← *don't forget the semicolon!*

val it = 6 : int

- special variable `it` stores the result of the last expression

```
- it * 2;
```
val it = 12 : int

- hotkeys: Press ↑ for previous command; ^C to abort;
  - ^Z (Unix/Mac) or ^D (Windows) to quit interpreter

Basic types (2.1)

name	description	Java	Example
`int`	integer	`int`	`3`
`real`	real number	`double`	`3.14`
`string`	multi-char. text	`String`	`"hello"`
`char`	single character	`char`	`#"Q"`
`bool`	logical true/false	`boolean`	`true`

other types

- `unit`, `tuple`, `list`, `function`, `record`

Operators

- same as Java
  - `+ - * /` basic math `int*int`, `real*real`
- different
  - `~` negation `int`, `real`
  - `div` integer division `int*int`
  - `mod` integer remainder `int*int`
  - `^` concatenation `string*string`

int and real

- cannot mix types
  - `1 + 2.3` is illegal! (why?)

- but you can explicitly convert between the two types
  - `real(`***`int`***`)` converts int to real
  - `round(`***`real`***`)` rounds a real to the nearest int
  - `ceil(`***`real`***`)` rounds a real UP to an int
  - `floor(`***`real`***`)` rounds a real DOWN to an int
  - `trunc(`***`real`***`)` throws away decimal portion

  - `real(1) + 2.3` is okay

Declaring a variable

```
val name: type = expression;
val name = expression;
```

- Example:

```
val pi: real = 3.14159;
```

- You may omit the variable's type; it will be *inferred*

```
val gpa = (3.6 + 2.9 + 3.1) / 3.0;
val firstName = "Daisy";
```

  - *identifiers*: ML uses very similar rules to Java
  - everything in ML (variables, functions, objects) has a type

The ML "environment"

- **environment**: view of all identifiers defined at a given point
  - defining a variable adds an identifier to the environment

`gpa`	`3.2`
`pi`	`3.14159`
`round`	*(function ...)*
`floor`	*(function ...)*
identifier	*value*
...	...

  - re-defining a variable replaces older definition (see 2.3.4)
    – different than assigning a variable a new value (seen later)

The if-then-else statement

`if` ***`booleanExpr`*** `then` ***`expr2`*** `else` ***`expr3`***

- Example:

```
- val s = if 7 > 10 then "big" else "small";
```

val s = "small" : string

- Java's if/else chooses between two (blocks of) *statements*
- ML's chooses between two *expressions*
  - more like the `? :` operator in Java

- there is no `if-then`; why not?

Logical operators

- similar to Java

`< <= >= >`	relational ops	`int*int`, `real*real`, `string*string`, `char*char`

- different

`=` `<>`	equality, inequality	`int*int`, `char*char`, `string*string`, `bool*bool`
`andalso`	AND &&	`bool*bool`
`orelse`	OR ||	`bool*bool`

Functions (3.1)

```
fun name(parameters) = expression;
```

- Example (typed into the interpreter):

```
- fun squared(x: int) = x * x;
```

val squared = fn : int -> int

- Many times parameter types can be omitted:

```
- fun squared(x) = x * x;
```

  - ML will *infer* the proper parameter type to use

More about functions

- In ML (and other functional languages), a function does not consist of a block of statements.
- Instead, it consists of an *expression.*
  - maps a *domain* of parameter inputs to a *range* of results
  - closer to the mathematical notion of a function

- Exercise: Write a function `absval` that produces the absolute value of a real number.

```
fun absval(n) = if n >= 0 then n else ~n;
```

  - *(ML already includes an abs function.)*

Recursion (3.2)

- functional languages in general do NOT have loops!
- repetition is instead achieved by **recursion**

- How would we write a `factorial` function in ML?

```
public static int factorial(int n) {  // Java
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Factorial function

```
fun factorial(n) =
    if n = 0 then 1
    else n * factorial(n - 1);
```

- has infinite recursion when you pass it a negative number (we'll fix this later)

Exercise

- Write a function named pay that reports a TA's pay based on an integer for the number of hours worked.
  - $8.50 for each of the first 10 hours worked
  - $12.75 for each additional hour worked
  - example: `pay(13)` should produce `123.25`

- `Solution:`

```
fun pay(hours) =
    if hours <= 10 then 8.50 * real(hours)
    else 85.00 + 12.75 * real(hours - 10);
```