Programming ~~Paradigms~~ Languages F28PL, Lecture 1

Jamie Gabbay

October 14, 2016

About me

My name is Murdoch James Gabbay—everybody calls me Jamie. I got a PhD in Cambridge in 2001. Since then I have been a professional researcher—that is, I make my living by discovering and proving mathematical certainties.

My job consists of 50% academic research (into logic and lambda-calculus), 50% teaching, and 50% administration.

Lectures are:

- Thursday at 13:15 in EM2.50 (lab/tutorial).
- Thursday at 15:15 in JN302.
- Friday at 10:15 in DB1.13.
- Friday at 13:15 in EM1.83.

The course webpage is
`http://www.macs.hw.ac.uk/~gabbay/F28PL/`.

About you

You are about twenty years old (unless you are a mature student). You are in your second year at university. You must:

- Turn up to lectures.
- Understand the course as it is delivered.
- Do not last-minute labs.

No exceptions.

Please note:

I don't teach. I offer learning opportunities. You aren't at school any more. You are at university.

It is up to you to take advantage of the Internet, exercise sheets, books, your colleagues, and your own intelligence. Your lecturers cannot and will not do this for you.

About the course

This course is ostensibly about programming languages, and it is indeed about three of them: ML, Python, and Prolog.

But there is a deeper message encoded in our putting these three very different languages together: they have different paradigms.

- ML and Prolog are declarative languages; there are two types of declarative languages:
  - ML is a functional programming language.
  - Prolog is a logic programming language.
- Python is an imperative programming language.

Let's do an exam question:

Q. Explain, with specific examples, the differences between imperative and declarative programming languages. (4 marks)

Give the specific examples! If you don't give 'em, I can't give you the marks.

Q. Comment on the different programming styles promoted by imperative and declarative programming languages. (2 marks)

Only 2 marks: be brief, but precise.

Q. Suggest one concrete advantage and one concrete disadvantage each to imperative vs. declarative programming. (6 marks)

'Concrete' means concrete, specific, precise. Vaguenessisms such as "It's better" score . . . zero.

6 marks. Hmm. Best give 3 distinct reasons each. Repeated reasons count only once!

Answers:

A. In an imperative such as Java (1 mark) language we instruct the computer what to do (1 mark). In a declarative language such as ML (1 mark) we instruct the computer what should be done, but without necessarily giving much detail how (1 mark).

A. Imperative languages promote a relatively detailed stepwise programming style. Declarative languages promote a relatively undetailed high-level style (precisely because we do not give details of execution).

A. Imperative: sometimes easier to hand-optimise e.g. for specific hardware (note specific example), harder to write and debug, more programmers familiar with imperative style.
Declarative: generally much quicker and easier to write, and also easier to automatically optimise. May run slower and may have poorer OS interface support, again because of the lack of detail. Programmers may be harder to find (but likely to be more productive once hired).

Imperative languages:

Imperative languages are concrete realisations of von Neumann machines (this is no coincidence!). They have:

- stored program
- memory
- associations between addresses and values.

A language is Church-Rosser when it doesn't matter what order you evaluate the instructions in. Imperative language instructions change memory—i.e. they change address/value association. Thus:

- Order of evaluation is fundamental. Imperative languages are not Church-Rosser.

Imperative languages:

Most languages you know are imperative: C, Java, Bash scripting, Fortran, Pascal, COBOL, Perl, Python, and so on.

You probably believed that 'Programming'='Imperative programming'. Prepare to enter a wider, brighter, crazier world.

Many languages are declarative: ML, Haskell, Erlang, Prolog, F#, and so on. Some of them are research languages. Some are industrial languages.

Imperative is not Church-Rosser

Program parts communicate by accessing common variables. Order determines result; e.g. swap x and y: Correct:

```
int x=3,y=2;    {(x,3),(y,2)}
t=x;            {(x,3),(y,2),(t,3)}
x=y;            {(x,2),(y,2),(t,3)}
y=t;            {(x,2),(y,3),(t,3)}
```

Exchange second two statements:Âă

```
int x=3,y=2;    {(x,3),(y,2)}
x=y;            {(x,2),(y,2)}
t=x;            {(x,2),(y,2),(t,2)}
y=t;            {(x,2),(y,2),(t,2)}
```

Exchange first two statements:

```
int x=3,y=2;    {(x,3),(y,2)}
t=x;            {(x,3),(y,2),(t,3)}
y=t;            {(x,3),(y,3),(t,3)}
x=y;            {(x,3),(y,3),(t,3)}
```

Imperative is not Church Rosser

Another example:

```
int inc(int * x)
{  return ++(*x);  }
```

`inc` changes `*x` as a side-effect.

```
int i = 0;
printf("%d\n",inc(&i)+i); ==> 2
printf("%d\n",i+inc(&i)); ==> 1
```

Declarative languages

Describe what is to be done, not how to do it. Two kinds of declarative language:

- Logic languages, like Prolog. These implement predicate calculus.
- Functional languages like ML and Haskell. These implement the lambda calculus.

So: imperative languages implement von Neumann machines, logic languages implement predicate calculus, and functional languages implement the λ-calculus.

Functional programming

What we don't have:

- No global state. All variables immutable (but we can use accumulators instead).
- No side-effects.
- Program often runs slower than well-optimised bug-free declarative code (if you can get it).
- Code may copy-with-changes instead of modify-in-place; inefficient for large structures.

What we do have is:

- Church-Rosser. Evaluation order irrelevant to final result.
- Easy to parallelise.
- Easier to formally certify (i.e. prove) programs correct.
- (Much) higher-level of programming. A little code can go a very long way. Less code = fewer errors.
- Easier to machine-optimise, thus shifting work from end-users (programmers) to compiler designers.

Running ML

I suggest this:

```
rlwrap sml
Standard ML of New Jersey v110.76 [built: Tue Oct 22 14:04:11 2
-
```

or

```
rlwrap poly
Poly/ML 5.2 Release
>
```

(To exit, type Control-D.)

Check out the Standard ML Basis Library
`http://sml-family.org/Basis/manpages.html`.

Running ML

Suggested program development cycle:

- Prepare program in file in one window
- Run SML system in another window
- While program not perfect do
  load program file into SML system
  if errors then
   change program in file & save file
  else
   test program
  if errors then
   change program in file & save file

Running ML

To load a file type:

```
- use "file_name";
```

File name is any valid file path enclosed in string quotes “...”.

By convention SML file names end with .sml.

To leave SML typoe Control-D.

Some ML features

- Strong types. You can't change the type associated with variable.
- Static typing. Types are checked at compile time. A program with a type error won't even compile.
- Parametric polymorphism. Type variables (similar to Java generics).
- Strict parameter passing / eager evaluation order. Parameters are evaluated before a function is applied to them, so that e.g. in `(fn x => x*x)(1+1)`, the calculation `1+1` is carried out once (for the one function application), not twice (for the two instances of `x`).
- Left to right evaluation. In `(1+1,2+2)`, `1+1` is calculated first, then `2+2`. In `st` (`s` applied to `t`) `s` is calculated first, then `t`.

Our first ML programs

```
- 42;
val it = 42 : int
- ~42;
> ~42 : int
val it = ~42 : int
- "Hello world";
val it = "Hello world" : string
```

Three programs: `42`, `~42`, and `"Hello world"`. They compute three values, which are the programs themselves.

Integers

`int` is the type of positive and negative integers.

```
- 42;
> 42 : int
- ~42;
> ~42 : int
```

~ is the unary minus/negation function.

How large is `int`?

Integers

How large is `int`? Depends!

```
- open Int;
[autoloading]
...
- maxInt;
val it = SOME 1073741823 : int option
- open Int64;
[autoloading]
...
- maxInt;
val it = SOME 9223372036854775807 : int option
- open LargeInt;
[autoloading]
...
- maxInt;
val it = NONE : int option
```

Check out `http://www.it.uu.se/edu/course/homepage/funpro/ht07/handout/f10-stdlib.html`.

Boring basic stuff for reference

`+` is addition. `*` is multiplication. `div` is integer division. `mod` is remainder. We group operations with brackets.

More interesting:

```
- op +;
val it = fn : int * int -> int
- (op +,op *,op div,op mod);
val it = (fn,fn,fn,fn)
  : (int * int -> int) * (int * int -> int) *
    (int * int -> int) * (int * int -> int)
```

Make sure you understand exactly what is going on here, including why we have to type `op`, and what the types mean, and why the interpreter displays `(fn,fn,fn,fn)`.

Precedence: `(...)` then `~` then `*` and `div` and `mod` then `+` and `-`. Left to right evaluation order.

So `~1+1` computes zero, not minus 2.

Real numbers

Much like integers; ad hoc polymorphism of + and $*$.

Differences: division on reals is `/` instead of `div`. Why? Because `div` is not division, it is division plus rounding down.

```
- op /;
val it = fn : real * real -> real
- 1E10;
val it = 10000000000.0 : real
- Real.maxFinite;
[autoloading]
...
val it = 1.79769313486E308 : real
- LargeReal.maxFinite;
[autoloading]
...
val it = 1.79769313486E308 : real
```

So `Real` is the same as `LargeReal` in the implementation on my machine. Largest floating point number is $1.79769313486 * 10^{308}$.

Ad hoc polymorphism

Also called overloading.

`~`, `+`, `-`, and `*` are overloaded for integers and reals.

Must be applied to two integers only or two reals only; try typing `1+1.0`.Âă

Conversion functions are available:

```
- real;
val it = fn : int -> real
- floor;
val it = fn : real -> int
```

We distinguish between `real` the type and `real:int->real` the function.

Functions

This is functional programming, after all.

Functions are written prefix. Evaluation is strict, meaning that given `st` (`s` applied to `t`) we evaluate `s`, then evaluate `t`, then evaluate the application.

So for instance, `(fn x => (x,x))(floor 6.789)` rounds 6.789 down once, not twice.

Function application takes high precedence. For instance `floor ~1` raises a type error, as does `floor real 1`. Try it.

You need to type `floor(~1)` and `floor(real 1)`.

`floor 12.3+4` is fine. `floor 12.3` evaluates to the integer `12` first.

(Much) more on functions later!

Strings

Type `string` of strings.

Any sequence of characters within quotes "...".

```
- "Hello␣World";
> "Hello␣World" : string
```

Escape sequences for non-printing characters:

- `\n` for newline,
- `\t` for tab.

Strings

There is a function `size : string -> int`.

What do you think it calculates? Yes, that's right.

```
- size "hello";
> 5 : int
```

There is a binary infix operator
`op ^:string*string -> string`.

What do you think it does? Yes, that's right.

```
- "Hello"^"␣"^"World";
> "Hello␣World" : string
```

The name of a function and its type are informative. Pay attention to them.

Chars

So if I tell you there's a type `char` and show you this

```
- #"a";
> #"a" : char
- chr;
val it = fn : int -> char
- ord;
val it = fn : char -> int
```

You should work out what this does. Let's take it further:

```
- open Char;
... opening Char
  type char = ?.char
  type string = ?.string
  val chr : int -> char
  val ord : char -> int
  val minChar : char
  val maxChar : char
  val maxOrd : int
  ...
```

Chars and strings

Note: a string is not a list of chars. But we do have functions:

```
- explode;
val it = fn : string -> char list
- implode;
val it = fn : char list -> string
- toString;
val it = fn : char -> string
```

What do you think they do? Why not try it out. For instance, type `explode "Hello world"` and see what you get.

For more innocent fun, try

- `(implode o rev o explode) "Hello world";` and
- ```
  (implode o rev o explode)
      "A man, a plan, a canal: Panama";
  ```
  ```

Booleans

Booleans are a type `bool`. Values are `true` and `false`. See also negation `not`:

```
> true;
val it = true : bool
> not;
val it = fn : bool -> bool
> not true;
val it = false : bool
```

Binary infix operators `andalso` (conjunction) and `orelse` (disjunction).

Precedence is `not`, then `andalso`, then `orelse`. This can matter!

```
> true orelse false andalso false;
val it = true : bool
> not true orelse false;
val it = false : bool
```

Booleans

Equality (is polymorphic and) generates boolean values:

```
> op =;
val it = fn : ''a * ''a -> bool
> 1 = 1;
val it = true : bool
> 1 = 2;
val it = false : bool
```

`andalso` will not bother to evaluate its second argument if its first argument evaluates to `false`.

`orelse` will not bother to evaluate its second argument if its first argument evaluates to `true`.

This can matter!

```
> true orelse (5 div 0 = 0);
val it = true : bool
> (5 div 0 = 0) orelse true;
Exception- Div raised
```

Tuples

A tuple is a collection of values of fixed length and type, muchÂăalike the fields of a Java object $(exp_1, exp_2, ..., exp_N)$.

```
> (1,1.0,"one");
val it = (1, 1.0, "one") : int * real * string
```

Select index:

```
> #1 (1,1.0,"one");
val it = 1 : int
> #2 (1,1.0,"one");
val it = 1.0 : real
> #3 (1,1.0,"one");
val it = "one" : string
> #4 (1,1.0,"one");
Error- ...
> #0 (1,1.0,"one");
Error-Labels must be 1,2,3,....
```

Tuples

Can be nested:

```
> (("Bianca","Castafiore"),"singer");
val it = (("Bianca", "Castafiore"), "singer") : (string * stri
```

Can pull nested indexes:

```
> #2 (#1 (("Cuthbert","Calculus"),"inventor"));
val it = "Calculus" : string
```

Equality types

An equality type is any type which allows equality testing.

All types except real, functions, and streams. `'a` (read α) ranges over all types; `''a` (read $\prime\alpha$) ranges over equality types.

Equality types have = and <>:

```
> op =;
val it = fn : ''a * ''a -> bool
> op <>;
val it = fn : ''a * ''a -> bool
```

Equality types

```
"banana" = "banana";
val it = true : bool
"banana" <> "banana";
val it = false : bool
true = true;
val it = true : bool
true = false;
val it = false : bool
(("Captain","Haddock"),"sailor") =
(("Captain","Haddock"),"sailor");
val it = true : bool
```

Reals are not an equality type; types must match:

```
> 1.0=1.0;
Error-Can't unify ''a with real
   (Requires equality type)
> 1=1.0;
Error-Can't unify Int32.int/int with real
```

Comparisons

Binary infix order operators returning `bool`:

```
> (op >,op <,op >=,op <=);
val it = (fn, fn, fn, fn)
: (int * int -> bool) * (int * int -> bool) *
  (int * int -> bool) * (int * int -> bool)
```

Comparisons

Overloaded for reals and strings by ad hoc polymorphism:

```
> 1<1;
val it = false : bool
> 1.0<1.0;
val it = false : bool
> "1"<"1";
val it = false : bool
> (op <):(real*real)->bool;
val it = fn : real * real -> bool
> (op <):(int*int)->bool;
val it = fn : int * int -> bool
> "1"<"1";
val it = false : bool
> (op <):(string*string)->bool;
val it = fn : string * string -> bool
```

Note: comparison of strings is lexicographic:

```
> "aa"<"z";
val it = true : bool
```

Comparisons

Precendence:

- (...) then
- function call, then
- arithmetic operator, then
- comparison, then
- boolean operator.

```
> 3*4>5*6;
val it = false : bool
```

not is a function so must bracket a comparison to negate it.Âă

```
> not 1<2;
Error
> not (1<2);
val it = false : bool
```