University of Auckland

415.360

Functional Programming

Christian Collberg

March 1, 1996

Introduction

Programming Paradigms I

A **programming paradigm**

- is a way to think about programs, programming, and problem solving,
- is supported by one or more programming languages.

Being familiar with several paradigms makes you a better programmer and problem solver. The most popular paradigms are

1. Imperative programming.
2. Functional programming.
3. Object-oriented programming.
4. Logic Programming.

- "When all you have is a hammer, everything looks like a nail."
- It's good to have many different tools.

Slide 1–1

Programming Paradigms II (a)

Imperative Programming

- Programming with **state**.
- Also known as **procedural programming**. The first to emerge in the 1940s-50s. Still the way most people learn how to program.
- FORTRAN, Pascal, C, BASIC.

Functional Programming

- Programming with **values**.
- Arrived in the late 50s with the LISP language. LISP is still popular and widely used by AI people.
- LISP, Miranda, Haskell, Gofer.

Slide 1–2

Programming Paradigms II (b)

Object-Oriented Programming

- Programming with **objects** that encapsulate data and operations.
- A variant of imperative programming first introduced with the Norwegian language Simula in the mid 60s.
- Simula, Eiffel, Modula-3, C++.

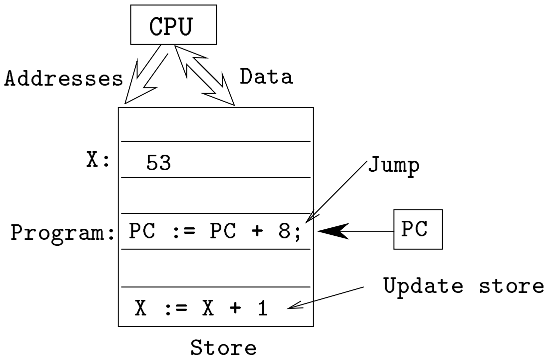
Logic Programming

- Programming with **relations**.
- Introduced in the early 70s. Based on predicate calculus. Prolog is popular with Computational Linguists.
- Prolog, Parlog.

Slide 1–3

Procedural Programming I

We program an abstraction of the Von Neumann Machine, consisting of a **store** (memory), a **program** (kept in the store), A **CPU** and a **program counter** (PC):



Computing `x:=x+1`

1. Compute `x`'s address, send it to the store, get `x`'s value back.
2. Add 1 to `x`'s value.
3. Send `x`'s address and new value to the store for storage.

Slide 1–4

Procedural Programming II

The programmer...

- uses **control structures** (IF, WHILE, ...) to alter the program counter (PC),
- uses **assignment statements** to alter the store.
- is in charge of **memory management**, i.e. declaring variables to hold values during the computation.

```
function fact (n:integer):integer;
var s,i :  integer := 1;
begin
   while i<=n do s:=s*i; i:=i+1; end;
   return s;
end fact.
```

Slide 1–5

Procedural Programming III

Procedural programming is difficult because:

1. A procedural program can be in a large number of states. (Any combination of variable values and PC locations constitutes a possible state.) The programmer has to keep track of all of them.
2. Any global variable can be changed from any location in the program. (This is particularly true of languages like C & C++ [Why?]).
3. It is difficult to reason mathematically about a procedural program.

Slide 1–6

Functional Programming

In contrast to procedural languages, functional programs don't concern themselves with state and memory locations. Instead, they work exclusively with **values**, and **expressions** and **functions** which compute values.

- Functional programming is not tied to the von Neumann machine.
- It is not necessary to know anything about the underlying hardware when writing a functional program, the way you do when writing an imperative program.
- Functional programs are more **declarative** than procedural ones; i.e. they describe **what** is to be computed rather than **how** it should be computed.

Slide 1–7

Functional Languages I

Common characteristics of functional programming languages:

1. Simple and **concise syntax** and semantics.
2. Repetition is expressed as **recursion** rather than iteration.
3. **Functions are first class objects**. I.e. functions can be manipulated just as easily as integers, floats, etc. in other languages.
4. **Data as functions**. I.e. we can build a function on the fly and then execute it. (Some languages).

Slide 1–8

Functional Languages II

5. **Higher-order functions**. I.e. functions can take functions as arguments and return functions as results.
6. **Lazy evaluation**. Expressions are evaluated only when needed. This allows us to build **infinit data structures**, where only the parts we need are actually constructed. (Some languages).
7. **Garbage Collection**. Dynamic memory that is no longer needed is automatically reclaimed by the system. GC is also available in some imperative languages (Modula-3, Eiffel) but not in others (C, C++, Pascal).

Slide 1–9

Functional Languages III

8. **Polymorphic types**. Functions can work on data of different types. (Some languages).
9. Functional programs can be more easily **manipulated mathematically** than procedural programs.

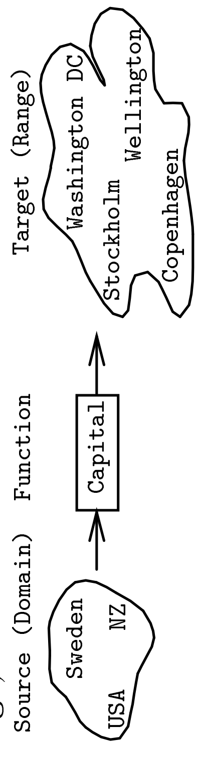
Pure vs. Impure FPL

- Some functional languages are **pure**, i.e. they contain no imperative features at all. Examples: Haskell, Miranda, Gofer.
- **Impure** languages may have assignment-statements, goto:s, while-loops, etc. Examples: LISP, ML.

Slide 1–10

What is a function?

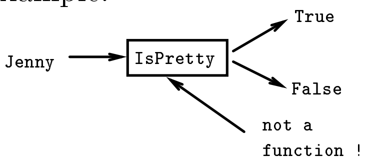
- A function **maps** argument values (inputs) to result values (outputs).
- A function takes argument values from a **source set** (or **domain**).
- A function produces result values that lie in a **target set** (or **range**).



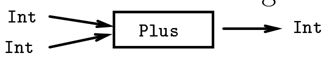
Slide 1–11

More on functions

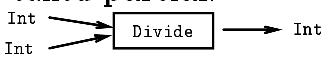
- A function must not map an input value to **more than one** output value. Example:



- If a function F maps every element in the domain to some element in the range, then F is **total**. I.e. a total function is defined for all arguments.



- A function that is undefined for some inputs, is called **partial**.



  `Divide` is partial since it is undefined when the numerator is 0 ($\frac{5}{0} = ?$).

Slide 1–12

Specifying functions I

A function can be specified in two ways, **extensionally** and **intentionally**.

Extensionally:

- Enumerate the elements of the (often infinite) set of pairs "`(argument, result)`". Another notation is "`Argument` $\mapsto$ `Result`".
- The extensional view emphasizes the **external behavior** (or **specification**), i.e. **what** the function does, rather than **how** it does it.

```
double = {···, (1,2), (5,10), ···}
even = {···, (0,True), (1,False), ···}
double = {···, 1↦2, 5↦10, ···}
isHandsome={Chris↦True,Hugh↦False}
```

Slide 1–13

Specifying functions II

Intensionally:

- Give a **rule** (i.e. **algorithm**) that computes the result from the arguments.
- The intentional view emphasizes the **process** (or algorithm) that is used to compute the result from the arguments.

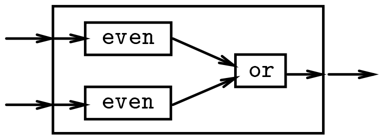
```
double x = 2 * x
even x = x mod 2 == 0
isHandsome x = if isBald x
               then True
               else False
```

Slide 1–14

Specifying functions III

Graphically:

- The graphical view is a notational variant of the intentional view.



eithereven

Slide 1–15

Application & Composition I

Function Application:

The most important operation in a functional program is **function application**, i.e. applying an ipout argument to the function, and retrieving the result. Function application is similar to Pascal's **function call**.

```
double 5 ⇒ 10
even 6 ⇒ True
```

Function Composition:

Another important operation is **function composition**, i.e. making the result of one function application the input to another application.

```
even (double 5) ⇒ 10
   even 10 ⇒
   True
```

Slide 1–16

Application & Composition II

Example: How many numbers are there between m and n, inclusive?

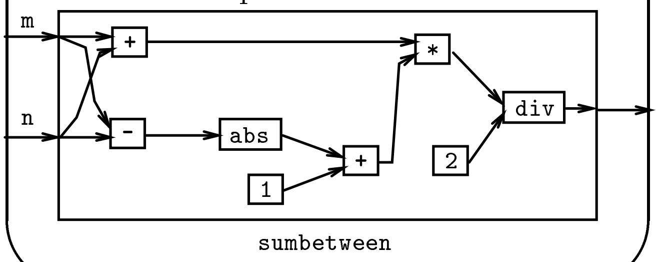
Extensional Definition:

`sumbetween m n =` $\{\cdots$
$(1,1) \mapsto 1, (1,2) \mapsto 2, \cdots, (2,10) \mapsto 9\}$

Intentional Definition:

```
sumbetween m n =
   ((m + n) * (abs (m-n) + 1)) div 2
```

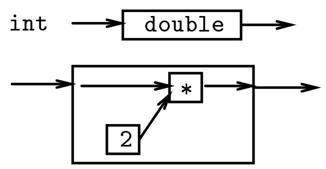
Graphical Definition:

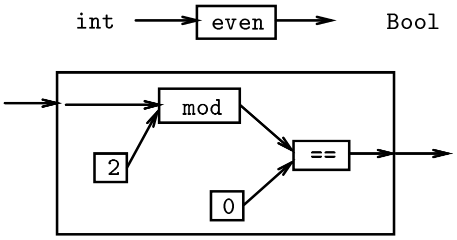


Slide 1–17

Function Signatures

When defining a function we usually need to give two pieces of information. First the **types** of the input and output sets (the domain and the range). This is the function's **signature**. Second, an algorithm that provides the mapping from inputs to outputs.





Slide 1–18

Referential Transparency I

- The most important concept of functional programming is **referential transparency**. Consider the expression
$$(2 * 3) + 5 * (2 * 3)$$
- $(2 * 3)$ occurs twice in the expression, but it **has the same meaning** (6) both times.
- RT means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs.
- This concept occurs naturally in mathematics, but is broken by imperative programming languages.
- RT makes functional programs easier to reason about mathematically.

Slide 1–19

Referential Transparency II

Pure functional programming languages are referentially transparent. This means that it is easy to find the meaning (value) of an expression. We can evaluate it **by substitution**. I.e. we can replace a function application by the function definition itself.

Evaluate `even (double 5)`:

```
double x = 2 * x
even x = x mod 2 == 0

even (double 5) ⇒
      even (2 * 5) ⇒
      even 10 ⇒
      10 mod 2 == 0 ⇒
      0 == 0 ⇒
      True
```

Slide 1–20

Referential Transparency III

So, isn't Pascal referentially transparent??? Well, sometimes, yes, but not always. If a Pascal function f has **side-effects** (updating a global variable, doing input or output), then $f(3) + f(3)$ may not be the same as $2 * f(3)$. I.e. The second $f(3)$ has a different meaning than the first one.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;

begin                    begin
   G := 0;                  G := 0;
   print f(3)+f(3);         print 2*f(3);
   {prints 4+5=9}           {prints 2*4=8}
end.                     end.
```

Slide 1–21

Summary

- B&W reading list:
  1. Preface: xi–xiii
  2. Functional programming: 1–6, 13–15
- This course will introduce two new programming paradigms, functional and logic programming.
- Functional programs use recursion to express repetition.
- We usually define a function by giving its intentional definition, i.e. an algorithm for computing its value.
- Function application is similar to function calls in imperative languages.
- Function composition is taking the output from one function and making it the input to another function.

Slide 1–22

Homework

- Here is a mathematical definition of the combinatorial function $\binom{n}{r}$ "n choose r", which computes the number of ways to pick r objects from n:

$$\begin{pmatrix} n \\ r \end{pmatrix} = \frac{n!}{r! * (n-r)!}$$

- Give an extensional, intentional, and graphical definition of the combinatorial function, using the notations suggested in this lecture.
- You may want to start by defining an auxiliary function to compute the factorial function, $n! = 1 * 2 * \cdots * n$.

Slide 1–23