

CSc 372

Comparative Programming Languages

2 : Functional Programming

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Christian Collberg

- During the next few weeks we are going to work with functional programming. Before I can explain to you what FP is, I thought I'd better put things into perspective by talking about other **programming paradigms**.
- Over the last 40 or so years, a number of programming paradigms (a programming paradigm is a way to think about programs and programming) have emerged.

Programming Paradigms...

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:

- 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.

Programming Paradigms...

_____ 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.

Programming Paradigms...

Object-Oriented Programming

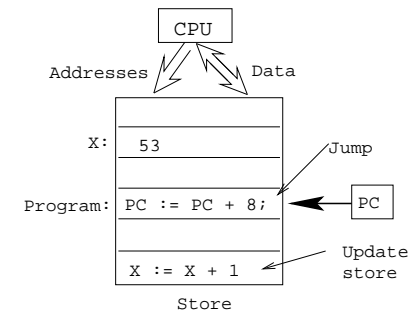
- Programming with **objects** that encapsulate data and operations.
- A variant of imperative programming first introduced with the Norwegian language Simula in the mid 60s.
- Simula, Eiffel, Modula-3, C++.

Logic Programming

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

Procedural Programming

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.
- 4 Increment PC.

Procedural Programming...

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.
```

Procedural Programming...

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.

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.

Functional Languages

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).

Functional Languages...

- 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 **infinite 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).

- ③ **Polymorphic types**. Functions can work on data of different types. (Some languages).
- ⑨ Functional programs can be more easily **manipulated mathematically** than procedural programs.

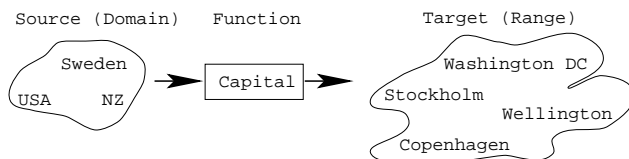
Pure vs. Impure FPL

- 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, Scheme.

Specifying Functions

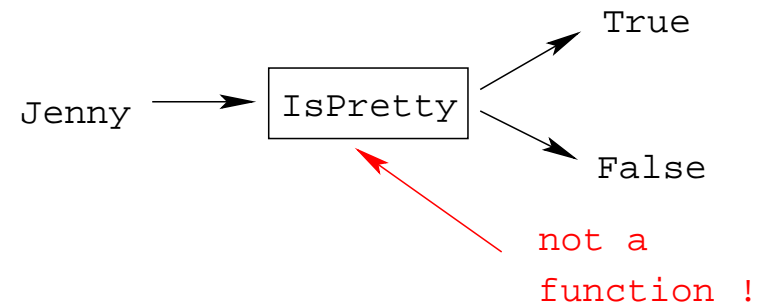
What is a function?

- 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**).



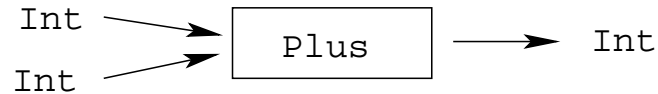
More on functions

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



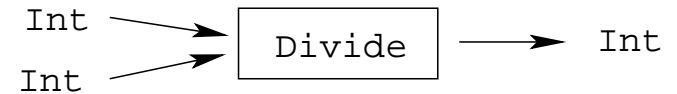
More on functions. . .

- 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.



More on functions. . .

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



- Divide is partial since $\frac{?}{0} = ?$ is undefined.

Specifying functions

A function can be specified **extensionally** or **intentionally**.

_____ Extensionally: _____

- Enumerate the elements of the (often infinite) set of pairs “(argument, result)” or “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 $\mapsto$ 2, 5 $\mapsto$ 10, ... }
isHandsome = { Chris $\mapsto$ True, Hugh $\mapsto$ False }
```

Specifying functions. . .

_____ Intentionally: _____

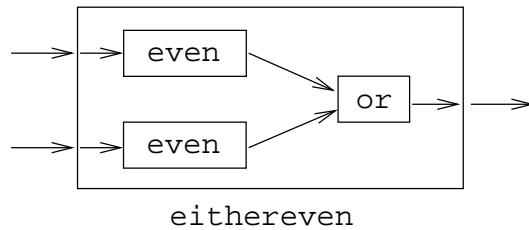
- Give a **rule** (i.e. **algorithm**) that computes the result from the arguments.
- The intentional view emphasizes the **process** (or algorithm) that is used to compute the result from the arguments.

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

Specifying functions...

Graphically: _____

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



Function Application

- The most important operation in a functional program is **function application**, i.e. applying a function to its argument(s), and retrieving the result:

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

```
double 5 ⇒ 10
even 6 ⇒ True
```

Function Composition

- Function composition** makes the result of one function application the input to another application:

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

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

Function Definition — Example

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

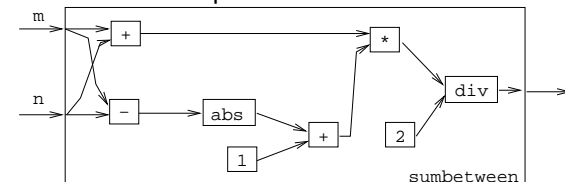
_____ Extensional Definition: _____

```
sumbetween m n = { ... (1,1) ↦ 1, (1,2) ↦ 2, ..., (2,10) ↦ 9 }
```

_____ Intentional Definition: _____

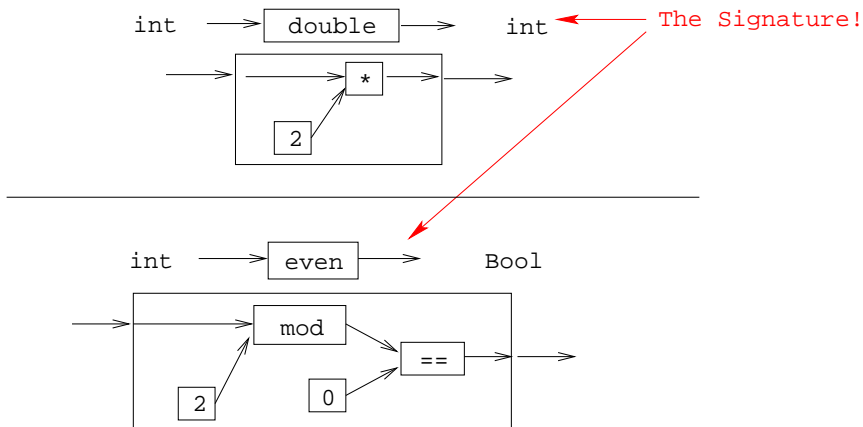
```
sumbetween m n = ((m + n) * (abs (m - n) + 1)) div 2
```

_____ Graphical Definition: _____



Function Signatures

To define a function we must specify the **types** of the input and output sets (domain and range, i.e. the function's **signature**), and an algorithm that maps inputs to outputs.



What's so Good About FP?

Referential Transparency

- 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.

Referential Transparency...

- Consider this Java expression:

```
f() + f()
```

- Could we replace it by the expression

```
2 * f()
```

- If this was mathematics, we could! But, in Java...

Referential Transparency...

- If our definition of $f()$ was

```
int f() {  
    return 5;  
}
```

then $f()+f()$ and $2*f()$ both mean the same.

- But, if $f()$ is

```
int X=5;  
int f() {  
    X++;  
    return X;  
}
```

then $f()+f()=6+7=13$ and $2*f()=2*6=12!$

Referential Transparency...

- What about these two Java expression:

```
f() + g()
```

and

```
g() + f()
```

- Are they equivalent? In math they are...

Referential Transparency...

- But, Java isn't math:

```
int X=5;  
int f() {  
    X++;  
    return X;  
}  
int g() {  
    return X;  
}
```

then $f()+g()=6+6=12$ and $g()+f()=5+6=11!$

Referential Transparency...

- Because of such side-effects, Java isn't referentially transparent.
- The same is true of any procedural language (Pascal, C, Modula-2, etc) and object-oriented language (Java, C++, C#).

Referential Transparency...

- 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.

Referential Transparency...

_____ 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
```

Referential Transparency...

In a pure functional language

- 1 Expressions and sub-expressions always have the same value, regardless of the environment in which they're evaluated.
- 2 The order in which sub-expressions are evaluated doesn't effect the final result.
- 3 Functions have no side-effects.
- 4 There are no global variables.

Referential Transparency...

- 5 Variables are similar to variables in mathematics: they hold a value, but they can't be updated.
- 6 Variables aren't (updatable) containers the way they are imperative languages.
- 7 Hence, functional languages are much more like mathematics than imperative languages. Functional programs can be treated as mathematical text, and manipulated using common algebraic laws.

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 :

$$\binom{n}{r} = \frac{n!}{r! * (n - r)!}$$

- 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, $n! = 1 * 2 * \dots * n$.