REBOOT

Simon Thompson
s.j.thompson@kent.ac.uk

CO545 Lecture 10

REBOOT

Essentials

Erlang works like a calculator: `erl`

```
1>2+3*4.
14
```

Running Erlang

Create a `foo.erl` file and load in Erlang

```
2>foo:bar(2,3).
true
```

Data types

Nums, Bools, atoms, tuples, lists, functions.

```
2.13, false,
[{foo,32},{bar,27}]
```

Pattern matching

Used to make choice and select data

```
{circle, {X,Y}, R}
```

Recursion

How to build loops and repetition.

```
fac(0)->1;
fac(N)->n*fac(N-1).
```

Lists

Building, analysing and using recursion

```
1> Xs = [2|[5|[]]].
[2,5]
```

Essentials

Erlang as a calculator

```
Eshell V7.1  (abort with ^G)
1> 2+3/4.
2.75
2> lists:reverse([1,45,1,76]).
[76,1,45,1]
3>
```

You can use the built-in operations, functions and modules …

```
3> c(biscuit).
{ok,biscuit}
4> biscuit:pieces(4).
11
5>
```

… as well as defining functions for yourself.

The Erlang programming model

In a *functional programming* language like Erlang

- computation is *evaluation of expressions* using functions, operators and values;
- programming is the process of *defining functions* for yourself, and devising *data* representations.

Infrastructure

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Save this file on `raptor.kent.ac.uk` …

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Save this file on `raptor.kent.ac.uk` …

… in a subfolder called `cake`.

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Save this file on `raptor.kent.ac.uk` …

… in a subfolder called `cake`.

Log into `raptor.kent.ac.uk` with putty.

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Save this file on `raptor.kent.ac.uk` …

… in a subfolder called `cake`.

Log into `raptor.kent.ac.uk` with putty.

Change into the subfolder by typing `cd cake` at the unix prompt.

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Save this file on `raptor.kent.ac.uk` …

… in a subfolder called `cake`.

Log into `raptor.kent.ac.uk` with putty.

Change into the subfolder by typing `cd cake` at the unix prompt.

Run erlang by typing `erl`.

Running Erlang

Edit a file `biscuit.erl` in a text editor of your choice.

Save this file on `raptor.kent.ac.uk` …

… in a subfolder called `cake`.

Log into `raptor.kent.ac.uk` with putty.

Change into the subfolder by typing `cd cake` at the unix prompt.

Run erlang by typing `erl`.

Compiler your file in erlang by typing `c(biscuit)` to the erlang prompt.

Running Erlang

```
sjt@raptor:~$ cd cake
sjt@raptor:~/cake$ erl
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:80:80] [async-threads:10]
[kernel-poll:false]

Eshell V7.1  (abort with ^G)
1>
```

Log into `raptor.kent.ac.uk` with putty.

Change into the subfolder by typing `cd cake`

Run erlang by typing `erl`.

Compiler your file in erlang by typing `c(bis`

On raptor, outside Erlang

In Erlang (`erl`)

```
Eshell V7.1  (abort with ^G)
1> c(biscuit).
{ok,biscuit}
2> biscuit:pieces(4).
11
3>
```

Erlang modules

```
-module(biscuit).
-export([pieces/1]).


pieces(0) -> 1;

pieces(N) -> N+pieces(N-1).
```

The *name* of the module has to be the same as the file: `biscuit.erl`

To be able to use a function you need to add it to the *export list*.

This number is the *arity*: the number of arguments of the function.

Common errors

```
1> c(biscuit).
{ok,biscuit}
2> pieces(4).
** exception error: undefined shell command pieces/1
3> biscuit:pieces(4).
11
4>
```

Remember to use `module:function` when you call a function.

```
2> c(biscuit).
biscuit.erl:4: Warning: function pieces/1 is unused
{ok,biscuit}
3> biscuit:pieces(4).
** exception error: undefined function biscuit:pieces/1
4>
```

Remember to include the function in the `export` list.

Data types

Numbers

Full precision integers and floats

```
123456789801912,
23.4, 3#121, …
```

Atoms

Atoms are just symbolic data:

```
atom, circle, ok,
'Fish finger', …
```

Booleans

Booleans are two particular atoms

```
true, false
```

Tuples

A collection of data: view as a whole.

```
{234,"23",{true,7}}
```

Lists

A collection of data: can “iterate” over

```
[2,1], [4,7,5],
"foo", [2|[1|[]]]
```

Functions

‘Anonymous’ funs don’t need naming.

```
fun (X)-> X*2 end,
fun foo:bar/2
```

Pattern matching

In traditional languages function / method definitions looks like this

```
method_name(Variable, Variable, …) -> …
```

In Erlang we can put *patterns* instead of variables:

```
xOr(true,X) ->
 not X;

xOr(false,X) ->
 X.
```

```
dist({X,Y}, {X1,Y1})->
 math:sqrt(
  sq(X-X1)+sq(Y-Y1)).

sq(Z) -> Z*Z.
```

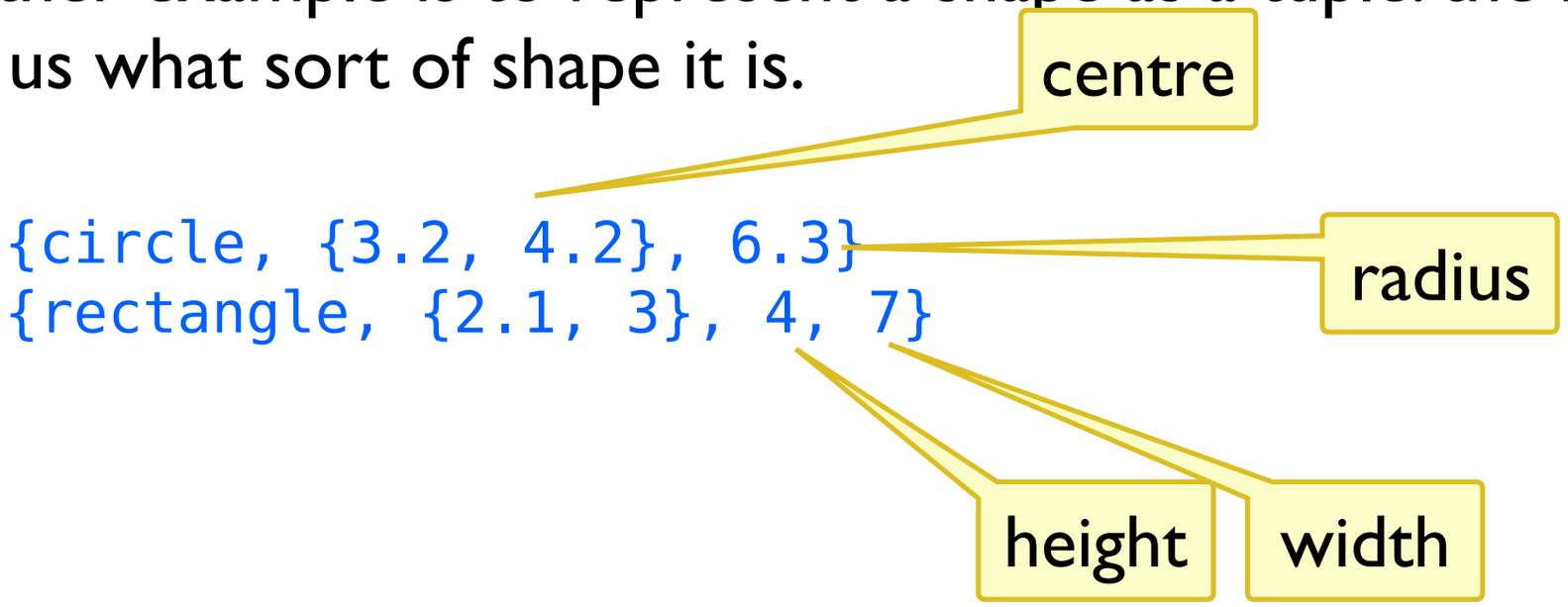
```
empty([]) -> true;

empty(_L) -> false.
```

Representing data

We often use tuples to represent fixed size composites of data, e.g. a pair `{X,Y}` to represent a point in 2D space.

Another example is to represent a shape as a tuple: the initial atom tells us what sort of shape it is.



Pattern matching

The function `inside` has two arguments: a shape and a point.

Pattern matching implements *choice*: circle … ?

… or rectangle?

```
inside({circle,{X,Y},R} , {PX,PY}) ->
          dist({X,Y},{PX,PY}) < R;

inside({rectangle,{X,Y},H,W} , {PX,PY}) ->
          X-W/2 < PX and PX < X+W/2 and
          Y-H/2 < PY and PY < Y+H/2.
```

Pattern matching also lets us *select* parts of a value: *pull out* `X`, `Y`, `H`, `W`, `PX` and `PY` to use in the definition …

Assignment is *single assignment*

Assignment is a special case of pattern matching

```
{A,B} = {2,3}
{C,B} = {45,3}
{D,B} = {54,2}
```

Bind `A` to `2` and `B` to `3`.

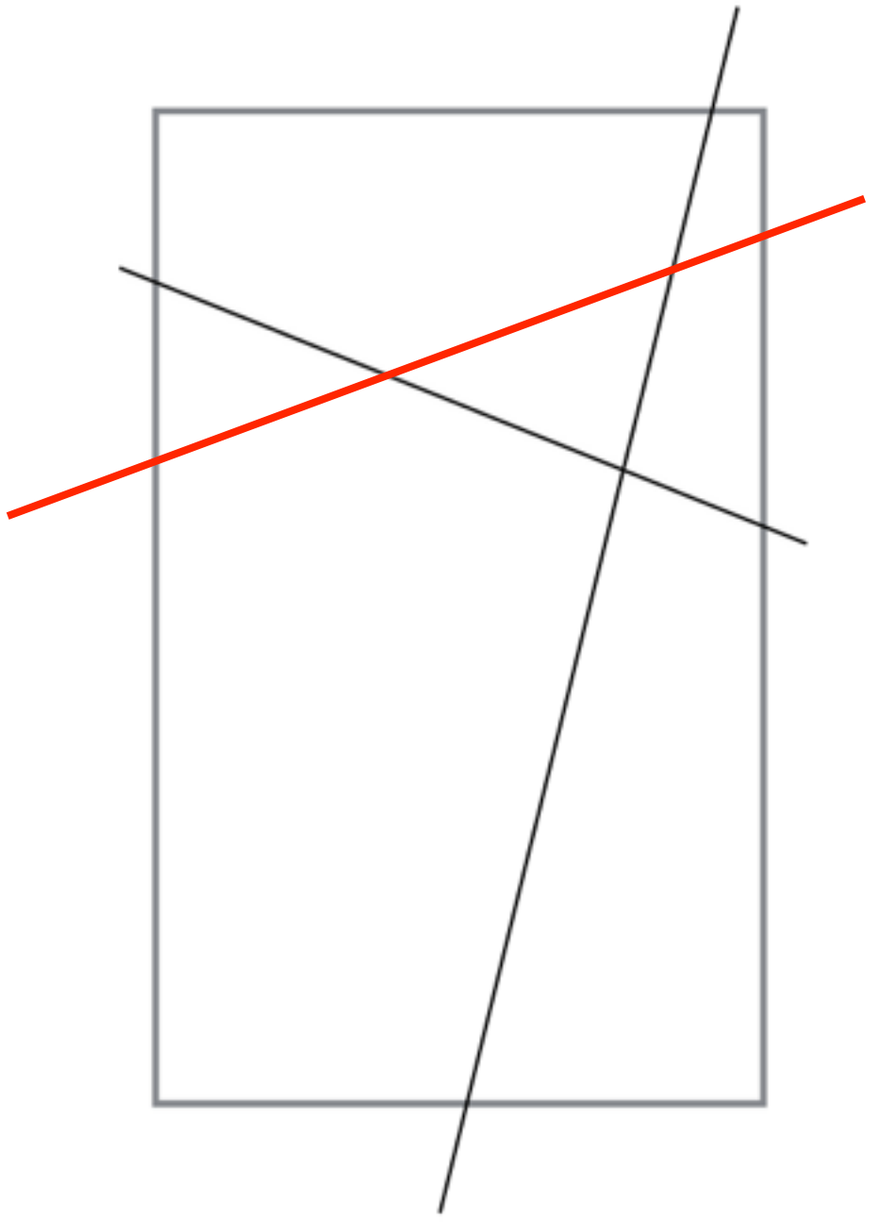
Bind `C` to `45` and *check that* `B` is `3` … OK

Bind `D` to `54` and *check that* `B` is `3` … Fails!

Pieces of paper

How many pieces with N cuts?

```
pieces(0) -> 1;
```

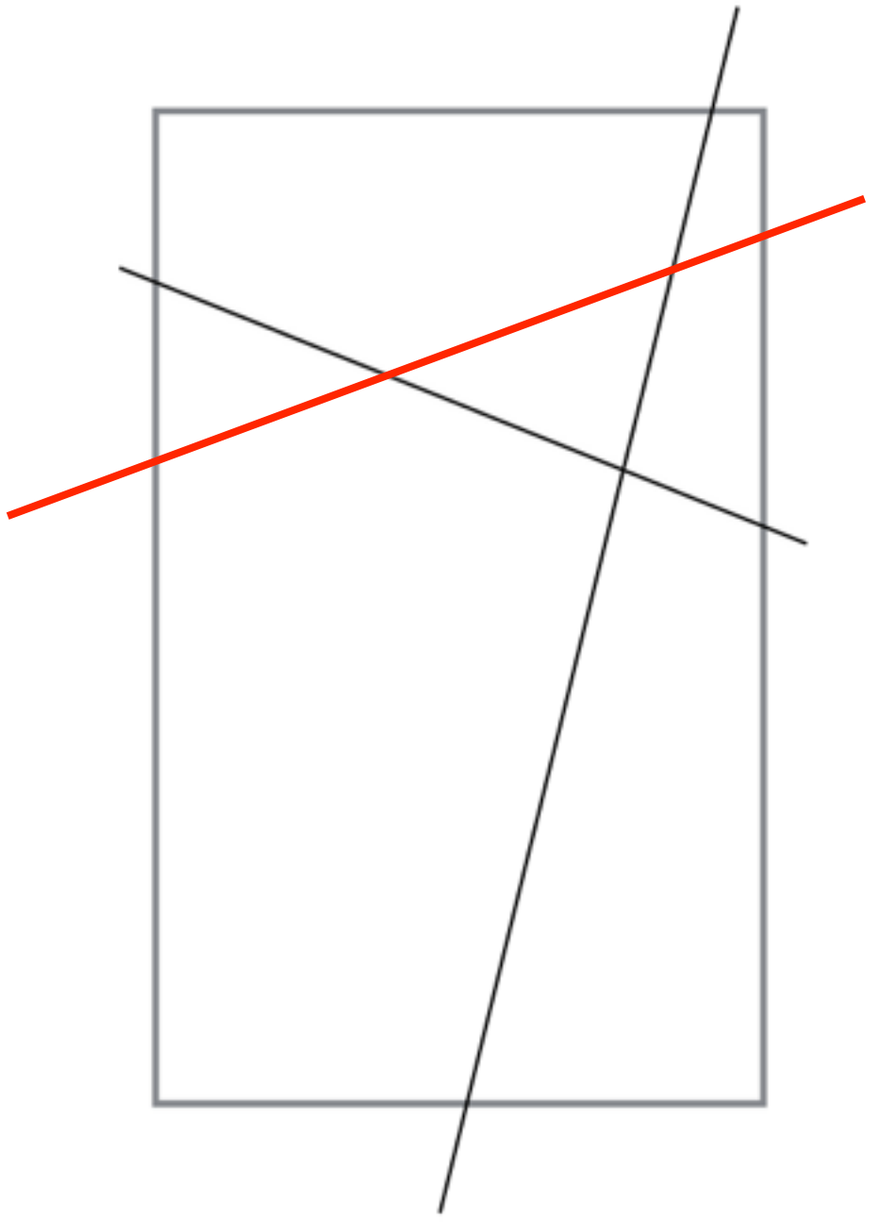


Pieces of paper

How many pieces with N cuts?

```
pieces(0) -> 1;

pieces(N) -> pieces(N-1) + N.
```



Recursion over numbers

The value for `0` outright, and value for `N` using the value for `N-1`.

```
foo(0) -> …;
foo(N) -> … foo(N-1) ….
```

Working it out

The value for `0` outright, and value for `N` using the value for `N-1`.

```
mystery(0) -> 1;
mystery(N) -> N - mystery(N-1).
```

Lists

Erlang list syntax can be confusing …

… but let's try to cut through that.

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]
```

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]

2>X.

3
```

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]

2>X.

3

3>Xs.

[5,2]
```

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

"Under the hood" all lists are built from `[]` using `[...|...]`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]

2>X.

3

3>Xs.

[5,2]
```

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

"Under the hood" all lists are built from `[]` using `[...|...]`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]

2>X.

3

3>Xs.

[5,2]
```

```
1>Xs = [2|[]].

[2]
```

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

"Under the hood" all lists are built from `[]` using `[...|...]`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]

2>X.

3

3>Xs.

[5,2]
```

```
1>Xs = [2|[]].

[2]

2>Ys = [3|Xs].

[3,2]
```

[X|Xs]

Lists are either empty `[]` or non-empty: every non-empty list has a *head* `X` and a *tail* `Xs`.

"Under the hood" all lists are built from `[]` using `[...|...]`.

```
1>[X|Xs] = [3,5,2].

[3,5,2]

2>X.

3

3>Xs.

[5,2]
```

```
1>Xs = [2|[]].

[2]

2>Ys = [3|Xs].

[3,2]

3>[3|[2|[]]]

[3,2]
```

Defining functions over lists

The value for `[]` outright, and value for `[X|Xs]` using the value for `Xs`.

```
product([])     -> 1;
product([X|Xs]) -> X * product(Xs).
```

Defining functions over lists

The value for `[]` outright, and value for `[X|Xs]` using the value for `Xs`.

```
foo([])     -> … ;
foo([X|Xs]) -> … product(Xs) … .
```

Working it out

The value for `[]` outright, and value for `[X|Xs]` using the value for `Xs`.

```
mystery([]) ->
  [];
mystery([X|Xs]) when X>0 ->
  [ X | mystery(Xs) ];
mystery([X|Xs]) when X=<0 ->
  [].
```

REBOOT

Essentials

Erlang works like a calculator: `erl`

```
1>2+3*4.
14
```

Running Erlang

Create a `foo.erl` file and load in Erlang

```
2>foo:bar(2,3).
true
```

Data types

Nums, Bools, atoms, tuples, lists, functions.

```
2.13, false,
[{foo,32},{bar,27}]
```

Pattern matching

Used to make choice and select data

```
{circle, {X,Y}, R}
```

Recursion

How to build loops and repetition.

```
fac(0)->1;
fac(N)->n*fac(N-1).
```

Lists

Building, analysing and using recursion

```
1> Xs = [2|[5|[]]].
[2,5]
```