Where were we?

- Fundamental idea: compute new values rather than assigning repeatedly to variables
- Write-once variables
- Lists
- Pattern matching

Today

- Goal: ability to read Erlang code and know what it means – or how to find out
- Modules and compilation
- Function definitions; the idea of *arity*
- Higher-order functions
- List comprehensions
- Pattern matching with *guards*
- (read about records, section 3.9)
- Exceptions
- Next time: concurrency

Modules and Compilation

- A module lives in a file named modulename.erl

```
geometry.erl
-module(geometry).                  arity
-export([area/1]). % only exported functions can
  be referenced from another module
                    head                    body
area({rectangle, Width, Height}) -> Width *
  Height;
area({circle, R}) -> 3.14159 * R * R.
```

- Compile a module before use

```
c(geometry).
```

Using functions from modules

`modulename:functionname(…)` % *or*

`-import(modulename, [functionname/arity, …])`

`functionname(…)`

- For python programmers: don't have to import the module itself

from module name import names ← python

import modulename ← python

Arity

- *Arity* refers to the number of arguments of a function (in other languages arity may refer to the number *and types* of the function arguments).
- Two functions in the same module with the same name but different arity are *different functions.*

```
-export([sum/1]).
sum([], S) -> S;
sum([H|T], S) -> sum(T, S+H). % tail
   recursion
sum(L) -> sum(L, 0).
```

Anonymous functions

- Functions as seen so far can only be defined in modules
- Anonymous functions can be defined in the shell or in modules

```
fun(X) -> 2*X end.
```

first-class values.

- Assign it or pass it as an argument

```
Double = fun(X) -> 2*X end.
DoubleList = map(fun(X) -> 2*X end,
  [1,2,3]). % or
DoubleList = map(Double, [1,2,3]).
```

product([1, 2, 3])

List processing (review 355)

- Processing one element at a time

squares([]) -> [];

squares([H|T]) -> [H*H|foo(T)]. % use map

- Combining all the elements

product([]) -> 1;

product([H|T]) -> H * ~~sum~~ product(T). % use fold

- Combining using an accumulator

product([], A) -> A;

product([H|T]),A -> product(T, H*A).

product(L) -> product(L, 1).

Higher-order functions

- Functions taking functions as arguments or returning functions as results

```
% erl -man lists
```

- map/2

```
squares(L) -> map(fun (X) -> X*X end, L).
```

- foldr/3, foldl/3

```
product(L) -> foldl(fun (Elem, Acc) ->
 Elem*Acc end, 1, L).
```

Functions as results

```
mult(N)-> (fun(M)-> N*M end).
```

could only go in module

Test your understanding: what's different between the above and

```
Mult = fun(N)  -> (fun(M)  -> N*M end) end.
```

Mult6 = mult(6).

Mult6a = Mult(6).

FortyTwo= Mult6(7),

FortyTwo = Mult6a(7).

mult(6)(7)

List Comprehensions

- Even more convenient way to write map-ish things

```
squares(L) -> [X*X || X <- L]. % read X*X
  for X in L
```

- Similarly, if L is a list of numeric tuples, to compute the list of products

```
products(L) -> [X*Y || {X,Y} <- L].
```

- Can make inclusion dependent on the data values with *filters*

```
sqrts(L) -> [sqrt(X) || X <- L, X>=0].
```

Pythagorean Triples

```
pythag(N) ->
   [ {A,B,C} ||
      A <- lists:seq(1,N),
      B <- lists:seq(1,N),
      C <- lists:seq(1,N),
      A+B+C =< N,
      A*A+B*B =:= C*C
   ].
```

Permutations

```
perms([]) -> [[]];
perms(L) ->
    [[H|T] ||
        H <- L,
        T <- perms(L--[H])
    ].
```

Pattern matching with guards

- List comprehensions combined *generators* and *filters*
- In function definitions can use *guards* to further limit matching

```
max(X,Y) when X>Y -> X;
max(X,Y) -> Y.
```

- Guards may be conjunctive (and) – combine with , or
- disjunctive(or) – combine with ;
- Side-effects in guards are not allowed

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Raising Exceptions

- exit(Why) % current process exits
- throw(Why) %
- erlang:error(Why)
- Have to go to extra effort to handle an exit() or erlang:error(). Otherwise similar.

Catching Exceptions

```
try FuncOrExpressionSequence of
   Pattern1 [when Guard1] -> Expressions1;
   …
catch
   ExType: ExPattern1 [when exGuard1] ->
           ExExpressions1;
   …
after
   AfterExpressions
end
```

Try notes

- You can omit the “of Patterni -> Expressionsi” part entirely
- You can omit the “after After” part entirely
- Questions
  - Do the catch phrases handle exceptions occuring during the Expressionsi?
  - What is `retry` ?