

Introducing the Erlang Language

- **Why Erlang?**
 - **Exemplifies the “you can have code and concurrency that is free from side effects...there is no other way” perspective**
 - **Real language used to build real systems**
 - **A pretty good, practical book exists to use as a text**
 - **Multi-core applications**
 - **Fault-tolerant applications**

Erlang is a Functional Language

- Lots of definitions but some implications
 - Computation is primarily done by evaluating functions that take immutable values as inputs and produce immutable values as outputs
 - Programs are written by defining functions (in terms of other functions)
 - Functions are side-effect free
- Other functional languages: Haskell, Standard ML, LISP (in some variants)
- You can *program functionally* in imperative languages (e.g. C, Java, etc.) but some make it easier than others
- Automatic storage management (GC) is almost always a feature of functional languages (but not vice-versa)

Erlang values

- Numbers
 - Integers with arbitrary precision (dd*)
 - Floating point (d*d.dd*)
- Tuples
 - {value [, value]*} ; 1-tuple allowed and different from the contained value: 1 != {1}
 - Tuple models data structures of known fixed size
- Atoms
 - Alphanumeric sequences starting with lower-case letter (may contain but not start with _ and @)
 - Any sequence enclosed in single quotes 'Another Atom'
 - The value of an atom is itself
- Lists
 - Sequence of values of unknown size
 - Compare to python tuples and lists – Erlang lists are not mutable!

Object
↓

An Erlang idiom using tuples 02

```
struct person { char *first; char *last }; -- C
```

{person, {first, "carl"}, {last, "hauser"}} – this is just a value; note the use of atoms as tags in the tuples

Erlang is a dynamically, strongly typed language.

Using tags helps people keep things straight about programs. Compare {"carl", "hauser"}

strongly typed – can't use a value incorrectly by passing to a operation that interprets it as something else.

More about lists

- List values can be written directly – [value, value, ..., value] or created one element at a time using the *list constructor* [value, ..., value | list]
- The first element of a list is called the *head*
- Everything else is the *tail*

$[1, 2, 3, 4]$
 $[1 | X]$ $[1, 2, 3 | X]$

erl

Strings

- Erlang strings are really just lists of integers
- `5> "abc".`
`"abc"`
- `6> [1|"abc"].`
`[1,97,98,99]`
- Shell has a special rule for printing lists that contain only printable characters
- The integer for a character can be written `$c`
- `7> $a.`
`97`
- This feels like a *hack* to me

bind
bound

bounded

X variable
X atom

Erlang variables

- Start with upper-case letter (enforced convention for human beings sake)
- Are *write-once*
 - A new variable is called *unbound*
 - After assignment a variable is said to be *bound* and cannot be further changed (limits the possibility for side effects!)
 - “assignment” is technically the wrong term – more properly called *binding*

Erlang approach to binding: assertion of equality

Idea: a form such as $X = \{1, 3\}$ is an assertion of equality between X and $\{1, 3\}$. If something equals something else then they can't later be not equal. Hence the bind-once rule.

9> $X = \{1,3\}$.

$\{1,3\}$

10> x .

x

11> X .

$\{1,3\}$

12> $X=X$.

$\{1,3\}$

13> $X=\{1,3\}$.

$\{1,3\}$

14> $X=\{1,4\}$.

** exception error: no match of right hand side value $\{1,4\}$

Binding using pattern matching

- You may recall this idea from 355 if you talked about ML there
- What if the LHS is not a single variable but something that looks like a value but with variables embedded at some places

```
17> {Y,Z}=X.
{1,3}
18> Y.
1
19> Z.
3
```

patterns

- You may have also noticed this in Python

```
>>> a = [1,2,3]
>>> [d,e,f] = a
>>> d
1
>>> e
2
>>> f
3
```

More pattern matching - lists

head

22> L1 = [0,1,2,3,4].

[0,1,2,3,4]

23> [H|T] = L1.

[0,1,2,3,4]

24> H.

0

25> T.

[1,2,3,4]

list constructor
cons

$[H, T] = [1, 2]$

tail

$[H | T] = [1]$

Assignment

- **Before next class**
 - **Skim Chapters 2-5; read chapter 7**
 - **Install Erlang and work through (at the very least) the examples in Chapter 6**
 - **Don't worry about the version: whatever precompiled version is available for your system should be fine**