Functional Programming

Why you should care
Why are we here?

(Alternative) programming languages [Functional (Erlang, Haskell, OCaml,...)]

- Never heard of, Not interested: 1.1%
- Never heard of: 32.2%
- Colour me interested!: 21.1%
- Not interested: 45.6%

From: ADASS 2019 LOC survey during ADASS 2018 and AAS jan 2019
i.e.
There’s unrealized potential

Easier to read

More concise

Requires fundamental changes
FP eliminates huge classes of bugs
- irrespective of language

No (global) variables/immutability
Inconsistent state of variables is impossible

Lines of code
Functional code is usually (a lot!) shorter

Logic errors
Because you have to really think about the solution
You don’t even have to change your favourite language

Java, Python, C++, all start to incorporate functional features

But if you can, Haskell’s pretty rad
A simple example

“Compute the sum of squares”
The hard part (thinking)

“For each element in the list(*), square the value, then sum up all those results”

(*) or sequence, iterable, collection, you get the picture ...
Template C++ implementation

```cpp
template <typename T>
T sumsq(vector<T> const& lst) {
    T total = 0; // 0.0? 0.f? ...
    typename vector<T>::const_iterator ptr;

    for (ptr = lst.begin(); ptr != lst.end(); ptr++)
        total += (*ptr * *ptr);

    return total;
}
```

Score: 7 LOC, 2 variables, 1 argument
Optimal Haskell ...

\[
\text{sumsq} = \text{foldl} (+) 0 \ . \ \text{map} \ (\backslash x \rightarrow x^*x)
\]

**Score:** 1 LOC, 0 variables, 1 argument (hidden)
Optimal Haskell ...

\[
\text{sumsq} = \text{foldl} \ (+) \ 0 \ . \ \text{map} \ (\lambda x \rightarrow x \times x)
\]

Score: 1 LOC, 0 variables, 1 argument
Optimal Haskell ...

```haskell
sumsq = foldl (+) 0 . map (\x -> x*x)
```

**Score:** 1 LOC, 0 variables, 1 argument
Optimal Haskell ...

sumsq = sum . map (^2)

Score: 1 LOC, 0 variables, 1 argument (hidden)
Optimal Haskell ... (function composition)

\[ f \circ g \]

produces new function \( f' \) such that:

\[ f'(x) = f( g(x) ) \]

thus:

\[ \text{sumsq} = \text{composition of:} \]

- transform inputs to their squares
- sum the results
template <typename Container>
auto sumsq(Container&& lst) -> typename remove_reference<typename decay<decltype(*begin(lst))>::type>::type {
    using underlying_type = typename remove_reference<typename decay<decltype(*begin(lst))>::type>::type;

    vector<underlying_type> tmp;

    // transform to sequence of squares (use C++11 lambda)
    transform(begin(lst), end(lst), back_inserter(tmp), [](underlying_type x) { return x*x; });

    return accumulate(begin(tmp), end(tmp), 0);
}
**Functional tools in C++11 ...**

```cpp
template <typename Container>
auto sumsq(Container&& lst) -> typename remove_reference<
typename decay<decltype(*begin(lst))>::type>::type {
    using underlying_type = typename remove_reference<
typename decay<decltype(*begin(lst))>::type>::type;

    vector<underlying_type> tmp;

    // transform to sequence of squares (use C++11 lambda)
    transform(begin(lst), end(lst), back_inserter(tmp), [] (underlying_type x) { return x*x; });

    return accumulate(begin(tmp), end(tmp), 0);
}
```

**Score:** 6 LOC, 1 variable, 2 arguments
Naïve Python implementation

```python
def sumsq(lst):
    total = 0
    for item in lst:
        total += (item * item)
    return total
```

**Score:** 5 LOC, 2 variables, 1 argument
**Naïve Python implementation**

```python
# this should be in the standard library: function composition
compose = lambda *fns: lambda x: reduce(lambda acc, f: f(acc), reversed(fns), x)

sumsq = compose(sum, partial(map, lambda x: x*x))

or

sumsq = sum([x*x for x in others])
```

**Score:** 1 LOC, 0 variables, 1 argument
From solving problems to problem declaration
Free Upgrades! *(T&C apply)*

Leave the optimizations to the optimizers

2 hours vs 1,3 seconds [Kosytorz 2019]
Even BSc Liberal Arts students can do it

AUC students building interpreters in Haskell
In Short: No More Excuses :)