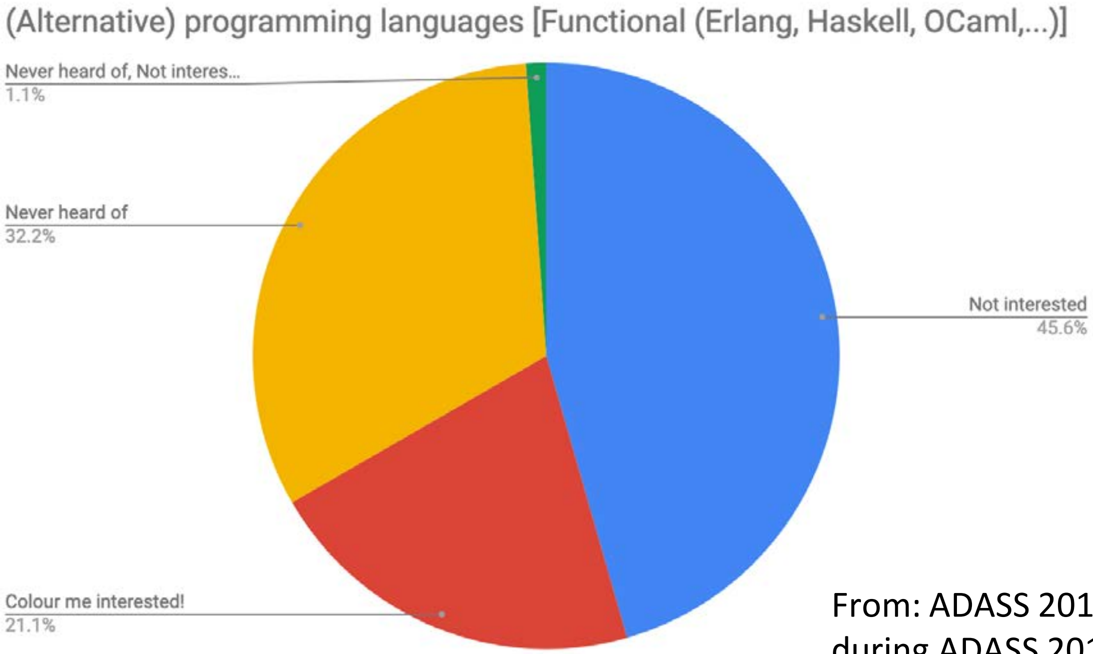
Functional Programming

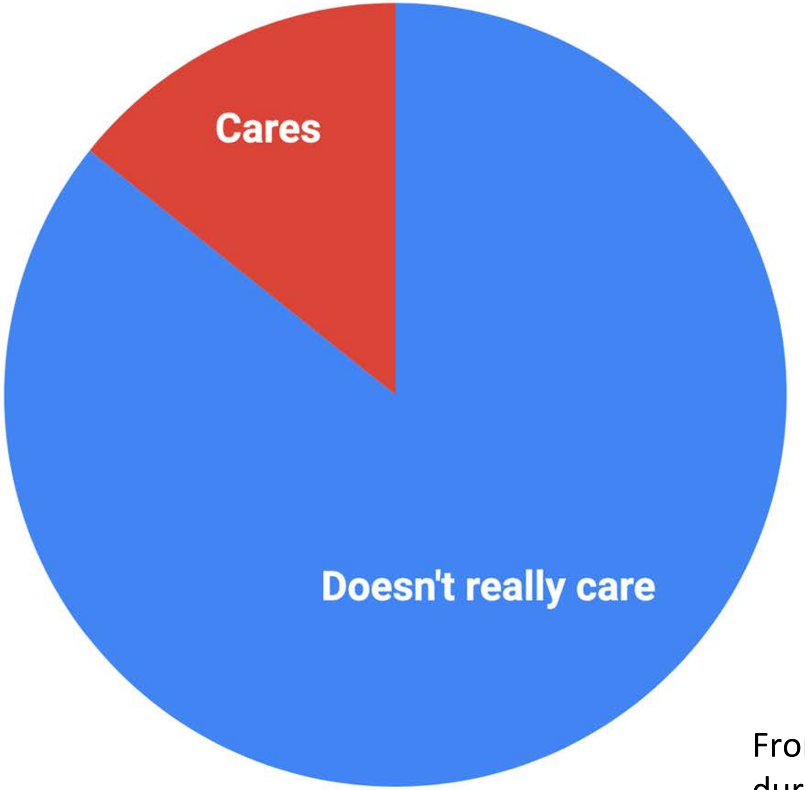
Why you should care

Why are we here?



From: ADASS 2019 LOC survey
during ADASS 2018 and AAS jan 2019

i.e.



From: ADASS 2019 LOC survey
during ADASS 2018 and AAS jan 2019

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 ...

Straightforward C++ implementation

```
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 …

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

Score: 1 LOC, 0 variables, 1 argument (hidden)

Optimal Haskell …

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

Score: 1 LOC, 0 variables, 1 argument

Optimal 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 . g

produces new function f’ such that:

f’(x) = f(g(x))

thus:

sumsq = composition of:

- transform inputs to their squares
- sum the results

Functional tools in C++11 ...

```
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

Functional tools in C++11 …

```
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

```
# 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])
```

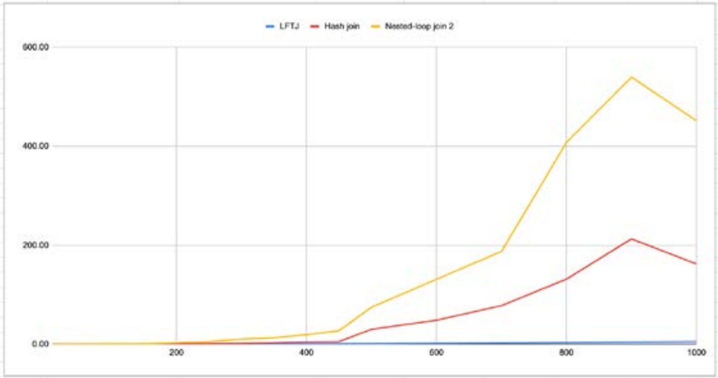
Score: 1 LOC, 0 variables, 1 argument

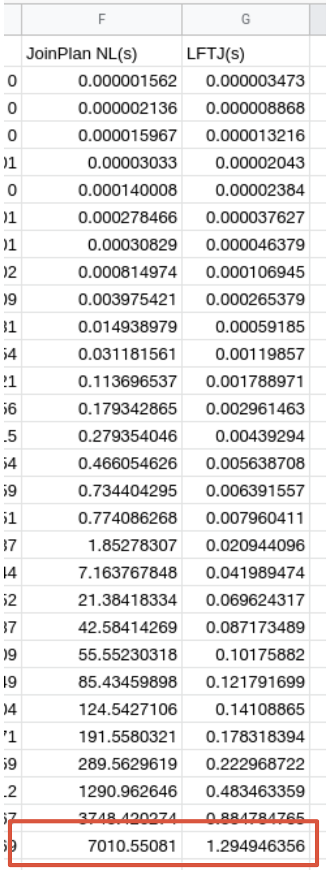
From solving problems to problem declaration

Free Upgrades! (T&C apply)

Leave the optimizations to the optimizers

2 hours vs 1,3 seconds[Kosytorz 2019]





	F	G
	JoinPlan NL(s)	LFTJ(s)
0	0.000001562	0.000003473
0	0.000002136	0.000008868
0	0.000015967	0.000013216
1	0.00003033	0.00002043
0	0.000140008	0.00002384
1	0.000278466	0.000037627
1	0.00030829	0.000046379
2	0.000814974	0.000106945
9	0.003975421	0.000265379
1	0.014938979	0.00059185
4	0.031181561	0.00119857
1	0.113696537	0.001788971
6	0.179342865	0.002961463
5	0.279354046	0.00439294
4	0.466054626	0.005638708
9	0.734404295	0.006391557
1	0.774086268	0.007960411
7	1.85278307	0.020944096
4	7.163767848	0.041989474
2	21.38418334	0.069624317
7	42.58414269	0.087173489
9	55.55230318	0.10175882
9	85.43459898	0.121791699
4	124.5427106	0.14108865
1	191.5580321	0.178318394
9	289.5629619	0.222968722
2	1290.962646	0.483463359
7	3748.420274	0.884784765
9	7010.55081	1.294946356

Even BSc Liberal Arts students can do it

AUC students building interpreters in Haskell

In Short: No More Excuses :)