

# **15-150 Fall 2021**

**Stephen Brookes**

## **LECTURE I**

Introduction to Functional Programming

# Plan

This is an IN-PERSON class

- Please show up, ***on time***
  - ***Masks are required***
  - *Obey the university's rules... your health is important!*
- Homework and exams
  - Do your own work!

# Logistics

- Get to know course staff
  - email, labs, office hours...
- Email me about any concerns
  - Class size is large, so please be patient
  - Can also cc my assistant,  
Christina Contreras (cc8k@andrew)

More  
Information



- Course website
- Canvas page
- Piazza
- Gradescope

# Diversity

This class aims to give full and fair consideration to students from diverse backgrounds.

Diversity will be appreciated as a resource, a strength and a benefit.

Course staff aim to be respectful, and responsive to needs.

If any class meetings or deadlines conflict with religious events, let me know in advance so we can make arrangements.

Your suggestions are encouraged and appreciated.

Please let me know ways to improve the course.

# Functional programming

LISP • APL • FP • Scheme • KRC • Hope  
Miranda™ • Erlang • Curry • Gofer • Mercury  
Charity • Cayenne • Mondrian • Epigram  
SML • Clean • Caml • Haskell



Everything else is just  
*dys*functional  
programming!

# The SML language

- *functional*

*computation = expression evaluation*

- *typed*

*only well-typed expressions are evaluated*

- *polymorphic*

*well-typed expressions have a most general type*

- *call-by-value*

*function calls evaluate their argument*

# Advantages

- *functional*

easy to design and analyze

- *typed*

common errors caught early

- *polymorphic*

easy to re-use code

- *call-by-value*

predictable control flow



# example

Standard ML of New Jersey [...]

```
fun length [ ] = 0
```

```
| length (x::L) = 1 + length L;
```

- val length = fn - : 'a list -> int

```
length [1, 2, 4, 8];
```

- val it = 4 : int

```
length [true, false];
```

- val it = 2 : int

```
length 42;
```

- type error!

# Features

- **referential transparency**
  - *equivalent code is interchangeable*
- **mathematical foundations**
  - use math to define *equivalence*
  - use logic to prove *correctness, termination, ...*
- **functions are values**
  - can be used as data in lists, tuples, ...
  - can be an argument or result of other functions
  - also a good slogan for T-shirts

# Referential transparency

- The *type* of an expression depends only on the *types* of its sub-expressions
- The *value* of an expression depends only on the *values* of its sub-expressions

safe substitution,  
compositional reasoning

# Equivalence

- Expressions of type **int** are *equivalent* if they evaluate to the same integer
- Functions of type **int -> int** are *equivalent* if they map *equivalent arguments* to *equivalent results*
- Expressions of type **int list** are *equivalent* if they evaluate to the same list of integers

*Equivalence is a form of semantic equality*

# Equivalence

- $21 + 21$  is equivalent to  $42$
- $[2,4,6]$  is equivalent to  $[1+1, 2+2, 3+3]$
- $\text{fn } x \Rightarrow x+x$  is equivalent to  $\text{fn } y \Rightarrow 2*y$

$$21 + 21 = 42$$

$$\text{fn } x \Rightarrow x+x = \text{fn } y \Rightarrow 2*y$$

$$(\text{fn } x \Rightarrow x+x) (21 + 21) = (\text{fn } y \Rightarrow 2*y) 42 = 84$$

We use  $=$  for equivalence

Don't confuse with  $=$  in ML

# equality in ML

- ML has a built-in `=` operator
- Can use with expressions of simple types like `int`, `bool`, `int list`, ... (called *equality types*)
- Will check if expressions evaluate to same value

`(2 + 2) = 4` evaluates to `true`  
=

# Equivalence

- For every type  $t$  there is a notion of *equivalence* for expressions of that type
  - We usually just use  $=$
  - When necessary we use  $=_t$

Our examples so far illustrate:

$=_{\text{int}}$

$=_{\text{int list}}$

$=_{\text{int} \rightarrow \text{int}}$

# Compositionality

- Replacing a sub-expression of a program with an *equivalent* expression always gives an *equivalent* program

The key to  
*compositional reasoning*  
about programs



# Parallelism

- Expression evaluation has ***no side-effects***
  - can evaluate *independent* code *in parallel*
  - evaluation order has *no effect* on *value*
- Parallel evaluation may be *faster* than sequential

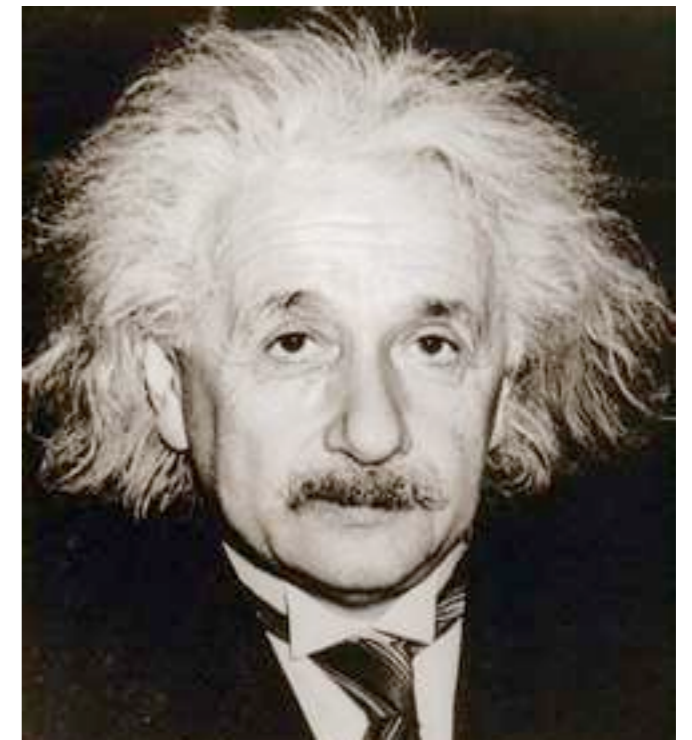
Learn to *exploit* parallelism!

# Principles

- **Expressions must be well-typed.**  
*Well-typed expressions don't go wrong.*
- **Every function needs a specification.**  
*Well-specified programs are easier to understand.*
- **Every specification needs a proof.**  
*Well-proven programs do the right thing.*

# Principles

- **Large programs should be *modular*.**  
*Well-interfaced code is easier to maintain.*
- **Data structures algorithms.**  
*Good choice of representation can lead to better code.*
- **Exploit parallelism.**  
*Parallel code may run faster.*
- **Strive for simplicity.**  
*Programs should be as simple as possible, but no simpler.*



# sum

```
fun sum [ ] = 0  
| sum (x::L) = x + sum(L)
```

A recursive function declaration  
using list patterns and integer arithmetic

- **sum** has type **int list -> int**
- **sum [1,2,3]** evaluates to **6**
- For all  **$n \geq 0$**  and integer values  **$v_1, \dots, v_n$**

$$\text{sum } [v_1, \dots, v_n] = v_1 + \dots + v_n$$

# sum

**fun** sum [ ] = 0

| sum (x::L) = x + sum(L)

sum [1,2,3]

[1,2,3] = 1 :: [2,3]

= 1 + sum [2,3]

= 1 + (2 + sum [3])

= 1 + (2 + (3 + sum [ ]))

= 1 + (2 + (3 + 0))

= 6

*equational  
reasoning*

# count

**fun** count [ ] = 0

| count (r::R) = (sum r) + (count R)

- **count** has type (int list) list -> int

# count

**fun** count [ ] = 0

| count (r::R) = (sum r) + (count R)

- **count** has type (int list) list -> int
- **count** [[1,2,3], [1,2,3]] evaluates to 12

# count

**fun** count [ ] = 0

| count (r::R) = (sum r) + (count R)

- **count** has type (int list) list -> int
- **count** [[1,2,3], [1,2,3]] evaluates to 12
- For all  $n \geq 0$  and integer lists  $L_1, \dots, L_n$   
count [L<sub>1</sub>, ..., L<sub>n</sub>] = sum L<sub>1</sub> + ... + sum L<sub>n</sub>



# count

Since

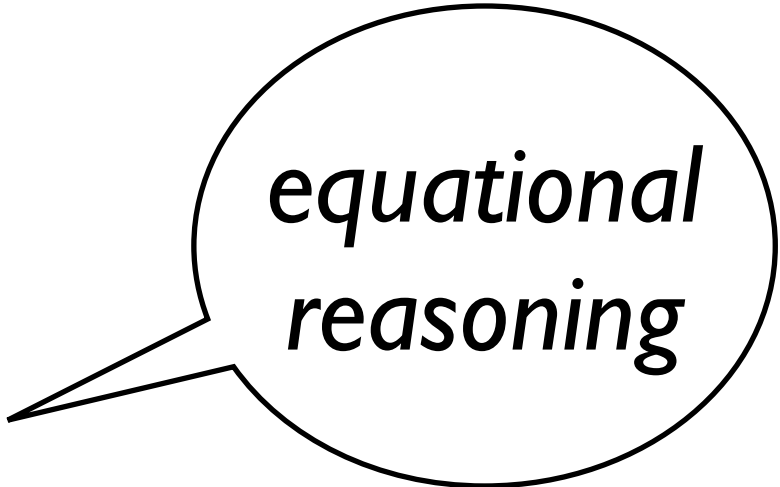
$$\text{sum } [1,2,3] = 6$$

and

$$\begin{aligned} \text{count } [[1,2,3], [1,2,3]] \\ = \text{sum}[1,2,3] + \text{sum } [1,2,3] \end{aligned}$$

it follows that

$$\begin{aligned} \text{count } [[1,2,3], [1,2,3]] \\ = 6 + 6 \\ = 12 \end{aligned}$$



*equational reasoning*

# tail recursion

```
fun sum [] = 0
```

```
| sum (x::L) = x + sum(L)
```

- The definition of `sum` is not tail-recursive
- Can define a tail recursive *helper* function `sum'` that uses an integer *accumulator*

```
sum : int list -> int
```

```
sum' : int list * int -> int
```

**Q:** This is a general technique. But why bother?

**A:** Sometimes tail recursion is more efficient.

# sum'

**fun** sum' ([ ], a) = a

| sum' (x::L, a) = sum' (L, x+a)

- sum' has type `int list * int -> int`
- `sum' ([1,2,3], 4)` evaluates to `10`
- For all integer lists `L` and integers `a`,  
$$\text{sum}'(L, a) = \text{sum}(L) + a$$

# Sum

**fun** sum' ([ ], a) = a

| sum' (x::L, a) = sum' (L, x+a)

**fun** Sum L = sum' (L, 0)

- Sum has type `int list -> int`
- Sum and `sum` are *equivalent*

For all integer lists L,

`Sum L = sum L`

# Hence...

**fun** count [ ] = 0

| count (r::R) = (sum r) + (count R)

**fun** Count [ ] = 0

| Count (r::R) = (Sum r) + (Count R)

- **Count** and **count** are *equivalent* because **Sum** and **sum** are equivalent.

# Evaluation

**fun** sum [ ] = 0

| sum (x::L) = x + sum(L)

sum (1::[2,3])  $\implies^*$  1 + sum [2,3]

$\implies^*$  1 + (2 + sum [3])

$\implies^*$  1 + (2 + (3 + sum [ ]))

$\implies^*$  1 + (2 + (3 + 0))

$\implies^*$  1 + (2 + 3)

$\implies^*$  1 + 5

$\implies^*$  6

$\implies^*$   
means  
“evaluates to,  
in finitely many steps”

pattern of recursive calls,  
order of arithmetic operations

# Evaluation

count [[1,2,3], [1,2,3]]

$\Rightarrow^*$  sum [1,2,3] + count [[1,2,3]]

$\Rightarrow^*$  6 + count [[1,2,3]]

$\Rightarrow^*$  6 + (sum [1,2,3] + count [ ])

$\Rightarrow^*$  6 + (6 + count [ ])

$\Rightarrow^*$  6 + (6 + 0)

$\Rightarrow^*$  6 + 6

$\Rightarrow^*$  12

# Analysis

(details later!)

code fragment	evaluation time proportional to
<code>sum(L), Sum(L)</code>	length of L
<code>count(R), Count(R)</code>	sum of lengths of lists in R

(tail recursion **doesn't** help here!)

These functions do *sequential evaluation*...



# parallelism

+ is *associative* and *commutative*

The combination *order* doesn't affect result,  
so it's safe to evaluate in parallel

Suppose we have a function map such that

$$\text{map } f [x_1, \dots, x_n] \Longrightarrow^* [f(x_1), \dots, f(x_n)]$$

and we can evaluate the  $f(x_i)$  in parallel...

# parallel counting

```
fun parcount R = sum (map sum R)
```

parcount [[1,2,3], [4,5], [6,7,8]]

⇒\* sum (map sum [[1,2,3], [4,5], [6,7,8]])

⇒\* sum [sum [1,2,3], sum [4,5], sum [6,7,8]]



parallel evaluation of sum[1,2,3], sum[4,5] and sum[6,7,8]

⇒\* sum [6, 9, 21]

⇒\* 36

# Analysis

- Let  $R$  be a list of  $k$  rows,  
and each row be a list of  $m$  integers
- *If we have enough parallel processors,*  
 $\text{parcount } R$  takes time proportional to  $k + m$

computes each row sum, in parallel  
then  
adds the row sums

Contrast:  $\text{count } R$  takes time proportional to  $k \cdot m$

With  $m=20$  and  $k=12$ ,

$k + m$  is 32, almost an 8-fold speedup over  $k \cdot m = 240$ .

# work and span

We will introduce techniques for analysing

- *work* (sequential runtime)
- *span* (optimal parallel runtime)

(that's how we did those runtime calculations)

# Themes

- **functional programming**
- correctness, termination, and performance
- types, specifications and proofs
- evaluation, equivalence and referential transparency
- compositional reasoning
- exploiting parallelism

# Objectives

- Write well-designed ***functional programs***
- Write ***specifications***, and prove correctness
- Techniques for analyzing runtime (***sequential*** and ***parallel***)
- Choose data structures wisely and exploit ***parallelism*** to achieve ***efficiency***
- Design code using ***modules*** and ***abstract types***, with clear interfaces

# Summary

- Don't worry if you don't know SML syntax
- Don't panic about so-far-undefined terminology
  - We will cover the details in lectures
- This introduction should help you appreciate the main ideas and see where we're going...