Welcome & Intro

Course Stuff, Functional Programming, and the SML Evaluation System

15-150 M21

Lecture 0521
21 May 2021

0 Course Stuff

The 150 Staff

Jacob Neumann

Avery Cowan, Eshita Kar, James Gallicchio, Joseph Rotella, Juhi Agrawal, Justin Zhang, Leah Restad, Lili Chang, Ryan Stoltzfus, Sam Banks, Sanjana Meduri, Siddharth Paratkar, Sonya Simkin, Thea Brick, & Will Fowlkes

Where to go for info

- Canvas
- Website (`cs.cmu.edu/~15150/`)
- Piazza
- Homework handouts

To-Do

- Make sure you're on the 150 M21 Piazza
- Make sure you can access the Canvas
- Fill out the lab availability form (if you haven't already)

  forms.gle/x4z7wWiqgDfwFzPq9
- Read the website
- Setup Lab (to be posted today, needs to be done by Tuesday)

Lectures

Three times a week (usually)

- Monday: Introduce & motivate topic
- Wednesday: Advanced Stuff
- Friday: Case study, application, or further topic

This fits in with homework cycle:

- Sunday: Homework released
- Monday: Lecture (should give you enough to start some parts of homework)
- Tuesday: Lab (should much better equip you to do homework)
- Wednesday: Lecture (should give you everything you need to do homework)
- Friday: Lecture (shouldn't ne necessary for homework, but perhaps helpful)
- Saturday: Homework due
- Sunday: Homework due (use late day)

How to get the most out of lecture

- Answer the in-lecture questions! (Async: pause and come up with answer)
- Fill out the worksheet!
- Quiz yourself afterwards!
- *Do* try this at home!

Lecture-Related Stuff

- Worksheets and solutions
- Lecture code (& numbering)
- Aux-Library (github.com/smlhelp/aux-library/)
- smlhelp (smlhelp.github.io)
- Additional "Check Your Understanding"s
- 5-minute breaks
- Key points/Key skills
- Slogans

Today's slogan:

Computation is evaluation

1 Functional Computation

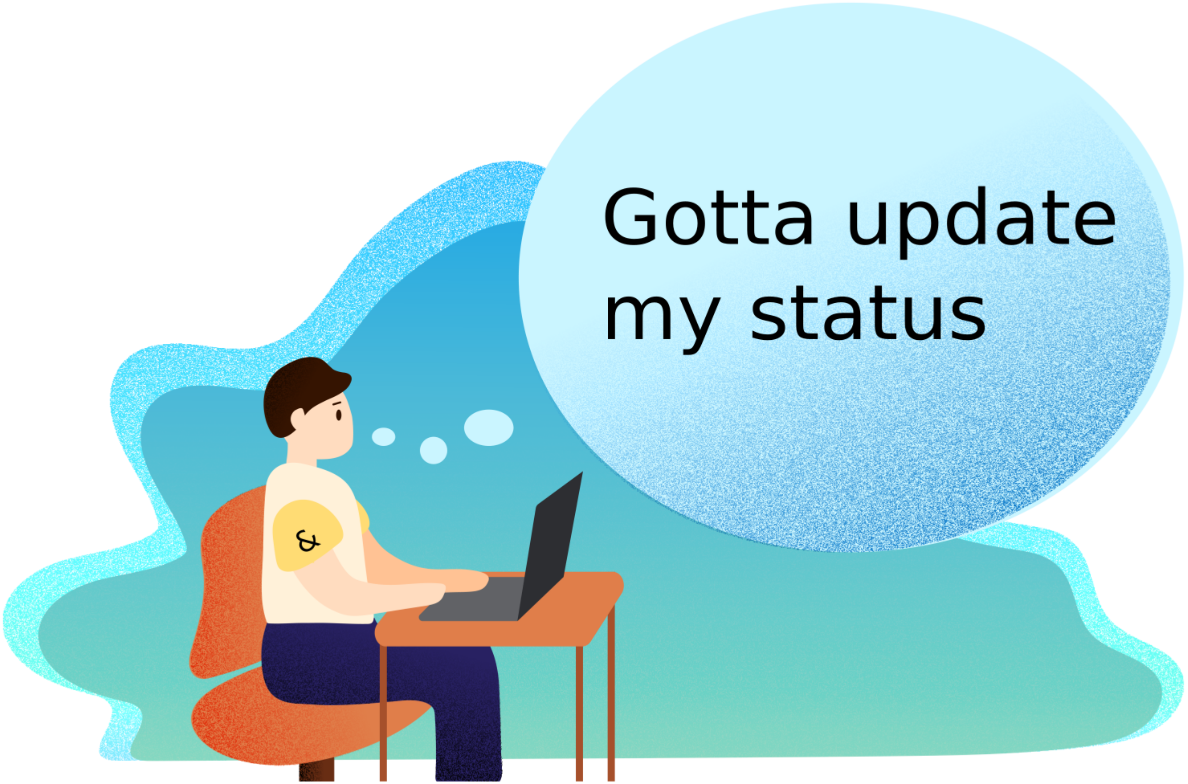
What do we use computers for?

Artwork credit: Mia Tang

We're interested in two particular answers:

To cause an effect

To calculate a value



Effect: A change to the state of the computer or the world

Value: A piece of data which is "fully calculated" or "fully simplified" – the kind of thing that can serve as an answer to a computational question (need to specify what this means)

Key Observation

Causing effects and calculating values are distinct kinds of computational tasks

Check Your Understanding

Suppose you got bored during quarantine and composed a lengthy work of fan fiction, typed into your computer (and saved your work, of course!). Is this causing an effect or calculating a value, or both?

What about if you subsequently checked the word count on your soon-to-be-bestseller?

Imperative Programming: computation is accumulating effects

An **imperative** program is a structured sequence of commands, specifying how to mutate the computer's state (that is, which *effects* to have, and in what order).

Imperative programs calculate values by *accumulating effects*: we initialize the computer's state, perform a bunch of effects, and then read off the result.

Example

When using an imperative program to calculate values, you have to be careful because it might matter *when* you ask.

The smart response to this:
A good programmer will avoid adverse side-effects

The bold response to this:
A good *programming paradigm* will *prohibit* adverse side-effects

Functional Programming: computation is the calculation of values (and maybe some effects happen along the way)

Pure Functional Programming: computation is the calculation of values (and *no* effects happen along the way)

A *functional program* is a description of how to calculate values, i.e. how to turn *unevaluated expressions* into *values*.

```
exp(17)  ⟹ 131072
```

Purely functional programs are inherently non-destructive, and therefore executing the same code will always give the same result

A **functional programming language** is a programming language designed around the functional model of computation.

It's important to remember:

- Functional aspects exist in most languages, and you can (and should!) use functional techniques in non-functional languages
- Functional languages (including SML) often allow for limited kinds of effects, but they have to fit into the overall functional nature of the language

- **May:** Basics of the functional model of computation
- **Early June:** Induction and recursion in functional programming
- **Mid June:** Abstracting common patterns of reasoning
- **Late June:** Designing elegant control flow using functional methods
- **Early July:** Building large pieces of software
- **Late July:** Elaborate code we can write in this framework
- **Early August:** Interaction with other programming paradigms

5-minute break

2 Expressions and Evaluation

We teach this course in a language called **Standard ML (SML)**. SML is:

- Functional
- Mostly pure
- Strongly-typed
- Statically-scoped
- Call-by-value, or "eager"
- Modular

SML is a functional language: rather than thinking of computation as *state mutations*, we think of computation as *evaluation of expressions*.

In this case, 2^{17} is an expression, which we want to *evaluate* down to obtain 131072.

What's 2^{17}?

Person 1: Hey, do you know what 2^{17} is?
Person 2: Yeah, it's 2^{17}.
Is Person 2 correct? Yes. Did they answer the question? No.

Person 1: Hey, do you know what 2^{17} is?
Person 2: Yeah, it's 2×2^{16}.
Is Person 2 correct? Yes. Did they answer the question? *Still no.*

The goal of functional programming is to produce *values*

Moral of the previous slide: Computational queries (like “what's 2^{17}?”) come with a built-in notion of what counts as an answer: 2^{17} and 2×2^{16} aren't acceptable answers to the question (whereas 131072 *is* an acceptable answer).

Functional programming has similar concepts:

- An **expression** is a syntactically-well-formed piece of code
- Some expressions are called **values**
- Expressions can be **evaluated** (or “reduced”), perhaps producing a value.

Example

Stepping

Evaluating an expression down to a value takes place in a finite number of discrete "steps". We trace out evaluations like follows.

```
(3+2)*(9-6) ⟹ 5 * (9-6)
            ⟹ 5 * 3
            ⟹ 15
```

Each of these is "one step", but generally the notation `e1` $\Longrightarrow$ `e2` means that evaluating `e1` steps to `e2` in *some finite number of steps*.

- For all expressions `e`, `e` $\Longrightarrow$ `e`
- If `e1` $\Longrightarrow$ `e2` and `e2` $\Longrightarrow$ `e3` then `e1` $\Longrightarrow$ `e3`

Values are the finished result of evaluation

So `exp(17)` $\Longrightarrow$ `131072`, but after that, we're *done*: there's no further `v` such that `131072` $\Longrightarrow$ `v` (besides `v` = `131072`).

An expression `e` is a **value** if evaluation *terminates* at `e`.

One problem: there are some expressions `e` which, if evaluated, do not result in a value.

- Raised exceptions:

```
1 div 0
```

- Looping forever:

$$\texttt{e1} \Longrightarrow \texttt{e2} \Longrightarrow \texttt{e3} \Longrightarrow \texttt{e4} \Longrightarrow \texttt{e5} \Longrightarrow \texttt{e6} \Longrightarrow \texttt{e7} \Longrightarrow \ldots$$

Claim For every syntactically-valid SML expression `e` (that we can evaluate), exactly one of the following holds:

- `e` $\Longrightarrow$ `v` for some value `v`
- the evaluation of `e` raises some exception
- the evaluation of `e` loops forever

Notation/Terminology: If `e` $\Longrightarrow$ `v` where `v` is a value, `e` is called **valuable**.
We'll also use the notation `e` $\hookrightarrow$ `v` to say that `e` $\Longrightarrow$ `v` and `v` is a value

Whether `e` is valuable, raises an exception, or loops forever is called the *runtime behavior* of `e`. Two expressions `e` and `e'` are *equivalent* if they have the same runtime behavior.

Defn. Two expressions `e` and `e'` are said to be **extensionally equivalent** (written `e` $\cong$ `e'`) if either:

- there is some value `v` such that `e` $\hookrightarrow$ `v` and `e'` $\hookrightarrow$ `v`
- the evaluation of `e` and `e'` both raise the *same* exception
- the evaluation of both `e` and `e'` loop forever

Referential Transparency: If `e` $\cong$ `e'`, then any instance of `e` can freely be replaced with `e'`

Summary

- Functional computation is the evaluation of expressions
- Evaluating a given expression either results in a value, raises an exception, or loops forever

Next Time

SML is:

- Functional
- Mostly pure
- Strongly-typed
- Statically-scoped
- Call-by-value, or “eager”
- Modular

Thank you!