Lecture 3: Evaluation and Operational Semantics

v1.1 23rd January 2025

Programming Languages (H)

Simon Fowler & Michele Sevegnani

Semester 2, 2024/2025

sli.do 1238123

University of Glasgow

The Chartered Institute for IT

BCSWomen Lovelace Colloquium at

University of Glasgow

- Great **speakers** from **diverse sectors**
- **Poster** competition – meet your peers, win **prizes**
- Experience talking about **technical topics**
- Network with the **Women in Computing** community
- **Careers fair** with major companies recruiting
- **Freebies** and goody bags
- Students from all levels (undergrad and postgrad) welcome

Event: Wed 16 April

Poster abstract deadline: Mon 3 Feb

more info here, or email Matthew.Barr@glasgow.ac.uk

sli.do 1238123

Reflection reflection

- Overall people seemed to enjoy, and understand most of, the first couple of lectures
- Several people found BNF grammars a bit tricky. They're a bit weird the first time you see them, but we'll be using them (for abstract syntax) throughout the course and will soon become familiar I hope – but if not, let me know
- One person stated "*[...] it's interesting to know just how much the CPU and BIOS helps out, providing all the basic text output services, storing details of PCI devices in memory*" – I assume this was for OS? I'll tell Paul!

sli.do 1238123

Dad Jokes (1)

- ″ What do you call it when Batman doesn't go to church? Christian Bale.″
- “Why did the scarecrow win the award? Because he was outstanding in his field”
- “I used to hate the hokey pokey, but I really turned myself around.”
- “Why do programmers prefer dark mode? Because light attracts bugs.”
- “What do you call a cow with no legs? Ground beef.”

sli.do 1238123

Dad Jokes (2)

- “I've been reading this book about anti-gravity and it's so good I can't put it down.”
- “I was going to try an all almond diet, but that's just nuts.”
- “What do you call a fake spaghetti? An imPasta!”
- “What do you call a man standing between two buildings? Ali!”
- My contribution: “My family wanted me to go to flamingo lessons, but I put my foot down!”

sli.do 1238123

Overview

- **Last time**
  - Introduction to the course and programming paradigms
  - Concrete and abstract syntax, grammars, ambiguity
- **This lecture:** Evaluation and Operational Semantics
  - Interpreters vs. Compilers
  - L_{If}: Extending L_{Arith} with Booleans and Conditionals
  - Introduction to big-step operational semantics
- **Labs start today!**
  - 15:00 – 17:00 in BO720 (what a way to start the weekend)

Part 1: Interpreters and Compilers

sli.do 1238123

Compilers

- We cannot run a program written in a high-level language directly on hardware!
  - (Mostly. There are some efforts to design specialised hardware for certain programming styles – see the HAFLANG project at Heriot-Watt https://haflang.github.io/)
- But for general-purpose hardware, a **compiler** translates code into (often a series of) lower-level languages, such that eventually they can be executed on hardware
- Often, even compiled code needs to be supported by a **runtime system** (that provides things like garbage collection)

sli.do 1238123

Example: OCaml Compiler Pipeline

Parse text into an AST

Simplify more complex constructs (e.g. optional arguments)

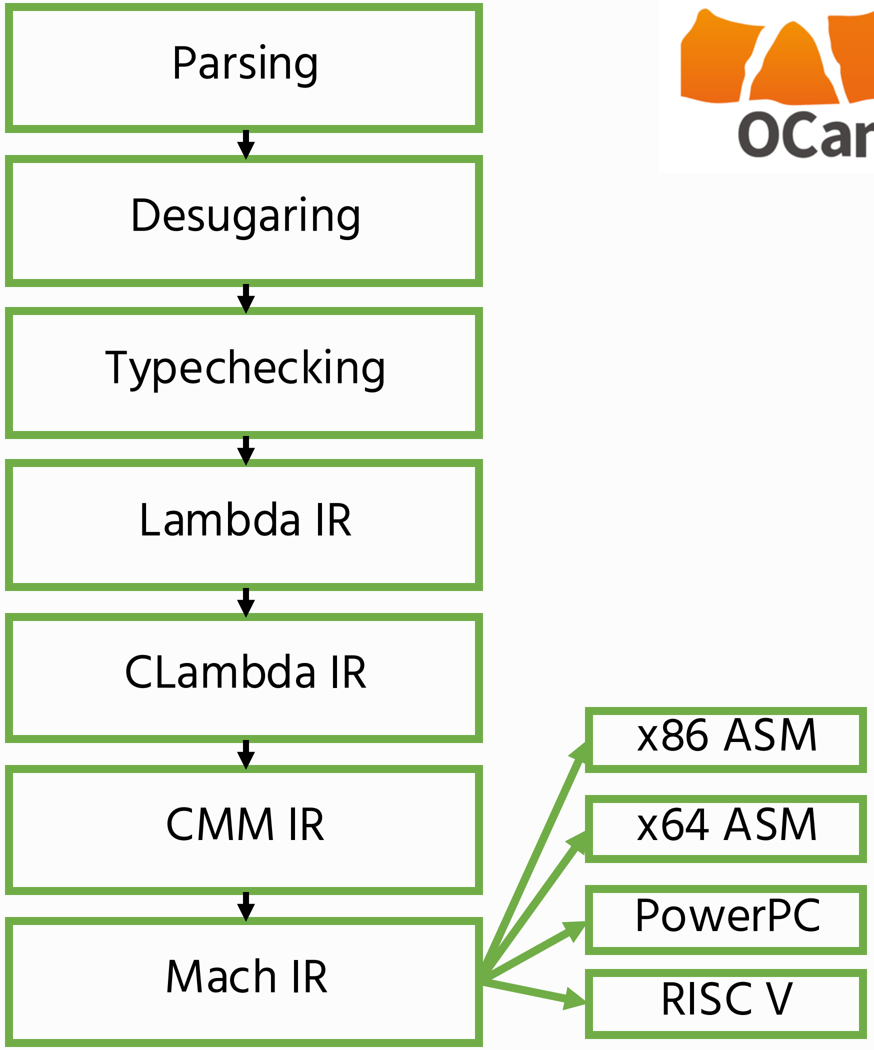
Ensure program is well typed, report type errors

Convert to first intermediate representation (IR) – small functional language based on the lambda calculus

Convert to second IR: make function environments explicit (closure conversion)

Convert to third IR: hoist all functions, small expression language

Convert to final IR: perform register allocation, explicit memory loads / stores



sli.do 1238123

Interpreters

- An **interpreter** is a program that accepts a program written in a given programming language, and executes it directly (without generating a separate executable).
- For example, Perl is fully interpreted: the interpreter is a separate program written in C
- Interpreters work by:
    - Fetching, analysing, and executing instructions (for imperative languages)
    - Evaluating subexpressions (for expression-based / functional languages)
- Generally, interpreters are **easier to write** but **slower** than compiled code
- The first lab will involve writing an interpreter for L_{If}

sli.do 1238123

Virtual Machines

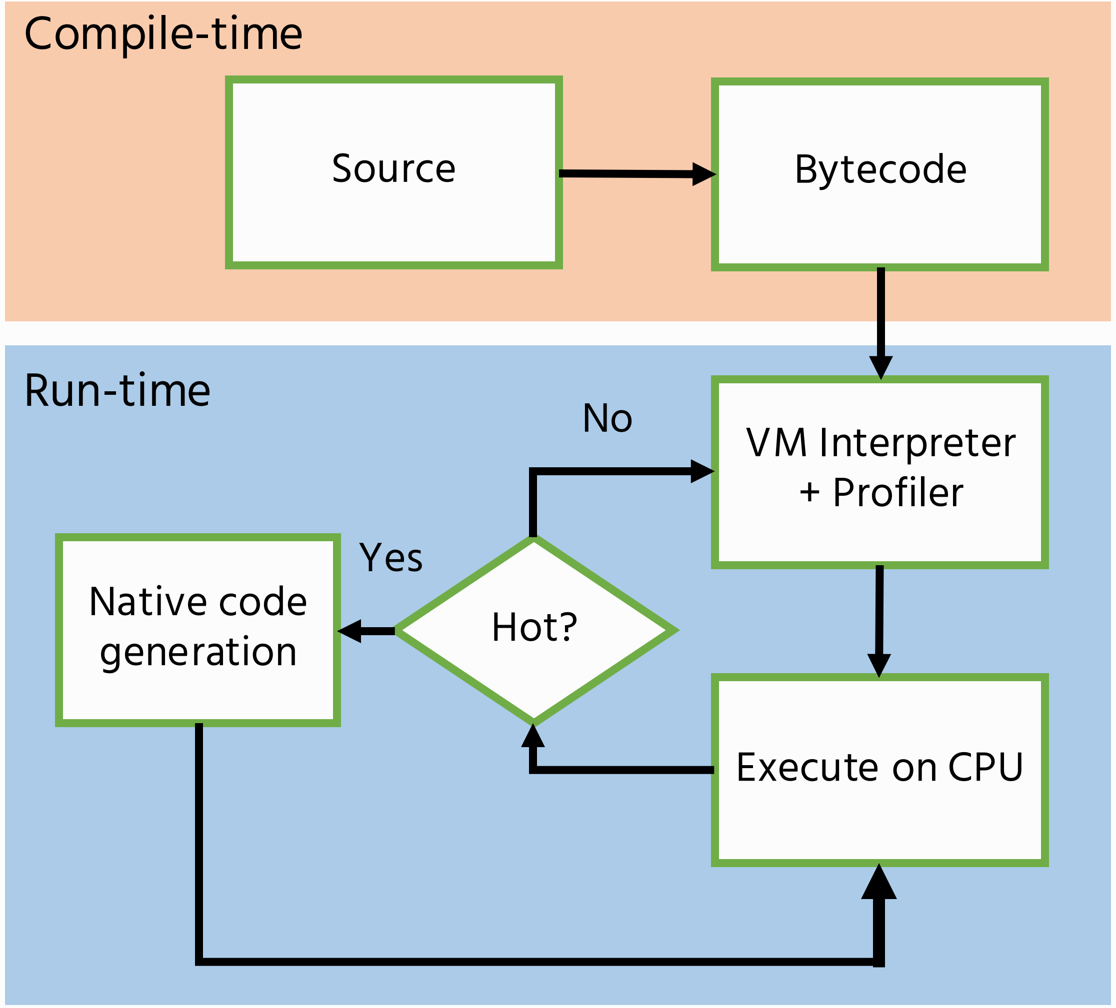
- A **physical machine** runs machine code (e.g. x86 assembly) directly
- In contrast, a **virtual machine** evaluates instructions (usually encoded in some sort of bytecode) by an interpreter

JVM

.NET

- Advantages of VMs:
  - **Platform independence**: Can run compiled code on multiple platforms
  - **Common backend**: multiple (quite different) languages can target the same backend – for example the .NET suite of languages target the .NET CLR, and all of { Java, Scala, Kotlin, Clojure } target JVM bytecode

Just-in-time (JIT) Compilers



- A JIT compiler is a middle-ground between compilers and interpreters, where code is compiled to native code **at run-time**
- JIT compilers operate **selectively**: they profile code and compile "hot" (i.e., frequently called) code
- An example is Java's HotSpot JIT compiler for Java

Diagram inspired by https://medium.com/@anshugiramkar2003/the-jit-compiler-revolution-empowering-developers-to-build-faster-smarter-software-25a404ef9a61

Sli.do break!

Part 2: Operational Semantics

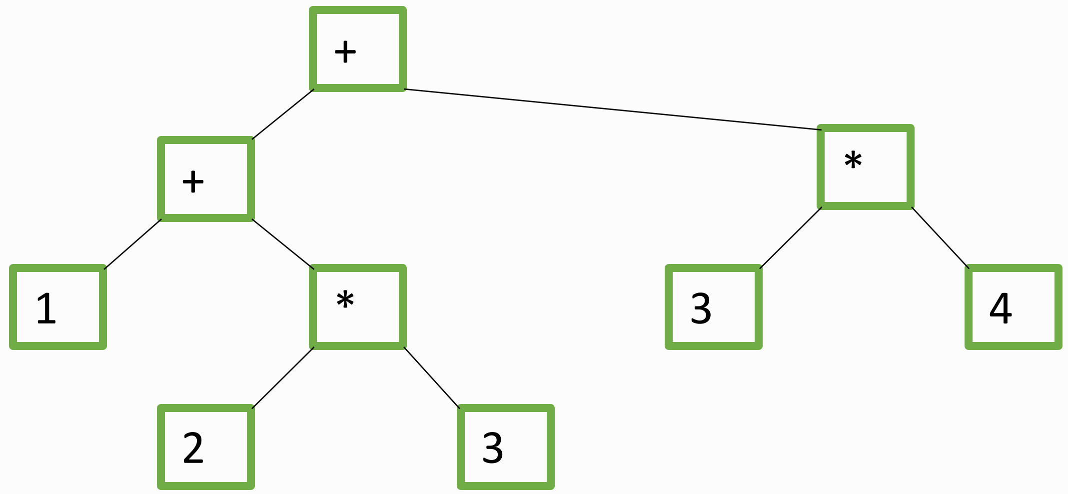
Recap: L_{Arith}

- In Lecture 2 we introduced L_{Arith}, a minimal programming language for representing arithmetic expressions
- L_{Arith} consists of integer literals *n*, and the four basic arithmetic operations
- We discussed the concrete syntax, but more importantly the **abstract syntax** that directly represents the expression structure and does not include syntactic noise like brackets

L_{arith} Abstract Syntax

```
Integers n
Operators ⊙   ::= + | - | * | /
Terms L, M, N ::= n
                | L ⊙ M
```

AST for `(1 + 2 * 3) + (3 * 4)`



sli.do 1238123

How do we say what an expression means?

- All we have so far is **syntax**: how an expression **looks**
- Intuitively we can work out how to ‘run’ an arithmetic expression – we perform the numeric computations and return the result
- Taking our running example of `(1 + (2 * 3)) + (3 * 4)`.
  - We start by multiplying 2 and 3
  - Once we’ve multiplied 2 and 3, we’ll get 6, and then we can add 1 to get 7
  - Then we can multiply 3 and 4 to get 12
  - Finally we can add 7 and 12 to get 19
- For arithmetic, this is all straightforward – but not all languages will be this simple!

sli.do 1238123

Aside: Determinism and the Church-Rosser Theorem

- We can reduce summands in **either order** and arrive at the same result: we say that evaluation is **deterministic** or satisfies the **Church-Rosser property**
- This is not the case for all PLs – especially those that have **side-effects** (e.g. sending packets, printing to the console)
- The **Theory of Computation** course (running again next year, hopefully!) treats this in much more depth

```
          (1 + 6) + (3 * 4)

7 + (3 * 4)             (1 + 6) + 12

               7 + 12

                 19
```

sli.do 1238123

Approaches to Programming Language Semantics

Operational

$$\rightarrow^{*}$$

Details **reduction** of an expression either to another expression (small-step) or a value (big-step)

Denotational

$$[\![-]\!]$$

Maps expressions to **semantic (mathematical) objects**.

Axiomatic

$$\{P\}\ M\ \{Q\}$$

Describes evaluation in terms of pre- and post-conditions on the program state

sli.do 1238123

Approaches to Programming Language Semantics

- **We will only consider operational semantics in this course.**
- Operational semantics are useful for seeing how the **state of a system evolves**, especially for systems with side-effects or concurrency.
  - However, they can sometimes be **verbose**, and reasoning about more intricate properties can be difficult
- Denotational semantics are extremely **powerful**, being particularly useful for **proving complex properties** such as program equivalence
  - However, modelling realistic language features (e.g. recursion or polymorphism) often requires **advanced mathematics** (e.g. domain theory / category theory)
- Axiomatic semantics are mostly used for verification of imperative programs, or describing the semantics of shared-memory concurrency in real-world processors (e.g. the specification of Arm processors)

sli.do 1238123

Textual descriptions

- Let's first write out textual descriptions for how to evaluate L_{Arith} expressions.
- We have two types of expression: integer literals *n*, and arithmetic operations $L \odot n$.

$$n$$

An integer *n* is a **value**: it is already in its simplest form

$$L \odot M$$

To evaluate an arithmetic operation, evaluate L to a value, evaluate M to a value, and then perform the operation on the results

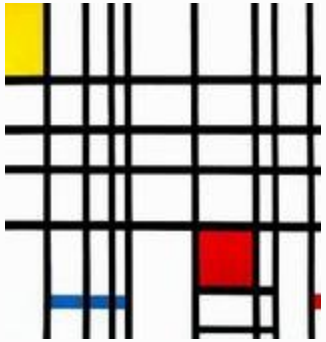
sli.do 1238123

Pitfalls of Textual Descriptions

- Textual descriptions can be **imprecise**: the language designer might mean something different to what is understood by the language implementer
- We can’t **prove properties** about textual descriptions: they are not mathematically defined
- Textual descriptions **don’t scale**: more complex language features require lots of (confusing) text to describe
- Textual descriptions leave room for **edge cases** due to ambiguity

sli.do 1238123

Example convoluted textual description



“Pops the top two values off the stack and "rolls" the remaining stack entries to a depth equal to the second value popped, by a number of rolls equal to the first value popped.

A single roll to depth *n* is defined as burying the top value on the stack *n* deep and bringing all values above it up by 1 place.

A negative number of rolls rolls in the opposite direction.

A negative depth is an error and the command is ignored.

If a roll is greater than an implementation-dependent maximum stack depth, it is handled as an implementation-dependent error, though simply ignoring the command is recommended.”

https://www.dangermouse.net/esoteric/piet.html

sli.do 1238123

Expressions and Values

- A **value** is data, and is the final result of a computation. It cannot evaluate further.
- **Expressions** in L_{Arith} include binary operations, which may need to evaluate further. In our language, every expression should eventually evaluate to a value.

L_{Arith} Abstract Syntax

```
Integers n
Operators ⊙   ::= + | - | * | /
Values V, W   ::= n
Terms L, M, N ::= n
                | L ⊙ M
```

sli.do 1238123

(Big-step) Operational Semantics

$$\boxed{M \Downarrow V}$$

Big-step operational semantics involves showing how an **expression M evaluates to a value V**. You can think of the judgement on the left being like a function signature `Expression -> Value`.

We then need **inference rules** to show how expressions evaluate: remember that an inference rule says that if the **premises** (P_1, P_2, …, P_n on the top of the rule) hold, then the **conclusion** (Q, on the bottom of the rule) holds.

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q}$$

Often, it is useful to read PL inference rules **bottom-up**

sli.do 1238123

L_{Arith} Semantics

$$\boxed{M \Downarrow V}$$

First evaluate L to a value V

Next, evaluate M to a value W

$$\frac{}{n \Downarrow n} \qquad\qquad \frac{L \Downarrow V \qquad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

All integers are values and cannot reduce further, so we are done

To evaluate L $\odot$ M…

The result is the actual integer operation on the two values V and W

sli.do 1238123

Showing how an expression evaluates

- Given the inference rules, we can now use the semantics to show how any expression evaluates down to a value by constructing a **derivation tree**
- We start with our expression at the bottom of the tree, and match the **outermost subexpression** to an inference rule
- We can then work upwards, until we reach an axiom (a rule without any premises), and finally fill in the values on the way down

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \ \Downarrow \ V \mathbin{\widehat{\odot}} W}$$

$$\frac{}{(1 + (2 * 3)) + (3 * 4) \Downarrow ?}$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \ \Downarrow \ V \mathbin{\widehat{\odot}} W}$$

$$\frac{\overline{1 + (2 * 3) \Downarrow ?} \qquad \overline{3 * 4 \Downarrow ?}}{(1 + (2 * 3)) + (3 * 4) \Downarrow ?}$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \;\Downarrow\; V \mathbin{\widehat{\odot}} W}$$

$$\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1} \qquad \dfrac{}{2 * 3 \Downarrow ?}}{1 + (2 * 3) \Downarrow ?} \qquad \dfrac{}{3 * 4 \Downarrow ?}}{(1 + (2 * 3)) + (3 * 4) \Downarrow ?}$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \;\Downarrow\; V \mathbin{\widehat{\odot}} W}$$

$$\dfrac{\dfrac{}{1 \Downarrow 1} \qquad \dfrac{\dfrac{}{2 \Downarrow 2} \qquad \dfrac{}{3 \Downarrow 3}}{2 * 3 \Downarrow ?}}{\dfrac{1 + (2 * 3) \Downarrow ?}{} \qquad \dfrac{}{3 * 4 \Downarrow ?}}$$

$$(1 + (2 * 3)) + (3 * 4) \Downarrow ?$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \ \Downarrow\ V \mathbin{\widehat{\odot}} W}$$

$$\cfrac{\cfrac{}{1 \Downarrow 1} \qquad \cfrac{\cfrac{}{2 \Downarrow 2} \quad \cfrac{}{3 \Downarrow 3}}{2 * 3 \Downarrow 6}}{1 + (2 * 3) \Downarrow ?} \qquad \cfrac{}{3 * 4 \Downarrow ?}$$

$$\overline{(1 + (2 * 3)) + (3 * 4) \Downarrow ?}$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\dfrac{\dfrac{}{1 \Downarrow 1} \quad \dfrac{\dfrac{}{2 \Downarrow 2} \quad \dfrac{}{3 \Downarrow 3}}{2 * 3 \Downarrow 6}}{1 + (2 * 3) \Downarrow 7} \qquad \dfrac{}{3 * 4 \Downarrow ?}$$

$$\overline{(1 + (2 * 3)) + (3 * 4) \Downarrow ?}$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\dfrac{\dfrac{}{1 \Downarrow 1} \qquad \dfrac{\dfrac{}{2 \Downarrow 2} \quad \dfrac{}{3 \Downarrow 3}}{2 * 3 \Downarrow 6}}{1 + (2 * 3) \Downarrow 7} \qquad \dfrac{\dfrac{}{3 \Downarrow 3} \quad \dfrac{}{4 \Downarrow 4}}{3 * 4 \Downarrow 12}$$
$$\overline{(1 + (2 * 3)) + (3 * 4) \Downarrow ?}$$

sli.do 1238123

Example 1

$$\frac{}{n \Downarrow n} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\dfrac{\dfrac{}{1 \Downarrow 1} \quad \dfrac{\dfrac{}{2 \Downarrow 2} \quad \dfrac{}{3 \Downarrow 3}}{2 * 3 \Downarrow 6}}{1 + (2 * 3) \Downarrow 7} \quad \dfrac{\dfrac{}{3 \Downarrow 3} \quad \dfrac{}{4 \Downarrow 4}}{3 * 4 \Downarrow 12}$$

$$\overline{(1 + (2 * 3)) + (3 * 4) \Downarrow 19}$$

sli.do 1238123

Example 2 (on the visualiser)

Sli.do break!

Part 3: Extending LArith with Booleans and Conditionals

sli.do 1238123

Additional features, informally

- L_{Arith} is still quite far from a programming language!
- We'll now extend L_{Arith} with Booleans and conditional expressions, to show our first instance of **branching control flow**
- We'll need some extra constructs:
  - **`true`** and **`false`**, which are our Boolean values
  - **Additional operators** that work on integers (<, >) and Booleans (&&, ||) and both (==)
  - A **conditional expression** `if L then M else N` that evaluates `M` if `L` evaluates to `true`, and `N` if `L` evaluates to `false`

sli.do 1238123

L_{If} Abstract Syntax

L_{If} Abstract Syntax

```
Integers n
Booleans b      ::= true | false
Operators ⊙     ::= + | - | *  | /
                  | < | > | && | ||
                  | ==
Values V, W     ::= n | b
Constants c     ::= n | b
Terms L, M, N ::= c
                  | L ⊙ M
                  | if L then M else N
```

- The syntax of L_{If} is on the left (new constructs highlighted).
- Note that there are some design decisions
  - We could include values V directly in terms L, M, N (rather than having the syntactic class for constants) but it's often good to have a syntactic separation of **static** and **runtime** terms
  - We could just include `true` and `false` directly in values / terms, rather than having a separate syntactic category

sli.do 1238123

L_{If} reduction rules

The rules from L_{Arith} are very similar. We generalise the value rule to arbitrary constants rather than just numbers. We don't need to change the binary operator rule.

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \qquad M \Downarrow W}{L \odot M \;\Downarrow\; V \mathbin{\widehat{\odot}} W}$$

We need **two** rules for conditional statements: one for if the predicate returns true (which evaluates the first branch), and another for if the predicate returns false (which evaluates the second branch).

$$\frac{L \Downarrow \mathtt{true} \qquad M \Downarrow V}{\mathtt{if}\, L \,\mathtt{then}\, M \,\mathtt{else}\, N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \qquad N \Downarrow V}{\mathtt{if}\, L \,\mathtt{then}\, M \,\mathtt{else}\, N \Downarrow V}$$

Finally we need **two** rules for equality: one if both expressions evaluate to two identical values, and one if not.

$$\frac{M \Downarrow V \qquad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \qquad N \Downarrow W \qquad V \neq W}{M == N \Downarrow \mathtt{false}}$$

sli.do 1238123

Example L_{If} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \;\Downarrow\; V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \mathtt{true} \quad M \Downarrow V}{\mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \quad N \Downarrow V}{\mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \mathtt{false}}$$

$$\frac{}{\mathtt{if}\ 5 > 6\ \mathtt{then}\ 3\ \mathtt{else}\ (4 * 5) + 6 \Downarrow ?}$$

sli.do 1238123

Example L_{If} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \ \Downarrow\ V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \mathtt{true} \quad M \Downarrow V}{\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \quad N \Downarrow V}{\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \mathtt{false}}$$

$$\frac{\dfrac{}{5 > 6 \Downarrow ?}}{\mathtt{if}\ 5 > 6\ \mathtt{then}\ 3\ \mathtt{else}\ (4 * 5) + 6 \Downarrow ?}$$

Example L_{lf} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \;\Downarrow\; V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \mathtt{true} \quad M \Downarrow V}{\mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \quad N \Downarrow V}{\mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \mathtt{false}}$$

$$\cfrac{\cfrac{\cfrac{}{5 \Downarrow 5} \quad \cfrac{}{6 \Downarrow 6}}{5 > 6 \Downarrow \mathtt{false}}}{\mathtt{if}\ 5 > 6\ \mathtt{then}\ 3\ \mathtt{else}\ (4 * 5) + 6 \Downarrow ?}$$

sli.do 1238123

Example L_{lf} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \;\Downarrow\; V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \mathtt{true} \quad M \Downarrow V}{\mathtt{if}\, L \,\mathtt{then}\, M \,\mathtt{else}\, N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \quad N \Downarrow V}{\mathtt{if}\, L \,\mathtt{then}\, M \,\mathtt{else}\, N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \mathtt{false}}$$

$$\cfrac{\cfrac{\cfrac{}{5 \Downarrow 5} \quad \cfrac{}{6 \Downarrow 6}}{5 > 6 \Downarrow \mathtt{false}} \qquad \cfrac{}{(4 * 5) + 6 \Downarrow ?}}{\mathtt{if}\ 5 > 6\ \mathtt{then}\ 3\ \mathtt{else}\ (4 * 5) + 6 \Downarrow ?}$$

sli.do 1238123

Example L_{If} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \texttt{true} \quad M \Downarrow V}{\texttt{if}\, L\, \texttt{then}\, M\, \texttt{else}\, N \Downarrow V} \qquad \frac{L \Downarrow \texttt{false} \quad N \Downarrow V}{\texttt{if}\, L\, \texttt{then}\, M\, \texttt{else}\, N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \texttt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \texttt{false}}$$

$$\frac{\dfrac{\overline{5 \Downarrow 5} \quad \overline{6 \Downarrow 6}}{5 > 6 \Downarrow \texttt{false}} \qquad \dfrac{\overline{4 * 5 \Downarrow ?} \quad \overline{6 \Downarrow 6}}{(4 * 5) + 6 \Downarrow ?}}{\texttt{if } 5 > 6 \texttt{ then } 3 \texttt{ else } (4 * 5) + 6 \Downarrow ?}$$

sli.do 1238123

Example L_{If} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \mathtt{true} \quad M \Downarrow V}{\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \quad N \Downarrow V}{\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \mathtt{false}}$$

$$\dfrac{\dfrac{\dfrac{}{5 \Downarrow 5} \quad \dfrac{}{6 \Downarrow 6}}{5 > 6 \Downarrow \mathtt{false}} \qquad \dfrac{\dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{5 \Downarrow 5}}{4 * 5 \Downarrow 20} \quad \dfrac{}{6 \Downarrow 6}}{(4 * 5) + 6 \Downarrow ?}}{\mathtt{if}\ 5 > 6\ \mathtt{then}\ 3\ \mathtt{else}\ (4 * 5) + 6 \Downarrow ?}$$

Example L_{If} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \texttt{true} \quad M \Downarrow V}{\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N \Downarrow V} \qquad \frac{L \Downarrow \texttt{false} \quad N \Downarrow V}{\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \texttt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \texttt{false}}$$

$$\frac{\dfrac{\dfrac{}{5 \Downarrow 5} \quad \dfrac{}{6 \Downarrow 6}}{5 > 6 \Downarrow \texttt{false}} \qquad \dfrac{\dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{5 \Downarrow 5}}{4 * 5 \Downarrow 20} \quad \dfrac{}{6 \Downarrow 6}}{(4 * 5) + 6 \Downarrow 26}}{\texttt{if}\ 5 > 6\ \texttt{then}\ 3\ \texttt{else}\ (4 * 5) + 6 \Downarrow ?}$$

sli.do 1238123

Example L_{If} derivation

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

$$\frac{L \Downarrow \mathtt{true} \quad M \Downarrow V}{\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \Downarrow V} \qquad \frac{L \Downarrow \mathtt{false} \quad N \Downarrow V}{\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \mathtt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \mathtt{false}}$$

$$\cfrac{\cfrac{\cfrac{}{5 \Downarrow 5} \quad \cfrac{}{6 \Downarrow 6}}{5 > 6 \Downarrow \mathtt{false}} \qquad \cfrac{\cfrac{\cfrac{}{4 \Downarrow 4} \quad \cfrac{}{5 \Downarrow 5}}{4 * 5 \Downarrow 20} \quad \cfrac{}{6 \Downarrow 6}}{(4 * 5) + 6 \Downarrow 26}}{\mathtt{if}\ 5 > 6\ \mathtt{then}\ 3\ \mathtt{else}\ (4 * 5) + 6 \Downarrow 26}$$

sli.do 1238123

Example L_{lf} derivation 2 (On visualiser)

sli.do 1238123

Conclusion

- In this lecture, we've begun to see:
  - (at a high level) how programs are run
  - how we can specify the meaning of a program using big-step semantics
- The next lecture will be more practical than theoretical (to prepare you for the labs): we will talk about the ANTLR toolkit and the SVM virtual machine
- The labs will show you how to use operational semantics to guide an implementation
- See you in 10 minutes!