What does a program mean?

- We have learned how to specify syntax.
- Example: let x = lambda lambda is not a valid L program
- But we have not yet talked about what the meaning of a program is.
- First Question: What is the meaning of a program in L?
- Answer: The value the program evaluates to
- Example: let x = 3 in x Value: 3

How to specify meaning of programs

- Option 1: Don’t worry too much
  - Developer of language has some informal concept of the intended meaning, implement a compiler/interpreter that does whatever the language designers believe to be reasonable.
  - Then, declare the meaning to be whatever the compiler produces
  - A terrible idea
- Option 2: Try to write out precisely the meaning of each language construct in documentation, then follow this description in implementation
  - Example: Describe the meaning of !e in the L language:
    - First attempt: “This evaluates to the head of e”
    - What if e is not a list?
    - Second attempt: “This evaluates to the head of e if e is a list, and to e otherwise”
    - What if e is Nil? …
How to specify meaning of programs: Option 2

- Written language is, by nature, ambiguous. It is very difficult to fully specify the meaning of all language constructs this way
- Easy to miss cases
- Results in long, complicated and difficult to understand specifications, but an improvement over no specification

Written specification in practice

- Let’s look at the ISO C++ standard: 879 pages, page 116:

Precisely Specifying Meaning

- Recall λ-calculus:
- To specify the meaning of expressions, we defined one single operation: $\beta$ reduction
- Specifically, we wrote $\lambda x. e_1 \ e_2 \to^\beta e_1[e_2/x]$
- Can read this as follows: If you see an expression of the form $\lambda x. e_1 \ e_2$, you can compute its result as $e_1[e_2/x]$.

Operational Semantics

- Let’s try the same in the language of arithmetic expression with the grammar:

  \[ S \rightarrow c \mid S_1 + S_2 \mid S_1 \times S_2 \]

- What is the meaning of an integer constant?
  - More precisely: If we see an expression of the form $c$, its value is $c$
  - We will write: $\vdash c : c$
  - Read as: “we can prove for any expression of the form $c$ that the meaning of this expression is $c$”

Operational Semantics Cont.

- How about the expression $S_1 + S_2$?
  - $\vdash S_1 + S_2$?
  - Problem: To describe the meaning of $S_1 + S_2$, we need to know the meaning (value) of $S_1$ and $S_2$
  - Solution: Use hypotheses: We want to say “Assuming $S_1$ evaluates to $e_1$ and $S_2$ evaluates to $e_2$, the value of $S_1 + S_2$ is $e_1 + e_2$”
  - We write this as:
    \[
    \vdash S_1 : e_1 \\
    \vdash S_2 : e_2 \\
    \vdash S_1 + S_2 : e_1 + e_2
    \]

Inference Rules

- This notation is known as inference rule:
  - Hypothesis 1
  - ... Hypothesis N
  \[ \vdash \text{Conclusion} \]
- This means “given hypothesis 1, ... N, the conclusion is provable”
- Example:
  - Midterm 1 grade $\geq 70$
  - ... Final grade $\geq 140$
  \[ \vdash \text{Final grade: A} \]
Inference Rules cont.

- A hypothesis in an inference rule may use other rules

- Example:
  \[
  \vdash S_1 : c_1 \\
  \vdash S_2 : c_2 \\
  \vdash S_1 + S_2 : c_1 + c_2
  \]

- You can tell this by a \( \vdash \) in at least one of the hypotheses.

- Such rules are called **inductive** since they define the meaning of an expression in terms of the meaning of subexpressions.

- Rules that to not have \( \vdash \) in any hypothesis are **base cases**

- A system with only inductive rules is nonsensical

---

Operational Semantics

- Back to the rule for +:
  \[
  \vdash S_1 : c_1 \\
  \vdash S_2 : c_2 \\
  \vdash S_1 + S_2 : c_1 + c_2
  \]

- Let’s focus on the first hypothesis \( \vdash S_1 : c_1 \).

- **Question:** Can you write \( S_1 = c_1 \)?

- **Answer:** Yes, but now your first hypothesis is: “Assuming \( S_1 \) is the integer constant \( c_1 \) ⇒ this rule no longer applies if, for example, \( S_1 = 2 \times 3 \).

- Read \( \vdash \) as “is provable by using our set of inference rules”.

---

Operational Semantics and Order

- **Important Point:** This notation does not specify an order between hypothesis.

- This means that
  \[
  \vdash S_1 : c_1 \\
  \vdash S_2 : c_2 \\
  \vdash S_1 + S_2 : c_1 + c_2
  \]

- and
  \[
  \vdash S_2 : c_2 \\
  \vdash S_1 : c_1 \\
  \vdash S_1 + S_2 : c_1 + c_2
  \]

have exactly the same meaning

---

Full Operational Semantics

- Here are the full operational semantics of the language
  \[
  S \rightarrow c | S_1 + S_2 | S_1 \times S_2
  \]

- \( \vdash c : c \)

- \( \vdash S_1 : c_1 \)

- \( \vdash S_2 : c_2 \)

- \( \vdash S_1 + S_2 : c_1 + c_2 \)

- \( \vdash S_1 \times S_2 : c_1 \times c_2 \)

---

Using Operational Semantics

- Consider the expression \((21 \times 2) + 6\)

- Here is how to derive the value of this expression with the operational semantics:
  \[
  \vdash 21 : 21 \\
  \vdash 2 : 2 \\
  \vdash (21 \times 2) : 42 \\
  \vdash (21 \times 2) + 6 : 48
  \]

- This is a formal proof that the expression \((21 \times 2) + 6\) evaluates to 48 under the defined operational semantics

- Observe that these proofs have a tree structure: Each subexpression forms a new branch in the tree

---

Operational Semantics of L

- Let’s try to give operational semantics to the L language:

- Start with integers: \( \text{Integer} \ 1 \)

- \( \vdash 1 : i \)

- The \( i \) in the hypothesis and to the left of the colon is the syntactic number in the source code of L

- The \( i \) after the colon is the value of the integer \( i \).

- This sounds nitpicky, but is important to understand this notation.
Operational Semantics of L

- Consider the (integer) plus expression in L:
  \[ \vdash e_1 : \text{i1 (integer)} \]
  \[ \vdash e_2 : \text{i2 (integer)} \]
  \[ \vdash e_1 + e_2 : e_1 + e_2 \]

- Side remark: The hypothesis can be written in separate lines (but not when giving a derivation tree).

- Here, the hypotheses require that \( e_1 \) and \( e_2 \) evaluate to integers.

- Question: What happens if \( e_1 \) evaluates to a list?

- Answer: No rule applies and computation is "stuck". This means the L program does not evaluate to anything.

- In practice: This is a run-time error.

Operational Semantics of L

- On to the key construct: \( \lambda \)

- Let's write semantics for the simple application \( (e_1 e_2) \):
  \[ \vdash (e_1 e_2) : e \]

- Answer: Nothing. The written order of hypotheses is irrelevant.

- Hypothesis: \( \vdash e_1 : \text{lambda x. e'}_1 \)

- Now, how do we evaluate \( (e_1 e_2) \) ? \( \vdash e'_1[e_2/x] : e \)

- Conclusion: \( \vdash (e_1 e_2) : e \)

- Final rule: \( \vdash e_1 : \text{lambda x. e'}_1 \)
  \( \vdash e'_1[e_2/x] : e \)
  \[ \vdash (e_1 e_2) : e \]

The Lambda Rule

- Question: What would change if we write the hypothesis as \( e_1 = \text{lambda x. e'}_1 \)
  \( \vdash e'_1[e_2/x] : e \)
  \[ \vdash (e_1 e_2) : e \]

- Answer: This would still give semantics to \( \text{(lambda x.x 3)} \), but no longer to let \( y=\text{lambda x.x in (y 3)} \)

Order of Evaluation

- What would change if we write:
  \[ \vdash e'_1[e_2/x] : e \]
  \[ \vdash e_1 : \text{lambda x. e'}_1 \]
  \[ \vdash (e_1 e_2) : e \]

- Answer: Nothing. The written order of hypotheses is irrelevant.

Consider the (integer) plus expression in L:

- Integer minus:
  \[ \vdash e_1 : \text{i1 (integer)} \]
  \[ \vdash e_2 : \text{i2 (integer)} \]
  \[ \vdash e_1 - e_2 : e_1 - e_2 \]

- Integer times:
  \[ \vdash e_1 : \text{i1 (integer)} \]
  \[ \vdash e_2 : \text{i2 (integer)} \]
  \[ \vdash e_1 * e_2 : e_1 * e_2 \]

Important Point: Operational semantics can encode order, but not through syntactic ordering.
The Lambda Rule cont.

- Consider both rules:
  \[ \frac{\vdash e_1 : \text{lambda } x. e'_1}{\vdash (e_1 e_2) : e} \]

- Consider the expression \((\text{lambda } x. 3 \times x) (77\times 3 - 2)\):
  - Rule 1 evaluates this expression to \('3'\)
  - Rule 2 “gets stuck” and returns no value since adding an integer and string is undefined (we have not given a rule)
  - Two reasonable ways of defining application, but different semantics!

Call-by-name vs. Call-by-value

- Not evaluating the argument before substitution is known as call-by-name, evaluating the argument before substitution as call-by-value.
  - Languages with call-by-name: classic lambda calculus, ALGOL 60, L
  - Languages with call-by-value: C, C++, Java, Python, FORTRAN, ...
  - Advantage of call-by-name: If argument is not used, it will not be evaluated
  - Disadvantage: If argument is uses \(k\) times, it will be evaluated \(k\) times!

Semantics of the let-binding

- Let’s try to define the semantics of the let-binding in L:
  \((\text{let } x = e_1 \text{ in } e_2)\)

  - One possibility:
    \[ \vdash e_1 : e'_1 \quad \vdash e_2[e'_1/x] : e \]
    \[ \vdash (\text{let } x = e_1 \text{ in } e_2) : e \]
  - What about the following definition?
    \[ \vdash e_2[e_1/x] : e \]
    \[ \vdash (\text{let } x = e_1 \text{ in } e_2) : e \]

  - Are these definitions equivalent?

Eager vs. Lazy Evaluation

- Evaluating \(e_1\) before we know that it is used is called **eager evaluation**
- Waiting until we need it is **lazy evaluation**.
- These are analogous to call-by-name/call-by value in trade offs.

Definition of let bindings

- But currently there is one problem common to both the eager and lazy definition of the let binding.
  - Consider the following valid L program:
    \[ \text{let } f = \lambda x. \text{ if } x <= 0 \text{ then } 1 \text{ else } x \times (f(x-1)) \text{ in } (f 2) \]

  - What happens if we use our definition of let on this expression? For brevity, let’s use the lazy one here, but the same problem exists with the eager one:
    \[ \vdash (f 2)[(\text{let } x = 0 \text{ then } 1 \text{ else } x \times (f(x-1))] : ? \]
    \[ \vdash (\text{let } f = \lambda x. \text{ if } x = 0 \text{ then } 1 \text{ else } x \times (f(x-1))] \text{ in } (f 2) : ? \]
**Let Binding**

- We have already seen this problem when studying lambda calculus.
- But this time, we want to solve it. After all, who wants to use the Y-combinator for every recursive function!
- **Solution:** Add an environment to our rules that tracks mappings between identifiers and values
- Specifically, write the let rule as follows:

\[
E ⊢ e_1 : e'_1 \\
E[x ← e'_1] ⊢ e_2 : e \\
E ⊢ \text{let } x = e_1 \text{ in } e_2 : e
\]

**Environments**

- You can think of the environment as storing information to be used by other rules
- An environment maps keys to values
- **Notation:** \(E[x ← y]\) means new environment with all mappings in \(E\) and the mapping \(x ↦ y\) added.
- If \(x\) was already mapped in \(E\), the mapping is replaced
- **Notation:** \(E (x) = y\) means bind value of key \(x\) in \(E\) to \(y\). If no mapping \(x ↦ y\) exits in \(E\), this “gets stuck”

**Environments Example**

- Consider the L program \(\text{let } x = 3 \text{ in } x\)
- Here is the proof that this program evaluates to 3:

\[
E ⊢ 3 : 3 \\
E[x ← 3] (x) = 3 \\
E ⊢ \text{let } x = 3 \text{ in } x : 3
\]

**Conclusion**

- We have seen how to formally give meaning to programs
- The formalism we have studied is called large-step operational semantics
- **Next time:** Semantics for more L constructs and another alternative formalism for specifying meaning of programs