COMP105 Assignment: Operational Semantics

Originally due Monday, February 2 at 11:59PM.
Extended to Wednesday, February 4 at 11:59PM by Winter Storm Juno.
Extended to Friday, February 6 at 5:59pm by Winter Storm Reprise.

The purpose of this assignment is to help you develop rudimentary skills with operational semantics, inference rules, and syntactic proof technique. You will use these skills heavily throughout the first two-thirds of the course, and you will use them again later if you ever want to keep up with the latest new ideas in programming languages or if you want to go on to advanced study.

Some of the essential skills are

- Understanding what judgment forms mean, how to read them, and how to write them
- Understanding what constitutes a valid syntactic proof, known as a *derivation*
- Understanding how a valid derivation in the operational semantics relates to the successful evaluation of an expression
- Proving facts about families of programs by reasoning about derivations, a technique known as *metatheory*
- Using operational semantics to express language features and language-design ideas
- Connecting operational semantics with informal English explanations of language features
- Connecting operational semantics with code in compilers or interpreters

Few of these skills can be mastered in a single assignment. When you've completed the assignment, IÂ hope you will feel confident of your knowledge of exactly the way judgment forms, inference rules, and derivations are written. On the other skills, you'll have made a start.

Part A: Adding Local Variables to the Interpreter (Work with a partner, 25 percent)

This exercise will help you understand how operational semantics is implemented, and how language changes can be realized in C code. You will do Exercise 30 from page 67 of Ramsey's book. **We recommend that you solve this problem with a partner, but this solution must be kept separate from your other solutions. Your programming partner, if any, must not see your other work.**

- Get your copy of the code from the book by running

  ```
  git clone linux.cs.tufts.edu:/comp/105/build-prove-compare
  ```

  or if that doesn't work, from a lab or linux machine, try

  ```
  git clone /comp/105/build-prove-compare
  ```

  You can find the source code from Chapter 2 in subdirectory `bare/impcore` or `commented/impcore`. The `bare` version, which we recommend, contains just the C code from the book, with simple comments identifying page numbers. The `commented` version, which you may use if you like, includes part of the book text as commentary.
- We provide new versions of `all.h`, `ast.c`, `definition-code.c` and `parse.c` that handle local variables. These version are found in subdirectory `bare/impcore-with-locals`. There are not many changes; to see what is different, try running

  ```
  diff -r bare/impcore bare/impcore-with-locals
  ```

  You may wish to try the `-u` or `-y` options with `diff`

In the directory `bare/impcore-with-locals`, you can build an interpreter by typing `make`, but when you run the interpreter, it will halt with an assertion failure. You'll need to change the interpreter to add local variables:

- In `impcore.c`, you will have to modify the functions in the initial basis to use the new syntax.
- In `eval.c`, you will have to modify the evaluator to give the right semantics to local variables. Local variables that have the same name as a formal parameter should hide that formal parameter, as in C.
- You also have the right to modify other files as you see fit.

- Create a file called `README` in this directory (your `impcore-with-locals` directory). Use this file to describe your solution to this problem.

Part B: Operational semantics, derviations, and metatheory (Individual work, 75 percent)

These are exercises intended to help you become fluent with operational semantics. **Do not share your solutions with any programming partners.** We encourage you to discuss ideas, but **nobody else may see your rules, your derivations, or your code**.
If you have difficulty, find a TA, who can help you work a couple of similar problems.

For these exercises you will turn in two files: `theory.pdf` and `16.imp`. For file `theory.pdf`, you will probably find it easiest to write your answers on paper and scan it. Please see the note about how to organize your answers.

- Do Exercises 15 and 16 on page 64 of Ramsey's book. The purpose of these exercises is to give you a feel for the kinds of choices language designers can make. Please include your answer to exercise 15 as part of `theory.pdf`. Please put your answer to exercise 16 in file `16.imp`.
- Do Exercise 12 on page 63 of Ramsey's book. The purpose of the exercise is to help you develop your understanding of derivations, so be sure to make your derivation *complete* and *formal*. You can write out a derivation like the ones in the book, as a single proof tree with a horizontal line over each node, or if you prefer, you can write a sequence of judgments, number each judgment, and write a proof tree containing only the numbers of the judgments, which you will find easier to fit on the page.

  Please include your answer as part of file `theory.pdf`.
- Do Exercise 13 on page 63 of Ramsey's book. Now that you know how to *write* a derivation, in this exercise you start *reasoning* about derivations.

  Please include your answer as part of file `theory.pdf`.
- Do Exercise 22 on pages 64–65 of Ramsey's book. In this exercise you raise your game again, reasoning about the *set* of all *valid* derivations. When you have got your thinking to this level, you can see how language designers use operational semantics to show nontrivial properties of their languages—and how these properties can guide implementors.

  Please include your answer in file `theory.pdf`.

  Metatheoretic proofs are probably unfamiliar, so you may want to look at some sample cases we have provided to help you. Also, to relieve some of the tedium (which is very common in programming-language proofs), we suggest that you allow your proof for the AddApply case to stand in for all other cases involving primitive operators. We also suggest that you simplify by leaving out the global environment .

Organizing the answers to Part B

To help us read your answers to Part B, we need for you to organize them carefully:

- The answer to each question must **start on a new page**.
- The answers **must appear in order**: Exercises 12, 13, 15, and finally 22.

Submitting

Before submitting code, **test it**. We do not provide any tests; you write your own.

- To submit part B, which you will have done by yourself, change into the appropriate directory and run `submit105-opsem-solo` to submit your work. In addition to files `16.imp` and `theory.pdf`, please also include a file called `README`. Use your `README` file to
  - Tell us *how long it took you to complete the entire assignment* (parts A and B)
  - Tell us how well you think you did on each of the four evaluation dimensions for operational semantics below (Semantics, Rules, Derivations, and Metatheory).
  - Tell us anything else you think it useful for us to know.

  (If you wish to use PDF, then please submit `README.pdf` instead of `README`.)
- To submit part A, which you will have done with a partner, change into `bare/impcore-with-locals` and run `submit105-opsem-pair` to submit your work.

How your work will be evaluated

Adding local variables to Impcore

	Exemplary	Satisfactory	Must improve
Locals	• Change to interpreter appears motivated either by changing the semantics as little as possible or by changing the code as little as possible. • Local variables for Impcore pass simple tests.	• Course staff believe they can see motivation for changes to interpreter, but more changes were made than necessary. • Local variables for Impcore pass some tests.	• Course staff cannot understand what ideas were used to change the interpreter. • Local variables for Impcore pass few or no tests.
Form	• Code taken from the book is either totally unchanged or is changed in a meaningful way. • All code, including code from the book, fits in 80 columns. • All code, except possibly machine-altered code from the book, respects the offside rule • Indentation is consistent everywhere. • No code is commented out. • Solution file contains no distracting test cases or print statements.	• One or two lines are wider than 80 columns. • The code contains one or two violations of the offside rule • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • Solution file may contain clearly marked test *functions*, but they are never executed. It's easy to read the code without having to look at the test functions.	• Code taken from the book has been changed cosmetically (e.g., by changing line breaks or indentation, or by changing names) without changing the code's function. • Three or more lines are wider than 80 columns. • The code contains three or more violations of the offside rule • The code is not indented consistently. • Solution file contains code that has been commented out. • Solution file contains test cases that are run when loaded. • When loaded, solution file prints test results.
Naming	• Where the code implements math, the names of each variable in the code is either the same as what's in the math (e.g., `rho` for *Ï–*), or is	• Where the code implements math, the names don't help the course staff figure out how the	• Where the code implements math, the course staff cannot figure out how the code

	an English equivalent for what the code stands for (e.g., `parameters` or `parms` for *Ï–*).	code corresponds to the math.	corresponds to the math.
Structure	• The code is so clear that course staff can instantly tell whether it is correct or incorrect. • There's only as much code as is needed to do the job. • The code contains no redundant case analysis.	• Course staff have to work to tell whether the code is correct or incorrect. • There's somewhat more code than is needed to do the job. • The code contains a little redundant case analysis.	• From reading the code, course staff cannot tell whether it is correct or incorrect. • From reading the code, course staff cannot easily tell what it is doing. • There's about twice as much code as is needed to do the job. • A significant fraction of the case analyses in the code, maybe a third, are redundant.

Operational semantics

	Exemplary	Satisfactory	Must improve
Semantics	• The program which is supposed to behave differently in Awk, Icon, and Impcore semantics behaves exactly as specified with each semantics.	• The program which is supposed to behave differently in Awk, Icon, and Impcore semantics behaves almost exactly as specified with each semantics.	• The program which is supposed to behave differently in Awk, Icon, and Impcore semantics gets one or more semantics wrong. • The program which is supposed to behave differently in Awk, Icon, and Impcore semantics looks like it is probably correct, but it does not meet the specification: either running the code does not print, or it prints two or more times.
Rules	• Every inference rule has a single conclusion which is a judgment form of the operational semantics. • In every inference rule, every premise is either a judgment form of the operational semantics or a simple mathematical predicate such as equality or set membership. • In every inference rule, if two states, two environments, or two of any other thing *must* be the same, then they are notated using a *single* metavariable that appears in multiple places. (Example: *Ï–* or *Ï•*)	• In every inference rule, two states, two environments, or two of any other thing *must* be the same, yet they are notated using different metavariables. However, the inference rule includes a premise that these metavariables are equal. (Example: $Ï–_1$â— =â— $Ï–_2$) • A new language design has a few too many new or changes a few too many existing rules. • Or, a new language design is missing a few rules that are needed, or it doesn't change a few existing rules that need to be changed.	• Notation that is presented as an inference rule has more than one judgment form or other predicate below the line. • Inference rules contain notation above the line that does not resemble a judgment form and is not a simple mathematical predicate. • Inference rules contain notation, either above or below the line, that resembles a judgment form but is not actually a

	• In every inference rule, if two states, two environments, or two of any other thing *may* be different, then they are notated using different metavariables. (Example: $\ddot{I}$– and $\ddot{I}$–$\hat{E}^1$) • New language designs use or change just enough rules to do the job. • Inference rules use one judgment form per syntactic category.		judgment form. • In every inference rule, two states, two environments, or two of any other thing *must* be the same, yet they are notated using different metavariables, and nothing in the rule forces these metavariables to be equal. (Example: $\ddot{I}$– and $\ddot{I}$–$\hat{E}^1$ are both used, yet they must be identical.) • In some inference rule, two states, two environments, or two other things *may* be different, but they are notated using a single metavariable. (Example: using $\ddot{I}$– everywhere, but in some places, $\ddot{I}$–$\hat{E}^1$ is needed.) • In a new language design, the number of new or changed rules is a lot different from what is needed. • Inference rules contain a mix of judgment forms even when describing the semantics of a single syntactic category.
Derivations	• In every derivation, every utterance is either a judgment form of the operational semantics or a simple mathematical predicate such as equality or set membership. • In every derivation, every judgement follows from instantiating a rule from the operational semantics. (Instantiating means substituting for meta variables.) The judgement appears below a horizontal line, and above *that* line is one derivation of each premise. • In every derivation, equal environments are notated equally. In a derivation, $\ddot{I}$– and $\ddot{I}$–$\hat{E}^1$ *must* refer	• In one or more derivations, there are a few horizontal lines that appear to be instances of inference rules, but the instantiations are not valid. (Example: rule requires two environments to be the same, but in the derivation they are different.) • In a derivation, the semantics requires new bindings to be added to some environments, and the derivation contains environments extended with the right new bindings, but not in exactly the right places.	• In one or more derivations, there are horizontal lines that the course staff is unable to relate to any inference rule. • In one or more derivations, there are many horizontal lines that appear to be instances of inference rules, but the instantiations are not valid. • A derivation is called for, but course staff cannot identify the tree structure of the judgments forming the derivation.

	to different environments. • Every derivation takes the form of a tree. The root of the tree, which is written at the bottom, is the judgment that is derived (proved). • In every derivation, new bindings are added to an environment exactly as and when required by the semantics.		• In a derivation, the semantics requires new bindings to be added to some environments, and the derivation does not contain any environments extended with new bindings, but the new bindings in the derivation are not the bindings required by the semantics. (Example: the semantics calls for a binding of *answer* to 42, but instead *answer* is bound to 0.) • In a derivation, the semantics requires new bindings to be added to some environments, but the derivation does not contain any environments extended with new bindings.
Metatheory	• Metatheoretic proofs operate by structural induction on derivations, and derivations are named. • Metatheoretic proofs classify derivations by case analysis over the final rule in each derivation. The case analysis includes every possible derivation, and cases with similar proofs are grouped together.	• Metatheoretic proofs operate by structural induction on derivations, but derivations and subderivations are not named, so course staff may not be certain of what's being claimed. • Metatheoretic proofs classify derivations by case analysis over the final rule in each derivation. The case analysis includes every possible derivation, but the grouping of the cases does not bring together cases with similar proofs.	• Metatheoretic proofs don't use structural induction on derivations (**serious fault**). • Metatheoretic proofs have incomplete case analyses of derivations. • Metatheoretic proofs are missing many cases (**serious fault**). • Course staff cannot figure out how metatheoretic proof is broken down by cases (**serious fault**).

Example cases for Exercise 22

Here are some sample cases for the inductive proof required in Exercise 22 on page 64 of Ramsey's book.

Consider the rule for `if`:

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle}{\langle \textsc{if}(e_1, e_2, e_3), \xi, \phi, \rho\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle.} \qquad (\textsc{IfTrue})$$

By the induction hypothesis, we can evaluate $\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle$ using a stack, and the evaluation will pop ρ and push ρ' without making a copy of ρ. Because ρ does not appear anywhere else in the rule, it is never used again, so it is safe to pop it and throw it away. We can use the induction hypothesis again to show that the evaluation of e_2 can pop ρ' and push ρ'', and ρ' is not copied. Moreover, ρ' is not used in the rule after the evaluation of e_2.

Finally, we see that ρ'' is used only as part of the result of the rule. We can conclude, then, that when e_1 evaluates to a nonzero value, we can safely evaluate $\textsc{if}(e_1, e_2, e_3)$ on a stack, and the evaluation effectively pops ρ, which is never used again, then pushes ρ''.

The FormalVar rule is one of the base cases; it doesn't require the induction hypothesis.

$$\frac{x \in \operatorname{dom} \rho}{\langle \textsc{var}(x), \xi, \phi, \rho\rangle \Downarrow \langle \rho(x), \xi, \phi, \rho\rangle} \qquad (\textsc{FormalVar})$$

By examining the rule, we see that it is possible to implement it as follows: pop ρ, test $x \in \operatorname{dom} \rho$, and compute $\rho(x)$. Then push ρ back on the environment stack, after which the only copy is once again on top of the stack.