Assignment: Operational Semantics

COMP 105

Due Tuesday, February 7, 2017 at 11:59PM

Contents

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 a successful, *terminating* 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, I 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.

Your questions answered

The questions you asked in class are answered online[1].

Part A: Reading Comprehension (individual work, 10 percent)

For questions 1–7, please read pages 17–26 (the book sections on environments and on operational semantics of expressions).

1. What is ξ?
2. What is ϕ?
3. What is ρ?
4. In operational semantics, what kind of a thing does e stand for?
5. In operational semantics, what kind of a thing does v stand for?
6. Examine the evaluation rules for all the syntactic forms of Impcore, and list all the forms to which more than one rule can apply. ("Syntactic form" is in the index *twice*; you will probably find both pages helpful.)

   Example answer: more than one rule applies to the VAR(x) form.

7. Pick three of the syntactic forms from the previous question. For each one, explain briefly in your own words *why* that form has more than one evaluation rule.

   Example answer: there are two rules for the VAR form because a name can be either a formal parameter or a global variable.

For question 8, please skim section 1.5, and carefully read section 1.5.2 on pages 43–53.

8. Pick one of the syntactic forms from the previous question. Find the place in the interpreter where the form is implemented, and answer the following two questions:

   (a) How does the interpreter decide which rule applies? Identify both the math and the code involved in the decision.

   (b) If neither rule applies, how does the interpreter know, and what does it do?

   Example answer: The VAR(x) form is evaluated in chunk 45a at the top of page 45. It decides which rule applies by checking the math $x \in dom\, \rho$—the code is `isvalbound(e->u.var, formals)`. If the answer is yes, rule FormalVar applies. If the answer is no, the only possible rule is GlobalVar.

   If neither rule applies, it is because $x \notin dom\, \rho$ and also $x \notin dom\, \xi$. In that case, no rule's conditions are satisfied. The interpreter knows by checking `isvalbound(e->u.var, formals)` and then if that fails, checking `isvalbound(e->u.var, globals)`. If both conditions fail, no rule applies, and the interpreter triggers a run-time error (by calling `runerror`).

For questions 9 and 10, please read section 1.1.5 (page 15) on the difference between primitive functions and predefined functions, and please study the rules for function application on pages 24 to 26.

[1] http://www.cs.tufts.edu/comp/105/handouts/opsem-q+a.pdf

9. Function `<=` is *predefined*. When evaluating `(<= 0 n)`, what rule of the operational semantics is used at the root of the derivation?

10. Function `<` is *primitive*. When evaluating `(< n 10)`, what rule of the operational semantics is used at the root of the derivation?

Part B: Adding Local Variables to the Interpreter (Work with a partner, 23 percent)

Related reading: Section 1.5, particularly 1.5.2 on pages 43-53. These pages will walk you through how the impcore interpreter is implemented.

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 81 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.*

For information on pair programming, consult the syllabus[2], the reading[3], and some timeless advice for pair programmers[4].

- 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`, `definition-code.c`, `parse.c`, `printfuns.c`, and `tableparsing.c` that handle local variables. These versions 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.~~ The interpreter you build will parse definitions containing local variables, but it will ignore them. To get local variables working, you'll need to make changes:

  – 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.

[2] ../syllabus.html#how-do-pair-programming-interactions-work
[3] ../readings/pairs.pdf
[4] http://www.cs.tufts.edu/comp/40-2011f/readings/other-pair.html

You may find the function `mkVL` in file `list-code.c` helpful.

- Create a file called `README` in your `impcore-with-locals` directory. Describe your solution in the README.

Part C: Operational semantics, derivations, and metatheory (Individual work, 67 percent)

Related reading:

- For an example of a derivation tree, see page 58.
- For rules of operational semantics, see pages 77–78.
- For metatheory, see section 1.6.2 on pages 58–59.

These exercises are 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.

- Do Exercise 13 on page 78 of Ramsey's book. The purpose of the exercise is to 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. 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.
- Do Exercise 14 on page 78 of Ramsey's book. Now that you know how to *write* a derivation, in this exercise you start *reasoning* about derivations.
- Do Exercises 20 and 21 on page 79 of Ramsey's book.

  Exercise 21 is a little unclear. The "program" referred to in the exercise is just a sequence of definitions. Create a file `21.imp`, which should contain a sequence of definitions with these properties:

  (i) Every definition in the sequence is *syntactically* valid Impcore.

  (ii) If you present the sequence of definitions to a standard Impcore interpreter, the result is a checked run-time error.

  (iii) If you present the sequence of definitions to an Impcore interpreter that has been extended with the Awk-like semantics from exercise 20, the last thing the interpreter does is print `1`.

  (iv) If you present the sequence of definitions to an Impcore interpreter that has been extended with the Icon-like semantics from exercise 20, the last thing the interpreter does is print `0`.

  NOTES: This is an exercise in language design. The main purpose of the exercise is to give you a feel for the kinds of choices language designers can make. But you must also be able to think about the consequences of language-design choices *before an implementation of the language has been built*. You do have an implementation that can verify properties (i) and (ii): it's the standard Impcore interpreter. But you don't have an Awk-like interpreter or an Icon-like interpreter, so you have no implementation that you can use to verity properties (iii) and (iv). Your choices are to think carefully about the semantics you have designed and the program you have written—or to

build two more interpreters, so that you can actually test your code. Thinking carefully is the sane choice.

- Do Exercise 19 on page 78 of Ramsey's book. In this exercise you prove that given a set of environments, the result of evaluating any expression e is completely determined. That is, in any given starting state, evaluation produces the same results every time. This proof requires you to raise your game again, reasoning about the *set* of all *valid* derivations. It's metatheory. 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.

  The way to tackle this problem is to assume you have two valid derivations same e and the same environments on the left, but different v's on the right—let's call them v_1 and v_2. You then prove that if both derivations are valid, $v_1 = v_2$.

  Metatheoretic proofs are probably unfamiliar, but you will have a crack at them in lecture and in recitation. Also, to relieve some of the tedium (which is very common in programming-language proofs), we decree that all programs are written in Theoretical Impcore, which is a restricted subset of Impcore:

  – There are no `while` or `begin` expressions.
  – Every function application has exactly two arguments.
  – The only primitive function is `+`.

  Using Theoretical Impcore reduces the number of cases to a manageable number.

Organizing the answers to Part ~~B~~ C

For these exercises you will turn in two files: `theory.pdf` and `21.imp`. For file `theory.pdf`, you could consider using LaTeX, but unless you already have experience using LaTeX to typeset mathematics, it's a bad idea. We recommend that you write your theory homework by hand, then scan or photograph it[5]. Either way, make sure that your PDF can be opened on a Halligan computer.

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 theory answers *must appear in order*: Exercises 13, 14, 19, and finally 20.

Your answer to exercise 21 should be in file `21.imp`.

Extra credit: Eliminating `begin`

Theoretical Impcore has neither `while` nor `begin`. You already have an idea that you can often replace `while` with recursion. For extra credit, you can show that you can replace `begin` with function calls.

Assume that ϕ binds the function `second` according to the following definition:

```
(define second (x y) y)
```

[5] http://www.cs.tufts.edu/comp/105/syllabus.html#then-how-should-theory-homework-be-written

I claim that if e_1 and e_2 are arbitrary expressions, you can always write `(second` e_1 e_2`)` instead of `(begin` e_1 e_2`)`. For extra credit, answer any or all of the following questions:

- **X1.** Using evaluation judgments, take the claim "you can always write `(second` e_1 e_2`)` instead of `(begin` e_1 e_2`)`" and restate the claim in precise, formal language.

  Hint: The claim is related to the claims in exercises 14 and 15 on page page 78 in the Impcore chapter.

- **X2.** Using operational semantics, prove the claim.

- **X3.** Define a translation for `(begin` $e_1 \cdots e_n$`)` such that the translated code behaves exactly the same as the original code, but in the result of the translation, every remaining `begin` has exactly two subexpressions. For example, you might translate

  ```
  (begin e1 e2 e3)
  ```

  into

  ```
  (begin e1 (begin e2 e3))
  ```

  You may use any notation you like, but the cleanest way to define the translation is by using algebraic laws.

Once you've defined the translation in step X3, you'll be ready to write a translation that eliminates `begin` entirely. But that translation is more appropriate to next week's homework.

How to organize and submit your work

Before submitting code, **test what you can**. We do not provide any tests; you write your own. All code can be fully tested except the code for exercise 21.

- To complete part A, which you do by yourself, download the questions[6], and edit the answers into the file `cqs.opsem.txt`. If your editor is not good with Greek letters, you can spell out their names: ξ is "xi," ϕ is "phi," and ρ is "rho." Keep file `cqs.opsem.txt` with your solutions to part C.

- To submit part B, which you will have done with a partner, `cd` into `bare/impcore-with-locals`. The directory should contain your code and a `README` file that documents your solution.

  As soon as you have these files, run `submit105-opsem-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

- To complete part C, which you do by yourself, create files `21.imp` and `theory.pdf`. Please also create a file called `README`, in which you tell us anything else you think is useful for us to know.

  We provide a template for your README at http://www.cs.tufts.edu/comp/105/homework/opsem-README-template

  As soon as you have the files for parts A and C, which you will have done by yourself, `cd` into the appropriate directory and run `submit105-opsem-solo` to submit a preliminary version of your

[6] ./cqs.opsem.txt

work. You'll need files `README`, `cqs.opsem.txt`, `21.imp`, and `theory.pdf`, but preliminary versions are good enough. Keep submitting and resubmitting until your work is complete; we grade only the last submission.

How your work will be evaluated

Adding local variables to Impcore

Everything in the general coding rubric[7] applies, but we will focus on three areas specific to this exercise:

	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.
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 ρ), or is an English equivalent for what the code stands for (e.g., `parameters` or `parms` for ρ).	• Where the code implements math, the names don't help the course staff figure out how the code corresponds to the math.	• Where the code implements math, the course staff cannot figure out how the code corresponds to the math.

[7] ../coding-rubric.html

	Exemplary	Satisfactory	Must Improve
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

Here is an extensive list of criteria for judging semantics, rules, derivations, and metatheoretic proofs. As always, you are aiming for the left-hand column, you might be willing to settle for the middle column, and you want to avoid the right-hand column.

Changed rules of Impcore (exercise 20)

	Exemplary	Satisfactory	Must Improve
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: ρ or σ) • 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: ρ and ρ') • New language designs use or change just enough rules to do the job. • Inference rules use one judgment form per syntactic category.	• 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: $\rho_1 = \rho_2$) • 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 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: ρ and ρ' 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 ρ everywhere, but in some places, ρ' 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.

Exemplary	Satisfactory	Must Improve

Program to probe Impcore/Awk/Icon semantics (exercise 21)

	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.

Derivations (exercises 13 and 14)

	Exemplary	**Satisfactory**	**Must Improve**
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, ρ and ρ' *must* refer 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 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. ● 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 (exercise 19)

	Exemplary	**Satisfactory**	**Must Improve**
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**).