COMP105 Assignment: Intro

Deadline Extended: Due Monday, January 31 at 11:59PM.

Getting Started

- To add the course binaries to your execution path, run

  ```
  use -q comp105
  ```

  You may want to put this command in your `.cshrc` or your `.profile`. The `-q` option is needed to prevent `use` from spraying text onto standard output, which may interfere with with `scp`, `ssh`, `git`, and `rsync`.
- **IMPORTANT NOTE**: Like all programming assignments for this class, the programming parts of this assignment are due one minute before midnight before a class day. You may turn them in *up to 48 hours after the due date*, which will cost you one or two extension tokens. If you wish not to spend an extension token, then when midnight arrives submit whatever you have. We are very willing to give partial credit.

Part A: Programming in Impcore (Individual work, 25 percent)

These are ``finger exercises'' to get you into the swing of the LISP syntax and style of programming. *You can start these exercises immediately after the first lecture*. If you find it entertaining, you may write very efficient solutions—but do not feel compelled to do so. **Do not share your solutions with any programming partners.** We encourage you to discuss ideas, but **noone else may see your code**.

- Do **exercises 2 through 7** on pages 51–52 of Ramsey and Kamin. Place your solutions to problems 2 through 7 in a file called `solution.imp`.

  You can find an `impcore` interpreter in `/comp/105/bin`; if you have run `use` as suggested above you should be able to run it just by typing

  ```
  ledit impcore
  ```

  The `ledit` command gives you the ability to retrieve and edit previous lines of input; see its man page.

  Note that you can run the contents of a file through the interpreter by typing `impcore < file`. You can eliminate unwanted prompts by running `impcore -q < file`. You may find it useful to create some test cases in a file `mytests`; you can then check your work by typing

  ```
  cat solution.imp mytests | impcore -q
  ```

  Don't include test cases in the `solution.imp` file you submit.

  You may find it more convenient to keep solutions in separate files as you develop them. If so, we recommend that you do so and combine them in the end with `cat`. For example,

  ```
  cat s2 s3 s4 s5 s6 s7 > solution.imp
  ```
- In doing problems 2 through 7, use helper functions where appropriate, but *do not use global variables*.
- The solutions you write for problems 2 through 7 should be in order in the file `solution.imp` (i.e. problem 2 first, problem 7 last) and each solution should be preceeded by a comment that looks like something like this:

  ```
  ;;
  ;; Problem N
  ;;
  ```

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

This exercise will help you understand how language changes can be realized in C code. You will do exercise 24 from page 55 of Ramsey and Kamin. *Before starting this problem, you should wait for the second lecture and possibly the third lecture*. **You should 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/book-code
  ```

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

  ```
  git clone /comp/105/book-code
  ```

  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`, `toplevel-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 and 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 C: Language structure and operational semantics (Individual work, 50 percent)

These are exercises intended to help you think about syntactic structure and to become fluent with operational semantics. *The third and fourth lecture, on operational semantics and proofs, will help with some of these exercises*, especially Exercise 18. **Do not share your solutions with any programming partners.** We encourage you to discuss ideas, but **noone else may see your code**.

For these exercises you will turn in three files: `1.pdf` (or `1.txt`), `theory.pdf`, and `13.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 Exercise 1 on page 51 of Ramsey and Kamin.
  I'm looking for several paragraphs—at most one page—about a language of your choice. *Give us an argument* about why you think some syntactic categories are essential for understanding and other are inessential. Also, I'd like your answer to provide a little more information about the essential syntactic categories:

- ♦ In Impcore, expressions are evaluated to produce values, and definitions are evaluated to add names to the top-level environments. In your chosen language, what happens to each of the syntactic categories?

Suggestions:

- ♦ Find a reference work that gives a *grammar* for the language (its concrete syntax) you have chosen.
- ♦ Unless the only language you know is C++, pick another language. C++ is frightfully complex.

Please prepare either a plain text file (Markdown formatting is OK) as file `1.txt`, or else use a word processor and submit `1.pdf`. **Do not go over one page!**

- Do exercises 12 and 13 on page 53 of Ramsey and Kamin. The purpose of these exercises is to give you a feel for the kinds of choices language designers can make. Include your answer to exercise 12 as part of `theory.pdf`. Please your answer to exercise 13 in file `13.imp`.
- Do exercise 9 on page 52 of Ramsey and Kamin. The purpose of the exercise is to help you develop your understanding of proof trees, so be sure to make your proof *complete* and *formal*. You can write out a proof tree with `BEGIN ... => 3 ...` in its conclusion, or if you prefer, you can write a sequence of judgments, and say for each one what rules and what previous judgments justify that judgment.

  Include your answer as part of file `theory.pdf`.
- Do exercise 10 on page 52 of Ramsey and Kamin. The purpose of the exercise is to help you start reasoning about proof trees.

  Include your answer as part of file `theory.pdf`.
- Do exercise 18 on pages 53–54 of Ramsey and Kamin. The purpose of this exercise is to help you see how language designers show nontrivial properties of their languages—and how these properties can guide implementors.

  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 xi.

Organizing the answers to Part C

To help us read your answers to Part~C, 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**: problems 9, 10, 12, and finally 18.

Submitting

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

- To submit parts A and C, which you will have done by yourself, change into the appropriate directory and run `submit105-intro-solo` to submit your work. In addition to files `solution.imp`, `13.imp`, `theory.pdf`, and either `1.txt` or `1.pdf`, please also include a file called `README`. Use your `README` file to
  - ♦ Tell us how to pronounce your name, as in "NORE-muhn RAM-zee" or "ANN-drew guh-LAHNT"
  - ♦ Tell us *how long it took you to complete the entire assignment* (parts A, B, and C)
- To submit part B, which you will have done with a partner, change into `bare/impcore-with-locals` and run `submit105-intro-pair` to submit your work.

Example cases for problem 18

Here are some sample cases for the inductive proof required in Exercise 18 of Ramsey and Kamin.

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.