CS152 Assignment: Intro

Due Tuesday, February 8 at 11:59PM.

Getting Started

- Edit your `~/.cshrc` file and add `~cs152/bin` to your path. To do this, you will probably just need to add the line `setenv PREPATH ~cs152/bin` to the top of your `~/.cshrc` file, just after the sample line provided; if you already have a similar line, then you will need to add `~cs152/bin` to the list of directories it already contains. If your `PREPATH` line is to contain multiple directories, they should be separated by colons but *not* spaces. Once you have updated your `~/.cshrc` file either log out and back in again or update your environment by typing `source ~/.cshrc`

  If you are bold enough to use the Korn shell, we applaud your choice and assume you can manage `PATH=~cs152/bin:$PATH`.
- **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 late days. If you wish not to spend a late day, then when midnight arrives submit whatever you have. We are very willing to give partial credit.
- The theory parts of the assignment are due on paper at 5:00PM on Wednesday, February 9. They can be left in the box labeled ``CS 152'' outside Maxwell Dworkin 133.

Part A: Programming in Impcore (25 points)

These are ``finger exercises'' to get you into the swing of the LISP syntax and style of programming. If you find it entertaining, you may write very efficient solutions---but do not feel compelled to do so.

- Create a directory called `cs152` in your home directory; inside that, create a directory called `intro`. Go to that directory.
- Do **exercises 1 through 6** on pages 47–48 of Ramsey and Kamin. Place your solutions to problems 1 through 6 in a file called `solution`. You can find an `impcore` interpreter in `~cs152/bin`; if you have updated your path as above you should be able to run it just by typing `impcore`. 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 mytests | impcore -q
  ```

  Don't include test cases in the `solution` 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`, and that you use Make to manage the combination. For example, you could try something like this in a Makefile:*
  >
  > ```
  > COMPONENTS=s1 s2 s3 s4 s5 s6
  >
  > solution: $(COMPONENTS)
  >           cat $(COMPONENTS) > solution
  > ```

```
submit: solution
        submit-intro

test: solution mytests
      cat solution mytests | impcore -q
```

(Be aware that we have not tested such a Makefile.)

- In doing problems 1 through 6, use helper functions where appropriate, but *do not use global variables*.
- The solutions you write for problems 1 through 6 should be in order in the file `solution` (i.e. problem 1 first, problem 6 last) and each solution should be preceeded by a comment that looks like something like this:

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

- When you get everything working, make sure your current directory is the directory you are submitting; then type `submit-intro` to submit your work.

Part B: Working with operational semantics (50 points)

These are exercises intended to help you become fluent with operational semantics.

- Do exercise 8 on page 49 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.

  Turn in your answer on paper.
- Do exercises 9 and 10 on page 49 of Ramsey and Kamin. The purpose of these exercises is to give you a feel for the kinds of choices language designers can make. Turn in your answer to exercise 9 on paper; turn in exercise 10 as file `solution10`.
- Do exercise 15 on pages 49––50 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.

  To make things simpler, don't worry about the global environment xi, and pick just one primitive rule. You may also want to look at some sample cases we have provided to help you.

  Turn in your answer on paper.

Part C: Adding Local Variables to the Interpreter (25 points)

This exercise will help you understand how language changes can be realized in C code. You will do exercise 21 from page 51 of Ramsey and Kamin.

- You can find the source code from Chapter 2 in `~cs152/software/bare/impcore` or `~cs152/software/commented/impcore` (or you can download all the software from the book). 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.
- Whichever you pick, copy our code into a subdirectory (`impcore`) of your submission directory.
- We provide new versions of `all.h`, `ast.c`, `toplevel-code.c` and `parse.c` that handle local variables. We provide these in the form of `diffs', i.e., output from `diff -c`. You can copy the diffs from `~cs152/software/exercises/impcore-local-diffs` or get them online. Once you have them, you apply the diffs by writing, e.g.,

```
patch <impcore-local-diffs
```

- 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.
- Create a file called `README` in this directory (your `impcore` directory). Use this file to describe your solution for this problem, tell us how to pronounce your name, and *indicate how long it took you to complete the entire assignment* (parts A, B, and C).
- Test your code throughly before submitting it. We do not provide you with any tests; you must write your own.
- If you like, you can submit up to three test cases in file `test-transcript`. They should be in the same format as the transcripts in the book, e.g.,

```
-> (define gcd (n m) (r)
      ... more code for gcd ...
   )
-> (gcd 48 30)
6
```

For every test case you submit, and for every interpreter another student submits that your test cases exposes a bug in, we will give you a point. For every test case that exposes a bug in the course software, we will give you at least 5 points (maybe more, depending on our judgment of the severity of the bug).

Submitting

- When you get everything working, cd into the directory you are submitting and type `submit-intro` to submit your work.