PLC: Homework 6 [75 points]

Due date: Wednesday, March 8, 10:30pm
3 point bonus if you turn in by Tuesday, March 7th, 10:30pm

About This Homework

The goal of this homework is to explore static scoping of variables by implementing a tool operating on programs in the Lettuce language, which we are looking at in class Thursday, March 2nd. The assignment also makes use of tries (`trie.agda` in the IAL), which we are discussing in class March 2nd.

As usual, the first thing you should do for this homework is copy the files from the `hw/hw6` subdirectory of the class repository, to a new `hw6` subdirectory of your personal repository, similarly to previous homeworks. You will just be modifying two of these files, namely `html.agda` and `html-thms.agda`, but we will refer to others of them in the instructions below.

There is extra credit possible for this assignment, so you can earn over 75 points (but 75 is a perfect score).

Partners Allowed

For this homework, you may work by yourself or with one partner (no more). See the instructions for `hw2` for how to create the `ack.txt` and `partner.txt` files that are required if you work with a partner

All files should go in a hw6 subdirectory (which you create) of your personal repository. Please call the subdirectory exactly that. Do not call it `Homework 4` or `Hw6` or other variations, or we will penalize you 5 points.

How to Turn In Your Solution

Make sure you have done a "Subversion add" on the `hw6` directory you created in your personal repository, and on the `.agda` files you have to submit for this assignment, and then do a "Subversion commit" on your personal repository directory to submit them. You can commit as many times as you want up to the deadline. If you commit after the deadline, we will use the last existing version before the deadline. Work submitted after the deadline will not count.

How to Check Your Solution

As for previous homeworks, you might want to make sure you have correctly submitted your solution via Subversion. Again, just go to the URL for your personal repository using a web browser, log in with your HawkId and password, and you can see what has been submitted.

Make sure each of the files you submit can be checked by Agda without any holes, yellow highlighting, or red highlighting. If any file does not check this way, we will penalize you 5 points total (if multiple files do not check, we will still just penalize you 5 points). Any problems you do not solve should simply be removed from the `.agda` files, so that there are no holes (question marks) in the files.

How To Get Help

You can post questions in the `hw6` section on Piazza, or elsewhere on Piazza. See the course's Google Calendar, linked from Piazza, for the locations and times for office hours, including evening Skype office hours for Prof. Stump.

1 Reading

Please read Sections 5.2, 5.4, and 5.5 of the book.

2 Preliminaries [25 points]

1. Change the definition of `vars-to-string` in `mainl.agda` so that it converts the given list of variables (type `var`, which is just a synonym for `string`) to a string where each variable occurs only once, and variables are separated by a space. Hint: this problem is very easy if you use a trie to collect all the variables and then just extract the strings from the trie with `trie-strings`. [10 points]
2. Modify the definition of `process-start` so that it checks if `g` declares the same variable twice. If it does, then `process-start` should return a string saying which variable is being declared more than once. Hint: you can write a helper function which can keep track of the variables declared, and return `just x` if you see a variable `x` which has already been declared, and `nothing` otherwise (here I am referring to the `maybe` type, in `maybe.agda`). [15 points]

3 Processing Let Expressions [55 points]

This problem is concerned with computing information about let-expressions. Sample input files ending in `.lett` can be used as test cases. Your tool will process the syntax trees for these expressions, to carry out a variety of computations. The syntax trees are defined in `lettuce-types.agda`. You can see what a syntax tree looks like by first compiling `mainl.agda` (as for the previous problem), and then running the executable on a test file, with the `--showParsed` command-line argument. So on Windows (or use `./mainl` instead of `mainl` on Mac/Linux):

```
mainl --showParsed test2.lett
```

The goal of this `lettuce` tool is to print some information about the let-expression given in the input file.

You should modify the `process-letterm` function in `mainl.agda` to generate a string which contains the following components in order. If you do not implement one of the parts below, that is fine. Just do not print anything out for that part. Please see below (after the parts are described) for an important note about testing your solution.

- `declared:` *vars*. Compute the list of variables that are declared either globally or by a `let` in the input, and print out those variables, separated by spaces. The order in which you print them out does not matter. So for `test0.lett`, your output would look like

  ```
  declared: x
  ```

  Use your `vars-to-string` function defined in the first problem to avoid printing the same variable twice. [15 points]

- `used:` *vars*. Compute the list of variables that are used somewhere in the input (regardless of whether or not they are declared with a `let`). So for `test0b.lett`, your code should print out:

  ```
  used: y z
  ```

  The variable `x` is declared but not used in `test0b.lett`, so it is not included. Again, avoid printing the same variable name twice.

  [15 points]

- `defs:` *D*, where *D* is a space-separated list of what we can call *definition descriptors*: they either look like

  $$var@posinfo \text{ local}(posinfo)$$

  or else

  $$var@posinfo \text{ global(is-declared)}$$

  Here, `is-declared` should be either `true` or `false`, depending on whether the variable has a global definition at the start of the file or not.

  For each use of a variable in the let-expression, you are printing whether it is local (and in that case, what is the position in the file where that variable is introduced by a `let`) or global. The `posinfo` values are numbers that you will find inside the syntax trees (defined in `lettuce-types.agda`). One approach to this problem would be to use a trie to keep track of the position (`posinfo`) for the deepest let-expression that defines each variable. You would initialize this trie by processing any global definitions, and recording that they are global (no `posinfo` needed in that case, actually). Then as you recurse through the let-expression, you would update this trie. When you see a `let`, you would record the `posinfo` for the declared variable. Then when you find a use of a variable, you will be able to check whether it is declared in the trie, and if so, whether it is local or global. Variables that are not even declared in the trie have `is-declared` as `false`.

[25 points]

If you are having trouble compiling on your computer: you can use the interpreter (Control-c Control-n) to run the `processText` function with a boolean telling whether or not you want to see the parse tree for the input string, and then the input string (which would be the contents of one of the `.lett` test files).

Testing. To test your `mainl` executable, you can run it on the `.lett` files I am including with the homework. I am providing a separate tool called `shooter` (like salad shooter!) which checks the format of your output. You just save the output from your tool to a file, and then run `shooter` on that file. You compile `shooter` similarly to `mainl`, using Control-c Control-x Control-c in `shooter.agda`.

If your output cannot be parsed in by `shooter`, then we will penalize you three points. So please make sure your output can be parsed by `shooter`. Your output is parsed successfully if you run it through `shooter` and do not see any output (produced by `shooter`). Otherwise, you will see a parse error from `shooter`. This only applies for cases where you do not report an error message for duplicate global variables. Note that shooter expects at least one newline at the end of your output (corner case: unless you have no output at all, but then you do not need to run shooter!).