PLC: Homework 5 [100 points/115 possible]

Due date: Friday, April 17th, 9pm
Late deadline: Monday, April 20th, 9pm, for a 10% penalty

About This Homework

In this homework, you will implement a compiler for a simple untyped functional language called Liskell ("Lisp + Haskell"). Your compiler will take in a Liskell program, and produce C output, which you will then need to compile with a C compiler.

As for the previous two homework, a perfect score is 100 points, but it is possible to earn up to 15 extra credit points (the max possible score is capped at 115). You can select parts of problems to solve or not.

As usual, please create a `hw5` subdirectory in your personal repository, and add it with "Subversion Add". You will submit your solutions in this directory.

Installing a C compiler

The `gcc` compiler comes with all Linux distributions, so there is nothing to install for Linux users. For Mac, it seems like it can be downloaded as part of Xcode. For Windows, MinGW (`www.mingw.org`) provides `gcc` for Windows. Instead of installing `gcc`, you can just use `gcc` on the CS department's linux machines (so you would have to copy over a .c file from your computer to, say, `linux.cs.uiowa.edu`, and then run `gcc` there).

Partners Allowed

For this homework, as for previous ones, you may work by yourself or with one partner (no more). The procedure for this is the same as for previous homeworks; the instructions for Homework 2 tell you what you need to do to submit your assignment with a partner.

How to Turn In Your Solution

Make sure you have done a "Subversion add" on the `hw5` directory you created in your personal repository, and on all the files you are turning in. Then do a "Subversion commit" on your personal repository directory to submit them. If you are working with a partner, one of you submits these files and the `ack_partner.txt` file, and the other submits just `partner.txt`.

How to Check Your Solution

The instructions here are similar to those for Homework 4. No file needs to check in safe mode with Agda, because we are not proving theorems for this assignment.

How To Get Help

You can post questions in the `hw5` section on Piazza, or elsewhere on Piazza. See the course's Google Calendar, linked from Piazza, for the locations and times for office hours (we have almost 10 hours a week of office hours).

You are free to borrow from any grammars we have done in class (in the `lectures` subdirectory of the course repo).

The Liskell Language

For this assignment, I have invented a small untyped functional programming language called Liskell. It combines the essential pair data structure of LISP, with pattern-matching recursive equations as in Haskell (and Agda). There are only two data structures in Liskell:

- pairs, denoted `[x , y]` for the pair consisting of `x` and `y`; and
- strings, like `"hi there"`.

Each Liskell input program is a single file, including the definition of a function called `main`. This function will be called with a list of strings for the command-line arguments given to the program. The resulting data structure computed by `main` will then be printed out for the user, when the program exits.

Liskell programs define functions recursively in **blocks**. Each block is a list of equations, where the left-hand side is a pattern, and the right-hand side is a term. Terms are:

- names (either pattern variables or defined symbols), which must start with an alphabetic character (either upper or lowercase), and then continue with alphanumeric characters (alphabetic or numeric)
- a pair of terms, denoted `[x,y]` with optional whitespace throughout
- a string literal, consisting of a sequence of characters (maybe none) between double quotation marks, where the characters in question are alphanumeric, comma, period, space, backslash, or escaped double quotation mark (`\"`), and then a double quotation mark.
- a parenthesized term, with whitespace throughout
- an application of one term to another, with whitespace in between those two terms

Patterns are like terms except that parenthesized terms and applications are not allowed. Also, if a pair is used as a pattern, then the two components must be variables (so you cannot write `["hi", x]` as a pattern, for example, since it has a string literal as one of the components). The whitespace mentioned above in terms (and patterns) is only allowed to consist of spaces (no newlines or tabs). So each equation must be written on a single line. Equations are terminated with a newline. There is an additional newline in between blocks. Here is some example Liskell code, from `slowrev.lkl`:

```
append [ x , y ] z = [x , append y z]
append x z = z
```

```
concat [ x , y] = append x (concat y)
concat x = x
```

Notice that only one function can be defined by the equations in a block (`append` in the first block, `concat` in the second). Your grammar cannot enforce this, but you will check it in your compiler. Liskell programs are executed by calling the `main` function with a list (right-associated nested pair structure) containing the command-line arguments:

```
[ "arg1" , [ "arg2" , ...]]
```

Terms are executed bottom-up: to call a function, all arguments are first evaluated to values (unless they diverge). Equations for a function are defined in order from top to bottom until one is found where all patterns match.

The assignment will walk you through implementing this compiler, and consider a number of extensions to the basic Liskell language.

1 Parsing Liskell [40 points]

Write a `gratr` grammar to parse Liskell programs in the format described above. You will need to use syntactic productions for terms, equations, and blocks (function definitions), while lexical productions could be used for string literals and names. The start symbol of your grammar must be called `file`, to interface correctly with the `main.agda` file I am providing you (which will not compile in Agda until you have written your grammar).

- You get 25 points for being able to handle the main test files. These are positive tests (which should be accepted by your grammar): `basic.lkl`, `slowrev.lkl`, `fulltree.lkl`, and `addall.lkl`. There are also negative tests: `no-1.lkl`, `no-2.lkl`, `no-3.lkl`, `no-4.lkl`, and `no-5.lkl`. Other test files are for subsequent problems.
- Add support for comment lines to your grammar. Comment lines begin at the very start of a line, with a `#` symbol. They extend to the end of the line, and can include the same characters as in string literals (but they do not have to be in double quotes, naturally). You can have 0 or more comment lines at the start of each function block, but nowhere else. `comment.lkl` is a positive test file, and `no-6.lkl` is a negative one. [5 points]
- Add support for multiline terms on the right-hand sides (only) of equations, if enclosed in curly braces. `multiline.lkl` is a positive example, while `no-7.lkl` is a negative one. [5 points]
- Add support for an infix string append operation, written with `^`. A positive test is `stringapp.lkl`. [5 points]

To do subsequent problems, you must do the first task above (grammar for the essential language, working for the main test cases). Comment lines, multiline terms, and string append are optional for subsequent problems.

2 Compiling to C [110 points]

Assuming you have your grammar properly parsing at least the main test files, you will now implement the compiler from Liskell to C. To do this, you will modify the `main.agda` file I am providing. As in previous homeworks, there is a datatype called `output-type`, for communicating a result back from the code you will write, to the code I am providing you. If there are no errors, you will use `c-output` to return a string containing the contents of a C source file that is the translation of the input Liskell file. If there is an error, you report it using `error-output`. The code I am providing takes care of writing the string you are giving me to a file called *name*`.c`, where *name* is the base name of the Liskell input file (like `basic` for `basic.lkl`); or printing the error message out for the user. The instructions below give tasks you can do. Problems 2.1 and 2.2 are independent, while subsequent problems depend on Problem 2.2 but are otherwise independent.

In `c-code.agda`, you will find the definition of some datatypes that you will use for generating C code. These datatypes represent syntax trees for a simple subset of C code, which is sufficient for compiling Liskell. Your code in `main.agda` will create a `C-file` value, and then my code (also in `main.agda`) will call `C-file-to-string` in `c-code.agda` to print out that C code. The string of C code which `C-file-to-string` is going to generate from a `C-file` will contain some boilerplate code, which is always included for every `C-file` that is printed. This boilerplate code defines types for pairs and strings, and functions for creating these. There is also a `print` function that can print any Liskell data structure (built with pairs and strings), and a `convert` function which will convert C arguments to a Liskell list. If no matching equation is found, the generated C code may call the `labort` function, also included in `c-start`.

There is one verification feature of the `C-code` datatype in `c-code.agda`. The constructor `CdecomposePair`, which represents C code for retrieving the components of a pair data structure `x`, requires a proof that, essentially, previous C code has already checked that `x` is a pair (by doing an `if`, represented by the `Cif` constructor). This ensures that the C code generated will never try to use a string as a pair, which would lead to a crash of the generated C program. The way the `C-code` type enforces this property (that surrounding C code must check that something is a pair before other C code tries to decompose it as a pair) is to keep track statically of the guards that are known to hold at a particular point in the represented program. The `C-code` type is indexed by a `C-guard`, and in passing to the `then`-part of a C `if`-statement, this index is augmented with the guard that has been checked.

2.1 Checking blocks for errors [35 points]

For this problem, you have to add code to `main.agda` to check for various errors that could occur in blocks. This is not necessary for the meat of the compiler, which is described in Problem 2.2 below. But I expect checking for errors may be easier than that problem, so there are a number of things you can check here. These are errors that cannot be caught by parsing with your grammar. To return an error message, your code should use the `error-output` constructor of `output-type` (the interface between the code you are writing and the code I have provided you).

1. Check blocks contain equations defining only a single function. When you have completed this, your tool should print an error message for `no-8.lkl` (which is otherwise syntactically

correct), as it contains a single block defining multiple different functions. [10 points]

2. Check that each equation in a block has the same number of arguments. When this is done, your tool should print an error for `no-9.lkl`. [10 points]
3. Check that `main` is defined in the file. When completed, your tool should print an error for `no-10.lkl`. [5 points]
4. Check that all variables used are already defined earlier in the file. When completed, your tool should print an error for `no-11.lkl`. [5 points]
5. Check that the patterns in an equation do not repeat variables. When completed, your tool should print an error for `no-12.lkl`[5 points]

2.2 Basic compilation[35 points]

Construct C code for input files with the basic syntax (no comments, multiline right-hand sides, or string append operators), where you may additionally assume:

- Each equation has only one pattern (`pat`) on the left-hand side which is not a variable. This pattern must be the first one.
- Functions are always called exactly the same number of arguments as their equations specify they take.

You need to generate C code which will call the program's `main` function with what you get back from the `convert` function (in `c-start` in `c-code.agda`) on the inputs `argc` and `argv` to the actual `main` function of the C source file. Also, your code should call the `print` function (also from `c-start`) to print the value retuned by the program's `main` function (if it terminates with a result).

Unlike in previous assignments, I am not giving you the C source files produced by my (currently incomplete) solution. You will need to consider the `.lkl` files to determine what output is expected when you run a compiled Liskell program.

It is fine if the C code you generate is flagged with (lots of) warnings when you compile with `gcc`. Just run `gcc` with the `-w` command-line parameter to suppress warning messages. Error messages, which prevent compilation, will not be suppressed. So you can run `gcc` like this, after running `main` on `basic.lkl` (for example), which generates `basic.c`:

```
gcc -w -o basic basic.c
```

This will generate the `basic` executable, which you can then run with command-line arguments; for example,

```
basic hi there bye
```

Partial credit will be based on which files you can handle:

- 15 points for `basic.lkl`
- Another 10 points for `fulltree.lkl`, where all functions have just one argument

- Another 10 points for `slowrev.lkl` and `addall.lkl` (note that a list of length N is used to represent the number N for this one, for understanding if you are seeing the correct output) – there are functions with two arguments (though they only give patterns for one of those arguments).

2.3 String append [10 points]

You can only do this problem if you do Problem 2.2, as well as the part of Problem 1 where you parse string appends. Add support for an infix string append operator, written with `^`. You will have to modify your grammar as well as `main.agda`. You will likely also have to add some code to `c-start` in `c-code.agda`. The test file `stringapp.agda` should work after this.

2.4 Tail Recursion [12 points]

You can only do this problem if you do Problem 2.2. Implement tail recursion. The test file `loop.lkl` should run without crashing after this (prior to that, it should fail with some kind of system error signaling that stack memory has been exhausted).

This is tricky to get working with multiple arguments (an example is `fastrev.lkl`), due to the internal verification aspect of the `C-code` type. It should be more manageable if you just implement it for functions with a single argument (an example is `loop.lkl`). You get 10 points for handling examples like `loop.lkl`, and 12 for ones like `fastrev.lkl`.

2.5 Functions with multiple patterns [15 points]

[I expect this to be very difficult – do not attempt unless you have enough other points already.] Extend your code for Problem 2.2 so it can handle equations with multiple patterns which are not variables. `multiarg.lkl` is an example.