PLC: Homework 6 [150 points, 170 maximum (capped)]

Deadlines

If the first letter of your HawkId is between 'l' and 'z' inclusive, then your solution is due by 9pm Wednesday, April 23rd. Otherwise, it is due by 9pm Friday, April 25th.

Remember that late homework is **not** accepted, but we will use second-chance grading as for previous homeworks. Do not forget to check your grade for the first-chance submission. We will post grades by Monday, April 28th at noon, at the latest (but likely earlier than that). Your second-chance submission is then due Tuesday, April 29th, by noon.

Update: You can earn extra credit, but this is capped at 170 points maximum for the assignment (even though there are point possibilities for more than that).

Subversion, Partners

The same instructions as for previous homeworks apply for turning in your solution via subversion, and for extra files you need to submit if you are working with a single partner.

For this assignment, you will check in files to the `hw6` subdirectory of your personal repository (see hw1 instructions for how to check out this repository).

Overview: the Mifl language

This assignment concerns a small language I have devised called Mifl (for "mini functional language", and pronounced "miffle"). Mifl resembles a very small subset of Agda. A Mifl program, which must be contained in just a single input file, consists of a sequence of one or more commands. A command is either:

- A datatype declaration, similar to a `data` declaration in Agda. An example of such a declaration is:

```
data nat where
  zero : nat
  suc : nat -> nat
```

- A function definition, like this (I am using a `fun` keyword, which we do not have in Agda, as well as a dot after each equation):

```
fun add : nat -> nat -> nat
add zero y = y .
add (suc x) y = suc (add x y) .
```

The exact syntax of these is described below. I have provided a couple example Mifl programs in several `.mifl` files in the `hw6` directory in the course repository.

In this assignment, you will first write a `gratr` grammar for Mifl, based on a starting version I am providing. Then you will implement a number of different operations on Mifl programs:

- a printer (to print out Mifl programs in Mifl syntax)
- a type checker
- a compiler (i.e. translator) to Java

In addition, there are a couple extra-credit opportunities.

1 Parsing Mifl [60 points]

For this problem, you should submit a grammar named `mifl` in a `mifl.gr` file. This grammar should implement the syntax of Mifl, which is described in detail below. Please start with the `mifl.gr` file I am providing in the `hw6` directory of the course repository. That file contains productions defining the `symb` nonterminal (for all symbols referenced in the description below) and the `ws` and `ows` nonterminals, for whitespace and optional whitespace, respectively. The `ws` nonterminal in particular is a little tricky, since I wanted it to include strings representing comments in Mifl code (see `empty.mifl` and `list.mifl` for example comments).

As mentioned above, a Mifl program consists of a sequence of one or more commands, which are either datatype declarations or function definitions. The syntax for these is described next. You will receive 40 points if your grammar accepts the `.mifl` test files I am providing, and rejects the files ending in `.notmifl`. 20 points will be awarded for additional hidden test files.

Because the grammar will be quite large, you will need to use the new `--use-string-encoding` command-line argument to `gratr`. For this, you will need to do an "svn update" for `gratr`, and then recompile `gratr`. On Mac/Linux, you may need to run `make clean` before you run `make`. On Windows, just running `compile` as you originally did is sufficient.

1.1 Datatype declarations

The components of a datatype declaration are:

1. the `data` keyword
2. then whitespace (not optional)
3. then a symbol, for the name of the datatype
4. then whitespace (not optional)
5. then the `where` keyword
6. either nothing or else whitespace (not optional) followed by a whitespace-separated list of constructor declarations

A constructor declaration looks like:

1. a symbol
2. optional whitespace
3. the `:` punctuation
4. optional whitespace
5. a type

Types are either:

- symbols
- arrow types, which look like:
    1. a type
    2. optional whitespace
    3. the `->` punctuation
    4. optional whitespace
    5. a type

    Note that arrow types associate to the right.
- parenthesized types (where we can have optional whitespace between the parentheses and the type which is being parenthesized)

1.2 Function definitions

A function definition looks like

1. The `fun` keyword
2. whitespace (not optional)
3. a symbol, for the name of the function
4. optional whitespace
5. the `:` punctuation
6. optional whitespace
7. a type (as above)
8. either nothing or else whitespace and then a whitespace-separated list of equations

An equation looks like

1. a term
2. optional whitespace
3. the `=` punctuation

4. optional whitespace
5. a term
6. optional whitespace
7. a dot (`.`)

A term is either

- a symbol
- a parenthesized term, where optional whitespace can come between the parenthesized term and the parentheses
- an application, which is a term, then whitespace, then another term

Note that applications associate to the left.

2 Printing Mifl [25 points + 7 extra-credit points]

For this problem and subsequent problems, you will be turning in `my-mifl-main.agda`, which you should create by copying the `mifl-main.agda` file generated by `gratr` from your grammar, and then modifying similarly to the way you modified `my-bvcalc-main.agda` in `hw5`.

Add a command-line option to `my-mifl-main.agda` called `--printMifl`. You just have to study the code `gratr` has already generated in `my-mifl-main.agda`, to see how to add a command-line option. If this command-line option is given, then your `process-start` function should print the parsed input back out in Mifl format, before it does anything else. So concretely, you will want to extend `process-start` with an extra boolean argument telling whether or not to print out the parsed input in Mifl format. And you will then want to write a function that can convert a `start` value into a string in Mifl format.

We will grade your solution to this problem by using your `my-mifl-main` executable to print some Mifl programs back out. We will then re-parse those programs with our own version of `my-mifl-main`, and confirm that they have the same parse trees (with `--showParsed`) as the original inputs. This approach – comparing the parse trees – will let us compare programs which have different whitespace. So you do not need to worry about somehow matching the whitespace of the original input, which would likely not be practical to try to do. Just print out the parse tree in Mifl syntax using any whitespace that would be legal to use.

Extra Credit. For 7 extra-credit points, add a new command-line option called `--printMinMifl`, which works similarly as `--printMifl`, except that it prints out types and terms with the minimal parentheses needed, given the associativites of applications (left associative) and arrow types (right associative). In practice, I suggest you do this by implementing a function called `dropParens` which takes a `start` value and returns another one where all types and terms have the syntax-tree nodes for parentheses dropped where they are not needed. You can drop parentheses around an arrow type if it is the right subexpression of another arrow type; so `T1 -> (T2 -> T3)` can be printed as just `T1 -> T2 -> T3`. And you can drop parentheses around an application if it is the left subexpression of another application; so `(t1 t2) t3` can be printed as just `t1 t2 t3`. Also, you

should drop parentheses if the expression in question already appears inside parentheses; so `((e))` should become `e`.

3 Checking Types [40 points (updated)]

Implement another command-line option `--checkTypes` which will tell your `process-start` function to check the types of the Mifl program it is given. It should perform type checking after any printing out of programs in Mifl syntax, from the previous problem. To check the types of Mifl programs, you will need to check datatype declarations as well as function declarations. You will likely want to use a trie data structure (as we were doing in class for the `petlet` example) to implement a *typing context* mapping symbols to information about those symbols. We will see an example in class April 15th and 17th.

If there is a type error, your tool should print some error message (even just ``` ``type error'' ``` is ok). If there is no type error, your tool should print nothing.

3.1 Type-checking datatype declarations

A datatype declaration is considered type correct iff the following conditions hold:

1. The name of the datatype is not already the name of a datatype, function, or constructor, and similarly for the names of the constructors.
2. The symbols appearing inside the types of constructors are all previously declared datatypes, possibly including the datatype currently being declared.

Once a datatype declaration has been checked for type correctness, the datatype name and the constructors become available for use in type checking later datatype declarations and later functions, and should be added to the typing context.

3.2 Type-checking function definitions

A function definition is type checked as follows.

1. First check that all the symbols occurring in the type which is declared first for the function are previously declared datatypes.
2. The type for the function which is currently being defined should be added to the typing context.
3. We will allow programs not to cover all possible inputs with their equations (since it is a little tricky to implement the check for this). So the next thing to check is that each equation in the datatype declaration is type correct. This is done as follows:
   (a) No symbol on the left-hand side of an equation is allowed to be a previously declared datatype or a previously defined function.

(b) Based on the type of the function, you can deduce what the types of the arguments must be, and confirm that any constructors used in the arguments (on the left-hand side of the equation) have the proper return types.

(c) Any other symbols used on the left-hand side of the equation are pattern variables. You should add type information about them to the typing context.

(d) Then you should type-check the right-hand side, making sure that functions and constructors are applied to arguments of the correct types, and that all symbols used are either previously declared/defined datatypes, constructors, or functions (including the current function), or else pattern variables. You should also confirm that the type of the entire right-hand side matches the return type of the function.

3.3 Points for this problem

You can earn these points:

- affirming that `simpleNat.mifl` is type correct: 5 points.
- affirming that `nat.mifl` is type correct: 5 points.
- affirming that `list.mifl` is type correct: 5 points.
- rejecting `nat1.notype` as not being type correct: 2 points.
- rejecting `nat2.notype` as not being type correct: 2 points.
- rejecting `list1.notype` as not being type correct: 2 points.
- rejecting `list2.notype` as not being type correct: 2 points.
- rejecting `list3.notype` as not being type correct: 2 points.
- additional testcases: 10 points.

Your solution is not allowed simply always to return "type error" or always to return nothing. You will not receive any points if we detect that your tool always answers the same thing.

4 Compiling Mifl to Java [40 points (updated), 8 extra-credit]

Implement another command-line option `--emitJava` which will tell your `process-start` function to compile the Mifl program it is given to Java. This sounds intimidating, but because we will not try to do anything too fancy, it will basically amount to printing out the code in Java syntax (somewhat similarly to printing out the code in Mifl syntax in the second problem). When given `--emitJava`, your `process-start` function should produce a string containing a Java version of the input Mifl formula (and anything else that was requested with other command-line options). That Java code, when compiled and executed, should execute the given Mifl program and print out any value that is computed (Mifl programs can run forever, so in those cases the Java code would also run forever when executed).

Note that you can get 10 points just for handling the relatively easy `simpleNat.mifl`, which does not require compiling nontrivial functions, only datatypes and a simple definition. So this could be a good target if the rest of this seems intimidating.

In more detail:

- Print out the definition of a class called `output` (see the example from class on Thursday, April 17, in particular `output.java`). Note that you are just printing out the Java code from your program. To compile the Java code, you will need to save it to a file called `output.java` (or copy-and-paste it into an online Java compiler like `http://www.compileonline.com/compile_java_online.php`). If you are testing on a Windows computer, you may find it helpful to run the Windows command shell from within `emacs`. To do that, type "Alt-x" in `emacs`, and then type the word "shell" and hit enter.

- Each datatype in your Mifl program should get translated to a parent class for the datatype, and subclasses for the constructors of that datatype. These classes are defined inside the `output` class. There are probably several ways to do this, but here is one suggestion. To compile

  ```
  data nat where
    zero : nat
    suc : nat -> nat
  ```

  you can generate these classes:

  ```
      public static abstract class nat {
  public static int zero_TAG = 0;
  public static int suc_TAG = 1;
  public abstract int getTag();
      }

      public static class zero extends nat {
  public int getTag() {
      return zero_TAG;
  }
  public zero() {}
  public String toString() {
      return "zero";
  }
      }

      public static class suc extends nat {
  public int getTag() {
      return suc_TAG;
  }
  protected nat suc_data0;
  public suc(nat suc_data0) {
      this.suc_data0 = suc_data0;
  ```

```
}
public nat get_suc_data0() {
    return suc_data0;
}

public String toString() {
    return "(suc " + (suc_data0.toString()) + ")";
}
    }
```

A couple notes:

- The parent class and subclasses have the same names as in the Mifl program.
- The parent class is abstract because of an abstract `getTag()` method, which will be defined in the subclasses.
- The idea with `getTag()` is that you can ask a `nat` what kind of `nat` it is, by calling `getTag()`. If `getTag()` returns `zero_TAG`, for example, then you know that the `nat` is really an instance of the `zero` subclass.
- For Mifl constructors like `suc` which take arguments, the `suc` class in the Java code will have fields for the arguments. Here, I just named the field `suc_data0`, and created an accessor method `get_suc_data0()` to retrieve this data. When you compile functions that do pattern-matching on `nat`s, this method will be useful.
- Each class should implement a `toString()` method, to print the data as a string. Constructors like `zero` with no arguments should be printed as `zero`. Constructors like `suc` that take arguments should be printed with parentheses around the constructor and argument. So `suc zero` should be printed as `(suc zero)`. We will grade your solution based on the exact format of the answers printed out, so it is important that your `toString()` methods adhere to this format.

- To compile a function in a Mifl program, you can assume, for full credit, that the patterns on the left-hand sides of the equations are at most one level deep: they are either pattern variables or a constructor applied to pattern variables (like `suc x`, for example). All the sample functions I am providing in the `.mifl` files have this property.
- You can assume that the functions you are asked to compile do indeed type-check in Mifl (even if you did not complete the previous problem to implement a type checker for Mifl). Code that does not type check in Mifl will simply get compiled to code that does not type check in Java – and that is fine.
- To compile a function, you first need to emit the declaration of the method in Java. For example, for this Mifl function

```
fun add : nat -> nat -> nat
add zero y = y .
add (suc x) y = suc (add x y) .
```

You will emit the following static method as part of your `output` class:

```
public static nat add(nat x0, nat x1) {
```

I am choosing to name the input variables uniformly with a number, but this is not required.

- You can then emit code for the equations defining the function, one at a time in order. For each equation, you first have to emit code that tests to see if the inputs match the patterns on the left-hand side, and define local variables (in Java) corresponding to the pattern variables. Then you can compile the right-hand side. You can assume that any function that is called is being called with all its arguments. When constructors are called, in the Java code you have to use the Java `new` keyword to create a new instance of the class. For example, `zero` in Mifl code will turn into `new zero()` in the Java code. For example, here is how I propose to compile the `add` function from Mifl:

```
    public static nat add(nat x0, nat x1) {
if (x0.getTag() == nat.zero_TAG) {
    nat y = x1;

    return y;
}

if (x0.getTag() == nat.suc_TAG) {
    nat x = ((suc)x0).get_suc_data0();
    nat y = x1;

    return new suc(add(x,y));
}

return null;
    }
```

- If the program defines `main` with type that is just a datatype (so `main` is not a function, but just data), then your `output.java` class should emit a `main` method which prints out the result of evaluating the Java version of `main`.

For a complete example of all this, see the `output.java` which I have included now in the `hw6` subdirectory of the class repository.

4.1 Grading

You will get 10 points for correctly handling `simpleNat.mifl`, 7 points for `nat.mifl`, and 8 points for `list.mifl`. We will reserve 10 points for hidden testcases.

4.2 Extra Credit [8 points]

Implement support for nested patterns on the left-hand sides of equations. An example program using this feature is:

```
data nat where
  zero : nat
  suc : nat -> nat

fun divtwo : nat -> nat
divtwo zero = zero
divtwo (suc zero) = zero
divtwo (suc (suc x)) = suc (divtwo x)
```