Final Project: Unifier

CS 440: Programming Languages and Translators, Spring 2020

Due Fri May 1, 11:59 pm?

1. Unifier

The final project is a Haskell program to find the most general unifier for a set of textual equations. You should define a function `solve :: String -> String` that

- Takes a string representation of a unification problem.
- Parses it using a recursive descent parser.
- Uses the unification algorithm to solve the problem.
- Returns a pretty-printed[1] version of the solution, as a `String` (or `"FAIL"` if there is no solution).

2. Collaboration

- This is a one-person project. (Sorry, no teams.) On the other hand, if there's code you can use from a homework assignment, you can use it, even if you did the homework assignment as part of a team.

3. Lexing and Parsing

- Write a function `parse :: String ->` … (*type representing a*) *unification problem* …
- We'll use expressions as the "terms" in unification equations (see Lecture 15), but we do need new syntax for unification problems and equations (see below).
- The grammar for expressions is mostly the one we've been using with the addition of "variables", which are like identifiers but they can be substituted for. "Identifiers" are named constants and can't be substituted for. The parse trees for expressions needs to have a variables case added in.
- You'll need some `data` or `type` structures to represent *unification problems*, *unification equations*, and *substitutions*.
- Grammar for unification problems (new rules for problems, equations, and modified rule for factors):

  Problem → { *Equations* }

  Equation → *Expr* = *Expr*

  Equations → *Equation* | *Equation* `,` *Equations*

  Expr → … *as before, with* `+`, `*` ...

  Factor → *identifier* | *variable*

  Factor → *constant* | `\(` *Expr* `\)` | *Function_call*

  Function_call → *identifier* `\(` (*Arguments* | ε) `\)` *// includes calls like* `f()`

  Arguments → *Term* | *Term* `,` *Arguments*

[1] For this assignment, "printing" just means returning a string, not using the Haskell print routines.

- Lexical issues:
  - *identifiers* and *variables* are a letter followed by alphanumerics or `\_`.
    - For *identifiers,* the initial letter is in lower case, as in `xY2_z`.
    - For *variables,* the initial letter is in upper case, as in `Xy2_Z`.
  - *Whitespace* can include `\n`.
  - *constants* are integer constants with an optional leading hyphen.

4. Unification and Substitution

- Write a function `unify ::` *…unification problem…* `->` *…unification solution….* to solve a unification problem (or fail). You'll want at least a helper function for handling a *single unification equation* (i.e., one iteration in the unification algorithm loop). The algorithms for unification is in Lecture 15.
- For substitution, you'll need to design `data / type` structures for *substitution bindings* and *lists of substitution bindings*. You'll need a function that applies a substitution binding to an expression and returns an expression. (Something of type *sub_binding* `->` *expr* `->` *expr*.) The algorithm for this is in Lecture 15. You'll also need to be able to apply a substitution binding to a unification problem, unification equation, and to substitutions themselves.
- Use your substitution function to also write a `substitute` function so that `substitute` e_2 *var* e_1 performs $e_2[var \mapsto e_1]$. (That way we'll have a consistent interface for testing.)
- **Handling Failure**: The unification algorithm can fail; the most natural way to handle this is have the algorithm return a `Maybe` *list_of_substitutions*. To print the result of substitution, you would check for `Nothing` versus `Just` a list of substitutions. But you can use whatever `data` / type you decide.
- **Extra Credit Handling Failure**: [Also see **Extra Credit** below] An alternative to having the unification algorithm produce simply `Nothing` if it fails, have the algorithm return the then-current problem and substitution so that to show what things looked like when the algorithm encountered failure, you "print" this problem and substitution. (You'll have to design a `data` type to return (one of) two different kinds of results, similar to the `Either` built-in type operation.)

5. Pretty Printing

- Write a function `pprint ::` … *unification solution* … `->` `String` that takes a unification problem solution (possibly failure) and returns a prettier string than `deriving Show` might give.
- To pretty-print the solution, follow the syntax above for unification problems, with the left-hand expression of each equation restricted to be a variable. For example, a solution to the problem "`{ f(X, b) = f(Y, Z)), g(c) = Y }`" might be "`{X = g(c), Z = b, Y = g(c)}`". You're encouraged to add a space or two in places to make the solution more readable.
- **Extra Credit for Pretty Printing**: [Also see **Extra Credit** below]
  - **Break up the output**: Insert `\n`'s in the output of pretty printing to keep each line from becoming (what you consider to be) too long.

- **Use the `Show` typeclass**: Name your pretty-printing function `show` and declare your unification output `data` type to be an instance of `Show`. (The type has to be a `data` type; restriction from Haskell.) This way, if you run the unification algorithm in `ghci`, the result will be printed out. To see how to declare a `data` type to be an instance of a typeclass, see *Typeclasses 102* in Chapter 8 (*Making Your Own Types and Typeclasses*) of the LYaH book. (If you want to make others of your data types instances of `Show`, go ahead.) *Hint:* Define a `pprint` function as above and declare `show = pprint`.
- Note: Until you actually declare your solution type to be an `instance` of `Show`, you can use `deriving Show` to at least see a regular representation of your solution value.

6. Testing

- It's left to you to design the bulk of test cases, but here are a few:

```
--
-- Some sample data:
--
p1 = "{X=Y,X=3}"              -- Solution: Y=3, X=3 (not Y=3, X=Y)
p2 = "{X=1,X=3}"              -- Unification fails (tries to unify 1 and 3)
p3 = "{f(a,Y) = f(X,b), c = Z}"    -- Solution: Z=c, Y=b, X=a
p4 = "{f(X) = g(Y)}"          -- Unification fails (different function names)
p5 = "{f(X,Y) = f(X)}"        -- Unification fails (different # of arguments)
p6 = "{f(f(f(a, Z),Y),X),W) = f(W,f(X,f(Y,f(Z,a))))}"   -- (found on Wikipedia)
-- Solution: Z=a, Y=f(a,a), X=f(f(a,a), f(a,a)), W = f(f(f(a,a), f(a,a)), f(f(a,a), f(a,a)))
```

7. What to Turn In

- Submit an `*.hs` file containing your solution. It should be complete in the sense that copying/pasting the file's contents into `ghci` between the usual `:{` and `:}` lines should compile your solution and make it ready for use. If you want to make your solution a `module`, that's okay but not required. In any case, if one can just `:load` your solution, please indicate that in the Blackboard comments.
- If you break up your project into multiple `*.hs` `module` files (not a requirement), `zip` them together and submit that as your solution. (In the Blackboard comments, please say what module is the main one to `:load`.)
- Every `*.hs` file you submit should include your name and A-id.
- To make testing easier, some standard function names are required. They're described above, but a summary:
  - `solve :: String -> String`. Takes a problem represented as a `String`, parses it, runs the unifier on it, and returns a pretty-printed string representing the solution (or failure).
  - `parse :: String ->` … (*data / type representing_ a unification problem* …. Parses a problem.
  - `unify ::` … *unification problem* … `->` … *unification solution* …. Runs the unification algorithm to solve the problem (or fail).
  - `substitute ::` *E* `->` *V* `->` *E* `->` *E* (where *E* is the `data` type representing expressions and *V* is the `data` / `type` representing variables). `substitute` e_2 *var* e_1 returns $e_2[var \mapsto e_1]$.

- `pprint` :: … *unification solution* … `-> String`. Produce a nicer-to-read version of the unification solution (which could be failure).

8. Grading Guide [100 points total]

- [12 pts] Lexical analysis
- [16 pts] Parsing (expressions, problems, equations)
- [32 pts] Solve unification problem
  - [8 pts] Initialize & run loop; Handle one equation: do case split
  - [9 pts] Simple cases: $X \equiv X$, $e \equiv X$, $c_1 \equiv c_2$, failure due to mismatch (const ≡ fcn call, e.g.)
  - [10 pts] Add substitution cases: X ≡ Y, X ≡ e; apply substitution binding, add to solution
  - [5 pts] Function call $f(\ldots) \equiv g(\ldots)$, check name, arity; add new equations
- [16 pts] Substitution (on problems, equations, substitutions, expressions)
- [14 pts] Pretty-print output (unification solution, expressions)
  - You're allowed to print redundant parentheses around additions.
- [10 pts] Code organization

9. Extra Credit

- Unification
  - [10 pts] Print remaining problem and partial solution if unification fails.
- Pretty Printing
  - [4 pts] No redundant parentheses when pretty printing expressions.
  - [5 pts] Add `\n` to avoid long output lines.
  - [3 pts] Use `Show` typeclass

10. Things left to you to work out

- Make the grammar LL(1).
- Make sure the lexer handles `\n` in whitespace.
- Add variables to `data` type for expressions.
- Design data / type to represent problems, equations, substitutions, ….
- Translate substitution and unification algorithms from Lecture 15 into Haskell.
  - Handle different cases with their own functions? Or as part of `substitute` or `unify`?
- Determine format of pretty-printed output.
  - Which parts of `pprint` problem, equation, expression deserve their own routine?
- Invent test data.
- ???

11. Hints

- Write short descriptions for each function; try to be precise with the relationship between arguments and results, especially for recursive functions.
- Write / enter / debug smallest cases & base cases first, larger / more-complicated / recursive cases later.
- Unit test thoroughly before integrating into rest of code.
- Initially, use stubs for applying substitutions to problems and substitutions (just return the problem / substitution unchanged). Replace stubs with real code later.

