PLC: Homework 5 [100 points, 105 possible]

Deadlines

If the first letter of your HawkId is between 'l' and 'z' inclusive, then your solution is due by 9pm Wednesday, April 9th (**new deadline**). Otherwise, it is due by 9pm Friday, April 11th (**new deadline**). As mentioned in class, students with HawkIds between 'l' and 'z' inclusive will be submitting Wednesdays for the homeworks due after Spring Break, and everyone else will be submitting Fridays.

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 7th at noon, at the latest (but likely much earlier than that). Your second-chance submission is then due Tuesday, April 8th, by noon.

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 `hw5` subdirectory of your personal repository (see hw1 instructions for how to check out this repository).

In response to your feedback, I have tried to split up the problems for this assignment into different files. We will grade each file separately. If one file fails to check in Agda, you will lose all possible points for that file only. This is different from homeworks 1 through 4, where if one file failed to check in Agda, then you lost all points for the whole assignment.

1 Reading

I am working on a Chapter 7 on parsing with `gratr` to the book. This explains how to download `gratr` and use it to generate parsers in Agda. You can find the book as `book/book.pdf` in the class repository, or directly here:

`https://svn.divms.uiowa.edu/repos/clc/class/111-spring14/book/book.pdf`

2 Grammars

For the problems below, you have to write grammars in the `gratr` format, for certain languages I am describing. To obtain `gratr`, just check out the following repository using subversion, with username 'guest' and password 'guest' (just like for the course repository):

`https://svn.divms.uiowa.edu/repos/clc/projects/gratr2`

The README file in the gratr2 directory you get this way has instructions on how to compile gratr. You do need to install OCaml first for this to work – but that is an easy installation.

For these problems, you do not need to be concerned about what the syntax trees look like for parsed input. Your goal is just to write a grammar that recognizes the given language. Your grammar must be processed by gratr without any error messages. Otherwise, you will not get any points for that grammar.

Each subsection below names a file you should submit as part of your solution. So you have to create, add, and then commit these files to your personal repository using subversion, to turn them in for this assignment. As noted above, contrary to howeworks 1 through 4, if you do not submit one of these files or if it has errors and cannot be processed by gratr, you will receive 0 points only for that file. Your grades for the other files will not be affected by having one file wrong.

2.1 abca.gr [10 points]

In a file abca.gr, create a grammar called abca for the language of strings which start with the letter 'a', then optionally have either a 'b' or a 'c', and then continue in the same way until the end of the string, where they end with an extra 'a' or 'b', followed optionally by a newline ('\n'). Examples that your grammar should accept are in abca-yes1.txt, abca-yes2.txt, and abca-yes3.txt. Accepting means that running the generated abca-main executable on the input file produces no output. Examples that your grammar should reject are in abca-no1.txt and abca-no2.txt. Rejecting means that running abca-main on the input file produces some error output. We will test your grammar on a few other examples. Note that whether your productions are syntactic or lexical does not matter for this problem – or for the next – as we are just testing whether your grammar correctly specifies the language described above. You will get 5 points for passing the tests we have provided, and 5 more points for passing additional tests we are not making public.

The only file you need to submit via subversion is abca.gr. We will use gratr to generate abca-main and run it on the various test files. Note that the name of this file and the name of the grammar (which is the first thing listed in any gratr file) must be exactly as specified, so our grading scripts can process your grammar. If you fail to name your file or the grammar correctly, you will receive 0 points for this problem. The same goes for the other problems below.

2.2 idlist.gr [10 points]

In a file called idlist.gr, create a grammar called idlist for recognizing newline-terminated nonempty whitespace-free comma-separated lists of identifiers, where

- an identifier must begin with a lowercase letter, and then it may have any number (including 0) of lowercase letters and numbers.
- commas (',') must come between identifiers.
- there is no whitespace (i.e., spaces or newlines), except that at the end of the input a single newline ('\n') is required (not optional).

- the list must contain at least one identifier.

If you find that the time required to compile the file with Agda is too long, my advice is to simplify the grammar, as you are developing it, by first considering only characters in the range `'a'` through `'c'`. When you seem to have testcases working for the smaller character range, you can easily add the bigger range (which is required for the final submission).

We have provided positive and negative testcases for you, such as `idlist-yes1.txt` and `idlist-no1.txt`. You get 5 points for passing the testcases we are providing, and 5 points for passing additional testcases we are not disclosing.

Again, you need only turn in the `idlist.gr` file with subversion, containing your `idlist` grammar. We will run `gratr` to generate the other files, and then run the tests with `idlist-main`.

2.3 `expr.gr` [10 points]

In a file called `expr.gr`, create a grammar for applicative expressions followed by a newline. An applicative expression is either a symbol – and for purposes of this problem, we will say that symbols are non-empty sequences of lowercase characters in the range `'a'` through `'c'` – or else it is a parenthesized expression, or else it is an application: one expression followed by a space and then another expression. An example of an applicative expression is

$$(f\ (h\ a\ (g\ b))\ (g\ (g\ c)))$$

Note that in a case like $a\ b\ c$, the applications should be viewed as grouped to the left. So we have $a\ b$ applied to c. Positive examples (ones your grammar should accept) are in `expr-yes1.txt`, `expr-yes2.txt`, and `expr-yes3.txt`; negative examples are `expr-no1.txt` and `expr-no2.txt`. You get 5 points for passing the provided tests, and 5 points for passing additional undisclosed ones.

2.4 `formula.gr` [20 points]

In a file called `formula.gr`, write a grammar called `formula` which accepts formulas followed by a newline, where formulas are described as follows:

- We have atomic formulas, which have a symbol, then an open paren, then a comma-separated list of symbols (possibly empty), and then a close paren. An example is `a(b,c,a)`. As above, we will limit symbols to lists of characters between 'a' and 'c'.
- We have implications, which have a formula, then $\to$, and then another formula.
- We have universal quantifications, which have a $\forall$, then a space, then a symbol, then a period ('.'), and then a formula.
- We can have parentheses around a formula.

The scope of the $\forall$ is as far to the right as possible. Implication is associated to the right. Positive test cases are in `formula-yes1.txt`, `formula-yes2.txt`, `formula-yes3.txt`, and `formula-yes4.txt`.

A negative testcase is in `formula-no1.txt`. As above, you can earn 10 points for passing the testcases we are giving you, and 10 more for hidden testcases.

3 The `bvcalc` program [50 points]

The goal of this problem is to create a calculator called `bvcalc`, for bitvector operations. A bitvector is just a sequence of 0's and 1's; for example, `00110`. Your calculator will be able to read a file containing a bitvector expression, and evaluate that expression to produce an answer. Bitvector operations include shifting left or right, and'ing, or'ing, xor'ing, and negating bitvectors. You first have to parse in bitvector expressions, and then you need to write Agda code to process the syntax trees obtained, and produce the desired output.

The two files you will be submitting are:

- `bvcalc.gr`
- `my-bvcalc-main.agda`

3.1 A Grammar for Bitvector Expressions [20 points]

In a file called `bvcalc.gr`, write a grammar called `bvcalc` for bitvector expressions. I suggest using `bv` as the name for the nonterminal for bitvector expressions. These can be:

- A bitvector literal, which is just a sequence of 0's and 1's, like `00111`. I suggest using `bvlit` as the name for the nonterminal for bitvector literals.
- A shift-left expression, which is a bitvector expression followed by $\ll$ (`\ll` in Agda mode) and then a decimal number. An example is $111 \ll 2$. The effect of the shift-left expression is to add the given number of zeros to the right (least significant position) of the bitvector. So the given example would evaluate to 11100.
- A shift-right expression, which is a bitvector expression followed by $\gg$ (`\gg` in Agda mode) and then a decimal number. The effect is to drop the given number of bits from the right end of the bitvector. So $110011 \gg 3$ evaluates to 110.
- Conjunction (`&`), disjunction (`|`), exclusive or ($\oplus$, which is `\oplus` in Agda mode), and negation (`~`) are as usual: the first three of these are infix operators on (two) bitvector expressions, and the last (negation) is a prefix operator on one bitvector expression.
- Parentheses can be placed around a bitvector expression.

All binary operations have equal precedence and should be treated as right associative. So `1 & 0 ⊕ 1` should be interpreted as `1 & (0 ⊕ 1)`, which evaluates to 1. The shift expressions have higher precedence (bind more tightly to their arguments). Finally, negation has highest precedence. These facts should be incorporated into your grammar by the addition of rewrite rules. I found that the number of rules was getting a bit out of hand unless I had a single production for binary operations, where the production uses a `binop` nonterminal for which binary operation is being used. Similary, I have a single production for shift operations, with a `shiftop` nonterminal.

Your grammar should allow optional whitespace everywhere. This entails defining a lexical nonterminal (I usually use `ows` as the name) for optional whitespace, and then incorporating it between symbols in your other productions. This will allow you to write expressions like `1 & 0 ⊕ 1`, instead of just `1&0⊕1`.

Positive testcases are in files like `bvcalc-yes1.txt`.

I found that with my full `bvcalc` grammar, it could take a minute and a half or so for Agda to compile the `.agda` files generated by `gratr`. You may find you want to develop your grammar a little at a time, so that you can test the parts you have without always having to wait such a long period of time with any change you make to the grammar.

3.2 Evaluating bitvector expressions [30 points]

Your goal now is to write Agda code to evaluate a bitvector expression. You will hook this code into the Agda code that is generated by `gratr`, by modifying the `bvcalc-main.agda` file. Actually, you should not modify that file, but instead create a copy of that file called `my-bvcalc-main.agda`. You will modify that file, and submit it via subversion for this part of the problem. Working on `my-bvcalc-main.agda` instead of `bvcalc-main.agda` will ensure that your code is not overwritten if you (or we) run `gratr` (since `gratr` will create the file `bvcalc-main.agda` every time). Note that you need to change the name of the module at the top of that file to be `my-bvcalc-main`.

In `my-bvcalc-main.agda`, you will see at the very top of the file a function called `process-X` where `X` is the start symbol of your grammar. Right now, this function just takes in a parse tree for `X`, and returns the empty string. You need to modify this function so that it calls an evaluator for bitvector expressions, and then returns the answer, which will be a bitvector literal, as a string. The string you return should be in the same format as bitvector literals in the input syntax (e.g., `10011`). Your evaluator should have a type like `bv → bvlit`, assuming that `bv` is the nonterminal you use for bitvector expressions in your grammar, and `bvlit` the nonterminal for bitvector literals. Your `process-X` will then need to call some additional helper function to convert the `bvlit` to a string. A couple notes about your evaluator:

- When evaluating a shift-right expression like `e ≫ n`, if the number n is 0 then the value is just the value of e. If the number is greater than the number of bits in the value of e, then then we will say that the answer should be the bitvector 0.
- When evaluating a shift-left expression like `e ≪ n`, if n is 0 then the value is just the value of e.
- Do not drop leading zeros from bitvectors when you are evaluating them.
- If a binary operator (like `&`) is applied to bitvectors of different length, then the shorter bitvector should be padded out on the right with zeros, until it is the same length as the other bitvector. Then the operation can be applied. For example, if you have `101 & 11`, then the shorter bitvector `11` needs to be padded out with zeros first on the right. So your evaluator should treat this as `101 & 110`. Then you perform the operation on the two bitvectors of the same length. So in this case, the answer would be `100`, because we take the conjunction ("and") of corresponding bits of the two input bitvectors to obtain the output bitvector.

There are some testcases in 10 files with names like `bvcalc-eval1.txt`. You get 2 points for passing each of these testcases, and then there are 10 points more for some hidden testcases.

3.3 Extra Credit: Verifying Your Evaluator [5 points]

In a file called `verified.agda`, prove a theorem about your evaluator. You can prove any theorem you want. The course staff will assign points between 1 and 5 for any theorem you prove, based on how nontrivial we judge your theorem to be. For example, proving a theorem that is just a testcase (like that calling your evaluator on the syntax tree representing `100 & 101` produces the syntax tree corresponding to `100`) will earn 1 point. Proving a theorem involving some variables – so you are covering an infinite number of testcases – will earn more.

4 Grading, Help

As for previous homeworks, each of the files you submit must check with no errors. Differently from previous homeworks, though, if one file does not check in Agda, you will only lose the points associated with that file. If other files do check, you will still get any points you earned for those files.

As for previous homeworks, you can post questions in the `hw5` section on Piazza. Note that you will likely get a faster answer if you ask on Piazza than if you email us.

You are also welcome to come to our office hours. See the "Course Staff" section of the Piazza page for times and locations of office hours, which we will likely change following Spring Break.