EECS 598 and EECS 498 (Fall 2021) – Assignment 1
Due date: Tuesday, September 14 at midnight EST

Collaboration. You are free to discuss the assignment with others or work together towards the solution. However, all of your code must be written by yourself. Your write-up must be your own as well. **Please list your collaborators at the top of your submissions.**

Goal. The goal of this assignment is to help you get familiar with a top-down search-based synthesizer. In particular, in this assignment, you will extend an existing synthesizer to make it support a new language.

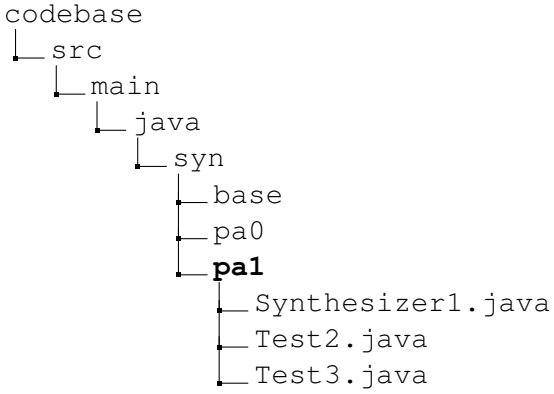
1 Getting started

Download the latest codebase from
https://web.eecs.umich.edu/~xwangsd/courses/f21/assignments/codebase.zip.

Replace the previous code base with this new one. We fixed one Linux-specific bug in the `mkDataframe` function in `Dataframe.java`.

Download `A1` from
https://web.eecs.umich.edu/~xwangsd/courses/f21/assignments/pa1.zip.

You will see a `pa1` folder. Put it under `syn`.



As mentioned in Assignment 0, there are some additional key classes in the `base` folder. In this assignment, we will explore how these classes work.

- `AST.java`: we represent R programs internally as abstract syntax trees (ASTs). `AST.java` provides some basic functionalities, such as `getChildren` and `getParent`, which allow us to traverse the AST. It also has a function called `toR` which translates an AST into a string that corresponds to an R program which can be directly executed in R Console. Another very important function is `expand` which we will explain in detail later when we explain how the synthesizer works.
- `ASTNode.java`: every node in the AST is an `ASTNode` object. This class tells us the grammar symbol, `symbol`, and the operator, `operator`, that this node corresponds to.
- `CFG.java`: this is a context-free grammar (CFG) that is used to define the search space. It simply maps each symbol in the grammar to all productions which use that symbol as the return symbol.

- `Production.java`: a production consists of a return symbol, an operator, and an array of argument symbols. Note that, in our code base, we treat terminal symbols as *nullary operators*. That is, if you have a production $df ::= x$, we treat x as an operator that takes no arguments.

- `Synthesizer.java`: this is the most important class that we will work with in this assignment (and also in subsequent assignments). A synthesizer has two important fields, `Interpreter` and `CFG`. That is, it needs an interpreter so it can execute programs and examine their outputs during search; it also needs a context-free grammar (CFG) so it knows what programs it's searching over, i.e., the search space. Our synthesizer has an additional field, `bound`, that essentially bounds the search space. You can choose your own way to bound the search space. For instance, you can bound the maximum number of operators a program may have, that is, the maximum number of nodes in the AST. You can easily do this using the function `numOfOperators` in the `AST` class. The key function in `Synthesizer.java` is the `run` function which is an abstract function. `Synthesizer1.java` in `pa1` provides a partial implementation of `run` which you will complete in this assignment.

2 A base synthesizer

In this assignment, we are going to work with `Synthesizer1.java` which extends `Synthesizer.java`. `Synthesizer1` includes two additional fields: `iterCounter` that counts the number of iterations of the worklist algorithm and `runTime` that records the actual running time of the synthesizer.

`Synthesizer1` provides a partially completed implementation of the `run` function. It first creates a worklist at the beginning, adds an initial AST into the worklist, and then enters a loop which implements the top-down search algorithm discussed in Lecture 3. **Your task in this assignment is to complete this loop.** Note that you may leverage two "helper" functions. One is `selectOpenNode` in `Synthesizer1`, which selects a bottom-left node with hole from the current AST. The other function is `expand` from `AST.java`. The implementation of `expand` is a bit involved but what happened there is nothing but replacing the hole with a concrete operator and creating its children.

`Test2.java` should give you a fairly good idea about how to run `Synthesizer1`. In particular, you need to first create a `CFG` object that essentially encodes a CFG. Then, you need to create a `Synthesizer1` object using this CFG. To run this synthesizer, you will need to create your input-output example and pass it to the `run` function. Once you complete your own `run` function implementation, run the `main` method in `Test2` and you should be able to see that it pretty quickly synthesizes a program. It should also print the value of `iterCounter` as well as how long it takes to synthesize the program.

Now let's take a closer look at the CFG. The `mkCFG` method in `Test2` essentially encodes the following grammar.

$$
\begin{array}{rcl}
df & ::= & x \\
 & | & \mathit{gather}(df, \mathit{newColName}, \mathit{newColName}, \mathit{oldColNum}, \mathit{oldColNum}) \\
 & | & \mathit{unite}(df, \mathit{newColName}, \mathit{oldColNum}, \mathit{oldColNum}) \\
\mathit{newColName} & ::= & \mathit{tmp1} \mid \mathit{tmp2} \mid \mathit{tmp3} \\
\mathit{oldColNum} & ::= & 1 \mid 2
\end{array}
$$

Here, *df* represents a dataframe: in the simplest case, it is simply the input variable x which is a dataframe. It could also be a *gather* function[1] or a *unite* function[2]. Both functions take *df* as the first argument, however, the remaining arguments are different. A *newColName* here is a string that refers to the name of a column in

[1] http://statseducation.com/Introduction-to-R/modules/tidy%20data/gather/
[2] http://statseducation.com/Introduction-to-R/modules/tidy%20data/unite/

the output dataframe, and an *oldColNum* is a column number in the input dataframe. This CFG only allows using three names, *tmp1*, *tmp2*, and *tmp3*, which are not particularly meaningful, though. Furthermore, this grammar only allows using two column numbers, 1 and 2. You can definitely enrich this grammar by adding more names, numbers, or even R operators, which is exactly what we will do next.

3 Use a different CFG

In this assignment, you will also test `Synthesizer1` with a different, slightly more complex CFG. You will do this in the `Test3.java` file. Similar to `Test2`, you can encode the grammar in the `mkCFG` function, and then create and run the synthesizer in the `test` method. We also print `IterCounter` and `runTime` after a program is synthesized.

More specifically, we will use the following CFG that has an additional *spread* function[3].

CFG:

$$
\begin{array}{rcl}
df & ::= & x \\
 & | & gather(df, newColName, newColName, oldColNum, oldColNum) \\
 & | & unite(df, newColName, oldColNum, oldColNum) \\
 & | & spread(df, oldColNum, oldColNum) \\
newColName & ::= & tmp1 \mid tmp2 \\
oldColNum & ::= & 1 \mid 2 \mid 3
\end{array}
$$

We will use the following input-output example.

Input (4×4 table):

	var	val	round	nam
1	22	0.1	round1	foo
2	11	0.2	round2	foo
3	22	0.5	round1	bar
4	11	0.9	round2	bar

Output (2×5 table):

	nam	val_round1	val_round2	var_round1	var_round2
1	bar	0.5	0.9	22	11
2	foo	0.1	0.2	22	11

Warning: consider setting a *timeout*, such as 1 hour, since it may take the synthesizer a while to terminate. You may also want to make sure the example you give to the synthesizer is correct. One simple sanity check is to *manually* come up with the desired R program and then follow what you did in Assignment 0 to check this program indeed produces the intended output.

4 Submission

Your Task. There are three tasks in this assignment.

- (70 points) Task 1: Complete `run` in `Synthesizer1`.
- (10 points) Task 2: Test your `Synthesizer1` on `Test2`. Take a screenshot of the output.

[3] http://statseducation.com/Introduction-to-R/modules/tidy%20data/spread/

- (20 points) Task 3: Write your `Test3`. Test `Synthesizer1` on `Test3` and observe its the behavior.

Canvas. Submit the following in one single .zip file:

- **Your code** including `Synthesizer1.java` and `Test3.java`.
- **Your report** in PDF with your screenshot in Task 2 and your observation and explanation in Task 3.

References