Team: OnTheCSide

Translator Report

The first post-midterm task we focused on was to clean up the convoluted code each of us had written. We split the remainder of the project by choosing tasks similar to the ones we worked on for the midterm project. For example, inheritance and method overloading prefaced well into SymbolTable building, while pre-midterm generics lead somewhat smoothly into class template specialization and two dimensional array building. Every Wednesday, we would meet after class for an hour to discuss our progress and delegate new tasks. We created a Trello board containing a list of items to be worked on, allowed us to assign them to team members, and checked them off after they were completed. The board allowed us to manage time well by providing a visual indication of what had been completed, how much work was left, and who to ask if we had questions about a certain section.

Since this is the second version of our translator we were more familiar with xtc and we had a better understanding of what needed to be done and how to implement our translator. We also made adjustments to how we coordinated as a group to keep track of our progress, which made it easier for all of us see how much more work still needed to be done. For the first version, it wasn’t until the week the translator was due that we started paying attention to how many tests we passed, so we didn’t know exactly how much more work needed to be done. This time, however, through use of Google Docs and Trello we were able to efficiently use the tests as benchmarks for our progress, which resulted in a better balanced workload.

Architecture

Translator.java is our driver class. It first directs TreeConverter.java to convert the Java AST to a C++ AST. While this conversion takes place, for each class parsed in the Java file, TreeConverter adds method and field information to a ClassDetail instance for that class (using ClassDetail.java). It also builds the symbol table, and for each definition of a symbol, it saves an instance of SymbValStruct which includes type information for the symbol (SymbValStruct.java).

CPPPrinter.java, an extension of xtc’s JavaPrinter.java, first uses HeaderPrinter.java to print out the header file for that class. Then it reads through the C++ AST to print out the .cc file. If it encounters a CallExpression node, it creates an instance of MethodOverloader (MethodOverloader.java) to return an overloaded version of the method call.

Once these procedures are complete, Translator.java generates a script to compile the generated files, which can be run by calling ‘./a.out’.

Method Overloading

Method overloading is implemented in two stages: (a) Method Declarations are altered during tree conversion and (b) Call Expressions are altered during file printing. For (a), we simply iterate through the argument types and append them to the end of the method name, then store that name in the class’s method list. For (b), we look up the arguments of the call expression in the symbol table, to find out what type they are. Arguments that are string literals, integer literals, selection expressions, and other call expressions, are handled separately. Once we discover the initial argument types, we append them to the method name and search for that complete name in the class's method list, as well as the superclass(es)' method lists. If we find a match, we return it as the overloaded call. If not, we systematically replace each argument type with its super type and search once more. This eventually returns the proper method call, and we print it.

Static Methods

We use the symbol table to discover if the caller of a call expression is actually the name of a class, in which case it is a static method of that class and we print it as such: __*<classname>::<methodname>(<args> ...)*. If there is no caller, and the name of the method is not 'init', then it is a static method in the overall test class, and we print it appropriately. Static methods are ignored when printing the vtable pointers and its constructor initialization list. Method overloading also supports static methods.

Revamped Constructors - 'Init'

We wrap each New Class Expression in a call to the class's 'init', which implicitly calls the constructors and init methods of all the class's superclasses and then runs the original constructor's code. This implementation helps avoid self-reference issues as well as 'upcasting' issues wherein the code passes a subtype of the required argument type.

Smart Pointers

To implement smart pointers, whenever we come across an array or a field of non-primitive type, we wrap the type of the field with a smart pointer.

Command line arguments

We need to access the command line arguments for two of the test cases (tests 22 and 23), which we access via smart pointers. The default argv array, however, cannot be accessed by smart pointer syntax, so in every main.cc file, we make a hard copy of the contents of argv and put them in a smart pointer array called argvPtr.

Arrays

While printing our C++ code we need to make sure to print out any array initializers as new instances of our self made C++ array class. This is done by simply replacing any of these initializers with new instances of our Array class rather than using the standard C++ arrays. All accessing of array elements keeps the same format as Java and the [] operator is overloaded in the Array class in order to have the same functionality.

The next step was to work out two dimensional arrays. In order to properly overload the [][] operator another class called Array2D was created. Any time during the printing of the C++ files, our printer checks whether a given array is one or two dimensional and instantiates the proper class. Accessing of two dimensional arrays is not changed during translation. Though one cannot directly overload the [][] operator in C++, we were still able to achieve the functionality by creating a new class called Access within our Array2D struct. The first [] access results in the calling of the [] overloaded operator in the Array2D class, which returns a type Access that in turn also overloads the [] operator and returns the proper item being called for. Access also overloads the -> operator in order to ensure that accessing the length field of the second dimension of the array is possible. In this way we were able to maintain our code's readability.

If we could start over, we would have met up more to work as a group. We spent most of our meeting time merging the code, because it was easier to discuss in person what sections were changed and added, and how those changes impacted the functionality of the translator and other's code. Despite more git usage, we still experienced problems with merging, conflicting functionality, and varying implementation methods. Ideally, we would have mitigated this problem by meeting for 4-5 hours per week to code together instead of just meeting to split work and merge code.

Given that our translator is primarily evaluated by the provided tests, we made sure that our translator passed them by checking that our translator gave the same output as the test files. The features our translator needed to support were checked by the tests, so, by passing the tests, we were convinced that our translator had correctly implemented these features.

Our translator does not have any limitations within the project requirements. It does, however, have limitations regarding translating any Java file outside the scope of our tests. For example, we do not support the translation of the use of Java libraries. We don't support translation for arrays larger than two dimensions. We also did not implement the ability to have multiple constructors within a class. Our translator does well within our given test files, but has a bit of a way to go if we want to extend its capability to translate absolutely any Java file.

Below is a table detailing the success/failure of each test case provided to us:

Test Number	Translator's output == Test file output?
1	TRUE
2	TRUE
3	TRUE
4	TRUE
5	TRUE
6	TRUE
7	TRUE
8	TRUE
9	TRUE
10	TRUE
11	TRUE
12	TRUE
13	TRUE
14	TRUE
15	TRUE
16	TRUE
17	TRUE
18	TRUE

Test Number	Translator’s output == Test file output?
19	TRUE
20	TRUE
21	TRUE
22	TRUE
23	TRUE
24	TRUE
25	TRUE
26	TRUE
27	TRUE
28	TRUE
29	TRUE
30	TRUE
31	TRUE
32	TRUE
33	TRUE
34	TRUE
35	TRUE
36	TRUE
37	TRUE
38	TRUE
39	TRUE
TOTALS	39/39