CSE 401 Project Report, Autumn 2020

Group ID: AC

Names: Joshua Ramirez, Adnan Ahmad

UW Net ID: joshur4, adnana2

Four score and seven years ago, we set out to make a MiniJava compiler. Overall would say the project was a success. We were able to implement every feature that was given to us in the project descriptions. So we were able to implement integer addition, subtraction and multiplication as well as assignments for variables of type integer and integer array. In terms of objects, our final MiniJava compiler can create new objects and arrays, call methods, and has inheritance and polymorphism. Control flow is another feature of our compiler that we added by implementing 'if' and 'while' loops as well as having boolean variables and expression. We were also able to have certain boolean operations like '&&' and '!'. Late into the project we wanted to implement the 'or' and 'super' expression; however, due to time constraints we decided to scrap the idea.

Throughout the project we tested extensively on each part and put tests in the 'test' folder. We also included a lot of tests in the 'SamplePrograms' folder. For the Scanner part of the assignment, we wrote many tests. Half of them were correct programs (so no error should have been printed), then the other half of the tests had an increasing number of unrecognized symbols. The goal of the error tests was to see if the Scanner would give the user the correct number of error messages. For parser, we wrote less tests, but we still tried to make sure that our visitors (that were printing stuff at the time) were working properly. So we wrote basic classes to make sure everything was good. For the semantics part of the project, we invested a lot more time into testing by trying to think of edge cases. We wrote some basic tests for each node e.g.

trying to test simple expressions, control flow statements, etc… We also tried to write more complicated tests for edge cases like polymorphism. We wanted to make sure there were no cycles, and that a superclass could be assigned to a subclass. We also tried to make sure that classes could override methods two layers deep e.g. C extends B which extends A.

However, we did not test enough which caused us to have some points be docked down on things like our program passing when an unknown type was being returned, or assigning variables to unknown types only under certain conditions. We also forgot that when a subclass overrides a superclass, the return type can be a subclass of the return type in the parent's method. There were language features that did not work. We fixed those issues and wrote tests to cover them.

For code generation, we tested a lot more heavily to avoid being docked down like in semantics. We wrote a lot of tests in 'SamplePrograms' after implementing each specific part in the project description. For example after finishing control flows, we heavily tested control flow. We ended up testing all the basic requirements for example regular expressions, boolean expressions, control flow, arrays, and inheritance. We tested a lot on making sure stack alignment was being implemented correctly by writing worst case scenarios. We also did some error handling e.g. arrays out of bounds. We also made sure to test that error handling was working properly by making sure arrays could not be assigned negative values, access negative indices, or access indices beyond the array's length. We also wanted to make sure our virtual tables were working as intended, so we created a test based on the slides in lecture which was assigning a superclass and a subclass to the same variable (One and Two). We finished testing and wrote testing scripts for both semantics and code generation to automate the process. We included these scripts in the main source directory.

We did not add any extensions or extra language features, but we did try to follow good programming practices. We heavily commented our code, wrote a lot of tests, and tried to reduce/simplify whenever possible. Commenting helped us debug and figure out what went wrong in certain places. We also added a lot of comments to our generated assembly code which helped us figure out where things like segmentation faults were occurring. For the code generation part of the project, we also tried using unique commands not mentioned in lecture e.g. ‘xor’ for the ‘not’ node and ‘movzbl’ plus ‘eax/al’ for the ‘less than’ node.

In terms of working on the project, we decided to do a pair programming approach. So for half the time one person would program the project while the other would lead. Outside of programming, we would have brainstorming sessions to figure out the best design for our compiler. This would usually take one to two days of us writing things out and weighing out pros and cons. Then we would make a rough schedule on when we wanted things finished by. This made our pair programming much easier by helping maintain a 50/50 split and making it easier for the leaders to guide the programmer in the right direction. We believe this was an extremely effective way of programming and while it did take a bit longer to complete projects, our final products overall had less errors than just programming separately.

Overall we think the project went extremely well and helped us expand our knowledge in terms of how languages are compiled. However, if we could go back we would have tested extensively from the beginning rather than doing it at the very end. Also we would have added a couple of extra features if given more time.