Mike Peterson
Independent Study: Compiler Construction
August 9th 2015
Advisor: Professor Fluet

Introduction:

The purpose of this Independent Study was to experiment learning Compiler Construction in an alternative way from the typical course that is taught at RIT. To do this, I followed a project outlined in Andrew Appel and Jens Palsberg's *Modern Compiler Implementation in Java*. The first 12 chapters of the book deal with teaching several topics and the project associated has the user create a compiler for the MiniJava language, which is a language similar to Java but having a more limited focus. The first half of the set of chapters read focused on the compiler front-end, while the second half dealt with actually translating the code so it can run. The following parts of the write up deal with what I worked on during each week during the Independent Study.

Week 1:

For the first week of the study, I read the first chapter of the book titled Introduction. As the title suggests, the chapter dealt with giving an overview to the sections covered in the book and the basic outline of implementing a compiler.

The project given at the end of the first chapter was a warm up project to get the reader prepared for the larger MiniJava project. The program itself was a straight-line program interpreter. The book gave 3 files, interp.java, prog.java, and slp.java. Out of the files, the user only needed to implement two parts inside of the interp.java file. The first part was implementing the maxargs function, which takes a statement and returns the maximum number of arguments for any print statement. To do this, the user needs to determine what kind of statement is being used and then recursively call maxargs to get a number to continue with. Once maxargs is given a PrintStm, a print statement, the number of arguments is returned to the prior level. Whichever PrintStm has the highest number of arguments is the number that gets returned by the entire function. The second part of the project deals with the interp function, which takes a statement and attempts to interpret it. To do this, I made a second function called interpStm, which takes a statement and a table. The function also returns a table. If the statement given is a compound statement, the statement is split into two parts and each of the parts is recursively sent to interpStm with the table given. If the statement is an assignment statement, the statement is sent to another function called interpExp, which takes and expression and a table. This returned an IntAndTable, a class given by the book. This keeps a list of ids and values. Since a new expression is found, the table is updated inside interpStm. Finally if the statement is a print statement, the table is sent to the printStatement function which returns an integer to be printed. This makes use of the lookup function which gets the value of the id in the table.

After implementing the Chapter 1 project, I read the second chapter of the book. This chapter was called Lexical Analysis which dealt with tokens and regular expressions. The MiniJava project started at the end of the chapter, but I ran into an issue when looking for the files. The website that Professor Fluet and I were using for the project was an outdated website which used the first edition of the book and did not contain the updated MiniJava project. It was not until after the meeting I had with Professor Fluet that we were able to find the project files given so that I could begin working on the proper project.

Week 2:

The first part of the work this week was making up the Chapter 2 project. This project asked me to create a lexical analyzer that would take MiniJava code and match up the code with a set of tokens. To do this the book offered me two options, JavaCC and SabelCC. I ended up choosing the JavaCC path, so I implemented the MiniJavaLexer.jj file which only had a test case and goal function to start. I was also given a Main.java file to run and test the case given. So I then made a list of tokens and list of skippable characters. When the JavaCC code is run, several Java files are created, which are then used by Main.java. After initial problems starting the project, actual implementation went smoothly and the test ran correctly.

I then proceeded to read Chapter 3 of the book, Parsing. The chapter focused on Context Free Grammars, Predictive Parsing, LR Parsing, and how to use JavaCC as a parser. This then lead into the project for Chapter 3, which asked the programmer to add on to the MiniJavaLexer.jj from the previous project.

In order to do parsing for the project, the programmer needs to understand the syntax of Mini-Java, which is detailed in Appendix A of the book. Since the tests I used were more expansive than the ones used for Chapter 2's project, I ended up adding some more tokens in the token section. I then created the actual parsing functions. These functions would start large and slowly parse through each section. These functions are: Program, MainClass, ClassDecl, VarDecl, MethodDecl, VorS (determines if the next parse is a Variable Declaration or a Statement), FormalList, FormalRest, Type, Statement, Exp, PSExp, PSExpPrime, MExp, MExpPrime, EQExp, EQExpPrime, BRExp, BRExpPrime, DExp, DExpPrime, AExp, ExpList, and ExpRest. Each of these functions uses one another to parse the code given using the tokens and whether or not any illegal tokens are used. It also determines whether or not the tokens are put in a valid order. I ran into one major issue while implementing this section. Every attempt to run the code would give me an error message stating that there was unreachable code. After discussing with Professor Fluet, we determined that the issue was that I was not handling left recursion. At the time, I only had a single Exp function. After breaking the function up into several smaller sections and adding prime functions, I was able to fix the problem. Using the test cases supplied by the book, I was able to get the code handling correctly.

Week 3:

At the start of the week I read Chapter 4 of the book, Abstract Syntax. This dealt with Semantic Actions, Abstract Parse Trees, and Visitors. From here, the project at the end of the chapter dealt with adding semantic actions to the programmer's existing code so that the parser is able to create abstract syntax.

The book supplies a lot of different files for the 4th project. To make it simple, all of the coding that I had to do was in the MiniJavaLexer.jj. The syntaxtree and visitor packages were already complete so all that needed to be done was add those packages to the project. A new updated main class was also given so that the result of the MiniJavaParser (which is created by compiling the MiniJavaLexer) is passed through the PrettyPrintVisitor (which is in the visitor package) to print out the parsed code given. Instead of rewriting the functions made in MiniJavaLexer.jj, I merely had to rework them to make use of the classes supplied by the syntaxtree package. I discovered how to work the lookahead operation given by JavaCC which allowed me to remove all the prime functions for exp while still handling left recursion. Each function would return a class until ultimate a program class is sent back to the main program. Each part of the parse would be related to a class given in the syntaxtree. Ultimately this part of the coding ran very smoothly, especially once I discovered the proper use of the lookahead operation. From there I used the test cases supplied to ensure the code's correctness.

I then went on to read Chapter 5, Semantic Analysis. This chapter dealt with Symbol Tables and how to properly Type Check the MiniJava language. At the time of the week, I was having trouble understanding how the book set up the visitor class given to me in Chapter 4. It was until a discussion with Professor Fluet and looking at the PrettyPrintVisitor more carefully that I was able to understand how to implement the type checking.

Week 4:

I started this week by starting to implement Chapter 4's project. This project dealt with adding type checking to visitors so that error messages would be thrown for mismatched types. With error messages, it became extremely important to make sure that error messages were as detailed as possible. This quickly became apparent in testing code, as initially I was getting errors for correct types and had to track down the issues in the code based off of the message. This chapter did not supply any new code, so I ended up editing TypeDepthFirstVisitor file to do the type checking. The final product would have me add a couple of types of symbol tables. I also created a method for printing error messages and exiting the program called errorMsg. While the basic functions were already supplied to me by the book, I had to overhaul them pretty heavily for type checking. These functions return a type, but before I changed them each function returned nothing. I also had to create functions that would crawl over the basic program supplied to get method names and class names so that when a piece of code is activated before definition, an error would not occur. These methods were checkMain, checkClass, checkVariableDecl, checkMethodDecl, and checkFormal. From here the old visit functions were used with updated code to check the types using the instanceof operation.

I also had to implement the symboltable package with a way to handle symbol tables. While there was a skeleton in the book to help start making a symbol class, I ended up implementing the code in a different way. I created three different classes which were SymbolTable (which held a hashtable of string and a ClassTable to relate to), MethodTable (which had a hashtable that kept method names, their returning types, variable declarations, and function parameters), and ClassTable (which had a hashtable that kept class names, variable declarations, and method declarations). These symbol table classes were very important when it came to trying to determine type checks for custom classes and returning methods. Figuring out how to implement this portion of the project gave me trouble and caused me to spend more time on the project than expected. I was hoping to get a start on the sixth chapter project, but this ended up pushing that project into week 5.

While I didn't start the project for Chapter 6 until week 5, I did end up reading the chapter. This chapter was called Activation Records and taught Stack Frames. This chapter is an important chapter in the book as its the first chapter that goes into how to generate code given rather than checking to see if the code given is contextual correct.

Week 5:

The Chapter 6 project caused a lot of interesting problems for me and took me most of the week to finish. While I initially was able to work on some of the Chapter 7 project, I quickly discovered I messed up some of the implementation for Chapter 6, causing me to go back and rewrite a good portion of the code. I also had to go back and change a few little details of the Chapter 5 project. In particular, while I had the symbol tables of Chapter 5 implemented, I had them listed under the wrong package name when the book needed it to be of the symbol package (as there were some files given that implemented said package), so I had to go back and change it Symbol and add the Symbol class to the package as well. Luckily the Symbol Class was mostly supplied in the pages of Chapter 5 so I only had to change a couple of inconsistencies throughout the package.

As mentioned before, Chapter 6 supplied some new files that required the use of the Symbol Class and package. The files given were put into the new Temp and Util packages. The first part of the project in this section dealt with implementing the Frame package. While I implemented the three classes (Access, AccessList, and Frame), these classes were very small and serve to be extended by other classes in the future. On top of that, bits of the code were detailed in Chapter 6 itself. This is where the second part of the project was implemented. The book asked the programmer to create a package that would extend the Frame package to compile. In particular, the book gave the user two options, Sparc or MIPS. I decided to implement a Mips package. This package contained three classes: InFrame, InReg, and MipsFrame. InFrame and InReg were small classes that were very quick to implement. The MipsFrame was larger and extended the Frame Class. As such four methods were used. The first was the MipsFrame constructor that would take a label and a boollist and keep them stored. The next was newFrame, which just called the constructor. The third was allocLocal which returns a reference to either a Frame or a Register depending on the boolean given. The final method of the project was the allocFormals method which would create an AccessList off of the boollist of formals supplied. One of the issues I ran into this week, was the fact that there was no real way to test the code. This would become an occurring problem with the next couple of projects in the book due to the fact they all relied on each other without a real way to test everything. I personally believe that if the book had supplied some form of testing, it would have been a little less confusing to implement.

While the project work this week was tough, I did still read Chapter 7 from the book. This chapter was title Translation to Immediate Code. This dealt with Intermediate Representation Trees, Translation into Trees, and Declarations. While the chapter itself was not too challenging to read through, the project for this part was a little confusing until I realized I had implemented some of the MipsFrame incorrectly for Chapter 6.

Week 6:

The project for Chapter 7 dealt with designing a set of visitors to translate a MiniJava program into the intermediate code discussed throughout the chapter. To do this, the book supplied the user with a Tree package, that held a pretty large number of Tree Classes. I did end up changing the name of the Exp class in the Tree package to Exp1, since there is also an EXP class in the same package. On my end, I made some updates to the InFrame and InReg classes in the Mips package. I also added a couple of methods supplied to me to the Frame class. This caused me to update MipsFrame class since it is an extension of the Frame class. These functions were originally not added until a week later when I realized there was an issue with the compiler. The first function I added to the class was the externalCall function which makes an external call on the tree. This takes in a String and ExpList and returns an Exp1. While some of the code was offered in the pages, I had to add some code to keep track of the labels. I also made temporaries for all the registers found on a MIPS machine. The final function added was the procEntryExit1 function. This take in a statement and returns a new body statement. To determine this, an access list is created from the formals of the frame. While the list remains full, a sequence of statements is built and creates the new body statement to return. After I finished up the new parts of the MipsFrame class, I went to implement the Translate package. This package includes the following classes: Cx, DataFrag, Ex, Exp, Frag, IfThenElseExp, Nx, ProcFrag, and Translate. While the Exp class was mostly shown in the book, the Cx, Ex, and Nx classes that extend it were mostly left to me to code. While that didn't mean much with the Nx class (its an expression that holds no value), each of these three classes had me write a method for unEx, unNx, and unCx. I also had to implement sections of the class IfThenElseExp where the unCx, unNx, and unEx methods were not defined by the book. I then worked on the DataFrag and ProcFrag classes which were just extensions of a simple Frag class and were very short. Finally, I created the Translate class which made use of the Frag classes.

Like previous weeks, this chapter project took most of the week, so I was not able to get started on the next chapter project. Fortunately, there was no project for Chapter 8, so I was able to read both Chapters 8 and 9. Chapter 8 was titled Basic Blocks and Traces and dealt with Canonical Trees and Taming Conditional Branches. Chapter 9 was titled Instruction Selection and dealt with Algorithms for Instruction Selection, CISC Machines, and how to use the Instruction Selection for the MiniJava compiler.

Week 7 and Week 8:

As I stated earlier, there was no project given for Chapter 8. However, there was code given to me by the book. The code given was the Canon package which contains four classes: BasicBlocks, Canon, StmListList, and TraceSchedule.

I began working on the project outlined in Chapter 9, but the project itself took me two weeks rather than the one week that I had been hoping. The project dealt with Instruction Selection, which meant translating instructions into assembly instructions using maximal munch. Since I choose to implement Mips, this meant translating into Mips. Not only would this project use the package given for Chapter 8, but it also gave me the Assem package which contained the following classes: Instr, InstrList, LABEL, MOVE, OPER, and Targets. I then went on to work on the new Codegen class for the Mips Package. While the basic outline of this code was given in the book, the outline itself was not for Mips but rather the Jouette class. The Codeine class itself holds a frame and two InstrLists. These InstrList would get new Instr added to them via the emit class, which would be called by the munch methods. There was also a method taken from the book called L, which was a shortcut to making TempLists to save space when creating a large TempList (It takes a Temp and a TempList so that it can append the two together and send it back to the user). From here I started working on one of the two very large functions in the Codegen class. munchStm would take a tree Stm and depending on what kind of Stm was passed in, a unique instruction would be sent to the emit function to add to the instrList. There were five acceptable types of statements here that would not throw an error: MOVE, EXP, JUMP, CJUMP, and LABEL. In the instance of a Stm being a MOVE, I needed to determine whether or not the Stm destination was to memory (MEM) or to a register (TEMP). If the statement was sent to a register, I simply sent a new instruction via the OPER class that had the destination at the register and the source would be a munchExp. If the MOVE statement had a definition in memory, a little more work was involved. Before creating an OPER, I had to determine if the expression given was either a CONST or a BINOP. Depending on the situation, a different OPER setup was sent to the InstrList. If the Stm ended up being an Exp, the statement would just be sent to the munchExp method. If the Stm was a JUMP or a CJUMP, a special OPER would be created that would have also have the targets of the JUMP listed. Finally, if the Stm was a LABEL, instead of creating an instruction using OPER, we would create a new LABEL instruction that had the statement's label.

The next method was the munchExp method which took an Exp1. First I had to determine whether the expression was an instance of the following: CONST, NAME, TEMP, BINOP, MEM, or CALL. If it wasn't one of those, an error would be thrown. If the expression was CONST, a new temporary was returned as well as the corresponding OPER instruction. If the expression was a NAME, the label would be loaded. If the expression was TEMP, a little investigation would be required. If the temp was a Frame Pointer, we needed to create a new temp and instruction that would take this into account. If this was not the case, the current temp was sent back. If the expression was a BINOP, some calculation was required. First, I had to determine what kind of bin operation was used. From there I needed to determine if any of the sides of the bin operation had a constant, as the constant would be used at the start and then the rest of the expression would be sent through munchExp. If neither sides were a constant, then both sides were sent to munchExp. If the expression was an instance of MEM, we needed to load a memory location into a temporary. Different actions would occur depending on whether the memory location loaded was a constant or a bin operation. Finally, if the expression was a CALL, a list of argument temporaries would be needed for the new function. To do this, I created a munchArgs function which took the number of arguments and an ExpList of the arguments. From here we would move the arguments into the appropriate registers. Finally the MipsFrame would be returned. The final method implemented in Codegen was the codegen method which return the Instruction List of a Statement given.

In addition to Codegen class, I also had to make some more updates to my Frame and MipsFrame classes. In particular, the book had me extend the Frame class to TempMap. Furthermore, using the Temps I had made in the previous project for the Mips registers, I had to make some TempLists that defined the types of registers. I also had to create some functions to access certain registers. With the tempMap being extended, I had to put all the registers into a tempMap as well. I also had to make a TempList called returnSink as detailed by the book. The book also had me create a procEntryExit2 and procEntryExit3 methods. To keep them clean, I also created an append function which would append two instruction lists together. Finally, I created the codegen method to connect the entire chapter 9 project together which called Codegen class to get the current instruction list based off of the current code generation.

Although I was behind on the project for these two weeks, I did also read up on Chapters 10 and 11. Chapter 10 was titled Liveness Analysis and dealt with the Solution to Dataflow Equations and how to deal with Liveness in the MiniJava compiler. Chapter 11 was titled Register Allocation and discussed Coloring by Simplification, Coalescing, Precolored Nodes, Graph-Coloring Implementation, and Register Allocation for Trees.

Week 9:

Week 9 dealt with me finishing up the two projects given at the end of the Chapter 10. This chapter gave me three packages. The first package given was the FlowGraph package that only contained the FlowGraph class. The next package was the Graph package which had the Graph, Node, and NodeList classes. The final package supplied was the RegAlloc package with InterferenceGraph and MoveList classes. The first project asked me to implement the AssemFlowGraph class which would go in the FlowGraph package and extend FlowGraph. As the name suggests, the class would take a list of assembly instructions and make a flow graph out of them. Since the class is an extension of the FlowGraph class, three methods had to be implemented: def, use, and isMove. These were pretty simple and pretty much looking up the Node supplied in the hashtable in the class and returning the instruction (or in the case of isMove, is the instruction an instance of MOVE). The main part of this project was the AssemFlowGraph method which takes a list of instructions. The aforementioned hashtable is created and each instruction is added to it with a corresponding node. I also kept a secondary hashtable in case the instruction was a label. This becomes important in the case of a jump. From here the edges needed to be added to each node. If an instruction has no jump, an edge is added between the current node and the next on the list, otherwise the edge is given to the jump. When the loop is done, the AssemFlowGraph is created.

The next project for Chapter 10 dealt with Liveness. As such, the Liveness class was created in the RegAlloc package. The class extends the InterferenceGraph and keeps track of the liveness Flow Graph. Since the extension of the InterferenceGraph, three methods had to be implemented: tnode, gtemp, and moves. The functions were small and relatively simple. The main functions was the actual Liveness function which would take a flow graph and given a specific node to determine when that node is live in the global live map. To do this we have to determine when a node is live in and live out. To do this I had to calculate the live intervals of a node and add interference edges.

In addition the project, I also read the final chapter of the Independent Study, Chapter 12. This chapter was called Putting It All Together. This dealt with the final touches to getting the compiler project to run.

Week 10:

This week dealt with me attempting to finish up the eleventh and twelfth projects detailed in the book. Despite my best efforts, I was unable to get the entire project running as requested by Chapter 12.

The Chapter 11 project dealt with adding two new class to the RegAlloc package. They were the RegAlloc and Color classes. While the method names for the classes were given, the actual implementation was all up to me. The way I choose to implement RegAlloc and Color was slightly different from some of the pseudocode suggested by the book in Chapter 11. Implementing the tempMap function was short, as it just took the color map determined by the register allocation and returned the tempMap from it given a temporary. The RegAlloc function was a little more intensive. First I created a list of registers and unused registers which caused me to add some more functionality to the Frame and MipsFrame classes. From there I created a AssemFlowGraph with the given list of instructions and an interference graph by sending the AssemFlowGraph to Liveness function. From here I added edges to all the register nodes inside the interference graph. I then sent the newly edited interference graph to Color Class as well as the frame and a list of unused registers. After retrieving the color map, the function then checks to see if there was a spill. If there was, then the program runs an error message. I then worked on the Color class which had a pretty massive coloring function. The function takes an InterferenceGraph, a TempMap, and a TempList of registers (in this case I chose unoccupied registers). To initialize the coloring, I needed the maximum number of colors, which would be the number of unused registers. From here I take out the nodes corresponding to the registers and precolor them. Afterwards, a loop is done in attempt to color each node in such a way that no two adjacent nodes share the same color. To do this, coalescing is used in an attempt to share colors and registers. While programming this initially I chose to code this without looking into the book which ended up psuedocoding most of the functions that I should use. When I went to go onwards with the project, I realized that my Color function was incorrect and restarted it using the functions supplied by the book. Due to this, I was still unable to complete the entirety of Chapter 11 by the deadline that Professor Fluet and I set. The following functions were the ones the books recommended I attempted to add in time: Build, AddEdge, MakeWorkList, Adjacent, NodeMoves, MoveRelated, Simplify, DecrementDegree, EnableMoves, AddWorkList, OK, Conservative, Coalesce, Combine, GetAlias, Freeze, FreezeMoves, SelectSpill, AssignColors, and RewriteProgram. I was able to get some of these functions ready for the RegAlloc portion but not the Color portion. It should be noted that instead of adding these functions to the Main file as asked for in the book, I decided that they should be used in RegAlloc and Color instead, since that what the project asked to implement.

Conclusion and Further Work

Given more time, I believe I could have not only completed the Chapter 11 portion of the project but also made some headway on the Chapter 12 project which actually involved testing Mips portion of the project using SPIM. I believe that the book did a very great job of teaching the reader on Compiler Construction, I think the one major disadvantage to following the project was the lack of testing in later chapters. This is not particularly the books fault however, as most of the later sections rely heavily on each other and a lot of work is needed to be done before testing can commence. I think this would have become apparent once I got the Chapter 12 project working, as I personally believe a lot of the test programs would run into errors that would then prompt me to check the compiler I have constructed. According to Professor Fluet, in the Compiler Construction class, a way to test the code early would be to have the other parts already working but unreadable to the student. This allows the student to connect their portion of the code into a compiler and test to see if their code is correct. While I think that the book at times was very useful at describing the topic of the chapter, sometimes I thought the actual project detailed at the end of the chapter was a little ambiguous. Some chapters really did a great job walking the programmer through coding the program and usually only supplied code when the classes were small and simple. Overall, I believe that the Compiler Construction Independent Study was a great experience and I look forward to taking more classes in the Language and Tools Cluster.

Statistics

Estimated around 3600 lines of code done for the entire project
Estimated amount of time working on the project is about 100 hours total

References:

Appel, Andrew W., and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge, UK: Cambridge UP, 2002. Print.