Compilers Project
December 15, 2019
Jonah Grier

Compiler Final Report

Introduction:

Over the course of this semester, our project was to design and build a compiler for the language: MiniJava. The MiniJava project had four major components: Scanner, Parser and Abstract Syntax Tree creation, Semantics and Type Checking, Code Generation. Each part required a whole new branch of logic and testing. This report will tackle each of the four components, mistakes, code tests, and any missed pieces.

Additions: I tried to add Double as a type to my minijava project. I was able to define a double in regex, scanner token, create a grammar production rule (terminal/nonterminal), and add all required AST Nodes and types, however, I was not able to finish adding it in the code generation.

Part 1: Scanner

Looking back, the implementation of the scanner was by far the easiest part of the project. However, this first part required understanding of how and what the .CUP and .jflex files work. Additionally, both files look and act as standard .txt files so there was very little guidance on mistakes.

To begin, the .flex file was the true beginning of the project. By specifying all tokens of the MiniJava language here, it allowed me to scan through a .txt or .java file and return each tokens "toString" value. For instance, the left curly bracket '{' would be replaced with LCBRACKET in text on the eclipse console. This immediate 1:1 representation from keyboard symbol to English string was the start of the project. Additionally, I used REGEX to specify whitespace, comments, integers, and identifiers such as trying to specify a new word to store an int value: int cheese (<- - "cheese" here would be the identifier). Other than understanding what was happening, there were not that many mistakes.

Scanner Related Links:

1. https://jflex.de/manual.html

Part 2: Parser

On the other hand, the .CUP file became littered with issues even as I got further and further along with the project. The .CUP file is where the grammar, and syntax starts to play a role. Once the tokens are specified, the .CUP file takes them and breaks them into terminals and non-terminals. Terminals are objects that cannot be broken down any further. These are symbols such as the plus sign (+), the Comma (,) a right curly bracket (}), or even larger

words such as ‘RETURN’ or ‘BOOLEAN’. The non-terminals are the ‘production handles’. They organize the flow of how our programs are broken down into smaller components.

In addition to having a lot of trouble getting the grammar to make sense, it wasn’t immediately clear how the language was actually. Like java, whitespace is meaningless so long as you have the correct “::=” symbol. Though I didn’t find it until later on, I forgot to set a couple token precedence’s which had significant impacts to part 3. Additionally, I had to re-write a couple production handles such as the Formal / FormalList. It turns out, when you are trying to specify the parameters in a method there is a HUGE difference when there are multiple arguments being passed and a comma is present. I was also required to change how I stored any “List” production rule, as I realized that there was overwriting to variables previously stored happening. In a perfect world, if the language found multiple instances of a specific production it would store them into a list, rather than overwrite them in previous result. An example of this can be found on line 155 (VarDeclList) in the minijava.cup file.

Lastly, the best part was hooking up our newly made grammar into the PrettyPrintVisitor and watching as it printed out our tokens.

.CUP File Link:

1. http://www2.cs.tum.edu/projects/cup/examples.php

Part 3: Semantics and Type Checking

Semantics and Type Checking was by far the hardest part of the project. Most of all my mistakes and displeasures came from this part. In this stage, we were required to start piecing together the parsed grammar creating symbol tables that store all the information about a specific class, method, and variable. Looking back, creating a symbol table isn’t really all that difficult, and by using the visitor classes provided helped tremendously. However, trying to build the symbol table and the correct types of each method and variable was overwhelmingly difficult. I ended up adding a type field to my ASTNode, as well as instance variables to each node that (…well, should) have a type. For example, in both the True/False nodes I added an instance variable “public Type type” and set it’s type equal to a new BooleanType() in the constructor. This allowed me to then store the AST nodes type correctly in the symbol table and allow me to type check correctly.

Constructing the actual symbol tables was also a nuisance. It took a long time to figure out what double-dispatch was truly doing, and how it was basically just allowing the program to venture deeper into the source code tokens (more-or-less). I also ran into major issues not setting the current class and current method before adding to a symbol table. This effect than had me at a loss of words later in the code generation when I realized my symbol tables were not as populated as I had originally thought.

I also modified the TypeCheckVisitor to return a type, rather than be void like the other visitor classes before it. By returning a type, I was able to type check an actual operator such as “Assign” and use it for debugging purposes with lots of System.out.println()’s. At the end of the day, I don’t think it’s required for it to be returning any value, however, I found it useful for debugging.

Semantics and Type Checking Links:

1. http://alumni.cs.ucr.edu/~vladimir/cs152/assignments.html
    - Specifically the "Assignment 5" portion of this link. It provided some skeleton code for Class, Method, and Variable

Part 4: Code Generation

The code generation portion, per say, wasn't the toughest part. It took a long time to wrap my head around the different registers, when and where they should be used, the differences between x86-32 and x86-64, how different operating systems use Intel vs AT&T, etc., etc. Unfortunately, while this part wasn't as difficult, since I already had the hang of using visitor classes and knew exactly what I needed to get done, this is where I started noticing all the errors I missed in other parts. This led me to going back to different parts of the project, exhausting more time than I had originally planned. As such, I was only able to get the basic printing (System.out.println()) and addition to be generated. While I know there are still multiplication, subtraction, conditionals, method calls, and many others, I have succumb to the pressure of adding more.

In terms of building the code generation part, I used another visitor class to help guide the process of getting all the symbol table node information. I used a BufferedWriter to write to a specified path (a new directory I built to store code generated files). Additionally, it made more sense to me to create private functions that were designed to do one specific thing such as popq, or movq. I then could just call the function I wanted to use and pass in the required register string and number if needed. This can be found at line 370 in the CodeGeneration.java class. These private functions were overloaded so that they could be called in different ways. For example, there are two ways to add in assembly. You can use: addq->%register1,%register2 or addq->$num,%register. I was then able to use another overloaded function called "AssemblyBuilder" (found line number 437) that would piece together all the information provided and create the final string that would be written using the BufferedWriter. This slimmed down my code and removed a lot of the \n's and \t's that originally made it a bit confusing (at least to me).

Code Generation Specific Links:

1. https://cs.nyu.edu/courses/spring17/CSCI-GA.1144-001/lecture4.pdf
2. https://jan.seyler.me/easy_x86-64/
3. https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Final Words:

This project has definitely been the toughest, most challenging, and by far longest project I've ever done. It was at times nail biting, head scratching – hair pulling, frustratingly challenging. And while I wasn't able to accomplish everything I wanted too, I am content with the progress I made and happy that I was at least able to get some code generation done.

Lastly, the next couple parts attempt to explain the *"especially tough parts"* that had me working on this right up to the final deadline.

Especially Tough Parts:

1. Adding types to AST Nodes
    1. From the get-go, I knew I needed to add types to at least a couple AST Nodes so that I could somehow compare the types during the type checking phase. This was all dandy and fine until I got to the type checking and realized that I was comparing an AST.IntegerType@023423 to a null. It took many, many, many hours until I realized when I was calling .type on an object it was referencing the super class: ASTNode's type (which is null since all the other AST nodes inherit from it). What I forgot to do was set the instance field in the specific class so that it would allow me to get the type of that object, not the type of the parent class. (duh).
2. Setting current class when constructing the symbol tables
    2. Another huge blip I had was the symbol table construction. I ran in a huge circle trying to figure out why when I added a specific method it wasn't there when I went to print it from a specific classes symbol table. It wasn't until I started doing line by line print statements that I realized I was in the wrong class. To add to the headache, I originally had an "Added method to class!" print statement that was inside a loop. I was dumbfounded seeing that the loop was being entered and the method was being added, yet no method was found.
3. INTLITERAL in .CUP File
    3. I'm still not entirely sure about a piece in the .CUP file. When I go to define what an Expression can be (line 253) the second rule defined: IntLiteral has some troubles. When I specify what the IntLiteral nonterminal is (line 347) I am required to first cast the arg1 to a String before being able to use Integer.parseInt(). I'm not sure why it requires me to cast as a String, the error message indicates that the argument is of type object… But I'm also not sure how that could be because INTLITERAL is specified through regex.
        - Casting to a string prior to changing it to an Int was the only fix, (thanks random lucky search to StackOverflow)

A couple other links I used throughout the project for help on what to do/where to go involved some previous compiler design githubs. Although all the completed git projects differed drastically and had obvious and subtle differences. However, they still provided forward motion.

1. https://github.com/hanselfmu/MiniJava
2. https://github.com/munkhbat1900/minijava