μC

The lightweight C compiler
for the Java Virtual Machine

Group Y
Jasdip Sekhon
Khuong Nguyen
Steven West

Introduction

This report documents our progress and the decisions we made for the class's compiler project. We used ANTLR to help with generating the parser and scanner, as well as borrow some of the code files we used in our class assignments.

We decided to call our language µC, or micro-C because it's basically an even more stripped down version of Small-C. In the next section we will detail the functionalities of the language.

Language and Jasmin Object Code

Our language implements many of the basic functionalities of C, with a couple creative liberties thrown in. We can't use the standard C libraries for IO, so we just defined some read/write functions as being part of the language.
We also don't have access to Malloc/pointers, so to overcome this we just changed the way that arrays work so that the expression used to define their size is evaluated and allocated at runtime (whereas in C, when not using Malloc they must be defined at compile time).

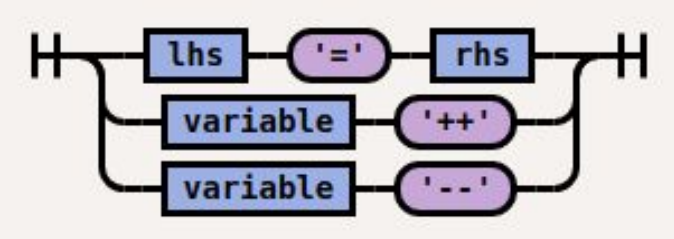
```
;
; FUNCTION fib
;
.method private static fib(I)I

.var 2 is arr [I
.var 1 is arrLen I
.var 3 is i I
.var 0 is index I
    iconst_0
    istore_1
    iconst_0
    istore_3
;
; 020 intarrLen=(index<=1)?2:index+1
;
    iload_0
    iconst_1
    if_icmple   L023
    iconst_0
    goto    L024
L023:
    iconst_1
L024:
    iconst_0
    if_icmpeq   L021
    iconst_2
    goto    L022
L021:
    iload_0
    iconst_1
    iadd
L022:
    istore_1
;
; 021 intarr[arrLen]
;

    iload_1
    newarray    int
    astore_2
```

Grammar and Semantics

Similar to C and C++ syntax, variable declarations can be done anywhere in the code, and separated or combined with assignment. For example, we can simultaneously declare an integer A and set the value to 10, or declare an integer B first, then set B to be equal to 5. Both are working syntax. Our compiler can also easily increment or decrement a variable, as well as assigning it to another variable.



We also implemented control statements such as the conditionals if/else-if/else (With an unlimited number of else-if options), and switch and cases statements. Our loop statements include do-while, while, and for loops. One such example can be seen below as a railroad diagram generated by ANTLR. Using it we can follow the logic of how a for loop is implemented, starting with “FOR” and open parenthesis, then it can accept and optional statement (usually used to assign initial value for variable, but can be any statement), followed by a semicolon then an expression used to determine whether to continue with the loop, another semicolon, then one more optional statement (this is normally used for incrementing a variable, but can be any statement), then a closing parentheses.

Our language also has optional modifiers for variables used to denote accessing elements of an array. Relational and comparison operators allow functionalities like addition, subtraction, multiplication, division, AND, and OR.

Sample Programs

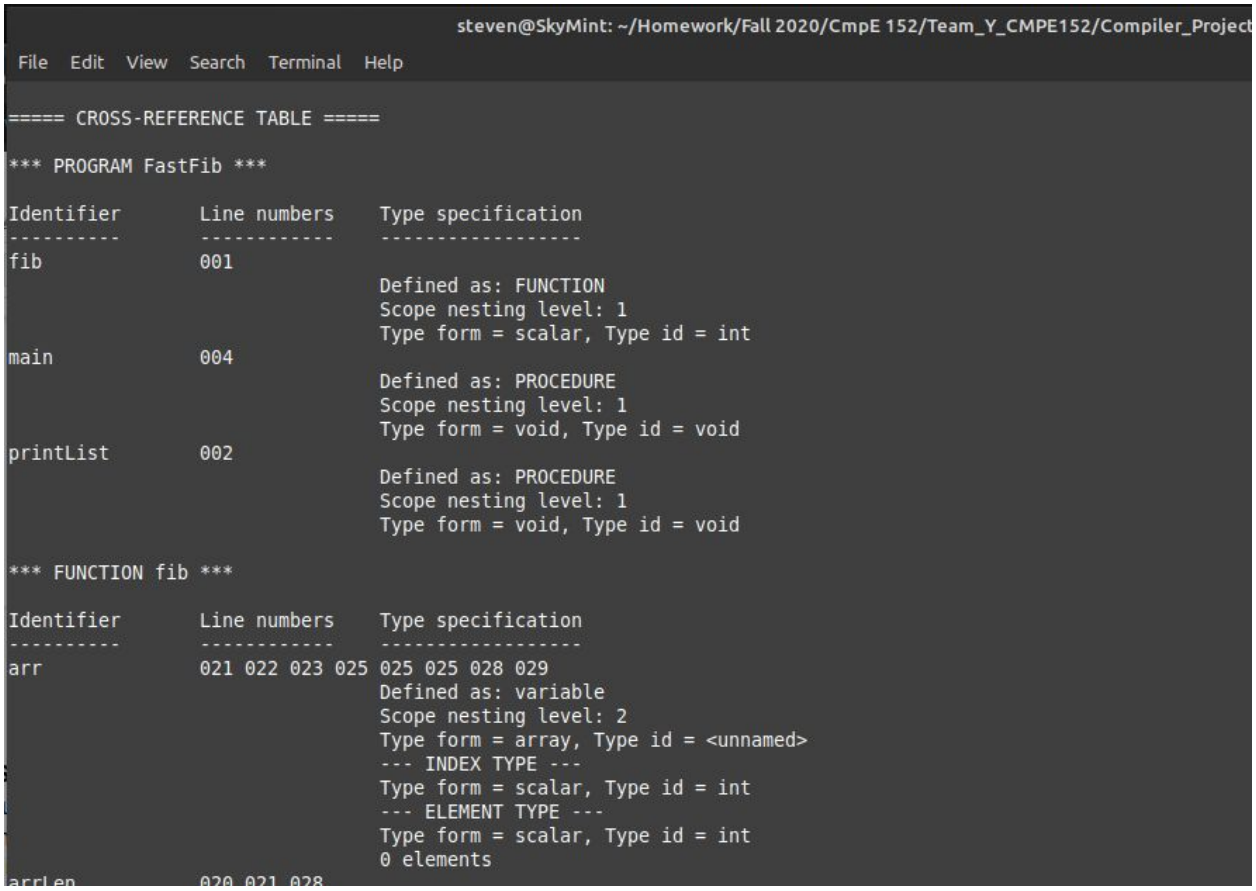
We wrote several sample programs to test various things. We had two variations of Fibonacci programs where the user enters a number and it calculates that Fibonacci number. One used recursion and tested function calls/recursion, and the other used dynamic programming and a dynamically-allocated array to test that arrays worked properly, and that we were able to dynamically allocate them based on a runtime value.

Then to test the full potential of the language and that everything was working properly, we converted the professor's favorite Hilbert example into μC, and ran it. Everything worked as intended, though lost a little bit of precision since we're using 32-bit floats instead of 64-bit doubles. But the result was still very close to what was expected.

Compilation

We are able to compile all of our test programs correctly, and we also added a few new semantic errors to handle situations that were not present in Pascal.

Since C requires a 'main' function, we added an additional check that checks to make sure that it is present, and if not, warns the user 'main function not present.' Pascal also declares and defines functions at the same time, whereas in C, functions can be declared and called, and then defined later in the program. So we added an additional piece of data/check that checks to see if the function has been defined, and if not warns the user 'function [function name] has been declared but not defined'



Changes and Challenges

Some changes that we made when adapting the compiler from Pascal to C are:

- Allowing for separate declaration and definition of functions
- Allowing variables to be declared/assigned separately or together, and be placed anywhere in the program instead of just at a “declaration section”
- Making arrays dynamically allocated at runtime due to lack of access to malloc
- Adding “else-if” to if statements, which Pascal does not have

Some (but not all) of the challenges that we faced were:

- Dealing with scoping
    - In C/C++, each loop and if/else statement is it’s own scope.
    - However this would be difficult to do with the way the JVM handles local variables, so instead we opted to just make those statements part of their parent’s scope
- Extracting the program name
    - Pascal has “Program [Program Name]” in the program itself but C doesn’t have this

    - Jasmin/Java needs the program name for naming static variables and compiling
    - When we invoke the compiler, we extract the name of the file we're compiling and treat that as the program name as a faux identifier.
- Debugging the code generator/Jasmin code
    - When an error occurred, it was difficult to find where the error was
    - Rather than running and crashing, Java just wouldn't run the program at all
    - The error messages that it gave were not very specific and didn't really tell WHERE the error was

Conclusion

There were plenty of challenges we faced in this project, not limited to just syntax and errors. Making a compiler is a hard task, which ANTLR made significantly more manageable. We were satisfied with the end result, knowing that the sample programs we tested were successful and we even got the Hilbert program converted and working with the compiler. While there were bumps throughout the process and limitations due to our programming knowledge, we know that this project helped us understand another complex topic and in the future this will help us prepare for more complicated systems.

Instructions for building and running our code

A duplicate set of these instructions can be found in our submitted zip file with our code in the Build directory and Outputs directory labeled “INSTALL” (Those are actually the original instructions) but here is a copy.

Note: These instructions and the associated makefile were developed for use on Linux with g++. It *should* also work on windows if using “Windows Subsystem for Linux” but has not been tested. I have no clue how it will fare on a Mac.
I have tried to package everything needed (runtime libraries, etc. excluding the c++ compiler itself and Java) into the project, so that it should be portable, but this also has limited testing.

Building/Compiling the compiler:

All of these steps will be done using the makefile located in the Build directory. Using the terminal, navigate to Build so that it is the current working directory.

We’ve included a pre-built executable, and included the header files that were auto-generated by ANTLR. If you want to get rid of these and compile everything from scratch, run the following two commands: ‘`make clean`’ and ‘`make clean_antlr`’. This will delete all the compiled/generated files and require everything to be recompiled.

First, create the ANTLR header files that are auto-generated from the grammar file by running ‘`make antlr`’ which will invoke ANTLR and place the created files in `Source/generated_source`.

Once the Antlr files have been created, build the executable by running ‘`make`’ (‘`make release`’ or ‘`make debug`’ should also work). If the Antlr files have not been created before running this, compilation will fail.

Additional capabilities:

These are not strictly needed for getting the compiler to work, but can be useful for visualizing things.

If you want to view the syntax/railroad diagram for the grammar, you can run ‘make railroad’ which will create syntax/railroad diagrams located in `Build/output/uC`. I personally recommend looking at the Index.html version since I find it easier to look through.

If you want to view the parse tree for a specific input file, you can run '`make parse_tree FILE_PATH=<path to input file>`' where the path is relative to the base directory. Normally this would be `./Input/[filename].uc`

Running/Using the compiler:

Once the compiler has been built, navigate to the root project directory (the directory that contains the folder Build, Input, Outputs, Source, etc.)

To run the compiler, run '`./Build/Team_Y.app -compile <input file> [output directory]`'. This will normally look something like '`./Build/Team_Y.app -compile ./Input/Hilbert.uc ./Output`'.
This helps keep the directory structure a bit cleaner by not having random files all over the place.
If you want to run just the lexer/semantics checker without actually creating an output file, you can replace the `-compile` with `-lex` (which was actually just created for testing the semantics before we had the code emitter working, but left it in as a feature)

Converting the .j files produced into .class files for Java:

Assuming you used the instructions above, the produced .j file should be located in the Outputs directory. Navigate into the output directory using the command line.

There is a bash script in the Outputs directory that handles invoking Jasmin. Run '`./jasmin.sh [filename].j`' which will create `[filename].class`. Lastly in order to run the compiled class file, run '`Java [filename]`' (the .class at the end is not needed for Java to find and run the file)

Our compiled code and results are also located in the Outputs directory since the submission instructions said our results were needed. If you want to recreate them from scratch, delete them and recompile.
Note: Fib and FastFib outputs are not present because they require user input (and I'm not sure how to redirect that from the terminal).