FLOYD COMPILER MANUAL

APRIL 23, 2018

CpS 450

Italo Moraes

Contents

Introduction

The purpose of this manual is to document a Floyd compiler written by Italo Moraes. The first section will cover usage, which includes information on how to build and run the compiler. Next are the features with details on the features the compiler supports and their limitations. The features section is proceeded by the technical notes, which are the biggest chunk of the manual. The technical notes will cover how the compiler is organized, information about memory management, the runtime stack, I/O, and strings. The final section is a list of tests, their outputs, and whether they pass or fail.

Usage

To use the compiler, the project and its dependencies must first be built. The compiler can be built and ran either manually or automatically using shell scripts.

Manual Compilation

First, open a command prompt in the Floyd directory. Then, run the following commands:

```
make
./gradlew clean install
```

Make will produce an object file of the C standard library. This object file will be necessary in the linking process of Floyd programs. *./gradlew clean install* will remove any old generated files and build the project.

Manual Execution

To run the compiler, run the following command from a command prompt in the Floyd directory:

```
build/install/floyd/bin/floyd [-ds] [-S] [-dp] <Floyd Program >
```

The bracketed arguments are optional command line arguments that are supported by the compiler. Refer to the Supported Command Line Options section bellow for more information. *<Floyd Program>* refers to the path of a Floyd program. Running a Floyd program names "test.floyd" with no command line options is as simple as this:

```
build/install/floyd/bin/floyd test.floyd
```

If there are no errors in the Floyd program, this command will produce a test executable file. To run the executable, simply execute the following command:

```
./test
```

Compilation and Execution by Shell Scripts

Two shell scripts are provided. compileProg.sh will build the project and its dependencies, then it will run the compile and provide it the file name listed in the script. To change the file name in the script, open it with a text editor and edit the fileName= line so that the right hand side of the = has the desired file name. For example, if the file name is test.floyd, it would look like this:

```
fileName="test.floyd"
```

To run the script, make sure it has execution privileges, and run it like this:

```
./ compileProg.sh
```

The second shell script, compileProgs.sh, builds the compiler and its dependencies, but it also compiles all Floyd programs in the current working directory. To run this script, simply use the following command:

```
./compileProgs.sh
```

Furthermore, compileProgs.sh also accepts one command line option called "clean." When run with the "clean" option, the script does not build the compiler. It cleans the Floyd folder of assembly files, object files, and any other files generated by the gradle build. This is how to run the script with the clean option:

```
./compileProg.sh clean
```

Note: Any other variation of command line arguments (or lack thereof) will simply build the compiler and compile all Floyd programs in the current working directory.

After either one of these scripts is executed, at least one Floyd executable will be in the current working directory. To run an executable generated from a "test.floyd" file, for example, run a command like this:

```
./test
```

Supported Command Line

The following three command line options are supported:

Option	Description
-ds	Produces a list of tokens generated by the lexer and outputs it to standard output.
-S	Stops the compiler at the code generation step. Generates an assembly file of the form fileName.S but does not produce an executable.
-dp	Displays a graphical parse tree of the program. Also displays the stack trace of syntax errors.

To run the compiler with one of these options, simply add it as a command line argument before the Floyd file. For example, to compile a program with all three command line options:

build/install/floyd/bin/floyd -S -ds -dp <Floyd Program>

Features

These are the features supported by the compiler. A red asterisk marks a feature that has a limitation and a blue asterisk marks a feature with an extension. More information on these features can be found in their respective sections.

Feature
Local and instance variable declarations*
Literal, identifier, parenthesized expressions
Assignment statement
If-Then-Else statement
Call statement
Loop While statement
Method declarations with arguments and an optional return type
Method calls (with recursion support)
Multiple class declarations*
Null, me and new support for objects
Run-time null pointer checks
String support
Predefined in/out variables for input and output
Compile time semantic checks

Limitations

Multiple class declarations: Inheritance is not supported at this time.

Extensions

Instance variable declarations: Memory is dynamically allocated for instance variables.

Technical Notes

Tools

These are the necessary tools to build and run the compiler:

Name	Version	Website
ANTLR	4.7.1	http://www.antlr.org/
Gradle	2.2.1	https://gradle.org/install/
GCC	5.4.0	https://gcc.gnu.org/releases.html
(GNU) Make	4.1	https://www.gnu.org/software/make/
Java	8	http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Compiler Organization

The compiler has three key classes, each having an important step in the compilation process. The first class is MyFloydLexer. MyFloydLexer turns the characters specified in a Floyd program into tokens that are stored in a parse tree. The parse tree is a representation of the program in tree form, which is vital for the semantic and code generation steps that will be discussed later. MyFloydLexer also performs syntactic checks. With the help of MyFloydErrorListener, syntax error messages provide information like the file name, line number, character position.

The second key class is the SemanticChecker class. It has the task of catching and reporting several semantic errors along with their file name, line number, and character position. Furthermore, the semantic checker has the task of decorating the generated parse tree with different information that may be needed during the semantic and code generation phases. The parse tree is decorated using a listener approach. For example, when processing expressions, the type of the expression must be passed all the way up the tree for the semantic checker to perform type checks. The SemanticChecker class depends on the SymbolTable class to accurately catch semantic errors. The SymbolTable is a singleton class that contains a symbol table, which is a stack of all the declared variables, methods, and classes. The symbol table also keeps track of information like a symbol's scope or type. Without the symbol table, it would not be possible to catch semantic errors.
Here is a list of the errors the semantic checker must catch:

Semantic Errors
Use of undeclared variables

Attempting to declare an already defined variable, method, or class.
Parameter mismatch: type and number
Attempting to use an undeclared feature
Type errors

The third and final key class is the CodeGen class. This class generates assembly instructions as it walks the tree that was decorated by the semantic checker. The CodeGen class uses a visitor approach as it traverses the tree, which means there's an option to choose when to visit what nodes. Every rule defined in the ANTLR grammar must be accounted for in the code generation phase, because each visited rule must generate a snippet of assembly.
The CodeGen class relies on the TargetInstruction class. The TargetInstruction class uses a Builder design pattern to facilitate in the generation of code snippets. CodeGen keeps a list of TargetInstructions; this is the whole assembly program in list form. Once CodeGen finishes traversing the tree, the entire TargetInstruction list is dumped into what is called the assembly file.

A separate, but just as important part of the compiler is the ANTLR grammar. The ANTLR grammar is stored in the Floyd.g4 file and it's a collection of regular expressions defining the Floyd grammar. The grammar is vital to the compiler because it generates the scanner that's used by MyFloydLexer.

Memory Management

Memory is dynamically allocated for non int and boolean types at the moment of variable declaration. This is done through a C function call to calloc. The size of the chunk of memory allocated is calculated by: (the number of instance variables * 4) + 8. The extra 8 bytes of memory are reserved for future feature implementations. The first four bytes of the object will contain a reference to the parent of the object's virtual function table. The next four bytes will be used for reference counting to free allocated memory no longer in use. This means that memory is not deallocated at this time. Here is what an object with two instance variables (x an y) would look like:

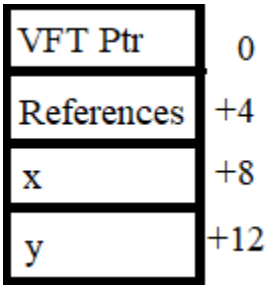


Figure 1: Object Structure

Runtime Stack

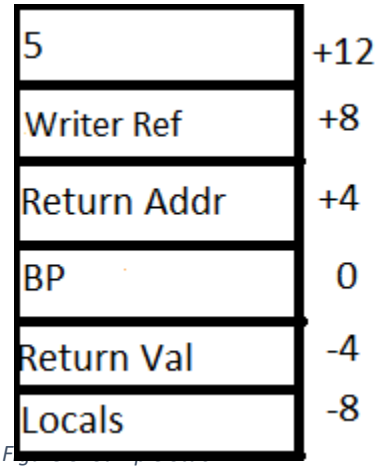
References to the stack are BP relative. There are reserved chunks of space above and below the BP, so method parameters begin at +12 and local variables begin at -8. Since it's easier to grasp the concept of a stack graphically, refer to figure 2 for an example. Figure 2 is what the stack would look like during the function call on line 7.

```
1  class Sample is
2    x: int
3    start() is
4    y : int
5    begin
6      y := 5
7      out.writeint(y)
8    end start
9  end Sample
```

Figure 2: Sample Floyd Program

Above BP there are two four-byte chunks of reserved memory. At +4 is the return address that will be used to jump back from the method call. +8 holds a reference to the caller object, which in this case, is out's type: Writer. At +12 is the parameter value that was passed into writeint, the number 5.

Below BP at -4 is a chunk of memory reserved for a return value. Finally, local variables are located beginning at -8 relative to BP. The reason behind such odd offsets for parameters and local variables should be much clearer now.



I/O

Input and output are implemented through the Reader and Writer classes defined in the Floyd standard library. These classes make use of the read and write syscalls by invoking them through the C standard library. The standard library function that calls read is Reader_io_read and the function that calls write is Writer_io_write. For example, the implementation of the write function is rather simple. It grabs the length of the given string using Floyd's length method and then loops through each character of the string, making syscalls to write for each character.

Strings

Strings are also implemented through the Floyd standard library. Similar to languages like C# and Java, Floyd's string built-in datatype is an alias of the String class. The similarities end there because behind the hood, Floyd strings are implemented using a linked list. Each character is a CharNode with a reference to the character it holds and a pointer to the next CharNode. String literal code generation is different from regular object code generation. Instead of simply calling calloc directly, a call to string_fromlit is made. String_fromlit is defined in the stdlib.c and it allocates a CharNode-sized chunk of memory for each character in the string and links them together to form a Floyd string.

Testing and Bug Report

Note: Testing with comma delimited inputs and outputs refer to multiple executions of the same program but with different input data which provides different output data.

Official Test Files

File Name	Result
assign1.floyd	**PASS**
breakit.floyd	**PASS**
loopy.floyd	**PASS**
testnum.floyd	**PASS**
cbasics.floyd	**PASS**
cchange.floyd	**PASS**
citerfact.floyd	**PASS**
cfact.floyd	**PASS**
cgcd.floyd	**PASS**
bchange.floyd	**PASS**
blist.floyd	**PASS**
bnulltest.floyd	**PASS**
bobjbasics.floyd	**PASS**
bstrbasics.floyd	**PASS**
Bstrlits.floyd	**PASS**

Phase 4

File Name	Input	Output	Result
assign1.floyd	[NONE]	15 2 14 -4 0	**PASS**
breakit.floyd	10	120 -120 120 -1 -2 1 -1 -2 2 -1 -2 3	**PASS**
loopy.floyd	[NONE]	3 2 1	**PASS**
testnum.floyd	0, 123, -55	0 9 9, 1 9, -1 9 9	**PASS**

Phase 5

File Name	Input	Output	Result
cbasics.floyd	[NONE]	0 10 20 0 10 -5 0 10 20 -5 25	**PASS**

cchange.floyd	105, 216, 1	4 0 1, 8 1 1 1, 0 0 0 1	**PASS**
cfact.floyd	5, 12, 1	120, 479001600, 1	**PASS**
citerfact.floyd	5, 12, 1	120, 479001600, 1	**PASS**
cgcd.floyd	125 225	25	**PASS**
cgcd.floyd	160 4900	20	**PASS**

Phase 5 (Continued)

File Name	Input	Output	Result
bchange.floyd	105	Quarters: 4 Dimes: 0 Nickels: 1 Pennies: 0	**PASS**
bchange.floyd	216	Quarters: 8 Dimes: 1 Nickels: 1 Pennies: 1	**PASS**
bchange.floyd	1	Quarters: 0 Dimes: 0 Nickels: 0 Pennies: 1	**PASS**
blist.floyd (tostring uncommented)	[NONE]	10 20 5 50 [10,20,5,50] []	**PASS**
bnulltest.floyd	[NONE]	5 10 Null pointer exception on line 54	**PASS**
bobjbasics.floyd	[NONE]	5 10 100 200 -5 10 100 200	**PASS**
bstrbasics.floyd	jaja	Enter a string of characters:jaja s has 4 characters. charAt(0) = 'j' s > ' ' s >= ' ' q = wowsers!	**PASS**
bstrbasics.floyd	:thinking:	s has 10 characters. charAt(0) = ':' s > ' ' s >= ' ' q = wowsers!	**PASS**
bstrbasics.floyd	[NONE]	s has 0 characters. charAt(0) = '□' ! s > ' ' ! s >= ' ' q = wowsers!	**PASS**
bstrlits.floyd	[NONE]	q = This is a test. q = This is a tab. carriage return. q = This is a newline. q = This is a form feed. q = This is a\backslash. q = This is a "quote" q = This is an octal tab.	**PASS**