WCTC - Who Compiles The Compiler

Aditya Tanwar 200057
Janhvi Rochwani 200467
Soham Samaddar 200990

April 2023

https://github.com/cliche-niche/WCTC

1 Introduction

WCTC - Who Compiler the Compiler - is a Java 17 to x86_64 compiler. It uses Flex for lexical analysis and Bison for parsing the JLS17 grammar. It then uses graphviz to create a DOT script to produce the Abstract Syntax Tree of a given program. The AST is traversed to create a global symbol table which is parent to the symbol tables of all the classes of a program, in turn parents to the symbol tables of their member functions.

The Abstract Syntax Tree and Symbol Table are used to generate the three-access code for the input program. Finally, the 3AC representation is translated to x86-64 bit assembly code.

The compiler is written in C++ and can be used by running the commands as given in Section 4.

2 Features Supported

The following language features are supported by WCTC:

- Primitive data types (e.g., `int, long, float, double, and boolean`)
- Multidimensional (max 3D) arrays with support for C-style declarations (e.g., `int a[][] = ...`).
- Basic Operators:
  - Arithmetic operators: +, -, *, /,
  - Preincrement, predecrement, postincrement, and postdecrement
  - Relational operators: ==, $\neq$, >, <, $\geq$, $\leq$
  - Bitwise operators: &, |, ^, ~, $\ll$, $\gg$, $\ggg$
  - Logical operators: &&, ||, !
  - Assignment operators: =, +=, -=, *=, /=, &=
  - Ternary operator
- Control flow via `if-else, for`, and `while`

- Methods and method calls, including both static and non-static methods
- Support for recursion
- Support the library function `println()` for integers
- Support for classes and objects, accessing and using member functions and field members
- Declarative and Non-declarative statements

3 Additional Features

- Arrays can have any number of dimensions, there is no upper bound.
- Variables can be used to initialize arrays
- Default constructors are also supported
- The `do-while` control flow is supported
- Methods and method calls, including both static and non-static methods
- Static polymorphism via method overloading

4 Running the Compiler

The implementation of the Parser and Lexer is in Flex+Bison, written in C++. Further, DOT is used to create an AST. If these are not already installed, the following commands should be executed:

```
sudo apt-get update
sudo apt-get install flex
sudo apt-get install bison
sudo apt-get install graphviz
```

A makefile has been provided along with the source code files, the executable `WCTC.o` can be made by simply running `make`.

The command line options provided by our program have been documented below:

- `-i` or `--input`: Filename from which the compiler reads the Java program
- `-o` or `--output`: Filename to which the compiler outputs the `DOT` file
- `-v` or `--verbose`: Prints the derivation (parse tree) in the command line
- `-t` or `--taco`: Filename to which the compiler outputs the 3AC file
- `-s` or `--asm`: Filename to which the compiler outputs the x86 assembly file.

- `-h` or `-help`: Manual page

By default, input is read from `test.java` and output is redirected to `tree.gv` and `tac.txt`. The assembly code is output in `asm.s`. An example of a successful execution is given below:

```
$ make
$ ./WCTC.o -i ../tests/test_1.java -s ../out/asm_1.s
```

There is also a folder called `scripts/` which houses `compile.sh` (in case the Makefile fails) and `script.sh` (asks for the number of test cases to run from the java files available in the `tests/` directory). The test cases have been sourced from the internet.

5 3AC Optimization

1. Removed Redundant Temporaries:
A redundant temporary is one that is defined once (appears only once in LHS) and is used with the same value elsewhere in RHS. All such instances were removed by replacing the redundant temporary with its defined value.

2. Renamed Temporaries and Jump Instructions:
The temporary for a node is made using a node number. We made the code more presentable and easy to use by naming the temporary variables chronologically. For this we just keep track of the number of temporaries seen and create new variable names accordingly.

3. Constant Folding:
The values of decimal expressions were calculated and replaced in the three-access code as required.

4. Constant Propogation:
The values calculated in constant folding are propagated to replace their defining temporaries.

5. Strength Reduction:
Algebraic strength reduction is executed so as to change expensive operations to relatively cheaper ones. The following are the paradigms used for strength reduction:

```
a + 0 = 0 + a = a
a - 0 = a
a * 0 = 0 * a = a
a / 0 = Error
0 / a = 0
a << 0 = a
0 << a = 0
a >> 0 = a
0 >> a = 0
a >>> 0 = a
0 >>> a = 0
a & 0 = 0 & a = 0
```

```
0 | a = a | 0 = a
a ^ 0 = 0 ^ a = a
a * 1 = a / 1 = a
false && a = a && false = false
true || a = a || true = true
a - a = 0
a / a = 1
a & a = a | a = a
```

The processes of Constant Folding, Constant Propagation, and Algebraic Strength Reduction are performed till there are no constant optimisations left to compute.

The 3AC generated directly from the syntax tree can be found in `../misc/tac_unopt.txt` and the 3AC after these optimizations is stored in `../misc/tac.txt`

6 Making the x86 Assembly Code

We have written the assembly code using the AT&T syntax as we compiled it with `gcc`. Instructions of different types are created - instruction, label, comment, and segment. The x86 code is made of subroutines obtained from the 3AC. It begins with the `.data` and `.globl` segments, followed by the `.text` segment.
We included two fixed subroutines to implement C library functions - printf and malloc, leaving future scope for more functionalities like scanf.

After these, we needed to get the subroutines of the entire 3AC. All the snippets of 3AC that are of the type "`begin_func foo ... end_func foo`" make up the subroutines for x86. Each subroutine has its sub_table and many such sub_tables make up the generated code (`codegen`) structure.

Every time a subroutine is called, we perform the caller duties. That is, we push the caller-saved registers to the stack, then push the calling parameters of the function. We also push the implicit this pointer if the function is non-static. When the subroutine is called, x86 implicitly pushes the return address. At the callee end, push the old base pointer and set it to the stack pointer. The callee-saved registers are then pushed to the stack. Local variables and temporaries can be pushed to the stack as needed in the call, and are popped before the function returns.

The callee-saved registers are now restored, including the old base pointer. Consequently, the stack pointer points at the stored return address and thus we go back to the caller with `ret`. On coming back to the caller, the pushed parameters and caller-saved registers are popped, and execution continues.

7 Manual Changes to the Generated x86 File

There are no manual changes that need to be made to run our generated x86 file.

8 Unsupported Features

The following are the limitations of WCTC:

- We supported full-fledged type casting till the 3AC stage and prohibited lossy conversions. For simplicity in stack manipulation, we did not support smaller integral types in Milestone 4.
- The compiler throws relevant errors if a void function returns with a non-void value, or if a non-void function returns with the wrong data type. However, it shows undefined behaviour if the return statement is missing in a non-void function.
- Member variables of an object must be accessed with the `this` pointer. This is because allowing access without the `this.x` method required us to change a lot of the code of milestones 2 and 3.
- Static methods and method calls are supported, but static variables are not.

9 Miscellaneous

After Milestone 1, to automate the process of generating the data structure of AST, we wrote a lexer file `semaction.l` that analyses `parser_empty.y`, which in essence is the JAVA Grammar, and populates `parser.y` with the correct semantic actions.

In Milestone 4, we also identified Basic Blocks which can be used for further optimizations with the concepts of Next Use and Liveness.

10 Test Cases

The following are the features highlighted by the test cases we have added in `../tests/` directory:

- `test_1.java`: Solves the Knapsack problem and hghlights usage of arrays, and passing them into functions as parameters. Also displays support for recursive functions
- `test_2.java`: Finding n^{th} Fibonacci number using recursive functions.
- `test_3.java`: Calculates n^{th} prime number and highlights different arithmetic operations.
- `test_4.java`: Highlights support for member field variables by finding n^{th} Fibonacci number using memoization. Also highlights use of variable-sized dimensions for arrays. For example `arr[n][m]`.
- `test_5.java`: Finds $n!$
- `test_6.java`: Finds sum of array
- `test_7.java`: Implements Binary Search
- `test_8.java`: Highlights support for function overloading / static polymorphism.
- `test_9.java`: Implements Bubble sort
- `test_10.java`: Implements Quick sort

11 Contribution

#	Member Name	Roll Number	Member Email	Contribution
1	Aditya Tanwar	200057	tanwar20@iitk.ac.in	33%
2	Janhvi Rochwani	200467	janhvir20@iitk.ac.in	30%
3	Soham Samaddar	200990	sohams20@iitk.ac.in	37%