WCTC - Who Compiles The Compiler

Aditya Tanwar 200057
Janhvi Rochwani 200467
Soham Samaddar 200990

April 2023

1 Running the Files

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
- `-t` or `-taco`: Filename to which the compiler outputs the 3AC file
- `-v` or `-verbose`: Prints the derivation (parse tree) in the command line
- `-h` or `-help`: Manual page

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

```
$ make
$ ./WCTC.o -i BubbleSort.java -t tac.txt
```

2 Optimizing the 3AC

In our last submission, we would create a new temporary variable for every new definition and use in the 3AC code. This means that for the following example code:

```
a = 0
b = 3
c = 4
a = 3*b + 2*c
```

The 3AC would look like:

```
1.  ##t23 = 0
2.  ##t22 = ##t23
3.  a = ##t22
4.  ##t32 = 3
5.  ##t31 = ##t32
6.  b = ##t31
7.  ##t41 = 4
8.  ##t40 = ##t41
9.  c = ##t40
10.  ##t49 = 3 * b
11.  ##t52 = 2 * c
12.  ##t48 = ##t49 + ##t52
```

where '31' in ##t31 represents the node number that the temporary variable originates from.

This representation can be made more efficient. Also, when working with x86_64 code, we would have a limited number of registers, so optimization is really important. The function `optimize_tac` traverses the original 3AC for:

1. Removing 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. So, we maintain two maps - the first map stores the temporary and its appearances in LHS, the second stores the temporary and its appearances in RHS.

We also ensure that the statement which uses this "redundant" temporary is not jumped to from another statement. For example, if some instruction jumps to line number 6 in the original 3AC, `t31 = t32` will not take place and `b` will use the old value of `t31`. For this, we maintain a set 'jumped_to' of instructions that can be reached with a jump statement and don't remove the variables relevant to the instructions of this set.

If the definition of a variable and its use are not consecutive, its definition may change in between, and hence the variable cannot be replaced with its definition.
For example: `t1 = a1; a1 = a2 + a3; t2 = t1;` Here, `t1` cannot be replaced with `a1`.

The code now looks like:

```
3.  a = 0
6.  b = 3
9.  c = 4
10.  ##t49 = 3 * b
11.  ##t52 = 2 * c
12.  ##t48 = ##t49 + ##t52
```

2. Renaming temporary variables

We make 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. The code is now:

```
3.  a = 0
6.  b = 3
9.  c = 4
10.  ##t1 = 3 * b
11.  ##t2 = 2 * c
12.  ##t3 = ##t1 + ##t2
```

3. Updating Instruction Numbers and Jump Instructions:

Again, for better readability and use, we rename the instruction numbers. We also need to update jump & goto instructions with new line numbers. The final 3AC code is:

```
1.  a = 0
2.  b = 3
3.  c = 4
4.  ##t1 = 3 * b
5.  ##t2 = 2 * c
6.  ##t3 = ##t1 + ##t2
```

3 Additional Features:

- To support function overloading, we need to allow multiple definitions with the same function name. Hence with different input parameters, the function in 3AC should be different. The `get_mangled_name` function does the needed. The input parameter types are added to the function name, separated by `@`, to create the mangled name of that function definition.

  For example, the 3AC `"begin_func funcname"` becomes `"begin_func classname.funcname@int@float"`.

- We added modifier check for the keyword `static` and disallowed referencing any non-static function or field variables.

- We added support to calculate offsets in order to properly work with pointers. In our initial 3AC, qualified names were printed as it is. Now, with the help of pointers and using offset, the right object and method can be called based on the name.

- For method or constructor declarations, if the method is not static, the `this` pointer is also added to the stack along with its parameters. When the function is called, the `this` pointer and its parameters can be popped.
- Similarly, when a method is invoked or an object constructor is called, the input parameters are pushed to the stack so they can be passed to the function.
- We added support for memory allocation for arrays of primitive types.
- We made space for pushed parameters, return type addresses, and all local variables used inside that function. Space for registers has not been allocated to make the compiler architecture independent.