WCTC - Who Compiles The Compiler

Aditya Tanwar 200057
Janhvi Rochwani 200467
Soham Samaddar 200990

February 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`.

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

```
$ make
$ ./WCTC.o -i BubbleSort.java
```

2 Creating the Symbol Table

The symbol table is created by traversing the Abstract Syntax Tree generated by parsing the program. To populate the symbol table correctly, we need to ensure that any entry being added to the symbol table is unique when needed (as in the case of variables in the same scope), is declared and initialized before being used, and has the correct datatype according to the grammar. Thus, we traverse the AST three times with helper functions to `create_scope_hierarchy()`, `populate_and_check()`, and `type_check()` all the nodes.

A new symbol table is created each time we encounter a new scope - a class, a function, a block. A typical symbol table stores a list of all its entries and maintains a list of children symbol tables. Its member variables keep track of its scope, number of sub-scopes, parent symbol table etc. Additionally, a global symbol table stores all its classes, a class symbol table stores all its member functions, a function symbol table stores all its parameters.

2.1 Create Scope Hierarchy:

The scope of every new symbol table is determined by its parent scope and its position in the parent's subscopes. For the following program:

```
public class HelloWorld {
    int x; {
    void func() {
        if(x>0){
            int y = 1;
            x = y;
        }
    }
}
```

The scope of variable y is set as HelloWorld#1/func#1/Block#1.

For classes, we add class symbol tables to the global symbol table, for methods, we add the function to the member functions of its parent class, etc. For constructors, we check that the name of the function matches its class name. In case a class does not have a constructor definition, we add a default empty constructor.

Since JAVA does not need forward declarations, in this step, we ensure that all classes, functions, and field variables are added before checking for local variables.

2.2 Populate and Check

The symbol tables are now populated with local variable declarations. For every declaration, we check whether the type of the object is one of the primitive types or defined class types before adding it to the symbol table. If not, we throw an error.

For initialization and use of variables, we invoke the `validate_expression()` method:

- If a function is used in an expression, the function must be a member of the classes available for that scope. An "Unknown method identifier" error is thrown otherwise. (Whether the return type of the function is complatible with this expression will be checked in the next walk)
- If the initialization or use invokes a constructor, the type of the object must be the same as the class of the constructor. A "Conflicting types for object" error is thrown otherwise.
- If there is a variable in the expression, it must be declared and initialized before use.
- If there is an initialization expression inside an expression (for example: `a = 3 + 4/(c=2)`), all the "sub-expressions" must also be valid. Hence, `validate_expression()` is called on all children of the main expression.
- If there is an array variable in the expression, the dimensions must also match. For example: `int a[][] = new int[2]` throws the error "Dimensions do not match for array".

2.3 Type Check

We do a DFS traversal of the AST with the `chill_traversal` and check that the type of each node is set correctly with the `type_check` function. The default data type of each note is set as Undefined. For nodes that do not typically have a type, say 'Expression', their type is set to the type of their children nodes. For example, in the Assignment `int a = 2 + 3`, the Expression node refers to "`2+3`" and has a child node `OPERATOR+` (which further has two children nodes `LITERAL 2` and `LITERAL 3`). The type of expression must be set to `int` so we can compare its type with the variable `a`. Further,

- For identifiers, the type was already set in the symbol table. The datatype of the node is set as the type of the symbol table entry.
- For qualified names, for example `int x = a.b.c.d`, each name should be a valid member of its calling class (b must be a member of a, c must be a member of b and so on). The datatype of the node will be the datatype of the last member (in this case, d must be type `int` or of a type that can be cast to `int`)
- For array creation, the dimensions as well as the type must be same.
- For array accesses, the dimensions as well as the type must be same. Also, the expressions in the square brackets must be type `int`.
- For return statements, the return keyword must be followed by a node of the same type as mentioned in the function declaration. For non-void functions, there should be no node after the return keyword.
- PostIncrement, PreIncrement, PostDecrement, PreDecrement can only be performed on variables, and not on values (numeric literals).
- In type casting, lossy conversions are disallowed. The result of a cast expression is always a value (a literal) and not a variable. Thus, statements like `(int a)++` throw an error.
- For method invocations, we first need to fetch a valid function from the accessible class symbol table. In case there are multiple definitions, we find a definition that is compatible with the invocation. This means that if the datatypes of parameters are not an exact match with the invocation parameters, we check if the invocation parameters can be cast to the types of the definition parameters without loss. For example, a statement with `func(a)` where `a` is an `int` and the function definition is `void func(float input)`.
- Type checking for constructors is done similarly.
- For variables, declaration may be done with initialization. The type of the initialization expression must either be the same as the type of the variable or it can be cast to the variable type without loss.
- For operator nodes, the datatype has to be calculated based on the operands. The `calc_datatype` function takes the operands and the operator as input and sets the datatype of the node according to type conversion rules (given in units 15.14 to 15.26 of the Expressions chapter of Java Language Specifications)

- For If statements, Do-while statements, For statements, and While statements, the conidition expression must be of boolean type.

- For all other nodes, the datatype is set to "UNDEFINED".

3 Generating IR - 3AC

We use the quadruple data structure to create 3AC representation. We do a depth first traversal on the AST, generating code for child nodes before the parent node. We use relative referencing to generate the code and make the 3AC more readable by printing absolute line numbers instead of labels. The quadruple generated for each node is pushed up to its parent node in a semantically decided order. In the method `generate_tac`:

- For literals, identifiers and qualified names, no new variables are created unless they need to be cast to some other type. For other nodes, variables have names like `__t23` where 23 is the node number in the AST. The qualified names are printed as it is.

- For expressions, codes of the form `result = arg1` are created. If the expression needs to be cast to another type, a new code is created and pushed to its list of `ta_codes`

- For if-then-else statements, the function `make_code_from_conditional` is used with number of instructions to jump in case the condition is false. The function `make_code_from_goto` is used with number of instructions to jump to get out of the if-then-else block.

- For while loops, we first create TA codes to calculate the while expression. Then, similarly as if-else statements, we create a TA code with jump instructions in case the condition is false, i.e, when the loop ends. Then we append the TA codes from the statement of the while loop, ending with a "Goto" TA code pointing to the begining of the loop.

- For do-while loops and for loops, similar methods are used to create TA codes.

- For array creations, we have only added support for integer arrays. The memory allocation for array will be done in the next stage.

- For array accesses, we determine the address of the access and then create a TA code for dereferencing.

- For method invocations, we push all arguments as TA codes onto the stack, create a code for the function call, and pop the arguments from the stack.

- For function declarations, we add a "begin_func", print the TA codes of the function instructions, and then add an "end_func".

- In variable declarations, we add the TA codes of the assignment.

- In case of casting, we create a TA code with the operator as the casting datatype.

- For return statements, we first generate TA codes for the statement, then the keyword return.

- PreIncrement, PostIncrement, PreDecrement, PostDecrement have been reduced to "+=" or "-=" expressions and accordingly, TA codes are generated.

- For binary operator nodes, a TA code corresponding to "child_node_1 op child_node_2" is added, followed by the TA codes of the children and similarly for unary operators.

- For assignment operators, say "a += b", the expression is expanded to "a = a + b" and dealt with accordingly.

- If an expression requires casting, we print a case TA code as well.

All of these codes are pushed to the root node (OrdinaryCompilationUnit) and printed with the instruction number.