Assignments 2–4: Static Checking

Zhenyan Zhu - z277zhu		Zijian Feng - z63feng		Richella Li - j2333li

Instructions

- To build the compiler, run **mak**e at the root directory.
- To compile a list of java file, we use the following usage:
- ./joosc [stdlib files] [java test files]
- Eg. ./joosc $(find stdlib/2.0/java/ -name *.java) test/self_test_a4/testFor.java

Design and Implementation

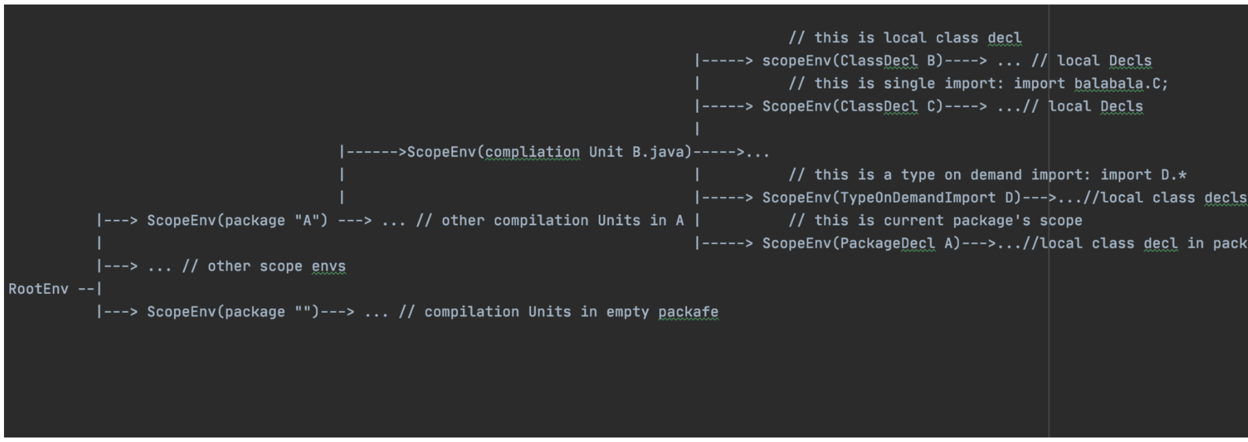
- Project file structure

  Our project is developed with java and a structure with sbt as our build tool.
  The structure of the files are listed below:

```
- build (store our intermediate files while building.)
- lib (store the third-party library and also the package we built)
- src (store the source files)
  |-- ast (the definitions of every Node in the AST Tree).
  |-- dataflowAnalysis (the definition of control flow graph, aka, CFG)
  |-- exception (the definition of SemanticError thrown during compiling.)
  |-- flex (jflex/cup config files)
  |-- hierarchy (the implementation of Hierarchy Checking)
  |-- lexer (parser/lexer generated by flex)
  |-- type (TypeChecking implementation including Environment builder and Type checker)
  |-- utils (some common tools functions used regarding the AST tree.)
  L-- visitors (TypeCheckVistor and StaticCheckVistor that used in the Visitor Design Pattern)
- stdlib (java standard library source code)
- test (store the test cases and some bash scripts for testing)
- boot.sh (boot the docker env)
- Dockerfile (to build the docker env)
- Makefile (build this compiler)
```

- Type environment
  a. We separate our Environment classes into two: RootEnvironment and ScopeEnvironment. RootEnvironment is the top-level environment, which is used for searching qualified name; Each package, class or interface, method, and block is a ScopeEnvironment, which is used for searching the closet simple name.
  b. RootEnvironment contains three core fields:

        i. Map<String, ScopeEnvironment> packageScopes that maps each package name to its ScopeEnvironment
        ii. Map<ASTNode, ScopeEnvironment> ASTNodeToScopes that maps each ASTNode to its enclosing environment, which serves convenience for future use
    c. ScopeEnvironment contains three core fields:
        i. Map<String, Referencable> localDecls, which is a mapping from qualified name to a Referencable type, a Referencable type a interface which TypeDecl, MethodDecl, FieldDecl, LocalVarDecl implements; this mapping is used to store every local declarations in scope that serves for looking up a qualified name
        ii. Map<ASTNode, ScopeEnvironment> childScopes, which is a mapping from each declaration to its corresponding scope used for look up qualified name
        iii. Environment parent, a reference to the parent of current scope. This is used for look up simple name
    d. The hierarchy of RootEnvironment is like:



```
                                                                                          // this is local class decl
                                                                               |-----> scopeEnv(ClassDecl B)----> ... // local Decls
                                                                               |          // this is single import: import balabala.C;
                                                                               |-----> ScopeEnv(ClassDecl C)----> ...// local Decls
                                                                               |
                                      |------>ScopeEnv(compliation Unit B.java)----->...
                                      |                                        |          // this is a type on demand import: import D.*
                                      |                                        |-----> ScopeEnv(TypeOnDemandImport D)--->...//local class decls
          |---> ScopeEnv(package "A") ---> ... // other compilation Units in A |          // this is current package's scope
          |                                                                    |-----> ScopeEnv(PackageDecl A)--->...//local class decl in pack
          |---> ... // other scope envs
RootEnv --|
          |---> ScopeEnv(package "")---> ... // compilation Units in empty packafe
```

- Hierarchy checker
    a. Our hierarchy checker runs in the scope environment built in the previous step. The checker starts at the root environment.
    b. In every compilation unit (a class or an interface), it will find all the super types of a class or an interface:
        i. List <Referenceable> generalBaseObjectClass: it contains all the declarations in java.lang.object, which will be included in a class/an interface without any superclasses/superinterfaces.
        ii. Map<ASTNode, List<Referenceable>> directParentMap, the map contains the superclasses or interface included in the extend clause or implement clause.
        iii. Map<ASTNode, List<Referenceable>> parentMap, we use directParentMap to build the parentMap, this map maps an ASTNode to all of its parents(superclasses and superinterfaces).
        iv. Map<ASTNode, List<Referenceable>> declareMap, it maps a node to all of the methods/fields/constructor declared in it.
        v. Map<ASTNode, List<Referenceable>> inheritMap, it maps a node to all of the methods/fields inherited from the node's parents.
- Type checker
    a. Visitor design pattern is implemented to traverse all ASTNodes for type checking, each ASTNode has an accept method to achieve the recursive visit mechanism, which

guarantees that the sub expressions' type are evaluated before current expression's type's evaluation

b. A Type **type** field is added in Expr, for each expression, we strictly follow the joos type rules and assign correct types to expression.**type**

c. Several steps to do name disambiguation (for PostFixExpr and MethodInvocation): for any ambiguous name 'a.b.c…x.y.z', we do:
    i. Check whether it's a static field or not:
        1. If 'a.b.c….x.y' is evaluated to be a ClassOrInterfaceType, we check whether 'y' is a static field of such type
        2. If it's not, then this name is not a static field access
    ii. Evaluate the first prefix (ex.'a.b') that can be a expression, record the type of such expression
    iii. We iterate through the remaining characters in the name, for each character, we check whether it's a accessible field of current type
    iv. For each field access while disambiguating the name, we perform static/protected check

- Unreachable statement checker
    a. CFG
        i. Vertex: In the CFG, we create the vertex structure to hold every node in the CFG. In every vertex, it contains its successors and predecessors stored in List<Vertex>. There are also two boolean fields in and out used in the reachable statement analysis.
        ii. CFG is a structure that holds a set of vertices contained in it, with special START and END vertices to recognize the start and end of the graph.
    b. Worklist
        i. The worklist is implemented by a queue stored in CFG. (Queue<Vertex> workList).
        ii. We init the worklist using the set of vertices stored in CFG, and execute it following the worklist algorithm given in class.
        iii. After running the algorithm, except for checking in[end], we also check the in field for every other vertex to make sure they are reachable.

Testing:

- A2: Tested environment scope correctness; tested import/package declaration correctness; Tested the keywords (abstract, final, static) in different hierarchies; tested the case handling emptyPackages; tested the differentReturn types in method signature checking; tested the tracing process of method Invocation usage.
- A3: Testing of the class containing the same field name as the class name: our approach to create scope uses a map to map some names of methods, fields, and some constructor to their ASTnodes. However, during the testing process, we found out there were some cases where we could use the same field names/method names/class names at the same time, but our approach could only handle the same method names but not the same field name and the class name.

- A4: Tested the recursive division of control flow data graph in if/else statements; tested the integer and boolean expression result in compiling. Tested the consecutive path of While/For statements/

Known issues:

The only issue is that our scope environment can only save fields, methods and constructors with different names (different methods/constructors can have the same name for overloading). However, after finishing A3, we notice that fields can have the same name with constructors as well as methods, and if we want to solve this case, everything would be restructured and rewritten, so we ignore this case.

Work

- Zhenyan Zhu - z277zhu
    - A2 - Environment Construction
        - Designed and implemented Environment hierarchy
        - Developed environment builder that recursively builds up the root environment by traversing all ASTNodes
        - Designed and implemented lookup(qualifiedName/simpleName)
    - A2 - TypeLinker
        - Designed and implemented TypeLinker that resolved names that can be syntactically determined to be names of types
        - Import package/ImportDecls scope into each corresponding CompilationUnit's scope
    - A3 - Visitor framework construction
    - A3 - Type checker: Design and Implement the main framework of Type checker
    - A3 - Type check rules including: CastExpr, MethodInvocation, ReturnStmt, PostFixExpr, staticCheck, protectedCheck, ArrayAccess, as well as main part of Name disambiguation
    - A4 - CFG
        - Designed and implemented CFG structure, internal methods
        - Built vertices from atomic statements and implement control flow in WhileStmt
- Zijian Feng - z63feng
    - A2 - Hierarchy Checking
        - Implement the checking of all the various rules given in Chapters 8 and 9 of the Java Language Specification.
        - Some pre-process regarding the inheritMap, declareMap, parentMap
        - Built the containMap.
    - A3 - Type Checking
        - Implement the rule check including keywords(abstract, protected), Literal, Operators, conditional statements, constructors, field access, and assignments.
    - A4 - CFG
        - Build the control flow graph for loop statements and if statements.
        - Add the integer calculation of all operator expressions in AST to support the detection of constant boolean in loop statements.
- Richella Li - j2333li

- A2 - Hierarchy Checking
    - Checked Class/Interface level error
    - Built inheritMap, declareMap, parentMap, directParentMap
- A3 - Name Disambiguation
    - Assigned type to simple names and some qualified names
    - Static field checking
- A4 - Worklist Algorithm
    - Implemented worklist algorithm to assign value of in and out for every node in the CFG graph
    - Checked unreachable nodes

Thoughts

How much time did you spend on the assignment? What did you like or dislike about the assignment? What would you change about it?

Assignment#	Richella Li	Zhenyan Zhu	Zijian Feng	Total Hours
A2	14 hours	15 hours	22 hours	51 hours
A3	25 hours	30 hours	12 hours	67 hours
A4	4 hours	5 hours	6 hours	15 hours

Like: The design pattern we used in this assignment has greatly exercised our logical thinking ability. We have to consider further to avoid other rebuilding work in the future.

Dislike:

- Lots of duplicate works to implement the rules of all visitors in the visitor design pattern. And the recursive thinking of the AST tree can be tricky since for some cases, we need to do Top-to-Bottom parsing and the other will do Bottom-to-Top.
- Assignment 3 is so confusing. Name disambiguation should be a part of type checking, which should be developed together with different type rules, but the assignment guideline separate this into two parts, and we found that it's so hard to do name disambiguation outside of type checking visitor, and thus we did a large amount of unnecessary work.

What aspects of your chosen programming language did you find helpful in writing the compiler? What aspects caused annoyance?

We use Java to implement the compiler. The aspect we found helpful in writing the compiler is it can determine the instance of the node. In many cases, we need to know which node it is, in hierarchy checking we want to know if it is a field, a MethodDecl or a ConstructorDecl, in type checking, we need it to evaluate its type and disambiguate all names…

Any other thoughts?

After finishing A3, we realized some of our approaches for neither organizing the code nor splitting parts are correct. We should try to come up with a big picture before actual implementation to avoid redundant and useless work. Now we are more likely to split one part into three small pieces rather than splitting to three total independent parts, trying to let everyone be involved and familiar with every part of our code.

The cooperation between people is a fine art. To make communication much easier as a programmer, the most important point is to learn how to write the comments besides the code. If we clearly show what these codes are doing, we can avoid much spending on the API misunderstanding.