CS4201 P1 - Semantic Analyzer for MiniJava

190006526

October 10, 2022

1 Instructions

To run the program on one of the test file (`<TestFile>`), run the following:

```
cd src/
javac minijava/*.java minijava/errors/*.java minijava/symbol/*.java
java minijava.MiniJavaMain ../tests/<TestFile>.java
```

This program is dependent on `antlr-4.7.2-complete.jar`

2 Abstract

For this practical, we were tasked with designing and implementing the semantic analysis stage of a compiler for MiniJava, a subset of Java described in the textbook "Modern Compiler Implementation in Java" (MCIiJ) [2]. To achieve this task, we were provided with a grammar file for MiniJava, which can be fed into the tool ANTLR[1] to create a parser for MiniJava, on top of which the semantic analyzer will be built. The semantic analysis stage includes checking the scope of variables, type checking, and some other miscellaneous properties of the program specific to Java, like inheritance checking. I have written a semantic analyzer that meets the specifications described in the practical specification, with robust error checking, and symbol table printing. In the rest of this report, I will describe the design decisions I made, and some of the specifics of the implementations.

3 Design

3.1 Grammar

While the provided grammar stayed mostly intact, a few modifications were made to make implementing the semantic analyzer easier, as well as supporting some more interesting language features. First of all, the return statement was changed from having to be at the end of each method to being able to be anywhere. This makes it so that return statements can occur inside control structures. Additionally, variable declaration can now be interleaved with other statements. While this adds a bit of complexity for the semantic analyzer, it's not too much trouble and makes for more readable code being allowed. As well as those features, labels were added to statement and expression cases, so that the listener could have separate hooks for these cases, instead of having to make multiple conditional checks for each different case. This lead to a cleaner implementation of the type checking phase. The order of the operator expressions and index expressions was also swapped. This was to fix a bug in the parser where an index expression on the right hand side of an operator would get misinterpreted.

3.2 Symbol Table Building

The first step for semantic analysis is building the symbol table. This must be done before the rest of the error checking, because types in MiniJava can be mutually recursive, and the symbol table needs to be fully built before it can be known whether a given type is correct. To represent the symbol table, hash maps are used to store variable, method, and class bindings for each scope, and the child scopes are also stored in a hash map. Additionally, the parent scope is stored. This creates a tree structure of scopes, where scopes never need to be destructively modified. When looking up an identifier in the tree, the current scope is checked, then the parent scope, and so on. This makes it quick to lookup identifiers,

as in each scope that identifier is being searched for in a hash map. Despite not being able to check for many errors during this first phase, the program still checks for duplicate identifier errors.

For the implementation of the symbols themselves, the symbol implementation in MCIiJ was adapted, which saves all of the symbols in a hash map, and then when a symbol is seen, it is grabbed out of the hash map. These symbol objects can be compared much more quickly than strings, as the check is just pointer comparison, as for each string there is only one symbol object [2].

Once the table is built, inheritance cycle checking is performed. The inheritance graph is built during the symbol table building, with the class bindings keeping track of which classes extend others. Because all classes exist in the top level scope, the symbols they extend are stored instead of the class scope itself, because a class could extend another class that hasn't been defined yet. Then when checking for cycles, for each class, a hash set is created, and if a class shows up more than once when tracing the inheritance, an error is thrown. This is also where the semantic analyzer checks that extended classes actually exist.

Symbol table printing is done after construction of the symbol table. The printing method makes use of the tree structure of the symbol table, and recurses through the tree, printing sub-scopes with an offset. An example of the output from the `BinaryTree.java` file has been provided in Appendix A. I based the output style on the UNIX `tree` command. This table printing was useful for checking that things were working as expected during development.

3.3 Scope Checking

Once the symbol table is built, a second pass can be made and the bulk of the error checking can be performed. Because a symbol table has been built, scope checking is straightforward. The current scope is kept track of, and a call to the symbol table can be made to ensure the identifier exists. Because of the tree structure of the symbol tables, if the symbol table does not contain the identifier in question, it can recursively check if its parent has it in scope. In addition to just scope checking, the program also checks if variables have been declared and initialized before they are used. To do so, each variable binding has flags for declaration and initialization. These flags are only set to false for local variables, because in Java variables that are declared in class scopes are automatically initialized, and method arguments do not need initialization, as they are set on the function call. Then, when the program sees that a function has been declared or initialized, the flag is set. If a variable is assigned before declaration, or used before initialization, an error is given.

3.4 Type Checking

Type checking is perhaps the most complex part of the semantic analysis process. To perform type checking, a stack of types is used to keep track of the type of sub expressions. Because a listener was used instead of a visitor, the return values of the visit functions couldn't be used. Instead, for a given expression, the types of the sub expressions are popped off the stack, those types are checked, and then the resultant type (if any) of the expression is pushed onto the stack. This allows the program to easily handle deeply nested expressions, without recursive calls. For an example, take the expression $(a+b)*c$. Since the traversal of nodes is leftmost first, and the program uses the exit handlers, the listener emits events in leftmost, post-order. So, the listener emits: a, b, $(a+b)$, c, $(a+b)*c$. When type checking, the type of a is pushed to the stack, then b, then those are popped off, and the left- and right-hand sides of $(a+b)$ are checked to ensure they are integers. If they are not, an error is given. Importantly, even if there is a problem with the types of the expression, the integer is still pushed to the stack. This is because although a sub expression may not be correct, we don't want to crash out or ignore the expression, we want to type check the rest of the expression. After the integer type is pushed to the stack, the type of c is pushed to the stack, then both are popped to check the type of $(a+b)*c$. Additionally, if the type of a symbol cannot be determined, the type "unknown" is pushed to the stack. This matches all other types, and lets the program continue after logging an error, instead of invalidating the entire expression.

One tricky case to be handled with this stack based type checking are method calls. This is because it is syntactically correct for methods to be called with more or less arguments than the method actually takes. Because of this, the program cannot know how many arguments were passed to it. To handle this, a stack is used to keep track of how many arguments have been seen. This number is popped when the type checking for methods is done, and checked against the type of the method signature. A stack is used instead of a number to handle nested method calls.

3.5 Error handling

To handle errors, there is an error handler class, that has a static method to add errors. As they run, the symbol table builder and type checker classes add errors to the error handler. Then, once the program has been checked for semantic errors, the errors are printed out. This has the advantage of being able to print the errors in line number order. To keep track of line number, a static line number variable is set at the start of each node seen by the listener. This makes it so the node handlers themselves in the listener don't have to keep track of what line they're on, making the code a bit easier to write.

For the errors themselves, an abstract error class was created, and then each kind of error extends this class. This allows the program to keep the text of the errors separate from where they are thrown, and allows for reuse of error types. A list of error types that the program checks for, along with their associated classes in the `minijava.errors` package, can be found in Appendix B.

4 Testing

To test the program, MiniJava programs that contained interesting scenarios and edge cases were given to the program to handle. In addition to testing basic scope/type checking, the tests in the `tests/` directory handle the following corner cases:

- The aforementioned grammar error where a statement like `a = 0 + arr[0]` would result in parser errors.
- Complex nested expressions with type errors.
- Array indexing related type errors
- Array declaration and instantiation errors
- Cyclic class dependency
- Non existent classes being extended
- Duplicate classes
- Duplicate methods
- Using inherited methods
- Nested method calls

As well as the test files I created, I used the existing test files to check that they did not show errors, and verified by hand that they did not in fact have errors. The output of each of the test files I created is shown in Appendix C.

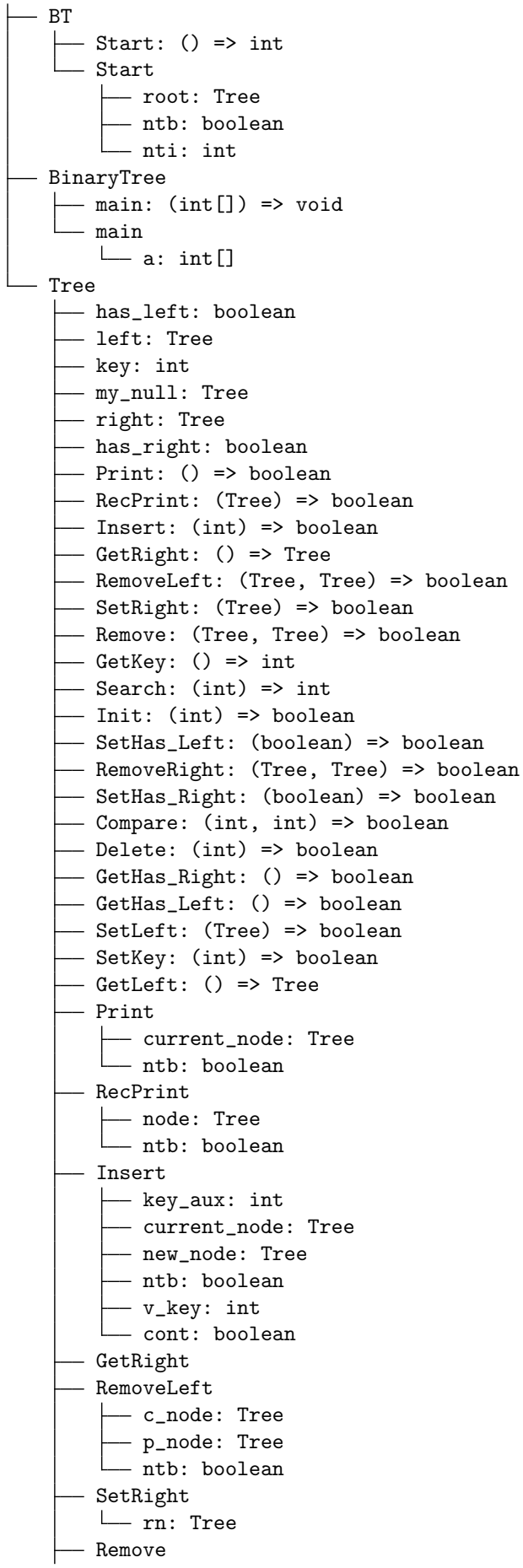
5 Conclusion

Overall, I greatly enjoyed this practical, and learned quite a bit from it. There are a few things I might have done differently with hindsight, such as using a visitor instead of a listener for type checking, to avoid handling the type stack, which was occasionally difficult to reason about. However, on the whole I am pleased with my implementation, and satisfied with the output it produces.

Appendices

A Sample Symbol Table Output

```
Global
├── class BT
├── class BinaryTree
├── class Tree
```


BT
Start: () => int
Start
root: Tree
ntb: boolean
nti: int
BinaryTree
main: (int[]) => void
main
a: int[]
Tree
has_left: boolean
left: Tree
key: int
my_null: Tree
right: Tree
has_right: boolean
Print: () => boolean
RecPrint: (Tree) => boolean
Insert: (int) => boolean
GetRight: () => Tree
RemoveLeft: (Tree, Tree) => boolean
SetRight: (Tree) => boolean
Remove: (Tree, Tree) => boolean
GetKey: () => int
Search: (int) => int
Init: (int) => boolean
SetHas_Left: (boolean) => boolean
RemoveRight: (Tree, Tree) => boolean
SetHas_Right: (boolean) => boolean
Compare: (int, int) => boolean
Delete: (int) => boolean
GetHas_Right: () => boolean
GetHas_Left: () => boolean
SetLeft: (Tree) => boolean
SetKey: (int) => boolean
GetLeft: () => Tree
Print
current_node: Tree
ntb: boolean
RecPrint
node: Tree
ntb: boolean
Insert
key_aux: int
current_node: Tree
new_node: Tree
ntb: boolean
v_key: int
cont: boolean
GetRight
RemoveLeft
c_node: Tree
p_node: Tree
ntb: boolean
SetRight
rn: Tree
Remove

```
    ├── auxkey1: int
    ├── auxkey2: int
    ├── c_node: Tree
    ├── p_node: Tree
    └── ntb: boolean
├── GetKey
├── Search
    ├── ifound: int
    ├── current_node: Tree
    ├── key_aux: int
    ├── v_key: int
    └── cont: boolean
├── Init
    └── v_key: int
├── SetHas_Left
    └── val: boolean
├── RemoveRight
    ├── c_node: Tree
    ├── p_node: Tree
    └── ntb: boolean
├── SetHas_Right
    └── val: boolean
├── Compare
    ├── num2: int
    ├── ntb: boolean
    ├── nti: int
    └── num1: int
├── Delete
    ├── is_root: boolean
    ├── current_node: Tree
    ├── key_aux: int
    ├── found: boolean
    ├── ntb: boolean
    ├── v_key: int
    ├── parent_node: Tree
    └── cont: boolean
├── GetHas_Right
├── GetHas_Left
├── SetLeft
    └── ln: Tree
├── SetKey
    └── v_key: int
└── GetLeft
```

B Error Types

Table 1 describes the different error classifications the program checks for, the associated classes, and a description when the error is thrown.

C Test File Error Output

Included in this section is the output for each of the test files I created. I have ommitted the tree printing and only included the errors.
Output for `BasicTypeDemo.java`:

```
NO OUTPUT PRODUCED
```

Output for `BasicTypeErrors.java`:

Error Type	Classes	Description

Duplication	`DuplicateArgumentError`
	`DuplicateClassError`
	`DuplicateMethodError`
	`DuplicateVariableError`
		Thrown when a symbol is already defined in the current scope.

Not Found	`ClassNotFoundError`
	`IdentifierNotFoundError`
	`MethodNotFoundError`
	`VarNotFoundError`
		Thrown when a symbol is used but cannot be found in the current scope.

Decl. & Instant.	`AssignmentBeforeDeclarationError`
	`IndexBeforeDeclarationError`
	`IndexBeforeInstantiationError`
	`VarNotInitializedError`
		Thrown when a variable or array index is used before declaration or instantiation.

Array Type	`ArrayAssignmentTypeError`
	`ArrayIndexTypeError`
	`NewArrayTypeError`
	`NonArrayIndexError`
		Thrown when there is a type error involving an array operation.

Method Arg Type	`MethodArgumentCountError` `MethodArgumentTypeError`	Thrown when there is a mismatch between a method calls arguments and its formal parameters.
Return	`NoMethodReturnError` `ReturnTypeError`	Thrown when there is either no method return, or the return statement is of the wrong type.

General Type	`AssignmentTypeError`
	`BoolTypeError`
	`ClassExpectedError`
	`IfTypeError`
	`OperatorTypeError`
	`WhileTypeError`
		Thrown when there is a mismatch between the expected type and the actual type.

Table 1: Error Types

```
Error: line 12: a is of type int, but is being assigned to type boolean
Error: line 14: Operator + takes operands of type int, but found operand of type boolean
Error: line 18: b is of type boolean, but is being assigned to type int
Error: line 20: b is of type boolean, but is being assigned to type int
Error: line 22: Operator + takes operands of type int, but found operand of type boolean
Error: line 26: If clause expected a boolean, received int
Error: line 29: Operator < takes operands of type int, but found operand of type boolean
Error: line 33: While clause expected a boolean, received int
Error: line 39: c is of type int[], but one of its elements is being assigned to type boolean
Error: line 41: c is of type int[], but one of its elements is being assigned to type boolean
Error: line 43: b is of type boolean, but is being assigned to type int
Error: line 45: c can only be indexed with type int, but found type boolean
Error: line 47: Operator * takes operands of type int, but found operand of type boolean
Error: line 47: c can only be indexed with type int, but found type boolean
Error: line 49: Cannot perform index operation on b, it is of type boolean, not int[]
Error: line 53: d is of type A, but is being assigned to type int
Error: line 55: d is of type A, but is being assigned to type int
Error: line 57: Operator < takes operands of type int, but found operand of type boolean
Error: line 57: Operator + takes operands of type int, but found operand of type boolean
Error: line 57: Operator + takes operands of type int, but found operand of type boolean
Error: line 57: c is of type int[], but one of its elements is being assigned to type boolean
Error: line 57: c can only be indexed with type int, but found type boolean
```

Output for `CircularClassError.java`:

```
Error: Cyclic inheritance involving B
Error: Cyclic inheritance involving E
Error: Cyclic inheritance involving F
Error: Cyclic inheritance involving D
Error: Cyclic inheritance involving A
Error: Cyclic inheritance involving C
```

Output for `DeclarationInstantiationDemo.java`:

```
Error: line 13: Cannot declare variable x, symbol already used
Error: line 17: Variable c assigned before it was declared
Error: line 27: Variable d used before initialization
Error: line 36: Array arr indexed before it was instantiated
Error: line 48: Cannot declare argument arg, symbol already used
Error: line 55: Cannot declare variable y, symbol already used
Error: line 62: Cannot declare variable arg, symbol already used
```

Output for `InheritanceDemo.java`:

```
Error: line 11: myB is of type B, but is being assigned to type A
Error: line 19: myC is of type C, but is being assigned to type A
Error: line 20: myC is of type C, but is being assigned to type B
Error: line 31: Method foobar not found in class C
Error: line 57: Duplicate class E
Error: line 57: Class Bar not found
```

Output for `MethodErrors.java`:

```
Error: line 9: Return statement has type boolean, but method returns int
Error: line 14: Cannot declare method foo, symbol already used
Error: line 20: Method foo takes 2 arguments, found 1
Error: line 23: Method foo takes 2 arguments, found 3
Error: line 26: foo passed an argument of type boolean but was expecting type int
Error: line 29: foo passed an argument of type boolean but was expecting type int
Error: line 30: Method bar takes 0 arguments, found 1
Error: line 31: Method foo takes 2 arguments, found 3
```

```
Error: line 36: Method foobar takes 2 arguments, found 1
Error: line 38: Method foobar takes 2 arguments, found 3
Error: line 40: foobar passed an argument of type boolean but was expecting type int
Error: line 42: Method foobar takes 2 arguments, found 1
Error: line 49: Return statement has type boolean, but method returns int
Error: line 54: Return statement has type int, but method returns boolean
Error: line 57: Method foofoofoo has return type int, but does not return a value.
```

References

[1] ANTLR, ANother Tool for Language Recognition. https://www.antlr.org/. Accessed: 2022-10-10.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.