Compiler Project
PA3 – Identification & Type Checking

Due: 2024-03-20 11:59pm

The goal for this milestone is to ensure that the input miniJava program complies with the semantics of the miniJava language. Syntactic constraints were accomplished in PA1 and PA2, and now we will finish the contextual constraints. After this checkpoint, passing source code files are valid miniJava programs. Note, miniJava is not Java, so take care in implementing contextual constraints as the implementation may be simpler than expected. In PA3, miniJava and Java will now begin to differ noticeably.

From this point forward, the AST classes are yours. You can add, edit, or remove them as you see fit. Maintain a list of changes to the ASTs and submit that with your PA3 as a text file called "`ASTChanges.txt`".

All new contextual analysis code should be in the package miniJava.ContextualAnalysis.

This document will assume you will do identification and type-checking in two separate traversals. Additionally, any contextual constraints that belong to neither are not a separate traversal, but instead are built in to either of the two mentioned traversals.

1. NullLiteral extension

The literal "`null`" should be added to miniJava. This literal can be assigned to any object and tested for equality/inequality against any object. Under the provided ASTs in PA2, `NullLiteral` should extend Terminal or Reference. When extending reference, the created AST should be `NullReference`. For Syntactic Analysis, a `LiteralExpr` or `NullReference` can be created at the highest precedence level.

Ensure your Visitor interface is updated to `visitNullLiteral/Reference`. You can remove `ASTDisplay.java` entirely, but it is still helpful for debugging, so it would be easier to update `ASTDisplay` instead.

```java
public Object visitNullLiteral(NullLiteral nl, String arg) {
     show(arg, quote(nl.spelling) + " " + nl.toString());
     return null;
}
```

2. Identification

The identification process takes an identifier and resolves where that identifier is declared. It is recommended that you add a `Declaration` field to the `Identifier` AST.

There exist a few ways for you to implement identification. The textbook recommends a weighty vector-based version storing every `ClassDecl`, `MemberDecl`, and `LocalDecl`. Then, resolving an identifier would involve checking the vector in reverse order until a suitable `Declaration` can be found.

This is easier accomplished through Scoped Identification (see Lecture 08 and briefly covered here). Scoped Identification involves a stack of identifier to declaration maps.

Scoped Identification. (Shorthand: SI)

Level 0	Class Names (ClassDecl)
Level 1	Method and Field Names (MemberDecl)
Level 2+	Parameters, and Local Variables (LocalDecl)

A single `IDTable` is a map where an identifier string maps to a `Declaration`. The `IDTable` for level 1 is special as it must resolve context. This is because multiple class declarations may use the same identifier for their fields or methods. Thus, for level 1, the `IDTable` may require some customization. For example, the `IDTable` for level 1 can take a context (`ClassDecl`) and identifier as input, and map to a `MemberDecl`.

For simplicity, a single globally shared SI object is recommended.

SI is easiest done through a Stack of `IDTable`: `Stack<HashMap<String,Declaration>>`. Level 0 and Level 1 are always present. How you choose to make another class's private members invisible is up to you.

You can either choose to check private members manually when referenced in other classes, or you can modify the `IDTable`s themselves so that external private identifiers are no longer present. Despite the latter sounding more elegant and sometimes easier to implement, it is less desirable when it comes to error reporting. Reporting "that identifier is private" is clearer than "that identifier cannot be resolved to a declaration."

SI Methods.

SI has four methods: `openScope`, `closeScope`, `addDeclaration`, and `findDeclaration`. Every time you `openScope`, you push a new `IDTable` on the stack. The `closeScope` method pops one `IDTable` from the stack. The method `addDeclaration` will add an entry to the top-most `IDTable` that maps a String to a `Declaration`. It should throw an `IdentificationError` if that name already exists at that level or level 2+. Lastly, `findDeclaration` will take an `Identifier` as an input (and optionally a context), and return that `Identifier`'s `Declaration`, if one exists.

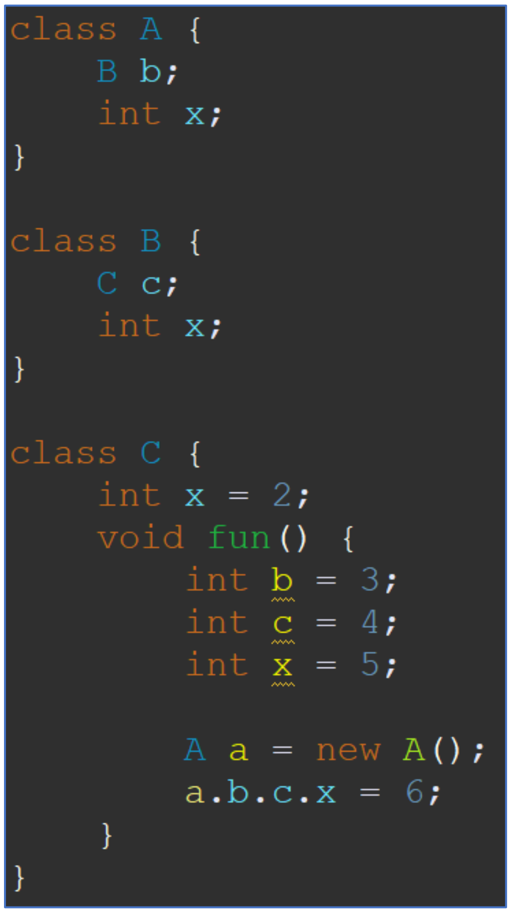
When you visit a method, your SI will enter a new scope. When a `BlockStmt` is visited, your SI will enter a new scope again. In such a created scope, whenever a `VarDecl` is visited, the top-most `IDTable` in SI is updated where that `VarDecl`'s name (a String) will map to that `VarDecl` via `addDeclaration`.

When finished visiting a `BlockStmt` or method, the SI will `closeScope`. Any local/parameter variables declared in the now-closed scope are no longer accessible and no longer hide lower-scope-level identifiers.

When visiting an identifier, find the associated `Declaration` in the `IDTables`. If there is no declaration, then that is an `IdentificationError`. You do not have to continue Contextual Analysis after an ID error. If a `Declaration` is found, then that `Identifier’s Declaration` field in the ASTs should be updated as a quick way to know where that `Identifier` is declared without needing to traverse with `IDTables` again.

3. Out-of-order Identification

Simply put, you can only reference something if it was declared. However, in miniJava and Java, you can reference a Class and member variables of a Class that is defined later. Thus, the traversal order requires some thought.



```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Before you visit the methods in a class, it makes sense to appropriately update your `IDTables` for level 0 and 1.

This will require manually iterating through the members (`fieldDeclList` and `methodDeclList`) and storing them in your `IDTables`. If you wish, you can ignore private methods and fields and add those only when processing that specific class. Make sure to remove those `Declaration`s from your tables when leaving that class. If not using this strategy, you can manually check to ensure private variables in other Classes are not accessed when resolving `Declaration`s later.

Note that `QualRef` (e.g. "`a.b.c.x`" on the left) is also done slightly differently. The identifier "`b`" is a local variable, but it is resolved in the context of "`a.b`" where "`a`" is declared as an "`A`" type. As such, qualified references bypass level 2+ for everything but the left-most reference (in this case, "`a`" is still a local variable).

4. More on contexts (static context, QualRef context)

In a `QualRef`: For miniJava, if the left-hand-side is a class name and NOT a 1+ variable, then the right-hand-side must be a static member. For example, when resolving "`A.x`", then `A` must be a class, and `x` must be a static member of `A`. Additionally, the left-hand-side cannot be a method (e.g. `A.someMethod.b.c` is not valid).

In a static method: You can only access static variables and other static methods. The keyword "`this`" is an identification error as it does not make sense in a static context.

5. Pre-defined names

There are no imports in miniJava, so we introduce a small number of predefined declarations to support some minimal functionality.

```
class System { public static _PrintStream out; }

class _PrintStream { public void println( int n ){} }

class String { }
```

Before processing anything, you must manually add these to your `IDTables`. While `String` has no members, it enables the declaration of your main method:

```
public static void main(String[] args)
```

Because these are predefined, there cannot exist a custom class named `String`, `_PrintStream`, or `System`. When initializing the `TypeDenoter` for `String`, change the type to a new `BaseType(TypeKind.UNSUPPORTED)` for use in type checking later.

6. Identification Implementation

Identification should be implemented through the `Visitor` interface. Create your miniJava.ContextualAnalysis package and create an `Identification.java` in that package. If doing SI, then create your `ScopedIdentification` class as well.

Carefully consider what you want your parameter and return types to be for the `Visitor` interface. Not every method will need to return something, and most will return `null`. Similarly, some methods will not use the passed parameter.

7. Type checking

Type checking starts from the leaf nodes of your AST and works bottom-up. Because Identification maps all identifiers to their declaration, the `TypeDenoter` for each identifier is also known.

The `UNSUPPORTED` type is not compatible with any type other than `ERROR`. The `ERROR` type is compatible with every type. If two types are incompatible, then report an error and the resultant type will be the `ERROR` type.

Ensure any assignment occurs with an `Expression` that resolves to the type of that variable. Calling a method also requires this type of check. There is no typecasting in miniJava. The table on the next page resolves *most* type-checking rules.

Type Checking Rules

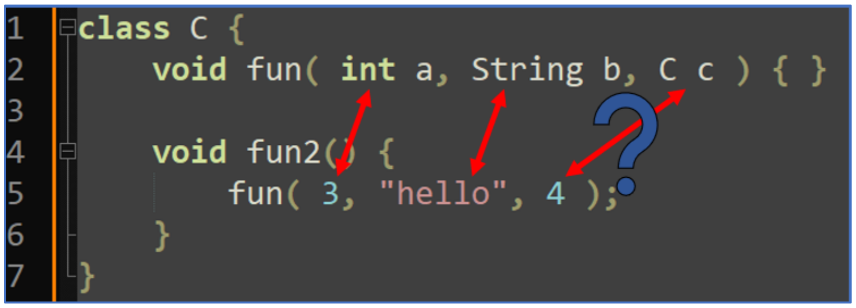
Operand Types	Operand	Result
boolean × boolean	&&, ||	boolean
int × int	>, >=, <, <=	boolean
int × int	+, -, *, /	int
$\alpha \times \alpha$	==, !=	boolean
int	(Unary) -	int
boolean	(Unary) !	boolean

Recall that `NullLiteral` can be assigned to any **object**. This means you will need a special type-checking rule when assigning/comparing an object/classtype to `null`. Similarly, you must carefully consider the type-checking of a `new` operation.

Type-Checking Implementation.

Type-checking should also implement the `Visitor` interface. Create your type-checking class inside the miniJava.ContextualAnalysis package. Most type-checking implementations return a `TypeDenoter` and do not utilize the parameter. As such, it is recommended to implement `Visitor<Object, TypeDenoter>`.

Most nodes will synthesize a `TypeDenoter`, and AST nodes that compare multiple types will ensure those types are compatible. When there is a type-checking error, you can continue type-checking.



```
class C {
    void fun( int a, String b, C c ) { }

    void fun2() {
        fun( 3, "hello", 4 );
    }
}
```

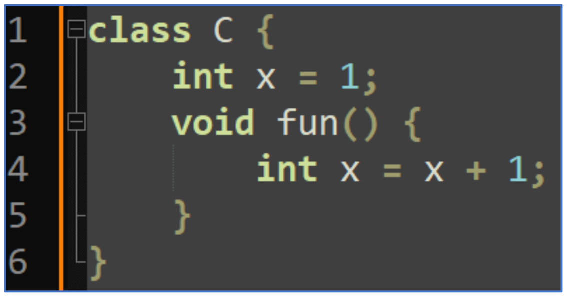
Note that calling a method also requires type checking. Ensure the number of parameters matches and the types match. Note parameter variables are being assigned expressions in a method call, thus type checking is necessary. The return statement should also match the type of the method.

8. Error Reporting

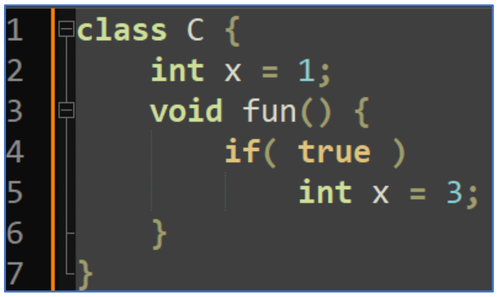
If there is an identification error, do not proceed to type-checking. If there is either a syntactic, identification, or type-checking error, output "Error" on the first line by itself (System.out.println), and any subsequent lines may contain useful output. If the input code passes all checks, output "Success" on the first line by itself.

Other Contextual Errors.

Note there are other types of contextual errors that do not fit in either Type-Checking or Identification. These other contextual can be placed in either, or a third traversal can be created. In Java, you cannot have a declaration in its own scope. You also cannot reference a variable currently being declared, even if that variable is available in an earlier scope.



```
class C {
    int x = 1;
    void fun() {
        int x = x + 1;
    }
}
```



```
class C {
    int x = 1;
    void fun() {
        if( true )
            int x = 3;
    }
}
```

In PA4, you will be required to ensure the last statement in a non-void method is a return statement. In PA5, this can optionally be made more complex by ensuring all paths have a return statement at the end, but for PA4 a simple enforced return at the end is required, and not a return within a block statement. You can choose to add this constraint now in PA3 or later in PA4, but do not do the more difficult PA5 optional constraint now, as that would interfere with the PA4 autograder.

9. Single Traversal

You can implement both type-checking and identification in a single traversal if you wish, but it is unnecessary. This is an extra credit item for PA5. In such traversals, the parameter is usually a context (an IDTable or null to signify use of ScopedIdentification) and the return type is TypeDenoter. The parameter is not always a copy of your SI object or an IDTable. There are many ways to accomplish this, but you do not have to for PA3.