COMP 520: Compilers
Compiler Project - Assignment 3

Assigned: Wed Mar 9, 2022
Due: Thu Mar 31, 11.59 PM

The goal for the miniJava project is that miniJava programs compile and execute with semantics given by the Java language. Syntactically valid miniJava programs are defined by the miniJava grammar (PA1/PA2) along with the contextual constraints specified in this assignment. There will remain a few cases where miniJava programs deviate from Java semantics.

1. miniJava extensions, syntax, and ASTs

The **`null`** literal should be added to miniJava. **`null`** can be assigned to any object or tested for (in)equality against any object. Aside from this, there are no further changes in miniJava syntax from PA2. Starting with this assignment you become the owner of the AST classes and may change them as you wish. Please maintain a summary of changes you make to AST classes since this summary will be a component of the final project submission at the end of the semester. If you add or remove AST classes, you must update the `Visitor` interface to properly traverse ASTs. While ASTs will no longer be inspected starting with this checkpoint, it may be useful to you to maintain the `ASTDisplay` capability so you can check the ASTs yourself.

2. Contextual analysis

Contextual analysis consists of identification and type checking of the AST.

2.1. Identification

Identification records declarations of names, and links uses ("applied occurrences") of names to their corresponding declaration. All declarations in a miniJava AST are subclasses of `Declaration` and every applied occurrence of a name is an `Identifier` node. Each identifier in a miniJava program should have a single corresponding declaration.

Declarations. Add a single new attribute, `Declaration decl`, to the `Identifier` class and link this attribute to the corresponding declaration node in the AST or report an error when there is no such declaration. The declaration for an identifier can vary with the identifier's position in a program statement (e.g. is it where a type is expected or where a variable name is expected). In addition, in Java and miniJava, declarations of class names and member names are not required to precede their use in the program text. Thus the traversal order requires some thought. Local variable and parameter names, on the other hand, must be declared textually before their first use. An identification example is included in Lecture 10, slides 17-18.

The declaration of an identifier may also depend on the surrounding scopes. We have the following *nesting* of scopes in a miniJava program.

1. class names
2. member names within a class
3. parameters names within a method

4+ local variable names in successively nested scopes within a method.

At each scope level we may have at most one declaration for an identifier and it can hide declarations with the same name in surrounding (lower-numbered) scope levels. However, declarations at level 4 or higher may not hide declarations at levels 3 or higher. Thus a local variable name can hide a class member name, but not a parameter name or any name declared in a surrounding scope within the same method. Duplicate declarations at the same level are erroneous.

You likely will want to implement a *scoped identification table* to support your traversal of the AST. Rather than the tedious linked-list implementation in our text, the use of a `Stack<HashMap<String,Declaration>>` representation is recommended .

Starting a statement block increases the scope level, and local variables can be declared anywhere within the statement block. Their scope is from the point of declaration forward to the end of the statement block. It is an error in Java to use the declared variable in the initializing expression. A variable declaration cannot be the solitary statement in a branch of a conditional statement.

References. A reference can denote a local variable or a method parameter, a member of the enclosing class, a class, or the current instance (i.e. the reference "`this`"). A qualified reference such as `b.x` may denote a member in another class (e.g. in the class denoted by the ClassType of b), provided it respects the visibility and access modifier in the declaration of `x`. A qualified reference to a member of another class may not have private visibility.

Each reference has a controlling declaration, e.g. the controlling declaration for reference `b.x` is the declaration of member `x` in whatever class b denotes. Thus for identification of references it makes sense to introduce an attribute `Declaration decl` in the `Reference` class as well, and to link it to the controlling declaration for the reference as part of identification.

Static members. Member declarations may specify static access. Thus identification of a reference should enforce the rules for static access. This means that in a qualified reference like `A.x`, where A denotes a class, `x` must be declared to have static access. It also means that within a **`static`** method in a class C, a reference cannot directly access a non-static member of class C.

Predefined names. There are no imports in miniJava; instead we introduce a small number of predefined declarations to support some minimal functionality normally provided by classes implicitly imported in Java. For miniJava these consist of:

```
class System { public static _PrintStream out; }
class _PrintStream { public void println(int n){}; }
class String { }
```

These declarations should be included at level 1. By the rules defined for static members, `System.out.println(42);` is a valid statement. By construction of the class name, an instance of `_PrintStream` cannot be declared in a miniJava program. Class `String` has no members, but enables the declaration of a main method **`public static void`** `main(String [] args)`.

2.2. Type checking

Type checking is performed bottom-up in the AST. The type of AST leaf nodes is known, either because the leaf is a Literal for which the type is manifest, or because it is an Identifier, for which we know the declaration and hence its type. The type of a non-leaf node is determined from the types of its children.

A simple strategy is to introduce a `type` attribute in appropriate AST classes, and use the type rules to set the value of this attribute in a bottom up traversal.

You should define a method to determine equality between types. Recall that, in Java, type equality is by name. In addition to the miniJava types described in PA1-PA2, you may wish to use two additional types. The *ERROR* type is equal to any type and can be used to limit the cascading of errors in contextual analysis once a type error is found. The *UNSUPPORTED* type is not equal to any type, hence values of this type when referenced should generate an error. You should assign `String` the *UNSUPPORTED* type, so that any attempts to reference such values generate an error. The *ERROR* and *UNSUPPORTED* types are included in the `TypeKind` enumeration)

2.3. Implementation of contextual analysis

You may implement contextual analysis in a single traversal that performs identification and type checking simultaneously, or you may implement it using two separate traversals. The latter will result in more code, but may be simpler to construct, understand, and extend.

Identification will use the scoped identification table and should have access to specific identification tables such as the list of ClassDecls for all classes and the list of MemberDecls for a given class. The idTables may be accessed during AST traversal through fields in the visitor, in which case identification can be an implementation of `Visitor<Object,Object>` where the arg and result of each visit method are always null. Alternatively an idTable can be supplied as an inherited attribute and returned as a synthesized attribute, in which case identification traversal can be an implementation of `Visitor<IdTable,IdTable>`. It is more efficient to use a single global scoped identification table during traversal rather than to attach idTable attributes to each node.

For type checking, each node synthesizes a `TypeDenoter` attribute from the types returned by its children, so a traversal could be constructed as an implementation of `Visitor<Object,Type>` where the argument to a visit method will always be null, and the result is always a `Type`.

The predefined class declarations allow us to introduce a mainclass with a suitable main method. However, we do not verify the existence of a mainclass and main method in contextual analysis (this is true of Java as well). This requirement will be included in PA4.

3. Reporting results

The Compiler.java mainclass in your compiler will function much like it did in PA2, exiting with termination code 0 in the case of a valid miniJava program (i.e. following syntactic and contextual analysis), and exiting with termination code 4 in the case of an invalid miniJava program. There are two changes: (1) there is no need to display the AST in the case of a valid miniJava program and (2) an input program that is found to be invalid in contextual analysis should write a text error message to stdout (not stderr). For example, if line 5 of the input program contains the assignment **`boolean`** `a = 3;` the output might be

```
*** line 5: Type error – incompatible types in assignment
```

The first nine characters of the error report (“`*** line `”) should follow the exact form shown here, and must be followed by the line number and a colon. Any text thereafter is free-form and terminated by a newline. Your contextual analyzer should attempt to check the entire program, even after an error is encountered. However, it is reasonable to stop contextual analysis upon failure in identification, because it may render further checking meaningless.