COMP 520: Compilers
Compiler Project - Assignment 3

Assigned: Thu Feb 20
Due: Sat Mar 22

The goal of the compiler project is that valid miniJava programs are compiled and execute following semantics given by the full Java language. Valid miniJava programs are defined by the grammar (PA1/PA2) and the contextual constraints specified in this assignment. There remain cases where we deviate from Java semantics. However, within limits, valid miniJava programs should give the same results using your compiler as when compiled using java.

1. miniJava syntax and ASTs

There are no further changes in miniJava syntax from PA2. Starting with this assignment you become 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. Also, when you add or delete classes, make sure you update the `Visitor` interface so all AST classes can be visited. While we will no longer check ASTs starting with PA3, it may be useful to you to maintain the `ASTDisplay` capability so you can continue to use it.

2. Contextual analysis

The PA3 assignment asks you to perform contextual analysis on the AST constructed in PA2. Contextual analysis consists of identification and type checking of the AST which should use the visitor interface in the miniJava AST classes.

2.1 Identification

Identification records defining occurrences of names, and links applied occurrences of names to their corresponding declaration. In the minJava AST every defining occurrence of a name is in a `Declaration` node and (nearly) every applied occurrence of a name is an `Identifier` node. An identifier in a miniJava program cannot be multiply declared or overloaded, so it has at most one corresponding declaration.

Thus for identification it suffices to introduce a single new attribute `Declaration decl` in the `Identifier` class and to traverse the AST to link this attribute to the corresponding declaration or to report an error when there is no such declaration.

However, the controlling declaration for an identifier can vary with the identifier's position in the source program (e.g. is it where a type is expected or where a variable name is expected). Furthermore, 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 requires some thought. Local variable and parameter names, on the other hand, must be declared textually before their first use.

The controlling definition 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 successive nested scopes within a method.

At each scope level we may have at most one declaration for a name and it may hide declarations in surrounding scope levels. However, declarations at level 4 or above 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 will likely want to implement a scoped identification table to support this part of this project. Efficiency of the identification table is not an issue for now.

A local variable can be declared anywhere in a statement block and has scope from the point of declaration forward to the end of the most immediate surrounding statement block. It is an error in Java to use the variable being declared in the initializing expression. Also, a variable declaration can not be the solitary statement in a branch of a conditional statement (why?).

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 of the form `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 modifier in the declaration of `x`. An indexed reference of the form `b[i]` denotes a reference to an element of array `b`.

In general (but not always), a 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 also makes sense to introduce an attribute `Declaration decl` in the `Reference` class 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 that is 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 { }
```

Note that by the rules defined for static members, the reference `System.out.println` is a valid reference since `println` does not have static access in `_PrintStream`. By construction of the class name, no instance of `_PrintStream` can be declared in a miniJava program. The class `String` has no members (for now).

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 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 thus far, you may wish to introduce 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. The unsupported type can be used for features that have not yet been implemented.

2.3. Implementation of contextual analysis

Contextual analysis can be implemented in a single traversal that performs identification and type checking simultaneously, or it can be implemented as two separate traversals. The latter may result in more code, but it may be simpler to construct and understand.

Identification uses one or more idTables to record and lookup names. These idTables can be accessed from an attribute in the visitor and/or can be supplied as an inherited attribute in a traversal. Generally no attribute is synthesized in identification. So Identification could be implemented using a `Visitor<IdentificationTable,Object>` where the result is of each visit method is always null.

For type checking, each node synthesizes a `Type` attribute from the types returned by its children. Thus the traversal could be implemented using a `Visitor<Object,Type>` where the argument to a visit method will always be null, but the result is always a `Type` value.

A `Visitor<IdentificationTable,Type>` could be used to implement a single pass contextual analyzer.

main() method. There should be exactly one **`public static void`** `main()` method among the classes in a miniJava program, and it must have the correct `String []` type for its single parameter. This should be checked as part of contextual analysis. The unique main method will be important to know in code generation.

3. Reporting errors in contextual analysis

Your contextual analyzer should attempt to check the entire program, even when errors are encountered. However, it is permissible to stop contextual analysis as a result of a failure in identification, because it can render further checking meaningless. Any error report issued by the contextual analyzer should start with three asterisks "***". In the grading, accurate identification of the error (e.g. by identifying the error location and parts of the problem) will receive more points than a vague error. As in previous submissions, an exit 0 should signal a valid miniJava program, and an exit 4 signals errors were encountered in miniJava syntactic or contextual analysis. You should not write out the AST in this submission.