COMP 520: Compilers
Compiler Project - Assignment 3

Assigned: Tue Feb 19
Due: Thu Mar 21

The goal of our compiler project is that any valid miniJava program should have 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 some cases where we deviate from Java semantics, for example, Java has many more keywords than miniJava, thus some miniJava identifiers could be keywords in Java that would have a different meaning. 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. This assignment will suggest that you rewrite the ASTs for references using additional classes. 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 checks the validity of declarations, and links names to their defining declaration. In Java and miniJava, declarations of class names and member names are not required to precede their use in the program text. Local variable and parameter names, on the other hand, must be declared textually before their first use.

A name in at a given position in a miniJava program must have a single controlling declaration, so it cannot be multiply declared or overloaded. However, the controlling declaration for a name can vary with its position in the source program (e.g. does it occur where a typename is expected or where a variable name is expected) and with the surrounding scopes.

Type names are defined in a class declaration (`class Foo { … }`) and are used within `ClassType` instances that may appear in a variable declaration and in other places (where?). Type names may also appear in qualified references. Identification must link each type name to the corresponding `ClassDecl`. An attribute to record this link is available in the AST node `ClassType`.

Variable or member names are defined in variable or member declarations and are used in identifiers that appear in references. Identification must link each such identifier to its corresponding declaration. An attribute to record this link is available in the AST node `Identifier`.

The controlling definition of an identifier may depend on the surrounding scopes. We have the following *nesting* of scopes in a miniJava class.

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

3+ 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 3 or above may not hide declarations at levels greater than 1. 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. 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 type name, or the current instance (i.e. the reference "`this`"). A 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 that the visibility and access modifiers are respected. An indexed reference of the form `b[i]` denotes a reference to element `i` of array b.

You may benefit from rewriting your AST to replace `QualifiedRef` nodes using a new set of `Reference` subclasses that more directly reflect the kinds of references we may encounter. (The parser can not build the AST this way, since it requires identification to tell apart the different kinds of references). We could introduce four `Reference` subclasses `LocalRef`, `MemberRef`, `ClassRef` and `ThisRef`, corresponding to the first four cases in the previous paragraph. In addition, you could introduce class `DeRef` to access a member of a reference to a class type. These five classes could be used together to describe an arbitrary `QualifiedRef`. The `IndexedRef` could continue to be used to index a reference that is an array type.

To replace the AST for a `QualifiedRef` *qr* in an AST node with a new `Reference` AST constructed using the new classes, the new AST could be returned as a result of the identification of reference *qr*. This would require a visitor with result type `Reference`, for example declared as `Visitor<Object,Reference>`. On return from `qr.visit()` the caller can replace *qr* with the returned AST. Reference ASTs constructed using these new classes correspond more directly to the structure needed for type checking and code generation.

Static members. Members may be *declared* to have static access. Thus a *reference* should obey the rules for static access. This means only static members maybe referenced from a ClassRef, while all (visible) members may be referenced from any other reference. It also means that within a `static` method in a class *C*, a MemberRef may not reference a non-static member. Finally, as a restriction to be enforced in PA3, a reference in a RefExpr CallExpr, CallStmt, or AssignStmt that yields a reference to a member with static access should yield a "PA3 no static

access" error. As a consequence, static methods cannot be invoked in PA3, and static fields cannot be used as a value or assigned a value. 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.

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

```
class _PrintStream
```

with member

```
public void println(int n);
```

By construction of the class name, no instance of `_PrintStream` can be declared in a miniJava program. We also need a type `String` with no members (for now) and

```
class System
```

with solitary member

```
public static _PrintStream out;
```

Note that by the rules defined for static members, the reference `System.out.println` is a valid reference in a CallStmt since `println` does not have static access in `_PrintStream`.

2.2. Type checking

To keep things simple, you can define a separate AST traversal for type checking. The traversal is bottom-up, each node synthesizes a `Type` attribute from the types returned by its children. Thus the traversal could be implemented using `Visitor<Object,Type>` where the argument type will always be null.

Introduce an attribute to record the type in appropriate AST classes, and use type rules to determine the value of this attribute. Following identification, each literal and each identifier can be associated with a type, and the types of other AST nodes can be determined using type rules for miniJava statements and expressions. The type rules for miniJava are simple since there is no overloading.

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 unsupported the error type. One use for the unsupported type is for the `String` type used in the solitary parameter in `main()`. This allows the main method to be declared, but any reference to its parameter will yield a type error.

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 "`***`". 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.