COMP 520: Compilers

Compiler Project - Assignment 3

Assigned: Tue Feb 26
Due: Mon Mar 25 11.59 PM

Our goal in the compiler project is that valid miniJava programs compile and execute with semantics given by the full Java language. Valid miniJava programs are defined by the 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 should update the `Visitor` interface so ASTs can be properly traversed. 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 still check ASTs yourself.

2. Contextual analysis

The PA3 assignment asks you to perform contextual analysis on the AST. Contextual analysis consists of identification and type checking.

2.1. Identification

Identification records declarations of names, and links uses ("applied occurrences") of names to their corresponding declaration. In the minJava AST every declaration of a name is a `Declaration` node and every applied occurrence of a name is an `Identifier` node. An identifier at a specific location in a miniJava program has a single corresponding declaration.

Thus for identification it suffices to introduce a single new attribute `Declaration decl` in the `Identifier` class and to link this attribute to the corresponding declaration node 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 example was shown in Lecture 10, slides 17-18.

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 successively 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 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. The use of a `Stack<HashMap<String,Declaration>>` representation is recommended in place of the book's tedious linked list implementation.

Starting a statement block increases the scope level, and local variables can be declared anywhere in the statement block with their scope being from the point of declaration forward to the end of the statement block. However, it is an error in Java to use the variable being declared in the initializing expression. Also, a variable declaration cannot 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`"). An indexed reference of the form `b[i]` denotes a reference to an element of array `b`. 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 and access modifier in the declaration of `x`. A qualified reference to a member of another class must 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 and to link it to the controlling declaration for a 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 { }
```

You can consider the names introduced here as a new scope at level 0, since they may be hidden by class declarations in a miniJava program. 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 (these are already present in the `TypeKind` enumeration). 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.

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 or more separate traversals. The latter may result in more code, but it will be simpler to construct, understand, and extend.

The identification traversal uses a scoped identification table and should also have access to specific identification tables such as the table of ClassDecls and the list of MemberDecls for a given class. The idTables can be accessed during AST traversal through fields in the visitor, in which case an identification traversal can be an implementation of `Visitor<Object,Object>` where the arg and result of each visit method are always null. Alternatively the idTable can be supplied as an inherited attribute and returned as a synthesized attribute, in which case an identification traversal could be constructed as 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 names 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 will be a requirement we add in the code generation stage.

3. Reporting errors in contextual analysis

Your contextual analyzer should attempt to check the entire program, even when errors are encountered. However, it is reasonable to stop contextual analysis as a result of a failure in identification, because it can render further checking meaningless. In your miniJava compiler, any error report issued by the contextual analyzer should start with three asterisks "`***`". The error report should be written to stdout, and not to stderr. In grading, accurate identification of the problem including its location in the source program will receive more points than a vague or simply incorrect error. At a minimum include the line number in the source file where the error occurs. As in previous submissions, an exit code 0 indicates a valid miniJava program, and an exit code 4 indicates errors encountered in miniJava syntactic or contextual analysis. It is not necessary to write out an AST in your PA3 submission.