4. Type Checking

4.1 Introduction

Type checking, or more formally semantic analysis, is the final step in the analysis portion of compilation. It is the compiler's last chance to collect that information necessary to begin the synthesis phase. Semantic analysis includes the following:

- Determining the types of all names and expressions.
- Type checking: insuring that all expressions are properly typed, for example that the operands of an operator have the proper types.
- A certain amount of storage analysis, for example determining the amount of storage that is required in the current stack frame to store a local variable (one word for **`int`**s, two words for **`long`**s). This information is used to allocate locations (at offsets from the base of the current stack frame) for parameters and local variables.
- A certain amount of AST tree rewriting, usually to make implicit constructs more explicit.

4.2 Semantic Analysis in j--

Semantic analysis of *j--* programs involves all of these operations.

- Like Java, *j--* is strictly-typed; that is, we want to determine the types of all names and expressions at compile time.
- A *j--* program must be well-typed.
- All *j--* local variables (including formal parameters) must be allocated storage and locations within a method's stack frame.
- The AST for *j--* requires a certain amount of sub-tree rewriting. For example, field references using simple names must be rewritten as explicit field selection operations. And declared variable initializations must be rewritten as explicit assignment statements.

4.2.1 The *j--* Types

4.2.1.1 Introduction to *j--* Types

A type in *j--* is either a primitive type or a reference type.

j-- primitive types

- **`int`** -- 32 bit two's complement integers.

- **`boolean`** -- taking the value true or false.
- **`char`** -- 16 bit Unicode (but most systems deal only with the lower 8 bits).

j-- reference types

- arrays
- objects of a type described by a class declaration.
- built-in objects **`java.lang.Object`** and **`java.lang.String`**.

j-- code may interact with classes from the Java library but it must be able to do so using only these types.

4.2.1.2 The Type Representation Problem

The question arises: how do we represent a type in our compiler? For example, how do we represent the types **`int`**, **`int[]`**, **`Factorial`**, **`String[][]`**? The question must be asked in light of two desires:

1. We want a simple, but extensible representation. We want no more complexity than is necessary for representing all of the types in *j--* and for representing any (Java) types that we may add in exercises.

2. We want the ability to interact with the existing Java class libraries.

Two solutions come immediately to mind:

1. Java types are represented by objects of (Java) type **`java.lang.Class`**. **`Class`** is a class defining the interface necessary for representing Java types. Since *j--* is a subset of Java, why not use **`Class`** objects to represent its types? Unfortunately, the interface is not as convenient as we might like.

2. A homegrown representation may be simpler. One defines an abstract class (or interface) **`Type`**, concrete sub-classes (or implementations) **`PrimitiveType`** and **`ReferenceType`**, and further **`ArrayType`**, etc.

4.2.1.3 Type Representation and Class Objects

Our solution is to define our own class **`Type`** for representing types, with a simple interface but also encapsulating the **`java.lang.Class`** object that corresponds to the Java representation for that same type.

But the parser does not know anything about types. It knows neither what types have been declared nor which types have been imported. For this reason we define two placeholder type representations:

1. **`TypeName`** -- for representing named types recognized by the parser like user-defined classes or imported classes until such time as they may be resolved to their proper **`Type`** representation.
2. **`ArrayTypeName`** -- for representing array types recognized by the parser like **`int[][]`** or **`String[]`**, until such time that they may resolved to their proper **`Type`** representation.

During analysis, **`TypeName`**s and **`ArrayTypeName`**s are *resolved* to the **`Type`**s that they represent. Type resolution involves looking up the type names in the symbol table to determine which defined type or imported type they name. More specifically,

- A **`TypeName`** is resolved by looking it up in the current context, our representation of our symbol table. The **`Type`** found replaces the **`TypeName`**[1]. Finally, the **`Type`**'s accessibility from the place the **`TypeName`** is encountered is checked.

- An **`ArrayTypeName`** has a base type. First the base type is resolved to a **`Type`**, whose **`Class`** representation becomes the base type for a new **`Class`** object for representing the array type[2]. Our new **`Type`** encapsulates this **`Class`** object.

- A **`Type`** resolves to itself.

So that **`ArrayTypeName`**s and **`TypeName`**s may stand in for **`Type`**s in the compiler, both are subclasses of **`Type`**.

One might ask why the *j--* compiler does not simply use Java's Class objects for representing types. The answer is two-fold:
1. Our **`Type`** defines just the interface we need.
2. Our **`Type`** permits the Parser to use its sub-types **`TypeName`** and **`ArrayTypeName`** in its place when denoting types that have not yet been resolved.

4.2.2 *j--* Symbol Tables

In general, a symbol table maps names to the things they name, for example, types, formal parameters and local variables. These mappings are established in a declaration and consulted each time a declared name is encountered.

---

[1] Even though externally defined types must be explicitly imported, if the compiler does not find the name in the symbol table, it attempts to load a class file of the given name and, if successful, declares it.

[2] Actually, because Java does not provide the necessary API for creating **`Class`** objects that represent array types, we create an *instance* of that array type and use **`getClass()`** to get its type's **`Class`** representation.

4.2.2.1 Contexts And Idefns: Declaring and Looking Up Types and Local Variables

In the *j--* compiler, the symbol table is a tree of **`Context`** objects, which spans the abstract syntax tree (AST). Each **`Context`** corresponds to a region of scope in the *j--* source program and contains a map of names to the things they name.

For example, reconsider the simple Factorial program. In this version we mark two locations in the program using comments: position 1 and position 2.

```
package pass;

import java.lang.System;

public class Factorial
{
    // Two methods and a field

    public static int factorial(int n)
    {
        // position 1:
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args)
    {
        int x = n;
        // position 2:
        System.out.println(n + "! = " + factorial(x));
    }

    static int n = 5;
}
```

The symbol table for this program, and its relationship to the AST, is illustrated in Figure 4.1.

In its entirety, the symbol table takes the form of a *tree* that corresponds to the shape of the AST. A context, that is a node in this tree, captures the region of scope corresponding to the AST node that points to it. For example, in Figure 4.1,

1. the context pointer from the AST's **`JCompilationUnit`** node points to the **`JCompilationUnitContext`**, which is at the root of the symbol table,

2. the context pointer from the AST's **`JClassDeclaration`** points to a **`ClassContext`**,
3. the context pointer from the AST's two **`JMethodDeclaration`**s each point to a **`MethodContext`**, and
4. the context pointer from the AST's two **`JBlock`**s each point to a **`LocalContext`**.

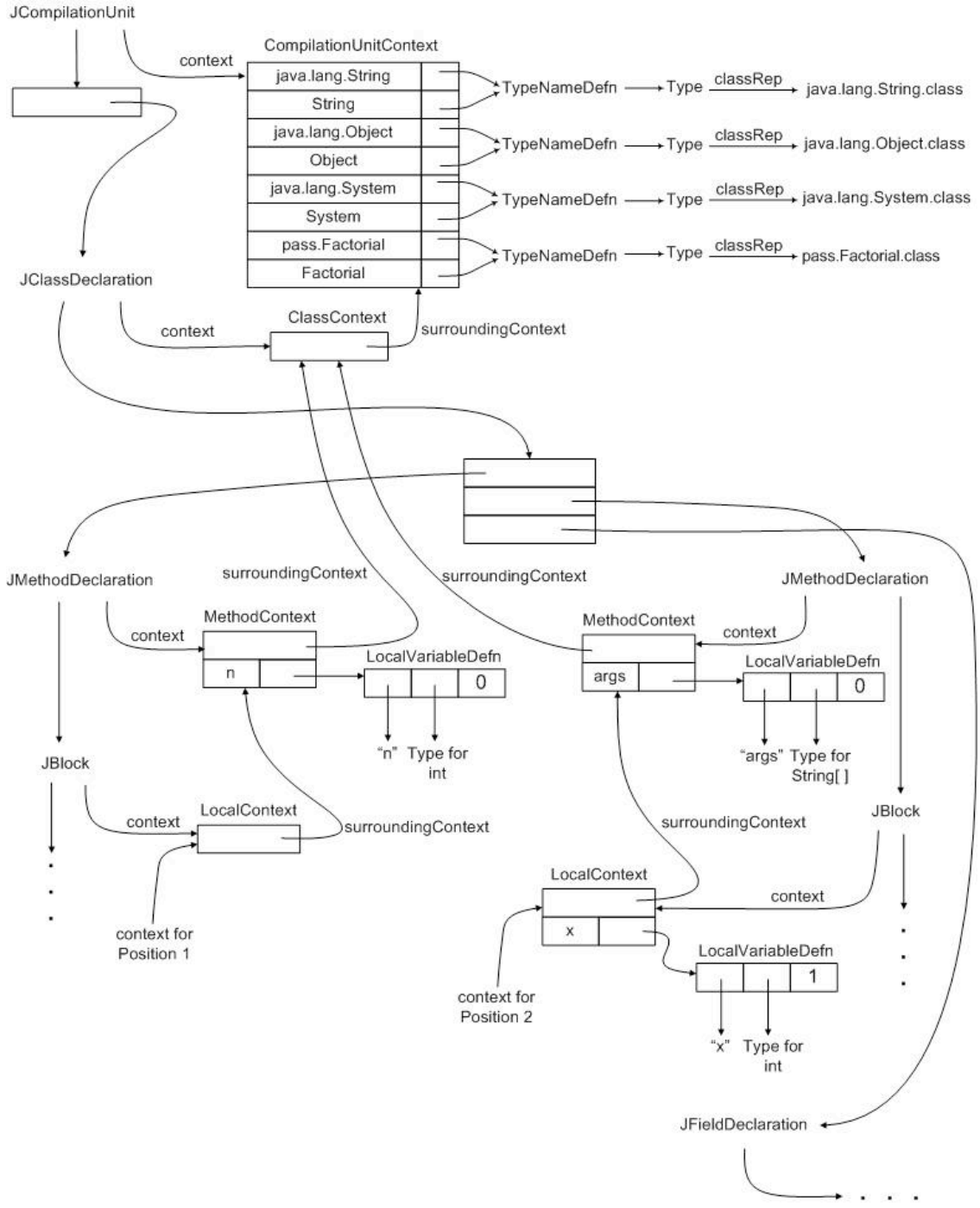

Figure 4.1 The Symbol Table for the Factorial Program

On the other hand, from any particular location in the program, looking back towards the root **`CompilationUnitContext`**, the symbol table looks like a *stack* of contexts. Each (**`surroundingContext`**) link back towards the **`CompilationUnitContext`** points to the context representing the surrounding lexical scope.

For example,

1. the *context for Position 2* pointer in Figure 4.1 points to a **`LocalContext`**, which declares the local variable **`x`** in the body of the **`main()`** method of the Factorial program;
2. its **`surroundingContext`** pointer points to a **`MethodContext`** in which the formal parameter, **`args`** is declared;
3. its **`surroundingContext`** pointer points to a **`ClassContext`** in which nothing is declared; and
4. its **`surroundingContext`** pointer points to a **`CompilationUnitContext`** at the base of the stack, which contains the declared types for the entire program.

During analysis, when the compiler encounters a variable, it looks up that variable in the symbol table by name, beginning at the **`LocalContext`** most recently created in the symbol table, for example that pointed to by the pointer labeled "context for Position 1." Type names are looked up in the **`CompilationUnitContext`**. To make this easier, each context maintains three pointers to surrounding contexts, as illustrated in Figure 4.2.

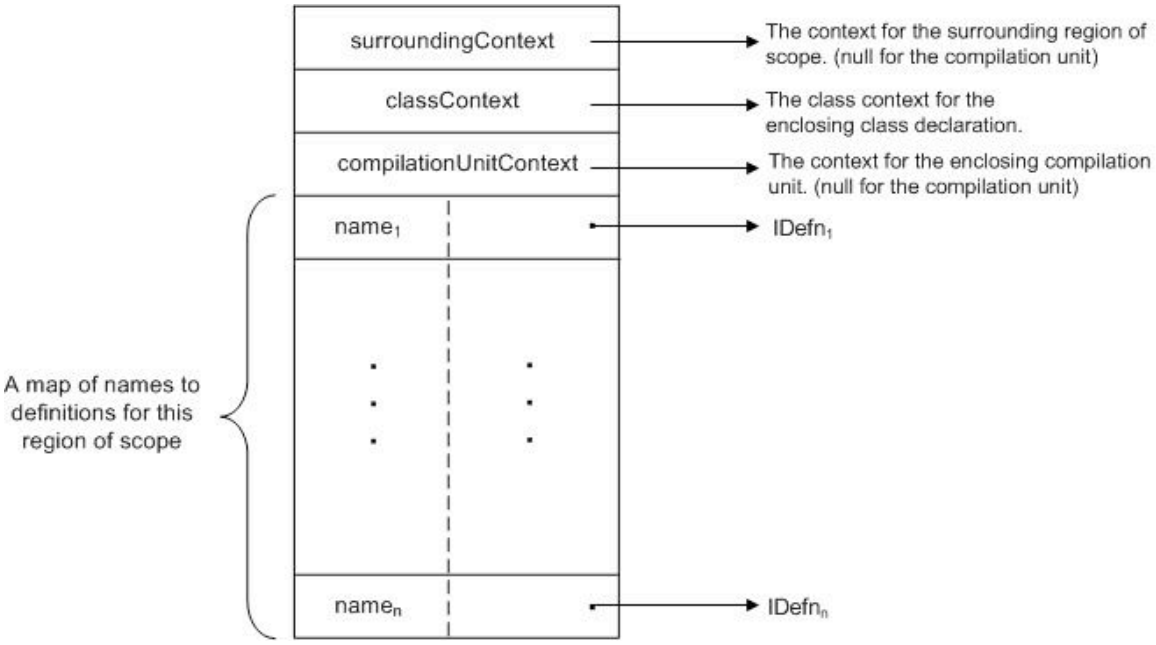

Figure 4.2 The Structure of a context

The pointer, **`surroundingContext`**, points to the context for the surrounding region of scope; the chain of these pointers is the stack that captures the nested scope in *j--* programs. The pointer, **`compilationUnitContext`**, points to the context for the enclosing compilation unit, i.e. a **`CompilationUnitContext`**. In the current definition of *j--* there is just one compilation unit but one might imagine an extension to *j--* permitting the compilation of several files, each of which defines a compilation unit. The pointer, **`classContext`**, points to the context (a **`ClassContext`**) for the enclosing class. As we shall see below, no names are declared in a **`ClassContext`** but this could change if we were to add nested type declarations to *j--*.

A **`CompilationUnitContext`** represents the scope of the entire program and contains a mapping from names to types:

- the implicitly declared types, **`java.lang.Object`**, and **`java.lang.String`**,
- imported types, and
- user defined types, i.e. types introduced in class declarations.

A **`ClassContext`** represents the scope within a class declaration. In the *j--* symbol table, no names are declared here. All members, that is all constructors, methods and fields are recorded in the **`Class`** object that represents the type; we discuss this in the next section. If we were to add nested type declarations to *j--,* they would be declared here.

A **`MethodContext`** represents the scope within a method declaration. A method's formal parameters are declared here. A **`MethodContext`** is a kind of **`LocalContext`**.

A **`LocalContext`** represents the scope within a block, that is the region between two curly brackets { … }. This includes the block defining the body to a method. Local variables are declared here.

Each kind of context derives from (extends) the class **`Context`**, which supplies the mapping from names to definitions (**`IDefn`**s). Because a method defines a local context, a **`MethodContext`** derives from a **`LocalContext`**. The inheritance tree for contexts is illustrated in Figure 4.3

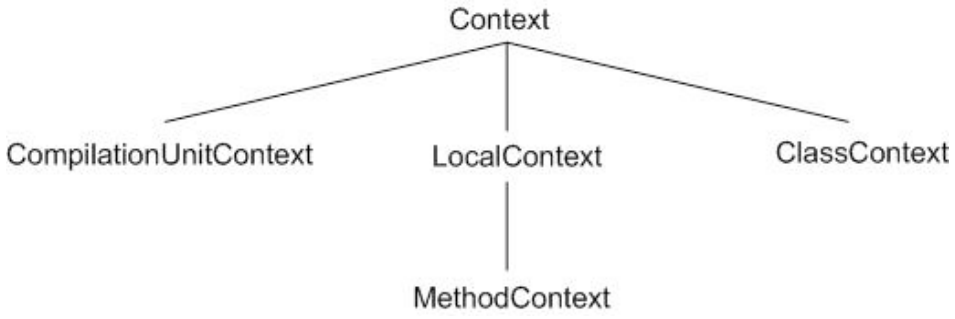

Figure 4.3 The inheritance tree for contexts

An **`IDefn`** is the interface type for symbol table definitions and has two implementations:

1. A **`TypeNameDefn`**, which defines a type name. An **`IDefn`** of this sort encapsulates the **`Type`** that it denotes.
2. A **`LocalVariableDefn`** defines a local variable and encapsulates the name, its **`Type`** and an offset in the current run-time stack frame. (We discuss stack frame offsets in section 4.2.4.)

4.2.2.2 Finding Method and Field Names in Type Objects

As we discussed in section 4.2.1.2, the types defined by classes, are represented in the same way as they are in the Java library, so that we may consistently manipulate types that we define and types that we import.

Class member (field and method in *j--)* names are not declared in a **`ClassContext`**, but in the **`Type`**s that they declare. We rely on the encapsulated **`Class`** object to store the interface information, and we rely on Java *reflection* to query a type for information about its members.

For example, **`Type`** supports a method **`fieldFor()`** which, when given a name returns a **`Field`** with the given name that is defined for that type. It delegates the finding of this field to a search of the underlying **`Class`** object and can be seen from the code that defines **`fieldFor()`**:

```
public Field fieldFor(String name)
{
    Class<?> cls = classRep;
    while (cls != null) {
        java.lang.reflect.Field[] fields =
            cls.getDeclaredFields();
        for (java.lang.reflect.Field field:fields) {
            if (field.getName().equals(name)) {
                return new Field(field);
            }
        }
        cls = cls.getSuperclass();
    }
    return null;
}
```

This code first looks for the named field in the **`Class`** object for the **`Type`** being defined. If it does not find it there, it looks in the **`Class`** object for the **`Type`**'s super type, and so on until either the field is found or we come to the base of the inheritance tree, in which

case **`null`** is returned to indicate that no such field was found. If we find the **`java.lang.reflect.Field`**, we encapsulate it within our own locally defined **`Field`**. More interface for querying a **`Type`** about its members is implemented, delegating the reflection to the underlying **`Class`** object that is kept in the **`Type`**'s **`classRep`** field.

The **`Class`** objects are created for declared classes in the **`preAnalyze()`** phase and are queried during the **`analyze()`** phase. This is made possible by the **`CLEmitter`**'s ability to create partial class files – class files that define the headers for methods but not the bodies and the types of fields but not their initializations. These analysis issues are discussed in the next two sections.

4.2.3 Pre-Analysis of *j--* Programs

4.2.3.1 An Introduction to Pre-Analysis

The semantic analysis of *j--* (and Java) programs requires two traversals of the AST because a class name or a member name may be referenced before it is declared in the source program. The traversals are accomplished by methods (the method **`preAnalyze()`** for the first traversal and the method **`analyze()`** for the second), which invoke themselves at the child nodes for recursively descending the AST to its leaves.

But the first traversal need not traverse the AST so deeply as the second traversal. The only names that may be referred to before they are declared are type names (i.e. class names in *j--*) and members.

So **`preAnalyze()`** must traverse down the AST only far enough for

- declaring imported type names,
- declaring user defined class names.
- declaring fields and
- declaring methods (including their *signatures* -- the types of their parameters).

For this reason, **`preAnalyze()`** need be defined only in the following types of AST nodes:

- **`JCompilationUnit`**,
- **`JClassDeclaration`**,
- **`JFieldDeclaration`**,
- **`JMethodDeclaration`**, and
- **`JConstructorDeclaration`**.

So the **`preAnalyze()`** phase descends recursively down the AST only so far as the member declarations, but not into the bodies of methods.

4.2.3.2 JCompilationUnit.preAnalyze()

For the **`JCompilationUnit`** node at the top of the AST, **`preAnalyze()`** does the following:

1. It creates a **`CompilationUnitContext`**.
2. It declares the implicit *j*-- types, **`java.lang.String`** and **`java.lang.Object`**.
3. It declares any imported types.
4. It declares the types defined by class declarations. Here it creates a **`Type`** for each declared class, whose **`classRep`** refers to a **`Class`** object for an empty class. For example, at this point in the pre-analysis phase of our Factorial program above, the **`Type`** for **`Factorial`** would have a **`classRep`**, the **`Class`** object for the class,

   ```
   class Factorial {}
   ```

   Of course, later on, in both **`analyze()`** and **`codeGen()`**, the class will be further defined.
5. Finally, **`preAnalyze()`** invokes itself for each of the type declarations in the compilation unit. As we shall see below (section 4.2.3.3), this involves, for class declaration, creating a new **`Class`** object that records the interface information for each member and then overwriting the **`classRep`** for the declared **`Type`** with this new (more fully defined) **`Class`**.

Here is the code for **`preAnalyze()`** in **`JCompilationUnit`**:

```
public void preAnalyze()
{
    context = new CompilationUnitContext();

    // Declare the two implicit types java.lang.Object and
    // java.lang.String
    context.addType(0, Type.OBJECT);
    context.addType(0, Type.STRING);

    // Declare any imported types
    for (TypeName imported: imports) {
        try {
            Class<?> classRep =
                Class.forName(imported.toString());
            context.addType(imported.line(),
                Type.typeFor(classRep));
        }
        catch (Exception e) {
            JAST.compilationUnit.reportSemanticError(
                imported.line(),
                "Unable to find %s", imported.toString());
```

```
        }
    }

    // Declare the locally declared type(s)
    CLEmitter.initializeByteClassLoader();
    for (JAST typeDeclaration: typeDeclarations) {
        ((JTypeDecl)
              typeDeclaration).declareThisType(context);
    }

    // Pre-analyze the locally declared type(s). Generate
    // (partial) Class instances, reflecting only the member
    // interface type information
    CLEmitter.initializeByteClassLoader();
    for (JAST typeDeclaration: typeDeclarations) {
        ((JTypeDecl)
          typeDeclaration).preAnalyze(context);
    }
}
```

4.2.3.3 JClassDeclaration.preAnalyze()

In a class declaration, **`preAnalyze()`** does the following:

1. It creates a new **`ClassContext`**, whose **`surroundingContext`** points to the **`CompilationUnitContext`**.
2. It resolves the class's super type.
3. It creates a new **`CLEmitter`** instance, which will eventually be converted to the **`Class`** object for representing the declared class.
4. It adds a class header, defining a name and any modifiers, to this CLEmitter instance.
5. It recursively invokes **`preAnalyze()`** on each of the class's members. This causes field declarations, constructors and method declarations (but with empty) bodies to be added to the **`CLEmitter`** instance.
6. If there is no explicit constructor (having no arguments) in the set of members, it adds the implicit constructor to the **`CLEmitter`** instance. For example, for the Factorial program above, the following implicit constructor is added, even though it is never used in the Factorial program:

```
public Factorial()
{
    super.Factorial();
}
```

7. Finally, the **`CLEmitter`** instance is converted to a **`Class`** object, and that replaces the **`classRep`** for the **`Type`** for the declared class name in the (parent) **`ClassContext`**.

The code for **JClassDeclaration**'s **preAnalyze()** is as follows.

```
public void preAnalyze(Context context)
{
    // Construct a class context
    this.context = new ClassContext(this, context);

    // Resolve superclass
    superType = superType.resolve(this.context);

    // Creating a partial class in memory can result in a
    // java.lang.VerifyError if the semantics below are
    // violated, so we can't defer these checks to analyze()
    thisType.checkAccess(line, superType);
    if (superType.isFinal()) {
        JAST.compilationUnit.reportSemanticError(line,
            "Cannot extend a final type: %s",
            superType.toString());
    }

    // Create the (partial) class
    CLEmitter partial = new CLEmitter();

    // Add the class header to the partial class
    String qualifiedName =
        JAST.compilationUnit.packageName() == "" ? name :
            JAST.compilationUnit.packageName() + "/" + name;
    partial.addClass(mods, qualifiedName, superType.jvmName(),
        null, false);

    // Pre-analyze the members and add them to the partial class
    for (JMember member: classBlock) {
        member.preAnalyze(this.context, partial);
        if (member instanceof JConstructorDeclaration &&
             ((JConstructorDeclaration) member).
                 params.size() == 0) {
            hasExplicitConstructor = true;
        }
    }

    // Add the implicit empty constructor?
    if (!hasExplicitConstructor) {
        codegenPartialImplicitConstructor(partial);
    }

    // Get the Class rep for the (partial) class and make it the
    // representation for this type
    Type id = this.context.lookupType(name);
    if (id != null &&
```

```
         !JAST.compilationUnit.errorHasOccurred()) {
        id.setClassRep(partial.toClass());
    }
}
```

4.2.3.4 JMethodDeclaration.preAnalyze()

Here is the code for **preAnalyze()** in **JMethodDeclaration**:

```
public void preAnalyze(Context context, CLEmitter partial)
{
    // Resolve types of the formal parameters
    for (JFormalParameter param: params) {
        param.setType(param.type().resolve(context));
    }

    // Resolve return type
    returnType = returnType.resolve(context);

    // Check proper local use of abstract
    if (isAbstract && body != null) {
        JAST.compilationUnit.reportSemanticError(line(),
            "abstract method cannot have a body");
    }
    else if (body == null && ! isAbstract) {
        JAST.compilationUnit.reportSemanticError(line(),
            "Method with null body must be abstract");
    }
    else if (isAbstract && isPrivate ) {
        JAST.compilationUnit.reportSemanticError(line(),
            "private method cannot be declared abstract");
    }
    else if (isAbstract && isStatic ) {
        JAST.compilationUnit.reportSemanticError(line(),
            "static method cannot be declared abstract");
    }

    // Compute descriptor
    descriptor = "(";
    for (JFormalParameter param: params) {
        descriptor += param.type().toDescriptor();
    }
    descriptor += ")" + returnType.toDescriptor();

    // Generate the method with an empty body (for now)
    partialCodegen(context, partial);
}
```

Basically, **`preAnalyze()`** does the following in a method declaration:

1. It resolves the types of its formal parameters and its return type.
2. It checks that any **`abstract`** modifier is proper.
3. It computes the *method descriptor*, which codifies the method's signature as a String[3]. For example, in the Factorial program above,
   - method **`factorial()`** has the descriptor **`(I)I`**, which indicates a method taking an **`int`** for an argument and returning an **`int`** result, and
   - method **`main()`** has the descriptor **`([Ljava.lang.String;)V`**, which indicates a method taking a **`String[]`** argument and not returning anything (i.e., a **`void`** return type).
4. Finally, it calls upon **`partialCodegen()`** to generate code for the method, but without the body. So the **`Class`** object that is generated for the enclosing class declaration has, after pre-analysis at least the interface information for methods, including the types of parameters and the return type.

The code for **`partialCodegen()`** is as follows.

```
public void partialCodegen(Context context, CLEmitter partial)
{
    // Generate a method with an empty body; need a return to
    // make the class verifier happy.
    partial.addMethod(mods, name, descriptor, null, false);

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        partial.addNoArgInstruction(RETURN);
    }
    else if (returnType == Type.INT ||
             returnType == Type.BOOLEAN ||
             returnType == Type.CHAR) {
        partial.addNoArgInstruction(ICONST_0);
        partial.addNoArgInstruction(IRETURN);
    }
    else {
        // A reference type.
        partial.addNoArgInstruction(ACONST_NULL);
        partial.addNoArgInstruction(ARETURN);
    }
}
```

4.2.3.5 JFieldDeclaration.preAnalyze()

[3] Method descriptors are discussed in Appendix D.

Pre-analysis for a **`JFieldDeclaration`** is similar to that for a **`JMethodDeclaration`**. In a **`JFieldDeclaration`**, **`preAnalyze()`** does the following:

1. It enforces the rule that fields may not be declared **`abstract`**.
2. It resolves the field's declared type.
3. It generates the JVM code for the field declaration, via the **`CLEmitter`** created for the enclosing class declaration.

The code itself is rather simple:

```
public void preAnalyze(Context context, CLEmitter partial)
{
    // Fields may not be declared abstract.
    if (mods.contains("abstract")) {
        JAST.compilationUnit.reportSemanticError(line(),
            "Field cannot be declared abstract");
    }

    for (JVariableDeclarator decl: decls) {
        // Add field to (partial) class
        decl.setType(decl.type().resolve( context));
        partial.addField(mods, decl.name(),
            decl.type().toDescriptor(), false);
    }
}
```

4.2.3.6 The Symbol Table Built by preAnalyze()

So pre-analysis recursively descends only so far as the class members declared in a program and constructs only a **`CompilationUnitContext`** (in which all types are declared) and a **`ClassContext`**. No local variables are declared in pre-analysis. Figure 4.4 illustrates how much of the symbol table is constructed for our Factorial program once pre-analysis is complete.

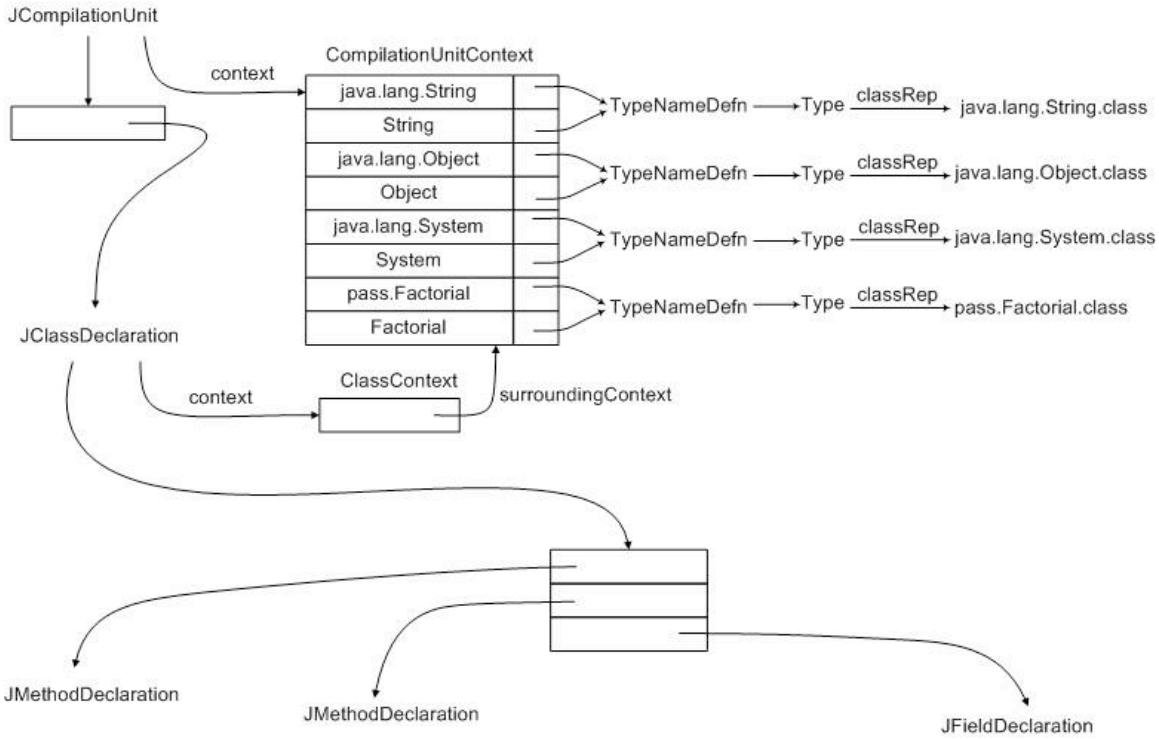

Figure 4.4 The Symbol Table Created by the Pre-Analysis Phase for the Factorial Program

4.2.4 Analysis of *j--* Programs

Once we have declared and loaded all imported types, and we have declared the types defined by class declarations and constructed `Class` objects for them, encapsulating information on fields and methods, the compiler can execute the analysis phase.

The analysis phase recursively descends throughout the AST all the way to its leaves,

- rewriting field and local variable initializations as assignments,
- declaring both formal parameters and local variables,
- allocating locations in the stack frame for the formal parameters and local variables,
- computing the types of expressions and enforcing the language type rules,
- reclassifying ambiguous names, and
- doing a limited amount of tree surgery.

4.2.4.1 The Top of the AST

4.2.4.1.1 Traversing the Top of the AST

At the top of the AST, **`analyze()`** simply recursively descends into each of the type (class) declarations, delegating analysis to one class declaration at a time:

```
public JAST analyze(Context context) {
    for (JAST typeDeclaration : typeDeclarations) {
        typeDeclaration.analyze(this.context);
    }
    return this;
}
```

Each class declaration in turn iterates through its members, delegating analysis to each of them. The only interesting thing is that, after all of the members have been analyzed, static field initializations are separated from instance field initializations, in preparation for code generation. This is discussed in the next section.

4.2.4.1.2 Rewriting Field Initializations as Assignments

In **`JFieldDeclaration`**, **`analyze()`** rewrites the field initializer as an explicit assignment statement, analyzes that and then stores it in the **`JFieldDeclaration`**'s initializations list.

```
public JFieldDeclaration analyze(Context context) {
    for (JVariableDeclarator decl : decls) {
        // All initializations must be turned into assignment
        // statements and analyzed
        if (decl.initializer() != null) {
            JAssignOp assignOp =
                new JAssignOp(decl.line(),
                              new JVariable(decl.line(),
                                            decl.name()),
                              decl.initializer());
            assignOp.isStatementExpression = true;
            initializations.add(
                new JStatementExpression(decl.line(),
                    assignOp).analyze(context));
        }
    }
    return this;
}
```

Afterwards, back up in **`JClassDeclaration`**, **`analyze()`** separates the assignment statements into two lists: one for the static fields and one for the instance fields.

```
// Copy declared fields for purposes of initialization.
for (JMember member : classBlock) {
    if (member instanceof JFieldDeclaration) {
        JFieldDeclaration fieldDecl = (JFieldDeclaration) member;
        if (fieldDecl.mods().contains("static")) {
            staticFieldInitializations.add(fieldDecl);
        } else {
            instanceFieldInitializations.add(fieldDecl);
        }
    }
}
```

Later, **`codegen()`** will use these lists in deciding whether or not to generate both an instance initializing method and a class initializing method.

For example, consider the static field, declared in the Factorial class:

```
static int n = 5;
```

Figure 4.5 illustrates how the sub-tree for the declaration is rewritten.

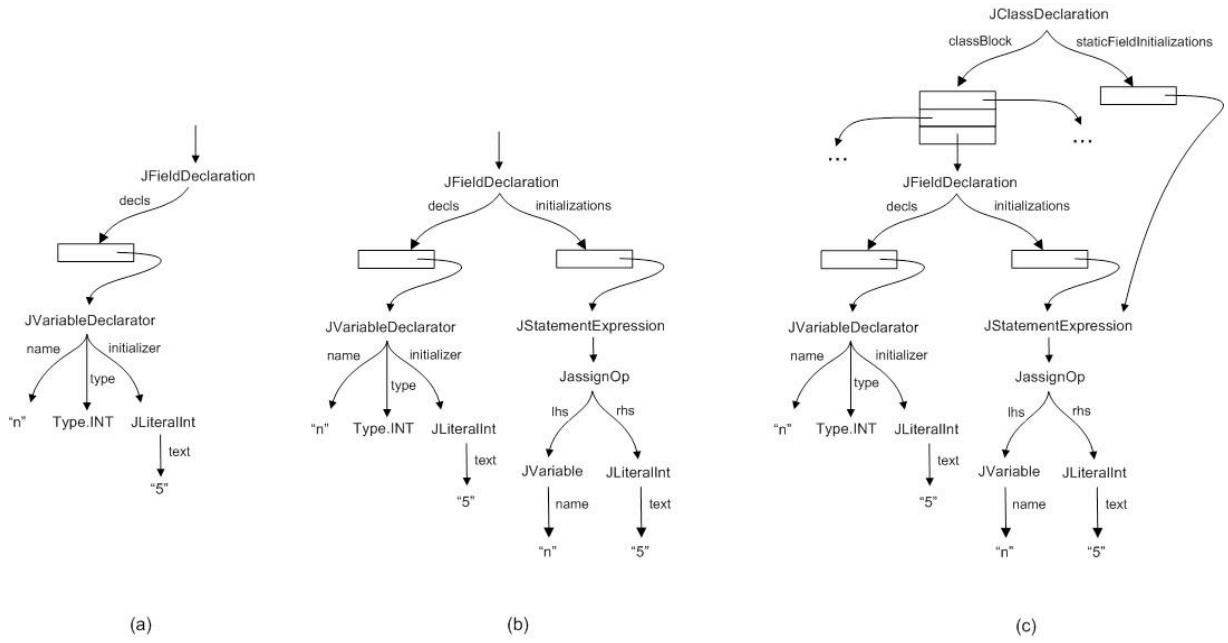

Figure 4.5 The Rewriting of a Field Initialization

The tree surgery in Figure 4.5 proceeds as follows.

1. The original sub-tree, produced by the parser for the static field declaration is shown in (a). The initial value, which might be any expression tree, is stored in **`JVariableDeclarator`**'s **`initializer`** field.
2. During analysis, **`JFieldDeclaration`**'s **`analyze()`** rewrites the initializing expression, 5 as an explicit assignment statement,

   ```
   n = 5;
   ```

   producing the sub-tree illustrated in (b).
3. Finally, when analysis returns back up the AST to **`JClassDeclaration`** (after recurring down the tree, these methods also back out and they may do tasks on the way back up), its **`analyze()`** copies the initializations to fields in its node. It separates the initializations into two lists: **`staticFieldInitializations`** for static fields, and **`instanceFieldInitializations`** for instance fields. In the case of our example, **`n`** is static, and so the initialization is added to **`staticFieldInitializations`** as illustrated in (c).

Besides this tree surgery on field initializations, **`analyze()`** does little at the top of the AST. The only other significant task is in **`JMethodDeclaration`**, where **`analyze()`** declares parameters and allocates locations relative to the base of the current stack frame. We discuss this in the next section.

4.2.4.2 Declaring Formal Parameters and Local Variables

4.2.4.2.1 MethodContexts and LocalContexts

Both formal parameters and local variables are declared in the symbol table and allocated locations within a method invocation's run-time stack frame. For example, consider the following class declaration.

```
public class Locals {
    public int foo(int t, String u) {
        int v = u.length();

        {
            int w = v + 5, x = w + 7;
            v = w + x;
        }
        {
            int y = 3;
            int z = v + y;
            t = t + y + z;
        }
        return t + v;
    }
```

```
}
```

The stack frame allocated for an invocation of **`foo()`** at run time by the JVM is illustrated in Figure 4.6. Because **`foo()`** is an instance method, space is allocated at location 0 for **`this`**.

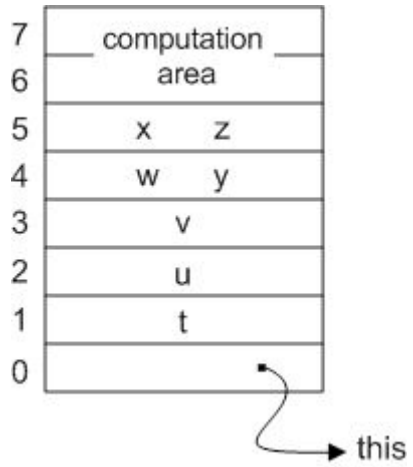

Figure 4.6 The Stack Frame for an Invocation of Locals.foo()

There are several regions of scope defined within the method **`foo()`**.

- Firstly, the method itself defines a region of scope, where the formal parameters **`t`** and **`u`** are declared.
- The method body defines a nested region of scope, where the local variable **`v`** is declared.
- Nested within the body are two blocks, each of which defines a region of scope:
    - In the first region, the local variables **`w`** and **`x`** are declared.
    - In the second region, the local variables **`y`** and **`z`** are declared.

  Because these two regions are disjoint, their locally declared variables may share locations on the run-time stack frame. Thus, as illustrated in Figure 4.6, **`w`** and **`y`** share the same location, and **`x`** and **`z`** share the same location.

During analysis, a context is created for each of these regions of scope: a **`MethodContext`** for that region defined by the method (and in which the formal parameters **`t`** and **`u`** are declared) and a **`LocalContext`** for each of the blocks.

The code for analyzing a **`JMethodDeclaration`** performs four steps.

1. It creates a new **`MethodContext`**, whose **`surroundingContext`** points back to the previous **`ClassContext`**.
2. The first stack frame offset is 0; but if this is an instance method then offset 0 must be allocated to **`this`**, and the **`nextOffset`** is incremented to 1.
3. The formal parameters are declared as local variables and allocated consecutive offsets in the stack frame.
4. It analyzes the method's body.

The code is straightforward:

```
public JAST analyze(Context context) {
    this.context = new MethodContext(context, returnType);

    if (!isStatic) {
        // Offset 0 is used to addr "this".
        this.context.nextOffset();
    }

    // Declare the parameters
    for (JFormalParameter param : params) {
        this.context.addEntry(param.line(), param.name(),
                new LocalVariableDefn(param.type(), this.context
                        .nextOffset(), null));
    }

    if (body != null) {
        body = body.analyze(this.context);
    }
    return this;
}
```

The code for analyzing a **`JBlock`** is even simpler; performs just two steps.

1. It creates a new **`LocalContext`**, whose **`surroundingContext`** points back to the previous **`MethodContext`** (or **`LocalContext`** in the case of nested blocks). Its **`nextOffset`** value is copied from the previous context.
2. It analyzes each of the body's statements. Any **`JVariableDeclarations`** declare their variables in the **`LocalContext`** created in step 1. Any nested **`JBlock`** simply invokes this two-step process recursively, creating yet another **`LocalContext`** for the nested block.

Again, the code is straightforward:

```
public JBlock analyze(Context context) {
    // { ... } defines a new level of scope.
    this.context = new LocalContext(context);

    for (int i = 0; i < statements.size(); i++) {
        statements.set(i, (JStatement) statements.get(i).analyze(
                this.context));
    }
    return this;
}
```

For example, the steps in adding the contexts to the symbol table for the analysis of method **foo()** are illustrated in Figure 4.7. In the contexts, the names should map to objects of type **LocalVariableDefn**, which define the variables and their stack frame offsets. In the figures, the arrows point simply to the offsets in parentheses.

Analysis proceeds as follows:

a. The **analyze()** method for **JMethodDeclaration** creates a new **MethodContext**. Because **foo()** is an instance method, location 0 is allocated to **this**, and the next available stack frame location (**nextOffset**) is 1.
b. It then declares the first formal parameter t in this **MethodContext**, allocating it the offset 1 (and incrementing **nextOffset** to 2).
c. It then declares the second formal parameter **u** in this **MethodContext**, allocating it the offset 2 (and incrementing **nextOffset** to 3). Analysis is then delegated to the method's block (a **JBlock**).
d. The **analyze()** method for **JBlock** creates a new **LocalContext**. Notice how the constructor for the new **LocalContext** copies the value (3) for **nextOffset** from the context for the enclosing method or block:

```
public LocalContext(Context surrounding) {
    super(surrounding, surrounding.classContext(),
          surrounding.compilationUnitContext());
   offset = (surrounding instanceof LocalContext)
       ? ((LocalContext) surrounding).offset()
       : 0;
 }
```

e. A **JVariableDeclaration** declares the variable **v** in the **LocalContext**, allocating it the offset 3.
f. **Analyze()** creates a new **LocalContext** for the nested **JBlock**, copying the **nextOffset** 4 from the context for the surrounding block.
g. A **JVariableDeclaration** declares the variable **w** in the new **LocalContext**, for the nested block, allocating it the offset 4.
h. A second **JVariableDeclaration** declares the variable **x** in the same **LocalContext**, allocating it the offset 5. The subsequent assignment statement will be analyzed in this context.
i. When this first nested **JBlock** has been analyzed, **analyze()** returns control to the **analyze()** for the containing **JBlock**; leaving the symbol table in exactly the state that it was in step (e).
j. **Analyze()** creates a new **LocalContext** for the *second* nested **JBlock**, copying the **nextOffset** 4 from the context for the surrounding block. Notice the similarity to the state in step (f). In this way, variables **x** and **z** will be allocated the same offsets in steps (k) and (l) as **w** and **y** were in steps (g) and (h).
k. A **JVariableDeclaration** declares the variable **y** in the new **LocalContext**, for the nested block, allocating it the offset 4.

l. A second **`JVariableDeclaration`** declares the variable **`z`** in the same **`LocalContext`**, allocating it the offset 5. The subsequent assignment statement will be analyzed in this context.
m. When this second nested **`JBlock`** has been analyzed, **`analyze()`** returns control to the **`analyze()`** for the containing **`JBlock`**; leaving the symbol table in exactly the state that it was in steps (e) and (i).
n. When the method body's **`JBlock`** has been analyzed, **`analyze()`** returns control to the **`analyze()`** for the containing **`JMethodDeclaration`**; leaving the symbol table in exactly the state that it was in step (c).
o. When the **`JMethodDeclaration`** has been analyzed, **`analyze()`** returns control to the **`analyze()`** for the containing **`JClassDeclaration`**; leaving the symbol table in exactly the state that it was before step (a).

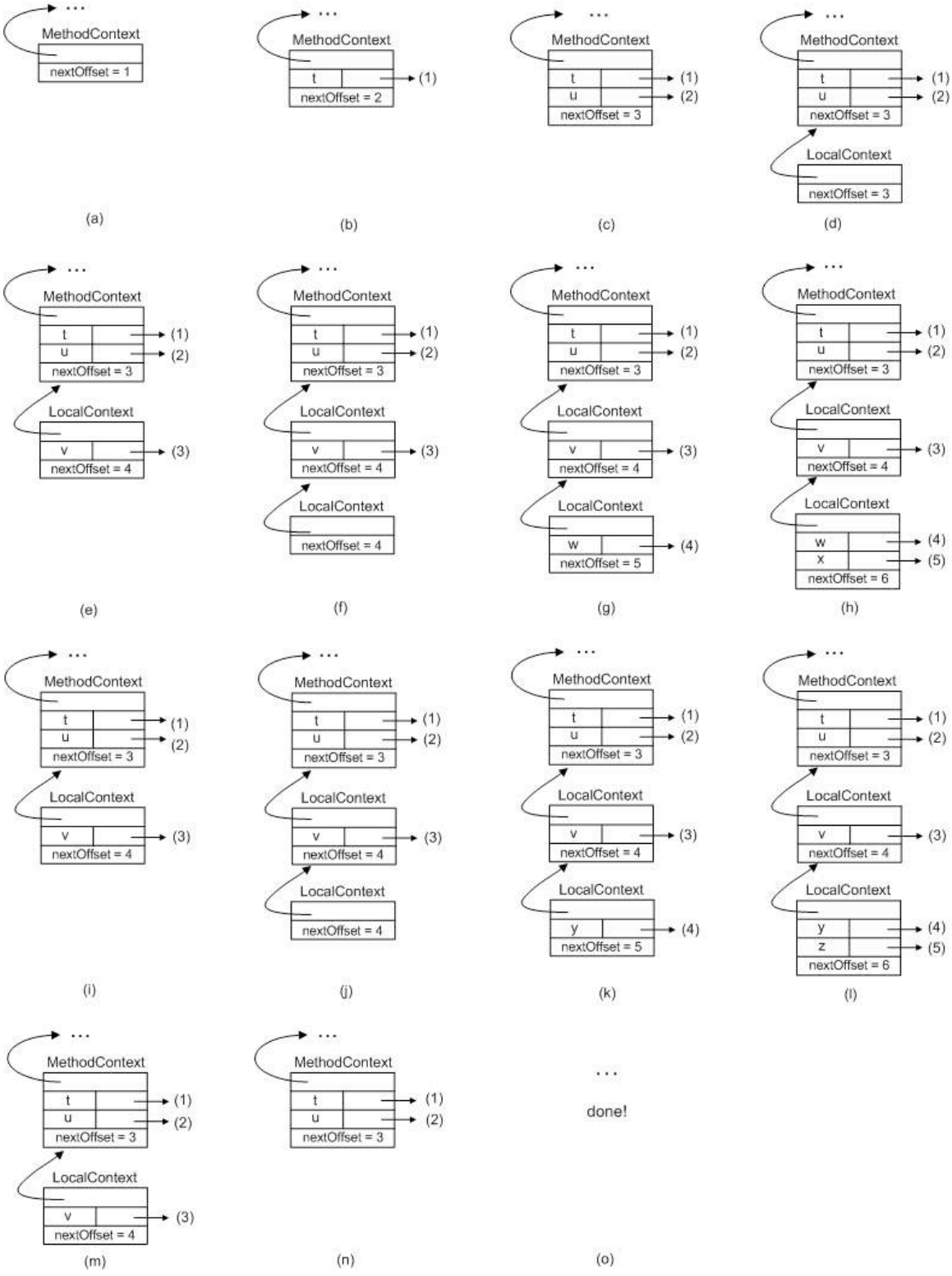

Figure 4.7 The Stages of the Symbol Table in Analyzing Locals.foo()

We address the details of analyzing local variable declarations in the next section.

4.2.4.2.2 Analyzing Local Variable Declarations and their Initializations

A local variable declaration is represented in the AST with a **`JVariableDeclaration`**. For example, consider the local variable declaration from Locals.

```
int w = v + 5, x = w + 7;
```

Before the **`JVariableDeclaration`** is analyzed, it appears exactly as it was created by the parser, as is illustrated in Figure 4.8.

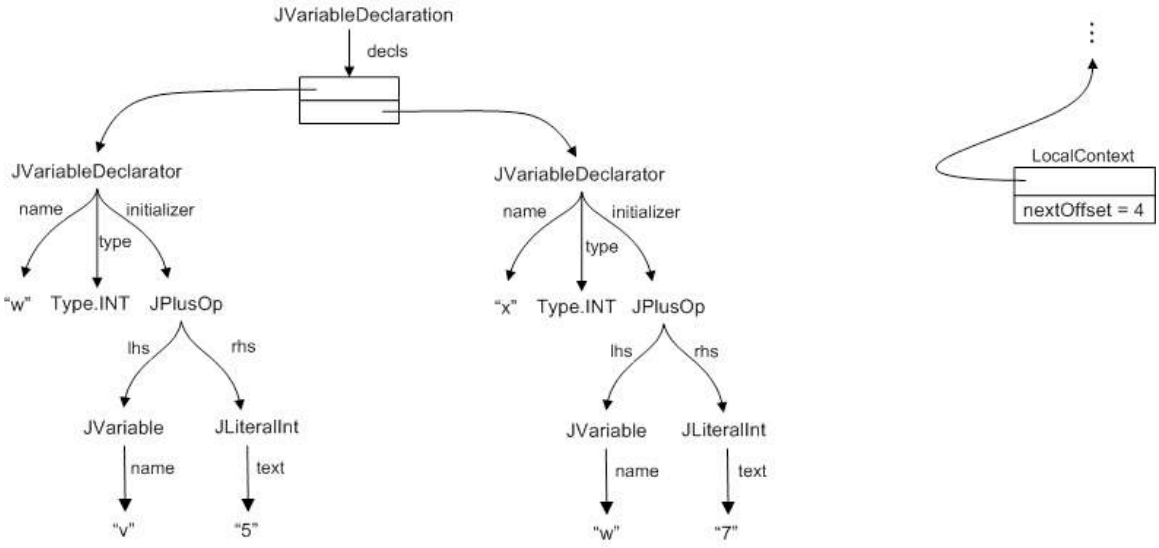

Figure 4.8 The Sub-tree for `int w = v + 5, x = w + 7;` Before Analysis

Figure 4.8 also pictures the **`LocalContext`** created for the nested block in which the declaration occurs, but before the declaration is analyzed.

The code for analyzing a **`JVariableDeclaration`** is as follows.

```
public JStatement analyze(Context context) {
    for (JVariableDeclarator decl : decls) {
        // Local variables are declared here (fields are
        // declaredin preAnalyze())
        int offset = ((LocalContext) context).nextOffset();
        LocalVariableDefn defn = new LocalVariableDefn(decl
            .type().resolve(context), offset);
```

```
        // First, check for shadowing
        IDefn previousDefn = context.lookup(decl.name());
        if (previousDefn != null
            && previousDefn instanceof LocalVariableDefn) {
            JAST.compilationUnit.reportSemanticError(decl.line(),
                "The name " + decl.name()
                    + " overshadows another local variable.");
        }

        // Then declare it in the local context
        context.addEntry(decl.line(), decl.name(), defn);

        // All initializations must be turned into assignment
        // statements and analyzed
        if (decl.initializer() != null) {
            defn.initialize();
            JAssignOp assignOp = new JAssignOp(decl.line(),
                new JVariable(decl.line(), decl.name()), decl
                    .initializer());
            assignOp.isStatementExpression = true;
            initializations.add(new JStatementExpression(decl
                .line(), assignOp).analyze(context));
        }
    }
    return this;
}
```

Analysis of a **`JVariableDeclaration`** such as that in Figure 4.8 involves the following.

1. **`LocalVariableDefn`**s and their corresponding stack frame offsets are allocated for each of the declared variables.
2. The code checks to make sure that the declared variables do not shadow existing local variables.
3. The variables are declared in the local context.
4. Any initializations are rewritten as explicit assignment statements; those assignments are re-analyzed and stored in an **`initializations`** list. Later, code generation will generate code for any assignment assignments in this list.

Figure 4.9 illustrates the result of analyzing the **`JVariableDeclaration`** in Figure 4.8. Notice that re-analyzing the assignment statements attaches types to each node in the sub-trees; more on that below.

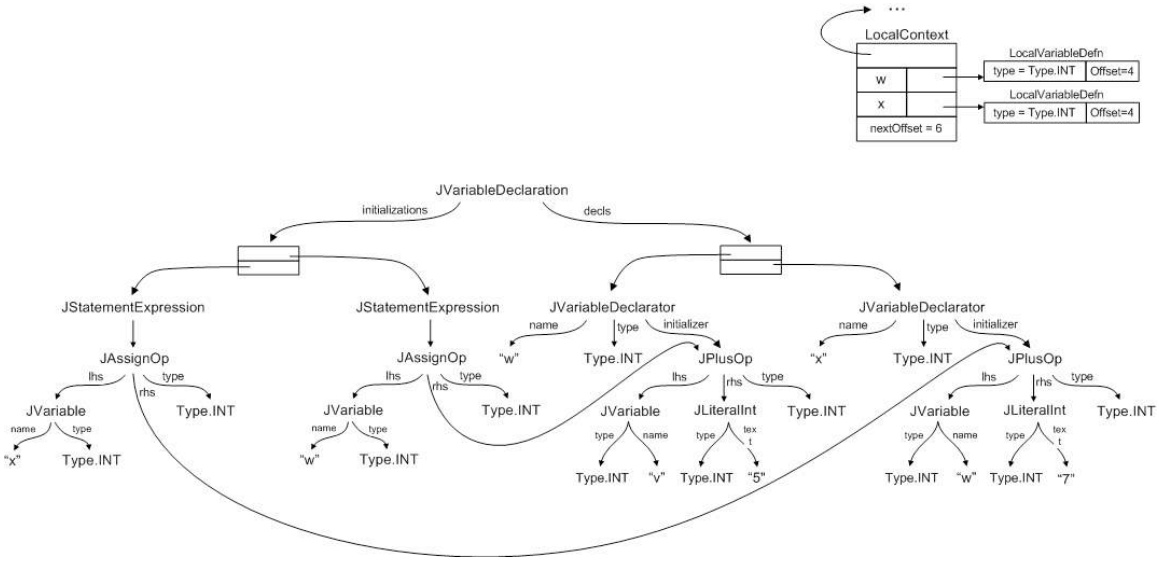

Figure 4.9 The Sub-tree for `int w = v + 5, x = w + 7;` After Analysis

4.2.4.3 Simple Variables

Simple variables are represented in the AST as **`JVariable`** nodes. A simple variable could denote a local variable, a field or a type. Analysis of simple variables involves looking their names up in the symbol table to find their types. If a variable is not found in the symbol table then we examine the **`Type`** for the surrounding class (in which the variable appears) to see if it is a field. If it is a field, then the field selection is made explicit by rewriting the tree as a **`JFieldSelection`**.

The code for **`analyze()`** in **`JVariable`** is as follows.

```
public JExpression analyze(Context context) {
    iDefn = context.lookup(name);
    if (iDefn == null) {
        // Not a local, but is it a field?
        Type definingType = context.definingType();
        Field field = definingType.fieldFor(name);
        if (field == null) {
            type = Type.ANY;
            JAST.compilationUnit.reportSemanticError(line,
                "Cannot find name: " + name);
        } else {
            // Rewrite a variable denoting a field as an
            // explicit field selection
            type = field.type();
```

```
            JExpression newTree = new JFieldSelection(line(),
                field.isStatic() ||
                (context.methodContext() != null &&
                 context.methodContext().isStatic()) ?
                        new JVariable(line(),
                            definingType.toString()) :
                            new JThis(line), name);
            return (JExpression) newTree.analyze(context);
        }
    } else {
        if (!analyzeLhs && iDefn instanceof LocalVariableDefn &&
            !((LocalVariableDefn) iDefn).isInitialized()) {
            JAST.compilationUnit.reportSemanticError(line,
                "Variable " + name + " might not have been
                    initialized");
        }
        type = iDefn.type();
    }
    return this;
}
```

For example, consider a simple case where a variable is declared locally, such as the variable **`v`** in our **`Locals`** class. When analyzing the return statement,

```
return t + v;
```

the analysis of **`v`** is pretty straightforward. And is illustrated in Figure 4.10.

1. Its name is looked up in the symbol table and is found to be associated with the **`LocalVariableDefn`** of a local variable with type **`Type.INT`** and offset 3.
2. The **`LocalVariableDefn`** is recorded in field iDefn for later use by code generation.
3. The type field is copied from the **`LocalVariableDefn`**.

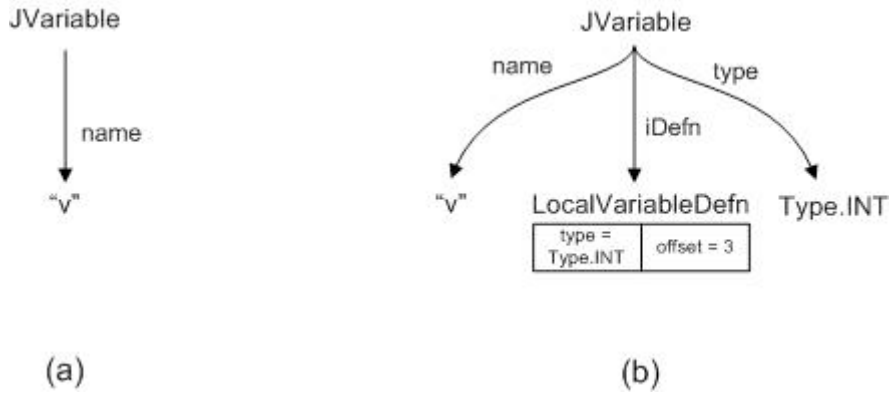

Figure 4.10 The Analysis of a Locally Declared Variable

(a) Before Analysis; (b) After Analysis

When the variable denotes a field, analysis is a little more interesting. For example, consider the analysis of the static field **`n`**, when it appears in the **`main()`** method of our **`Factorial`** class example above.

1. Figure 4.11 (a) shows the **`JVariable`** before it is analyzed.
2. Its name is looked up in the symbol table but is not found. So the defining type (the type declaration enclosing the region where the variable appears) is consulted; **`n`** is found to be a static field in class **`Factorial`**.
3. The implicit static field selection is made explicit by rewriting the tree to represent

   ```
   pass.Factorial.n
   ```

   This produces the **`JFieldSelection`** illustrated in Figure 4.11 (b).
4. The **`JFieldSelection`** produced in step 3 is recursively analyzed to determine the types of its target and the result, as illustrated in Figure 4.11 (c).
5. It is this sub-tree that is returned to replace the original **`JVariable`** (a) in the parent AST. This tree re-writing is the reason that **`analyze()`** everywhere returns a (possibly re-written) sub-tree.

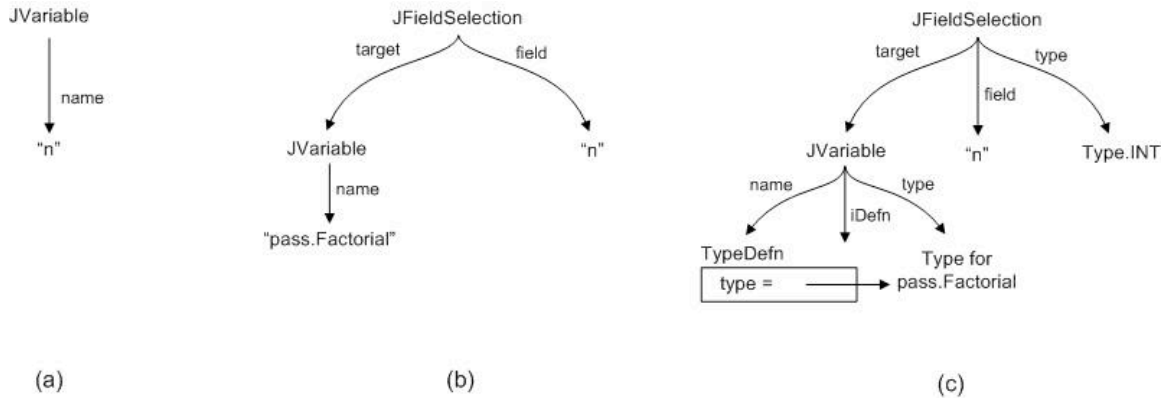

Figure 4.11 The Analysis of a Variable That Denotes a Static Field

Just how the sub-tree in Figure 11 (b) is analyzed to produce (c) is discussed in the next section.

4.2.4.4 Field Selection and Message Expressions

4.2.4.4.1 Reclassifying an Ambiguous Target

Both field selections and message expressions have *targets*. In a field selection, the target is either an object or a class from which one wants to select a field. In a message expression, the target is an object or class to which one is sending a message. Unfortunately, the parser cannot always make out the syntactic structure of a target.

For example, consider the field selection

`w.x.y.z`

The parser knows this is a field selection of some sort and that **`z`** is the field. But, without knowing the types of w, x and y, the parser cannot know whether

- **`w`** is a class name, **`x`** is a static field in **`w`**, and **`y`** is a field of **`x`**;
- **`w`** is a package containing class **`x`**, and **`y`** is a static field in **`x`**; or
- **`w.x.y`** is a fully qualified class name like **`java.lang.System`**.

For this reason, the parser packages up the string "**`w.x.y`**" in an **`AmbiguousName`** object, attached to either the **`JFieldSelection`** or **`JMessageExpression`**, so as to put off the decision until analysis.

The first thing that analysis does in either sort of expression is to reclassify the ambiguous target, in the context in which it appears. The **`reclassify()`** method in **`AmbiguousName`** is based on the rules in the Java Language Specification for reclassifying an ambiguous name:

```
public JExpression reclassify(Context context) {
    // Easier because we require all types to be imported.
    JExpression result = null;
    StringTokenizer st = new StringTokenizer(name, ".");

    // Firstly, find a variable or Type.
    String newName = st.nextToken();
    IDefn iDefn = null;

    do {
        iDefn = context.lookup(newName);
        if (iDefn != null) {
            result = new JVariable(line, newName);
            break;
        } else if (!st.hasMoreTokens()) {
            // Nothing found. :(
            JAST.compilationUnit.reportSemanticError(line,
                "Cannot find name " + newName);
            return null;
        } else {
            newName += "." + st.nextToken();
        }
    } while (true);

    // For now we can assume everything else is fields.
    while (st.hasMoreTokens()) {
        result = new JFieldSelection(line, result,
                        st.nextToken());
    }
```

```
    return result;
}
```

For example, consider the message expression,

```
java.lang.System.out.println(…);
```

The parser will have encapsulated the target **`java.lang.System.out`** in an **`AmbiguousName`** object. The first thing **`analyze()`** does for a **`JMessageExpression`** is to reclassify this **`AmbiguousName`** to determine the structure of the expression that it denotes. It does this by looking at the ambiguous **`java.lang.System.out`** from left to right.

1. Firstly, **`reclassify()`** looks up the simple name, **`java`** in the symbol table.
2. Not finding that, it looks up **`java.lang`**.
3. Not finding that, it looks up **`java.lang.System`**, which (assuming **`java.lang.System`** has been properly imported) it finds to be a class.
4. It then assumes that the rest of the ambiguous part, that is **`out`**, is a field.
5. Thus the target is a field selection whose target is **`java.lang.System`** and whose field name is **`out`**.

4.2.4.4.2 Analyzing a Field Selection

```
public JExpression analyze(Context context) {
    // Reclassify the ambiguous part.
    …
    target = (JExpression) target.analyze(context);
    Type targetType = target.type();

    // We use a workaround for the "length" field of arrays.
    if ((targetType instanceof ArrayTypeName)
        && fieldName.equals("length")) {
        type = Type.INT;
    } else {
        // Other than that, targetType has to be a
        // ReferenceType
        if (targetType.isPrimitive()) {
            JAST.compilationUnit.reportSemanticError(line(),
                "Target of a field selection must "
                    + "be a defined type");
            type = Type.ANY;
            return this;
        }
        field = targetType.fieldFor(fieldName);
        if (field == null) {
```

```
            JAST.compilationUnit.reportSemanticError(line(),
                "Cannot find a field: " + fieldName);
            type = Type.ANY;
        } else {
            context.definingType().checkAccess(line,
                (Member) field);
            type = field.type();

            // Non-static field cannot be referenced from a
            // static context.
            if (!field.isStatic()) {
                if (target instanceof JVariable &&
                    ((JVariable) target).iDefn() instanceof
                       TypeNameDefn) {
                     JAST.compilationUnit.
                         reportSemanticError(line(),
                         "Non-static field " + fieldName +
                            " cannot be referenced from a static
                              context");
                }
            }
        }
    }
    return this;
}
```

After reclassifying any ambiguous part and making that the target, analysis of a **`JFieldSelection`** proceeds as follows:

1. It analyzes the target and determines the target's type.
2. It then considers the special case where the target is an array and the field is **`length`**. In this case, the type of the "field selection" is **`Type.INT`**.[4]
3. Otherwise, it ensures that the target is not a primitive and determines whether or not it can find a field of the appropriate name in the target's type. If it cannot, then an error is reported.
4. Otherwise, it checks to make sure the field is accessible to this region, a non-static field is not referenced from a static context, and then returns the analyzed field selection sub-tree.

Analyzing messages expressions is similar, but with the added complication of arguments.

4.2.4.4.3 Analyzing a Message Expression

[4] This is a hack. But so is the (both *j*-- and Java) language in this instance; this isn't really a field selection but an operation on arrays.

After reclassifying any **`AmbiguousName`**, analyzing a **`JMessageExpression`** proceeds as follows.

1. It analyzes the arguments to the message and constructs an array of their types.
2. It determines the surrounding, defining class (for determining access).
3. It analyzes the target to which the message is being sent.
4. It takes the message name and the array of argument types and looks for a matching method defined in the target's type. In *j--,* argument types must match exactly. If no such method is found, it reports an error.
5. Otherwise, the target class and method are checked for accessibility, a non-static method is now allowed to be referenced from a static context, and the method's return type becomes the type of the message expression.

```
public JExpression analyze(Context context) {
    // Reclassify the ambiguous part
    …

    // Then analyze the arguments, collecting
    // their types (in Class form) as argTypes
    argTypes = new Type[arguments.size()];
    for (int i = 0; i < arguments.size(); i++) {
        arguments.set(i, (JExpression) arguments.get(i).analyze(
            context));
        argTypes[i] = arguments.get(i).type();
    }

    // Where are we now? (For access)
    Type thisType = ((JTypeDecl) context.classContext
        .definition()).thisType();

    // Then analyze the target
    if (target == null) {
        // Implied this (or, implied type for statics)
        if (!context.methodContext().isStatic()) {
            target = new JThis(line()).analyze(context);
        }
        else {
            target = new JVariable(line(),
                        context.definingType().toString()).
                            analyze(context);
        }
    } else {
        target = (JExpression) target.analyze(context);
        if (target.type().isPrimitive()) {
            JAST.compilationUnit.reportSemanticError(line(),
                "cannot invoke a message on a primitive type:"
```

```
                    + target.type());
        }
    }

    // Find appropriate Method for this message expression
    method = target.type().methodFor(messageName, argTypes);
    if (method == null) {
        JAST.compilationUnit.reportSemanticError(line(),
            "Cannot find method for: "
                + Type.signatureFor(messageName, argTypes));
        type = Type.ANY;
    } else {
        context.definingType().checkAccess(line,
                                           (Member) method);
        type = method.returnType();

        // Non-static method cannot be referenced from a
        // static context.
        if (!method.isStatic()) {
            if (target instanceof JVariable &&
                ((JVariable) target).iDefn() instanceof
                    TypeNameDefn) {
                JAST.compilationUnit.reportSemanticError(line(),
                 "Non-static method " +
                 Type.signatureFor(messageName, argTypes) +
                 "cannot be referenced from a static context");
            }
        }
    }
    return this;
}
```

4.2.4.5 Typing Expressions and Enforcing the Type Rules

Much of the rest of analysis, as defined for the various kinds of AST nodes is about computing and checking types and enforcing additional *j--* rules. Indeed when one reads through any compiler, one finds lots of code whose only purpose is to enforce a litany of rules.

For most kinds of AST nodes, analysis involves
1. analyzing the sub-trees, and
2. checking the types.

4.2.4.5.1 Analyzing a Subtraction Operation

For example, analyzing a **`JSubtractOp`** is straightforward:

```
public JExpression analyze(Context context) {
     lhs = (JExpression) lhs.analyze(context);
     rhs = (JExpression) rhs.analyze(context);
     lhs.type().mustMatchExpected(line(), Type.INT);
     rhs.type().mustMatchExpected(line(), Type.INT);
     type = Type.INT;
     return this;
 }
```

4.2.4.5.2 Analyzing a + Operation Causes Tree Rewriting for Strings

On the other hand, analyzing a **`JPlusOp`** is complicated by the possibility that one of the operands to the **`+`** is a string; in this case, we simply rewrite the **`JPlusOp`** as a **`JStringConcatenationOp`** and analyze *that*.

```
public JExpression analyze(Context context) {
    lhs = (JExpression) lhs.analyze(context);
    rhs = (JExpression) rhs.analyze(context);
    if (lhs.type() == Type.STRING || rhs.type() == Type.STRING) {
        return (new JStringConcatenationOp(line, lhs, rhs))
            .analyze(context);
    } else if (lhs.type() == Type.INT && rhs.type() == Type.INT){
        type = Type.INT;
    } else {
        type = Type.ANY;
        JAST.compilationUnit.reportSemanticError(line(),
            "Invalid operand types for +");
    }
    return this;
}
```

And, analyzing a **`JStringConcatenation`** is easy because we *know* at least one of the operands is a string, so the result has to be a string.

```
public JExpression analyze(Context context) {
    type = Type.STRING;
    return this;
}
```

4.2.4.5.3 Analyzing a Literal

Analyzing a literal is trivial. We know its type. For example, the **`analyze()`** method for **`JLiteralInt`** follows.

```
public JExpression analyze(Context context) {
```

```
    type = Type.INT;
    return this;
}
```

4.2.4.5.4 Analyzing a Control (e.g. if) Statement

Finally, analyzing a control statement isn't much more difficult. For example, analysis of the if statement involves only

1. analyzing the test and checking that it is a Boolean;
2. analyzing the consequent (the **`then`** part); and finally
3. if there is an alternate (an **`else`** part), analyzing that.

The code for analyzing the **`JIfStatement`** follows.

```
public JStatement analyze(Context context) {
    test = (JExpression) test.analyze(context);
    test.type().mustMatchExpected(line(), Type.BOOLEAN);
    consequent = (JStatement) consequent.analyze(context);
    if (alternate != null) {
        alternate = (JStatement) alternate.analyze(context);
    }
    return this;
}
```

4.2.4.6 Analyzing Cast Operations

The *j--* language is stricter than is Java when it comes to types. For example, there are no implied conversions in *j--*. So, when one assigns an expression to a variable, the types must match exactly. The same goes for actual parameters to messages matching the formal parameters of methods.

This doesn't exclude polymorphism. For example if type **`Foo1`** extends (that is, is a sub-type of) type **`Foo`**, if **`foo1`** is a variable of type **`Foo1`** and **`foo`** is a variable of type **`Foo`**, we can say

```
foo = (Foo) foo1;
```

to keep the *j--* compiler happy. Of course, the object that **`foo1`** refers to could be of type **`Foo1`** or any of *its* sub-types. Polymorphism hasn't gone away.

Analysis, when encountering a **`JCastOp`** for an expression such as

(<Type2>) <expression of Type1>

must determine two things:

1. That an expression of type Type1 can be cast to Type2, that is that the cast is valid.
2. The type of the result. This part is easy: it is simply Type2.

To determine (1) we must consider the possibilities for Type1 and Type2. These are specified in section 4.5 of the *Java Language Specification*.

1. Any type may be cast to itself. Think of this as the *Identity* cast.
2. An arbitrary reference type may be cast to another reference type if and only if either,
    a. The first type is a sub-type of (extends) the second type. This is called *widening* and requires no action at run time.
    b. The second type is a sub-type of the first type. This is called *narrowing* and requires a run-time check to make sure the expression being cast is actually an instance of the type it is being cast to.

The following table summarizes other casts. In reading the table, think of the rows as Type1 and the columns as Type2. So the table says whether or not (and how) a type labeling a row may be cast to a type labeling a column.

	boolean	**char**	**int**	**Boolean**	**Character**	**Integer**
boolean	Identity	Error	Error	Boxing	Error	Error
char	Error	Identity	Widening	Error	Boxing	Error
int	Error	Narrowing	Identity	Error	Error	Boxing
Boolean	Unboxing	Error	Error	Identity	Error	Error
Character	Error	Unboxing	Error	Error	Identity	Error
Integer	Error	Error	Unboxing	Error	Error	Identity

3. A **`boolean`** can always be cast to an **`Boolean`**. The **`Boolean`** simply encapsulates the **`boolean`** value so the operation is called *boxing*. Similarly, a **`char`** can be boxed as a **`Character`**, and an **`int`** can be boxed as an **`Integer`**.
4. A **`Boolean`** can be cast to a **`boolean`** using *unboxing*, which involves simply plucking out the **`boolean`** value that is encapsulated by the **`Boolean`**. Similarly for **`Characters`** and **`Integers`**.
5. One primitive type may often be cast to another.
    a. For example, a **`char`** may be cast to an **`int`**. This is an example of widening and requires no run-time action.
    b. An **`int`** may be cast to a **`char`**. This is an example of narrowing and requires the execution of the **`i2c`** operation at run time.
6. Some types may not be cast to other types. For example, on may not cast a **`boolean`** to an **`int`**. Such casts are invalid.

The code for **`analyze()`** in **`JCastOp`** follows.

```
public JExpression analyze(Context context) {
    expr = (JExpression) expr.analyze(context);
    type = cast = cast.resolve(context);
    if (cast.equals(expr.type())) {
        converter = Converter.Identity;
    } else if (cast.isJavaAssignableFrom(expr.type())) {
        converter = Converter.WidenReference;
    } else if (expr.type().isJavaAssignableFrom(cast)) {
        converter = new NarrowReference(cast);
    } else if ((converter =
        conversions.get(expr.type(), cast)) != null) {
    } else {
        JAST.compilationUnit.reportSemanticError(line,
            "Cannot cast a " + expr.type().toString() + " to a "
                + cast.toString());
    }
    return this;
}
```

In the code, **`cast`** is the type that we want to cast the expression **`expr`** to. The code not only decides whether or not the cast is valid, but if it is valid, computes the **`Converter`** that will be used at code generation time to generate any run time code required for the cast. If no such converter is defined then the cast is *invalid*.

One example of a converter is that for narrowing one reference type to another (more specific) reference sub-type; the converter is **`NarrowReference`** and its code is as follows.

```
class NarrowReference implements Converter {

    private Type target;

    public NarrowReference(Type target) {
        this.target = target;
    }

    public void codegen(CLEmitter output) {
        output.addReferenceInstruction(CHECKCAST,
                                       target.jvmName());
    }
}
```

The compiler creates a **`NarrowReference`** instance for each cast of this kind, tailored to the type it is casting to.

The **`codegen()`** method is used in the next code generation phase. We address code generation in the next chapter.

4.2.5 The Visitor Pattern and the AST Traversal Mechanism

One might ask, "Why does the compiler not use the *visitor pattern* for traversing the AST in pre-analysis, analysis and code generation?

The visitor pattern is one of the design patterns introduced by Erich Gamma, et al (Gamma, 1995). The visitor pattern serves two purposes:

1. It separates the tree traversal function from the action taken at each node. A separate mechanism, which can be shared by tree-printing, pre-analysis, analysis and code generation, traverses the tree.
2. It gathers all of the actions for each phase together into one module (or method). Thus, all of the tree-printing code is together, all of the pre-analysis code is together, all of the analysis code is together, and all of the code generation code is together.

The idea is that it separates traversal from actions taken at the nodes (a separation of concerns) and it makes it simpler to add new actions. For example, to add an optimization phase one need simply write a single method, rather than having to write an **`optimize()`** method for each node type.

Such an organization would be useful if we were planning to add new actions. But we are more likely to be adding new syntax to *j--*, together with its functionality. Making extensions to *j--* would require our modifying each of the phases.

Moreover, often the traversal order for one node type differs from that of another. And sometimes we want to perform actions at a node before, after and even in-between the traversal of the sub-trees. For example, take a look at **`codegen()`** for the **`JIfStatement`**.

```
public void codegen(CLEmitter output) {
    String elseLabel = output.createLabel();
    String endLabel = output.createLabel();
    condition.codegen(output, elseLabel, false);
    thenPart.codegen(output);
    if (elsePart != null) {
        output.addBranchInstruction(GOTO, endLabel);
    }
    output.addLabel(elseLabel);
    if (elsePart != null) {
        elsePart.codegen(output);
        output.addLabel(endLabel);
    }
```

}

Mixing the traversal code along with the actions is just so much more flexible. For this reason, the *j--* compiler eschews the visitor pattern.

4.3 Attribute Grammars

Context-free grammars, as powerful as they are, cannot express context-sensitive conditions, such as ensuring the same value for *n* in a string $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$. We can employ context-sensitive grammars for such problems. However, these grammars are impractical for specifying context-sensitive aspects of a programming language. In this section, we will explore techniques for augmenting a context-free grammar in order to express context-sensitive conditions. These augmented grammars are called attribute grammars.

Attribute grammars, first developed by Donald Knuth in 1968 as a means of formalizing the semantics of a context-free language, are useful in specifying the syntax and semantics of a programming language. An attribute grammar can be used to specify the context-sensitive aspects of a language, such as checking that a variable has been declared before use and that the use of the variable is consistent with its declaration. Attribute grammars can also be used to specify the operational semantics of a language by defining a translation into machine-specific lower-level code.

In this section, we will first introduce attribute grammars through examples, then provide a formal definition for an attribute grammar, and finally present attribute grammars for a couple of constructs in *j--*.

4.3.1 Examples

An attribute grammar may be informally defined as a context-free grammar that has been extended to provide context-sensitivity using a set of attributes, assignment of attribute values, evaluation rules, and conditions. In a parse tree representing the input sentence (source program), attribute grammars can pass values from a node to its parent, using a *synthesized* attribute, or from a node to its child, using an *inherited* attribute. In addition to passing values up or down the tree, the attribute values may be assigned, modified, and checked at any node in the tree. We clarify these ideas using the following examples.

Let us try and write a grammar to recognize sentences of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$. The sentences **aabbcc** and **abc** belong to this grammar, but the sentences **abbcc** and **aac** do not. Consider the following context-free grammar (from Slonneger, 1994) that describes this language.

letterSequence ::= aSequence bSequence cSequence

aSequence ::= **a** | aSequence **a**

bSequence ::= **b** | aSequence **b**

cSequence ::= **c** | aSequence **c**

It is easily seen that the above grammar, besides generating the acceptable strings such as **aabbcc**, also generates the unacceptable strings such as **abbcc**. As we have seen in the previous chapter, it is impossible to write a context-sensitive grammar for the language under consideration here. Attribute grammars come to our rescue. Augmenting our context-free grammar with an attribute describing the length of a letter sequence, we can use these values to make sure that the sequences of **a**, **b**, and **c** have the same length.

We associate a synthesized attribute *size* with the nonterminals aSequence, bSequence, and cSequence, and a condition at the root of the parse tree that the *size* attribute for each of the letter sequences has the same value. If the input consists of a single character, *size* is set to 1; if it consists of a sequence followed by a single character, *size* for the parent character is set to the size of the child character plus one. Added to the root is the condition that the sizes of the sequences be the same. Here is the augmented grammar.

letterSequence ::= aSequence bSequence cSequence

condition : *size*(aSequence) = *size*(bSequence) = *size*(cSequence)

aSequence ::= **a**
size(aSequence) ← 1
| aSequence **a**
size(aSequence) ← *size*(aSequence) + 1

bSequence ::= **b**
size(bSequence) ← 1
| bSequence **b**
size(bSequence) ← *size*(bSequence) + 1

cSequence ::= **c**
size(cSequence) ← 1
| cSequence **c**
size(cSequence) ← *size*(cSequence) + 1

The augmented attribute grammar successfully parses the legal strings such as **aabbcc**, but does not parse the illegal sequences such as **abbcc**. Though such sequences satisfy the BNF part of the grammar, they dot satisfy the condition required of the attribute values.

Another approach is to pass information from one part of the parse tree to some node, and then have it inherited down into other parts of the tree. Let *size* be a synthesized attribute for the sequence of **a**'s, and *iSize* be the inherited attribute for the sequences of **b**'s and **c**'s. We synthesize the size of the sequence of **a**'s to the root of the parse tree, and set the *iSize* attribute for the **b** sequence and the **c** sequence to this value and inherit it down the tree, decrementing the value by one every time we see another character in the sequence. When we reach the node where the sequence has a child consisting of a single character, we check if the inherited *iSize* attribute equals one. If so, the size of the sequence must be the same as the size of the **a** sequence; otherwise, the two sizes do not match and the parsing is unsuccessful. The following attribute grammar clarifies this.

letterSequence ::= aSequence bSequence cSequence

iSize(bSequence) ← *size*(aSequence)
iSize(cSequence) ← *size*(aSequence)

aSequence ::= **a**
size(aSequence) ← 1
| aSequence **a**
size(aSequence) ← *size*(aSequence) + 1

bSequence ::= **b**
condition : *iSize*(bSequence) = 1
| bSequence **b**
iSize(bSequence) ← *iSize*(bSequence) - 1

cSequence ::= **c**
condition : *iSize*(cSequence) = 1
| cSequence **c**
iSize(cSequence) ← *iSize*(cSequence) - 1

For the nonterminal aSequence, *size* is a synthesized attribute, but for the nonterminals bSequence and cSequence, *iSize* is an inherited attribute passed from the parent to child. As before, the attribute grammar above cannot parse illegal sequences such as **abbcc**, since these sequences do not satisfy all the conditions associated with attribute values.

In this grammar the sequence of **a**'s determines the desired length against which the other sequences are checked. Consider the sequence **abbcc**. It could be argued that the sequence of **a**'s is at fault and not the other two sequences. However, in a programming language with declarations, we use the declarations to determine the desired types against which the remainder of the program is checked. In other words, the declaration information is synthesized up and is inherited by the entire program for checking.

In our next example, we illustrate the use of attribute grammars in specifying semantics using an example from Knuth's 1968 seminal paper on attribute grammars. The example grammar computes the values of binary numbers. The grammar uses both synthesized and inherited attributes.

Consider the following context-free grammar describing the structure of binary numbers. N stands for binary numbers, L stands for bit lists, and B stands for individual bits.

N ::= L

N ::= L **.** L

L ::= B

L ::= L B

B ::= **0**

B ::= **1**

Examples of binary numbers that follow the above grammar are both integral numbers such as 0, 1, 10, 11, 100, and so on, and rational numbers with integral fractional part such as 1.1, 1.01, 10.01, and so on.

We want to compute the values of the binary numbers using an attribute grammar. For example, 0 has the value 0, 1 the value 1, 0.1 the value 0.5, 10.01 the value 2.25, and so on.

Knuth provides a couple of different attribute grammars to define the computation; the first one uses synthesized attributes only, and the second uses both inherited and synthesized attributes. We will illustrate the latter.

In this attribution, each bit has a synthesized attribute *value* which is a rational number that takes the position of the bit into account. For example, the first bit in the binary number 10.01 will have the value 2 and the last bit will have the value 0.25. In order to define these attributes, each bit also has an inherited attribute scale used to compute the value of a 1-bit as value = 2^{scale}. So for the bits in 10.01, scale will be 1, 0, -1, -2, respectively.

If we have the values of the individual bits, these values can simply be summed up to the total value. This is done by adding synthesized attributes *value* both for bit lists and for binary numbers. In order to compute the scale attribute for the individual bits, an inherited attribute *scale* is added also for bit lists (representing the scale of the rightmost

bit in that list), and a synthesized attribute *length* for bit lists, holding the length of the list.

Knuth uses the abbreviations v, s, and l for the value, scale, and length attributes. Here is the resulting Knuth's attribute grammar.

N ::= L
    v(N) ← v(L)
    s(L) ← 0

N ::= L_1 **.** L_2
    v(N) ← v(L_2) + v(L_2)
    s(L_1) ← 0
    s(L_2) ← - l(L_2)

L ::= B
    v(L) ← v(B)
    s(B) ← s(L)
    l(L) ← 1

L_1 ::= L_2 B
    v(L_1) ← v(L_2) + v(B)
    s(B) ← s(L1)
    s(L_2) ← s(L_1) + 1
    l(L_1) ← l(L_2) + 1

B ::= **0**
    v(B) ← 0

B ::= **1**
    v(B) ← $2^{s(B)}$

4.3.2 Formal Definition

An attribute grammar is a context-free grammar augmented with attributes, semantic rules, and conditions. Let G = (N,T,S,P) be a context-free grammar. For each nonterminal X ∈ N in the grammar, there are two finite disjoint sets I(X) and S(X) of inherited and synthesized attributes. For X = S, the start symbol, I(X) = ϕ.

Let A(X) = I(A) ∪ S(X) be the set of attributes of X. Each attribute A ∈ A(X) takes a value from some semantic domain (such as integers, strings of characters, or structures of

some type) associated with that attribute. These values are defined by semantic functions or semantic rules associated with the productions in P.

Consider a production p ∈ P of the form $X_0 ::= X_1\ X_2\ \ldots\ X_n$. Each synthesized attribute A ∈ $S(X_0)$ has its value defined in terms of the attributes in $A(X_1) \cup A(X_2) \cup \ldots \cup A(X_n) \cup I(X_0)$. Each inherited attribute A ∈ $I(X_k)$ for $1 \leq k \leq n$ has its value defined in terms of the attributes in $A(X_0) \cup S(X_1) \cup S(X_2) \cup \ldots \cup S(X_n)$. Each production may also have a set of conditions on the values of the attributes in $A(X_0) \cup A(X_1) \cup A(X_2) \cup \ldots \cup A(X_n)$ that further constrain an application of the production. The derivation of sentence in the attribute grammar is satisfied if and only if the context-free grammar is satisfied and all conditions are true.

4.3.3 *j--* Examples

We now look at how the semantics for some of the constructs in *j--* can be expressed using attribute grammars. Let us first consider examples involving just synthesized attributes. Semantics of literals involves just the synthesis of their type. This can be done using a synthesized attribute *type*.

```
literal ::= INT_LITERAL
              type(literal) ← INT
            | CHAR_LITERAL
              type(literal) ← CHAR
            | STRING_LITERAL
              type(literal) ← STRING
```

Semantics of a multiplicative expression involves checking the types of its arguments and making sure that they are numeric (int), and also synthesizing the type of the expression itself. Hence the following attribute grammar.

```
multiplicativeExpression ::= unaryExpression1
                                {STAR unaryExpression2}
   condition : type(unaryExpression1) = INT and type(unaryExpression2) = INT
                                type(multiplicativeExpression) ← INT
```

Now, consider a *j--* expression x * 5. The variable x must be of numeric (int) type. We cannot simply synthesize the type of x to int, but it must be synthesized elsewhere, that is, at the place where it is declared, and inherited where it is used. The language semantics also requires that the variable x be declared before the occurrence of this expression. Thus, we must inherit the context (symbol table) that stores the names of variables and their associated types, so we can perform a lookup to see if the variable x is indeed declared. The attribute grammar below indicates how we synthesize the context for

variables at the place where they are declared and how the context is inherited at the place the variables are used.

```
localVariableDeclarationStatement ::= type variableDeclarators SEMI
                        foreach variable in variableDeclarators
                            add(context, defn(name(variable), type))

variableDeclarator ::= IDENTIFIER [ASSIGN variableInitializer]
                                    name(variableDeclarator) ← IDENTIFIER

primary ::= …
          | variable
            defn ← lookup(context, name(variable))
            if defn = null
               error("Cannot find variable " + name(variable))
            else
               type(primary) ← type(defn)
```

`add()` and **`lookup()`** are auxiliary functions defined for adding variable definitions into a context (symbol table) and for looking up a variable in a given context. In the above attribute grammar, context is synthesized in localVariableDeclarationStatement and is inherited in primary.

Further Reading

The Java Language Specification, Third Edition (Gosling et al, 2005) is the best source of information on the semantic rules for the Java language.

Attribute grammars were first introduced in (Knuth, 1968). Chapter 3 of (Slonneger, 1994) offers an excellent treatment of Attribute Grammars; Chapter 7 discusses the application of Attribute Grammars in code generation. A general compilers text that makes extensive use of attribute grammars is (Waite and Goos, 1984). Chapter 5 of (Aho, Lam, Sethi and Ullman, 2007) also treats attribute grammars.

(Gamma, 1995) describes the Visitor Pattern as well as many other design patterns. Another good text describing how design patterns may be used in Java programming is (Jia, 2003). Yet another resource for design patterns is (Freeman et al, 2004).

Exercises

The *j--* compiler does not enforce all of the Java rules that it might. The following exercises allow the reader to rectify this.

4.1. The compiler does not enforce the Java rule that only one type declaration (i.e. class declaration in *j--*) be declared public. Repair this in one of the **`analyze()`** methods.

4.2. The **`analyze()`** method in **`JVariable`** assigns the type **`Type.ANY`** to a **`JVariable`** (e.g. **`x`**) in the AST that has not been declared in the symbol table. This prevents a cascading of multiple errors from the report of a single undeclared variable. But we could go further by declaring the variable (as **`Type.ANY`**) in the symbol table so as to suppress superfluous error reports in subsequent encounters of the variable. Make this improvement to analysis.

4.3. Go through *The Java Language Specification, Third Edition* (Gosling et al, 2005), and for each node type in the *j--* compiler, compile a list of rules that Java imposes on the language but are not enforced in *j--*.

a. Describe how you might implement each of these.
b. Which of these rules cannot be enforced?
c. Implement the enforcement of those rules that have not been implemented in *j--* but that one can.

4.4. Write an attribute grammar expressing the semantics of each of the statements in *j--*.

statement ::= block
| **if** parExpression statement [**else** statement]
| **while** parExpression statement
| **return** [expression] **;**
| **;**
| statementExpression **;**

4.5 Write an attribute grammar expressing the semantics of expressions in *j--*, that is, for productions from statementExpression through literal. See Appendix B for the *j--* syntax specification.

Implementation of analysis for new functionality is left for Chapter 6.