Type Checking

Outline

1. Introduction
2. The *j*-- Types
3. *j*-- Symbol Tables
4. Pre-analysis of *j*-- Programs
5. Analysis of *j*-- Programs

Introduction

Introduction

Type checking (aka semantic analysis) is the final step in the analysis phase, and includes the following:

Introduction

Type checking (aka semantic analysis) is the final step in the analysis phase, and includes the following:

- Determining the types of all names and expressions

Introduction

Type checking (aka semantic analysis) is the final step in the analysis phase, and includes the following:

- Determining the types of all names and expressions
- Insuring that all expressions are properly typed, for example that the operands of an operator have the proper types

Introduction

Type checking (aka semantic analysis) is the final step in the analysis phase, and includes the following:

- Determining the types of all names and expressions
- 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

Introduction

Type checking (aka semantic analysis) is the final step in the analysis phase, and includes the following:

- Determining the types of all names and expressions
- 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

Introduction

Type checking (aka semantic analysis) is the final step in the analysis phase, and includes the following:

- Determining the types of all names and expressions
- 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

Semantic analysis of *j--* programs involves all of the above

The j-- Types

The j-- Types

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

The j-- Types

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

j-- primitive types:

The 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

The 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`

The 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 many systems deal only with the lower 8 bits)

The 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 many systems deal only with the lower 8 bits)

j-- reference types:

The 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 many systems deal only with the lower 8 bits)

j-- reference types:

- arrays

The 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 many systems deal only with the lower 8 bits)

j-- reference types:

- arrays
- objects of a type described by a class declaration

The 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 many 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`

The 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 many 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

The j-- Types

The j-- Types

How do we represent the types `int`, `int[]`, `Factorial`, `String[][]`?

The j-- Types

How do we represent the types `int`, `int[]`, `Factorial`, `String[][]`?

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

The j-- Types

How do we represent the types `int`, `int[]`, `Factorial`, `String[][]`?

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

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

The j-- Types

How do we represent the types `int`, `int[]`, `Factorial`, `String[][]`?

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

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

Possible solutions:

The j-- Types

How do we represent the types `int`, `int[]`, `Factorial`, `String[][]`?

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

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

Possible solutions:

1. Java types are represented by objects of (Java) type `java.lang.Class`; since *j--* is a subset of Java, why not use `Class` objects to represent its types?

How do we represent the types `int`, `int[]`, `Factorial`, `String[][]`?

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

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

Possible solutions:

1. Java types are represented by objects of (Java) type `java.lang.Class`; since *j*-- is a subset of Java, why not use `Class` objects to represent its types?
2. Define an abstract class (or interface) `Type`, and concrete sub-classes (or implementations) `PrimitiveType`, `ReferenceType`, and `ArrayType`

The j-- Types

The j-- Types

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

The j-- Types

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

Since the parser does not know anything about types, we define two placeholder type representations:

The j-- Types

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

Since the parser does not know anything about types, 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

The j-- Types

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

Since the parser does not know anything about types, 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 `String[]`, until such time that they may resolved to their proper `Type` representation

The j-- Types

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

Since the parser does not know anything about types, 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 `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

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

Since the parser does not know anything about types, 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 `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

More specifically:

The j-- Types

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

Since the parser does not know anything about types, 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 `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

More specifically:

- A `TypeName` is resolved by looking it up in the current context, our symbol table representation, and the `Type` found replaces the `TypeName`, and finally, the `Type`'s accessibility from the place the `TypeName` is encountered is checked

The j-- Types

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

Since the parser does not know anything about types, 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 `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

More specifically:

- A `TypeName` is resolved by looking it up in the current context, our symbol table representation, and the `Type` found replaces the `TypeName`, and finally, the `Type`'s accessibility from the place the `TypeName` is encountered is checked
- Since an `ArrayTypeName` has a base type, the base type is resolved to a `Type`, whose `Class` representation becomes the base type for representing the array type

The j-- Types

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

Since the parser does not know anything about types, 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 `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

More specifically:

- A `TypeName` is resolved by looking it up in the current context, our symbol table representation, and the `Type` found replaces the `TypeName`, and finally, the `Type`'s accessibility from the place the `TypeName` is encountered is checked
- Since an `ArrayTypeName` has a base type, the base type is resolved to a `Type`, whose `Class` representation becomes the base type for representing the array type
- A `Type` resolves to itself

j-- Symbol Tables

j-- Symbol Tables

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

j-- Symbol Tables

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

In the *j--* compiler, the symbol table is a tree of `Context` objects, which spans the abstract syntax tree, with each `Context` corresponding to a region of scope in the *j--* source program

j-- Symbol Tables

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

In the *j*-- compiler, the symbol table is a tree of `Context` objects, which spans the abstract syntax tree, with each `Context` corresponding to a region of scope in the *j*-- source program

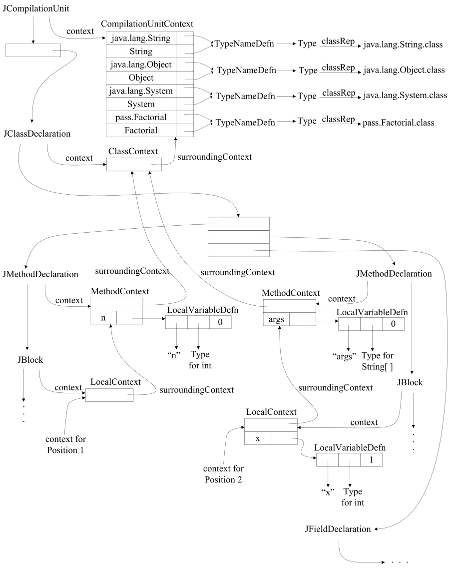
For example, reconsider the simple `Factorial` program. In this version we mark two locations in the program using comments: `position 1` and `position 2`

```
package pass;

import java.lang.System;

public class Factorial {
    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) {
        // position 2:
        int x = n;
        System.out.println(n + "! = " + factorial(x));
    }

    static int n = 5;
}
```

j-- Symbol Tables

j-- Symbol Tables

The symbol table for the `Factorial` program, and its relationship to the AST, is illustrated in figure below



j-- Symbol Tables

j-- Symbol Tables

The symbol table takes the form of a tree that corresponds to the shape of the AST

j-- Symbol Tables

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

j-- Symbol Tables

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

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

j-- Symbol Tables

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

1. The context pointer from the AST's `JCompilationUnit` node points to the `JCompilationUnitContext` that is at the root of the symbol table
2. The context pointer from the AST's `JClassDeclaration` points to a `ClassContext`

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

1. The context pointer from the AST's `JCompilationUnit` node points to the `JCompilationUnitContext` that 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`

j-- Symbol Tables

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

1. The context pointer from the AST's `JCompilationUnit` node points to the `JCompilationUnitContext` that 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`
4. The context pointer from the AST's two `JBlock`s each point to a `LocalContext`

j-- Symbol Tables

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

1. The context pointer from the AST's `JCompilationUnit` node points to the `JCompilationUnitContext` that 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`
4. The context pointer from the AST's two `JBlock`s each point to a `LocalContext`

From any particular location in the program, looking back towards the root `CompilationUnitContext`, the symbol table looks like a stack of contexts

The symbol table takes the form of a tree that corresponds to the shape of the AST

A `context`, ie, a node in this tree, captures the region of scope corresponding to the AST node that points to it

For example, in the above figure:

1. The context pointer from the AST's `JCompilationUnit` node points to the `JCompilationUnitContext` that 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`
4. The context pointer from the AST's two `JBlock`s each point to a `LocalContext`

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

j-- Symbol Tables

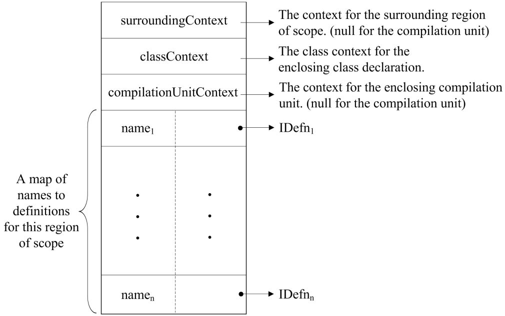
j-- Symbol Tables

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

j-- Symbol Tables

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

Type names are looked up in the `CompilationUnitContext`; to facilitate this, each context maintains three pointers to surrounding contexts, as illustrated in the following figure



j-- Symbol Tables

j-- Symbol Tables

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

j-- Symbol Tables

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`

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

j-- Symbol Tables

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
- User-defined types, that is, types introduced in clss declarations

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
- User-defined types, that is, types introduced in clss declarations

A `ClassContext` represents the scope within a class declaration; in the *j--* symbol table, no names are declared here, but if we were to add nested type declarations to *j--*, they might be declared here

j-- Symbol Tables

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
- User-defined types, that is, types introduced in clss declarations

A `ClassContext` represents the scope within a class declaration; in the *j*-- symbol table, no names are declared here, but if we were to add nested type declarations to *j*--, they might be declared here

A `MethodContext` represents the scope within a method declaration; a method's formal parameters are declared here

j-- Symbol Tables

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
- User-defined types, that is, types introduced in clss declarations

A `ClassContext` represents the scope within a class declaration; in the *j--* symbol table, no names are declared here, but if we were to add nested type declarations to *j--*, they might be declared here

A `MethodContext` represents the scope within a method declaration; a method's formal parameters are declared here

A `LocalContext` represents the scope within a block, which includes the block defining the body to a method; local variables are declared here

j-- Symbol Tables

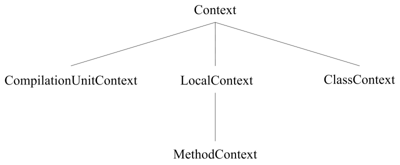
j-- Symbol Tables

Each kind of context derives from (extends) the class `Context`, which supplies the mapping from names to definitions (`IDefns`)

j-- Symbol Tables

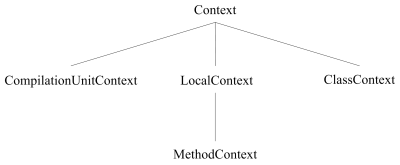
Each kind of context derives from (extends) the class `Context`, which supplies the mapping from names to definitions (`IDefns`)

The inheritance tree for contexts is illustrated in the following figure



Each kind of context derives from (extends) the class `Context`, which supplies the mapping from names to definitions (`IDefns`)

The inheritance tree for contexts is illustrated in the following figure

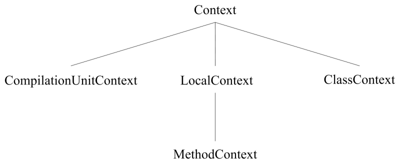


An `IDefn` is the interface type for symbol table definitions, which has two implementations:

j-- Symbol Tables

Each kind of context derives from (extends) the class `Context`, which supplies the mapping from names to definitions (`IDefns`)

The inheritance tree for contexts is illustrated in the following figure

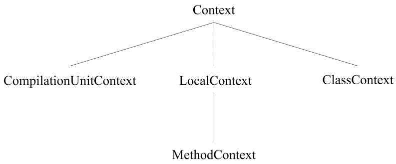


An `IDefn` is the interface type for symbol table definitions, which has two implementations:

1. A `TypeNameDefn`, which defines a type name; an `IDefn` of this sort encapsulates the `Type` that it denotes

Each kind of context derives from (extends) the class `Context`, which supplies the mapping from names to definitions (`IDefns`)

The inheritance tree for contexts is illustrated in the following figure



An `IDefn` is the interface type for symbol table definitions, which 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

j-- Symbol Tables

j-- Symbol Tables

Class member (field and method in *j--*) names are not declared in a `ClassContext`, but in the `Type`s that they declare

j-- Symbol Tables

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

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

```
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;
}
```

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Therefore, `preAnalyze()`need be defined only in the following types of AST nodes:

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Therefore, `preAnalyze()`need be defined only in the following types of AST nodes:

- `JCompilationUnit`

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Therefore, `preAnalyze()`need be defined only in the following types of AST nodes:

- `JCompilationUnit`
- `JClassDeclaration`

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Therefore, `preAnalyze()`need be defined only in the following types of AST nodes:

- `JCompilationUnit`
- `JClassDeclaration`
- `JFieldDeclaration`

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Therefore, `preAnalyze()`need be defined only in the following types of AST nodes:

- `JCompilationUnit`
- `JClassDeclaration`
- `JFieldDeclaration`
- `JMethodDeclaration`

Pre-analysis of j-- Programs

The semantic analysis of *j--* 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 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

The `preAnalyze()` method must traverse down the AST only far enough for:

- Declaring imported type names
- Declaring user-defined class names
- Declaring fields
- Declaring methods (including their signatures - the types of their parameters)

Therefore, `preAnalyze()`need be defined only in the following types of AST nodes:

- `JCompilationUnit`
- `JClassDeclaration`
- `JFieldDeclaration`
- `JMethodDeclaration`
- `JConstructorDeclaration`

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

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

Pre-analysis of j-- Programs

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

1. It creates a `CompilationUnitContext`

Pre-analysis of j-- Programs

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`

Pre-analysis of j-- Programs

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

Pre-analysis of j-- Programs

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 declaration, ie, creates a `Type` for each declared class, whose `classRep` refers to a `Class` object for an empty class; for example, 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 {}
```

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 declaration, ie, creates a `Type` for each declared class, whose `classRep` refers to a `Class` object for an empty class; for example, 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 {}
```

5. Finally, `preAnalyze()` invokes itself for each of the type declarations in the compilation unit

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

```
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);
    }
```

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

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

Pre-analysis of j-- Programs

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

1. It firstly creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`

Pre-analysis of j-- Programs

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

1. It firstly creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`
2. It resolves the class's super type

Pre-analysis of j-- Programs

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

1. It firstly 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

Pre-analysis of j-- Programs

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

1. It firstly 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

Pre-analysis of j-- Programs

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

1. It firstly 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, which causes field declarations, constructors and method declarations (but with empty bodies) to be added to the `CLEmitter` instance

Pre-analysis of j-- Programs

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

1. It firstly 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, which 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

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

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

1. It firstly 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, which 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

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

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

Pre-analysis of j-- Programs

Here is the code for `preAnalyze()` in `JClassDeclaration`

```
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-analysis of j-- Programs

Pre-analysis of j-- Programs

```
    // 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());
    }
}
```

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

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

Pre-analysis of j-- Programs

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

1. It resolves the types of the formal parameters

Pre-analysis of j-- Programs

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

1. It resolves the types of the formal parameters
2. It resolves the return type

Pre-analysis of j-- Programs

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

1. It resolves the types of the formal parameters
2. It resolves the return type
3. It checks proper use of the `abstract` modifier

Pre-analysis of j-- Programs

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

1. It resolves the types of the formal parameters
2. It resolves the return type
3. It checks proper use of the `abstract` modifier
4. It computes the method descriptor

Pre-analysis of j-- Programs

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

1. It resolves the types of the formal parameters
2. It resolves the return type
3. It checks proper use of the `abstract` modifier
4. It computes the method descriptor
5. It generates (partial) code for the method

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

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);
}
```

Pre-analysis of j-- Programs

The code for `partialCodegen()` is as follows:

```java
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);
    }
}
```

Pre-analysis of j-- Programs

Pre-analysis for a `JFieldDeclaration` is similar to that for a `JMethodDeclaration`, and does the following:

Pre-analysis of j-- Programs

Pre-analysis for a `JFieldDeclaration` is similar to that for a `JMethodDeclaration`, and does the following:

1. Enforces the rule that fields may not be declared `abstract`

Pre-analysis for a `JFieldDeclaration` is similar to that for a `JMethodDeclaration`, and does the following:

1. Enforces the rule that fields may not be declared `abstract`
2. Resolves the field's declared type

Pre-analysis of j-- Programs

Pre-analysis for a `JFieldDeclaration` is similar to that for a `JMethodDeclaration`, and does the following:

1. Enforces the rule that fields may not be declared `abstract`
2. Resolves the field's declared type
3. Generates the JVM code for the field declaration, via the `CLEmitter` created for the enclosing class declaration

Pre-analysis for a `JFieldDeclaration` is similar to that for a `JMethodDeclaration`, and does the following:

1. Enforces the rule that fields may not be declared `abstract`
2. Resolves the field's declared type
3. Generates the JVM code for the field declaration, via the `CLEmitter` created for the enclosing class declaration

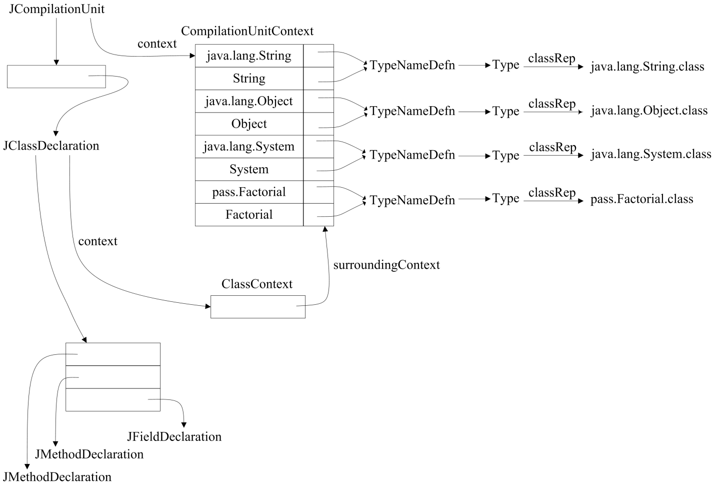
The code itself is rather simple

```
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);
    }
}
```

Pre-analysis of j-- Programs

Pre-analysis of j-- Programs

The following figure illustrates how much of the symbol table is constructed for our `Factorial` program once pre-analysis is complete



Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing field and local variable initializations as assignments

Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing field and local variable initializations as assignments
- Declaring both formal parameters and local variables

Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing 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

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing 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

Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing 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

Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing 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
- Doing a limited amount of tree surgery

Analysis of j-- Programs

The analysis phase, ie, the `analyze()` method, recursively descends throughout the AST all the way to its leaves:

- Re-writing 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
- Doing a limited amount of tree surgery

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;
}
```

Analysis of j-- Programs

Analysis of j-- Programs

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;
}
```

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;
}
```

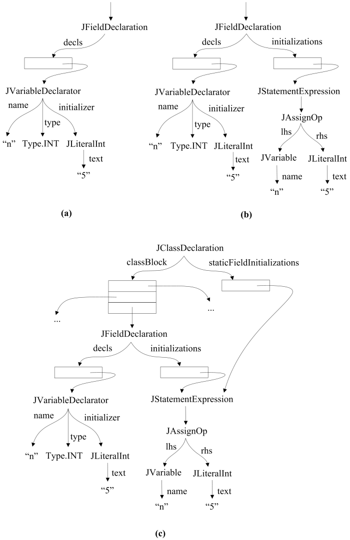
In `JClassDeclaration`, `analyze()` separates the assignment statements into two lists: one for the static fields and one for the instance fields

```
// 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);
        }
    }
}
```

Analysis of j-- Programs

Analysis of j-- Programs

The following figure shows how the static field declaration (`static int n = 5;`) in the `Factorial` program is rewritten



Analysis of j-- Programs

Both formal parameters and local variables are declared in the symbol table and allocated locations within a method invocation's run-time stack frame

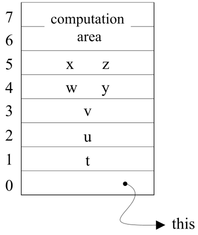
Both formal parameters and local variables are declared in the symbol table and allocated locations within a method invocation's run-time stack frame

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;
    }
}
```

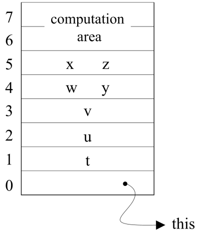
Analysis of j-- Programs

The stack frame allocated for an invocation of `foo()` at run time by the JVM is shown below



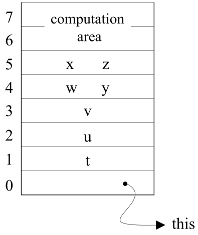
Analysis of j-- Programs

The stack frame allocated for an invocation of `foo()` at run time by the JVM is shown below



The code for analyzing a `JMethodDeclaration` performs four steps:

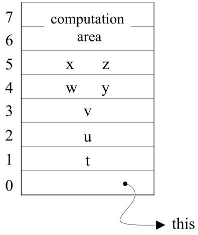
The stack frame allocated for an invocation of `foo()` at run time by the JVM is shown below



The code for analyzing a `JMethodDeclaration` performs four steps:

1. It creates a new `MethodContext`, whose `surroundingContext` points back to the previous `ClassContext`

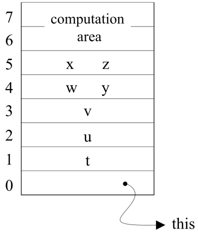
The stack frame allocated for an invocation of `foo()` at run time by the JVM is shown below



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

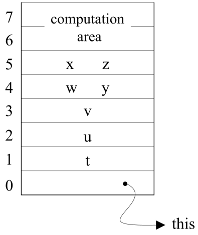
The stack frame allocated for an invocation of `foo()` at run time by the JVM is shown below



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

The stack frame allocated for an invocation of `foo()` at run time by the JVM is shown below



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

Analysis of j-- Programs

```
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;
}
```

Analysis of j-- Programs

The code for analyzing a `JBlock` performs two steps:

The code for analyzing a `JBlock` performs 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

The code for analyzing a `JBlock` performs 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 `JVariableDeclaration`s 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

The code for analyzing a `JBlock` performs two steps:

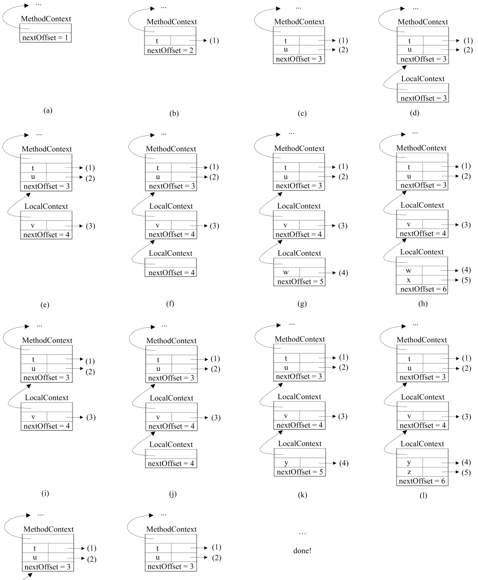
1. It creates a new `LocalContext`, whose `surroundingContext` points back to the previous `MethodContext` (or `LocalContext` in the case of nested blocks); its `nextOffset` value is copied from the previous context
2. It analyzes each of the body's statements; any `JVariableDeclaration`s declare their 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

```
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;
}
```

Analysis of j-- Programs

Analysis of j-- Programs

The stages of the symbol table in analyzing `Locals.foo()`



Analysis of j-- Programs

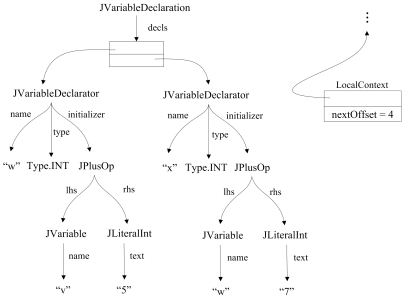
A local variable declaration is represented in the AST with a `JVariableDeclaration`; for example, consider the local variable declaration from `Locals`

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

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 below



Analysis of j-- Programs

Analysis of a `JVariableDeclaration` involves the following:

Analysis of a `JVariableDeclaration` involves the following:

1. `LocalVariableDefn`s and their corresponding stack frame offsets are allocated for each of the declared variables

Analysis of a `JVariableDeclaration` 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

Analysis of a `JVariableDeclaration` 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

Analysis of a `JVariableDeclaration` involves the following:

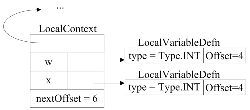
1. `LocalVariableDefn`s and their corresponding stack frame offsets are allocated for each of the declared variables
2. The code checks to make sure that the declared variables do not shadow existing local variables
3. The variables are declared in the local context
4. Any initializations are rewritten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

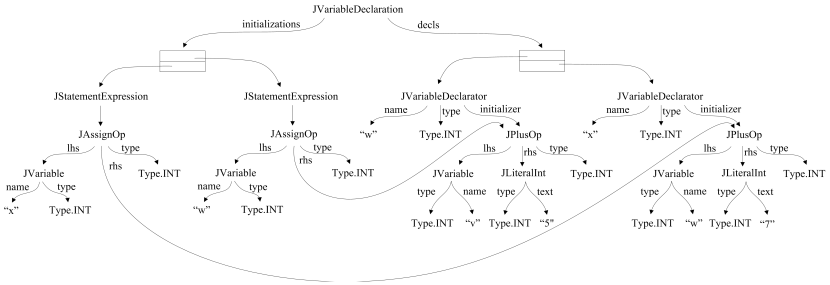
Analysis of j-- Programs

```
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 j-- Programs

The sub-tree for `int w = v + 5, x = w + 7;` after analysis is shown below





Analysis of j-- Programs

Analysis of j-- Programs

Simple variables (local variables or fields) are represented in the AST as `JVariable` nodes

Analysis of j-- Programs

Simple variables (local variables or fields) are represented in the AST as `JVariable` nodes

Analysis of simple variables involves looking their names up in the symbol table to find their types

Analysis of j-- Programs

Simple variables (local variables or fields) are represented in the AST as `JVariable` nodes

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, 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`

Analysis of j-- Programs

Simple variables (local variables or fields) are represented in the AST as `JVariable` nodes

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, 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`

```
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);
        }
```

Analysis of j-- Programs

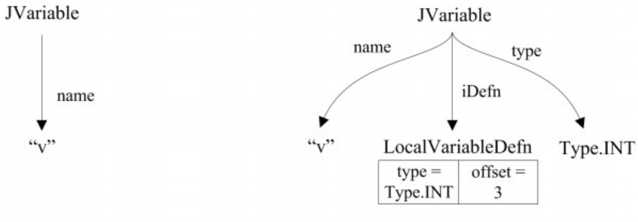
Analysis of j-- Programs

```
    } 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;
}
```

Analysis of j-- Programs

```
    } 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;
}
```

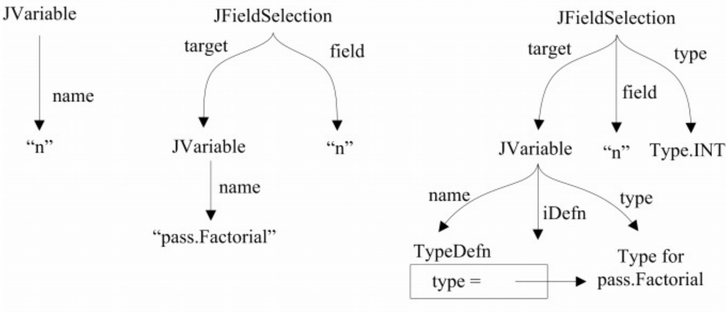
For example, the AST node for the local variable `v` in the statement `return t + v;` in `Locals.foo()`, before and after analysis, is shown below



Analysis of j-- Programs

As another example, consider the analysis of the static field `n`, when it appears in the `main()` method of our `Factorial` class; the AST node for the field, before and after analysis, is shown below

As another example, consider the analysis of the static field `n`, when it appears in the `main()` method of our `Factorial` class; the AST node for the field, before and after analysis, is shown below



Analysis of j-- Programs

Both field selections and message expressions have targets

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, and in a message expression, the target is an object or class to which one is sending a message

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, and 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

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, and 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:

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, and 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`;

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, and 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

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, and 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`

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, and 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`, deferring the decision until analysis

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, and 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`, deferring the decision until analysis

The `reclassify()` method in `AmbiguousName` is based on the rules in the Java Language Specification for reclassifying an ambiguous name

Analysis of j-- Programs

Analysis of j-- Programs

```java
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;
}
```

Analysis of j-- Programs

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

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 the `AmbiguousName` to determine the structure of the expression that it denotes, which it does by looking at the ambiguous `java.lang.System.out` from left to right:

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 the `AmbiguousName` to determine the structure of the expression that it denotes, which it does 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.

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 the `AmbiguousName` to determine the structure of the expression that it denotes, which it does 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`

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 the `AmbiguousName` to determine the structure of the expression that it denotes, which it does 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.

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 the `AmbiguousName` to determine the structure of the expression that it denotes, which it does 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

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 the `AmbiguousName` to determine the structure of the expression that it denotes, which it does 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`

Analysis of j-- Programs

Analysis of j-- Programs

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

Analysis of j-- Programs

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

Analysis of j-- Programs

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`

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`
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

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`
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

Analysis of j-- Programs

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`
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

```
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;
        }
```

Analysis of j-- Programs

```
        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;
}
```

Analysis of j-- Programs

Analysis of j-- Programs

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

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

Analysis of j-- Programs

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)

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

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), and if no such method is found, it reports an error

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), and 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();
    }
```

Analysis of j-- Programs

```
// 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());
    }
}
```

Analysis of j-- Programs

```
    // 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;
}
```

Analysis of j-- Programs

Analysis of j-- Programs

The rest of analysis, as defined for the various kinds of AST nodes is about computing and checking types and enforcing additional *j*-- rules

Analysis of j-- Programs

The rest of analysis, as defined for the various kinds of AST nodes is about computing and checking types and enforcing additional *j*-- rules

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

The rest of analysis, as defined for the various kinds of AST nodes is about computing and checking types and enforcing additional *j--* rules

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

Analysis of `JIfStatement`

```
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;
}
```

Analysis of j-- Programs

The rest of analysis, as defined for the various kinds of AST nodes is about computing and checking types and enforcing additional *j*-- rules

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

Analysis of `JIfStatement`

```
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;
}
```

Analysis of `JSubtractOp`

```
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;
 }
```

Analysis of j-- Programs

Analysis of j-- Programs

Analysis of JPlusOp

```
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;
}
```

Analysis of j-- Programs

Analysis of `JPlusOp`

```
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;
}
```

Analysis of `JStringConcatenateOp`

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

Analysis of j-- Programs

Analysis of `JPlusOp`

```
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;
}
```

Analysis of `JStringConcatenateOp`

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

Analysis of `JLiteralInt`

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

Analysis of j-- Programs

Analysis of j-- Programs

The *j--* language is stricter than is Java when it comes to types

The *j--* language is stricter than is Java when it comes to types

There are no implied conversions in *j--*; when one assigns an expression to a variable, the types must match exactly, and the same goes for actual parameters to messages matching the formal parameters of methods

The *j*-- language is stricter than is Java when it comes to types

There are no implied conversions in *j*--; when one assigns an expression to a variable, the types must match exactly, and the same goes for actual parameters to messages matching the formal parameters of methods

This does not exclude polymorphism; for example if type `Bar` extends type `Foo`, if `bar` is a variable of type `Bar` and `foo` is a variable of type `Foo`, we can say

```
foo = (Foo) bar;
```

to keep the *j*-- compiler happy

The *j*-- language is stricter than is Java when it comes to types

There are no implied conversions in *j*--; when one assigns an expression to a variable, the types must match exactly, and the same goes for actual parameters to messages matching the formal parameters of methods

This does not exclude polymorphism; for example if type `Bar` extends type `Foo`, if `bar` is a variable of type `Bar` and `foo` is a variable of type `Foo`, we can say

```
foo = (Foo) bar;
```

to keep the *j*-- compiler happy

Of course, the object that `bar` refers to could be of type `Bar` or any of its sub-types; polymorphism has not gone away

The *j*-- language is stricter than is Java when it comes to types

There are no implied conversions in *j*--; when one assigns an expression to a variable, the types must match exactly, and the same goes for actual parameters to messages matching the formal parameters of methods

This does not exclude polymorphism; for example if type `Bar` extends type `Foo`, if `bar` is a variable of type `Bar` and `foo` is a variable of type `Foo`, we can say

```
foo = (Foo) bar;
```

to keep the *j*-- compiler happy

Of course, the object that `bar` refers to could be of type `Bar` or any of its sub-types; polymorphism has not gone away

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

```
(Type2) expression of Type1
```

must determine two things:

Analysis of j-- Programs

The *j--* language is stricter than is Java when it comes to types

There are no implied conversions in *j--*; when one assigns an expression to a variable, the types must match exactly, and the same goes for actual parameters to messages matching the formal parameters of methods

This does not exclude polymorphism; for example if type `Bar` extends type `Foo`, if `bar` is a variable of type `Bar` and `foo` is a variable of type `Foo`, we can say

```
foo = (Foo) bar;
```

to keep the *j--* compiler happy

Of course, the object that `bar` refers to could be of type `Bar` or any of its sub-types; polymorphism has not 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`, ie, that the cast is valid

Analysis of j-- Programs

The *j--* language is stricter than is Java when it comes to types

There are no implied conversions in *j--*; when one assigns an expression to a variable, the types must match exactly, and the same goes for actual parameters to messages matching the formal parameters of methods

This does not exclude polymorphism; for example if type `Bar` extends type `Foo`, if `bar` is a variable of type `Bar` and `foo` is a variable of type `Foo`, we can say

```
foo = (Foo) bar;
```

to keep the *j--* compiler happy

Of course, the object that `bar` refers to could be of type `Bar` or any of its sub-types; polymorphism has not 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`, ie, that the cast is valid
2. The type of the result, which is simply `Type2`

Analysis of j-- Programs

To determine (1) we must consider the possibilities for `Type1` and `Type2`

To determine (1) we must consider the possibilities for `Type1` and `Type2`

1. Any type may be cast to itself (aka identity cast)

To determine (1) we must consider the possibilities for `Type1` and `Type2`

1. Any type may be cast to itself (aka identity cast)
2. An arbitrary reference type may be cast to another reference type if and only if either one of the following holds:

To determine (1) we must consider the possibilities for `Type1` and `Type2`

1. Any type may be cast to itself (aka identity cast)
2. An arbitrary reference type may be cast to another reference type if and only if either one of the following holds:
    1. The first type is a sub-type of (extends) the second type; this is called widening and requires no action at run time

To determine (1) we must consider the possibilities for `Type1` and `Type2`

1. Any type may be cast to itself (aka identity cast)
2. An arbitrary reference type may be cast to another reference type if and only if either one of the following holds:
    1. The first type is a sub-type of (extends) the second type; this is called widening and requires no action at run time
    2. 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

To determine (1) we must consider the possibilities for `Type1` and `Type2`

1. Any type may be cast to itself (aka identity cast)
2. An arbitrary reference type may be cast to another reference type if and only if either one of the following holds:
   1. The first type is a sub-type of (extends) the second type; this is called widening and requires no action at run time
   2. 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
3. The following table summarizes other casts, and 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

Analysis of j-- Programs

Analysis in JCastOp

```
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;
}
```

Analysis in `JCastOp`

```
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;
}
```

A converter for narrowing one reference type to another (more specific) reference sub-type

```
class NarrowReference implements Converter {
    private Type target;

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

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

Analysis of j-- Programs

In Java, every variable (whether it be a local variable or a field) must be definitely assigned before it is accessed in a computation, ie, it must appear on the left hand side of the `=` operator before it is accessed

In Java, every variable (whether it be a local variable or a field) must be definitely assigned before it is accessed in a computation, ie, it must appear on the left hand side of the `=` operator before it is accessed

We do not have this rule in *j*--, so we need not enforce it in our compiler

In Java, every variable (whether it be a local variable or a field) must be definitely assigned before it is accessed in a computation, ie, it must appear on the left hand side of the `=` operator before it is accessed

We do not have this rule in *j*--, so we need not enforce it in our compiler

Enforcing the definite assignment rule requires data flow analysis, which determines where in the program variables are defined (assigned values), where in the program the variables' values are used and so where in the program those values are valid (from assignment to last use)

Analysis of j-- Programs

In Java, every variable (whether it be a local variable or a field) must be definitely assigned before it is accessed in a computation, ie, it must appear on the left hand side of the `=` operator before it is accessed

We do not have this rule in *j--*, so we need not enforce it in our compiler

Enforcing the definite assignment rule requires data flow analysis, which determines where in the program variables are defined (assigned values), where in the program the variables' values are used and so where in the program those values are valid (from assignment to last use)

Our JVM to MIPS translator performs data-flow analysis as part of computing live intervals for register allocation