# Backstage Java *(draft)*

Zachary Palmer
Joe Riley

March 24, 2010

# Contents

# Preface

***Draft comment:*** *TODO*

# Chapter 1

# Introduction

***Draft comment:*** *TODO*

## 1.1 Terminology

***Draft comment:*** *TODO: definitions of "metaprogram", "object program", "stage", etc.*

## 1.2 Example Programs

***Draft comment:*** *TODO*

# Chapter 2

# Grammar

The BSJ language is a superset of the Java language; that is, any legal Java program is a legal BSJ program. As a result, the lexical and syntactic structure of BSJ is very similar to that of Java. Notation for the grammar presented within this document is given in §2.1. The few additions to lexical structure that exist are discussed in §2.2. The syntax of BSJ is discussed throughout the remainder of this document and can be found in §3, §5, and §7.

## 2.1 Notation

As BSJ is a supplement to the Java language, this document uses the same grammar format as the *Java Language Specification*. Each production for a rule is displayed on a single line, indented from the original rule declaration. The suffix "opt" is used to refer to a part of a production which is optional. For further information about this syntax, please consult §2.4 of the *Java Language Specification*.

For sake of convenience and illustration, this document will often include rules from the *Java Language Specification* in modified and unmodified forms. If a rule is included that has been changed from its presentation in the *Java Language Specification*, its name is suffixed with the term "*modified*" in parentheses. If it has not been changed, its name is suffixed with the term "*unmodified*".

## 2.2 Lexical Structure

***Draft comment:*** *TODO: discuss #depends and other such keywords. Note that many new symbols in BSJ (such as [:) are composed of existing Java lexical structure.*

# Chapter 3

# Metaprograms

A metaprogram is introduced in BSJ by the use of the `[:` and `:]` metaprogram declaration operators. These operators may appear in any of the following locations:

- At the top level of a compilation unit where a type declaration would normally appear.
- At the top level of a type declaration, such as immediately within a class or interface.
- Anywhere a statement may appear.

If the metaprogram declaration operators have no content (that is, they are separated only by whitespace), then the semantics of the metaprogram are the same as the semantics of a non-operation (`;` in the Java syntax). This fact is a consequence of the following rules but bears mentioning here for purposes of clarity and confirmation.

The formal syntactic effect of the above is illustrated below. Some rules have been reproduced from the *Java Language Specification* for convenience.

*TypeDeclaration (modified):*

> *ClassDeclaration*
> *InterfaceDeclaration*
> *BsjMetaprogram*

*ClassBodyDeclaration (modified):*

> *ClassMemberDeclaration*
> *InstanceInitializer*
> *StaticInitializer*
> *ConstructorDeclaration*
> *BsjMetaprogram*

*InterfaceMemberDeclaration (modified):*

> *ConstantDeclaration*
> *AbstractMethodDeclaration*
> *ClassDeclaration*

*InterfaceDeclaration*

*BsjMetaprogram*

`;`

*AnnotationTypeElementDeclaration (modified):*

*AnnotationTypeElementDeclaration:*

*AbstractMethodModifiers*$_{opt}$ *Type Identifier* `( )` *DefaultValue*$_{opt}$ `;`

*ConstantDeclaration*

*ClassDeclaration*

*InterfaceDeclaration*

*EnumDeclaration*

*AnnotationTypeDeclaration*

*BsjMetaprogram*

`;`

*BlockStatements (unmodified):*

*BlockStatement*

*BlockStatements BlockStatement*

*BlockStatement (modified):*

*LocalVariableDeclarationStatement*

*ClassDeclaration*

*BsjMetaprogram*

*Statement*

*BsjMetaprogram:*

`[ :` *Preamble*$_{opt}$ *BlockStatements* `: ]`

As demonstrated by the previous grammar description, a metaprogram consists of an optional preamble followed by a number of block statements. The block statements are used as the body of the metaprogram to be executed. The execution semantics of metaprograms are discussed in §6.2.

It should be noted that, like `@interface`, the metaprogram start and end operators are comprised of two different tokens; that is, it is legal to write "`[ : :]`" to indicate an empty metaprogram (using a space between the bracket and colon). For clarity, however, this should not be done in practice. This allows parsers with a rigid set of assumptions about the Java language to be more easily adapted to parsing BSJ.

## 3.1 Preamble

The metaprogram preamble allows the metaprogrammer to define attributes of the metaprogram (the metaprogram metadata). These attributes control the execution semantics and the execution order of the metaprograms in a compilation unit. A preamble consists of a number of preamble statements; preamble statements always begin with a preamble statement keyword. To avoid collision with the existing Java namespace, preamble statements are always prefixed with the hash sign (`#`) which does not appear anywhere in the syntax of the Java language itself.

*Preamble:*

*MetaprogramImportDeclarationList*$_{opt}$ *MetaprogramModeDeclaration*$_{opt}$ *MetaprogramTargetDeclaration*$_{o}$
*MetaprogramDependencyDeclaration*$_{opt}$

### 3.1.1 Metaprogram Imports

Metaprogram imports take on the same form as normal imports except with the `#import` keyword in place of the `import` keyword. The body of the metaprogram reacts to the metaprogram import in the same way that the body of an object program reacts to a normal import.

*MetaprogramImportDeclarationList:*

*MetaprogramImportDeclaration*

*MetaprogramImportDeclarationList MetaprogramImportDeclaration*

*MetaprogramImportDeclaration:*

`#import` *MetaprogramImportBody* `;`

*MetaprogramImportBody:*

*TypeName*

*PackageOrTypeName* `.` `*`

`static` *TypeName* `.` *Identifier*

`static` *TypeName* `.` `*`

As a form of syntactic sugar, metaprogram imports may also appear in the import declarations section of a compilation unit. Placing a metaprogram import statement in the compilation unit's import declarations is eqvuialent to placing that same metaprogram import statement in the preamble of every metaprogram contained in that source unit. Such a metaprogram import is termed a "global metaprogram import." For instance, the compilation unit

```
#import com.example.*;
public class Example {
    [:
        foo();
    :]
}
[:
    bar();
:]
```

is equivalent to the compilation unit

```
public class Example {
    [:
        #import com.example.*;
        foo();
    :]
}
[:
    #import com.example.*;
    bar();
:]
```

If a metaprogram introduces another metaprogram to the compilation unit, the latter metaprogram's global imports are determined after the prior metaprogram's execution terminates. If the prior metaprogram adds or removes global metaprogram imports to or from the compilation unit, those modifications will affect the latter metaprogram; this is true whether the prior metaprogram modified the global metaprogram imports first or added the latter metaprogram first.

In addition to the imports already listed, the following imports are assumed for every metaprogram:

- `edu.jhu.cs.bsj.compiler.ast.*`
- `edu.jhu.cs.bsj.compiler.ast.node.*`
- `edu.jhu.cs.bsj.compiler.ast.node.meta.*`
- `edu.jhu.cs.bsj.compiler.metaprogram.*`

### 3.1.2 Mode Declarations

The operational mode of a metaprogram affects the nodes it can access and modify. By default, metaprograms operate in `normal` mode, giving them the ability to perform a restricted set of modifications to their peers and the ability to read any node in the AST. This is done to limit the scope of impact that metaprograms have and prevent counterintuitive side effects.

In some cases, however, metaprograms require the freedom to arbitrarily mutate some nodes or to perform insertions outside of their normal scope. In this case, a mode declaration can be provided which alters the metaprogram's mode of operation. A full discussion of metaprogram modes is found in §3.3.

*MetaprogramModeDeclaration:*

    `#mode` *MetaprogramModeList* `;`

*MetaprogramModeList*

    *MetaprogramMode*

    *MetaprogramModeList* `,` *MetaprogramMode*

*MetaprogramMode:*

    *MetaprogramLocalMode*

    *MetaprogramPackageMode*

*MetaprogramLocalMode:*

    `readOnly`

    `insert`

    `mutate`

    `fullMutate`

*MetaprogramPackageMode:*

    `packageRead`

    `packageInsert`

It should be noted that none of the metaprogram modes are keywords in BSJ; they may be used as variable names, method names, and so on just as in Java. Mode names are always lexically compatible with identifiers and may be tokenized as such, but a parser is obligated to produce an AST which contains only legal modes.

### 3.1.3 Dependency Declarations

Metaprogram dependency declarations control the order in which metaprograms run. Each metaprogram has a *target* which groups it with other metaprograms for purposes of execution. Metaprograms can also depend upon targets; these dependencies form the basis of the execution order. See §4 for a complete explanation of the semantic meaning of these declarations.

*MetaprogramDependencyDeclaration:*

`#depends` *MetaprogramTargetNameList* `,`$_{opt}$ `;`

*MetaprogramTargetNameList:*

*MetaprogramTargetName*

*MetaprogramTargetNameList* `,` *MetaprogramTargetName*

*MetaprogramTargetDeclaration:*

`#target` *MetaprogramIdentifierList* `,`$_{opt}$ `;`

*MetaprogramIdentifierList:*

*Identifier*

*MetaprogramIdentifierList* `,` *Identifier*

The syntax and semantics of *MetaprogramTargetName* are discussed in §3.2.

## 3.2 Target Names

§6.5 of the *Java Language Specification* specifies the procedure used to determine the meaning of a name in the Java language. Initially, names are placed in one of six categories. The BSJ language introduces a seventh category of name: *MetaprogramTargetName*. The syntax for a metaprogram target name is similar to that of other names in the Java language:

*MetaprogramTargetName:*

*Identifier*

*TypeName* `.` *Identifier*

Once names are initially categorized, the rules in §6.5.1 of the *Java Language Specification* indicate how names are classified according to context. Metaprogram targets are discussed in two locations in the BSJ language: in metaprogram target declarations and in metaprogram dependency declarations. When a target is declared, it may not be qualified; as a result, it is impossible for a metaprogram target name to appear in a metaprogram target declaration. When a metaprogram dependency is declared, the provided name is always unambiguously a *MetaprogramTargetName*. Thus, the following amendment to §6.5.1 applies:

A name is syntactically qualified as a *MetaprogramTargetName* in these contexts:

- In a metaprogram dependency declaration (§3.1.3).

A name is syntactically qualified as a *TypeName* in these contexts:

- To the left of a `.` in a *MetaprogramTargetName*.

Metaprogram target names are declared by the use of the *MetaprogramTargetDeclaration* (§3.1.3) and explicitly indicate only the trailing identifier. If the metaprogram target has a qualified name, that name is inferred based on the location of the metaprogram (§3.2.1). When a *MetaprogramDependencyDeclaration* (§3.1.3) is issued, it may either use that qualified name or a simple name, the latter of which is interpreted according to the rules in §3.2.2.

As indicated in §4, it is legal for multiple metaprograms to share the same target. It is also legal for a metaprogram to participate in multiple targets. Metaprograms may depend upon any number of targets. If a metaprogram depends upon a target which contains no metaprograms, a compile-time error occurs.

### 3.2.1 Target Name Qualification

The hierarchical structure of the Java namespace is utilized to express the names of BSJ metaprogram targets. A metaprogram target may or may not have a qualified name; if it does not, it is impossible to refer to that metaprogram target by use of a qualified name. In this fashion, metaprogram target names are similar to type names. The rules used to determine the qualified name of a metaprogram target are as follows:

- If the target is declared in a metaprogram at the top level of a compilation unit, the qualified name of the metaprogram target is the qualified name of the public type which could appear in the compilation unit followed by a `.` and followed by the identity of the target.
- If the target is declared in a metaprogram whose enclosing type has a qualified name, the qualified name of the metaprogram target is the qualified name of the enclosing type followed by a `.` and followed by the identity of the target.
- Otherwise, the metaprogram target has no qualified name.

The following example code contains comments which help to illustrate this concept. It is assumed that the following code is contained in a file in the default package named `Foo.bsj`.

```
[: #target x; :] // QN = Foo.x
public class Foo {
    [: #target x; :] // QN = Foo.x
    [: #target y; :] // QN = Foo.y
    static class Bar {
        [: #target x; :] // QN = Foo.Bar.x
    }
}
[: #target y; :] // QN = Foo.y
class Baz {
    [: #target x; :] // QN = Baz.x
    private Object o = new Object() {
        [: #target x; :] // no QN for this target
    }
}
```

The first and second metaprograms (whose qualified target names are both "Foo.x") are members of the same target. This is of particular significance when determining the order in which metaprograms are executed.

### 3.2.2 Dependency Names

When a metaprogram expresses a dependency on a target, it may refer to the target in one of two ways: either by a simple name or by a qualified name. The meaning of this reference is determined as follows:

- If the metaprogram uses a simple name and appears at the top level of a compilation unit, the metaprogram dependency refers to the target with a name of the concatenation of the following: the qualified name of the public type which could appear in the compilation unit, a `.`, and the simple name of the dependency.
- If the metaprogram uses a simple name and appears in an enclosing type which has a qualified name, the metaprogram dependency refers to the target with a name of the concatenation of the following: the qualified name of the enclosing type, a `.`, and the simple name of the dependency.
- If the metaprogram uses a simple name and appears in an enclosing type which has no qualified name, the metaprogram dependency refers to the target with the same name which appears within the same enclosing type.
- If the metaprogram uses a qualified name, the metaprogram dependency refers to the target with that qualified name's canonical form.

The following example code will help to illustrate this concept and is discussed below. It is assumed that the following code is contained in a file in the default package named `Foo.bsj`. Metaprograms are given arbitrary names here to help clarify the resulting dependencies.

```
[: #target x; :] // Metaprogram A
public class Foo
{
    [: #target x; :] // Metaprogram B
    static class Bar
    {
        [: #target x; :] // Metaprogram C
        [: #depends x; :] // Metaprogram D
    }
}

[: #depends x; :] // Metaprogram E
[: #depends Foo.x; :] // Metaprogram F
[: #depends Foo.Bar.x; :] // Metaprogram G
```

```
class Baz
{
    private Object o = new Object(){
        [: #target x; :] // Metaprogram H
        [: #depends x; :] // Metaprogram I
    }
}
```

In the above example, metaprogram D uses a simple name to refer to its target. As a result, this simple name is interpreted as being qualified by its enclosing type and results in the qualified metaprogram target name “Foo.Bar.x”. This indicates that Metaprogram D depends on Metaprogram C.

Metaprogram E also uses a simple name but has no enclosing type. Since this compilation unit is named such that its public type must be named “Foo”, the target name “x” is interpreted as the qualified name “Foo.x”. As a result, metaprogram E dependson metaprograms A and B (because, as shown in the example in the previous section, both of those metaprograms share the same target).

Metaprograms F and G use qualified names to indicate their dependencies. Metaprogram F depends on metaprograms A and B while metaprogram G depends on metaprogram C.

Metaprogram I uses a simple name from within an anonymous class. As anonymous classes do not have qualified names, the simple name for metaprogram I is taken to indicate the target in the same enclosing type with the same name. Thus, metaprogram I depends on metaprogram H.

## 3.3 Permissions and Modes

The execution of a metaprogram does not have the ability to affect the entire AST in an arbitrary way. In order to prevent accidental modification leading to cryptic and subtle bugs, metaprograms are only allowed to modify specific portions of the AST. The access to the AST held by a metaprogram is termed its *permission*. Permission is granted on a node-by-node basis. BSJ metaprograms can hold three kinds of permission to a given node: *Read*, *Insert*, or *Mutate*.

The permissions held by a metaprogram are primarily determined by two factors: the location of the metaprogram in the source file and the operational mode of the metaprogram. The operational mode may be modified by a metaprogram mode declaration (§3.1.2). Metaprogram modes are divided into two categories: *local modes* which describe the permissions a metaprogram has in its immediate vicinity and *package modes* which describes the permissions a metaprogram has to package nodes throughout the complete AST. The legal local modes are `readOnly`, `insert`, `mutate`, and `fullMutate`; the legal package modes are `packageRead` and `packageInsert`. It is a compile-time error for more than one local mode or more than one package mode to appear in the same metaprogram mode declaration.

It is legal for a metaprogram to skip declaring one or both of these kinds of modes. If no declaration appears for a local mode, `insert` is assumed. If no declaration appears for a package mode, `packageRead` is assumed.

A metaprogram’s permissions are assigned as follows:

***Draft comment:*** *TODO: add rules for annotational metaprograms here*

- All metaprograms are granted *Read* permission to the root package.
- If the metaprogram declares `insert` local mode or does not declare a local mode, then
    - ♦ If the metaprogram's anchor is in the top level of a compilation unit, it is given *Insert* permission to the compilation unit.
    - ♦ If the metaprogram's anchor is a member of a type declaration, it is given *Insert* permission to the type declaration.
    - ♦ If the metaprogram's anchor is a member of a block of statements, it is given *Insert* permission to the block.
- If the metaprogram declares `mutate` local mode, then
    - ♦ If the metaprogram's anchor is in the top level of a compilation unit, it is given *Mutate* permission to the compilation unit.
    - ♦ If the metaprogram's anchor is a member of a type declaration, it is given *Mutate* permission to the type declaration.
    - ♦ If the metaprogram's anchor is a member of a block of statements, it is given *Mutate* permission to the block.
- If the metaprogram declares `fullMutate` local mode, then it is given *Mutate* permission to the enclosing compilation unit.
- If the metaprogram declares `packageInsert` package mode, then it is given *Insert* permission to the root package.
- If a metaprogram has *Read* permission for a node, it has *Read* permission for all children of that node.
- If a metaprogram has *Insert* permission for a node, it has *Insert* permission for all children of that node which do not represent compilation units.
- If a metaprogram has *Mutate* permission for a node, it has *Mutate* permission for all children of that node which do not represent compilation units.
- If a metaprogram has *Mutate* permission for a node, it has *Insert* permission for that node.
- If a metaprogram has *Insert* permission for a node, it has *Read* permission for that node.

Metaprograms always have *Mutate* permission to AST fragments which are not connected to the root package.

## 3.4 Evaluation Time

***Draft comment:*** *TODO*

# Chapter 4

# Dependencies

Due to the fact that they are operating on the same data structure, the order of metaprogram execution may have significant impact on the semantics of the resulting object program. As a result, metaprograms are capable of specifying their dependencies upon other metaprograms. A BSJ compiler provides support for ensuring that metaprograms are run in the correct order, detecting cycles in the metaprogram dependency graph, and detecting some occasions in which the expressed dependencies are incorrect (typically in the form of detecting a dependency which was not listed).

Metaprograms have (among their various metadata) unique identities, targets, and dependencies. The unique identities of metaprograms are implementation-specific and not user-specified. A target is a set of metaprograms; a metaprogram target declaration indicates that the metaprogram is part of the specified target. A dependency indicates that the metaprogram must be executed after all metaprograms in the target on which it depends. If a metaprogram does not declare a target, it is part of its own unique target on which no metaprogram can depend.

For example, consider the following sequence of metaprograms:

```
[: #target X; :]              // metaprogram A
[: #target X; :]              // metaprogram B
[: #target X, Y; :]           // metaprogram C
[: #target Z; #depends X; :]  // metaprogram D
[: #target Z; #depends Y; :]  // metaprogram E
[: #depends Z; :]             // metaprogram F
[: :]                         // metaprogram G
```

The names given to each of these metaprograms in the comments are arbitrary and solely for the purposes of the following discussion; the BSJ compiler is under no obligation to use those names or generate anything similar to them. These metaprograms form the directed dependency graph shown in Figure 4.1.

For example, the graph shows a direct path from metaprogram D to metaprogram A; this indicates that metaprogram A must execute before metaprogram D. More indirectly, metaprogram F is required to execute before metaprogram C because a path exists from F to C. Note that no path containing metaprogram G contains any other metaprograms; thus, metaprogram G may execute in any order with respect to the other metaprograms.

If the preambles of metaprograms in the program being compiled construct a dependency graph which contains a cycle, a compile-time error occurs.

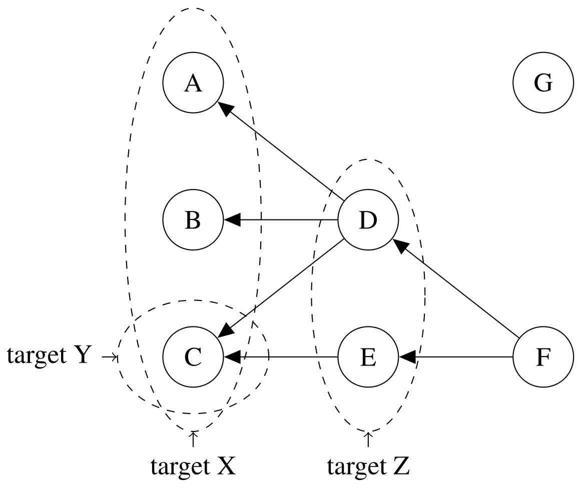

Figure 4.1: Dependency Graph Example

## 4.1 Conflicts

The purpose of metaprogram dependencies is to allow metaprograms to be executed in an explicit order when necessary as execution order can clearly affect the semantics of a metaprogram. If, for instance, metaprogram A adds a field to a class declaration and metaprogram B iterates over the fields of that class declaration to generate an implementation of `toString()`, whether metaprogram A executes before or after metaprogram B has an impact on the `toString()` method that is generated.

BSJ provides the dependency system not only to permit control over metaprogram execution order but also to require that metaprogram execution order is controlled. When two metaprograms may potentially operate in a conflicting manner, their execution order must be explicitly specified or a compile-time error occurs. In this way, the order in which the BSJ compiler chooses to execute the metaprograms does not affect the object program which is generated; either the object program is generated (in which case only one object program could be generated for the given dependency graph and metaprogram runtime input) or no object program is generated (due to the compile-time error). The latter case is said to represent a *conflict* of at least two metaprograms.

In order to define conflicts precisely, we require definitions of some supporting terms. Firstly, each node in an AST has a set of *attributes*. For most nodes, there is a one-to-one correspondence between attributes and properties. (For a complete discussion of attributes, see §4.2.) Each time an attribute on an AST is accessed, it is done so using one of three *access types*: *Read*, *WeakWrite*, or *StrongWrite*. Each access of an attribute creates an access record which contains the access type, the attribute which was accessed, the node on which the attribute is found, and the identity of the metaprogram performing the access.

Two metaprograms are said to *cooperate* if there is an explicit dependency order between them. More formally, two metaprograms cooperate if and only if there exists some path on the metaprogram dependency graph which contains both metaprograms. Cooperation of metaprograms ensures that they can be executed in only one order, preventing any actions they take from conflicting with each other.

A conflict is then defined as any pair of accesses for which the following are all true:

- Both accesses refer to the same attribute and the same node.
- The metaprograms of each access do not cooperate with each other.
- At least one of the following is true:
    - ♦ One of the accesses is a *Read* and the other is a *WeakWrite*.
    - ♦ One of the accesses is a *Read* and the other is a *StrongWrite*.
    - ♦ Both of the accesses are *StrongWrite*s.

## 4.2 Attributes

Most nodes have exactly one attribute per property. For instance, a `BinaryExpressionNode` has three attributes: one for the left operand, one for the right operand, and one for the binary operator applied to the operands. The only exceptions to this rule are the descendents of the `ListNode` class and the `PackageNode` class.

***Draft comment:*** *Discussion of what permissions are necessary to modify attributes. (Goes here?)*

### 4.2.1 ListNode

***Draft comment:*** *More accurate approximations of Church-Rosser over this list exist, the strongest of which would be a full precondition/postcondition analysis of the list operations in a metaprogram. A somewhat more powerful (and more practical) version might have an infinite number of attributes - one for each element value - and track presence in the list that way. This complicates things quite a lot, though, so we're going for the simple cell-based approach for now.*

The number of attributes contained in a `ListNode` is proportional to the number of elements in the list. Each element in a `ListNode` is associated with three attributes: a *Between* attribute which appears before it, a *Between* attribute that appears after it, and a *Present* attribute which specifically corresponds to that element. The *Between* attribute immediately following an element is the same as the *Between* attribute that immediately preceeds the following element. In addition to the element-specific attributes, each list also contains a *Size* attribute.

The *Present* attributes correspond to the list's data cells rather than to the element directly. When a value is removed, the attributes corresponding to its cell are also removed. When an element is added, a new cell is created and its attributes are correspondingly added. As a result, an element may be removed from the list by one metaprogram and introduced into the list in a different position by another metaprogram without fear of conflict.

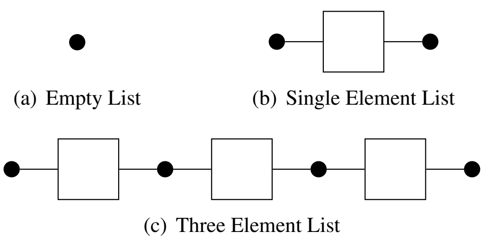

Figure 4.2: List Attribute Example. Boxes represent list elements and the corresponding *Present* attributes; dots represent *Between* attributes.

### Order Dependence

The order of nodes in an AST's list may or may not have relevance to the semantic meaning of a program. For instance, the order of block statements in a block is critical to the semantic meaning of a program. The order of method declarations in a type declaration's body has no impact; they are stored in a list simply to allow metaprograms to have a more intuitive understanding of their environment. As a result of this distinction, list elements are separated into two varieties: order-dependent and order-independent. Whether an element is order-dependent or not depends both on the type of node and the type of the list. The following cases are order-dependent:

- Any element contained in a list of the following types:
  - ♦ `AnnotationElementListNode`
  - ♦ `AnnotationValueListNode`
  - ♦ `BlockStatementListNode`
  - ♦ `CaseListNode`
  - ♦ `CatchListNode`
  - ♦ `DeclaredTypeListNode`
  - ♦ `EnumConstantDeclarationListNode`
  - ♦ `ExpressionListNode`
  - ♦ `ReferenceTypeListNode`
  - ♦ `StatementExpressionListNode`
  - ♦ `TypeArgumentListNode`
  - ♦ `TypeParameterListNode`
  - ♦ `VariableDeclaratorListNode`
  - ♦ `VariableInitializerListNode`
  - ♦ `VariableListNode`
- Any element which is an `InitializerDeclarationNode`.

Order dependence is used to control what type of access is logged to the attributes of elements when they are retrieved, added, or removed.

**List Operations**

The Java `List` interface provides a number of different ways in which a list can be manipulated. The list contained in a `ListNode` can be called in any of these ways but all of these calls must log access information. The intention of this access logging is to ensure that metaprograms which conflict are properly detected. The algorithm described in this section is a conservative approximation of that property and is required to be used by BSJ compilers.

For purposes of discussion, the access type *ElementWrite* will be used. If the relevant element is order-dependent, an *ElementWrite* is equivalent to a *StrongWrite*. If the relevant element is order-independent, an *ElementWrite* is equivalent to a *WeakWrite*. The access information which must be logged for each `List` interface method is explained below:

- For each element added to the list via `add(T)`, `add(int, T)`, `addAll(Collection<? extends T>)`, or `addAll(int, Collection<? extends T)`, a *StrongWrite* is performed for the element's *Present* attribute and an *ElementWrite* is performed for the *Between* attributes before and after that element. A *WeakWrite* is also logged to the *Size* attribute.
- For each element removed from the list via `remove(int)`, a *StrongWrite* is performed for the element's *Present* attribute and an *ElementWrite* is performed for the *Between* attributes before and after that element. A *WeakWrite* is also logged to the *Size* attribute.
- For each element removed from the list via `remove(Object)`, `removeAll(Collection<?>)`, `retainAll(Collection<?>)`, or `clear()`, a *StrongWrite* is performed for the element's *Present* attribute and a *WeakWrite* is performed to the *Size* attribute. No access is logged for the *Between* attributes before or after the element.
- For each element retrieved from the list via `get(int)`, a *Read* is performed for the element's *Present* attribute, the *Between* attrributes before and after the element, and the *Size* attribute of the list.
- For each object in calls to `contains(Object)` or `containsAll(Collection<?>)`,
    - ♦ If the object in question is found in the list, the corresponding *Present* attribute is accessed via *Read*.
    - ♦ If the object in question is not found in the list, every *Between* attribute in the list is accessed via *Read*.
- For each call to `toArray()` or `toArray(Object[])`, every attribute in the list is accessed via *Read*.
- For each call to `set(int,T)`, a *StrongWrite* is performed on the *Present* attribute for that element. If either the new element or the old element is order-dependent, a *StrongWrite* is performed on the *Between* attributes before and after that position; otherwise, a *WeakWrite* is performed on the *Between* attributes before and after that position. No access is recorded for the *Size* attribute.
- For each call to `isEmpty()` or `size()`, a *Read* is performed on the *Size* attribute.
- For each call to `indexOf(Object)`,

  - If the object in question is found in the list, the *Present* attribute for that element is accessed via *Read*. Additionally, every *Between* and *Present* attribute which appears before it in the list is accessed via *Read*. The *Size* attribute is also accessed via *Read*.
  - If the object in question is not found in the list, every attribute in the list is accessed via *Read*.

- For each call to `lastIndexOf(Object)`,
  - If the object in question is found in the list, the *Present* attribute for that element is accessed via *Read*. Additionally, every *Between* and *Present* attribute which appears after it in the list is accessed via *Read*. The *Size* attribute is also accessed via *Read*.
  - If the object in question is not found in the list, every attribute in the list is accessed via *Read*.

- When any of `iterator()`, `listIterator()`, or `listIterator(index)` is called, it must return an iterator that behaves in the following manner:
  - For each element added to the list via `add(E)`, a *StrongWrite* is performed for the element's *Present* attribute and an *ElementWrite* is performed for the *Between* attributes before and after that element. A *WeakWrite* is also logged to the *Size* attribute.
  - For each element removed from the list via `remove()`, a *StrongWrite* is performed for the element's *Present* attribute and an *ElementWrite* is performed for the *Between* attributes before and after that element. A *WeakWrite* is also logged to the *Size* attribute.
  - For each call to `set(E)`, the corresponding *Present* attribute is accessed using a *StrongWrite*. No *Between* attributes nor the *Size* attribute are affected.
  - For each call to `nextIndex()` or `previousIndex()`, the *Size* attribute is accessed using a *Read*.
  - For each calll to `hasNext()` or `hasPrevious()`, the *Between* attribute after or before (respectively) the cursor is accessed via *Read*.
  - For each call to `next()` or `previous()`, the *Between* attribute after or before (respectively) the cursor is accessed via *Read*. If the call returns an element, the element's *Present* attribute is accessed via *Read*.

- All calls to a `ListNode`'s list via a proxy obtained from `subList(int,int)` must behave in the same fashion as if the equivalent call were made to the original list directly.

For example, consider Figure 4.3, which represents the following sequence of accesses to an initially empty list:

1. Add child **A** to the list.
2. Add child **B** to the end of the list.
3. Remove child **A** from the list.
4. Add child **A** to the end of the list.

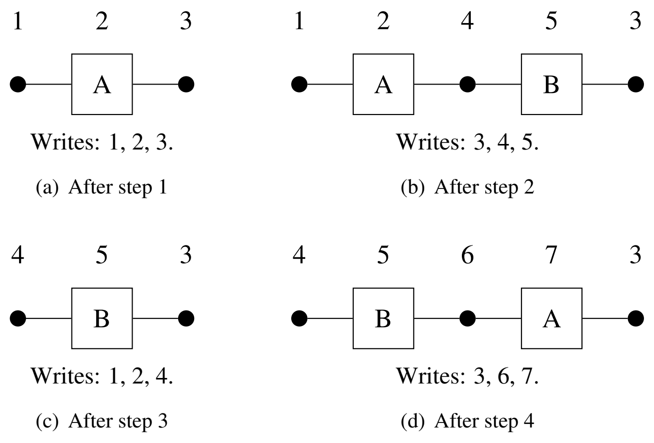

Figure 4.3: Attribute Access Example. UIDs for the attributes are indicated above the attribute symbol. Text below the diagrams indicates which attributes have been written during that step.

Observe that, although the child node A was added back to the list in step 4, the attributes associated with it are new. If the fourth step were performed by a different metaprogram, a conflict would occur because both metaprograms performed write operations on the list node's attribute with UID 3.

It should be noted that Figure 4.3 is an example of merely one implementation of this mechanism. Another implementation might keep the attribute with UID 1 at the beginning of the list at all times and add a new *Between* attribute after elements rather than before them when new elements are introduced.

***Draft comment:*** *Are these two implementations equivalent?*

### 4.2.2 PackageNode

Nodes of type `PackageNode` also have numerous attributes. Package nodes only permit the addition of children; children cannot be removed from them. Children are of one of two types: compilation units or immediate subpackages. Children are keyed by their names. Therefore, one attribute exists for each possible name. Each addition of a child causes a write operation to the attribute corresponding to the child's name; each access (through iteration or through direct request) causes a read operation to the same attribute.

***Draft comment:*** *Is this sufficient? What about when iteration is completed? What about when an element which does not exist is requested?*

# Chapter 5

# Directives

***Draft comment:*** *TODO*

# Chapter 6

# Compilation

***Draft comment:*** *TODO*

## 6.1 Semantics of BSJ Files

***Draft comment:*** *TODO*

## 6.2 Execution of Metaprograms

***Draft comment:*** *TODO*

Chapter 7

# Code Literals

***Draft comment:*** *TODO*

## 7.1 Reserve Types

***Draft comment:*** *TODO*

# Chapter 8

# Syntax

***Draft comment:*** *TODO*

***Draft comment:*** *Also need a references section*