MACROS THAT PLAY: MIGRATING FROM JAVA TO MAYA

by

Jason Baker

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2001

Copyright © Jason Baker 2001

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Jason Baker

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

______________ ____________________________

Chair: Wilson C. Hsieh

______________ ____________________________

Matthew Flatt

______________ ____________________________

Jay Lepreau

______________ ____________________________

Gary Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Jason Baker in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Wilson C. Hsieh
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Maya is a version of Java that allows users to write their own syntax extensions, which are called Mayans. Mayans can reinterpret or extend Maya syntax by expanding it to other Maya syntax: they operate on abstract syntax trees, and their expansion is triggered during parsing as semantic actions. Maya's expressiveness comes from treating grammar productions as generic functions, and Mayans as multimethods on those generic functions. Mayans are defined using a rich set of parameter specializers: Mayans can be dispatched on type of an AST node, the static type of an expression, the value of a token, or the substructure of any AST node. Multiple dispatch allows users to extend the semantics of the language by overriding the language's base actions.

This thesis explores explores the design and implementation space for compile-time meta-programming systems through Maya. Maya uses multiple-dispatch to provide an expressive meta-programming environment. Maya also uses novel implementation techniques to statically detect certain errors in code that builds syntax trees. We use MultiJava (described at OOPSLA 2000) as an example of the syntax and semantics extensions that can be expressed in Maya.

For my mother.

CONTENTS

ABSTRACT . . . . . **iv**

LIST OF TABLES . . . . . **ix**

LIST OF FIGURES . . . . . **x**

ACKNOWLEDGMENTS . . . . . **xi**

CHAPTERS

1. INTRODUCTION **1**

2. TUTORIAL **5**

- 2.1 Simple Mayans, or String Concatenation 7
- 2.2 Hygiene, or String Coercion 9
- 2.3 Structural Reflection, or More String Coercion 11
- 2.4 Case Statements, or Optimizing String Expressions 14
- 2.5 More Hygiene, or Generating Run Methods 17

3. LANGUAGE DESIGN **20**

- 3.1 Overview 20
- 3.2 Abstract Mayans 22
- 3.3 Concrete Mayans 26
  - 3.3.1 Argument Trees 26
  - 3.3.2 Defining and Importing Mayans 28
  - 3.3.3 Dispatch Rules 30
- 3.4 Templates and Unquotes 32
  - 3.4.1 Templates 32
  - 3.4.2 Unquoting 33
  - 3.4.3 Dynamic Restrictions 33
  - 3.4.4 Calling the Next Mayan 35
- 3.5 Case Statements 36
- 3.6 Hygiene 36
  - 3.6.1 Explicit Name Management 37
  - 3.6.2 Mayans 38
  - 3.6.3 Grammar Support 39
- 3.7 API 41
  - 3.7.1 Node Creation 41
  - 3.7.2 Class States 43

3.8 Java Issues . . . 44
3.8.1 Keywords and Reserved Words . . . 44
3.8.2 Qualified Name Handling . . . 45

4. IMPLEMENTATION . . . 47

4.1 Data Structures . . . 47
4.1.1 Nodes . . . 48
4.1.2 RightSymbols . . . 50
4.1.3 Actions . . . 50
4.1.4 Productions . . . 51
4.2 Generating Parse Tables . . . 52
4.2.1 Token Categories . . . 52
4.2.2 Start Symbols . . . 53
4.2.3 Conflict Resolution . . . 55
4.3 Pattern Parsing . . . 56
4.3.1 The Parsing Algorithm . . . 56
4.3.2 Proof of Correctness . . . 57
4.3.3 Dispatch and Hooks . . . 59

5. EVALUATION . . . 61

5.1 Macros . . . 61
5.1.1 Arrays from collections . . . 62
5.1.2 Assert and debug . . . 62
5.1.3 Iteration . . . 63
5.1.4 Formatted output . . . 65
5.2 MultiJava Overview . . . 66
5.2.1 Multimethods . . . 67
5.2.2 Open Classes . . . 68
5.3 Implementing Multimethods . . . 68
5.3.1 Interpreting Concrete Syntax . . . 69
5.3.2 Generic Functions and Type Checking . . . 70
5.4 Implementing Open Classes . . . 72
5.4.1 Using Open Classes in Maya . . . 73
5.4.2 Weaving . . . 74
5.4.3 Enforcing MultiJava Semantics . . . 75
5.5 Discussion . . . 76

6. RELATED WORK . . . 80

6.1 Dispatch and Pattern Matching . . . 80
6.2 Macro Systems . . . 81
6.2.1 `syntax-case` . . . 81
6.2.2 Dylan . . . 82
6.2.3 XL . . . 83
6.2.4 MS^2 . . . 83
6.3 Compiler Toolkits . . . 84
6.3.1 A* . . . 84

6.3.2 JTS . . . 85
6.3.3 Micros . . . 85
6.4 Compile-time MOPs . . . 86
6.4.1 Overview . . . 86
6.4.2 User-Defined Syntax . . . 87
6.4.3 Dispatch . . . 87
6.4.4 Expansion Order . . . 87
6.4.5 Safety . . . 88

7. CONCLUSIONS . . . 89

7.1 Language Features . . . 89
7.2 Implementation Techniques . . . 90
7.3 Expressiveness . . . 90

APPENDICES

A. GRAMMAR . . . 92

B. LIBRARY CODE . . . 99

REFERENCES . . . 103

LIST OF TABLES

5.1 Size of utility macro code 62
5.2 Size of MultiJava code 77

LIST OF FIGURES

1.1 Compiling language extensions and extended programs 2
2.1 Maya's string concatenation operator 8
2.2 Mapping primitive to boxed types. 12
2.3 The dotted name node class hierarchy 14
2.4 Optimizing string coercion 16
2.5 Templates for run methods 17
3.1 Overview of Maya's internal architecture 21
3.2 Translation of Maya meta-grammar to standard BNF 23
3.3 The syntax for a block that takes parameters 23
3.4 A Mayan's formal parameters 26
3.5 A complex Mayan argument tree 28
3.6 Illegal code whose expansion depends on itself 29
3.7 Partial order of Mayans. 31
3.8 Sorting parameters that match integer addition 31
3.9 Syntactic sugar for unquoting. 34
3.10 The states of classes being compiled 43
4.1 The node class hierarchy 49
4.2 Pattern parsing example 57
5.1 `foreach` Mayan to walk iterators. 64
5.2 `FormatState` members involved in error checking 66
5.3 Interposing on every class declaration 74

ACKNOWLEDGMENTS

I thank Wilson Hsieh and the rest of my thesis committee for assisting in the design and description of Maya. I thank Eric Eide and Patrick Tullmann for help with writing and TEX. Finally, I thank Godmar Back, Eric Eide, Alastair Reid, and Timothy Stack for helping me prepare the oral defense.

CHAPTER 1

INTRODUCTION

Application-specific language extensions are useful for a wide range of programming tasks, from database queries to parsing. For example, embedded SQL extends its host language with database query syntax. Parser generators such as YACC define a language for declaring grammars, and facilities for defining semantic actions in a general-purpose programming language. Finally, `emacs` is a primitive dialect of Lisp extended with powerful text-editing features. In spite of its weaknesses, `emacs` has been used for a variety of applications, including news readers and web browsers.

Design patterns can be viewed as work-arounds for specialized features missing from general-purpose languages. For instance, the visitor pattern implements multiple-dispatch in a single-dispatch language. In response, some language designers have chosen to specialize their languages. For example, C# [22] includes explicit support for the state/observer pattern and delegation [15]. However, unless we are willing to wait for a new language each time a design pattern is identified, such an approach is unsatisfactory. Instead, a language should admit programmer-defined syntax extensions.

Macro systems [9, 24, 27] support a limited form of syntax extension. In most systems, a macro call consists of the macro name followed by zero or more arguments. Such systems do not allow macros to define infix operators. In addition, macros cannot change the meaning of the base language's syntax. Even in Scheme, which has no reserved words, a macro cannot redefine the procedure application syntax.

Many languages allow programmers to define new operators or overload built-in operators through other means. Languages such as ML allow new operators

to be defined in terms of functions. Languages such as C++ and Haskell allow operators to be overloaded through method definition. Since operator definition and overloading only deal with expressions, they can be supported without a meta-programming interface.

Recent systems allow more sophisticated forms of syntax rewriting than simple macro systems. These systems range from aspect-oriented languages [28] to compile-time meta-object protocols [6, 26]. Compile-time MOPs allow a meta-class to rewrite most syntax in the base language. However, these systems typically have limited facilities for defining new syntax.

This thesis describes an extensible version of Java called Maya. Figure 1.1 shows Maya's overall architecture. In Maya, macros, which are called Mayans, are written as code that generates abstract syntax trees. After macros have been compiled, they can be loaded into the compiler, and used to compile applications written in an extended version of Maya.

Maya's primary research contribution is to explore the design and implementation space for compile-time meta-programming systems. Maya uses multiple-dispatch to provide an expressive meta-programming environment. Maya also uses two novel implementation techniques, pattern parsing and static hygienic renaming,

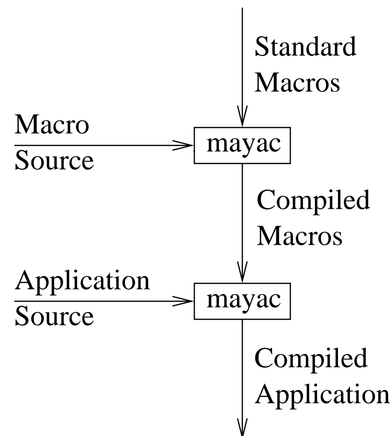

Figure 1.1. Compiling language extensions and extended programs

to statically detect certain errors in code that builds syntax trees.

A Mayan can reinterpret or extend Maya syntax by expanding it to other Maya syntax. Mayans are dispatched from the parser to generate abstract syntax trees. Maya borrows features such as multiple dispatch from advanced languages.

Maya's expressiveness comes from treating grammar productions as generic functions, and Mayans as multimethods on those generic functions. Generic functions in multiple-dispatch languages are a generalization of virtual functions in languages such as C++ and Java. A method on a generic function is selected based on the run-time types of all arguments, rather than just the receiver. Mayans can be defined using a rich set of parameter specializers: in addition to run-time argument types, Mayans can be dispatched on the static type of an expression, the value of a token, or the substructure of any AST node. Multiple dispatch allows users to extend the semantics of the language by overriding the language's base actions, just as generic functions in Dylan [24] allow the behavior of classes to change without subclassing.

In many ways, Maya is more powerful than compile-time meta-object protocols. In particular, a Mayan can add any production to the grammar, provided the resulting language can be recognized by Maya's parser. Also, Mayans are not tightly coupled with classes in the program being compiled. Mayan programmers may change the semantics of any syntactic form by defining multimethods on the corresponding generic function. We feel that Maya's modeling of grammar productions as generic functions is more elegant than metaclasses.

The implementation of Maya uses both *lazy type checking* and *lazy parsing*. That is, types and abstract syntax trees are computed on demand. Lazy type checking allows a Mayan to dispatch on the static types of some arguments, and create variable bindings that are visible to other arguments. The latter arguments must have their types checked lazily, after the bindings are created. Lazy parsing allows Mayans to be imported at any point in a program. Syntax that follows an imported Mayan must be parsed lazily, after the Mayan defines any new productions. The lazy evaluation of syntax trees is exposed explicitly to the Maya programmer, so

that it is possible to control its effects. We describe laziness in Section 3.2.

Maya allows programmers to describe macro expansion through the use of templates, a facility like backquote in Lisp [25]. Maya uses a novel parsing technique described in Section 4.3 to statically check the bodies of templates for syntactic correctness. This technique allows a template to build an arbitrary piece of syntax (including user-defined syntax). Other meta-programming systems, such as JTS [2], possess less general template mechanisms.

Maya supports hygiene and referential transparency through an unusual compile-time renaming step. Maya's implementation of hygiene detects most references to free variables when templates are compiled, but at the expense of features such as implicit parameters found in advanced macro systems.

Chapter 2 introduces Maya's features using examples from the standard library, which implements many Java operators as Mayans. Chapter 3 discusses Maya in detail, and provides formal descriptions of lazy parsing and Mayan dispatch. Chapter 4 describes Maya's implementation. In particular, this chapter describes the technique used to parse templates, and its correctness. Chapter 5 discusses two language extensions implemented in Maya: multimethods and open classes as defined by [8]. Chapter 6 compares Maya to related work, including high-level macro systems and open compilers. Finally, Chapter 7 summarizes our conclusions.

CHAPTER 2

TUTORIAL

This chapter describes the basics of how Maya is used: how Mayans are defined and imported, how hygienic Mayans can be written, and how Mayans are dispatched. Mayans rewrite abstract syntax trees: A Mayan takes one or more trees as arguments and returns a replacement tree.

The following Mayan implements debug logging:

```
Expression syntax Debug(debug(Expression val)) {
  return new Expression { System.err.println($val) };
}
```

A Mayan declaration looks much like a method declaration. A Mayan declaration contains modifiers, a return type, the keyword `syntax`, the Mayan's name, a formal parameter list, and a body. In the above example, the formal parameter list consists of the token `debug`, followed by an `Expression` named `val` in parentheses. The body returns an expression that prints the Mayan's argument.

Maya decouples Mayan definition from use: a Mayan such as `Debug` is not implicitly loaded into the compiler at the point of declaration, but must be loaded explicitly. This feature provides great flexibility, since both Mayan definitions and Mayan imports are lexically scoped. Although a local Mayan definition cannot be imported into its own scope, a Mayan can use any value in its environment.[1] Additionally, a Mayan can expose other Mayans to its arguments without resorting to higher-order templates that expand to Mayan declarations.

The `Debug` Mayan can be used in a method body as follows:

---

[1]Chez Scheme's `syntax-case` works in the opposite way.

```
void unexpected() {
  use Debug;
  debug("how did I get here");
  System.exit(-1);
}
```

The `use` directive extends the current lexical environment with a Mayan instance. Mayans can also be imported at the top level and on the command line. The latter allows a programmer to compile a file using different Mayan implementations.

Maya implements automatic hygiene and referential transparency for lexically scoped names. A hygienic macro system guarantees that variables declared in a macro body cannot capture references in a macro argument, while a referentially transparent macro system guarantees that variables local to a macro's call site cannot capture references in the macro's body. These properties eliminate a large class of macro implementation errors. Maya implements hygiene by assigning fresh names to local variable declarations generated by Mayans. If necessary, this behavior can be explicitly suppressed by the programmer.

The following example demonstrates both hygiene and referential transparency:

```
new Expression { ({ String once = $e; (once == null) ? "null" : once; }) }
```

Hygiene ensures that the local variable `once` cannot be referenced in `e`, while referential transparency ensures that `String` refers to the class `java.lang.String`.

This Mayan uses Maya's begin expression syntax, which should be familiar to `gcc` users. A begin expression is delimited by a set of braces inside a set of parentheses. The expression's body is a list of statements followed by an expression. The value of the begin expression is the value of the last expression it contains, in this case the conditional on `once == null`.

Not all names in Maya are hygienic. Since member names are not lexically scoped, they are not subject to hygienic renaming. Since top-level class names are visible throughout the program, hygiene never hides them. However, at times it is useful to define members whose names cannot appear in the source program. In these situations, a fresh name can be explicitly generated with `Environment.-makeId()`.

Mayans are executed inside the parser according to multiple-dispatch rules. Each production in the Maya grammar is called an *abstract Mayan*. Each semantic action in the grammar is called a *concrete Mayan*, or simply a Mayan. In multiple-dispatch terminology, an abstract Mayan is a generic function, and a concrete Mayan is a method. Note that abstract Mayans actually define concrete syntax, and are abstract in the sense that methods may be abstract: they are not associated with code.

Several concrete Mayans may be defined on a single abstract Mayan. Each time the compiler reduces an abstract Mayan, it selects and executes the most applicable concrete Mayan. Mayans are dispatched based on the expansion-time types of their arguments, as well as secondary specializers. Arguments are subtypes of `Node`, which is the class that corresponds to an abstract syntax tree node. Section 3.3 describes Maya's dispatching rules in detail.

2.1 Simple Mayans, or String Concatenation

The code in Figure 2.1 is part of Maya's standard library. These Mayans implement the string concatenation operator described in [17, §15.18.1] and provide a good overview of Maya.

The abstract Mayan on line 2 declares the binary '`+`' production. Neither an abstract Mayan nor its parameters are named. However, abstract Mayan declarations may include precedence and associativity values. An operator with higher precedence binds more tightly than operators with lower precedence. Precedence constants such as `ADD` are defined in `maya.tree.Precedence`.

String concatenation is implemented by three method-local Mayans: `CatSA` on line 12, `CatAS` on line 15, and `CatSS` on line 18. To use the string concatenation operator, one would write `use StringConcat;` at any point where a class declaration is allowed. The argument to `use` is a class that implements `MetaProgram`. A Mayan declaration defines a class that implements `MetaProgram`. An instance of this class is allocated, and its run method is called to update the environment. `StringConcat.run()` instantiates the local Mayans that implement

```
1  // The following line may be omitted, since binary + is built-in to Maya:
2  abstract Expression syntax(Expression + Expression) with precedence = ADD;
3
4  public class StringConcat implements MetaProgram {
5    Expression cat(Expression s1, Expression s2) {
6      return new Expression {
7        String.coerce($(s1)).concat(String.coerce($(s2)))
8      };
9    }
10
11   public Environment run(Environment env) {
12     Expression syntax CatSA(Expression:String s1 + Expression s2) {
13       return cat(s1, s2);
14     }
15     Expression syntax CatAS(Expression s1 + Expression:String s2) {
16       return cat(s1, s2);
17     }
18     Expression syntax CatSS(Expression:String s1 + Expression:String s2) {
19       return cat(s1, s2);
20     }
21
22     env = new CatSA().run(env);
23     env = new CatAS().run(env);
24     return new CatSS().run(env);
25   }
26 }
```

Figure 2.1. Maya's string concatenation operator

string concatenation. These local Mayans have full access to the enclosing instance of `StringConcat`.

The local Mayans in `StringConcat` take parameters with static type specializers. In `CatSA`, `s1` has the static type `String`: this static type ensures that `CatSA` only applies to '`+`' expressions with a `String`-valued expression on the left-hand side. Two other Mayans, `CatAS` and `CatSS`, handle other kinds of string concatenation expressions. Each expression must be unambiguously handled by exactly one Mayan. Consider three string concatenation expressions:

- `"the letter " + 'U'`: Since the second argument is not a `String`, only `CatSA` applies.

- `0 + " other possibilities"`: Since the first argument is not a `String`, only `CatAS` applies.

- `"jingle " + "ID"`: All three Mayans apply. However, `CatSS` is more specific than `CatAS` on the first argument, and more specific than `CatSA` on the second argument. Hence, `CatSS` is executed. If `CatSS` were not defined, neither `CatAS` nor `CatSA` would be more applicable than the other; the compiler would signal an error on such an ambiguous Mayan call.

Each of the three Mayans in Figure 2.1 computes its return value by calling `StringConcat.cat()`, which performs the real work of string concatenation. `StringConcat` is implemented in terms of another Mayan: The Mayan `String.coerce()` implements Java's string conversion rules. Among other things, these rules specify that '`(String) null`' should be converted to `"null"`.

`StringConcat.cat()` returns an expression built from the template pattern on line 6. A `Node` object can be allocated through either the standard `new` form, where the type name is followed by the constructor argument list, or a template, where the type name is followed by a template pattern in braces. When a template is expanded, it constructs the syntax tree corresponding to its pattern. `Node`-valued expressions may be embedded in a template using the '`$`' operator, which is analogous to the unquote operator, comma, in Lisp. The value of an unquoted expression is substituted into the template's tree. The method `StringConcat.cat()`'s behavior can be summarized with an example:

```
StringConcat.cat(new Expression { "the numeral " }, new Expression { 2 });
```

returns the expansion of

```
String.coerce("the numeral ").concat(String.coerce(2))
```

2.2 Hygiene, or String Coercion

The string coercion Mayans exploit Maya's hygiene and referential transparency rules in two ways. First, the concrete syntax for coercion uses referential transparency to prevent name clashes. Second, the Mayans that implement coercion rely on hygiene to prevent unwanted variable capture.

Some high level languages, such as Scheme and Dylan, provide fully hygienic and referentially transparent macros. Traditional Lisp allows for hygiene, but not referential transparency; a Lisp macro may generate fresh names for local variables using `gensym`. In contrast to these systems, Maya provides automatic hygiene and referential transparency only for names that are lexically scoped.

The string conversion Mayan is written with Maya's referential transparency rules in mind. Since `StringCoerce` must be defined for string concatenation to work, `StringCoerce` must not violate Java compatibility. In particular, it must not define a new keyword. Instead, `StringCoerce` reuses the static method call syntax. It rewrites calls to a unary method `coerce` on `java.lang.String`. The referential transparency of class names guarantees that a user-defined class p`.String` in some package p other than `java.lang` behaves normally.

Like `StringConcat`, `StringCoerce` is defined by a family of Mayans. The simplest coerces a string to a string:

```
Expression syntax
StringCoerceString(String\.coerce(Expression:String e)) {
  return new Expression {
    ({ String once = $e; (once == null) ? "null" : once; })
  };
}
```

The begin syntax allows us to store the result of `e` in a local variable. We must ensure that `e` is evaluated exactly once, because it may contain a method call or other subexpression with side effects. The local variable `once` is automatically renamed, which makes it invisible to the expression `e`.

A reference value should be converted to `"null"` when the value is the null pointer, when its `toString` method returns `"null"`, and when its `toString` method return the null pointer. Thus, coercion on reference types can be defined as:

```
Expression syntax
StringCoerceObject(String\.coerce(Expression:Object e)) {
  return new Expression {
    ({
      Object once = $e;
      once == null ? "null" : String.coerce(once.toString());
    })
  };
}
```

Note that this Mayan calls `StringCoerceString` in an environment containing a hygienic variable `once`. `StringCoerceString` also declares a hygienic variable `once`. Since each hygienic variable is given a fresh name, these declarations do not interfere with each other.

2.3 Structural Reflection, or More String Coercion

The Java language specification allows primitive types to be converted to strings in a number of ways, with various space/time tradeoffs. The method described here relies on our ability to convert primitive types to their corresponding box classes, as shown in Figure 2.2.

This code uses Maya's structural reflection API, which mirrors and extends the Java reflection API. Maya's structural reflection classes are defined in the `gnu.bytecode` package. `Type` is analogous to `java.lang.Class`; `Method`, `Constructor` and `Field` are analogous to the corresponding classes in `java.lang.reflect`; and `Access` is analogous to `java.lang.reflect.Modifier`.

There is no analogue of `Class`'s static method `forName()`. In Maya, all types are contained in a `TypeSpace`. In particular, they are contained in the `TypeSpace` found in the static field `MayanSpace.the`. `TypeSpace` supports several methods for looking up `Types`. `forSig()` searches for a type matching a signature, `forClass()` searches for a loaded type corresponding to a `Class` object, and `forName()` searches for a type matching a name.

`TypeSpace.forName()` is a bit more flexible than `Class.forName()`. Primitive types can be looked up by name, and array types can be located by name, or by their `Class.forName()` signature. For example.

```
s.forName("[Ljava.lang.Object;") == s.forName("java.lang.Object[]")
```

is `true` for any `TypeSpace` `s`.

The code in Figure 2.2 converts the name of a primitive type to the `Type` object of the corresponding boxed type. The method `primToBoxed()` can be used to convert a primitive value to a string through the static `toString` method in the corresponding box class:

```
import maya.tree.*;
import maya.grammar.*;
import gnu.bytecode.Type;
import java.util.Hashtable;

public class StringCoerce implements MetaProgram {
  static Hashtable boxes = new Hashtable();
  static {
    // populate boxes in the class initializer
    boxes.put("boolean", "Boolean");

    boxes.put("byte", "Byte");
    boxes.put("short", "Short");
    boxes.put("char", "Character");
    boxes.put("int", "Integer");
    boxes.put("long", "Long");

    boxes.put("float", "Float");
    boxes.put("double", "Double");

    boxes.put("void", "Void");
  }

  static Type primToBoxed(String prim)
  {
    String boxed = "java.lang.".concat((String) boxes.get(prim));
    try
      {
        return MayanSpace.the.forName(boxed);
      }
    catch (ClassNotFoundException _)
      {
        String err = "standard class ".concat(boxed).concat(" missing");
        throw new RuntimeException(err);
      }
  }
}
```

Figure 2.2. Mapping primitive to boxed types.

```
1  Expression syntax
2  StringCoerceLong(String\.coerce(Expression:long e)) {
3    Type t = StringCoerce.primToBoxed(e.getStaticType().getName());
4    StrictClassName n = (StrictClassName) StrictTypeName.make(t);
5    return new Expression { $(as StrictName n).toString($e) };
6  }
```

`StringCoerceLong` handles conversions of integer types, which are subtypes of `long`. Since the expression `String.coerce(1)` should expand to a call to the `toString()` method on `Integer` rather than `Long`, `StringCoerceLong` must do some work before returning a template.

The type of `e` is determined in line 3 of `StringCoerceLong`. The method `Expression.getStaticType()` returns the static semantic type of an expression. These `Type` objects support a `getName()` method analogous to that of `java.lang.Class`.

Building a method call expression from `e`'s type remains slightly problematic. In Maya, type names come in two flavors. `LazyTypeName` nodes arise out of class and member declaration parsing, and `StrictTypeName` nodes arise out of statement parsing. `LazyTypeNames` are resolved to types on demand, which allows Maya to handle various forms of mutual recursion, whereas `StrictTypeNames` are resolved to types immediately. The exact hierarchy of dotted name types is shown in Figure 2.3. In this figure, grammar symbols are circled, and interfaces are italicized. The node type `StrictClassName` is compatible with two grammar symbols: `StrictScalarType` and `StrictName`. Other `StrictName` node types denote references, array length expressions, and package prefixes. Since a template cannot implicitly coerce a `Type` object to either `StrictTypeName` or `LazyTypeName`, we must explicitly construct a `StrictTypeName`.

The call to `StrictTypeName.make()` on a class type returns a `StrictClassName`. Since method call syntax, which is described by the following abstract Mayans,

```
abstract MethodName syntax(StrictName \. Identifier);
abstract Expression syntax(MethodName (list(Expression, ‘,‘)))
  with precedence = PRIMARY;
```

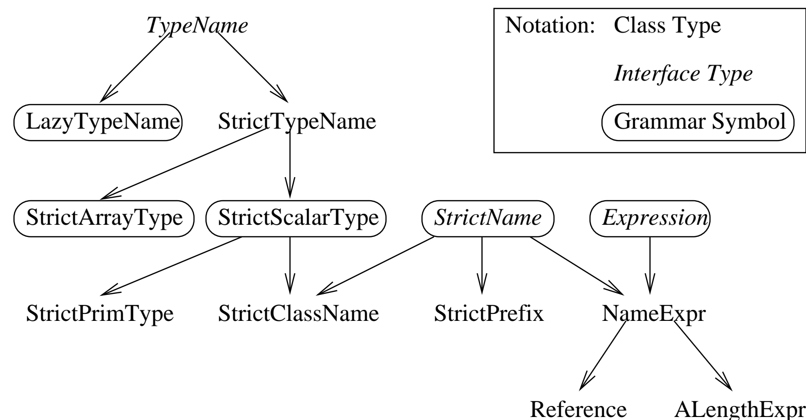

Figure 2.3. The dotted name node class hierarchy

expects a `StrictName`, we must coerce the `StrictClassName n`. We use the template conversion syntax `$(as` *grammar-symbol expr*`)`, but in this case a cast would work just as well.

Coercion on floating-point types is defined similar to that of integer types, but characters and booleans can be converted more quickly. Both are shown below:

```
Expression syntax
StringCoerceChar(String\.coerce(Expression:char e)) {
  return new Expression { new String(new char[] { $e }) };
}

Expression syntax
StringCoerceBool(String\.coerce(Expression:boolean e)) {
  return new Expression { $e ? "true" : "false" };
}
```

The '`.`' token literals in the previous examples were escaped with backslash. At times, identifier and '`.`' literals must be escaped to avoid ambiguity with grammar symbol names. For instance, the '`.`' in the `MethodName` abstract Mayan must be escaped. However, other backslahes in these examples can be omitted.

2.4 Case Statements, or Optimizing String Expressions

Java requires that constant string concatenation expressions be evaluated at compile time. Whereas Java requires that binary '`+`' expressions be constant folded,

Maya goes a step further by constant folding all calls to `String.concat()`, as shown below:

```
Expression syntax Fold(Expression:String ec1.concat(Expression:String ec2)) {
  Expression e1 = ec1.constantFold();
  Expression e2 = ec2.constantFold();

  if (e1 instanceof StringLiteral && e2 instanceof StringLiteral) {
      String s1 = ((StringLiteral) e1).getValue();
      String s2 = ((StringLiteral) e2).getValue();
      return new StringLiteral(s1.concat(s2));
  }
  else
     return nextRewrite($(e1).concat($e2));
}
```

`Fold` does not use multiple dispatch to intercept calls on `String` literals. Instead, it is executed on all calls to `String.concat()`. `Fold` uses the method `Expression.constantFold()` to determine whether an expression is a compile-time constant.

If both the receiver and argument to `concat()` are constants, `Fold` evaluates the call at compile time. In this case, `Fold` converts each `String` literal to the corresponding `String` using `getValue`. Otherwise, `Fold` uses `nextRewrite` to create a method call node. The `nextRewrite` form, which can only appear in a Mayan body, expands its arguments according to the next most applicable Mayan. `Fold` passes constant-folded arguments to the next Mayan to avoid duplicated work.

Constant folding is just one optimization needed to produce reasonable code for the string concatenation operator. Java compilers perform other string optimizations based on a number of assumptions about the runtime library. Java compilers know the exact behavior of `toString()` methods for boxed types. Specifically, these methods do not return `null`, and calls to instance `toString()` methods are interchangeable with calls to static `toString()` methods with the corresponding primitive value. Additionally, the method `String.concat()` does not return null.

The string coercion code described in the previous sections does not evaluate most constant conversions at compile time. However, Maya can eliminate the conditionals generated by `StringCoerceBool` by standard constant-folding rules. The string coercion code also fails to optimize null pointer tests for certain strings.

An optimizing version of `StringCoerceString` is shown in Figure 2.4. This code uses Maya's `syntax case` statement. Expressions that are known not to be `null`, such as literals, calls to `String.concat()` and calls to `StringBuffer.-toString()` are not wrapped in conditionals. This optimization is critical to `Fold`. If `CatSS` wrapped literal arguments in conditionals, constant string concatenation expressions would be much harder to find.

The `syntax case` statement takes a `Node`-valued expression, in this example `e`, and matches it against a series of clauses. Each clause consists of the word `on`, followed by a Mayan argument list and a body. A `syntax case` statement may also contain a default clause. Maya compares the case value to each argument list in order. Maya will execute the body of the first matching clause, with formal parameters bound to the corresponding subtrees of the case value.

```
Expression syntax
StringCoerceString(String.coerce(Expression:String ec))
{
  Expression e = ec.constantFold();
  syntax case (e) {
    on (StringLiteral lit) {
      return lit;
    }
    on (Expression:String s1.concat(Expression s2)) {
      return e;
    }
    on (Expression:StringBuffer sb.toString()) {
      return e;
    }
    default {
      return new Expression {
        ({ String once = $(e); once == null ? "null" : once; })
       };
    }
  }
}
```

Figure 2.4. Optimizing string coercion

The `syntax case` statement greatly simplifies `StringCoerceString`'s implementation. `StringCoerceString` could find trivial coercions with Mayan dispatch alone. However, this would require additional Mayan calls, since `StringCoerceString`'s results depend on constant folding its argument.

Maya constant-folds the conversion of primitives to `Strings`. The conversion from `char` to `String` takes advantage of `syntax case`.

2.5 More Hygiene, or Generating Run Methods

To finish the string coercion example, we must again write a `run` method that runs several Mayans in turn. Maya can be used to automate this process. The methods in Figure 2.5 build a `run` method that, runs all `MetaPrograms` in the array `classNames`.

The methods in `RunMayansHelper` can be used to define a family of Mayans

```
class RunMayansHelper {
  static Expression runSubs(Node[] classNames, Expression env)
  {
    for (int i = 0; i < classNames.length; i++)
      {
        env  = new Expression {
          new $((StrictName) classNames[i])().run($env)
        };
      }
  }

  static Declaration generateRunMethod(final Node[] classNames)
  {
    return new Declaration {
      public Environment run(Environment env) {
        return $(runSubs(classNames, new Expression{ env }));
      }
    };
  }
}
```

Figure 2.5. Templates for run methods

to simplify Mayan loading, but they also demonstrate some key template features. Hygienic variables are lexically scoped: The parameter `env` is available inside the body of `run()`. Occurrences of `env` will be renamed in the template where `run()` appears, and in nested templates. The method `runSubs()` takes `env` as an explicit parameter, because `runSubs()` cannot generate references to a hygienic variable declared in another method.

Maya compiles lazy portions of templates, such as the `run()` method body in Figure 2.5, into thunks. These thunks are inner-class instances that are expanded when the corresponding syntax would be parsed. The standard Java rules for local classes apply to template thunks: all local variables that occur free in the thunk code must be declared final. In particular, `generateRunMethod()`'s parameter, `classNames`, must be final.

`RunMayansHelper` is used by several Mayans that abstract away Mayan construction and invocation. The first simply generates a `run()` method:

```
abstract Declaration syntax(runMayans{ list(StrictName, ‘,‘) });

public Declaration syntax RunMayans0(runMayans{ list(StrictName, ‘,‘) mayans })
{
  return RunMayansHelper.generateRunMethod(mayans.getChildren());
}
```

This example uses `list` grammar symbols. The symbol `list(StrictName, ‘,‘)` matches a list of zero or more qualified names delimited by the comma token. Maya reduces this grammar symbol to a `ListNode` and saves `StrictName` nodes from left to right in an array. This array is returned by `getChildren()`.

Just as we can abstract away a `run()` method declaration, we can abstract away an entire `MetaProgram` class declaration. The syntax below generates `MetaProgram` subclasses that do nothing but export other `MetaPrograms`; however, this syntax possesses limitations.

```
abstract Declaration syntax(list(Modifier) \mayanContainer(Identifier)
                            { list(StrictName, ‘,‘) });

public Declaration syntax
DefineMayanContainer(list(Modifier) mods \mayanContainer(Identifier name)
                     { list(StrictName, ‘,‘) subs }) {
  return new Declaration {
    $(mods) class $(name) implements MetaProgram {
      $(RunMayansHelper.generateRunMethod(subs.getChildren()))
    }
  };
}
```

The code in `RunMayansHelper` expects an array of `StrictClassNames`, since it creates constructor calls. However, `DefineMayanContainer` is used at the top-level, where types should be parsed to `LazyTypeName`. Since `DefineMayanContainer` ignores this guideline, it breaks mutual recursion: A class declared with `define-MayanContainer` cannot refer to types defined later in the same source file, nor can it refer to types that refer to it.

CHAPTER 3

LANGUAGE DESIGN

This chapter describes Maya's features in detail. It begins by providing an overview of how Maya compiles source code to Java class files. It then describes extensions to Java that allow Mayan programmers to control the compilation process. Next, it describes Maya's API. Finally, it discusses Java incompatibilities.

3.1 Overview

Translating Maya source code to a fully expanded syntax tree involves dispatching and running Mayans. Dispatch may require type checking, while expansion may change the types of subtrees. Furthermore, mutually recursive classes cannot be parsed and type checked in one pass, but Mayan dispatch requires that parsing and type-checking be interleaved. Maya satisfies these constraints by parsing and type checking lazily, i.e., computing syntax trees and their types on demand.

Figure 3.1 shows the major components of our Maya compiler, `mayac`. The *file parser* reads type declarations from source files. Our compiler then processes each type declaration in two stages. The *parser* is invoked at each step to incrementally refine a declaration's AST. The *class shaper* parses the class body and computes member types. To compute the shape of a class C, all super-types of C and the types of all members of C must be found. The *class compiler* parses member initializers, including method and constructor bodies. Similarly, to compile C, the shapes of all types used in expressions in C must be known.

Maya exposes a class declaration's position on the top row of Figure 3.1 to the programmer through `ClassType.getState()`. A declaration leaves the file parser in `ClassType.FOUND`, the class shaper in `ClassType.SHAPED`, and the class compiler in `ClassType.COMPILED`. The Mayan programmer can use the class pro-

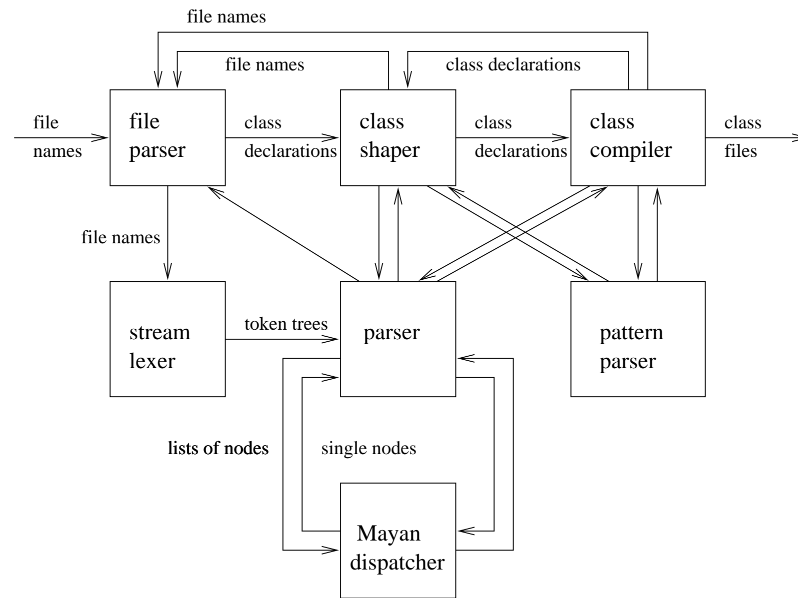

Figure 3.1. Overview of Maya's internal architecture

cessing model to deal with mutually recursive class declarations, as described in Section 3.7.2.

The *stream lexer* enables lazy parsing. It generates a tree of tokens rather than a flat stream. Specifically, the stream lexer creates a subtree for each set of matching delimiters: `(`, `)`, `[`, `]`, {, and }. These subtrees are called `ReaderTrees`, and they provide input to the parser. The stream lexer resembles a Lisp reader in that it builds trees from a simple context-free language.

Maya's lexer also recognizes several tokens that are not part of Java: `‘`, `@`, `#`, `$`, and \. These tokens can be used in abstract Mayans to define new operators.

Unless otherwise noted, all arcs coming into the *parser* are token trees, and all arcs going out are ASTs. The parser builds ASTs with the help of the *Mayan dispatcher*: on each reduction, the dispatcher executes the appropriate Mayan to build an AST node from its children. Mayan dispatch may involve recursive parsing.

The *pattern parser* is used when compiling Mayans and templates. It takes a stream of both terminal and nonterminal input symbols, and returns a partial parse

tree.

3.2 Abstract Mayans

Abstract Mayans define productions in a high-level meta-grammar that supports explicit laziness. Figure 3.2 shows how abstract Mayans are translated to BNF. Abstract Mayan arguments are either token literals, node types, matching-delimiter subtrees, or instances of the parameterized grammar symbols `list`, `nonEmptyList`, and `lazy`. Both subtrees and parameterized symbol instances are nonterminal. As Maya translates these symbols to BNF, it ensures that the appropriate rules are defined in the grammar. For instance, Maya translates every occurrence of `list(Modifier)` to the same grammar symbol, and ensures that this symbol along with its left-recursive helper is defined. Semantic actions on these implicitly defined rules produce useful AST nodes. It should be noted that declaration modifiers are not separated by '`,`' or any other token. Hence, the single-argument version of `list` is used.

The most subtle parameterized grammar symbol defers parsing of tokens in the current input stream. For example, the abstract Mayan:

```
abstract Expression syntax(let UnboundLocal = VarInitializer
                           in lazy({ end }, Expression) end)
  with precedence = ASSIGN - 1;
```

is translated to two BNF productions.

$$\begin{array}{rcl} \mathit{Expression} & \rightarrow & \texttt{let}\ \mathit{UnboundLocal}\ \texttt{=}\ G\ \texttt{in}\ \texttt{end} \\ G & \rightarrow & \epsilon \end{array}$$

A new symbol G will be created when the compiler first encounters '`lazy({ end }, Expression)`', and reused by other occurrences of the same symbol. The semantic action on G is called after the parser consumes the lookahead token '`in`', and builds a `ReaderTree` from tokens up to the first occurrence of `end`. The semantic action then constructs a `DelayedNode` that lazily parses the subtree to an `Expression`. We discuss the nonterminal `UnboundLocal` and the times when it must be used in Section 3.6.3.

The abstract Mayan in Figure 3.3 defines syntax for blocks that take parameters. It is translated to the set of productions below:

$\llbracket$`abstract` T `syntax` `(`$S_0\ S_1\ \ldots$`)`$\rrbracket$	$=$	$T \rightarrow \llbracket S_0\ S_1\ \ldots \rrbracket$				
$\llbracket$`(` $S_0\ S_1\ \ldots$ `)`$\rrbracket$	$=$	$\llbracket$`(` G `)`$\rrbracket$	where	G	$\rightarrow$	$\llbracket S_0\ S_1 \ldots \rrbracket$
$\llbracket$`(` S `)`$\rrbracket$	$=$	G	where	G	$\rightarrow$	$\mathit{ParenTree}\ ^*$
$\llbracket$`[` $S_0\ S_1\ \ldots$ `]`$\rrbracket$	$=$	$\llbracket$`[` G `]`$\rrbracket$	where	G	$\rightarrow$	$\llbracket S_0\ S_1 \ldots \rrbracket$
$\llbracket$`[` S `]`$\rrbracket$	$=$	G	where	G	$\rightarrow$	$\mathit{BracketTree}\ ^*$
$\llbracket$`{` $S_0\ S_1\ \ldots$ `}`$\rrbracket$	$=$	$\llbracket$`{` G `}`$\rrbracket$	where	G	$\rightarrow$	$\llbracket S_0\ S_1 \ldots \rrbracket$
$\llbracket$`{` S `}`$\rrbracket$	$=$	G	where	G	$\rightarrow$	$\mathit{BraceTree}\ ^*$
$\llbracket$`list(`S`)`$\rrbracket$	$=$	G_0	where	G_0	$\rightarrow$	$G_1\ ^{**}$
					$\mid$	ϵ
				G_1	$\rightarrow$	$G_1 \llbracket S \rrbracket \mid \llbracket S \rrbracket$
$\llbracket$`list(`S`, '`L`')`$\rrbracket$	$=$	G_0	where	G_0	$\rightarrow$	$G_1\ ^{**}$
					$\mid$	ϵ
				G_1	$\rightarrow$	$G_1\ L \llbracket S \rrbracket \mid \llbracket S \rrbracket$
$\llbracket$`nonEmptyList(`S`)`$\rrbracket$	$=$	G_0	where	G_0	$\rightarrow$	$G_1\ ^{**}$
				G_1	$\rightarrow$	$G_1 \llbracket S \rrbracket \mid \llbracket S \rrbracket$
$\llbracket$`nonEmptyList(`S`, '`L`')`$\rrbracket$	$=$	G_0	where	G_0	$\rightarrow$	$G_1\ ^{**}$
				G_1	$\rightarrow$	$G_1\ L \llbracket S \rrbracket \mid \llbracket S \rrbracket$
$\llbracket$`lazy(`$\mathit{ParenTree}$`, `T`)`$\rrbracket$	$=$	G	where	G	$\rightarrow$	$\mathit{ParenTree}\ ^\dagger$
$\llbracket$`lazy(`$\mathit{BracketTree}$`, `T`)`$\rrbracket$	$=$	G	where	G	$\rightarrow$	$\mathit{BracketTree}\ ^\dagger$
$\llbracket$`lazy(`$\mathit{BraceTree}$`, `T`)`$\rrbracket$	$=$	G	where	G	$\rightarrow$	$\mathit{BraceTree}\ ^\dagger$
$\llbracket L_0$ `lazy({` $L_1\ L_2\ \ldots$ `}, `T`)` $S_0\ \ldots \rrbracket$	$=$	$G\ L_0 \llbracket S_0\ \ldots \rrbracket$	where	G	$\rightarrow$	$\epsilon\ ^\ddagger$
$\llbracket S_0\ S_1\ \ldots \rrbracket$	$=$	$\llbracket S_0 \rrbracket \llbracket S_1\ \ldots \rrbracket$	if $S_1 \neq$ `lazy({` $L_0\ L_1\ \ldots$`}, `T`)`			
$\llbracket T \rrbracket$	$=$	T				
$\llbracket L \rrbracket$	$=$	L				

Symbols:

S	An arbitrary symbol.
T	A grammar symbol corresponding to a type.
L	A literal token.
G	A symbol generated by Maya.
ParenTree	The token category corresponding to the type `maya.tree.ParenTree`
BracketTree	The token category corresponding to the type `maya.tree.BracketTree`
BraceTree	The token category corresponding to the type `maya.tree.BraceTree`

Semantic actions:

- * The semantic actions for these productions recursively parse their arguments trees to S.
- ** The semantic actions for these productions collapse their subtrees, so that `getChildren` returns an array of all S descendents.
- † These productions construct `DelayedNodes` that parse the subtree to T on demand.
- ‡ This production is reduced after the lookahead token L_0 is consumed. It collects tokens into a lexer, stopping on the first token $L \in \{L_1, L_2, \ldots\}$. The production then constructs a `DelayedNode` that parses the lexer to T on demand.

Figure 3.2. Translation of Maya meta-grammar to standard BNF

```
abstract Statement syntax(MethodName(Formal) lazy(BraceTree, BlockStmts));
```

Figure 3.3. The syntax for a block that takes parameters

$$
\begin{array}{rcl}
\textit{Statement} & \rightarrow & \textit{MethodName}\ G_0\ G_1 \\
G_0 & \rightarrow & \textit{ParenTree} \\
G_1 & \rightarrow & \textit{BraceTree}
\end{array}
$$

Semantic actions on G_0 and G_1 produce AST nodes from unstructured subtrees. G_0's semantic action recursively parses its right-hand side to a *Formal*, while G_1's semantic action stores the *BraceTree* on its right-hand side in a `DelayedNode` for later parsing.

Lazy symbols allow Mayans to expose bindings to blocks, as in the following example:

```
Statement syntax EForEach(Expression:Enumeration enumExp . foreach(Formal var)
                          lazy(BraceTree, BlockStmts) body) {
  // ...
  return new Statement {
    for (/* ... */) {
      $(new DeclStmt(var));
      $(Reference.make(var)) = /* ... */;
      $body;
    }
  };
}
```

Since `EForEach` exposes the variable `var` to its body, `body` cannot parsed and type checked until `EForEach` has expanded and has bound `var`.

To define an abstract Mayan for `EForEach`, one must understand Maya's grammar and conflict resolution techniques. One might consider declaring the `foreach` syntax by stripping all parameter specializers from `EForEach`:

```
abstract Statement syntax(Expression \. foreach(Formal)
                          lazy(BraceTree, BlockStmts));
```

This approach does not lead to grammar conflicts, but produces unwanted behavior. Since the identifier `foreach` is embedded in the grammar, it acts as a reserved word to the right of a '`.`' and members with the same name cannot be accessed.

To avoid making `foreach` reserved, one could replace the literal `foreach` with `Identifier`, but this leads to a shift/reduce conflict. The abstract Mayan

```
abstract Statement syntax(Expression \. Identifier(Formal)
                          lazy(BraceTree, BlockStmts));
```

is translated to the following production:

$$\textit{Statement} \rightarrow \textit{Expression} \ . \ \textit{Identifier}\ G_0\ G_1$$

where G_0 and G_1 are defined as in the previous example. This production conflicts with `MethodName`, because a `MethodName` is also reduced with the lookahead `ParenTree`:

$$\begin{array}{rcl} \textit{Statement} & \rightarrow & \textit{Expression}\ ; \\ \textit{Expresssion} & \rightarrow & \textit{MethodName}\ G_2 \\ \textit{MethodName} & \rightarrow & \textit{Expression}\ .\ \textit{Identifier} \\ G_2 & \rightarrow & \textit{ParenTree} \end{array}$$

An abstract Mayan is valid if it does not introduce conflicts into the grammar. The Maya parser generator attempts to resolve conflicts with precedence rules. Unlike YACC, shift/reduce conflicts are not resolved in favor of shifts; nor are reduce/reduce conflicts resolved based on the lexical order of productions.

Unlike most parser generators, though, Maya can resolve certain reduce/reduce conflicts during parsing. When a reduce/reduce conflict involves productions with identical right-hand sides, the parser chooses between productions dynamically. If only one production has applicable semantic actions, that production will be chosen. Otherwise a parse error is signalled. The rules governing Mayan applicability are described in Section 3.3.3. Maya's recursive parsing techniques regularly lead to dynamically resolved conflicts.

The abstract Mayan in Figure 3.3 leads to just such a conflict: the parenthesized formal rule (G_0) conflicts with the argument-list rule (G_2). The parameterized block and method call syntax both take a `MethodName` followed by a `ParenTree` token. Maya resolves the conflict between G_0, which corresponds to `(Formal)`, and G_2, which corresponds to `(list(Expression, ‘,‘))`, dynamically. Maya recursively parses the `ParenTree` to the start symbol in the following grammar:

$$\begin{array}{rcl} S & \rightarrow & \textit{Formal} \\ & | & G_3 \\ G_3 & \rightarrow & G_4 \\ & | & \epsilon \\ G_4 & \rightarrow & \textit{Expression}\ ,\ G_4 \\ & | & \textit{Expression} \end{array}$$

Maya resolves the conflict between G_0 and G_2 based on the result of the parse. If the `ParenTree` is parsed to a `Formal` node, G_0 is reduced. Otherwise, the subtree

must be parsed to a `ListNode`, and G_2 is reduced.

3.3 Concrete Mayans

3.3.1 Argument Trees

A Mayan argument tree, like a formal parameter list, gives runtime arguments names. Argument trees serve two additional purposes. First, the grammar symbols corresponding to top-level arguments determine the abstract Mayan on which a concrete Mayan is defined. Second, an argument tree's shape and specializers control which reductions the concrete Mayan expands.

An argument tree is a partial parse tree, where named parameters are leaves, even though they may be nonterminal symbols. The pattern parser builds Mayan argument trees from unstructured lists of tokens and named parameters. For example, the concrete Mayan below is defined on the binary '+' abstract Mayan:

```
Expression syntax AM(Expression a+Expression b * Expression c)
{ ... }
```

The Mayan's arguments are parsed according to its return type, in this case `Expression`. Parsing `AM`'s arguments yields the tree in Figure 3.4. The '*' in this figure is circled, since it is not part of the AST and will not appear in `AM`'s third actual argument. Level-1 nodes in this tree correspond to the Mayan's actual arguments: AST nodes that `AM` must reduce to a single `Expression` value. The formal parameters `b` and `c` are children of the third argument, which is a node of type `MulExpr`. Maya infers the node types and layouts corresponding to interior

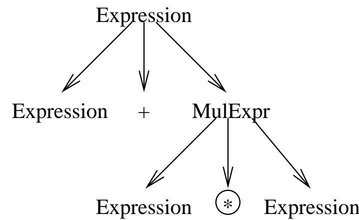

Figure 3.4. A Mayan's formal parameters

productions based on the built-in Mayans. As a result, user-defined syntax may only appear at the root of the tree.

For example, the following Mayan is legal

```
Statement syntax F1(Expression e.foreach(Formal f) {
                      for (ForHeader hd) Statement body
                    })
{ ... }
```

since `for` syntax is built-in to Maya, and the built-in Mayan is known to generate `ForStmt` nodes. However,

```
Statement syntax F2(Expression e1.foreach(Formal f1) {
                      Expression e2.foreach(Formal f2) {
                        lazy(BraceTree, BlockStmts) body
                      }
                    })
{ ... }
```

is illegal. The body of the outer `foreach` will be parsed and expanded before `F2` is executed. Because a Mayan may expand the inner `foreach` to an arbitrary statement, neither the exact type nor the layout of this node is known when compiling `F2`.

Maya usually infers a parameter's grammar symbol from its type. For instance,

```
Expression syntax AM2(Expression a + MulExpr b) { ... }
```

matches the same syntax as `AM`: binary '`+`' expressions where the right-hand side is a binary '`*`' expression. Maya infers that a `MulExpr` parameter is defined on an `Expression` symbol from the class hierarchy.

There are two cases when a grammar symbol cannot be inferred from a node type. First, parameterized grammar symbols are specified in place of the corresponding node types. Second, node types that inherit from multiple grammar symbols are ambiguous. A parameter specialized on an ambiguous type must mention its grammar symbol explicitly. The type specializer appears in angle brackets following the grammar symbol. For instance, the following Mayan matches a '`+`' expression whose right-hand side is a reference to an instance field:

```
Expression syntax AF(Expression a + Expression<InstanceRef> b) { ... }
```

A Mayan that that takes an `InstanceRef` argument must explicitly declare it as an `Expression` or `StrictName` symbol.

In addition to a node type, a Mayan parameter may contain a secondary attribute. This attribute may be either a vector of subparameters, a token value, a static expression type, or a class literal type. The Mayan below demonstrates these secondary attributes:

```
Expression syntax
CoerceInt(String.coerce(Expression:int e))
{ ... }
```

This Mayan overrides the static method call syntax. Its parse tree is shown in Figure 3.5. `CoerceInt`'s first parameter is a `MethodName` that includes subparameters. The first subparameter, `String`, matches all `StrictTypeName` nodes that refer to `java.lang.String`. The second subparameter matches the identifier `coerce`. The Mayan's second argument is a `ListNode` that contains expressions. This parameter also includes substructure: it matches lists of a single expression, `e`. The parameter `e` is specialized on the static type `int`.

3.3.2 Defining and Importing Mayans

Maya decouples Mayan definition from imports. A Mayan declaration defines a Mayan class. A Mayan is imported when an environment is extended with a Mayan instance. Mayans may be imported through the `use` directive, the '`-use`' command-line switch, or programmatically, through `Environment.run()`. This

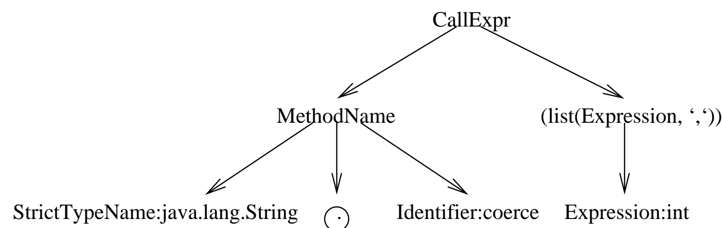

Figure 3.5. A complex Mayan argument tree

decoupling gives the Maya programmer flexibility in defining inner Mayans and defining several top-level Mayans in a source file. However, the increased flexibility imposes subtle restrictions.

Inner Mayans share state just as inner classes do; however, inner Mayans cannot be imported with `use`, since they are closed over their environment. Such Mayans can be imported with `Environment.run()`, which takes an instance rather than a class name, or with the `UseStmt` and `UseDecl` constructors described in Section 3.7.1.

Decoupling Mayan definition from import gives the programmer fine-grained control over implementation of large metaprograms: when a group of cooperating Mayans are defined in the same source file, the first Mayan is not implicitly imported to expand subsequent declarations. A Mayan can be imported in the same source file where it is defined, provided the Mayan is not imported into the scope of classes it refers to. This restriction prevents an inner Mayan from being imported into its enclosing class. It also disallows the circularity shown in Figure 3.6.

```
Expression syntax M(m(Expression x)) {
  return C.transform(x);
}

class C {
  static Expression transform(Expression x) {
    return D.transform(x);
}

use M;            // an error, even if an old version of D.class is available

class D {
  static Expression transform(Expression x) {
    return x;
  }
}
```

Figure 3.6. Illegal code whose expansion depends on itself

3.3.3 Dispatch Rules

Mayan dispatching is at the core of Maya. Each time an abstract Mayan is reduced, the parser dispatches to the appropriate concrete Mayan. This Mayan is selected by first finding all Mayans applicable to the production's right-hand side, then choosing the most applicable Mayan from this set. Although multiple-dispatch is present in many languages, such as Cecil [5], CLOS [25], and Dylan [24], the exact formulation of Maya's rules is fairly unique. Mayan dispatching is based on parameter types, but also considers secondary parameter attributes. Mayan dispatching is symmetric but supports an unusual form of lexicographic tie-breaking.

The predicate *mayanApplicable* in Figure 3.7 determines whether a Mayan is applicable to a particular right-hand side, and the relations in this figure determine the most applicable Mayan. In this figure, a Mayan parameter is modeled as the tuple $\langle t, \vec{S}, v, s, c \rangle$ where t is the node type the parameter matches, $\vec{S}$ is a vector of subparameters, v is a token value (or *null*), s is a static semantic type (or $\top$), and c is the type of a literal class name (or $\top$). The vector $\vec{S}$ corresponds to the parameter's children in the parse tree, while v and c are computed from a literal parameter's value. Although a class-literal parameter is applicable only to nodes that denote the same type, an expression parameter is applicable to any expression that is a subtype. A Mayan instance imported into an environment e is represented as the pair $\langle \vec{P}, i \rangle$ where $\vec{P}$ is a vector of parameters and i is an integer such that $\langle \vec{P}, i \rangle$ is the ith Mayan imported in e.

Mayan parameters are ordered based on the runtime types of nodes and secondary attributes. Secondary attributes are used to compare formal parameters with identical node types. Static expression types are compared using subtype relationships. This leads to orders such as that of Figure 3.8. Every parameter in this figure is more specific than those above it.

Dispatch in Mayan is symmetric. If two Mayans are more specific than each other on different arguments, neither is more specific overall, and an error is signaled. This approach avoids the surprising behavior described in [11], and is consistent with Java's static overloading semantics.

Domains:	
n	$\in$ *Node*, an AST node
t	$\in$ *Type*, a type
s, c	$\in$ *Type* types in the source program
v	a token value
i, j, m	integers
$\vec{N}$	a vector of AST nodes
$\vec{P}$, $\vec{S}$, $\vec{X}$, $\vec{Y}$	parameter vectors

Node Functions:	
getClass:	$Node \to Type$
getChildren:	$Node \to \overrightarrow{Node}$
getStaticType:	$Node \to Type$
toType:	$Node \to Type$

n_i, p_i, x_i y_i are elements of the corresponding vectors.

$$mayanApplicable(\langle \vec{P}, i\rangle, \vec{N}) \iff |\vec{P}| = |\vec{N}| \text{ and } \forall j \in [0, |\vec{P}|],\ paramApplicable(p_j, n_j)$$

$$\begin{aligned} paramApplicable(\langle t, \vec{S}, v, s, c\rangle, n) \iff\ & getClass(n) \leq: t, \text{ and} \\ & \forall j \in [0, m],\ paramApplicable(s_j, n_j), \text{ and} \\ & v \in \{null, n\}, \text{ and} \\ & getStaticType(n) \leq: s, \text{ and} \\ & toType(n) \in \{\top, c\}. \end{aligned}$$

Order of Mayans: $\langle \vec{P}_x, i_x\rangle < \langle \vec{P}_y, i_y\rangle \iff \vec{P}_x < \vec{P}_y$ or $(\vec{P}_x = \vec{P}_y$ and $i_x < i_y)$

Order of parameter vectors:

$$\vec{X} < \vec{Y} \iff |\vec{X}| = |\vec{Y}| \text{ and } \forall j \in [0, n], x_j \leq y_j \text{ and } \exists j \in [0, n] \text{ s.t. } x_j < y_j$$

Order of token values: $v < null \iff v \neq null$

Order of types: $t_1 <: t_2$ if and only if t_1 is a proper subtype of t_2

Order of Mayan parameters: $\langle t_x, \vec{S}_x, v_x, s_x, c_x\rangle < \langle t_y, \vec{S}_y, v_y, s_y, c_y\rangle \iff$
either $t_x <: t_y$,
or $((t_x = t_y$ and $\vec{S}_x \leq \vec{S}_y$ and $v_x \leq v_y$ and $s_x \leq: s_y$ and $c_x \leq: c_y)$
and $(\vec{S}_x < \vec{S}_y$ or $v_x < v_y$ or $s_x <: s_y$ or $c_x <: c_y))$

Figure 3.7. Partial order of Mayans.

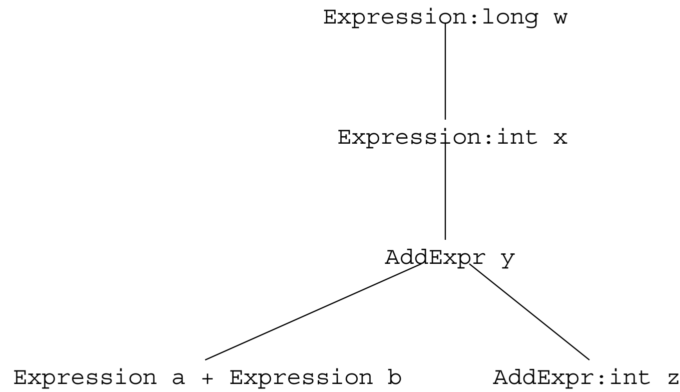

Figure 3.8. Sorting parameters that match integer addition

Lexical tie-breaking allows several Mayans to be imported with identical argument lists. Where several Mayans are equal according to the argument lists, the last Mayan imported with `use` is considered most applicable. Many of the Mayans described in Chapter 5 do not specialize their arguments. These Mayans are more applicable than the built-in Mayans only because the built-in Mayans are imported first.

3.4 Templates and Unquotes

Maya guarantees a template is syntactically correct by parsing its body at compile time. The pattern parser generates a parse tree from a sequence of tokens and expressions unquoted with '`$`'. Maya determines an unquoted expression's grammar symbol from either the expression's static type or an explicit coercion.

3.4.1 Templates

Maya templates are subject to many of the same compile-time checks as code, but some errors are only detected at expansion time. By parsing a template at compile time, Maya is able to detect syntax errors. Additionally, the hygiene rules described in Section 3.6 detect most references to unbound variables. However, full type checking is performed at expansion time.

The template parse tree is compiled into code that performs the same sequence of shifts and reductions the parser would have performed on the template body. Since Mayan dispatch occurs inside reductions, the template code generates a fully expanded syntax tree.

Templates honor laziness: subtemplates corresponding to lazy syntax are compiled into local thunk classes that are expanded when the corresponding syntax would be parsed. Template thunks are subject to the usual Java constraint on inner classes: a free reference to a local variable is allowed only if that variable is `final`. Mayan programmers can use template thunks in conjunction with the class processing model described in Section 3.7.2 to delay computation.

3.4.2 Unquoting

The following abstract Mayans define Maya's base unquote syntax:

```
abstract TemplateSymbol syntax ($ lazy(ParenTree, UnquoteExpr));
abstract UnquoteExpr    syntax (Expression);
abstract UnquoteExpr    syntax (as NtSpec Expression);
```

A template may contain '`$`' followed by an `UnquoteExpr` in parentheses. An `UnquoteExpr` is either an expression, or an '`as`' coercion. An '`as`' coercion consists of the word '`as`', followed by one of the grammar symbols defined in Figure 3.2 (on page 23) and the expression to coerce.

When a simple expression is unquoted, its grammar symbol is determined from its static type. In some cases, the grammar symbol cannot be inferred. For instance, a `ListNode` may match any `list` or `nonEmptyList` symbol, and a `Reference` may match `Expression` or `StrictName`. In these cases, the expression must be explicitly coerced to a grammar symbol with `as`.

Unquoted expressions are parsed lazily to avoid extra work: as template thunks are created, unquote expressions must be reinterpreted in the template class. By deferring their parsing, Maya avoids unnecessary work in computing closures.

The built-in Mayans on `UnquoteExpr` only accept `Node`-valued expressions, but Maya's standard library defines several implicit coercion rules such as `UnquoteString` in Figure 3.9. These rules convert primitive and `String`-valued expressions to the corresponding literal tokens.

Mayan's standard library defines convenient unquote syntax for statement expressions and dotted references, also shown in Figure 3.9. A dotted name may be unquoted without parentheses, and a begin expression may be unquoted as `$({ s; e; })`, rather than `$(({ s; e; }))`.

3.4.3 Dynamic Restrictions

Templates dynamically enforce a number of restrictions on unquoted nodes that prevent the type of an expression from changing during parsing. These restrictions prevent a number of subtle errors in Mayans, and ease the compiler's implementa-

```
abstract TemplateSymbol syntax($ StrictName);
abstract UnquoteExpr syntax(lazy(BraceTree, BeginBody));

UnquoteExpr syntax UnquoteString (Expression:String e) {
  // create a string literal with no source file or line number
  return new UnquoteExpr { (new StringLiteral($(e)) };
}
// ...
TemplateSymbol syntax UnquoteNoParen($ StrictName<NameExpr> r) {
  // expand $x.y to $(x.y)
  return new TemplateSymbol { \$($(r)) };
}
UnquoteExpr syntax UnquoteBegin(lazy(BraceTree, BeginBody) b) {
  // expand $({ s; e; }) to $(({ s; e; }))
  return new UnquoteExpr { ($(b)) };
}
```

Figure 3.9. Syntactic sugar for unquoting.

tion. Maya correctly compiles any program that passes these checks, regardless of node sharing.

Maya's dynamic restrictions ensure that certain bugs involving laziness will always lead to an expansion-time error, rather than occasionally being masked by shadowing. Suppose that `foreach`'s body could be an arbitrary statement rather than a block. A simple body would be parsed before `foreach` was expanded. This would often, but not always, lead to an error accessing an unbound variable:

```
1  class C {
2    int[] arr = ...;
3    double num;
4    int sum = 0;
5    // instance init block:
6    {
7       arr.foreach(int num)
8         sum += num;                // reference changes
9    }
10 }
```

In this example, `num` on line 8 is parsed as a reference to `C.num`, and Mayans defined on '`+=`' are expanded based on the reference's type. When `foreach` on line

7 is expanded, it shadows `C.num`. Maya detects this error when expanding `foreach`.

Mayans such as `foreach` place their arguments in new environments. At expansion time, Maya checks that unquoted nodes are valid in their new environment. A `DelayedNode` that has not yet been parsed is always valid. Maya recursively walks other nodes to ensure that:

- the type of `this` cannot change,
- variable and label references must refer to the same entity before and after expansion,
- type names must resolve to the same type before and after expansion, (`TypeName` nodes created by templates or explicitly with `StrictTypeName.make` or `LazyTypeName.make` resolve to the same type, regardless of their environments), and
- method and constructor calls must resolve to the same method before and after expansion.

3.4.4 Calling the Next Mayan

Maya's `nextRewrite` syntax calls the next applicable Mayan. If `nextRewrite` is called without arguments, the enclosing Mayan's parameters are passed unmodified. Otherwise, arguments to `nextRewrite` are expanded as a template, and the result is passed to the next Mayan.

The arguments to `nextRewrite` must generate the same production that the enclosing Mayan is defined on, and must produce the same dispatch order as the Mayan's actual arguments. While Maya statically checks that `nextRewrite` generates the appropriate production, an expansion-time error can occur if `nextRewrite` results in the wrong dispatch order.

3.5 Case Statements

The following three abstract Mayans define Maya's `case` statement:

```
abstract Statement syntax(syntax case (UnquoteExpr)
                          { nonEmptyList(CaseClause) });
abstract CaseClause syntax(on (list(BindingSymbol))
                           lazy(BraceTree, BlockStmts));
abstract CaseClause syntax(default lazy(BraceTree, BlockStmts));
```

Case statements allow Maya code to perform explicit pattern matching, in addition to the matching done within Mayan dispatch.

The case statement matches a `Node`-valued argument against one or more patterns. The argument expression may be coerced to a grammar symbol using `as`.

A case statement's `on` clause accepts the same kinds of parameters as a Mayan. However, case clauses are not subject to dispatch: the first clause that matches the case argument will be executed. Unlike `switch` statements, case statement clauses are not subject to fall-through: after a matching clause is executed, Maya executes the statement following the case.

A case statement may include a `default` clause. If the argument does not match any `on` clause, Maya will execute the `default` clause if it is present.

3.6 Hygiene

Hygiene and referential transparency maintain lexical scoping when a macro introduces code into its call site. These techniques were developed in Scheme, where every named entity is a lexically scoped variable. Since many names in Maya do not denote lexically scoped variables, hygiene and referential transparency is a bit more complex.

In Maya, local variables and labels are hygienically renamed. Within a template, local variable declarations and labels are given fresh names, and references to these entities are suitably renamed. Member and top-level type names are not lexically scoped; instead, their visibility is controlled by access modifiers. Therefore, such declarations are not automatically renamed.

Although fully qualified top-level type names are not lexically scoped, simple type names introduced by `import` and type declarations are. Similarly, a field

name is not lexically scoped, but implicit `this` is. Maya ensures that class and static field references in templates are expanded transparently. For instance, the following code binds e to a `Node` that denotes a reference to the static field `p.C.i`:

```
package p;

class C {
  static int i = 0;
  static Expression e = new Expression { i };
}
```

It is an error for a template body to refer to instance fields, local variables, labels, and inner classes visible to the template expression, since these values exist in the compiler rather than the code being compiled. Hence, Maya rejects the following program:

```
class C {
  int i = 0;
  Expression e = new Expression { i };
}
```

Automatic hygiene and referential transparency prevent simple mistakes, but they also limit expressiveness. A Mayan could not expose a new variable to its arguments without violating hygiene, nor could a Mayan refer to a variable in its calling context without violating referential transparency.

3.6.1 Explicit Name Management

Maya provides mechanisms for explicitly breaking hygiene and referential transparency, as well as declaring hygienic class and member names. Identifiers can be constructed explicitly, and `Environment` objects can be used to explicitly control hygiene. The caller's lexical environment is stored in the static variable `Environment.current`.

An unrenamed `Identifier` may be allocated with `new`. The `Identifier` constructor takes one argument: the identifier's name. Unrenamed `Identifiers` can be used inside templates to export local variables and labels. The following template exposes an integer variable `state` to a lazily parsed block `body`:

```
new Statement {
  { int $(new Identifier("state")) = 0; $body }
};
```

`Environment.lookupId(String)` returns the variable or field that a name denotes in an environment, or null if no such entity exists. References to such a location can be embedded in templates. The following template increments the variable `state` visible to the caller:

```
final Location stateLoc = Environment.current.lookupString("state");
new Expression { $(as Expression Reference.make(stateLoc))++ };
```

Hygienically renamed `Identifiers` must be allocated through an `Environment`. The method `Environment.makeId(String)` returns a fresh `Identifier` whose name is derived from the provided string. This identifier can be used as the name of a hygienic method. The following example adds a static field to a class.

```
final Identifier name = Environment.current.makeId("class");
thisClassDecl.addDeclaration(new Declaration {
  static private Class $name;
});
Field f = (Field) Environment.current.lookupString(name.getName());
```

Maya ensures that such a generated name will not clash with names used in the source program directly. Maya also ensures that such a generated name is unique within a compilation unit. The calls `e1.makeId("class")` and `e2.makeId("class")` may return the same identifier only if `e1` and `e2` described environments in two distinct source files.

3.6.2 Mayans

Referential transparency does not apply to concrete Mayans, since they are values rather than names. A template is expanded to an AST using the set of Mayans visible to the source file being compiled. Occasionally, a template author may wish to refer to Mayans that are not visible at the point of use. For instance, `CatSS` in Figure 2.1 may be used in an environment without `StringCoerce`. This type of error is potentially annoying, but referential transparency cannot prevent such errors without a loss of polymorphism and pointer equality of Mayan instances.

An abstract Mayan can be viewed as a complicated name at the file scope, and abstract Mayans are indeed referentially transparent. A template body must be syntactically correct according to the set of abstract Mayans in scope at the template expression. Template expansion amounts to reducing the productions by which the template body was parsed.

Abstract Mayans can be thought of as generic functions, which are roughly analogous to virtual functions in Java and C++. A concrete Mayan can be thought of as a method defined on some abstract Mayan. When viewed this way, restricting template expansion to concrete Mayans in scope where the template is defined would be analogous to restricting a virtual function call to those methods in scope at the call site. This restriction would lead to the same loss of polymorphism.

As an example, consider `+=`. This Mayan operates on any arguments for which '`+`' is defined. If concrete Mayans where treated referentially transparently, '`+=`' could only operate on arguments for which '`+`' was defined at compile time. These arguments could only be numeric types, for which '`+`' is a built-in operation, and strings. This limitation would be unacceptable.

3.6.3 Grammar Support

Scheme macro systems make hygiene and referential transparency decisions at expansion time, when the syntactic role of each identifier is known. Since Maya statically checks templates for syntactic correctness, it can determine the role of each identifier at compile time. Therefore, hygiene and referential transparency decisions are made at compile time, with the help of hygiene-related productions.

Before an `Identifier` token is parsed to a local declaration, it must be reduced by one of the following productions:

```
abstract UnboundLocal  syntax(Identifier);
abstract UnboundLClass syntax(Identifier);
abstract UnboundLabel  syntax(Identifier);
```

Productions that define local variables, classes, and labels, respectively, use these symbols rather than `Identifier`. For instance, labeled statements are defined as follows:

```
abstract ExplicitLabel syntax(UnboundLabel :);
abstract Statement     syntax(ExplicitLabel Statement);
```

Maya hygienically renames every `Identifier` that passes through one of these productions.

A hygienic binding's lifetime is determined by Maya's standard rules:

- Method and catch clause parameters are in the scope of the corresponding blocks.

- A statement's label and declarations inside `for` headers and `switch` bodies remain in scope for the duration of the statement.

- Other names are in scope until the end of an enclosing lazy node, or the end of the template itself.

Note that the scope of a variable declared inside a Mayan may be overestimated. As a result, templates such as the following can be compiled, even though they expand to invalid code:

```
1 new Statement {{
2   $(e).foreach(Object o) { $(hash) ^= o.hashCode(); }
3   return o.hashCode();
4 }};
```

When this template is expanded, all occurrences of `o` will be replaced with a fresh identifier. Since `o` on line 3 is renamed and the declaration on line 2 is not in scope, this template will always produce an error at expansion time.

Mayans that bind local variables must take `UnboundLocal` rather than `Identifier` parameters to ensure hygiene. In the case of `foreach`, this is done implicitly, since the name of a formal is an `UnboundLocal`. `UnboundLocal` can also be used directly, as in the `let` example on page 22.

```
abstract Expression syntax(let UnboundLocal = lazy({ in }, VarInitializer)
                           in lazy({ end }, Expression) end)
  with precedence = ASSIGN - 1;
```

If `UnboundLocal` were replaced with `Identifier`, `let` could still be used directly in a source program. However, if such a `let` appeared in a template, the let variable would not be renamed, and could not be referenced in the template body.

3.7 API

This section discusses aspects of Maya's API in detail. In particular, it discusses node constructors that are used to build syntax trees where templates are inappropriate, and how the compiler's main loop interacts with lazy templates and class declaration updates.

3.7.1 Node Creation

This subsection describes node constructors and node-generating methods that work around limitations of Maya's concrete syntax. These methods allow abstract syntax trees to be created when no corresponding template can be written.

A Mayan programmer may wish to create an array of four elements with the component type `Object[]`. Unfortunately, '`new (Object[])[4]`' is not a valid expression in either Java or Maya. However, the abstract syntax for this operation can be created with an explicit constructor: `ArrayNewExpr(Type, ClosedDims)` builds an array allocation expression from a component type and a list of dimensions with sizes. For example, an expression that allocates array of four elements of type `t` can be created as follows: `new ArrayNewExpr(t, new ClosedDims{ [4] })`.

Mayans examine the overall structure of a source program through the structural reflection API, but alter the source program through abstract syntax. A number of AST-building methods are provided to map structural reflection objects back to syntax objects: `LazyTypeName.make()` and `StrictTypeName.make()` build `TypeName` nodes from `Types`, while `Reference.make()` builds a reference to a `gnu.bytecode.Location`, which is either a `Field` or a `Variable`.

Mayans defined on `Declaration` syntax often need to return more than one declaration. Such Mayans can return `SimpleDeclList` nodes. This is permitted because `SimpleDeclList` is a subtype of `Declaration`, even though there is no production `abstract Declaration syntax(SimpleDeclList);`. Similarly, a Mayan can expand `Statement` syntax to a sequence of statements by returning a `StmtList`. These nodes fill the role of `begin` in Scheme.

One often needs to build `ListNodes` incrementally, but this cannot be done with templates. A `list()` grammar symbol is defined using a single rule, and a

left-recursive helper. The single rule flattens the list of nodes to an array. If `list()` symbols themselves were left recursive, the following code would append an element to an argument list:

```
args = new list(Expression, ‘,‘) { $arg1, $arg2 };
args = new list(Expression, ‘,‘) { $args, $arg3 };
```

However, list symbols are not left-recursive. The correct code to append an element to an argument list is:

```
args = new list(Expression, ‘,‘) { $arg1, $arg2 };
args = args.append(arg3);
```

List nodes can be built up using three methods: `ListNode.insert(int, Node)` returns a new `ListNode` like its receiver except that it contains an additional element. `l.push(n)` is equivalent to `l.insert(0, n)`, and `l.append(n)` is equivalent to `l.insert(l.getChildren().length, n)`.

Maya supports inner Mayans, but inner Mayans cannot be explicitly imported by a source program. A `use` directive fails, since an inner Mayan's constructor takes arguments. However, `use` directives can be generated programmatically. The constructors `UseStmt(MetaProgram, DelayedStmt)` and `UseDecl(MetaProgram, DelayedNode)` allow a `use` directive to be built from a `MetaProgram` instance, rather than a type. The second argument of each constructor corresponds to the rest of the parser input. The Mayan is exported to the lazy syntax in the constructor's argument.

Mayans can control the abstract syntax generated by a given piece of concrete syntax, but in general Mayans cannot define new AST node types. There are three exceptions to this rule. First, Mayans can and do extend the type `DelayedNode`: template thunks are best implemented as inner classes that subtype `DelayedNode`. For instance, `mayac` compiles the following template

```
new Expression { ({ $(getExp()) + 1; }) }
```

to an anonymous subclass of `DelayedNode` that evaluates `getExp()` when the `BeginExpr`'s full syntax tree is needed. Second, Mayans can extend `LazyTypeName`

to control the way simple names are resolved. The inner class below is closed over two local variables: `LazyTypeName lt` and `Environment outerEnv`. For example, the class below wraps `lt`, and forces the type to be resolved in `outerEnv`:

```
new LazyTypeName() {
  public Type computeType(Environment env) {
    return lt.computeType(outerEnv);
  }
}
```

Finally, Mayans can extend `ExternalModifier`, a node class that provides a hook into the declaration processing discussed in Section 3.1.

3.7.2 Class States

Recall that a class declaration's current status is returned by `ClassType.-getState()`. This method reflects the class's position in the state machine in Figure 3.10, and implies that all classes that this class refers to are at most one step behind it.

Maya extensions can take advantage of this compilation model to support mutually referential classes. For instance, a class body can be wrapped in a template thunk that resolves `LazyTypeNames`, and a method body can be wrapped in a template thunk that examine members of other classes.

Maya exposes class processing through another mechanism: the class `ExternalModifier` may be subtyped, and these user-defined modifiers can be embedded in modifier lists. `ExternalModifier` defines two methods: `void interpret(Declaration)` is called when a declaration is interpreted in `Environment.current`,

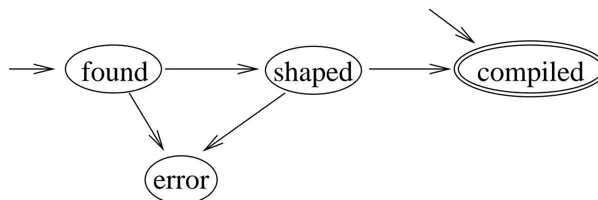

Figure 3.10. The states of classes being compiled

and `void finish(Declaration)` is called after the top-level class that contains the declaration has been compiled.

As a class declaration moves from `FOUND` to `SHAPED`, its members are interpreted and the `interpret()` methods of all `ExternalModifiers` are run. Finally, the class itself it interpreted. As a class declaration moves from `SHAPED` to `COMPILED`, after byte-code has been generated, the `finish()` methods of all declarations in that class are run.

The default implementation of `ExternalModifier.interpret()` generates a compilation error, while the default implementation of `ExternalModifier.finish()` does nothing. Since `ExternalModifier.interpret()` is not abstract, multimethods can easily be defined on this virtual function with the Maya extension described in Chapter 5: a subclass can specialize `interpret()` on subtypes of `Declaration` without defining an unspecialized method.

Maya also provides mechanisms to destructively update class declarations. `TypeDeclaration.members()` returns a `ListIterator` that allows a class body to be inspected and mutated. `TypeDeclaration.addDeclaration(d)` adds a `Declaration` to the class body: this method is equivalent to `members().add(d)`. A class cannot be updated in a way that invalidates previous type-checking and Mayan dispatch. Calls to `add()` and `remove()` that break this invariant throw a `java.lang.IllegalStateException`.

Maya allows members to be added and removed from a class declaration based on the class's state. Members may not be removed, and nonprivate members may not be added, after a class has entered the `SHAPED` state. Private members may be added to a class declaration, provided they do not shadow names in the class's environment, until the class enters `COMPILED`. These rules ensure that class mutation cannot change the semantics of any well-typed code that has already been parsed.

3.8 Java Issues

3.8.1 Keywords and Reserved Words

Maya adds a number of keywords to Java. However, most of these keywords are not reserved, and can be used normally in other contexts. Maya does define

three new reserved words: `use`, `syntax` and `nextRewrite`. These words are not identifiers and cannot name user-defined entities.

Other Maya keywords carry more subtle constraints. Objects cannot be created from the simple class names `list`, `nonEmptyList`, and `lazy`, since these names appear in template productions. In addition, the simple name `as` cannot be used in unquote expressions. However, the identifiers `with`, `precedence`, `associativity`, `left`, and `right` may be used without restriction.

3.8.2 Qualified Name Handling

In Maya, unlike Java, a class name can be shadowed by a field or local variable. This incompatibility arises from three fundamental differences between the languages. First, Java is context-free, but Maya is not. Second, Java compilers perform parsing and semantic analysis in separate passes, whereas Maya must interleave the two. Finally, Java's implementation grammar is opaque, while Maya's is exposed to the user.

Java avoids the context sensitivity of C++ through a number of simplifications, one of which is quite subtle [16, §19.1.5]: Java does not allow operator overloading. In particular, Java does not allow overloading of unary `+` and `-`.

Consider the following Java expressions, where `i` and `j` are `int` variables and `T` is a class name.

```
(i) + j
(T) +i
```

In both cases, we have a qualified name inside parentheses, followed by the token `+`. The first adds a parenthesized expression to a variable reference, while the second attempts to cast an `int` expression to a reference type. Since an LALR(1) parser cannot decide whether `(T)` is an expression or a cast type based on the next token, Java delays the decision.

The Java 1.0 specification defines cast syntax as follows:[1]

[1]The Java Language Specification uses symbols such as $Dims_{opt}$: this abbreviation has been removed for clarity.

$$
\begin{array}{lcl}
\textit{CastExpression} & \rightarrow & (\ \textit{PrimitiveType Dims}\ )\ \textit{UnaryExpression} \\
 & | & (\ \textit{PrimitiveType}\ )\ \textit{UnaryExpression} \\
 & | & (\ \textit{Expression}\ )\ \textit{UnaryExpressionNotPlusMinus} \\
 & | & (\ \textit{Name Dims}\ )\ \textit{UnaryExpressionNotPlusMinus}
\end{array}
$$

These productions force `(T) +i` to be parsed as a sum. This parse is safe because unary `+` is only defined on primitive types, and primitive types are never denoted by qualified names such as `T`. The third production accepts a superset of the Java language, including expressions such as `(String+1) o`. After parsing a cast expression, the compiler must verify that a cast type is in fact a qualified name that denotes a type.

Java's solution to the cast problem not only avoids the context-sensitive tricks found in C and C++ grammars, but also maintains a clean separation of type and expression syntax. Java exploits this separation to maintain separate namespaces for classes and variables (including instance and class variables) [17, §6.5].

Maya cannot avoid context sensitivity, since the unary `+` and `-` operators may be overloaded on reference types. In addition, Maya is context-sensitive in a more fundamental way. The result of Mayan dispatch, and hence the AST generated for a sequence of tokens, depends on the static semantic types of subexpressions. Finally, the static type of a variable reference depends on its context: specifically, the corresponding declaration.

Maya solves the cast ambiguity by allowing operator overloading at the expense of namespace separation. A type name may be shadowed by a local variable name in three contexts: local variable declarations, `instanceof` expressions, and cast expressions.

CHAPTER 4

IMPLEMENTATION

Maya's implementation relies on a number of novel parsing techniques. Maya's parse tables include multiple start symbols, and allow arbitrary identifiers to act as keywords. Maya uses these tables to parse patterns containing nonterminal leaves, as well as token streams. Patterns come in two varieties: binding patterns represent Mayan argument trees and `syntax case` pattern, while template patterns represent template bodies. Maya makes hygiene and referential transparency decisions as template patterns are parsed. In fact, template parsing and hygiene code is mutually dependent.

Maya is implemented in Scheme, which is compiled to Java byte-code. Specifically, `mayac` is written in a highly modified version of Kawa [3] that supports Java's object model. The two compilers share a great deal of infrastructure. Both use the same structural reflection and byte-code-generation code. The Maya compiler also reuses other parts of Kawa's implementation. In particular, `mayac` relies on Kawa's framework for embedding cyclic data structures in class files. Maya's compiler also reuses much of the library code that allows Scheme programs to define JVM classes: class name and overloaded method resolution is implemented once for both Scheme and Maya.

4.1 Data Structures

The Maya compiler uses four fundamental data structures. `Nodes` represent abstract syntax trees. The internal structure of `Nodes` allows code such as the Mayan dispatcher to treat lazy nodes, strict interior nodes, and leaves uniformly. `RightSymbols` represent syntax tree shapes, and are used throughout the compiler. The pattern parser builds `RightSymbol` trees, Mayans are dispatched with

`RightSymbol` trees describing their arguments, and built-in Mayans initialize their `Nodes` using the `RightSymbols` these Mayans match. `Action` is the superclass of all concrete Mayans, and associates a Mayan with the corresponding `Production`. `Productions` represent grammar productions, including abstract Mayans and helper rules for parameterized grammar symbols. Maya's parser generator associates precedence and associativity values solely with `Productions`, unlike YACC, which also associates this information with tokens.

4.1.1 Nodes

Maya defines node types in terms of one interface hierarchy and several implementation hierarchies. `Node` and the semantic categories `Expression`, `Statement` and `Modifier` are interfaces. Several node representations are defined as abstract base classes: `Token` is the class of objects generated by the stream lexer, `DelayedNode` is the class of parser thunks, `PairNode` is a subclass of Kawa's cons-cell type, and `InteriorNode` is the class of ordinary nonleaves.

Maya defines two subinterfaces on `Expression`: `LValue` and `SideEffectExpr`. Expression types that implement these interfaces can appear on the left-hand side of an assignment and at a statement context, respectively. Maya enforces these semantic restrictions with dispatch parameters.

Figure 4.1 shows the relationships between fundamental node types. An interface that is a subtype of `Node` is given a superscript, as is each class that implements it. For instance, `Expression` has superscript 3, as does its implementors: `InteriorExpr`, `Literal`, and `DelayedExpr`.

In addition to public methods such as `error()` and `getStaticType()`, node classes implement a set of methods that are used internally by the compiler. These methods generate Java byte-code, control laziness, and provide the internal reflection framework from which pattern matching is built.

Mayans access a node's children through pattern matching in argument lists and `syntax case` statements. To implement these features, `mayac` relies on simpler mechanisms. Maya examines a node's substructure through `Node[] Node.getChildren()`. `InteriorNodes` store their children in an array that this method

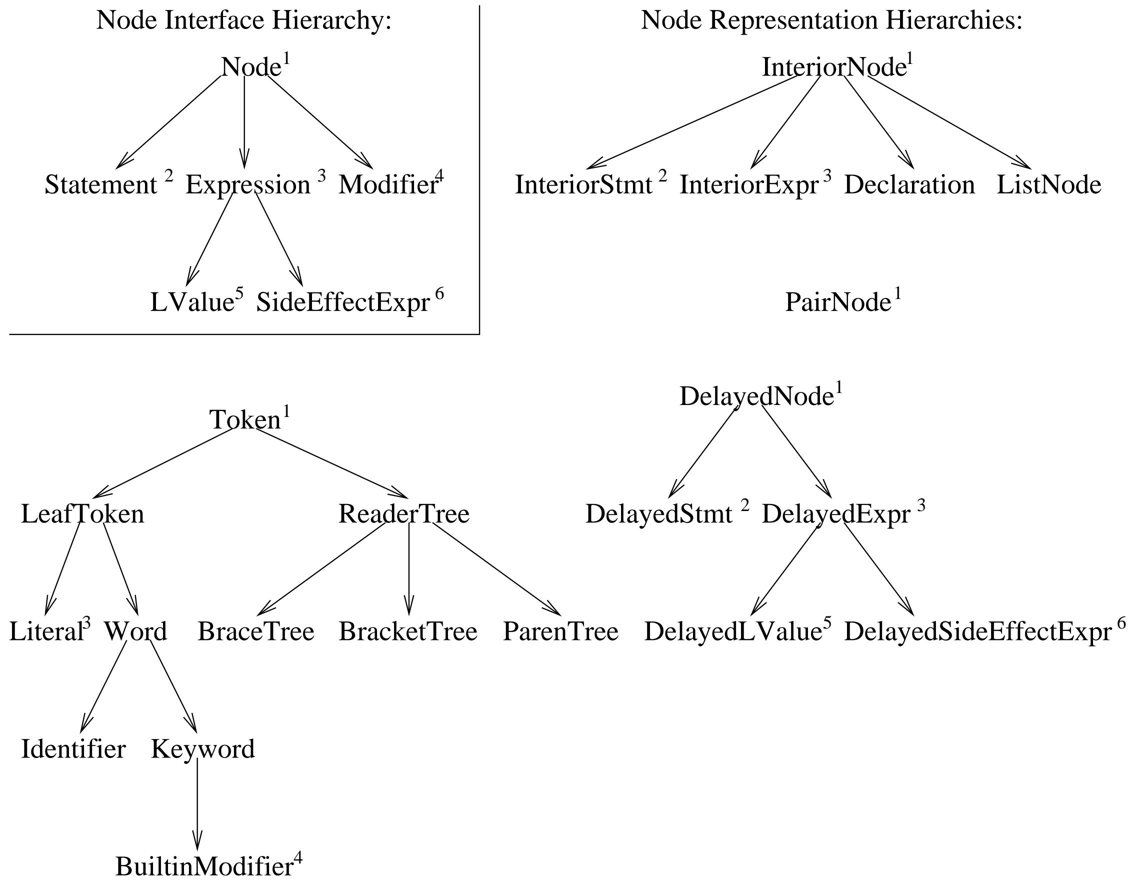

Figure 4.1. The node class hierarchy

returns. `DelayedNodes` implement this method through delegation. `PairNodes`, which are generated from left-recursive lists, return an array of `cdr` followed by `car`. Finally, `Tokens` return a zero-length array.

Each `InteriorNode` contains a fixed-length array of children. This array is large enough to hold every distinct child saved in every production that generates the node type. `String[] InteriorNode.getLayout()` returns an array of the symbolic names of corresponding child slots.

4.1.2 RightSymbols

`RightSymbols` contain all the information needed to dispatch Mayans, create nodes from built-in Mayans, and compile templates. To compile a template, `mayac` must simulate all shifts and reductions the parser would perform, and to dispatch Mayans based on argument structure, `mayac` must know the relationship between `RightSymbol` and `Node` children. Like `Nodes`, `RightSymbols` contain an optional array of subtrees. But unlike `Node.getChildren()`, `RightSymbol.structure` contains children in a parse tree. `RightSymbol` also contains an offset field. This field contains a corresponding `Node`'s offset in its parent's array of children.

`RightSymbol` has several subclasses. Maya defines `RightSymbol` subclasses for Mayan parameters specialized on static expression types, and those specialized on token values. Subclasses are also defined for parameterized grammar symbols and matching-delimiter subtrees. These `RightSymbols` must ensure that the appropriate productions are in the grammar.

4.1.3 Actions

All concrete Mayan classes are subtypes of `Action`, but abstract Mayans are not explicitly represented by values. A concrete Mayan ensures the existence of its `Production` and helper rules on its right-hand side when the Mayan is imported into an environment. An action carries the array of `RightSymbols` corresponding to its argument tree, and the grammar symbol it returns. These attributes, along with precedence and associativity, are sufficient to find the `Production` this action is defined on.

Like user-defined Mayans, built-in Mayans are represented by `Actions`. Most built-in Mayans are instances of `Builder`, a subclass of `Action` that allocates an `InteriorNode`, initializes its array of children, and calls `InteriorNode.reduce()` to perform any additional initialization. Other built-in Mayans, such as actions on single rules, and `list()` helpers are defined by other `Action` subclasses.

In Maya, the relationship between a built-in semantic action and the tree it produces is explicit. The pattern parser uses this information to build partial parse trees. Mayan arguments and unquoted expressions in templates are represented by `RightSymbols`. The pattern parser reduces these `RightSymbols` to build parse trees according to the built-in Mayans. If Maya's semantic actions were simply code, as in YACC parsers, pattern parsing would be more difficult.

4.1.4 Productions

`Productions` are used to define grammars and dispatch `Actions`. In Maya, productions define both an ambiguous grammar and the precedence and associativity values that resolve ambiguities. When a `Production` is reduced, `mayac` must find and execute the most applicable Mayan. Each `Environment` maintains a hash table that maps a `Production` to the set of Mayans defined on it. Mayan dispatch code is defined in the method `Node reduce(Environment, Pair)`.

Maya dispatches a reduction in several steps. First, it finds the set of Mayans defined on a `Production` in the current environment. Maya then recursively parses `ReaderTree` tokens that are not lazy. The set of Mayans applicable to the right-hand is computed and sorted as described in Section 3.3.3. Finally, the most applicable Mayan, if it exists, is called with two arguments: the right-hand side, and the `nextRewrite()` procedure. `nextRewrite()` is implemented by a local procedure that is closed over the sorted list of applicable Mayans.

Built-in Mayans implement many type-checking rules through dispatch. For instance, the '`*`' operator is defined on `long`, `double` × `long`, `double`. Attempts to multiply references or booleans are reported as no-applicable-Mayan errors.

4.2 Generating Parse Tables

Maya's LALR(1) parser and parser-generator are unusual in a number of respects. In Maya, a token may match multiple terminal symbols. For instance, `with` is both an identifier and a keyword used in abstract Mayan declarations. A Maya grammar contains multiple start symbols that are used to parse streams and `ReaderTrees`, and these symbols do not interfere with each other. Since generating parse tables is one of the dominant costs of compiling Mayans, several techniques have been developed to control this cost. A state machine can be reused when adding a start symbol to an existing grammar, and an efficient parser generator can be used after a Maya grammar can be transformed to one with a single start symbol. Currently Maya uses an efficient parser generator, `bison` [12], but recomputes all states between generator invocations. Maya defines precedence-based conflict resolution in an unusual way, and resolves certain reduce/reduce conflicts dynamically, as described in Chapter 3.

4.2.1 Token Categories

In mayac, a terminal symbol is represented by a pair <*type, value*>, where *type* is a `Token` type, and *value* is either a token instance or `#f`. Identifiers such as `with` may match a number of terminal symbols: <`Identifier`,`with`>, <`Identifier`,`#f`>, <`Word`,`#f`>, <`LeafToken`,`#f`> and <`Token`,`#f`>. If a parser state accepts two of the above symbols, the grammar is ambiguous. Since the parser generator is not aware of such ambiguities in token classification, Maya does not signal an error for them while generating parse tables.

In the parser, actions are sorted based on the presence of a token value and the hierarchy of token types. A token matches the most specific element of this sort, so that `with` matches <`Identifier`,`with`> in preference to <`Identifier`,`#f`>, and <`Identifier`,`#f`> in preference to <`Token`,`#f`>. <`Word`,`with`> is not included in this example, since literal token patterns use the token's most specific type.

4.2.2 Start Symbols

The parser is invoked several times to fully parse an input file. Because all invocations share the same grammar, the grammar includes multiple start symbols, and one start symbol may appear in a derivation of another. Each start symbol introduces EOF to the lookaheads of some items. To avoid spurious reduce/reduce conflicts, we define a distinct EOF token for each start symbol.

Start symbols come from two sources. First, an abstract Mayan that uses a matching-delimiter tree ensures that the tree's contents is a start symbol. For instance,

```
abstract Expression syntax(MethodName(list(Expression, ‘,‘))
  with precedence = PRIMARY;
```

ensures that `list(Expression, ‘,‘)` is a start symbol. Second, the pattern parser must ensure that the symbol the pattern matches is a start symbol.

Two approaches can be used to make start-symbol changes efficient. First, an unoptimized parser generator, such as Mark Johnson's Scheme parser generator [18] can be modified to reuse intermediate data from previous runs. Second, a highly optimized parser generator such as `bison` can be used with little change. Maya adopts the second approach.

Note that adding a start symbol has two effects: additional states are created, and lookaheads on a new EOF symbol are added to existing states. If one is careful to preserve state numbers, parse tables can be destructively updated with new start symbols: a parse begun with the old tables can continue with the new tables. This approach allows us to reuse the state machine and lookahead sets for an old parser. With this change, the cost of updating the state machine and repropagating lookaheads is negligible, but it is not obvious how to memoize conflict resolution decisions.

In `mayac`, the cost of resolving conflicts and converting a state machine to parse tables is significant. It accounts for roughly 1/3 of the time spent in the Scheme-based parser generator. Currently, `mayac` uses an optimized parser-generator that supports neither multiple start symbols, nor the start-symbol optimization de-

scribed above. To use `bison`, the Maya compiler must transform the grammar to contain a single entry point, then transform `bison`-generated parse tables to contain multiple start states.

A simple transformation eliminates multiple start symbols in a Maya grammar. Each augmenting start symbol $S_i \in S_0, \ldots S_n$ defines a corresponding EOF token E_i. Before a Maya grammar can be passed to `bison`, we must create a single start symbol S'. For each augmenting symbol S_i and the corresponding grammar symbol, T_i, we define two productions:

$$\begin{array}{lcl} S' & \rightarrow & S_i \\ S_i & \rightarrow & E_i\ T_i\ E_i \end{array}$$

As in the Scheme parser, distinct EOF tokens prevent reduce/reduce conflicts, but these tokens are also used to determine the initial state for each start symbol S_i. Starting from `bison`'s initial state, `mayac` finds the initial state for each start symbol by shifting the corresponding EOF.

With `bison`, the Maya compiler does not reuse state machines when adding start symbols, since this optimization would require sweeping changes to the `bison` code. However, we can still control the number of times a start symbol is added. If a Maya program contains two syntax case statements on `MethodName` with five clauses each, we would like to add the start symbol `MethodName` and regenerate parse tables once, rather than ten times. The parser keeps local references to action and goto vectors, and the corresponding table is destructively updated with new entry points. This update replaces the action and goto vectors, and does not interfere with parsing in progress.

Using `bison` as a Java Native Library, `mayac` generates parse tables roughly 10 times faster than with the Scheme parser generator. However, the cost of resolving conflicts is magnified. On a 350MHz Pentium-II, `bison` generates an LALR(1) state machine in 40ms, while conflict resolution and table generation accounts for 160ms. In part, this cost comes from the unusual way the Maya handles conflicts.

4.2.3 Conflict Resolution

Whereas YACC associates precedence values with both productions and tokens, `mayac` associates precedence values only with productions, which complicates shift/reduce conflict resolution. Whereas YACC resolves reduce/reduce conflicts based on the lexical order of rules, `mayac` resolves reduce/reduce conflicts based on the actual values being reduced.

To resolve a shift/reduce conflict `mayac` examines the core items of the shift action's state. The compiler checks that all items in the shift state belong to rules with identical precedence and associativity values. This restriction forces the rules

$$\begin{array}{rcl} \mathit{StrictName} & \rightarrow & \mathit{StrictName}\ .\ \mathit{Identifier} \\ \mathit{MethodName} & \rightarrow & \mathit{StrictName}\ .\ \mathit{Identifier} \end{array}$$

to have the same precedence, but does not constrain the precedence of

$$\mathit{QualifiedName} \rightarrow \mathit{QualifiedName}\ .\ \mathit{Identifier}$$

When generating the Java grammar, `mayac` resolves over 2,000 shift/reduce conflicts on 43 distinct shift states. Since computing shift precedence involves examining each core item, Maya caches state precedences rather than recomputing it on each conflict or precomputing the precedence of over 600 states.

Maya also resolves certain reduce/reduce conflicts dynamically. Maya's parser supports three types actions. A token may be shifted, a rule may be reduced, or a `RuntimeResolver` may be invoked. `RuntimeResolvers` take values on the right-hand side of a production and return the rule that should be reduced. When a `RuntimeResolver` chooses a rule, the parser reduces it, updates the stack and follows the appropriate goto.

In the Java grammar, three sets of rules create reduce/reduce conflicts. First, in array `new` expressions, `BraceTree` may reduce to either an open dimension with no size, or a closed dimension with a size. Second, `StrictName` may reduce to either `StrictTypeName` or `Expression`. Finally, `Expression` can reduce to a `ParenTree` containing either a subexpression, a cast type, or a begin expression body. The Maya compiler may recursively parse subtrees to resolve reduce/reduce conflicts.

Although the number of distinct reduce/reduce conflicts in the Java grammar

is quite small, these conflicts arise in 104 actions. To speed up table generation, `mayac` caches `RuntimeResolver` objects in an associative list.

4.3 Pattern Parsing

A Mayan parameter list is translated to a tree of `RightSymbols` in two stages. First, each parameter and literal token is translated to a `RightSymbol`. Next, these `RightSymbols` are used as input to the pattern parser, which constructs a parse tree. Template bodies are parsed similarly. The pattern parser also provides hooks needed to enforce hygiene and referential transparency.

While some systems have developed ad hoc approaches for template parsing [2, 6, 26, 27], Maya opts for a more general solution. Mayans can be defined on any rule whose left-hand side is a node type, and templates can generate both nonterminal node types and parameterized grammar symbols.

4.3.1 The Parsing Algorithm

The description of the pattern parsing algorithm uses the function names and lexical conventions defined in [1, §4.4]: upper case letters represent nonterminal symbols, lower-case Greek letters represent strings of both terminals and nonterminals, lower case Roman letters represent strings of terminals, and FIRST maps a string of symbols to the set of all terminals that appear first in some derivation.

The pattern parser uses parse tables in much the same way as a normal LALR(1) parser. Terminal input symbols are shifted onto the stack, or are used to trigger reduction normally. nonterminal symbols require some special handling. When the pattern parser encounters a syntactically valid input $A\gamma$, it must be in some state s_0 such that either:

1. s_0 contains an item $B \rightarrow \alpha.A\beta$: actions on $\textsc{first}(A)$ are all shifts to the same state, and this state contains a goto for A leading to s_n, or

2. s_0 contains an item $C \rightarrow \delta.$, where C is a nonterminal such that. $\alpha \stackrel{*}{\Rightarrow} \zeta C$: the actions on $\textsc{first}(A\gamma)$ all reduce the same rule $C \rightarrow \delta$.

In the first case, A is shifted onto the stack, and the goto is followed. In the second case, the stack is reduced leading to a state s_1 in which one of the above conditions holds. Figure 4.2(a) shows a simple grammar in which both cases arise. Figures 4.2(b) and 4.2(c) depict the parser configuration in each case.

4.3.2 Proof of Correctness

Given an LALR(1) grammar G that contains the production $A \rightarrow \gamma$, the pattern parser P_{new} will accept $w_0 A w_2$ if and only if $\forall w_1$ such that $A \overset{*}{\Rightarrow} w_1$, a conventional

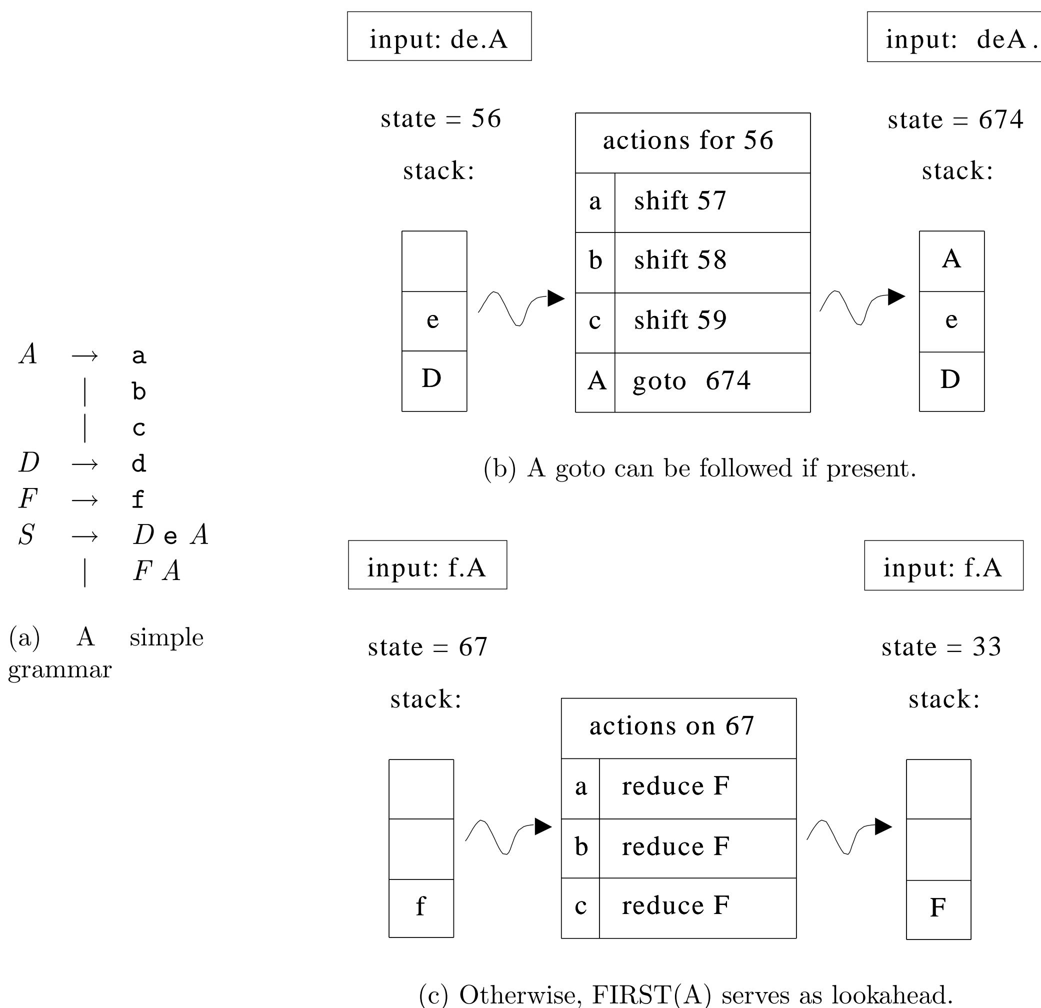

Figure 4.2. Pattern parsing example

parser P_{old} accepts $w_0w_1w_2$, and w_0Aw_2 is a sentential form of G ($S \overset{*}{\Rightarrow} w_0Aw_2$). We prove the backward case, that P_{new} accept will accept input when the proper conditions are met, directly. The forward case, that the proper conditions are met when P_{new} accepts input, is proven by showing its contrapositive: that P_{new} rejects input when the conditions are not met.

- If P_{old} accepts $w_0w_1w_2$ and $S \overset{*}{\Rightarrow} w_0Aw_2$, P_{new} accepts w_0Aw_2. Both P_{old} and P_{new} will consume the terminal input w_0, and arrive at the state s_0 as described in the parsing algorithm.

  In case 1, P_{new} will shift A onto the stack and follow the goto. If w_1 is empty, P_{old} will perform one or more ϵ reductions, eventually following the goto for A. If w_1 is non-empty, P_{old} will begin shifting it, eventually returning to s_0 and following the goto.

  In case 2, P_{old} 's lookahead token (the first token in w_1w_2) is an element of $\textsc{first}(Aw_2)$. Both P_{old} and P_{new} will reduce C, and arrive at a state s_1 such that one of the cases above holds.

  Since P_{old} accepts $w_0w_1w_2$, both P_{old} and P_{new} will eventually reach s_n with the same stack. From this configuration, P_{old} and P_{new} will behave identically, because w_2 contains only terminals,

- If either P_{old} rejects $w_0w_1w_2$ or w_0Aw_2 is not a sentential form of G, P_{new} rejects w_0Aw_2.

  – If P_{old} does not accept $w_0w_1w_2$, P_{new} does not accept w_0Aw_2. If P_{old} fails before consuming w_0 or after shifting A, P_{new} fails in the same way. Otherwise, both P_{old} and P_{new} will consume the terminal input w_0, arriving at a state s_0. P_{new} may perform zero or more reductions based on $\textsc{first}(Aw_2)$ to arrive at a state s_n. P_{old} performs the same reductions based on its lookahead token.

    The state s_n does not contain any item $B \rightarrow \alpha.A\beta$, and therefore does not contain a goto for A. If s_n did contain $B \rightarrow \alpha.A\beta$, P_{old} would be

able to shift w_1, eventually returning to s_n and following the goto for A. But, P_{old} fails while parsing w_1, leading to a contradiction.

P_{new} does not accept $w_0 A w_2$ because it is unable to shift A.

– If P_{old} accepts $w_0 w_1 w_2$, but S does not derive $w_0 A w_2$, P_{new} will not enter a state containing $B \rightarrow \alpha.A\beta$ after consuming w_0, and will fail. If P_{new} where able to move into such a state, P_{old} would parse w_1 as A, leading to a contradiction.

Not only does P_{new} accept or reject a nonterminal input properly, but after a nonterminal is shifted, it enters the same configuration P_{old} enters after the symbol is reduced. Given this fact, P_{new} can be shown to parse a stream of one or more nonterminals correctly through simple induction. The close relationship between configurations in P_{old} and P_{new} also ensures that P_{new} generates `RightSymbol` trees of the appropriate shapes.

4.3.3 Dispatch and Hooks

The parse trees that the pattern parser generates are used in two distinct ways. `RightSymbols` in Mayan argument lists are matched against node values and used to extract subtrees, while `RightSymbols` in templates are compiled into code that shifts and reduces arbitrary nodes.

Consequently, the pattern parser must determine the type and layout of each `RightSymbol` in a Mayan argument tree, but need determine only the parse tree that corresponds to a template body. Maya allows templates to generate user-defined syntax, where the type and layout of each descendent is not known until expansion time. The pattern parser must also handle the reduce/reduce conflicts that render Maya context-sensitive. In particular, the pattern parser must distinguish between a `StrictTypeName` and `NameExpr`. The pattern parser satisfies these requirements by simulating Mayan dispatch and providing two hooks to its callers.

When the pattern parser reduces a production, it finds all actions applicable to the right-hand side. If at least one built-in Mayan is applicable and all built-in Mayans produce the same node type and store arguments at the same offsets, the

reduction is simulated. A `RightSymbol` that denotes the left-hand side is created and passed to the *reduce hook*, which is free to mutate it. If no built-in Mayans are applicable, the pattern parser calls the *unreduceable hook*. This hook may return a `RightSymbol` that denotes the left-hand side, or `null` if the pattern is invalid.

Every qualified name enters the pattern parser as an `Identifier`. The reduce hook fills in substructure when a qualified name is reduced to `StrictName`, `LazyTypeName`, or `MethodName`. In these cases, a qualified name is resolved to a type name. A `StrictName` symbol's node type is set, allowing the pattern parser to distinguish `NameExprs` from `StrictTypeNames`. Names that are reduced by other rules must be simple identifiers.

The hooks provided for Mayan argument list parsing ensure that user-defined syntax only appears at the root, and enforces referentially transparent class-name handling. The unreduceable hook is called when no built-in Mayans are applicable, and signals an error unless it is called for the final reduction.

Template parsing is more involved. Since fields and local variables may shadow class names, a qualified name's parse tree depends on the surrounding template. The Maya compiler uses reduce and unreduceable hooks to interleave hygiene and referential transparency checks with template parsing. These hooks parse templates lazily, mirroring the order in which code is parsed. The Maya compiler also uses the unreduceable hook to build parse trees where no built-in Mayans are applicable.

CHAPTER 5

EVALUATION

To show that Maya can be used for everything from simple macros to full-blown language extensions, we implemented both types of meta-programs. Maya's standard macro library includes several utility macros in addition to derived Java syntax. Maya also includes an implementation of MultiJava [8], a Java extension that defines multiple-dispatch and open classes.

Mayans can take advantage of several language features that are not available to base-level Maya programs: multiple dispatch, pattern matching, explicit laziness, and a form of parameterized types. Methods are not subject to multiple dispatch. Other advanced features can only be used with AST nodes, rather than arbitrary objects. MultiJava provides base-level Maya programs with some of the expressiveness available to meta-programs.

By examining these macros and our MultiJava implementation, one can answer several questions about Maya. Is it practical to implement a macro as an abstract Mayan declaration followed by a concrete Mayan definition? Is Maya powerful enough to implement language extensions? Must a language extension's semantics be tweaked for an implementation in Maya? Finally, how easily can Mayans be composed?

5.1 Macros

Maya defines several utility macros for simple tasks such as copying `Collections` into arrays, assertions, debug logging, iteration, and formatted output. These Mayans range in complexity from trivial to straightforward. They show the minimum overhead involved in writing a useful Mayan. They also demonstrate that several simple Mayans can be composed without interfering with each other.

The size of each source file is shown in Table 5.1. The second column shows the total number of lines in each file. Subsequent columns break down a file's contents into four categories: lines containing only comments, lines containing only whitespace, lines containing only punctuation characters, and other lines (which presumably do something useful).

5.1.1 Arrays from collections

In Java, it is common to accumulate objects in a `Vector`, then copy these objects into an array for more permanent storage. The Mayan `ArrayFromCollection` abstracts code such as the following

```
MultiMethod[] meths = new MultiMethod[v.size()];
v.copyInto(meths);
```

into a single line: `MultiMethod[] meths = new MultiMethod[](v)`.

5.1.2 Assert and debug

Assertions and debug logging are two common uses for macros. In Maya, assert and debug are statements that have the following syntax:

```
abstract Statement syntax(assert(Expression););
abstract Statement syntax(debug(Identifier, list(Expression, ‘,‘)););
```

Assert expands to code that throws an `Error` if the invariant expression is not true. A small recursive method checks that assertion tests have no side-effects, which allows assertions to be safely turned off by the `NDebug` meta-program. Unlike the equivalent C header file, `NDebug` also turns off debug logging.

Table 5.1. Size of utility macro code

FILE	LINES	COMNT	BLANK	PUNCT	CODE
ArrayFromCollection.maya	31	8	3	5	15
Assert.maya	32	0	5	8	19
Debug.maya	54	13	7	8	26
NDebug.maya	23	0	5	4	14
ForEach.maya	139	10	19	27	83
Printf.maya	387	13	54	81	239
PrintfRT.maya	43	3	5	10	25

Maya's debug macro was written on the assumption that a class called `Debug` will be accessible from each source file. This class should contain a static boolean field that corresponds to category of debug messages that can be generated. Debug flags could be represented in other ways, such as bit masks. However, static fields can be set from the command line through simple use of reflection: the command line parser need not know which flags are defined.

The debug macro takes a `Debug` field name followed by either an arbitrary `Expression` that is passed to `System.err.println`, or a printf argument list that is formatted to `System.err`. Building a reference to a static field in the class `Debug` is more complicated in Maya than it would be in a simple macro system based on textual substitution: It takes three lines to convert the string `"Debug"` to a `StrictTypeName` object.

5.1.3 Iteration

Maya defines an iteration construct, `foreach`, on several types. Implement ions of `foreach` use many features, such as argument destructuring, dispatch on static types, and hygiene to a greater extent than the Mayans described above.

Recall from Chapter 3 that `foreach` is defined on the abstract Mayan:

```
abstract Statement syntax(MethodName(Formal) lazy(BraceTree, BlockStmts));
```

Implementations of `foreach` override this production: the `MethodName` receiver is an expression, and the name itself is `foreach`. This formulation prevents `foreach` from acting as a reserved word.

`IForEach` in Figure 5.1 defines foreach on iterator objects. It uses dispatch on static types to select the `foreach` statements it handles, and relies on hygiene to properly declare the iterator variable `enumVar`. `IForEach` also constructs nodes using several static methods, where templates do not suffice.

Other Mayans are defined similarly, and implement `foreach` on `Enumerations`, `ListNodes`, and arrays. Maya also optimizes `foreach` on calls to `Vector.elements()`. Rather than walking a `java.util.Vector` using its `Enumeration`, we copy the vector contents into a local array. Calls to `foreach` on `maya.util.Vector`

```
Statement
syntax IForEach(Expression:Iterator enumExp.foreach(Formal var)
                lazy(BraceTree, BlockStmts) body)
{
  final StrictTypeName castType = StrictTypeName.make(var.getType());

  return new Statement {
    for (Iterator enumVar = $(enumExp); enumVar.hasNext(); ) {
      $(DeclStmt.make(var))
      $(as Expression Reference.make(var)) = ($castType) enumVar.next();
      $body
    }
  };
}
```

Figure 5.1. `foreach` Mayan to walk iterators.

are optimized further. This class exports a public method to retrieve its internal array, and `foreach` iterates over this array without copying.

Like `IForEach`, most `foreach` definitions select the appropriate calls by specializing on the static type of the receiver object. This is not possible for the array iterator, `AForEach`, because in Java there is no single type for all arrays. Since `AForEach` is applicable to types such as `Collection`, for which the `foreach` operation is not defined, `AForEach` performs type checking by hand:

```
Statement syntax AForEach(Expression e.foreach(Formal var)
                          lazy(BraceTree, BlockStmts) body)
{
  final Type t = e.getStaticType();

  if (!t.isArray())
    throw e.error("foreach not defined for " + e.getStaticType());
  // ...
}
```

`AForEach` reports errors in the format by throwing an unchecked exception, `CompilationError`. An error generated by `AForEach` is slightly more informative than the error that `mayac` would have generated on its own, “no applicable mayans.” In more complicated Mayans, user-defined error messages are more important.

5.1.4 Formatted output

Maya's standard library includes a formatted printing facility, a variant on C's `printf` formats. We use it to demonstrate that Mayans can take a variable number of arguments and how Mayans can perform explicit type checking. The integer size specifiers, `h` and `l`, have been removed and some formats, `%c` and `%s`, have been subsumed by a polymorphic format `%a`. Floating-point formats and `%n`, which updates an argument with the number of characters written, have not been implemented.

Like many C compilers, Maya parses and type-checks formats at compile time. However, Maya's `printf` has two advantages over C: user-defined format directives can be type checked, and the format string is not re-parsed at runtime.

Several Mayans are defined in terms of the format parser. The simplest, `Sprintf`, is shown below:

```
Expression syntax Sprintf(sprintf(list(Expression, ‘,‘) args)) {
  FormatState state = new FormatState(args);
  Expression ret = state.parse();

  for (Expression e = state.parse();  e != null;  e = state.parse())
    ret = new Expression { $ret + $e };
  return ret;
}
```

`Sprintf` overloads Maya's method-call syntax. Specifically, `Sprintf` rewrites calls to a method named `sprintf` with no explicit receiver. The parameter `args` matches the comma-separated list of method argument expressions. `Sprintf` builds a string concatenation expression from string-valued expressions computed by `FormatState.parse()`, which does all of the work.

While the code to parse a format directive is uninteresting, it is useful to examine how an argument corresponding to a '`*`' width specifier is consumed. `FormatState.parse()` calls `pop(int.class)`, where the method `FormatState.pop()` is defined in Figure 5.2. The method `Node.error()` builds a `CompilationError` and generates an error message with a file name and line number. `FormatState` performs explicit type checking using `Expression.getStaticType()` and Maya's introspection API.

```
class FormatState {
  Node[] args;
  int argInd = 1;
  // ...
  Expression pop() {
    if (argInd == args.length)
      throw args[0].error("too few arguments to format");
    return (Expression) args[argInd++];
  }

  Expression pop(Class need) {
    Expression e = pop();
    Type nType = MayanSpace.the.forClass(need);  // map java Class to maya Type
    Type fType = e.getStaticType();

    if (!nType.isAssignableFrom(fType))
      throw e.error("format expected a " + nType + " but found a " + fType);
    return e;
  }
  // ...
}
```

Figure 5.2. `FormatState` members involved in error checking

The following program fails to compile with the error message '`PrintfEx.maya:5:21:format expected a int but found a java.lang.String`':

```
use maya.util.Printf;

public class PrintfEx {
  static final String st = sprintf("%*a", "a string", 10);
}
```

If `FormatState` did not explicitly check the types of width parameters, Maya would generate a less useful error message, '`PrintfEx.maya:5:6:no applicable methods for maya.util.PrintfRT.fill`'.

5.2 MultiJava Overview

MultiJava [8] extends Java with multimethods and open classes. Multimethods are used to structure method implementation according to the types of all

arguments, rather than just the receiver type. Open classes offer an alternative to the Visitor design pattern [15]. Rather than maintaining a visit method on each class in a hierarchy and a hierarchy of visitor classes, one can simply augment the visited class hierarchy with externally defined methods. External methods can be added without recompiling the visited class hierarchy, but a class is free to override external methods that it inherits.

5.2.1 Multimethods

MultiJava integrates multimethods with standard Java's overloading semantics. Each overloaded method is a generic function. Multimethod definitions include both the static type of a parameter and its specializer type. For instance,

```
class C {
  void m(C c) { }
  void m(C@D d) { }
}
class D extends C {
  void m(C c) { }
  void m(C@D d) { }
}
```

defines four methods on a single generic function: `C.m(C)`. Only one change to the Maya grammar is needed to support multimethods:

```
abstract Formal syntax(list(Modifier) LazyTypeName@LazyTypeName
                       LocalDeclaratorId);
```

The type preceding '`@`' in a method parameter declaration is the type of that parameter in the generic function's signature. The type that follows '`@`' is the specializer that declares the runtime type that the multimethod is applicable to.

MultiJava supports *symmetric multiple dispatch*, which means that the most applicable method for a particular argument tuple must be most applicable to each argument. MultiJava statically checks that a generic function is *unambiguous*, which means that there is at most one best method for every possible argument tuple. MultiJava also statically checks that generic function is *complete*, which means that there is at least one applicable method for every possible argument tuple for every argument.

These static checks are made possible by two restrictions on how arguments can be specialized: Only class-typed arguments may be specialized, and then only by other class types. Although abstract classes need not implement multimethods for all combinations of argument types, concrete classes must define or inherit concrete multimethods for abstract argument types.

5.2.2 Open Classes

MultiJava's open class feature allows *external methods* to be declared at the top level. An external method augments its receiver class, and can be used just like any other method. For instance, a new method `n` may be added to the class hierarchy above:

```
void C.n() { m(this); }
void D.n() { super.n() }
```

External methods must be declared at the top level, and their syntax is as follows:

```
abstract Declaration
syntax(list(Modifier) LazyTypeName QualifiedName \. Identifier (FormalList)
       Throws lazy(BraceTree, BlockStmts));

abstract Declaration
syntax(list(Modifier) void QualifiedName \. Identifier (FormalList)
       Throws lazy(BraceTree, BlockStmts));
```

Within an external method, `this` is bound to the receiver instance. However, external methods in MultiJava may not access private fields of `this`, nor can they access nonpublic fields of `this` if the receiver class is defined in another package.

MultiJava imposes other restrictions on external methods: First, all external methods on a virtual function must have identical access modifiers and must declare the same set of thrown exceptions. Second, external methods may be neither `abstract` nor `static`. Finally, an external method cannot override a method in a different compilation unit. External methods can, however, be overridden by internal methods in other compilation units.

5.3 Implementing Multimethods

Multiple dispatch is implemented by translating each multimethod to a private method whose formal parameter types are the specializer types, and generating

a dispatcher method that implements the generic function signature. A multimethod call is compiled in the same as any other method call. Super calls from multimethods do, however, require special handling. A super call invokes the next applicable method either by calling a private method in the same class, or calling the dispatcher method in a superclass.

Multimethod declarations are handled in three steps. First, the concrete syntax for type specializers is recognized, and multimethods are associated with declaring classes. Next, a dispatch method is created for each virtual function for which multimethods are defined. Finally, the dispatch code is generated and ambiguous and incomplete generic functions are detected.

Dispatch code for a generic function consists of a series of instanceof tests that select the most applicable method. For instance,

```
class C {
  void m(C c) { System.out.println("CC"); }
  void m(C@D d) { System.out.println("CD"); }
}
class D extends C { }
```

will be translated as follows:

```
class C {
  private void m'(C c) { System.out.println("CC"); }
  private void m''(D d) { System.out.println("CD"); }
  void m(C arg0) {
    if (arg0 instanceof D) m''((D) arg0);
    else m'(arg0);
  }
}
class D extends C { }
```

5.3.1 Interpreting Concrete Syntax

Maya's implementation of multimethods is carefully organized around class declaration states. Code that works with method declarations must be careful not to call `LazyTypeName.toType()` until the enclosing class declaration is ready. A `MethodsModifier` is added to each class declaration. `MethodsModifier` is a subclass of `ExternalModifier`, and keeps multimethod processing in step with class declaration processing.

`MethodsModifier` exports two local Mayans to the class it modifies. These Mayans rename methods with parameter specializers, and save these methods in a per-class list. Any parameter specializer that these local Mayans do not handle, such as a specializer on a constructor or catch clause parameter, generates an error.

To support class declarations such as the following:

```
class C {
  void m(D d) { }
  void m(D@E e) { }
  class D { }
  class E extends D { }
}
```

`LazyTypeName.toType()` must not be called on a method parameter until `MethodsModifier.interpret()` is called on the enclosing class. External modifiers such as `MethodsModifier` are run just before a class enters the shaped state, after all members have been interpreted. The `interpret()` method is responsible for determining the generic function of each multimethod in a class. This method calls `GenericFunction.generateDispatcher()` to create dispatch methods, and type checking is deferred until the dispatch method body is generated.

5.3.2 Generic Functions and Type Checking

MultiJava's restrictions on multiple dispatch greatly simplify its implementation. Since a concrete multimethod must be provided for an abstract argument type, a generic function must define a method for each combination of specializers to be complete. Since specializers follow a single-inheritance hierarchy, completeness can be determined by considering only specializer types. For multimethods in classes `C` and `D` to be ambiguous, both must apply to the same receiver, and one must be a subtype of the other. Hence, one can detect ambiguous and incomplete generic functions based only on local and inherited methods.

MultiJava considers additional methods when typechecking generic functions. Specifically, MultiJava considers methods in *visible classes* (classes mentioned in the source-file of the class being compiled). In the following example:

C.java

```
class C {
  static D d = new D();
  void m(E e1, E e2)   { }
  void m(E e1, E@F E2) { }        // ambiguous with E.m
}

class E { }
class F extends E { }
```

D.java

```
class D extends C {
  void m(E@F e1, E e2) { }          // ambiguous with C.m
}
```

MultiJava reports an error in both `C.m()` and `D.m()`. (`C` is visible from `D.java` since `D` inherits from `C`, and `D` is visible from `C.java` since `D` is used in a field declaration.) However, if the static field `C.d` is removed, `D` is no longer visible from `C.java` and an error will be reported only in `D.m()`.

In Maya there is no practical way to determine the set of classes visible to a source file. Therefore, Maya's multimethod implementation generates fewer error messages than MultiJava, but will reject invalid programs provided that build dependencies are honored.

To determine whether a generic function is ambiguous, Maya's multimethod implementation considers methods defined in subclasses that are specializers in local and inherited multimethods. Maya's implementation can detect binary method ambiguities even if out-of-date subclasses are not recompiled. Such an ambiguity is shown below:

C.java

```
class C {
  void m(C@D c) { }               // ambiguous with method in specializer
}
```

D.java

```
class D extends C {
  void m(C c) { }                 // ambiguous with inherited method
}
```

Dispatch methods are generated in the simplest way possible. The set of multimethods is sorted from most to least specific and `instanceof` tests are per-

formed for every specializer on the first argument. Inapplicable methods are removed and subsequent arguments are recursively dispatched until either a single method is applicable or all arguments have been tested. Incomplete and ambiguous generic functions are detected while generating dispatch code. These tests would be straightforward if methods in subclasses where excluded, but in fact they are subtle.

5.4 Implementing Open Classes

MultiJava defines type checking rules and a compilation strategy that allows external methods and their receiver classes to be compiled separately. Essentially, an external virtual function is compiled to a dispatcher class, and calls to an external method are replaced with calls to a method in the dispatcher class.

We chose to implement MultiJava's semantics using a different compilation strategy. Rather than patch calls to external methods, we borrow techniques from aspect-oriented programming [28] and weave external methods' syntax trees into their receiver class declarations. For instance,

```
class C { }
void C.m() { System.out.println("m called"); }
```

is translated to

```
class C {
  void m() {
    // use mayans that enforce encapsulation
    System.out.println("m called");
  }
}
```

We chose this implementation strategy for its simplicity. However, recent clarifications to the MultiJava language [7, 23] make the original compilation strategy more tractable.

Currently, MultiJava only allows `super` sends from external methods to the same virtual function. Hence, the call to `super.n()` in `D.n` on page 68 is legal, while a call to `super.m(this)` would be illegal. This rule allows classes to reliably interpose

on method calls from the outside world. More importantly from an implementor's perspective, however, it eliminates the need to add accessor methods to binary files.

All external methods on a virtual function must have identical access modifiers, as described in [8]. Additionally, all external methods must throw the same set of checked exceptions. (Internal methods may override methods that throw a superset of checked exceptions.) Implementing covariant exceptions in a full MultiJava compiler would not be difficult, since the JVM does not verify `Exception` attribute definitions. However, implementing such semantics in Maya would involve generating `try` statements to placate Maya's typechecker.

5.4.1 Using Open Classes in Maya

Our implementation of open classes enforces MultiJava's type checking rules, so that every valid external method declaration in our weaver is valid in MultiJava, but weaving does not preserve all of MultiJava's semantics. Our implementation limits the package in which an external method can appear, the visibility of external methods, and the way in which the compiler is invoked.

In the weaver, an external method must be defined in the same package as the class it augments. This restriction allows external methods to reliably refer to package-private classes.

Weaving also changes the visibility of external methods. In MultiJava, classes and external methods can be imported separately: '`import p.C;`' will not import external method `C.n`, although '`import p.*;`' will. With our weaver, clients cannot distinguish between internal and external methods. This approach offers benefits: a library built by the weaver can be used by any Java compiler, and code compiled by the weaver dispatches external method calls with the standard `invokevirtual` instruction.

The biggest difference between MultiJava and the weaver is the way that files are compiled. In the weaver, external methods and the classes they augment must be compiled together. There is also a constraint on the order of files on the command line: a class declaration must appear no earlier than any external method on that

class. If an external method on a class is defined in a separate file, that file must appear first. These constraints are easily satisfied in a makefile:

```
end.maya:
        echo "use maya.multi.WeavingDone;" >$@

stamp: $(EXT_METHODS) $(CLASSES) end.maya
        mayac -use maya.multi.Weave $^
        touch stamp
```

The meta-program `maya.multi.WeavingDone` executes after all code has been compiled. It checks that all external methods have been woven into the appropriate classes and enforces MultiJava's modular type checking restrictions.

5.4.2 Weaving

When the weaver encounters an external method declaration, it associates the method with its receiver class name. The weaver also records the method's lexical environment, which allows the code described in Section 5.4.3 to implement MultiJava semantics. External method declarations appear at the top level, but are not translated to top-level declarations. The Mayans that implement external methods expand to an empty list of declarations.

The Mayan `EClass` in Figure 5.3 weaves method declarations into class bodies. `EClass` uses Maya's `nextRewrite` syntax to call the next applicable Mayan. Ar-

```
Declaration syntax
EClass(list(Modifier) mods class Identifier name SuperClass sup
       Implements impl lazy(BraceTree, list(BodyDecl)) body)
{
  if (Environment.current.enclosingClass() == null)
    return nextRewrite($mods class $name $sup $impl {
                         $(Weave.cdecl(name, body))
                       });
  else
    return nextRewrite();
}
```

Figure 5.3. Interposing on every class declaration

guments to `nextRewrite` are expanded as a template, and the result is passed to the next Mayan. If `nextRewrite` is called without arguments, the current Mayan's parameters are passed unmodified. When `EClass` encounters a top-level class declaration, it inserts external methods defined on that class into the class body. In particular, `EClass` passes a template thunk to the next applicable Mayan.

The call to `Weave.cdecl()` in Figure 5.3 will be evaluated when the class `name` is shaped. Since a class is shaped only after its file has been parsed, delayed execution allows the weaver to handle files where external methods appear after their classes in one file:

```
class C {
  void m1() { }
}
void C.m2() { }
```

Note that `EClass` does not match against expression types. In fact, `EClass` does not specialize any arguments. Maya dispatches to `EClass` in favor of the primitive `ClassDeclaration` builder based on lexical tie-breaking. Since the `ClassDeclaration` builder was imported first, it will be called last.

5.4.3 Enforcing MultiJava Semantics

MultiJava's type checking restrictions fall into two groups. Some restrictions apply to each external method independently, and enforce encapsulation. Other restrictions deal with the overall structure of the program and make separate type checking possible.

The weaver enforces local restrictions when it translates an external method declaration to an internal one. This code also ensures that type names are resolved using imports from the proper file.

Translating an external method declaration to an internal declaration involves wrapping each type in the method signature with an `ExternalTypeName` object. These wrappers ensure that top-level types are resolved properly, while allowing nonprivate member types of the augmented class to be used. Translating the method body is a bit more complicated.

Restrictions on external method bodies are enforced by local Mayans. These Mayans are able to access per-external method variables such as the environment in which a method is defined and the type into which it is weaved. These local Mayans are exported to an external method body by a statement-level `use` directive.

Global type checking restrictions are enforced after all files have been compiled. `WeavingDone` first checks that every external method declaration is associated with a `gnu.bytecode.Method` object, then checks MultiJava restrictions with Maya's introspection API.

5.5 Discussion

Maya has been used to implement several small macros and a large language extension. Both applications can be evaluated in terms of expressiveness, flexibility, and composability. These applications can also be evaluated in terms of coverage: are Maya's numerous features actually being used?

The simplest macros, such as `ArrayFromCollection` are as much as an order of magnitude larger than equivalent `cpp` macros. However, Maya offers the macro writer far greater flexibility and robustness. The `foreach` family of Mayans uses Maya's dispatching mechanisms to great advantage. The macros described in Section 5.1 can be used together without problems, since they are defined on disjoint syntax patterns.

In terms of line counts, Mayans compare far more favorably with hand-written language extensions. MultiJava has been implemented in Maya, and as a modified version of the Kopi Java compiler [10], `kjc`. The size of the Maya implementation is shown in Table 5.2. The Kopi implementation involved materially changing or adding 20,000 lines to a 50,000 line program [7]. By the line-count metric, using Maya saved somewhere between 87% and 92% of the work. It should be noted Clifton's MultiJava implementation involved fixing bugs and implementing standard Java features in `kjc`.

Another important measure of expressiveness is Maya's ability to implement MultiJava as specified. We have compromised in the definition of visible classes, but the compromise effects the way in which errors are reported rather than the

Table 5.2. Size of MultiJava code

FILE	LINES	COMNT	BLANK	PUNCT	CODE
UTILITIES:					
Debug.maya	19	2	4	1	12
IncludeHack.maya	14	3	3	2	6
Access.maya	9	0	2	2	5
PartialComparator.maya	18	0	3	3	12
TypeComparator.maya	17	0	2	3	12
MethodComparator.maya	36	4	5	9	18
ExternalMethRewriter.maya	19	6	2	2	9
ExternalMeth.maya	178	12	19	34	113
OPEN CLASS WEAVING:					
ErrorF.maya	22	0	5	4	13
ExternalGF.maya	150	28	14	21	87
Weave.maya	582	95	53	109	325
WeavingDone.maya	9	0	1	2	6
MULTIMETHODS:					
MultiMethodComparator.maya	29	0	4	3	22
GenericAttr.maya	120	10	14	23	73
SpecModifier.maya	19	2	3	3	11
MethodsModifier.maya	252	19	23	50	160
MultiMethod.maya	386	43	39	73	231
GenericFunction.maya	570	67	66	112	325
Methods.maya	45	9	6	6	24
TOTAL	2494	300	268	462	1464

language semantics. Other extensions cannot be implemented so easily: If MultiJava allowed covariant throws on external methods, a Maya implementation of the MultiJava compilation strategy would entail extra effort and bloat the generated code. GJ's [4] bridge methods cannot be expressed in Maya at all. To generate a class that contains two methods that differ only on return type, the Mayan author cannot use templates, but must generate Java byte-code by hand using the `gnu.bytecode.CodeAttr` class.

Composing the two features of MultiJava was not trivial. The weaver was implemented first, then multimethod support. To make full MultiJava work, the weaver was rewritten with knowledge of multimethods. Initially, external method declarations where reduced to `MethodDecl` objects on the spot. This caused `SpecModifiers`, which associate type specializers with declarations, to be interpreted, rather than filtered out by `MethodsModifers`' local Mayans. Multiple dispatch

also affects the way that external method visibility is checked. `MethodsModifier` transforms a public multimethod to a private hygienically named Java method. The weaver must not use `Method.getModifiers()` to check that this multimethod is public; instead it examines the `Modifier` syntax that appeared in the original declaration.

One of the trickiest issues in composing multimethods and open classes is sends to `super`. Within a multimethod, `super` is overloaded to call the next applicable method, and within an external method `super` sends are restricted to the current generic function. Consider the following example:

```
1  class C { }
2  class D extends C { }
3
4  void C.m(C c)   { }
5  void C.m(C@D c) { super.m(c); }
6  void D.m(C c)   { super.m(c); }
7  void D.m(C@D c) { super.m(c); }
```

The super send on line 5 is valid, even though `C` does not inherit a method `m`. This call invokes the base multimethod on line 4. The super send on line 7 is ambiguous. The weaver must allow this call, and the one on line 6 to proceed, even though neither is compiled into the dispatcher method `D.m()`.

In a certain respect, it is not surprising that multimethods and open classes cannot be blindly glued together. These extensions were developed in parallel and are somewhat interdependent. In general, one cannot expect reasonable semantics from blindly composing language extensions. For instance, a GJ-style parameterized type cannot sensibly be used as a multimethod parameter specializer. Some thought is always required to merge language extensions.

The macros and language extensions described in this chapter make use of different Maya features. Macros such as `ArrayFromCollection` and `ForEach` rely heavily on hygiene and referential transparency. Open classes and multimethods do not use these features at all. Of all the meta-programs described, only `ForEach` requires complex dispatching based on node structure, token values, and the static types of expressions.

The language extensions rely on inner Mayans to share state, and make heavy use of other language features such as `syntax case` and `nextRewrite`. multi-method code relies on the `ListIterator` returned by `TypeDeclaration.getMembers()` and the ability to extend `ExternalModifier`.

CHAPTER 6

RELATED WORK

6.1 Dispatch and Pattern Matching

Most multiple-dispatch systems allow dispatch on more than just types. For instance, CLOS [25] dispatches on argument values using `eql` specializers, and Dylan [24] generalizes this feature with singleton and union types. Maya's dispatching rules bear a particular resemblance to the grand unified dispatcher (GUD) in Dubious [14]. Like GUD, Maya uses pattern matching. GUD dispatches based on the structure of an argument and binds formal parameters to argument fields.

Dispatch in Maya does not correspond exactly to dispatch in Dubious. Maya can dispatch on the static types of expressions: it uses the class hierarchy of code being compiled to dispatch Mayans. Dubious can dispatch methods based on arbitrary boolean expressions, with a notion of overriding based on implication. Alghough this mechanism is more general than dispatch on static types, it is also less powerful: Mayans need not know static type relations a priori.

Languages such as Dubious and Cecil perform static checks to ensure that all generic function invocations unambiguously dispatch to a method. Maya defers these checks to expansion time.

Maya is certainly not the only syntax manipulation system to support pattern matching. Scheme's `syntax-rules` [19] and its descendants [13, 24] allow macros to be defined using case analysis. Whereas these systems do pattern matching over concrete syntax (s-expressions), Maya does pattern matching over well-structured AST nodes.

6.2 Macro Systems

Macro borrows many ideas from high-level macro systems such as Chez Scheme's `syntax-case`, Dylan macros, and XL [21]. However, Maya makes a different set of tradeoffs than these systems. Specifically, Maya's ability to statically check template syntax comes at the cost of perfect hygiene, and local Mayans can share state more easily than local macros.

Macro systems descended from Scheme's `syntax-rules` make hygiene decisions at expansion time, when the syntactic role of each identifier is known. All identifiers inside macro bodies are provisionally renamed. After the macro is expanded, the system resolves each reference to either a variable declared by the macro, in which case the renaming stands, or a variable shared by the macro and caller's environments, in which case the identifier must be un-renamed.

In Maya, the situation is different. Since templates are parsed at compile time, the syntactic role of each identifier is known before expansion takes place. Moreover, Maya's handling of type and variable names requires the syntactic role of each identifier to be known in order to parse a template. Before the pattern parser can reduce a literal identifier to either a `Reference` or a `StrictTypeName`, it must know whether and how the identifier is declared in the enclosing template.

Context sensitivity in Maya's grammar suggests a static approach to hygiene: The syntactic role of each identifier must be known to parse a template, just as it must be known to perform hygienic renaming. However, this approach places an extra burden on the Maya programmer. Mayans that map names to variable declarations must take `UnboundLocal` arguments rather than plain `Identifiers`. If Mayans do not follow this convention, hygiene breaks down when one Mayan is used in the expansion of another.

6.2.1 `syntax-case`

Chez Scheme's `syntax-case` provides programmable hygienic macros. This system offers many advantages over Scheme's `syntax-rules`. First, macros are defined by Scheme code rather than a pattern language. Second, macros can export names to their callers with violating hygiene. Finally, `syntax-case` optimizes hygienic

renaming by deferring work. Maya provides features not found in `syntax-case`, but where the two systems overlap, hygiene and local macro definitions, they make different decisions.

In `syntax-case` a macro can construct implicit parameters. These identifiers are hygienically renamed as if they appeared in the macro's caller, whether a macro is called from another macro or directly by the source program. Using implicit parameters, a macro and its calling macro may share names without violating hygiene. This technique relies on a dynamic implementation of hygiene, and is not implemented in Maya. Instead, a Mayan exposes names to its caller by breaking hygiene, and exposing the name all the way up the call chain.

In `syntax-case`, a macro can expose local macros to an argument by wrapping the argument in `let-syntax`. These macro definitions are local to the code being expanded and cannot easily share compile-time state. However, complex immutable data structures such as associative lists and vectors can be embedded in local macro definitions as literals. In contrast, local Mayans are objects that share state and can be imported into arbitrary environments. Since Java does not allow literals more complicated than a string, `syntax-case`'s formulation of local macros would be impractical for Java system.

6.2.2 Dylan

Dylan supports automatic hygiene, but we are not aware of any hygienic macro systems for Java or C++. Since Dylan macros are pattern-based, they are significantly less flexible than programmable macro systems such as `syntax-case` and MS^2 [27].

Dylan macros fall into one of four basic categories: block-structured definitions such as `define class`, short definitions such as `define variable`, statements such as `if`, and function-like macros. Dylan defines a single grammar production for each basic category, and recognizes macro keywords through lexical tie-ins.

Macro productions accumulate arguments in a matching-delimiter-based tree — in Dylan, the role of Maya's reader is subsumed by the LALR(1) parser. Macro

arguments are tokens, list fragments produced by matching delimiter parsing, other macro calls, and AST fragments parsed during pattern matching.

The macro expansion process itself is an extension of Scheme `syntax-rules`. Macros are expanded from the top down, using pattern substitution, and enforcing hygiene. The result of macro expansion is a sequence of tokens and other fragments, which is fed back into the LALR(1) parser.

Two aspects of hygiene in Dylan are of particular interest: The escape mechanism and slot-name handling. Within a macro, `?=` treats the following symbol as an implicit parameter, and `##` concatenates parameters and symbols to generate implicit parameter names much as in `cpp`. These features make the Dylan macro system more expressive than its ancestor, Scheme `syntax-rules`.

Maya is like Dylan in that neither hygienically renames fields. In Dylan, the dotted name `a-point.x` is alternate syntax for `x(a-point)`. The definition

```
define class <point> (<object>)
  slot x
  slot y
end class
```

must appear at the top level. In addition to defining the class `<point>`, this form ensures that generic functions `x`, `x-setter`, `y`, and `y-setter` exist, and defines slot accessors as methods on these functions. Since the names introduced by define-class are module-scoped, hygiene dictates that that they are not renamed.

6.2.3 XL

XL [21] is a macro system for a strongly typed version of Scheme. XL exposes the typing environment to macros, allowing them to perform the kind of explicit type-checking used in Section 5.1. Unlike OpenJava and Maya, XL does not use static types to dispatch macros.

6.2.4 MS^2

MS^2 [27] is essentially Common Lisp's `defmacro` for C. Macro functions are evaluated by a C interpreter and may expand declarations, statements, or expressions.

As in Dylan, macro keywords are recognized by the lexer, and macro parameters may be recursively parsed according to their types.

MS^2 defines template syntax for building ASTs, and supports a polymorphic unquote operator, `$`. Maya shares these features. When parsing templates, MS^2 type-checks `$` expressions to determine the grammar symbols they produce. In MS^2, templates can only generate declarations, statements, and expressions, but three additional node types can be unquoted: identifiers, numeric literals, and type specifiers. MS^2's recursive-descent parser is written to accept macro calls and unquoted expressions at these few well-defined places.

6.3 Compiler Toolkits

Systems such as A* and JTS provide language extensions and library procedures to manipulate syntax trees. A* extends Awk to operate on a particular source language, while JTS is a set of composable extensions to Java that is used to write composable extensions to Java. Unlike Maya and open compilers, these systems do not use semantic information such as types to guide translation.

6.3.1 A*

A* is a family of languages based on Awk: `ac` is a variant on Awk that operates on C syntax trees, `aawk` is variant of Awk that operates on Awk syntax trees, and `aprl` is a variant of Awk that operate on the syntax trees of a language called PRL5. A* can be used to implement a variety of source code analysis and translation tools.

A* supports a limited form of pattern matching: a case guard may be either boolean expression or a YACC production. A production case matches nodes generated by the corresponding production in A*'s parser, and introduces new syntax for accessing the node's children. Unlike in Maya, A* patterns cannot contain substructure or additional attributes such as identifier values or expression types.

6.3.2 JTS

JTS [2] is a framework for writing Java preprocessors. JTS and Maya come at the problem of extensible languages from opposite directions, and make different tradeoffs between flexibility and expressiveness. Whereas Maya can be used to implement macros, it is impractical to define a simple macro by building a JTS preprocessor.

JTS language extensions can define new productions and new node types. JTS even allows extensions to define arbitrary nonterminals. Although abstract Mayans define productions, Maya severely restricts the definition of node types: only two types, `LazyTypeName` and `ExternalModifier`, may be subclassed. In Maya, each node in the syntax tree must denote a well-typed Java program fragment, but JTS has a much weaker requirement: the Java code a preprocessor eventually generates must be well-typed.

JTS's model for syntax rewriting is more flexible than Maya's. A JTS extension can mutate syntax trees in arbitrary ways, whereas Maya severely restricts mutation. Maya allows members to be added to or removed from a class declaration, provided that certain dynamic conditions are met. This restriction ensures that class updates cannot change the results of previous type checking and Mayan dispatch. JTS avoids these thorny issues, since type checking is left to `javac`.

Like MS^2, JTS has a limited template facility. JTS defines 17 AST constructor expressions and 17 distinct unquote forms. Each of these 34 syntactic forms is defined by its own grammar rule. These tree constructors suffer from some limitations. They do not support automatic hygiene and referential transparency. By default, user-defined syntax cannot appear in tree constructors: any AST node that appears in a tree constructor must implement a method `reduce2ast` that builds the corresponding Java `new` expression.

6.3.3 Micros

Maya resembles MzScheme's McMicMac facility [20] in that both allow meta-programs to expand syntax to an intermediate format. McMicMac implements macros, which translate source to source, in terms of micros. A macro is translated

to a micro that generates and recursively expands a syntax tree. Like micros, Mayans return an intermediate format, abstract syntax trees. Templates provide a macro-like facility: evaluating a template fully expands the concrete syntax in its body.

6.4 Compile-time MOPs

Maya shares many mechanisms with high-level macro systems, such as hygienic declaration handling and pattern matching. However, Maya is more closely related to compile-time meta-object protocols such as OpenC++ [6] and OpenJava [26], in that meta-programs can override the translation of built-in syntax based on static type information. This section provides a detailed comparison of Maya and OpenJava.

6.4.1 Overview

Both OpenJava and Maya implement introspection APIs based on that of `java.lang.Class`. OpenJava extends this API with the visitor pattern. OpenJava calls a metaclass's expand methods to rewrite expressions involving that metaclass's associated classes. Each metaclass inherits introspection methods, and may override expand methods.

In OpenJava, a class is associated with a metaclass through the `instantiates` clause, which appears after `implements` in a class declaration. A metaclass may extend OpenJava syntax in three ways. First, it can define a recursive descent parser to accept options in an `instantiates` clause. Second, it can define a recursive descent parser to accept options following an instance class name. Finally, it can define modifiers visible in an instance class body.

OpenJava provides for two kinds of expansion: *callee-side* translation allows a metaclass to modify instance class declarations, and *caller-side* translation allows a metaclass to expand expressions involving its instance types. OpenJava orders expansion such that callee-side translation occurs before caller-side, and that callee-side translation of a superclass precedes that of a subclass. OpenJava also provides the `waitTranslation()` method to order callee-side translation of unrelated classes.

Cyclic calls to `waitTranslation()` throw an exception.

6.4.2 User-Defined Syntax

The primary difference between Maya and OpenJava lies in a Mayan's ability to define arbitrary syntax. In Maya, new syntax is defined through abstract Mayans. User-defined abstract Mayans are indistinguishable from the abstract Mayans that form Maya's base grammar. A concrete Mayan may be defined on any syntax, rather than the subset of expressions and declaration options that OpenJava supports.

6.4.3 Dispatch

Multiple dispatch allows Mayans to specialize based on a wider array of static types than OpenJava metaclasses. Mayan parameters can be specialized on primitive types, array types, and even `Object`. In OpenJava, the behavior of every type could only be modified by modifying the standard metaclass, `OJClass`.

Open classes, as described in Chapter 5, also demonstrate Maya's flexibility. External method dispatch is caller-side transformation, but on callers of the receiver, rather than callers of the external method.

OpenJava and OpenC++ share one feature that Maya lacks: a mechanism for the base program to explicitly associate meta-programs with types. This feature could be easily implemented in Maya.

6.4.4 Expansion Order

Both Maya and OpenJava provide subtle mechanisms for controlling the order of expansion. Although these mechanisms are not strictly comparable, Maya's mechanism simplifies the implementation of certain real meta-programs such as multimethods.

Maya does not distinguish between caller-side and callee-side translation. Instead, the expansion order is controlled by class states and laziness. Like OpenJava, Maya expands supertype member declarations before subtypes and the member declarations of a class before expressions on that class. However, a Mayan defined

on a method declaration can delay processing by moving it to the method body.

Consider multimethods. Their implementation is certainly a callee-side translation, since specialized method declarations must be gathered into generic functions and dispatch methods must be generated. However, ambiguous generic functions must be detected during callee-side translation: A method can introduce ambiguity if it specializes an argument on a subclass of the receiver. Maya's multi-method implementation performs these checks while generating the dispatch method, and uses laziness to ensure that full type information is available. It is unclear how to use OpenJava's callee-side methods to achieve the same effect.

6.4.5 Safety

Maya also includes amenities to make meta-programming more pleasant. OpenJava does not guarantee syntactic safety [27]. Its macros can generate illegal pieces of syntax, because they allow meta-programs to convert arbitrary strings to syntax. Mayans, on the other hand, must always generate correct syntax. In addition, OpenJava metaclasses inspect nodes through accessor methods, whereas Mayans inspect nodes through pattern matching. OpenJava also allows class members to be updated after member types have been used to expand other code. Finally, OpenJava seems to ignore the technology, hygiene and referential transparency, that makes macros work [9].

CHAPTER 7

CONCLUSIONS

Maya is an extension to Java that supports compile-time meta-programming. Mayans can extend the language syntax and override the behavior of base language syntax. The compiler exposes static type information as well as syntactic structure to meta-programs.

Maya borrows features such as hygiene, referential transparency and pattern matching from high-level macro systems, and uses multiple dispatch. These features are implemented with two novel techniques: pattern parsing and static hygienic renaming.

Mayans can be used for a wide range of applications, from the kind of simple macros found in C programs to general-purpose language extensions. Mayans are probably most useful for tasks somewhere between these extremes of complexity.

7.1 Language Features

Mayans expand abstract syntax to other abstract syntax through templates. These templates are akin to Lisp's backquote operator, but provide hygiene and referential transparency. Maya also provides mechanisms for breaking hygiene and for generating unique names for entities that are not lexically scoped.

Mayans are called from the parser based on multiple dispatch rules. Argument specializers can include substructure, which allows Mayan dispatch to act as pattern matching, and the static semantic types of expressions, which allows Mayans to be coupled with base-level classes.

Mayans are called by the LALR(1) parser to generate syntax trees. Whereas macro systems typically expand programs from the top down, Maya expands programs from the bottom-up. Mayans can override most productions in the Maya

grammar, and can override new productions defined by abstract Mayan declarations.

Maya's model of bottom up expansion based on static type information is made possible by lazy parsing. The Mayan used to reduce a production may depend on the static types of subexpressions, which cannot be computed in a single pass. Maya uses laziness parse each expression when its environment is known. Mayans interact with lazy parsing in two ways. First, lazy templates can be used to delay computation. Second, `ExternalModifier` objects provide a hook into declaration processing.

7.2 Implementation Techniques

Maya uses a novel technique to parse streams of both terminal and nonterminal inputs. Pattern parsing localizes the grammar support for argument trees and templates, whereas other techniques require unquote place-holders to be defined throughout the grammar. The pattern parser makes Maya's templates more flexible than other systems, which typically allow only subset of AST nodes to be unquoted or generated as the root of a template.

Maya also implements hygiene in a unique way. Hygiene and referential transparency decisions are made statically, by examining the template's parse tree. This approach allows Maya to statically detect unbound variable references in templates, but comes at a cost. Unlike `syntax-case` and `Dylan` macros, Mayans cannot take implicit parameters. Additionally, abstract Mayans such as `let` on page 40 must be written carefully to preserve hygiene when the expansion of one Mayan invokes another.

7.3 Expressiveness

Maya is not the only Java extension that exposes static type information to macros. OpenJava provides more limited functionality through metaclasses. Maya allows meta-programs to define a wider range new syntax and provides a more expressive model of expansion. Mayans can change the behavior of a source program without any changes to the program text, and Mayan expansion can be tied to a

wider variety of types including primitives and array types.

Maya can be used for a wide range of tasks. The sort of macros typically written in `cpp` can be implemented in Maya, but less succinctly. General purpose language extensions can be implemented in Maya, but it is unclear whether Maya's restrictions on evaluation order make it worthwhile.

Maya is not a simple language. To define a new syntactic form, the programmer must add a production to the LALR(1) grammar. Maya's model of lazy parsing and type-checking can be subtle as well. It remains to be seen whether Maya is simple enough to be usable.

APPENDIX A

GRAMMAR

A.1 Files

```
abstract CompilationUnit syntax(list(ImportDecl) SimpleDeclList);
abstract CompilationUnit syntax(PkgDecl list(ImportDecl) SimpleDeclList);
abstract PkgDecl syntax(package QualifiedName ;);
abstract ImportDecl syntax(import QualifiedName ;);
abstract ImportDecl syntax(import QualifiedName . * ;);
```

A.2 Declarations

```
abstract Declaration syntax(list(Modifier) LazyTypeName syntax
                            Identifier (list(BindingSymbol))
                            lazy(BraceTree, BlockStmts));
abstract Declaration syntax(list(Modifier) LazyTypeName syntax
                            (list(AbstractSymbol))
                            WithClause ; lazy({ EOF }, SimpleDeclList));
abstract Declaration syntax(use LazyTypeName ; lazy({ EOF }, SimpleDeclList));
abstract Declaration syntax(list(Modifier) interface Identifier SuperIfs
                            lazy(BraceTree, SimpleDeclList));
abstract Declaration syntax(list(Modifier) class Identifier SuperClass
                            Implements lazy(BraceTree, SimpleDeclList));
abstract Declaration syntax(list(Modifier) Identifier (FormalList) Throws
                            lazy(BraceTree, CtorBody));
abstract Declaration syntax(list(Modifier) ReturnVoid Identifier (FormalList)
                            Throws lazy(BraceTree, BlockStmts));
abstract Declaration syntax(list(Modifier) LazyTypeName Identifier (FormalList)
                            Throws lazy(BraceTree, BlockStmts));
abstract Declaration syntax(list(Modifier) ReturnVoid Identifier (FormalList)
                            Throws ;);
abstract Declaration syntax(list(Modifier) LazyTypeName Identifier (FormalList)
                            Throws ;);
abstract Declaration syntax(lazy(BraceTree, BlockStmts));
abstract Declaration syntax(BuiltinModifier lazy(BraceTree, BlockStmts));
abstract Declaration syntax(;);
abstract Declaration syntax(FieldDeclList);
abstract Modifier syntax(BuiltinModifier);
abstract LazyTypeName syntax(LazyTypeName [Empty]);
abstract LazyTypeName syntax(QualifiedName);
abstract LazyTypeName syntax(PrimTypeName);
abstract QualifiedName syntax(Identifier)
```

```
  with precedence = PRIMARY + 1;
abstract QualifiedName syntax(QualifiedName . Identifier)
  with precedence = PRIMARY + 1;
abstract WithClause syntax();
abstract WithClause syntax(Identifier nonEmptyList(ProdClause, ‘,‘));
abstract ProdClause syntax(associativity = Identifier);
abstract ProdClause syntax(precedence = lazy({ , ; }, Expression));
abstract SimpleDeclList syntax(list(Declaration));
abstract SuperClass syntax();
abstract SuperClass syntax(extends LazyTypeName);
abstract Implements syntax();
abstract Implements syntax(implements nonEmptyList(LazyTypeName, ‘,‘));
abstract SuperIfs syntax();
abstract SuperIfs syntax(extends nonEmptyList(LazyTypeName, ‘,‘));
abstract ReturnVoid syntax(void);
abstract FormalList syntax(list(Formal, ‘,‘));
abstract Formal syntax(list(Modifier) LazyTypeName LocalDeclarator);
abstract Throws syntax();
abstract Throws syntax(throws nonEmptyList(LazyTypeName, ‘,‘));
abstract FieldDeclList syntax(list(Modifier) LazyTypeName
                              nonEmptyList(FieldDeclarator, ‘,‘) ;);
abstract FieldDeclarator syntax(FieldDeclaratorId);
abstract FieldDeclarator syntax(FieldDeclaratorId
                                = lazy({ , ; EOF },  VarInitializer));
abstract FieldDeclaratorId syntax(Identifier);
abstract FieldDeclaratorId syntax(FieldDeclaratorId [Empty]);
abstract LocalDeclList syntax(StrictTypeName
                              nonEmptyList(LocalDeclarator, ‘,‘);););
abstract LocalDeclList syntax(nonEmptyList(Modifier) StrictTypeName
                              nonEmptyList(LocalDeclarator, ‘,‘) ;);
abstract LocalDeclarator syntax(LocalDeclaratorId);
abstract LocalDeclarator syntax(LocalDeclaratorId
                                = lazy({ , ; EOF }, VarInitializer));
abstract LocalDeclaratorId syntax(UnboundLocal);
abstract LocalDeclaratorId syntax(LocalDeclaratorId [Empty]);
abstract VarInitializer syntax({VarInitList});
abstract VarInitializer syntax(Expression);
abstract VarInitList syntax();
abstract VarInitList syntax(VarInitList0);
abstract VarInitList syntax(,);
abstract VarInitList syntax(VarInitList0 ,);
abstract VarInitList0 syntax(VarInitializer);
abstract VarInitList0 syntax(VarInitList0 , VarInitializer);
abstract LocalMayanDecl syntax(StrictScalarType syntax UnboundLClass
                               (list(BindingSymbol))
                               lazy(BraceTree, BlockStmts));
abstract LocalMayanDecl syntax(nonEmptyList(Modifier) StrictScalarType syntax
                               UnboundLClass (list(BindingSymbol))
                               lazy(BraceTree, BlockStmts));
abstract LocalClassDeclaration syntax(class UnboundLClass SuperClass Implements
                                      lazy(BraceTree, SimpleDeclList));
abstract LocalClassDeclaration syntax(nonEmptyList(Modifier) class
```

```
                                    UnboundLClass SuperClass Implements
                                    lazy(BraceTree, SimpleDeclList));
abstract UnboundLocal syntax(Identifier) with precedence = STATEMENT + 1;
abstract UnboundLClass syntax(Identifier);
abstract Dim syntax([Empty]);
abstract Empty syntax();
abstract ClosedDims syntax([Expression]);
abstract ClosedDims syntax(ClosedDims [Expression]);
abstract AbstractSymbol syntax(\ DottedName);
abstract AbstractSymbol syntax(\ Token);
abstract AbstractSymbol syntax(Token);
abstract AbstractSymbol syntax({list(AbstractSymbol)});
abstract AbstractSymbol syntax([list(AbstractSymbol)]);
abstract AbstractSymbol syntax((list(AbstractSymbol)));
abstract AbstractSymbol syntax(NtSpec);
abstract BindingSymbol syntax(\ DottedName);
abstract BindingSymbol syntax(\ Token);
abstract BindingSymbol syntax(Token);
abstract BindingSymbol syntax({list(BindingSymbol)});
abstract BindingSymbol syntax([list(BindingSymbol)]);
abstract BindingSymbol syntax((list(BindingSymbol)));
abstract BindingSymbol syntax(NtSpec);
abstract BindingSymbol syntax(NtSpec < LazyTypeName > : LazyTypeName
                              UnboundLocal);
abstract BindingSymbol syntax(NtSpec : LazyTypeName UnboundLocal);
abstract BindingSymbol syntax(NtSpec < LazyTypeName > UnboundLocal);
abstract BindingSymbol syntax(NtSpec UnboundLocal);
abstract NtSpec syntax(DottedName);
abstract NtSpec syntax(PsymSpec);
abstract DottedName syntax(QualifiedName);
abstract PsymSpec syntax(nonEmptyList (ListParams));
abstract PsymSpec syntax(list (ListParams));
abstract PsymSpec syntax(choice (nonEmptyList(ChoiceRhs, ‘,‘)));
abstract PsymSpec syntax(lazy (LazyParams));
abstract ListParams syntax(NtSpec , ‘ Token ‘);
abstract ListParams syntax(NtSpec);
abstract ChoiceRhs syntax([list(AbstractSymbol)]);
abstract LazyParams syntax({list(Token)} , NtSpec);
abstract LazyParams syntax(LazyTypeName , NtSpec);
```

A.3 Statements

```
abstract Statement syntax(syntax case ImplicitLabel (UnquoteExpr)
                          { nonEmptyList(CaseClause) });
abstract Statement syntax(use StrictTypeName ; lazy({ EOF }, StmtList));
abstract Statement syntax(throw Expression ;);
abstract Statement syntax(return ;);
abstract Statement syntax(return Expression ;);
abstract Statement syntax(synchronized (Expression)
                          lazy(BraceTree, BlockStmts));
abstract Statement syntax(try lazy(BraceTree, BlockStmts) list(CatchClause));
abstract Statement syntax(try lazy(BraceTree, BlockStmts) list(CatchClause)
```

```
                          FinallyClause);
abstract Statement syntax(switch ImplicitLabel (Expression) {SwitchBody});
abstract Statement syntax(do ImplicitLabel Statement while (Expression) ;);
abstract Statement syntax(while (Expression) ImplicitLabel Statement);
abstract Statement syntax(for ImplicitLabel (ForHeader) Statement);
abstract Statement syntax(if (Expression) Statement)
  with precedence = STATEMENT + 1, right = true;
abstract Statement syntax(if (Expression) Statement else Statement)
  with precedence = STATEMENT + 1, right = true;
abstract Statement syntax(lazy(BraceTree, BlockStmts));
abstract Statement syntax(;);
abstract Statement syntax(Expression ;);
abstract Statement syntax(continue LabelName ;);
abstract Statement syntax(continue ;);
abstract Statement syntax(break LabelName ;);
abstract Statement syntax(break ;);
abstract Statement syntax(ExplicitLabel Statement);
abstract Statement syntax(DeclStmt);
abstract ImplicitLabel syntax();
abstract CaseClause syntax(default lazy(BraceTree, BlockStmts));
abstract CaseClause syntax(on (list(BindingSymbol))
                           lazy(BraceTree, BlockStmts));
abstract CatchClause syntax(catch (Formal) lazy(BraceTree, BlockStmts));
abstract FinallyClause syntax(finally lazy(BraceTree, BlockStmts));
abstract SwitchBody syntax(SwitchGroups);
abstract SwitchBody syntax(SwitchGroups nonEmptyList(SwitchLabel));
abstract SwitchGroups syntax(SwitchGroup);
abstract SwitchGroups syntax(SwitchGroups SwitchGroup);
abstract SwitchGroup syntax(nonEmptyList(SwitchLabel) Stmts);
abstract SwitchLabel syntax(default :);
abstract SwitchLabel syntax(case Expression :);
abstract ForHeader syntax(list(Expression, ‘,‘) ; ; list(Expression, ‘,‘));
abstract ForHeader syntax(list(Expression, ‘,‘) ; Expression ;
                          list(Expression, ‘,‘));
abstract ForHeader syntax(DeclStmt ; list(Expression, ‘,‘));
abstract ForHeader syntax(DeclStmt Expression ; list(Expression, ‘,‘));
abstract LabelName syntax(Identifier);
abstract ExplicitLabel syntax(UnboundLabel :);
abstract UnboundLabel syntax(Identifier);
abstract DeclStmt syntax(LocalMayanDecl);
abstract DeclStmt syntax(LocalClassDeclaration);
abstract DeclStmt syntax(LocalDeclList);
abstract CtorBody syntax(StmtList);
abstract CtorBody syntax(CtorCall StmtList);
abstract CtorCall syntax(StrictName . super (list(Expression, ‘,‘)) ;)
  with precedence = PRIMARY;
abstract CtorCall syntax(Expression . super (list(Expression, ‘,‘)) ;)
  with precedence = PRIMARY;
abstract CtorCall syntax(super (list(Expression, ‘,‘)) ;)
  with precedence = PRIMARY;
abstract CtorCall syntax(this (list(Expression, ‘,‘)) ;)
  with precedence = PRIMARY;
```

```
abstract BlockStmts syntax(StmtList);
abstract StmtList syntax();
abstract StmtList syntax(Stmts);
abstract Stmts syntax(Statement);
abstract Stmts syntax(Stmts Statement);
```

A.4 Expressions

```
abstract Expression syntax(new PsymSpec lazy(BraceTree, list(TemplateSymbol)));
abstract Expression syntax(new StrictScalarType
                           lazy(BraceTree, list(TemplateSymbol)));
abstract Expression syntax(nextRewrite lazy(ParenTree, list(TemplateSymbol)));
abstract Expression syntax((lazy(BraceTree, BeginBody)));
abstract Expression syntax(new StrictArrayType {VarInitList});
abstract Expression syntax(Expression = Expression)
  with precedence = ASSIGN, right = true;
abstract Expression syntax(Expression ? Expression : Expression)
  with precedence = CONDITIONAL, right = true;
abstract Expression syntax(Expression | Expression)
  with precedence = BITOR;
abstract Expression syntax(Expression ^ Expression)
  with precedence = XOR;
abstract Expression syntax(Expression & Expression)
  with precedence = BITAND;
abstract Expression syntax(Expression != Expression)
  with precedence = EQUALITY;
abstract Expression syntax(Expression == Expression)
  with precedence = EQUALITY;
abstract Expression syntax(Expression instanceof StrictTypeName)
  with precedence = RELATIONAL;
abstract Expression syntax(Expression <= Expression)
  with precedence = RELATIONAL;
abstract Expression syntax(Expression > Expression)
  with precedence = RELATIONAL;
abstract Expression syntax(Expression >= Expression)
  with precedence = RELATIONAL;
abstract Expression syntax(Expression < Expression)
  with precedence = RELATIONAL;
abstract Expression syntax(Expression >>> Expression)
  with precedence = SHIFT;
abstract Expression syntax(Expression >> Expression)
  with precedence = SHIFT;
abstract Expression syntax(Expression << Expression)
  with precedence = SHIFT;
abstract Expression syntax(Expression / Expression)
  with precedence = MUL;
abstract Expression syntax(Expression % Expression)
  with precedence = MUL;
abstract Expression syntax(Expression * Expression)
  with precedence = MUL;
abstract Expression syntax(Expression - Expression)
  with precedence = ADD;
```

```
abstract Expression syntax(Expression + Expression)
  with precedence = ADD;
abstract Expression syntax(+ Expression)
  with precedence = PREFIX;
abstract Expression syntax(- Expression)
  with precedence = PREFIX;
abstract Expression syntax((StrictTypeName) Expression)
  with precedence = PREFIX;
abstract Expression syntax(StrictName . new Identifier (list(Expression, ‘,‘)))
  with precedence = PRIMARY;
abstract Expression syntax(Expression . new Identifier (list(Expression, ‘,‘)))
  with precedence = PRIMARY;
abstract Expression syntax(new StrictScalarType (list(Expression, ‘,‘)))
  with precedence = PRIMARY;
abstract Expression syntax(MethodName (list(Expression, ‘,‘)))
  with precedence = PRIMARY;
abstract Expression syntax(StrictName . this)
  with precedence = PRIMARY;
abstract Expression syntax(this);
abstract Expression syntax(Literal);
abstract Expression syntax((Expression));
abstract Expression syntax(Expression [Expression:int])
  with precedence = ARRAY_REF;
abstract Expression syntax(new StrictScalarType ClosedDims list(Dim))
  with precedence = PRIMARY;
abstract Expression syntax(Expression . Identifier)
  with precedence = PRIMARY;
abstract Expression syntax(StrictName)
  with precedence = POSTFIX;
abstract BeginBody syntax(Stmts Expression ;);
abstract StrictName syntax(StrictName . Identifier) with precedence = PRIMARY;
abstract StrictName syntax(Identifier) with precedence = PRIMARY;
abstract StrictTypeName syntax(StrictArrayType);
abstract StrictTypeName syntax(StrictScalarType);
abstract StrictArrayType syntax(StrictScalarType nonEmptyList(Dim));
abstract StrictScalarType syntax(StrictName);
abstract StrictScalarType syntax(PrimTypeName);
abstract MethodName syntax(Identifier)
  with precedence = PRIMARY;
abstract MethodName syntax(StrictName . Identifier)
  with precedence = PRIMARY;
abstract MethodName syntax(StrictName super . Identifier)
  with precedence = PRIMARY;
abstract MethodName syntax(super . Identifier)
  with precedence = PRIMARY;
abstract MethodName syntax(Expression . Identifier)
  with precedence = PRIMARY;
abstract TemplateSymbol syntax(\ \);
abstract TemplateSymbol syntax(\ $);
abstract TemplateSymbol syntax($ lazy(ParenTree, UnquoteExpr));
abstract TemplateSymbol syntax(Token);
abstract TemplateSymbol syntax({list(TemplateSymbol)});
```

```
abstract TemplateSymbol syntax([list(TemplateSymbol)]);
abstract TemplateSymbol syntax((list(TemplateSymbol)));
abstract TemplateSymbol syntax(DottedName);
abstract UnquoteExpr syntax(Expression);
abstract UnquoteExpr syntax(Identifier NtSpec Expression);
```

APPENDIX B

LIBRARY CODE

B.1 ArrayFromCollection

```
package maya.util;
import maya.tree.*;
import maya.grammar.*;
import java.util.*;
import gnu.bytecode.*;

/*
 * This particular constructor only needs one argument, but why limit
 * oneself.
 */
abstract Expression syntax(new StrictArrayType (list(Expression, ‘,‘)))
  with precedence = PRIMARY;

/*
 * I am sick and tired of copying a collection into an array in two
 * lines!
 */
public Expression syntax
ArrayFromCollection(new StrictArrayType t(Expression:Collection c))
{
  final Type inner = t.toType().getComponentType();

  return new Expression {
    ({
      $(StrictTypeName.make(c.getStaticType())) c = $(c);
      $(t) ret = $(new ArrayNewExpr(inner, new ClosedDims { [c.size()] }));
      c.copyInto(ret);
      ret;
    })
  };
}
```

B.2 Assert

```
package maya.util;
import maya.tree.*;
import maya.grammar.*;
```

```
use Syntax;
use ForEach;

abstract Statement syntax (assert(Expression););

public class Assert implements MetaProgram {
  void checkEffects(Node parent)
  {
    if (parent instanceof SideEffectExpr)
      throw parent.error("Side effects in assertion");
    parent.getChildren().foreach(Node k) {
      if (k != null) checkEffects(k);
    }
  }

  public Environment run(Environment env)
  {
    Statement syntax A(assert(Expression e);) {
      checkEffects(e);
      return new Statement {
        if (!$e)
          throw new Error("Assertion failed");
      };
    }

    return new A().run(env);
  }
}
```

B.3 Debug

```
package maya.util;
import maya.tree.*;
import maya.grammar.*;
import kawa.lang.*;                // XXX
import gnu.bytecode.Type;

use Syntax;
use RunMayans;

abstract Statement syntax(debug(Identifier, list(Expression, ‘,‘)););

/*
 * Print a string to System.err, if a static field in the class
 * ‘Debug’ is set.  We look up Debug according to the current module’s
 * imports, but typically it should be a class in the home package.
 */
public class Debug implements MetaProgram {
  Statement checkFlag(Identifier name, Expression then)
  {
    Pair qname = new Pair(new Identifier(null, "Debug"), null);
    Type type = Environment.current.lookupType(qname, false);
```

```
    StrictTypeName t = StrictTypeName.make(type);

    return new Statement {
      if ($(as StrictName (StrictClassName) t).$name)
        $then;
    };
  }

  public Environment run(Environment env)
  {
    /*
     * With two args, the second is passed to System.err.println
     * directly.
     */
    Statement syntax
    DebugExp(debug(Identifier name, Expression exp);) {
      return checkFlag(name,
                       new Expression { System.err.println($exp) });
    }

    /*
     * With n != 2 args, the second argument is treated as a format
     * specifier.
     */
    Statement syntax
    DebugF(debug(Identifier name, list(Expression, ‘,‘) fmt);) {
      return checkFlag(name,
                       new Expression { System.err.printf($fmt) });
    }

    return runMayans { env, DebugExp, DebugF };
  }
}
```

B.4 NDebug

```
package maya.util;
import maya.tree.*;
import maya.grammar.*;

use RunMayans;

abstract Statement syntax(debug(Identifier, list(Expression, ‘,‘));););
abstract Statement syntax(assert(Expression););

public class NDebug implements MetaProgram {
  public Environment run(Environment env) {
    Statement syntax
    ND(debug(Identifier _, list(Expression, ‘,‘) __);) {
      return new Statement { ; };
    }
```

```
    Statement syntax NA(assert(Expression _);) {
      return new Statement { ; };
    }

    return runMayans { env, ND, NA };
  }
}
```

REFERENCES

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Batory, D., Lofaso, B., and Smaragdakis, Y. JTS: Tools for implementing domain-specific languages. In *5th International Conference on Software Reuse* (1998).

[3] Bothner, P. Kawa—compiling dynamic languages to the Java VM. In *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track* (New Orleans, LA, June 1998), USENIX Association.

[4] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. Making the future safe for the past: adding genericity to the java programming language. In *Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications* (1998), pp. 183–200.

[5] Chambers, C. *The Cecil Language Specification and Rationale: Version 2.0*, 1995.

[6] Chiba, S. A metaobject protocol for C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '95* (1995), pp. 285–299.

[7] Clifton, C. The MultiJava project. `http://www.cs.iastate.-edu/~cclifton/multijava/index.shtml`.

[8] Clifton, C., Leavens, G., Chambers, C., and Millstein, T. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '00* (Minneapolis, MN, Oct. 2000), pp. 130–146.

[9] Clinger, W. Macros that work. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages* (1991), pp. 155–162.

[10] Divoky, W., Forgione, C., Graf, T., Laborde, C., Lemonnier, A., and Wais, E. The Kopi project. http://www.dms.at/kopi/index.html.

[11] Docournau, R., Habib, M., Huchard, M., and Mugnier, M. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '92* (1992), pp. 16–24.

[12] Donnelly, C., and Stallman, R. Bison — the YACC-compatible parser generator. `http://www.gnu.org/software/bison/bison.html`.

[13] Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (1993), pp. 295–326.

[14] Ernst, M., Kaplan, C., and Chambers, C. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the European Conference on Object-Oriented Programming* (1998), pp. 186–211.

[15] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[16] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[17] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification*, second ed. The Java Series. Addison-Wesley, 2000.

[18] Johnson, M. LALR: LALR parser generator for Scheme. `http://-www.cs.cmu.edu/afs/cs/project/airepository/ai/lang/scheme/code/-parsing/lalr/`.

[19] Kelsey, R., Clinger, W., and J. Rees (Eds.). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998).

[20] Krishnamurthi, S. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

[21] Maddox, W. Semantically-sensitive macroprocessing. Master's thesis, University of California, Berkeley, 1989.

[22] Microsoft. C# language specification. `http://msdn.microsoft.com/-library/dotnet/csspec/vclrfcsharpspec_Start.htm`.

[23] Millstein, T., 2001. private communications.

[24] Shalit, A. *Dylan Reference Manual*. Addison-Wesley, 1996.

[25] Steele Jr., G. *Common Lisp, the Language*, second ed. Digital Press, 1990.

[26] Tatsubori, M., Chiba, S., Killijian, M., and Itano, K. *Reflection and Software Engineering*, vol. 1826 of *Lecture Notes in Computer Science*. Springer Verlag, 2000, ch. OpenJava: A Class-based Macro System for Java.

[27] Weise, D., and Crew, R. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation* (Albuquerque, NM, June 1993), pp. 156–165.

[28] Xerox. The AspectJ primer. http://www.aspectj.org/doc/primer/.