Maya: Multiple-Dispatch Syntax Extension in Java

Jason Baker and Wilson C. Hsieh
University of Utah

ABSTRACT

We have designed and implemented Maya, a version of Java that allows programmers to extend and reinterpret its syntax. Maya generalizes macro systems by treating grammar productions as generic functions, and semantic actions on productions as multimethods on the corresponding generic functions. Programmers can write new generic functions (i.e., grammar productions) and new multimethods (i.e., semantic actions), through which they can extend the grammar of the language and change the semantics of its syntactic constructs, respectively. Maya's multimethods are compile-time metaprograms that transform abstract syntax: they execute at program compile-time, because they are semantic actions executed by the parser. Maya's multimethods can be dispatched on the syntactic structure of the input, as well as the static, source-level types of expressions in the input.

In this paper we describe what Maya can do and how it works. We describe how its novel parsing techniques work and how Maya can statically detect certain kinds of errors, such as code that generates references to free variables. Finally, to demonstrate Maya's expressiveness, we describe how Maya can be used to implement the MultiJava language, which was described by Clifton et al. at OOPSLA 2000.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Java, metaprogramming, macros, generative programming

1. INTRODUCTION

Syntax extension can be used to embed a domain-specific language within an existing language. For example, embedded SQL extends its host language with database query syntax. Syntax extension can also be used to add language features when they are found to be necessary. For example, design patterns [19] can be viewed as work-arounds for specialized features missing from general-purpose languages: the visitor pattern implements multiple dispatch in a single-dispatch language. Some language designers have chosen to specialize their languages to support certain patterns: the C# [25] language includes explicit support for the state/observer pattern and delegation. However, unless we are willing to wait for a new language each time a new design pattern is identified, such an approach is unsatisfactory. Instead, a language should admit programmer-defined syntax extensions.

Macro systems [13, 27, 30] 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 an extremely powerful macro system, a macro cannot redefine the procedure application syntax.

Other kinds of systems allow more sophisticated forms of syntax rewriting than simple macro systems. These systems range from aspect-oriented languages [31] to compile-time metaobject protocols (MOPs) [10, 29]. Compile-time MOPs allow a metaclass to rewrite syntax in the base language. However, these systems typically have limited facilities for defining new syntax.

This paper describes an extensible version of Java called Maya, which supports both the extension of its syntax and the extension of its base semantics. Figure 1 shows how Maya is used. Extensions, which are called Mayans, are written as code that generates abstract syntax trees (ASTs). After Mayans have been compiled, they can be loaded into the Maya compiler and used while compiling applications. Mayans are dispatched from the parser at application compile-time; they can reinterpret or extend Maya syntax by expanding it to other Maya syntax.

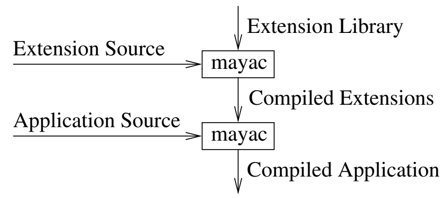


Figure 1: Compiling language extensions and extended programs with the Maya compiler, `mayac`

PLDI'02, June 17-19, 2002, Berlin, Germany.

Maya has the following combination of features:

- Mayans operate on abstract syntax. Maya can ensure that Mayans produce valid ASTs, because AST nodes are well typed. Since Mayans do not operate on flat token streams, they are not subject to precedence errors that occur in typical macro systems. For example, the Java Syntactic Extender (JSE) [2] allows macros to be defined with a case statement that matches concrete syntax against patterns. Because JSE macros operate on concrete syntax, they can generate many of the same parse and precedence errors that C macros generate.

- Maya treats grammar productions as generic functions, and semantic actions (Mayans) as multimethods on those generic functions. Mayans can be dispatched on a rich set of parameter specializers: AST node types, the static types of expressions, the concrete values of tokens, and the syntactic structure of AST nodes. Multiple dispatch allows users to extend the semantics of the language by overriding Maya's base semantic actions.

- Maya allows programmers to generate ASTs with templates, a facility like quasiquote in Lisp [28]. Maya templates can be used to build arbitrary pieces of abstract syntax. Most other systems provide less general template mechanisms (`<bigwig>` [6] is an exception). For example, JTS [5] is a framework for building Java preprocessors that provides limited support for templates, in that it only defines template syntax corresponding to a fixed subset of the JTS grammar's nonterminals.

- Like Java classes, Mayans are lexically scoped. Local Mayan declarations can capture the state of enclosing instances. In addition, Mayan definitions are separate from imports. Imported Mayans are only applicable within the scope of their import. These features allow great flexibility in the way that syntax transformers share state and are exposed to the base code. In comparison, compile-time metaobject protocols such as OpenJava [29] typically provide fixed relationships between transformers, state, and the lexical structure of base code.

To support these features, Maya makes use of three new implementation techniques:

1. To support dispatch on static types, Maya interleaves *lazy type checking* with *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 the effects of laziness can be controlled.

2. Maya uses a novel parsing technique that we call *pattern parsing* to statically check the bodies of templates for syntactic correctness. The pattern parsing technique allows a programmer to use quasiquote to generate any valid AST.

3. Maya supports hygiene and referential transparency in templates through a compile-time renaming step.[1] This renaming is possible because binding constructs must be explicitly declared as such in Maya's grammar. Maya's implementation of hygiene detects most references to free variables when templates are compiled, but it does not support Dylan-like implicit parameters.

The rest of this paper is organized as follows. Section 2 reviews some of the systems that we compare Maya against. Section 3 introduces the basics of the Maya language with an extended example. Section 4 describes the high-level design and implementation of the Maya compiler, including how Mayans are compiled and integrated into the Maya parser, how lazy parsing and type checking work, how templates are parsed, and how hygiene and dispatch work. Section 5 sketches how Maya can be used to implement an interesting language extension: namely, open classes and multimethods as defined in MultiJava by Clifton et al. [12]. Section 6 describes related work, and Section 7 summarizes our conclusions.

2. BACKGROUND

In this section we describe three systems that are closely related to Maya, and to which we compare Maya in the rest of the paper: JSE, JTS, and OpenJava.

JSE [2] is a port of Functional Objects' procedural Dylan macros to Java. JSE macros are defined using Dylan patterns, but a macro's expansion is computed by JSE code rather than by pattern substitution. JSE macros must follow one of two syntactic forms: method-like and statement-like macros. JSE recognizes macro keywords through a class naming convention.

JSE provides a quasiquote mechanism to build macro return values. However, because macros operate on unparsed trees of tokens and matching delimiters, JSE cannot statically check that quasiquotes produce syntactically correct output. In addition, macro expansion does not honor precedence.

JTS [5] is a framework for writing Java preprocessors that operate on ASTs. JTS and Maya approach 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, AST node types, and grammar symbols. JTS extensions are also free to mutate syntax trees, since typechecking is performed by a standard Java compiler, after extensions have run to completion. JTS provides great flexibility at the syntactic level but ignores static semantics.

OpenJava [29] is a compile-time metaobject protocol for Java. It allows metaclasses to be associated with classes in the base program. A class declares its metaclass with an `instantiates` clause, which appears after `implements` in a class declaration. Metaclasses inherit introspection methods similar to the Java reflection API, and control translation to Java through the visitor pattern. OpenJava provides for two kinds of macro expansion: *caller-side* translation allows a metaclass to expand expressions involving its instance types through visit methods on various expression and declaration forms, and *callee-side* translation allows a metaclass to

[1]Hygiene means that variable references in Mayans cannot by default capture variables in scope at the use of a Mayan; referential transparency means that variable references in Mayans capture variables in scope at the definition of a Mayan [13]. As is common in the literature, we use the single term "hygiene" to mean both hygiene and referential transparency.

modify instance class declarations. OpenJava also permits limited extensions to the syntax of `instantiates` and type names.

OpenJava controls syntax translation based on the static types of expressions, but imposes some limitations. Metaclasses must be explicitly associated with base-level classes through the `instantiates` clause. As a result, primitive and array types cannot be used to dispatch syntax expanders. Additionally, caller-side expanders override visit methods defined on a subset of Java syntax.

OpenJava lacks some features that make compile-time metaprograms robust: Its macros can generate illegal pieces of syntax, because they allow metaprograms to convert arbitrary strings to syntax. In addition, OpenJava metaclasses inspect nodes through accessor methods rather than pattern matching. Finally, OpenJava does not support hygiene.

3. MAYA OVERVIEW

Maya can be used to implement simple macros as well as language extensions such as MultiJava and aspects [4]. Maya provides a macro library that includes features such as assertions, printf-style string formatting, comprehension syntax for building arrays and collections, and `foreach` syntax for walking them. This section describes the features that a `foreach` macro should have, and the way that these features can be implemented in Maya. In the examples that follow, we use bold text for keywords, and italic text for binding instance names.

Given a `Hashtable` variable `h`, the following use of `foreach`:

```
h.keys().foreach(String st) {
  System.err.println(st + " = " + h.get(st));
}
```

should expand to:

```
for (Enumeration enumVar$ = h.keys();
     enumVar$.hasMoreElements(); ) {
  String st;
  st = (String) enumVar$.nextElement();
  System.err.println(st + " = " + h.get(st));
}
```

where the identifier `emumVar$` does not appear in source program. Many macro systems support exactly this sort of macro. Although JSE cannot express our chosen concrete syntax, it allows a similar macro to be written. OpenC++ [10] includes specific support for member statements such as `foreach`.

Macro overloading is useful with statements such as `foreach`. We might want a version of `foreach` that works on arrays. Alternatively, we might want overloading for optimization. For instance, the following code:

```
maya.util.Vector v;
v.elements().foreach(String st) {
  System.err.println(st);
}
```

could be expanded to more efficient code with a specialized version of `foreach`:

```
maya.util.Vector v;
{
  Vector v$ = v;
  int len$ = v$.size();
  Object[] arr$ = v$.getElementData();
  for (int i$ = 0; i$ < len$; i++) {
    String st = (String) arr$[i$];
    System.err.println(st);
  }
}
```

This code can avoid both object allocation and method calls because `maya.util.Vector` exposes its underlying object array by overriding `Vector.getElementData()`.

To support the optimization of `foreach` on a vector's elements, Maya allows semantic actions (called Mayans) to be dispatched on both the structure and static types of arguments. For this example, the left-hand side of the specialized `foreach` must be a call to `elements()` and the receiver of the call must have type `maya.util.Vector`. Although macro systems such as Scheme `syntax-rules` [21] and JSE support syntactic pattern matching, and although compile-time MOPs such as OpenJava dispatch on static types, Maya is the first compile-time metaprogramming system to unify these features. Maya's multiple-dispatch model provides benefits over the case statements of `syntax-case` [17] and JSE in that the behavior of a syntactic form can be extended without modifying the original definition.

This example shows the central challenge in providing flexible macro dispatching. The `foreach` statement's expansion depends on the static type of `v.elements()`, yet the loop body cannot be type checked until the `foreach` statement is expanded and the variable `st` is declared. Some code cannot even be parsed until surrounding Mayans have been expanded. Maya addresses this challenge through lazy parsing and lazy type checking.

3.1 Production Declarations

In Maya, a syntax extension is defined in two parts. First, a new LALR(1) production may need to be added, if the new syntax is not accepted by the existing grammar. Second, Mayans define semantic actions for a production, and are dispatched based on values on the production's right-hand side.

Before Mayans can be defined to implement `foreach`, we must extend the grammar to accept the `foreach` syntax. The following production would suffice (the concrete Maya syntax for describing this production is given in the next paragraph):

Statement → *MethodName* **(** *Formal* **)** *lazy-block*

We choose this syntax to avoid making `foreach` a reserved word in this context. The *MethodName* nonterminal matches everything left of '`(`' in a method invocation. In particular, *MethodName* matches the '*Expression* **.** *Identifier*' sequence accepted by `foreach`. In this example, *lazy-block* matches a Java block that is lazily parsed: it is not parsed or type checked until its syntax tree is needed.

The production above is declared in Maya by the following code:

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

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

The production is introduced through the **`abstract`** and **`syntax`** keywords. The **`syntax`** keyword indicates that a grammar production or Mayan is being defined. (Recall that a Mayan is a semantic action in the grammar.) The **`abstract`** keyword indicates that a production is being defined. Intuitively, a production is like an abstract method in that it defines an interface; a corresponding Mayan is a concrete method that "implements" the interface.

`Statement` is the return type of the production (i.e., the left-hand side); it is also the return type of any corresponding Mayans (semantic actions). The arguments to the production are the right-hand side of the production. The **`lazy`** keyword indicates lazy

```
1  Statement syntax
2  EForEach(Expression:Enumeration enumExp
3           \. foreach(Formal var)
4           lazy(BraceTree, BlockStmts) body)
5  {
6    final StrictTypeName castType
7      = StrictTypeName.make(var.getType());
8
9    return new Statement {
10     for (Enumeration enumVar = $enumExp;
11          enumVar.hasMoreElements(); ) {
12       $(DeclStmt.make(var))
13       $(Reference.makeExpr(var.getLocation()))
14          = ($castType) enumVar.nextElement();
15       $body
16     }
17   };
18 }
```

Figure 2: The Mayan that implements `foreach` **on** `Enumerations`

parsing. The particular use of **`lazy`** above means that a `BraceTree` (a portion of code surrounded by braces) is to be lazily parsed into a `BlockStmts` (a block of statements).

Most symbols in the Maya grammar are AST node types such as `Statement`. Productions and Mayans may only be defined on node-type symbols. Maya also supports several kinds of *parameterized grammar symbols* that are used to define repetition and control lazy parsing. **`lazy`**`(BraceTree, BlockStmts)`, is an example of the use of a parameterized grammar symbol.

3.2 Mayan Declarations

After we define the production for `foreach`, we can declare Mayans to translate various kinds of `foreach` statements to standard Maya syntax. Note that if no Mayans are declared on a new production (that is, no semantic actions are present on the production), an error is signaled on input causes the production to reduce.

A Mayan declaration differs from a production declaration in three ways: Mayan parameters have specializers and names; Mayan declarations have bodies; and Mayans do not begin with the **`abstract`** keyword. Figure 2 shows the `EForEach` Mayan that implements `foreach` for `Enumerations` (not the optimized version).

The `EForEach` Mayan is defined on the LALR(1) production described earlier, which takes the left-hand side of a method invocation followed by a formal parameter and a block. Maya determines that this production corresponds to `EForEach` when the `EForEach` parameter list is parsed. A Mayan parameter list serves two purposes. First, it determines which occurrences of syntax a Mayan can be applied to. Second, it binds formal parameters to actual arguments and their substructure. Mayan parameter lists and case patterns in functional languages serve similar purposes. In fact, Maya's pattern matching facility is made available through a `syntax case` statement as well as through Mayan dispatch.

Parameter specializers are used to narrow Mayan applicability: `EForEach` only applies to `MethodName` nodes that contain an explicit receiver expression. The receiver expression is bound to `enumExp` and must have the static type `Enumeration`. The final identifier in the `MethodName` syntax is also specialized to a particular token value: namely, `foreach`. Maya's ability to dispatch on the values of identifiers such as `foreach` allows macros to be defined without introducing reserved words. `EForEach` binds the loop variable and loop body to `var` and `body` respectively.

The body of a Mayan is Maya code that generates an AST. For example, `EForEach`'s body consists of a local variable declaration and a return statement. The return value is computed using a *template* expression that builds ASTs. Template can be used to build the ASTs from concrete syntax. For example, a template containing '`1 + 2 * 3`' builds the corresponding tree. A template may also contain expressions unquoted with '`$`': the values of these expressions are substituted into the resulting AST when the template is evaluated. Templates are statically parsed to ensure syntactic correctness.

Maya's templates provide automatic hygiene for lexically scoped names; programmers are given mechanisms to explicitly break hygiene and to explicitly generate fresh names if they so desire. In the case of `EForEach`, hygiene ensures two things: first, that the loop variable will not interfere with references to other variables called `enumVar` in the loop body; second, that the loop variable will have the specific type `java.util.Enumeration`, regardless of the calling context.

Like OpenJava metaclasses, Mayans have access to a variant of the Java reflection API. References to `Type` objects are available through methods such as `VarDeclaration.getType()` and `Expression.getStaticType()`. `Type` objects support `java.lang.Class`'s introspection API and a limited form of intercession that allows member declarations to be added to a class body.

Mayans can use the reflection API to insulate themselves from some details of Maya's AST representation. For example, the abstract syntax trees for `String[] args` and `String args[]` have different shapes, but both declare variables of the same type. `EForEach` uses the reflection API in two places. First, line 7 of Figure 2 builds a `StrictTypeName` node from the object that represents the type of a variable. Second, line 13 generates a reference to a local variable directly, rather than generating an occurrence of the variable's name. `Reference.makeExpr` also allows fields to be referenced when they are shadowed by local variables. Line 12 translates a formal parameter declaration into a form that may be used in a statement context.

3.3 Using Mayans

Maya decouples Mayan definition from use: a Mayan is not implicitly loaded into the compiler at the point of declaration, but must be loaded explicitly. This feature allows a Mayan to be compiled independent of its uses, and allows local Mayans to use any value in their environments.

A Mayan declaration, such as `EForEach` in Figure 2, is compiled to a class that implements `MetaProgram`. An instance of the class is allocated when a Mayan is imported. A programmer uses the **`use`** directive to import `MetaProgram` instances into a lexical scope; the argument to **`use`** can be any class that implements `MetaProgram`. For example, `EForEach` can be used in a method body as follows:

```
void showEm(Enumeration e) {
  use EForEach;
  e.foreach(Object o) { System.out.println(o); }
}
```

Imports of Mayans are lexically scoped. In this example, the scope of the translation defined by `EForEach` consists only of the method body of `showEm`.

The **`use`** directive is also valid in class bodies and at the top level. Additionally, Maya provides a `-use` command line option that allows a programmer to compile a file using different Mayan implementations.

```
abstract Statement
syntax(typedef(Identifier = StrictClassName)
       lazy(BraceTree, BlockStmts));

public Statement syntax
Typedef(typedef(Identifier var=StrictClassName val)
         lazy(BraceTree, BlockStmts) body)
{

  // a local Mayan
  StrictName syntax Subst(Identifier id) {
    if (id.getName() == var.getName())
      // substitute the type variable
      return val;
    else
      // resolve this name normally
      return nextRewrite();
  }

  // Construct a statement node that exposes
  // an instance of Subst to the typedef body
  return new UseStmt(new Subst(), body);
}
```

Figure 3: Local Mayans, as well as local classes, may refer to arguments of an enclosing Mayan or method.

Since Mayans are typically small units of code, they can be aggregated into larger metaprograms. An instance of such a class is allocated, and its `run` method is called to update the environment. For example, the class `maya.util.ForEach` defines a single `run` method, which instantiates and runs each built-in `foreach` Mayan in turn. As a result, a programmer need only **`use`** the `maya.util.ForEach` class to import all of the built-in `foreach` Mayans.

Like class declarations, Mayan declarations can appear within method bodies. Local Mayan instances are closed over their lexical environment in the same way as local class instances. Because a local Mayan definition can use any value in its environment, it cannot be imported until these values exist. As a result, local Mayans can never be imported with **`use`**; however, local Mayans can be instantiated and run by other metaprograms. The advantage of local Mayans is that one Mayan can expose state to other Mayans without resorting to templates that define Mayans. As a result, nontrivial metaprograms can be structured as a group of classes and a few small Mayans, rather than as a series of macro definitions.

Figure 3 demonstrates the use of local Mayans. The `Typedef` Mayan defines an alternate name for a class within a block of statements. Type variables are substituted by a local Mayan that compares each identifier in the block against the argument to `Typedef`. A local Mayan type, `Subst`, is defined within `Typedef`. Like a local class, this type can only be refered to in its lexical scope. `Typedef` allocates an instance of `Subst`, and exposes that instance to `body` through a `UseStmt` node. (The code '**`use`** `EForEach; ...`' in the previous example is translated into a `UseStmt` node.) `UseStmt` nodes contain the metaprogram that is imported and the list of statements in which it is visible.

4. DESIGN AND IMPLEMENTATION

Translating Maya source code to a fully expanded syntax tree involves dispatching and running Mayans. Dispatch may require type checking, while executing Mayans may change the types of subtrees. This mutual recursion precludes parsing and type checking in one pass; Mayan dispatch requires that they be interleaved. Maya satisfies these constraints by parsing and type checking lazily, i.e., computing syntax trees and their types on demand.

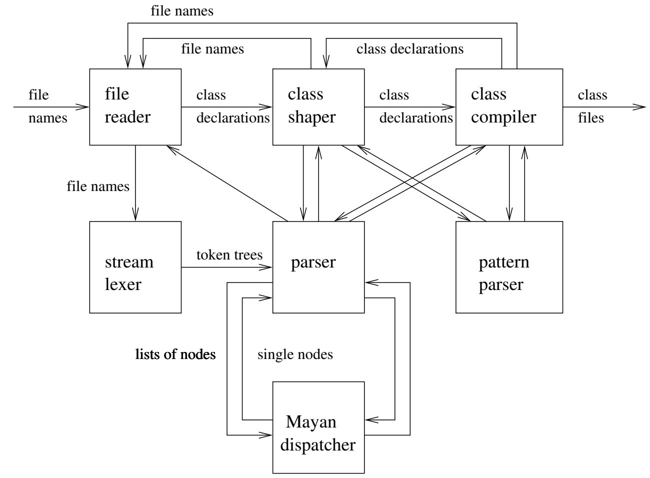


Figure 4: Overview of Maya's internal architecture

Figure 4 shows the major components of our Maya compiler, `mayac`. The *file reader* reads class declarations from source files. Our compiler then processes each class declaration in two additional stages. The *class shaper* parses the class body and computes member types; the *class compiler* parses member initializers, including method and constructor bodies. The parser is invoked in all three steps to incrementally refine a declaration's AST. To compute the shape of a class C, all super-types of C and the types of all members of C must be found. Similarly, to compile C, the shapes of all types referred to by C's code must be known. Maya provides class-processing hooks that execute user-defined code as a class declaration leaves the shaper.

The *stream lexer* enables lazy parsing by generating a tree of tokens rather than a flat stream. Specifically, the stream lexer creates a subtree for each pair of matching delimiters: parentheses, braces, and brackets. These subtrees are called *lexers* since they can provide input to the parser. The stream lexer resembles a Lisp reader in that it builds trees from a simple context-free language. This tree structure allows the Maya compiler to quickly find the end of a method body or field initializer without full parsing. Maya processes toplevel declarations and class bodies in phases so that forward references can be resolved and typechecking can be performed within the parser.

Unless otherwise noted, all arcs coming into the *parser* are lexers 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. Mayan dispatch may involve recursive parsing, as shown in Figure 4.

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.

The remainder of this section describes several of Maya's features. Section 4.1 discusses Maya's lazy grammar and how new productions can be written to extend it. Section 4.2 discusses pattern matching in Mayan parameter lists, AST templates, and the parsing techniques used to implement these language features. Section 4.3 describes Maya's static approach to hygiene. Section 4.4 discusses Maya's dispatching rules in detail. Finally, Section 4.5 compares Maya's features to those of related Java extensions.

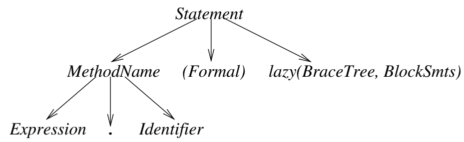


Figure 5: The structure of `EForEach`'s formal parameters

4.1 Parsing

Maya productions are written in a high-level metagrammar that explicitly supports laziness. Users can define general-purpose productions on any builtin nonterminal of the Maya grammar. User-defined productions are indistinguishable from those built into Maya. In addition, parameterized grammar symbols may implicitly define productions on new nonterminals.

Production arguments (right-hand sides) consist of token literals, node types, matching-delimiter subtrees, or parameterized symbols. (**`lazy`**`(BraceTree, BlockStmts)` is an example of a parameterized symbol.) The former two symbol types are used directly by the LALR(1) grammar, while the latter two require special handling. When a subtree or parameterized symbol is encountered, Maya ensures that the corresponding production is defined in the LALR(1) grammar, and uses the left-hand side in the outer production.

For instance, the production used by `foreach` includes both a subtree and a parameterized symbol:

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

It is translated to the set of productions below:

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

Semantic actions (which are not shown) on G_0 and G_1 produce AST nodes from unstructured subtrees. The semantic action for G_0 recursively parses the `ParenTree` to a *Formal*, which is mentioned explicitly in the production. The action for G_1 delays the parsing of the `BraceTree`, as specified by the **`lazy`** symbol. If the productions and actions already exist in the grammar, they are not added again. For example, the production and action for G_0 are used to parse both `foreach` and catch clauses; those for G_1 are used throughout the Maya grammar.

A production is valid if it does not introduce conflicts into the grammar. The Maya parser generator attempts to resolve conflicts with operator precedence relations. Unlike YACC, Maya does not resolve shift/reduce conflicts in favor of shifts or reduce/reduce conflicts based on the lexical order of productions. The parser generator rejects grammars that contain unresolved LALR(1) conflicts.

4.2 Pattern Parsing

Maya uses patterns in two ways: first, to establish bindings in Mayan parameter lists and `case` clauses; and second, to compile templates that dynamically construct AST nodes. Although these uses are very different, they involve the same data structure: a partial parse tree built from a sequence of both terminal and nonterminal input symbols. The pattern parser differs from a standard parser in that its input may include nonterminals as well as tokens. Just as the pattern parser accepts nonterminal input symbols, it also generates parse trees that may contain nonterminal leaves.

Recall that `EForEach` in Figure 2 is defined on the production given in Section 3.1. The pattern parser must infer the structure of `EForEach`'s argument list, which is shown in Figure 5. Since `EForEach` is a semantic action, it takes three actual arguments: a `MethodName`, a parenthesized `Formal`, and a lazily parsed block. However, the first argument does not explicitly appear in `EForEach`'s formal parameter list. The pattern parser infers the structure of the first argument by parsing the symbols in the argument list.

The pattern parser also parses template bodies. Maya guarantees that a template is syntactically correct by parsing its body when the template is compiled. The pattern parser generates a parse tree from a sequence of tokens that may be interleaved with expressions unquoted with '`$`'. An unquote expression's grammar symbol is determined by its static type or an explicit coercion operator.

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. Templates honor laziness: sub-templates that correspond to lazy syntax are compiled into local thunk classes that are expanded when the corresponding syntax would be parsed.

While some systems have developed ad hoc approaches for template parsing [5, 10, 26, 29, 30], Maya opts for a more general solution. Our pattern-parsing algorithm allows Mayans to be defined on any nonterminal node type, and templates can generate both nonterminal node types and parameterized symbols. A proof of this algorithm's correctness is available elsewhere [3].

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

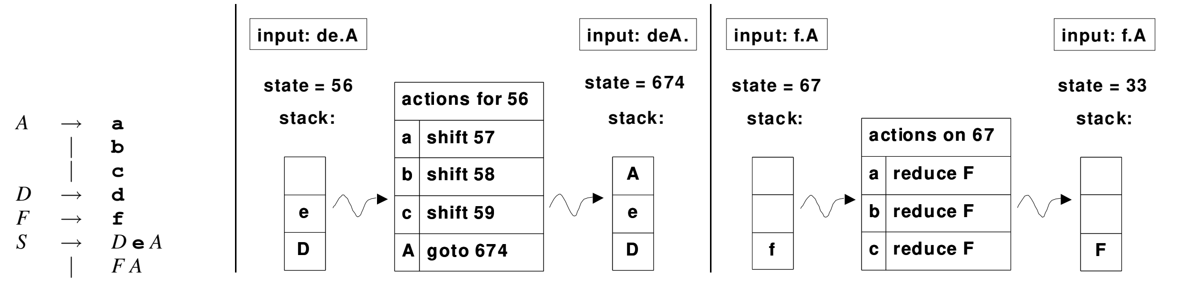
The pattern parser uses parse tables in much the same way as a normal LALR(1) parser. Terminal input symbols are shifted onto the stack or trigger reductions normally. However, nonterminal symbols require some special handling. Figure 6 provides concrete examples of how nonterminals can be parsed. When the pattern parser encounters a syntactically valid input $X\gamma$, there must be a production $Y \rightarrow \alpha X \beta$ in the grammar, and the parser must be in some state s_0 such that one of the following cases holds:

1. s_0 contains an item $Y \rightarrow \alpha.X\beta$: actions on $\text{FIRST}(X)$ are all shifts, and s_0 contains a goto for X that leads to some state s_n. In this case, X is shifted onto the stack, and the goto is followed. That is, the current parsing state will "accept" an X because there is an entry in the goto table.

   Figure 6(b) illustrates this case, given the grammar in Figure 6(a). In this example, the metavariable X corresponds to A; Y corresponds to S; α corresponds to D **e**; and β and γ correspond to ϵ.

2. s_0 contains an item $Z \rightarrow \delta.$, where Z is a nonterminal such that $\alpha \overset{*}{\Rightarrow} \zeta Z$, and the actions on $\text{FIRST}(X\gamma)$ all reduce the same rule $Z \rightarrow \delta$. In this case, the stack is reduced, which leads to a state s_1 in which one of the above conditions holds. That is, the current parsing state will "accept" an X because we can perform a reduction on the input before X.

   6(c) illustrates this case, again given the grammar in Figure 6(a). In this example, X corresponds to A; Y corresponds to S; α and Z correspond to F; and β, γ, and ζ correspond to ϵ.



(a) A simple grammar (b) A goto can be followed if present (c) Otherwise, FIRST(A) serves as lookahead

Figure 6: Pattern parsing example

If neither case holds, the input must not be valid. Note that X could be invalid in the second case, and that the pattern parser may not detect the error until it has performed some reductions.

4.3 Hygiene

Maya supports compile-time determination of variable hygiene in templates, unlike most macro systems. Maya's static approach to hygiene detects references to unbound variables when a template is compiled, rather than when it is executed. In general hygiene decisions can only be made when the syntactic role of each identifier is known. In most systems, this information is only available after all macros have been expanded. The key to Maya's hygiene rules is that a Mayan writer must make explicit which identifiers are bound and which are not: productions that establish lexically scoped bindings must use special nonterminals such as `UnboundLocal`.

Maya examines trees produced by the pattern parser to decide where hygienic and referentially transparent renaming should occur. Referential transparency in Mayan parameter lists ensures that a class name matches nodes that denote the same fully qualified name, and that class names in templates refer to the appropriate classes. Maya implements referential transparency by generating nodes that refer to classes directly.

Maya's implementation of hygiene relies on direct generation of bytecode. Maya assigns fresh names to local variable declarations generated by Mayans. The names that Maya generates contain the character '`$`' and are guaranteed to be unique within a compilation unit. In a source-to-source translator, hygiene would have to be implemented through a more complicated mechanism.

As an aside, implementing referential transparency in a translation to Java source code would be difficult, since class names can be completely inaccessible in certain packages. For instance, `java.lang.System` cannot be referenced inside the package below, due to Java's name rules:

```
package p;
class java { }
class System { }
```

Maya provides mechanisms both to explicitly generate unique names and to explicitly break hygiene. `Environment.makeId()` generates a fresh `Identifier`. Mayans can violate hygiene by unquoting `Identifier`-valued expressions in templates, rather than embedding literal names. Referential transparency can also be bypassed through the introspection API. Maya's hygiene rules do not support implicit parameters that are only shared between a Mayan and its point of use: names are either fully hygienic or fully unhygienic.

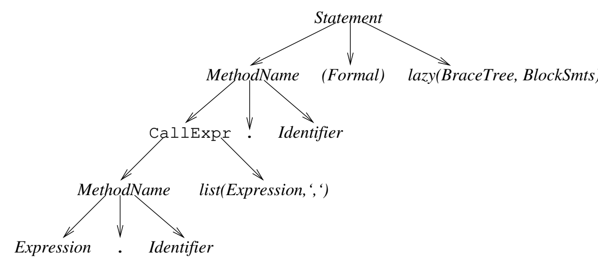


Figure 7: The structure of `VForEach`'s formal parameters

4.4 Dispatch

Mayan dispatching is at the core of Maya. Each time a production is reduced, the parser dispatches to the appropriate 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. While multiple dispatch is present in many languages such as Cecil [9], CLOS [28], and Dylan [27], the exact formulation of Maya's rules is fairly unique.

Mayan dispatching is based on the runtime types of AST nodes, but also considers secondary parameter attributes. As described earlier, the pattern parser builds Mayan parameter trees from unstructured lists of tokens and named parameters. A Mayan parameter consists of a node type and an optional secondary attribute. This attribute may contain subparameters, a token value, a static expression type, or a class literal type. For example, in Figure 5, `EForEach` specializes `MethodName` on its structure, `Expression` on its static type, and `Identifier` on the token value `foreach`.

Secondary attributes are used to compare formal parameters with identical node types. Static expression types are compared using subtype relationships; substructure is compared recursively; class types and token values must match exactly. For example, the following optimized `foreach` definition

```
Statement syntax
VForEach(Expression:maya.util.Vector v
         \.elements()\.foreach(Formal var)
         lazy(BraceTree, BlockStmts) body)
{ /* ... */ }
```

overrides `EForEach` for certain input values. The structure of `VForEach`'s formal parameters is shown in Figure 7. `VForEach` is more specific than `EForEach` because the `foreach` "receiver" is specialized on the `CallExpr` node type, while `EForEach` does not specialize the receiver node type.

Mayan dispatching 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 by Ducournau et al. [16], and is consistent with Java's static overloading semantics.

Maya supports an unusual form of lexical tie breaking among Mayans that are equally specific: Mayans that are imported later override earlier Mayans. 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. For example, many of the Mayans that we have written do not specialize their arguments. Such Mayans are more applicable than the built-in Mayans only because the built-in Mayans are imported first.

Maya provides an operator that is necessary to support the layering of macros. The `nextRewrite` operator calls the next-most-applicable Mayan, and is analogous to `super` calls in methods. This operator may only be used within Mayan bodies, whereas other Maya features such as templates may be used in any Maya code.

4.5 Discussion

Maya's expressiveness can be compared to that of the other languages discussed in Section 2. Neither JSE, JTS, nor OpenJava support automatic hygiene. JSE and JTS do not provide static type information, while JSE and OpenJava provide limited facilities for defining new syntax.

JTS allows extensions to define arbitrary new productions, while Maya only allows general-purpose productions to be defined on a fixed set of left-hand sides. JTS extensions are also free to extend the abstract syntax tree format. Since Maya must be able assign standard Java meaning to every syntax tree, only a few specific node types may be subclassed by users.[2]

Finally, JTS extensions may walk syntax trees in any order and perform mutation. In contrast, Mayans are executed by the parser, and can only mutate class declaration by adding and removing members when certain dynamic conditions are met. This restriction ensures that class updates cannot change the results of previous type checking and Mayan dispatch.

OpenJava and Maya both allow some form of syntax extension, dispatch based on static information, and the association of syntax transformation code with the lexical structure of the program. Maya provides more flexibility in these areas.

OpenJava allows three extensions to declaration syntax: new modifiers may be defined, clauses may be added to the class declaration syntax, and suffixes may be added to the type in a variable declaration. In all cases, a keyword signals the presence of extended syntax, and the metaclass must provide a parsing routine. In contrast, Maya allows new productions to be defined on any grammar symbol, and does not require that keywords be used.

OpenJava dispatches *caller-side* visit methods to transform an input program based on its static types. However, visit methods are only defined for a few expression and declaration forms. In contrast, Mayans may be defined on all productions in the base grammar, as well as user-defined productions.

OpenJava associates syntax transformation methods with class declarations through the `instantiates` clause. A metaclass is free to examine and modify the bodies of its instance classes. Although Maya does not provide a similar mechanism, local Mayan definitions and imports allow user-defined metaobjects to be created easily. These metaobjects need not be associated with classes.

[2] This restriction is not currently enforced in the implementation.

5. MULTIJAVA

To evaluate Maya, we implemented MultiJava, an extension to Java that Clifton et al. have specified [12]. The first author's thesis [3] contains several smaller examples of how Maya can be used; it also discusses the MultiJava implementation in greater detail.

MultiJava adds generic functions to Java through two new constructs: open classes and multimethods. Open classes allow methods to be declared at the top level, and multimethods allow dispatch based on the runtime types of all method arguments. MultiJava's formulation of open classes and multimethods supports separate compilation.

5.1 MultiJava Overview

Open classes in MultiJava offer an alternative to the visitor design pattern. Rather than maintaining a visit method on each class in a hierarchy and a separate 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 can override external methods that it inherits.

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 defines a compilation strategy and type checking rules that together allow 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.

In addition to open classes, MultiJava supports multimethods. Multimethods allow a class to define several methods with the same name and signature, but distinct parameter specializers. Each virtual function is treated as a generic function, so that multiple dispatch and static overloading can be used together. A `super` call in a multimethod (to the same generic function) selects the next applicable method, rather than the method defined by a superclass.

MultiJava imposes some restrictions on parameter specializers to support separate compilation. First, only class-typed parameters may be specialized, and they may only be specialized by subclasses. Second, a concrete class must define or inherit multimethods for all argument types (including abstract types). These restrictions allow MultiJava to statically ensure that there is exactly one most applicable method for every possible call to a generic function.

5.2 Maya Implementation

We implemented MultiJava with a mixture of class and Mayan definitions. The majority of code is defined in ordinary methods, rather than in Mayan bodies. Maya provides several features that make a MultiJava implementation practical:

- The full power of Maya's extensible LALR(1) grammar is used to extend Java's concrete syntax in two ways. First, MultiJava allows methods to be declared outside of their receiver class. These methods declarations include the receiver class name immediately after their return type.

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

  Second, MultiJava allows method parameters to be declared with both a base type that determines the method's virtual function and a type specializer that narrows the method's applicability at runtime. A parameter specializer appears after an '@' in a parameter declaration.

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

- Maya's lexical tie-breaking rule lets MultiJava transparently change the translation of base Java syntax. Our MultiJava implementation examines every ordinary method declaration to determine whether its parameters include specializers, and whether it overrides an external method. Maya dispatches to these Mayans rather than built-in Mayans because of the dispatcher's lexical tie-breaking rule.
- Maya provides standard Java type information to Mayans, which MultiJava needs to enforce its type checking rules.
- Semantic actions are implemented as local Mayans, which allows them to share state.
- The lexical scoping of imported Mayans is used to limit the scope of translations.

Local Mayan declarations and lexically scoped imports allow our MultiJava implementation to be structured hierarchically into high-level constructs. For example, our implementation instantiates classes named `MultiMethod` and `GenericFunction` to keep track of MultiJava language constructs. These objects are similar to instances of metaclasses in OpenJava; unlike metaclasses, our classes are not a fixed part of the Maya language. `MultiMethod` and `GenericFunction` store information that is used to ensure that generic function definitions cannot produce dispatch errors, and to compute the method of `super` sends from multimethods. The actual translation of `super` sends is performed by a method-local Mayan defined in `MultiMethod`. This Mayan relies on its enclosing `MultiMethod` instance, and is exported locally within a multimethod body.

`GenericFunction` generates code to dispatch multimethods. For example, if `D` is a subclass of `C`,

```
class D extends C {
  int m(C c)   { return 0; }
  // a multimethod: executes if c is dynamically
  // a D
  int m(C@D c) { return 1; }
}
```

`D` will be translated as follows, with the Java method `m` generated by `GenericFunction`:

```
class D extends C {
  private int m$1(C c) { return 0; }
  private int m$2(D c) { return 1; }
  int m(C c) {
    return c instanceof D ? m$2((D) c) : m$1(c);
  }
}
```

The body of `m` is generated by a recursive method, `GenericFunction.dispatchArg()`. A simplified version of this method, which does not perform error checking or support `void` methods, is shown in Figure 8.

5.3 Discussion

Clifton [11] implemented MultiJava by modifying the Kopi [15] Java compiler, `kjc`. He added or materially altered 20,000 of the 50,000 lines in `kjc`. In contrast, our MultiJava implementation is less than 2,500 noncomment, nonblank lines of code. It should be noted that Clifton's MultiJava implementation involved fixing bugs and implementing standard Java features in `kjc`.

```
Expression dispatchArg
  (VarDeclaration[] vars, // dispatcher params
   Vector applicable,     // applicable methods
   int n)                 // argument to dispatch
{
  if (n == vars.length || applicable.size() == 1){
    // Applicable methods are sorted from most to
    // least specific.
    MultiMethod m =
      (MultiMethod) applicable.elementAt(0);
    return dispatchCall(vars, m);
  }

  // For each type specializer on the nth
  // argument, find all methods that may be
  // applicable when that type is encountered.
  // Sort the array of specializers and methods so
  // that subtypes come before supertypes.  This
  // gives us a valid order in which to perform
  // instanceof tests.
  Map.Entry[] ents = sortOnArg(applicable, n);

  // Generate dispatch code from right to left,
  // i.e., generate superclass cases first.
  int specInd = ents.length - 1;
  Vector v = (Vector) ents[specInd].getValue();
  Expression ret = dispatchArg(vars, v, n + 1);
  for (specInd-- ; specInd >= 0; specInd--) {
    Type t = (Type) ents[specInd].getKey();
    Vector app = (Vector)ents[specInd].getValue();
    ret = new Expression {
         ($(as Expression Reference.make(vars[n]))
          instanceof $(StrictTypeName.make(t)))
         ? $(dispatchArg(vars, app, n + 1))
         : $ret
    };
  }
  return ret;
}
```

Figure 8: Simplified version of code to that generates mulitmethod dispatching expressions.

One might consider implementing MultiJava in the languages described in Section 2. It is hard to imagine a MultiJava implementation in a macro system such as JSE, because JSE macros cannot transparently expand method declarations. JTS is a more viable option: since JTS allows the Java grammar to be extended and reinterpreted, it is certainly powerful enough to express MultiJava. However, one would have to implement Java's standard type checking rules along with MultiJava's rules, since JTS relies on `javac` for its type checking.

OpenJava provides some of the features that one would need. OpenJava allows the translation of a class to be changed when its declaration is annotated with an `instantiates` clause. A simple extension to OpenJava might allow the default metaclass to be changed transparently. OpenJava also exposes static type information to metaprograms, and supports lexical scoping through callee-side translation. However, OpenJava supports very a limited form of syntax extension that cannot express new kinds of declarations such as external methods. OpenJava's model of lexically scoped macros is also a poor match for MultiJava, since OpenJava supports class, but not method, metaobjects.

6. RELATED WORK

Most multiple-dispatch systems allow dispatch on more than just types. For instance, CLOS [28] dispatches on argument values using `eql` specializers, and Dylan [27] generalizes this feature with singleton and union types. Maya's dispatching rules bear a particular resemblance to the grand unified dispatcher (GUD) in Dubious [18]. Like GUD, Maya uses pattern matching. Additionally, GUD allows dispatch on user-defined predicates whereas Maya allows dispatch on the static types of expressions. Languages such as Dubious and Cecil [9] perform static checks to ensure that all generic function invocations unambiguously dispatch to a method. Maya defers these checks to expansion time.

A* [23] is a family of languages for writing preprocessors: it extends Awk to operate on syntax trees for particular source languages. A* supports a limited form of pattern matching: a case guard may be either a 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 Maya's patterns, A* patterns cannot contain substructure or additional attributes such as identifier values or expression types.

Maya is certainly not the only syntax manipulation system to support pattern matching. Scheme's `syntax-rules` [21] and its descendants [17, 27] allow macros to be defined using case analysis.

Maya borrows many ideas from high-level macro systems such as Chez Scheme's `syntax-case` [17], Dylan [27], and XL [24]. 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 hygienic implicit parameters, and local Mayans can share state more easily than local macros.

Chez Scheme's `syntax-case` provides programmable hygienic macros with lexical scoping. Unlike Chez Scheme, Maya separates the concepts of local macro definition and local macro import. This separation allows Mayans to share context directly. In `syntax-case`, context can only be shared indirectly: data must be encoded as literals and exposed to inner macros through `let-syntax` bindings. This technique can be cumbersome in any language, and is particularly unattractive in languages such as Java that limit literals to numbers, strings, and arrays of literals.

McMicMac [22] is an extended macro facility for Scheme that overcomes some of these limitations. McMacMic allows macros to explicitly pass inherited attributes as arguments and synthetic attributes as multiple return values. McMicMac also allows macros to be defined on "special" grammar constructs such as procedure application, but it is less powerful than Maya's multiple-dispatch model.

MS^2 [30] is essentially Common Lisp's `defmacro` for the C language. 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 syntactic 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.

MacroML [20] extends ML with a type-safe macro system. In MacroML, macro expansion cannot produce type errors. Macro return values are statically checked for both syntactic correctness and type safety. In contrast, Maya checks the syntactic structure of a template at compile-time, but only type checks the resulting tree when a template is expanded. MacroML supports three macro forms. Function macros consist of the macro name followed by one or more arguments, and may not establish bindings. Lambda-like macros consist of the macro name, a variable name, and a body in which the variable is bound. Finally, let-style macros consist of the `let` keyword, the macro name, a variable name and initializer, and a body in which the variable is bound. Since MacroML makes bindings explicit in a macro's form, it can make hygiene decisions statically.

Cardelli et al. [8] define an extensible LL(1) parsing package with macro-like features. A semantic action may not contain arbitrary code. Instead, it consists of calls to a predefined set of AST node constructors and references to parameters on the production's right hand side. Semantic actions in a language extension can also be defined through templates written in the base language syntax. This system takes an unusual approach to hygiene and referential transparency: hygienic names must be explicitly declared as `local`, but referential transparency is provided automatically.

`<bigwig>` [6] includes an unusual pattern-based macro facility in which macro expansion is guaranteed to terminate. `<bigwig>` macros define new productions in an extended LL(1) grammar. Macro productions consist of a macro keyword followed by zero or more terminal and nonterminal arguments. `<bigwig>` also allows the programmer to define new nonterminal symbols called metamorphs. Metamorphs are named and can be mutually recursive, features not present in Maya's parameterized grammar symbols.

A `<bigwig>` macro body is a template that contains concrete syntax and references to nonterminal arguments. `<bigwig>` allows all nonterminals to appear in macro arguments and templates (as Maya does), but Brabrand and Schwartzbach do not indicate whether their template implementation uses extra grammar productions or direct parser support.

`<bigwig>`'s extended LL(1) parser defines two conflict resolution rules. First, longer rules are chosen over shorter rules. Second, smaller FIRST sets are considered more specific than larger ones. This second rule handles identifiers used as keywords elegantly, but has farther-reaching implications. `<bigwig>`'s definition of well-formed grammars would allow:

$$
\begin{array}{lll}
S & \rightarrow & \mathtt{a}\ \mathtt{a} \mid T\ \mathtt{b} \\
T & \rightarrow & \mathtt{a} \mid \mathtt{b}
\end{array}
$$

even though `<bigwig>` cannot parse "a b". Maya's LALR(1) parser can accept this language, but more importantly, Maya signals an error when grammar conflicts cannot be resolved.

Maya's parsing strategy offers several advantages over `<bigwig>`'s. First, it allows parsing and type checking to be interleaved. Second, it uses standard LALR(1) parsing techniques, which are more flexible than LL(1). Finally, Maya rejects grammars that it cannot recognize.

Camlp4 [14] is a preprocessor for the OCaml programming language. Camlp4 parses OCaml source code using a variant of LL(1) parsing. Macros may extend the OCaml grammar, but as in `<bigwig>`, conflicts are not detected. Camlp4 allows syntax trees to be generated through quotations that are similar to Maya's templates. Quotations can also be used in pattern matching. Maya provides similar functionality through the `syntax case` statement. Maya differs from Camlp4 in that it exposes static type information to macros and that it rejects grammar extensions that produce conflicts.

ELIDE [7] is a Java preprocessor that allows programmers to define new declaration modifiers in terms of syntax transformer classes. Like OpenJava and Maya, ELIDE allows syntax transformers to modify declarations through an extension of the Java reflection API. OpenJava and ELIDE are also similar in that they allow strings to be dynamically converted to syntax objects and do not support general syntax extensions. However, where OpenJava allows a single metaclass to control the compilation of a class declaration, ELIDE allows an arbitrary number of modifiers.

7. CONCLUSIONS

We have described Maya, a version of Java that supports user-defined syntax extensions. These extensions are written as multimethods on generic functions, which makes them expressive and powerful. Local Mayan declarations and lexically scoped imports allow great flexibility in how language extensions are structured. Several implementation mechanisms make Maya's macros easy to write:

- Pattern parsing is a novel technique for clean and general quasiquoting in languages like Java.
- Lazy type checking allows Maya to dispatch arbitrary Mayans based on static type information, even though Mayans may create variable bindings.
- Lazy parsing allows lexically scoped Mayans to change the grammar.
- Maya's static template parsing and hygienic renaming allows large classes of macro errors to be detected: syntax errors are detected in parsing, and references to unbound variables are detected during hygiene analysis.

While Maya is well suited to implement language extensions such as aspect weaving [4] and MultiJava, it is equally well suited for smaller tasks such as the `foreach` macro described in Section 3. The Maya compiler and our MultiJava implementation are available at `http://www.cs.utah.edu/~jbaker/maya`.

Acknowledgments

We thank Eric Eide for his many comments on drafts of this paper. We also thank Sean McDirmid and John Regehr for their feedback. This research was supported in part by the National Science Foundation under CAREER award CCR–9876117 and the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreement number F33615–00–C–1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

8. REFERENCES

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

[2] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '01*, pages 31–42, Tampa Bay, FL, Oct. 2001.

[3] J. Baker. Macros that play: Migrating from Java to Maya. Master's thesis, University of Utah, Dec. 2001. `http://www.cs.utah.edu/~jbaker/maya/thesis.html`.

[4] J. Baker and W. C. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002.

[5] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proc. of the 5th International Conference on Software Reuse*, pages 143–153, Victoria, Canada, 1998.

[6] C. Brabrand and M. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation '02*, Portland, OR, Jan. 2002. `http://www.brics/dk/~mis/macro.ps`.

[7] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002.

[8] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, DEC SRC, Feb. 1994.

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

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

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

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

[13] W. Clinger and J. Reese. Macros that work. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, Toronto, Ontario, Jan. 1991.

[14] D. de Rauglaudre. Camlp4 reference manual. `http://caml.inria.fr/camlp4/manual/`, Jan. 2002.

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

[16] R. Ducournau, M. Habib, M. Huchard, and M. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '92*, pages 16–24, Vancouver, BC, Oct. 1992.

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

[18] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, pages 186–211, Vancouver, BC, Oct. 1998.

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

[20] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage

computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of the International Conference on Functional Programming '01*, pages 74–85, Florence, Italy, Sept. 2001.

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

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

[23] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, Nov. 1995.

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

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

[26] M. Poletto, W. Hsieh, D. Engler, and M. Kaashoek. "'C and tcc: A Language and Compiler for Dynamic Code Generation". *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

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

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

[29] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Proceedings of the OOPSLA '00 Reflection and Software Engineering Workshop*, Minneapolis, MN, Oct. 2000.

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

[31] Xerox. The AspectJ programming guide. `http://www.aspectj.org/doc/dist/progguide/`.