THE METALEXER LEXER SPECIFICATION LANGUAGE

by

Andrew Michael Casey

School of Computer Science

McGill University, Montréal

June 2009

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Compiler toolkits make it possible to rapidly develop compilers and translators for new programming languages. Recently, toolkit writers have focused on supporting extensible languages and systems that mix the syntaxes of multiple programming languages. However, this work has not been extended down to the lexical analysis level. As a result, users of these toolkits have to rely on ad-hoc solutions when they extend or mix syntaxes. This thesis presents MetaLexer, a new lexical specification language that remedies this deficiency.

MetaLexer has three key features: it abstracts lexical state transitions out of semantic actions, makes modules extensible by introducing multiple inheritance, and provides cross-platform support for a variety of programming languages and compiler front-end toolchains.

In addition to designing this new language, we have constructed a number of practical tools. The most important are a pair of translators that map MetaLexer to the popular JFlex lexical specification language and vice versa.

We have exercised MetaLexer by using it to create lexers for three real programming languages: AspectJ (and two extensions), a large subset of Matlab, and MetaLexer itself. The new specifications are easier to read and require much less action code than the originals.

Résumé

Les outils de compilation moderne permettent de développer rapidement des compilateurs pour de nouveaux langages de programmation. Récemment, les auteurs de ces outils ont travaillé à supporter des langages et systèmes extensibles qui mélangent la syntaxe de plusieurs langages de programmation. Cependant, ce travail n'a pas été étendu au niveau de l'analyse lexicale. Le résultat est que les utilisateurs de ces outils doivent se fier à des solutions improvisées quand ils augmentent ou mélangent la syntaxe de leurs langages. Cette thèse présente MetaLexer, un nouveau langage de spécification lexical qui remédie à ce manque.

MetaLexer a trois aspects principaux : il sépare les transitions d'états lexicaux des actions sémantiques, il rend les modules extensibles en introduisant un système d'héritage multiple, et il offre un support multi-plateforme pour une variété de langages de programmation et d'outils de compilation.

En plus de la conception de ce nouveau langage, nous avons implémenté un nombre d'outils pratiques. Le plus important étant une pair de programmes de traduction qui traduisent de MetaLexer au populaire JFlex et vice-versa.

Nous avons testé MetaLexer en l'utilisant pour créer des spécifications lexicales pour trois languages de programmations : AspectJ (et deux extensions), un large sous-ensemble du langage Matlab, et MetaLexer lui-même. Les nouvelles spécifications sont plus lisibles et demandent beaucoup moins de code d'action que les originales.

Acknowledgements

I would like to thank my supervisor, Laurie Hendren, without whom this thesis would not be what it is today. I am grateful for her feedback throughout the entire course of my research.

I would also like to thank the McLab team, whose challenging parsing requirements inspired this work and gave it its first practical test. In particular, I would like to thank Toheed Aslam for being the first brave soul to extend a MetaLexer specification.

I would like to thank my bilingual colleagues, Maxime Chevalier-Boisvert and Raphael Mannadiar, who were kind enough to translate my abstract into French.

I also owe a debt of gratitude to Torbjörn Ekman for his help with the JastAdd tool and to the creators of JFlex for the inspiration their tool provided.

This work was funded by the National Science and Engineering Research Council (NSERC), McGill University, and the McGill University School of Computer Science (SOCS).

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Much work has been done in the area of extensible compilers. JastAdd [EH07b] is an extensible attribute grammar framework that can be used to build compilers with extensible abstract syntax trees (ASTs), transformations, and analyses. The Polyglot Parser Generator [NCM03] is a extensible parser specification language (PSL). Unfortunately, little work has been done to make lexical specification languages (LSLs) similarly extensible. As a result, extensible compilers are forced to rely on ad-hoc solutions for lexing (e.g. [HdMC04]).

To remedy this deficiency, we have created a new LSL, MetaLexer, that is more modular and extensible than traditional LSLs.

This chapter describes the motivation behind MetaLexer's creation, lists contributions, and outlines the subsequent chapters.

1.1 Key Features

Three key features distinguish MetaLexer from its predecessors:

1. Lexical state transitions are lifted out of semantic actions (*Section 1.1.1*).
2. Modules support multiple inheritance (*Section 1.1.2*).

3. The design is cross-platform (*Section 1.1.3*).

1.1.1 Key Feature: State Transitions

Lexers for non-trivial languages nearly always make use of lexical states to handle different regions of the input according to different rules. The transitions between these states are buried in the semantic actions associated with rules and are language- and tool-dependent.

For example, *Listing 1.1* shows a JFlex[1] lexer with three states: initial, within a class, and within a string. Whenever an opening quotation mark is seen, whether in the initial state or within a class, the lexer transitions to the string state. Note that the previous state must be stored so that the lexer can return once the closing quote has been seen.

```
1  <YYINITIAL> {
2    \" { yybegin(STRING_STATE); prev = YYINITIAL; }
3    /* other rules related to lexing in the base state */
4  }
5  <CLASS> {
6    \" { yybegin(STRING_STATE); prev = CLASS; }
7    /* other rules related to lexing within a class */
8  }
9  <STRING_STATE> {
10   \" { yybegin(prev); return STRING(text); }
11   /* other rules that build up the string stored in text */
12 }
```

Listing 1.1 JFlex State Transitions

As in *Listing 1.1*, it is often the case that state transitions occur upon observing a particular sequence of tokens. Furthermore, transitions are often stack-based, like method calls. When a transition is triggered, the triggering lexical state is saved so that it can be restored once a terminating sequence of tokens is observed.

In other words, lexer transitions can often be described by rules of the form

When in state S_1, transition to state S_2 upon seeing token(s) T_1; transition back upon seeing token(s) T_2.

[1] http://jflex.de/

For example,

When in state BASE, transition to state COMMENT upon seeing token(s) START_COMMENT; transition back upon seeing token(s) END_COMMENT.

MetaLexer makes these rules explicit by associating "meta-tokens" with rules and then using a "meta-lexer" to match patterns of meta-tokens and trigger corresponding transitions. This organization gives rise to two different types of modules: *components* and *layouts*.

A *component* contains rules for matching tokens. It corresponds to a single lexical state in a traditional lexer.

A *layout* contains rules for transitioning amongst components by matching meta-tokens.

For example, *Figure 1.1* shows a possible organization of a Matlab lexer. A (blue) layout – *Matlab* – refers to three (green) components – *Base*, *String*, and *Comment*. Each of the components describes a lexical state and the layout describes their interaction.

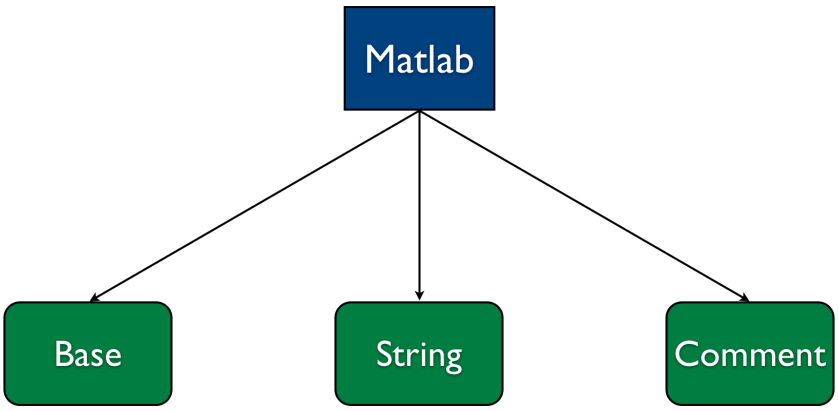


Figure 1.1 Layout (blue) and components (green) for Matlab

This division of specifications into components and layouts promotes modularity because components are more reusable than layouts. For example, many languages have the same rules for lexing strings, numbers, comments etc. Factoring out the more reusable components from the more language-specific layouts reduces coupling.

For example, *Figure 1.2* extends *Figure 1.1* to show how a second layout – *Lang X* – might share some components in common with the original layout – *Matlab*. In particular, the

other lexer might treat strings the same way, but comments differently. If so, it could reuse the same string component, but create its own comment component.

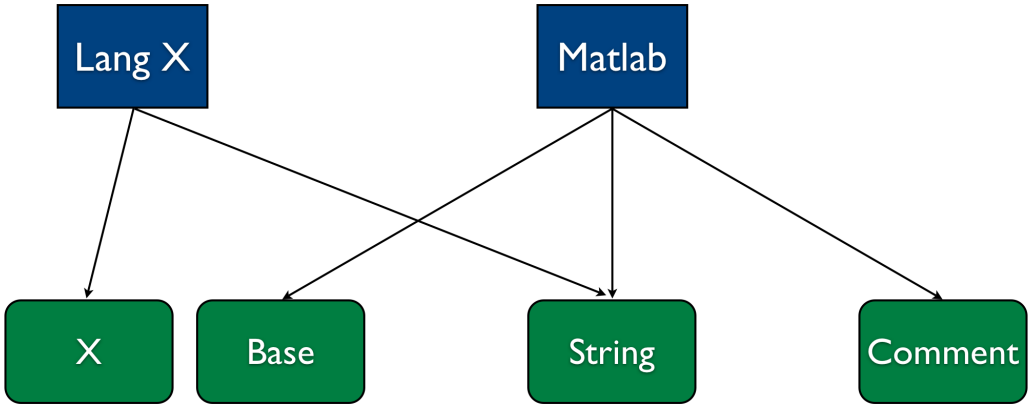


Figure 1.2 Two layouts sharing components

We have found that this sharing of modules is very useful in practice. Components, in particular, are very reusable. For example, the layouts of MetaLexer languages – component and layout – use many of the same components (*Section 8.3.1*). Additionally, the components of the abc language inherit many of the same helper components (*Section 8.2*).

1.1.2 Key Feature: Inheritance

MetaLexer uses multiple inheritance to achieve extensibility and modularity.

For example, *Figure 1.3* shows how inheritance can be used to extend an existing lexer. Given an existing Matlab lexer, one might wish to extend the syntax of strings, perhaps allowing new escape sequences. One could do this by inheriting the *String* component in a new *String++* component which adds the new escape sequences. Then one could inherit the *Matlab* layout in a new *Matlab++* layout which replaces all references to *String* with references to *String++*. Note that this process would leave the original Matlab lexer (i.e. layout and components) intact.

On the other hand, *Figure 1.4* shows how inheritance can improve modularity by factoring out useful "helper" fragments into separate layouts/components. In this case, since the

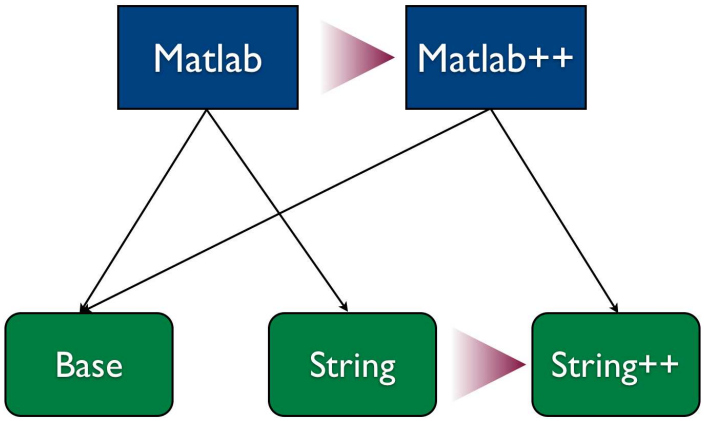


Figure 1.3 Using inheritance to extend the syntax of Matlab strings

components *Base* and *Class* share rules in common (keywords, and comment syntax), these rules have been factored out into "helper" components (shown with dashed borders) that are then inherited by both true components. The same modularity can be achieved with layouts.

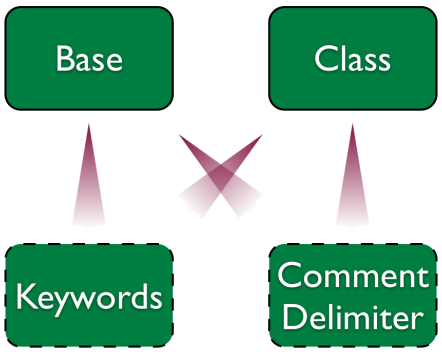


Figure 1.4 Using inheritance to improve modularity

The inheritance mechanism in MetaLexer is an extension of basic textual inclusion. Consequently, anything that can be achieved using inheritance, can also be achieved by judicious duplication and merging of existing files. In particular, common ancestors are not shared, but duplicated.

Both layouts and components support multiple inheritance.

1.1.3 Key Feature: Cross-Platform Functionality

In designing and implementing MetaLexer, great care was taken not to tie it to a specific language or toolset. This effort was threefold (details in *Section 6.10*).

First, the syntax and features of MetaLexer are not closely tied to those of an existing LSL. For example, rather than providing all of the same directives as JFlex, MetaLexer provides an **`%option`** directive that passes directives through to the underlying lexer. It also avoids LSL-specific quirks and advanced features.

Second, the features of MetaLexer are not tied to those of an existing PSL. For example, it was not assumed that the meta-lexer could peak into the token stream, which is very PSL-specific.

Third, the AIL of MetaLexer is not fixed. In fact, it should be possible to use nearly any procedural or object-oriented language.

1.2 Examples

Mixed language programming is not a new concept. Since the early days of C, programmers have been inserting blocks of assembly with *asm* regions [KR78]. Around the same time, C was being embedded in Lex specifications [LS75]. However, mixed language programming is growing in popularity, especially in the web development community. HTML documents often contain embedded JavaScript[2]. Languages like ASP[3] and JSP[4] go a step further and mix general purpose languages with HTML.

In all of the examples above, the paired languages exist independently and are combined after-the-fact. However, this need not be the case. We can also view more homogeneous languages through the lens of mixed language programming. Javadoc[5], for example, does not exist independently of Java. It is, however, a separate language with its own lexing and

[2] http://www.w3.org/TR/html4/interact/scripts.html
[3] http://www.asp.net/
[4] http://java.sun.com/products/jsp/
[5] http://java.sun.com/j2se/javadoc/

parsing rules. Similarly, the aspect language of AspectJ [KHH+01] has no independent implementation, but it can be viewed as its own language, mixed with the Java language in AspectJ.

In the extreme, we can view data-type literals as their own languages, mixed with the more general language that contains them. For example, Ruby[6] contains regular expression literals. Clearly, they are lexed and parsed differently from the rest of Ruby. Similarly, most languages contain string literals. String literals may have very simple lexing and parsing rules, but that does not mean that they cannot be viewed as their own language.

MetaLexer is particularly well-suited to dealing with mixed language lexing. It allows the lexers to be developed separately and then combined. This makes specifications both easier to understand and more modular. For example, if the C programming language is to be used in two different mixed language environments, then the same modules can be used in both cases. More detailed examples are described below and in *Chapter 8*.

1.2.1 Javadoc

Though Javadoc does not exist independently of Java, it possesses its own syntax rules and even its own compiler (the eponymous javadoc). Indeed, one can imagine writing separate, standalone lexers for Javadoc and Java. In some ways, this is the simplest approach – the Java lexer allows the Java parser to consider Javadoc blocks opaque and vice versa.

For example, *Figure 1.5* shows a typical combination of Java and Javadoc. A Java compiler can ignore the green regions and a Javadoc compiler can ignore the blue regions.

Of course, the regions are not perfectly separate. If they were, there would be little incentive to put them in the same file. Rather, Javadoc tags reference Java elements and the two must be kept synchronized. Hence, while we desire the modularity of separate lexers, we require the error-checking capabilities of a unified syntax tree. With MetaLexer, exactly this is possible.

[6] http://www.ruby-lang.org/

```
/**
 * A sample Java application.
 */
public class HelloWorld {
    /**
     * This program prints "Hello World!" and exits.
     * @param args Command-line arguments.  Not used.
     */
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 1.5 Java/Javadoc as mixed language programming

1.2.2 AspectJ

AspectJ is an even more interesting use case because it requires both extension and modularity. AspectJ can be regarded as a mix of three languages: the aspect language, the pointcut language, and Java [HdMC04]. In MetaLexer, the three can be specified separately and then combined with a common layout.

The aspect language is an extension of Java. It introduces new keywords, such as *aspect*, *pointcut*, *before*, *after*, and *around*.

The pointcut language is completely separate from both the aspect language and Java. It has its own lexing rules that allow it to express a variety of patterns.

Figure 1.6 shows a sample AspectJ file. Initially, we are in the Java language for the package and import statements. Upon seeing the *aspect* keyword, some additional tokens, and a left brace, we switch to the aspect language. We revert to the Java language upon seeing an unmatched right brace, but before that we see other language regions. In this example, we switch to the pointcut language after seeing the *before* keyword, but more generally, we could be in a per declaration (*percflow*, *perthis*, etc) or a pointcut declaration (*pointcut*). Note that if conditions within the pointcut language are written in the Java language (delimited by the if keyword and an (unbalanced) right parenthesis). We revert to the aspect language upon seeing a left brace (i.e. the beginning of the corresponding

action). The text of the action, delimited by left and right braces is written in the Java language. Similarly, the text of the nested class, delimited by the *class* keyword and an unbalanced right brace, is written in the Java language.

```
package foo;

aspect Aspect {
    before() : execution(* Clazz.* (..)) || if(Clazz.flag) {
        System.out.println("Hello");
    }

    class Clazz {
        static boolean flag = false;

        public static void foo() {
            flag = !flag;
        }
    }
}
```

Figure 1.6 AspectJ as mixed language programming

In practice, the implementation is more complicated than described above. Details can be found in *Section 8.2*.

1.3 Contributions

In creating MetaLexer, we have identified and filled a gap in the existing extensible compiler toolset. We have designed our new approach around three key features – abstraction of lexical state transitions, multiple inheritance, and cross-platform support – and established a pattern for future implementations. Finally, we have used MetaLexer to build lexers for real-world languages.

1.3.1 Reference Implementation

Two different code generation engines for MetaLexer specifications are available online[7]. One produces JFlex code that can be compiled into Java classes (see *Section 5.1*) and the other produces flat (i.e. inheritance-free) MetaLexer (see *Section 5.2*). Source code and binaries are available for both.

1.3.2 JFlex Translator

To help developers get started with MetaLexer, we have also provided a tool for translating existing JFlex lexer specifications into MetaLexer (see *Section 5.1*). It should be noted that, while the MetaLexer produced by the translator is guaranteed to be correct, it is not guaranteed to be written in proper MetaLexer style.

1.3.3 Lexer Specification for McLab

The McLab project[8] being developed by the Sable Lab will eventually be a framework for building optimizing compilers for scientific languages (e.g. Matlab[9], SciLab[10], and Modelica[11]). It is beginning, however, by building a single optimizing compiler for a slightly simplified version of Matlab called Natlab.[12] We have built the Natlab lexer using MetaLexer and a colleague, Toheed Aslam, is using it to create the lexer for an extended language called AspectMcLab[13]. See *Section 8.1* for details.

[7] http://www.cs.mcgill.ca/metalexer/
[8] http://www.sable.mcgill.ca/mclab/
[9] http://www.mathworks.com/products/matlab/
[10] http://www.scilab.org/
[11] http://www.modelica.org/
[12] It omits, among other things, the convoluted command syntax, which complicates both lexing and parsing.
[13] http://www.sable.mcgill.ca/mclab/

1.3.4 Lexer Specification for AspectJ

As described above (*Section 1.2.2*), AspectJ is an ideal candidate for MetaLexer lexing. As an experiment, we have replaced the lexers for abc [ACH+05], an open-source AspectJ implementation, and two of its extensions – Extended AspectJ (eaj) and Tracematches (tm). Details are provided in *Section 8.2*.

1.3.5 Lexer Specification for MetaLexer

Finally, to show our confidence in MetaLexer, we have bootstrapped it. The lexer classes used by the MetaLexer frontend (i.e. for the layout and component languages) are actually generated from MetaLexer specifications. See *Section 8.3* for details.

1.4 Organization of Thesis

The remainder of the thesis is organized as follows. *Chapter 2* provides some background material on lexing and parsers for readers less familiar with the domain. It can be skipped. *Chapter 3* describes the syntax of the MetaLexer LSL. It contains several examples which will make it easier to understand concepts introduced in later chapters. *Chapter 4* describes the new semantics of MetaLexer – those that differ from JFlex and other existing LSLs. *Chapter 5* provides instructions for running the tools that translate specifications to and from MetaLexer. Once the mechanics have been explained, *Chapter 6* highlights some of the design decisions behind MetaLexer and *Chapter 7* describes some of the implementation issues. *Chapter 8* presents three case studies comparing MetaLexer to JFlex: McLab, abc, and MetaLexer itself. *Chapter 9* describes previous work in this field and, in particular, other approaches that were considered and rejected. *Chapter 10* summarizes the thesis and its conclusions and *Chapter 11* describes logical directions for future work. Finally, *Appendix A* provides a glossary of acronyms used in the thesis, *Appendix B* is a reference for developers who wish to modify the MetaLexer source code, and *Appendix C* contains the specification for the MetaLexer lexer, as both an example and a definition.

Chapter 2

Background

This chapter provides some background information on lexing and parsing for readers who are less familiar with the domain.[1]

2.1 Parsing

Intuitively, parsing is the process of extracting meaning from a body of text[2]. For example, to a human, the sequence `5 + 3 * (2 + 4)` looks quite meaningful. To a computer, however, it is no different from any other sequence of 15 characters[3]. Hence, we need to give the computer some way to extract the arithmetic structure that we know is present. In particular, we want the computer to build the expression tree shown in *Figure 2.1*.

A parser is a computer program that extracts structure from bodies of text. To be more precise, we will need to define a few terms.

An *alphabet* is a set of symbols. For example, the English alphabet we use every day is a set of 26 symbols (52 if we include uppercase). Similarly, the digits `0,1,2,3,4,5,6,7,8,9` form an alphabet.

[1] For greater detail we recommend [App98] and [Mar03].

[2] Of course, in a more general context, the input need not be text – it can be any sequence of symbols.

[3] Did you remember to count the spaces?

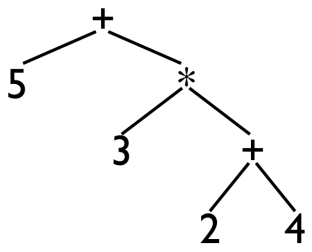


Figure 2.1 An expression tree for "5 + 3 * (2 + 4)"

A *string* over an alphabet is a finite sequence of symbols from that alphabet. For example, `214` is a string over the alphabet of decimal digits.

A *formal language* is a set of strings over a finite alphabet. For example, the strings {`1`, `11`, `111`, ...} are a formal language over the alphabet of decimal digits.

A *grammar* is a succinct description of a formal language[4]. It captures structure in such a way that the structure of any string in the language can be recovered using the grammar. For example, a grammar for the language of all valid arithmetical expressions would encapsulate the information needed to turn flat expressions into expression trees.

More precisely then, a parser is a computer program that encapsulates the grammar of a formal language. Given a string in that language, it can extract the structure of the text.

2.2 Lexing versus Parsing

The previous section neglected to define *symbol*. Through examples, it was implied that a symbol is simply a character, but this need not be the case. In fact, anything with a finite representation will do. In particular, there is no reason that we cannot use an entire string as a symbol. For example, nearly all programming languages contain identifiers (i.e. names). There are two ways to look at identifiers: they can be strings or they can be symbols. That is, the name "foo" may be regarded as either a string of symbols ('f', 'o', 'o') or as a symbol

[4]A more formal definition is beyond the scope of this chapter.

of its own (identifier).

This paradigm shift actually has important practical implications. Grouping multiple characters into each symbol reduces the size of the input to the parser. For example, “foo bar” is seven characters. However, if we are only interested in the identifier level of granularity, then the input consists of only two symbols. Therefore, if we can break the sequence of characters into symbols more quickly than we can parse, then we can reduce our total execution time.

Lexing is the process of breaking a body of text into symbols (usually called tokens in the lexer). In order to remain simpler, and thus faster, than parsing, lexers are restricted to a simple class of formal languages called regular languages.

While lexing is not strictly necessary, it does reduce the time required for parsing and simplify parser specifications (since the resulting symbols are much more abstract).

2.3 Traditional Lexing Tools

The first widely used lexical specification language (LSL) was lex [LS75], developed by Mike Lesk and Eric Schmidt at Bell Laboratories. It was designed to work closely with the fledgling C programming language [KR78] and yacc parser generator [Joh75], also from Bell Laboratories.

Lex was re-implemented by the GNU project as Flex[5]. Flex has supplanted the proprietary Lex and is now the de-factor standard for lexing in C/C++.

Several LSLs exist for Java, but the most popular are JLex[6] and its successor JFlex[7].

All of these tools provide approximately the same functionality, though some of the newer ones have better performance and include more advanced features.

[5] http://flex.sourceforge.net/
[6] http://www.cs.princeton.edu/ appel/modern/java/JLex/
[7] http://jflex.de/

2.4 Lexical States

Sometimes the boundary between lexing and parsing is unclear (unsurprisingly, since it is arbitrary). One particularly common case is that of nested comments. Since comments can appear virtually anywhere in the syntax of a programming language, a grammar that includes comments is bloated and hard to read. Fortunately, most comments have no semantic effect and can safely be ignored. If comments are filtered out of the input by the lexer – which does not need to consider context and so can avoid specifying them repeatedly – then the parser can be made much simpler. Unfortunately, because nested comments require balanced start- and end-delimiters, they cannot be captured by regular expressions – a new construct is needed.

Nested comments can be handled by introducing *lexical states*. Each lexical state of a lexer has a different set of lexing rules. That is, the current state of the lexer determines how subsequent input will be interpreted. If the designer of the lexer can programmatically control the transitions between these lexical states, then they increase the power of the lexer.

For example, in the case of nested comments, a lexer could contain two lexical states – one for nested comments, and one for the other rules. Upon encountering a start-delimeter, the lexer would transition to the nested comment transition lexical state. It would then track the balancing of delimiters in a state variable of the lexer and postpone the transition back until balance was achieved. In this way, it could hide the contents of all nested comments from the parser but handle all other input as usual.

In MetaLexer, these lexical states become components and their interactions are governed by layouts, rather than by action implementation languate (AIL) code in actions and helper methods.

Chapter 3

MetaLexer Syntax

MetaLexer actually consists of two specification languages: one for components and one for layouts. Components take the place of lexical states; they contain the lexical rules. Layouts specify the interaction of the components, the transitions between the lexical states. This chapter describes the syntax of both languages.

3.1 Example

We begin with an example. Suppose we want to write a parser for Java property files. A property consists of a key and a value, separated by an equals sign. The key is an alphanumeric identifier and the value is a string that starts after the equals sign and ends at the end of the line. Each line contains a key-value pair, a comment (from '#' to end-of-line), or whitespace. *Listing 3.1* shows a sample properties file. It specifies three key-value pairs: (name, 'properties'), (date, '2009/09/21'), and (owner, 'root'). Everything else is ignored.

Clearly, we could extract all of this information within the lexer, but to be more illustrative we will tokenize the file for a hypothetical parser.

Ultimately, we will create a number of components and join them together using a layout.

```
#some properties
name=properties
date=2009/09/21

#some more properties
owner=root
```

Listing 3.1 Syntax Example – A Properties File

For now, we'll start with a single component that corresponds closely with the description above. *Listing 3.2* shows the *key* component that will be the workhorse of our lexer. This listing is fairly intuitive. First, we specify the name of our component (**`%component`**). Then we list methods that we plan to use but we expect to be defined elsewhere (**`%extern`**). After a separator, we specify lexical rules. As one might expect, **`%%inherit`** pulls in the macros we need from another file, in this case *macros.mlc*. Finally, we note that one of the rules is followed by an extra identifier, *ASSIGN*. This is a meta-token; it will be processed by the layout to determine if a transition is necessary.

```
%component key

%extern "Token symbol(int)"
%extern "Token symbol(int, String)"
%extern "void error(String) throws LexerException"

%%

%%inherit macros

{lineTerminator} {: /*ignore*/ :}
{otherWhitespace} {: /*ignore*/ :}
"=" {: return symbol(ASSIGN); :} ASSIGN
%:
{identifier} {: return symbol(KEY, yytext()); :}
{comment} {: /*ignore*/ :}
%:
<<ANY>> {: error("Unexpected character '" + yytext() + "'"); :}
<<EOF>> {: return symbol(EOF); :}
```

Listing 3.2 Syntax Example – key.mlc

MetaLexer rules are very similar to JFlex rules except for three main differences. First,

MetaLexer introduces a new (top-level) **<<ANY>>** pattern which is used to designate the catchall rule (described below). Second, each rule may optionally be followed by a meta-token declaration. Whenever, the pattern is matched, in addition to executing the action code, the component will send the meta-token to the coordinating layout. Meta-tokens do not need to be declared, nor do they need to be unique. Finally, for disambiguation reasons, colons have been added inside the curly brackets (see *Section 6.10.1* for an explanation).

The *key* component has a rule for constructing key tokens, but not for constructing value tokens. For that, we will need another component. *Listing 3.3* shows the *value* component, wherein entirely different lexical rules apply. It has many of the same features as *Listing 3.2* – a component name, external declarations, inheritance of macros, meta-tokens – but it also has a new construct, an **%append** block. The append block means that the goal of the whole component is to build up a single token. Instead of returning tokens themselves, the rules call *append()* to concatenate strings onto a shared buffer. When the component is 'complete' (as decided by the layout), the body of the **%append** block will be executed and a single token will be returned.

```
1  %component value
2
3  %extern "Token symbol(int, String, int, int, int, int)"
4
5  %append{
6     return symbol(VALUE, text, startLine, startCol, endLine, endCol);
7  %append}
8
9  %%
10
11 %%inherit macros
12
13 {lineTerminator} {: :} LINE_TERMINATOR
14 %:
15 %:
16 <<ANY>> {: append(yytext()); :}
17 <<EOF>> {: :} LINE_TERMINATOR
```

Listing 3.3 Syntax Example – value.mlc

Listing 3.4 shows the *macros* helper component that is inherited by both *key* and *value*. The **%helper** directive indicates that the module is only to be inherited, never used directly.

Notice how it encapsulates the code shared by the *key* and *value* components so that the code does not have to be duplicated. The macros themselves are just as in JFlex.

```
1 %component macros
2 %helper
3
4 lineTerminator = [\r\n] | "\r\n"
5 otherWhitespace = [ \t\f\b]
6 identifier = [a-zA-Z][a-zA-Z0-9_]*
7 comment = #[^\r\n]*
```

Listing 3.4 Syntax Example – macros.mlc

Finally, *Listing 3.5* shows the *properties* layout that joins everything together. It is the layout that we will compile into a working lexer. Like a normal LSL specification (Flex, JFlex, etc), the layout begins with a free-form header. In MetaLexer, however, the header is split in two. The first section is specific to the current layout, whereas the second section will be inherited by any layout that extends this one.

After the header sections comes the option section. It begins with the layout name (**`%layout`**) and the lexer options (**`%option`**). Each lexer option is given an identifier so that it can be deleted or replaced in an extension of the lexer. The string part is passed directly to the underlying LSL. Following the options are declarations in the AIL (surrounded by `%`{ and `%`}). These methods will be added directly to the lexer class. Each one is shared with the components of the lexer via a **`%declare`** directive. The **`%lexthrow`** directive reflects the fact that, by calling *error(String)*, a lexer action may raise a *LexerException*. At the end of this section, the components to be used are imported (**`%component`**) and a start component is specified (**`%start`**). Until a transition occurs, the lexer will remain in the start component.

The last section contains embeddings (i.e. transitions). In this case, if an *ASSIGN* meta-token is seen while in the *key* component, then the lexer will transition to the *value* component. It will remain there until a *LINE_TERMINATOR* meta-token is seen and then transition back to the *key* component.

In general, an embedding may be read as *"When in component HOST, upon observing meta-pattern START, transition to component GUEST. Transition back upon observing*

meta-pattern END.”

```
1  package properties;
2  %%
3  import static properties.TokenTypes.*;
4  %%
5  %layout properties
6
7  %option public "%public"
8  %option final "%final"
9  %option class "%class PropertiesLexer"
10 %option unicode "%unicode"
11 %option function "%function getNext"
12 %option type "%type Token"
13 %option pos_line "%line"
14 %option pos_column "%column"
15
16 %declare "Token symbol(int)"
17 %declare "Token symbol(int, String)"
18 %declare "Token symbol(int, String, int, int, int, int)"
19 %{
20    private Token symbol(int symbolType) {
21       return symbol(symbolType, null);
22    }
23    private Token symbol(int symbolType, String text) {
24       return new Token(symbolType, text, yyline + 1, yycolumn + 1,
             yyline + 1, yycolumn + yylength());
25    }
26    private Token symbol(int symbolType, String text, int startLine, int
          startCol, int endLine, int endCol) {
27       return new Token(symbolType, text, startLine + 1, startCol + 1,
             endLine + 1, endCol + 1);
28    }
29 %}
30
31 %declare "void error(String) throws LexerException"
32 %{
33    private void error(String msg) throws LexerException {
```

```
34        throw new LexerException(msg);
35     }
36 %}
37
38 %lexthrow "LexerException"
39
40 %component key
41 %component value
42
43 %start key
44
45 %%
46
47 %%embed
48 %name key_value
49 %host key
50 %guest value
51 %start ASSIGN
52 %end LINE_TERMINATOR
```

Listing 3.5 Syntax Example – properties.mll

Obviously, this simple example does not exercise the full syntax of MetaLexer. Read on for a more complete description.

3.2 Components

Each component is divided into two sections. First there is an option section containing configuration details and then there is a rule section. The sections are separated by a section separator, **%%**.

Unless otherwise indicated, each item listed below should begin on a new line.

3.2.1 Option Section

The option section consists of a **`%component`** directive, followed by a mixture of other directives and code regions (order unimportant), followed by a list of macro declarations.

Name

%component *name* – EXACTLY 1 – The name of the component must correspond to the name of the file. The component X must appear in the file X.mlc (case-sensitive); the component X.Y must appear in the file Y.mlc in the directory X (case-sensitive).

Directives

%helper – AT MOST 1 – If this directive is present, then the component can be inherited by other components but not used in a layout. Checks related to missing declarations will be postponed until the component is incorporated into an inheriting component.

The following directives relate to lexical states. **These are advanced directives and should not be used under normal circumstances.**

%state *name, name, ...* – ANY NUMBER – This directive comes from JFlex. In rare circumstances, it is necessary to use a lexical state in place of a component. **`%state`** declares such a state. In particular, it declares an *inclusive* state. This means that, when the lexer is in the declared lexical state, only those rules that are labelled with its name and those that are unlabelled will be considered. An inclusive state called *YYINITIAL* is declared by default.

%xstate *name, name, ...* – ANY NUMBER – This directive comes from JFlex. In rare circumstances, it is necessary to use a lexical state in place of a component. **`%state`** declares such a state. In particular, it declares an *exclusive* state. This means that, when the lexer is in the declared lexical state, only those rules that are labelled with its name (but not those that are unlabelled) will be considered.

%start *name* – AT MOST 1 – In cases where lexical states have been declared using (**`%state`** or **`%xstate`**), it may be desirable to start in one of the declared states rather than in the default *YYINITIAL* state. This directive indicates in which state the component should start. If this directive is absent, then the component will start in the automatically declared *YYINITIAL* state.

The following directives relate to external requirements of the component.

%extern *"signature"* – ANY NUMBER – This directive indicates that the component expects any layout making use of it to provide an entity with the specified signature. In particular, the layout must include **`%declare`** *"signature"*.

%import *"class"* – ANY NUMBER – This directive indicates that the top-level lexer should import/include/require (depending on the AIL; e.g. C for Flex, Java for JFlex, etc) the specified class/module/file. Unlike an **`%extern`** directive, the **`%import`** directive actually effects the change it requires. That is, it is sufficient on its own – no additional import is required in the layout.

The following directives relate to exceptions that might be thrown by the component.

%lexthrow *"exception type"*, ... – ANY NUMBER – This directive indicates that an action (or the special append action method) may throw an exception of one of the listed types.

%initthrow *"exception type"*, ... – ANY NUMBER – This directive indicates that the code in an **`%init`** block may throw an exception of one of the listed types.

Code Regions

%{ *declaration code* **%}** – ANY NUMBER – This code region is for declaring fields, methods, inner classes, etc.

%init{ *initialization code* **%init**} – ANY NUMBER – This code region is for initializing the entities declared in %{ %} blocks. For example, if the AIL were Java or C++, then this code would be inserted in the constructor of the lexer class.

%append{ *method code* **%append**} – AT MOST 1 – An append block is both a directive and a code region. First, its presence indicates that the component is an append component. This means that an *append(String)* method will be available in all actions of the component. Second, its code is the body of a special append action method that will be called when appending is finished (see *Section 4.9* for details). The method is like any other action block and may (optionally) return a token. It will receive integer parameters *startLine, startCol, endLine*, and *endCol* and string parameter *text* indicating the position and contents of the text passed to *append(String)*. *The positions will be indexed in the same way as the underlying LSL (e.g. zero-indexed for JFlex).*

%appendWithStartDelim{ *method code* **%appendWithStartDelim**} – AT MOST 1 – An appendWithStartDelim is very similar to an append block. It indicates that the component is an append component (i.e. *append(String)* is available) and creates a special append action method that will be called when appending is finished. However, when the append action method is called, the arguments it receives will incorporate the start delimiter created by *appendToStartDelim(String)* (see *Section 4.9.1* for details). In particular, the values of *startLine*, *startCol* and *text* will be different from what they would be in an otherwise identical append block. *The positions will be indexed in the same way as the underlying LSL (e.g. zero-indexed for JFlex).*

Macros

macro = *regex* – ANY NUMBER – This line declares a macro (a named regular expression) with the specified name and value. Regular expressions are as in JFlex. The entire declaration must appear on a single line.

3.2.2 Rule Section

The rules section is a mix of rules and inheritance directives.

A rule is of the following form.

pattern {**:** *action code* **:**} *meta-token*

An inheritance directive indicates that another component should be inherited. It is of the following form.

%%inherit *component*

If the character sequence "%%inherit" appears in a regular expression, it must be quoted to distinguish it from the directive.

Each inheritance directive is immediately followed by zero or more delete directives, which prevent certain rules from being inherited. They are of the following form.

%delete <*state, state, . . .* > *pattern*

If the character sequence "%delete" appears in a regular expression, it must be quoted to distinguish it from the directive.

If a rule with the given pattern appears in one of the listed states of the inherited component, then it is not inherited. In most specifications, the state list will be empty – this is equalivalent to a state list containing only the default *YYINITIAL* lexical state.

Rule Order

As in JFlex, if two different patterns match the input, then the longer match is chosen. If there is more than one longest match, then textual order is used as a tie-breaker. Clearly, this gets more complicated when multiple inheritance is incorporated. To reduce complexity, MetaLexer recognizes and separates three types of rules.

1. **Acyclic** rules can match only finitely many strings. Conceptually, their minimal DFAs are acyclic.

2. **Cyclic** rules are neither Acyclic nor Cleanup rules.

3. **Cleanup** rules are either catchall – **`<<ANY>>`**– or end-of-file – **`<<EOF>>`**– rules.

Acyclic rules are listed first, followed by a group separator – **`%:`**, then cyclic rules are listed, followed by a group separator, and finally cleanup rules are listed. If the cleanup rules are absent, then the second group separator may be omitted. If both cleanup and cyclic rules are absent, then both group separators may be omitted. Otherwise, all group separators are required, even around empty groups.

A new Acyclic-Cyclic-Cleanup group begins after the section separator – **`%%`** – and after each **`%%inherit`** directive.

See *Section 6.7* for the importance of and the rationale behind this distinction between different types of rules.

3.3 Layouts

Each layout is divided into four sections: the local header, the inherited header, the options section, and the rule section. The sections are separated by section separators, **`%%`**.

Unless otherwise indicated, each item listed below should begin on a new line.

3.3.1 Local Header

The local header is a block of free-form text that will be inserted at the top of the generated lexer class (i.e. the file generated by the underlying LSL (e.g. JFlex) rather than the file generated by MetaLexer). It is not incorporated into inheriting components. It is generally used for something like a package declaration – something that will probably change in an inheriting component.

3.3.2 Inherited Header

The inherited header is another block of free-form text. It will be inserted just below the local header at the top of the generated lexer class. It is exactly like the local header except that it will be incorporated into inheriting components. It is generally used to declare imports, macros, etc.

3.3.3 Options Section

The option section is very similar to the corresponding section in a component. It consists of a **`%layout`** directive, followed by a mixture of other directives and code regions (order unimportant).

Name

%layout *name* – EXACTLY 1 – The name of the layout must correspond to the name of the file. The layout X must appear in the file X.mll (case-sensitive); the layout X.Y must appear in the file Y.mll in the directory X (case-sensitive).

Directives

%helper – AT MOST 1 – If this directive is present, then the layout can be inherited by other layouts but not compiled into a lexer. Checks related to missing declarations will be postponed until the layout is incorporated into an inheriting layout.

%option *name "lexer option"* – ANY NUMBER – This directive inserts its text, verbatim, in the option section of the generated lexer specification. The name is included so that the option can be filtered out by inheriting layouts. Names must be unique.

%declare *"signature"* – ANY NUMBER – This directive indicates that the layout will satisfy any referenced components with **`%extern`** *"signature"*.

The following directives relate to exceptions that might be thrown by the lexer.

%lexthrow *"exception type", ...* – ANY NUMBER – This directive indicates that an action (or a special append action method) may throw an exception of one of the listed types.

%initthrow *"exception type", ...* – ANY NUMBER – This directive indicates that the code in an **`%init`** block may throw an exception of one of the listed types.

The following directives relate to the use of components.

%component *name, name, ...* – AT LEAST 1 – This directive declares that the layout will make use of the named components.

%start *name* – EXACTLY 1 – This directive indicates in which component the layout will start.

Code Regions

%{ *declaration code* **%**} – ANY NUMBER – This code region is for declaring fields, methods, inner classes, etc.

%init{ *initialization code* **%init**} – ANY NUMBER – This code region is for initializing the entities declared in **`%`**{ **`%`**} blocks. For example, if the AIL were Java or C++, then this code would be inserted in the constructor of the lexer class.

3.3.4 Rules Section

The rules section is a mix of embeddings and inheritance directives.

An embedding is of the following form (order matters).

%%embed
%name *name*

%host *component***,** *component***,** *. . .*
%guest *component*
%start *meta-pattern*
%end *meta-pattern*
%pair *meta-token***,** *meta-token*

Zero or more **%pair** lines may be included.

The embedding is named so that inheriting layouts can exclude it, if necessary. The rest may be read as *When in component HOST, upon observing meta-pattern START, transition to component GUEST. Transition back upon observing meta-pattern END.* For each pair, if the first element is observed, the next occurrence of the second element is suppressed (i.e. not matched).

An inheritance directive indicates that another layout should be inherited. It is of the following form.

%%inherit *layout*

Each inheritance directive is immediately followed by zero or more unoption, replace, and unembed directives (in that order).

Unoption directives filter out options from inherited layouts. They are of the following form.

%unoption *name*

Replace directives replace all references to one component with references to another. This is very useful when a new layout uses an extended version of a component used by an inherited layout (as in *Figure 1.3*). They are of the following form.

%replace *component, component*

Unembed directives filter out embeddings from inherited layouts. They are of the following form.

%unembed *name*

Meta-Patterns

The basic meta-patterns are meta-tokens (from component rules) and regions. A region is a component name surrounded by percent-signs. It indicates that a component with the given name has just been completed.

The basic meta-patterns can be included in a classes – space-separated lists surrounded by square brackets. Normal classes are simply shorthand for alternation. Negated classes (those with a caret just inside the open square bracket) match any single meta-token or region not listed in the class. The special **<ANY>** class matches any single meta-token or region.

The **<BOF>** meta-pattern matches the beginning of the meta-stream (i.e. the stream of meta-tokens and regions passed to the meta-lexer by the lexer).

Finally, parentheses, juxtaposition, alternation, $+$, $*$, and ? work as they do in regular expressions.

3.4 Comments

Both layouts and components support Java-style single-line (`//`) and multi-line (`/* */`) comments.

Chapter 4

MetaLexer Semantics

The previous chapter described the syntax of MetaLexer. This chapter will describe the semantics, focusing on differences between MetaLexer and JFlex.

4.1 JFlex Semantics

A JFlex lexer has one key method: *nextToken()*. When called, the method reads characters from the input stream, attempting to match a lexer rule and return a token. This process is summarized in *Listing 4.1*. Within the current lexical state, all rules are tested in order. The rule matching the longest prefix of the input is selected. If there is a tie, then the first to appear textually is selected (accomplished in this case by simply not updating the *matchedRule* variable). Then the input pointer is advanced past the longest match and the corresponding action is executed.

Notice that the loop has no exit condition and the method has no return statement. It is up to the action code to break out of the loop by returning a token. If an action does not return a value, then the loop will perform another iteration. Any number of rules may be matched before a token is returned – there is no one-to-one correspondence.

```
1  public Token nextToken() {
2     while(true) {
3        matchedRule = null
4        maxString = null
5        for each rule r in lexicalState {
6           s = prefix of input matched by r
7           if(s longer than maxString) {
8              matchedRule = r
9              maxString = s
10          }
11       }
12       if(matchedRule != null) {
13          advanceInputPast(maxString);
14          switch(matchedRule) {
15          case rule1:
16             perform action 1
17             break;
18          case rule2:
19             perform action 2
20             break;
21          ...
22          }
23       }
24    }
25 }
```

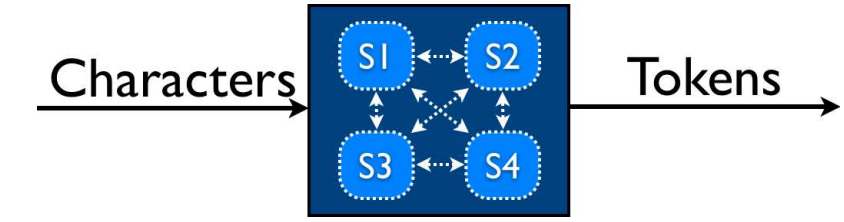
Listing 4.1 Pseudo-Code for the Main JFlex Loop

4.2 Meta-Lexing

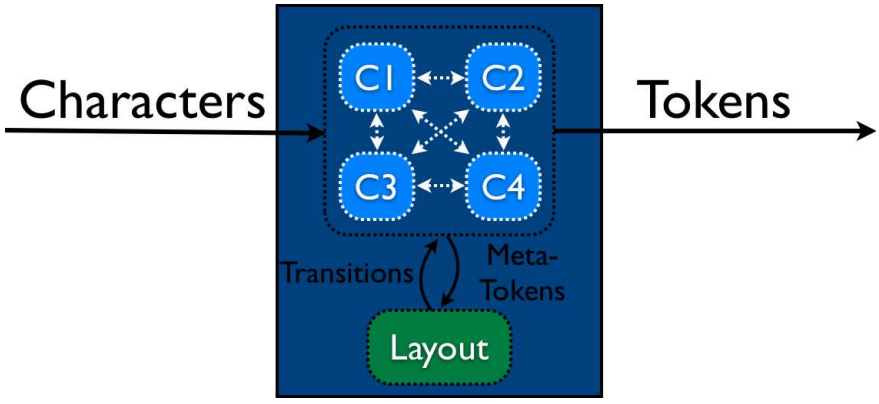
MetaLexer uses the same basic lexing loop as JFlex (see *Listing 4.1*). Lexical rules are tested in order and the earliest occurring longest match is selected. However, MetaLexer uses a different mechanism for partitioning these rules and determining which subset should be used.

Though MetaLexer supports lexical states, a well-written MetaLexer specification will eschew their use. Instead, it will use components to perform the same function and coordinate the transitions between components using a layout. This difference is illustrated in *Figure 4.1*. In *Figure 4.1a*, JFlex is shown reading a stream of characters and producing a stream of tokens. Internally, it moves amongst a number of lexical states that determine

which set of lexical rules will be used. Externally, *Figure 4.1b* is very similar. MetaLexer reads a stream of characters and produces a stream of tokens. However, it uses a different mechanism to determine which set of matching rules will be used. In place of lexical states, it has components. These components, in addition to producing tokens, produce meta-tokens that are consumed by the layout. Based on this stream of meta-tokens, the layout determines which component should be used. This process – choosing the current component based on a stream of meta-tokens – is referred to as *meta-lexing*.



(a) JFlex



(b) MetaLexer

Figure 4.1 A comparison of JFlex and MetaLexer

Observe that the job performed by the layout – reading in a stream of meta-tokens, matching against a list of meta-patterns, and producing a stream of transitions – is very similar to the job performed by a normal lexer. Instead of characters, we have meta-tokens; instead of regular expressions, we have meta-patterns; and instead of tokens, we have transitions. If we imagine converting each meta-token to a character and each transition to a token, then it becomes quite clear how meta-lexing works.

For example, consider the layout fragment shown in *Listing 4.2*. It is part of a Java lexer.

The *base* component lexes the package and import statements that exist outside classes in a Java file. The *class* component lexes what we usually think of as the Java language – fields, methods, statements, expressions, etc. The *string* and *comment* components lex string literals and multi-line comments respectively. Intuitively, we transition from the *base* component to the *class* component when we see a class declaration (i.e. a class keyword, a name and maybe some superclasses/interfaces and an open brace) and back when we see the end of the class declaration (i.e. the *corresponding* closing brace). We transition from the *class* component to the *string* component on an open-quote and back on a close-quote. Similarly, we transition from the *class* component to the *comment* component on an open-comment symbol (*'/*'*) and back on a close-comment symbol (*'*/'*).

More formally, we transition based on sequences of meta-tokens. Of course, there is a clear correspondence between tokens and meta-tokens in this case: *CLASS_KW* is generated by the *class* keyword token, *LCURLY* is generated by the open brace symbol token, etc.

With this in mind, we can read *Listing 4.2* as follows. We see that we should transition from the *base* component to the *class* component upon seeing the meta-pattern *CLASS_KW [^LCURLY]* LCURLY*. If we assign *CLASS_KW* the character 'a' and *LCURLY* the character 'b' then, we can rewrite this meta-pattern as *(a[^b]*b)*. If we were to put this abbreviated rule in a separate lexer, we would attach an action that returned the appropriate transition from *base* to *class*. Similarly, the action attached to the end meta-pattern would return the reverse transition, from *class* back to *base*.

Obviously, this is not all there is to meta-lexing. Astute readers will have noticed that, in the example above, once we have transitioned to the *class* component, we no longer wish to match the rule for *(a[^b]*b)*. We solve this problem by associating rules with specific embeddings. We keep track of the current embedding on a stack and only match patterns that make sense in the current embedding.[1]

If we always know which embedding we have encountered most recently, then we also know which component we are in – the guest component of that embedding. For example, if we have just started the embedding *class_embedding*, then we are currently in the *class*

[1]When the stack is empty, we consider ourselves to be in a special degenerate embedding with no end meta-pattern or pair filter.

```
%%embed
%name class_embedding
%host base
%guest class
%start CLASS_KW [^LCURLY]* LCURLY
%end RCURLY
%pair LCURLY, RCURLY

%%embed
%name string_embedding
%host class
%guest string
%start START_STRING
%end END_STRING

%%embed
%name comment_embedding
%host class
%guest comment
%start START_COMMENT
%end END_COMMENT
```

Listing 4.2 MetaLexer Example

component.

Now, knowing our current embedding and component, we can decide which meta-patterns we need to match. First, we need to watch for the beginning of another embedding – in particular, those embeddings that are hosted by the current component. Second, we need to watch for the end of the current embedding.

For example, if we have just started the embedding *class_embedding*, then we need to look out for any start patterns that begin in *class* (i.e. those for *string_embedding* and *comment_embedding*) as well as the end pattern for *class_embedding*.

In the event that more than one meta-pattern matches, start meta-patterns are preferred to end meta-patterns and earlier start meta-patterns are preferred to later start meta-patterns.

Extraneous meta-tokens, those not matched by any meta-pattern, are discarded – they will not cause errors. They will, however, disrupt any meta-patterns for which prefixes have been matched.

There is one substantial difference between the meta-lexer and a traditional lexer. A traditional lexer, upon determining that a prefix of the input matches a given rule, will postpone the selection of a rule until it has been determined that no rule matches a longer prefix. These are often referred to as 'longest match' semantics. In contrast, the meta-lexer selects a rule as soon as any prefix of the input matches. This corresponds to 'shortest match' semantics.[2]

4.2.1 Pair Filters

In many cases, the meta-lexing procedure described above will be sufficient. However, sometimes we want to ignore certain meta-tokens. In particular, many programming languages use a nested structure delimited by pairs of brackets (e.g. curly braces in Java). To prevent balanced pairs of brackets (or other meta-tokens) from interfering with our meta-lexing, we may wish to remove them from the stream entirely. We accomplish this using pair filters.

At first glance, it is not clear why we need, or even want, pair filters. After all, parenthesis balancing is traditionally the domain of the parser. However, a simple example makes the need readily apparent. Consider an aspect in AspectJ. Outside an aspect, we use the Java lexer, but inside we use the AspectJ lexer. Switching from Java to AspectJ is easy – we just look for the *aspect* keyword. Unfortunately, switching back is harder and *Listing 4.3* shows why. We need to switch back upon seeing a closing brace. However, not just any closing brace will do – we need to find the brace the corresponds to the opening brace at the beginning of the aspect. To do this, we must ignore balanced pairs of braces between the two (e.g. the braces around the advice in *Listing 4.3*).

This example is quite representative. Most modern programming languages have hierarchical structures of this sort, delimited by bracketing tokens. In order to be able to use these delimiters as lexical state boundaries in our lexers, we need to be able to find them.

Pair filters sit between the components and the layout. They act directly on the meta-token

[2] Extraneous meta-tokens are handled by treating each rule as though it began with an implicit **<ANY>***. This replaces an explicit catch-all rule that would always be the shortest match.

```
1  package xyz;
2
3  import foo.*;
4
5  public class Klass {
6     public static void main(String args[]) {
7        //...
8     }
9  }
10 aspect Aspect {
11    void around(): execution(* * (..)) {
12       if(/* ... */) {
13          proceed();
14       }
15    }
16 }
```

Listing 4.3 Pair Filter Example

stream and prevent selected meta-tokens from reaching the layout. If a pair filter is in place, then every meta-token is considered by the filter. If a meta-token is the closing element of a pair and if the pair has been opened but not closed (i.e. only the opening element has been seen), then the meta-token will be suppressed and nothing further will occur. If, on the other hand, the meta-token does not complete a pair, then it will be passed to the layout. For example, *Figure 4.2* shows a number of sample meta-token streams and the effects of the *class_embedding* pair filter thereon. Italicized meta-tokens never reach the layout. In example 1, we begin in the *base* component and transition to *class* upon seeing *CLASS_KW IDENTIFIER LCURLY*. Within the embedding, there is no *LCURLY* to begin a pair, so the *RCURLY* is untouched. In example 2, on the other hand, the first *RCURLY* is removed from the stream because of the *LCURLY* that precedes it. Note that this is exactly what we want – we do not exit the *class* component until we see an *unbalanced RCURLY*. In example 3, the opening *LCURLY* is absent again and so the first *RCURLY* ends the embedding. The second one is extraneous and is discarded.[3] Example 4 generalizes example 2, showing a more elaborate balanced-pair sequence. Once again, the embedding is correctly ended on the final *RCURLY*.

[3] It is important to distinguish between this meta-token, which is read from the stream and then discarded, and the other meta-tokens in this example, which are removed from the stream before they are read.

1. CLASS_KW IDENTIFIER LCURLY RCURLY
2. CLASS_KW IDENTIFIER LCURLY LCURLY *RCURLY* RCURLY
3. CLASS_KW IDENTIFIER LCURLY RCURLY RCURLY
4. CLASS_KW IDENTIFIER LCURLY LCURLY *RCURLY* LCURLY LCURLY *RCURLY RCURLY* RCURLY

Figure 4.2 Streams of meta-tokens after filtering

Pair filters are embedding-specific. For example, in *Listing 4.2*, *LCURLY* and *RCURLY* are only paired in the embedding *class_embedding*, not in *string_embedding* or *comment_embedding*.

4.3 Regions

Meta-tokens are not the only information passed from the components to the layout. Whenever an end meta-pattern is matched, the lexer transitions back to the current embedding's host component. This triggers the generation of a special *region* object corresponding to the component just exited (i.e. the guest).

For example, consider the layout in *Listing 4.2*. If we were in the embedding *class_embedding* and we saw an (unpaired) *RCURLY* meta-token, then we would transition back to the *base* component. Since this would indicate the "completion" of the *class* component, a special *%class%* region would be added to the meta-token stream.

Alternatively, suppose we were writing a lexer for a language with named scopes – bracketed regions preceded by string literal names (e.g. *Listing 4.4*). We could break this construct into two components: one for the string literal and another for the block. The corresponding layout would look something like *Listing 4.5*. Notice that the start meta-pattern of the *named_block* embedding is a region. It will be matched whenever the lexer is in the *base* component and has just completed a *string_literal* component.

Like meta-tokens, each region can be thought of as corresponding to a distinct character. In this way, they can be used in meta-patterns in the same way as meta-tokens.

```
"Region A" {
  ...
};
```

Listing 4.4 Syntax Example – Regions – Language Code

```
%%embed
%name string_literal
%host base
%guest string_literal
%start START_STRING
%end END_STRING

%%embed
%name named_block
%host base
%guest block
%start %string_literal%
%end SEMICOLON
```

Listing 4.5 Syntax Example – Regions – Layout

4.4 Inheritance

Inheritance in MetaLexer has more in common with textual inclusion than with object-oriented (OO) inheritance. An **`%%inherit`** directive instructs MetaLexer to merge the contents of the referenced module into the current module, as if they had been typed directly into the file. Unlike straight textual inclusion, however, MetaLexer splits up the inherited file and distributes its elements to the appropriate parts of the current module. That is, headers go in the header section, options go in the option section, rules go in the rules section, etc. Most inherited elements are inserted at the ends of their corresponding sections. Embeddings and lexical rules are more complicated. They are discussed below.

4.4.1 Embedding Ordering

When a layout inherits another layout, the inherited embeddings are not added to the end of the file. This would be too inflexible. Instead, they are inserted at the location of the

corresponding **%%inherit** directive. This allows the child layout to insert new embeddings both before and after the inherited embeddings. Furthermore, it clarifies the relative positions of the embeddings inherited from different parent layouts.

4.4.2 Lexical Rule Ordering

Lexical rules are more complicated than embeddings. Their position is determined by the position of the corresponding **%%inherit** directive, but they are not inserted at that exact location. Instead, the order of the merged lexical rules is as follows.

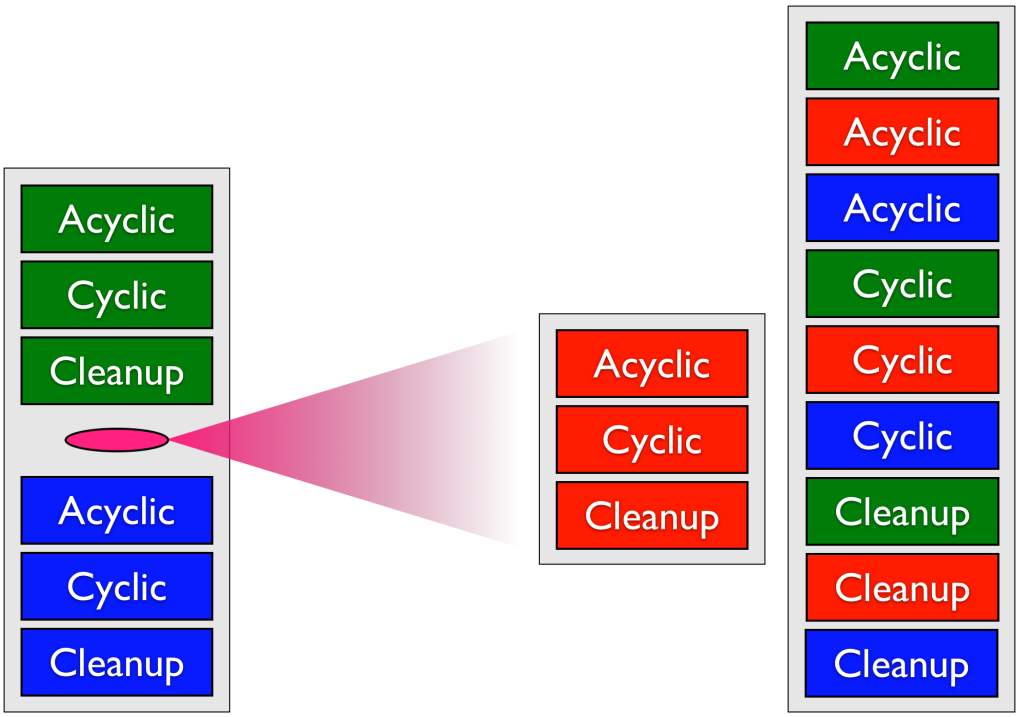
- Child acyclic rules preceding the first **%%inherit** directive.
- Acyclic rules from the first parent component.
- Child acyclic rules preceding the second **%%inherit** directive.
- Acyclic rules from the second parent component.
- *And so on, for subsequent parent components.*
- Child acyclic rules following the last **%%inherit** directive.

- Child cyclic rules preceding the first **%%inherit** directive.
- Cyclic rules from the first parent component.
- *And so on, for subsequent parent components.*
- Child cyclic rules following the last **%%inherit** directive.

- Child cleanup rules preceding the first **%%inherit** directive.
- Cleanup rules from the first parent component.
- *And so on, for subsequent parent components.*

- Child cleanup rules following the last **`%%inherit`** directive.

This conceptual rearrangement, which takes place during the inheritance process, is illustrated by *Figure 4.3*. *Figure 4.3a* shows a component (left) inheriting rules from its parent (right). *Figure 4.3b* shows the order of the rules in the flattened component (i.e. after inheritance).



(a) Before inheritance **(b)** After inheritance

Figure 4.3 Ordering of inherited rules

Consider the more concrete example shown in *Listings 4.6-4.8*. *Listing 4.6* shows a component that extends the component defined in *Listing 4.7*. The original component had rules for some keywords and a number token, but the new component adds some new keywords and an identifier. *Listing 4.8* shows the result. Note the order of the keywords in *Listing 4.8* – two keywords precede the originals, but one follows. In this way we control the precedence of new rules. If they precede the old rules then they have higher precedence; otherwise they have lower precedence. Note also that the identifier rule follows all of the keyword rules, even the inherited ones.

```
%component inheriting_comp
%%
new_keyword1 {: action5 :}
new_keyword2 {: action6 :}
%:
{identifier} {: action7 :}
%:
<<ANY>> {: action8 :}
<<EOF>> {: action9 :}

%%inherit inherited_comp

new_keyword3 {: action10 :}
```

Listing 4.6 Rule Order Example – Inheriting Component

```
%component inherited_comp
%%
keyword1 {: action1 :}
keyword2 {: action2 :}
keyword3 {: action3 :}
%:
{number} {: action4 :}
```

Listing 4.7 Rule Order Example – Inherited Component

4.5 Conflicts

Since MetaLexer allows modules to inherit from multiple parents, there are frequently conflicts. For example, macros, exceptions, and lexical rules can be declared in two parents of a single component. Similarly, options, declarations, and embeddings can be declared in two parents of a single layout. When this occurs, the instance from the first module to be inherited dominates.

To increase consistency, both internally and with JFlex's precedent, MetaLexer extends this idea to other potential conflicts. For example, an option may be declared in both a layout and its parent or even twice within the same layout. Rather than calling this an error, MetaLexer resolves the conflict in favour of the first textual occurrence.

For example, suppose that a layout declared the option *%option name "%name foo"* and its

```
1  %component inheriting_comp
2  %%
3  new_keyword1 {: action5 :}
4  new_keyword2 {: action6 :}
5  keyword1 {: action1 :}
6  keyword2 {: action2 :}
7  keyword3 {: action3 :}
8  new_keyword3 {: action10 :}
9  %:
10 {identifier} {: action7 :}
11 {number} {: action4 :}
12 %:
13 <<ANY>> {: action8 :}
14 <<EOF>> {: action9 :}
```

Listing 4.8 Rule Order Example – Merged Component

parent declared the same option with a different value *%option name "%name bar"*. Then, since the parent option is added to the *end* of the options section, the child option dominates. That is, the final value of the option *name* would be *"%name foo"*. A warning would be issued, flagging the duplication, but there would be no error. If the parent declaration actually appeared in the same file, the behaviour would be the same.

Conflicts amongst embeddings [lexical rules] are handled in the same way, after taking into account the more complicated textual order described in *Section 4.4.1* [*Section 4.4.2*].

4.6 Error Checking

Problems with MetaLexer specifications are broken into two categories. Errors are problems that invalidate a specification, preventing further action. Warnings, on the other hand, are problems (or potential problems) that will not result in incorrect behaviour but are probably worthy of the developer's attention.

The following problems are errors:

1. missing declarations (lexical states, macros, and meta-tokens),
2. missing modules (layouts and components),

3. circular dependencies (macros, components, and layouts),
4. misclassified lexical rules (see *Section 4.4.2*),
5. lexical states that are both inclusive and exclusive,
6. unsatisfied **`%extern`**'s,
7. empty character ranges in regular expressions,
8. components without lexical rules,
9. layouts without component references, and
10. layouts without start components.

The following problems are warnings:

1. deletions with no effect (lexical rules, options, and embeddings),
2. replacements with no effect,
3. clobbered definitions (lexical states, macros, options, and embeddings), and
4. unused definitions (lexical states, macros, and component imports).

4.6.1 Finalization

To improve the quality of error messages, modules are finalized before being inherited. A component is finalized once all parents are finalized, all parents have been incorporated, and all error checks have been performed. Similarly, a layout is finalized once all parents are finalized, all parents have been incorporated, all referenced components are finalized, and all error checks have been performed. This means that problems with a module will be reported in that module, rather than in an inheriting module.

Once the top-level layout is finalized, the result is a single flat (i.e. inheritance-free) layout referring to one or more flat components.

4.6.2 Helper Modules

Sometimes, a module exists only as a repository for shared code. For example, one might create a single component that contains all of the macros for our lexer and then inherit that component in every other component. However, the lexer should never transition to the macro component. In fact, since it contains no rules, it is invalid. The macro component is an example of a helper module – a module that can only be used through inheritance.

If a module is flagged **`%helper`**, then some of its error (and warning) checks will be omitted. As soon as it is inherited into a non-helper module, however, the checks will be applied and any unresolved errors will be caught.

Deferred checks include: missing state declaration, missing macro declaration, missing **`%append`** block, missing component import, missing start component, missing **`%declare`**. Note that these are all deficiencies that could be remedied in a child.

The **`%helper`** directive is not inherited. That is, the child of a helper module will not be a helper module unless it also includes a **`%helper`** directive.

4.7 Scoping

Scopes are important for preventing name collisions. In MetaLexer, each component is a separate scope. AIL code regions, macro declarations, lexical states, and lexical rules are not visible outside the component – neither to non-descendent components nor to layouts. Meta-token declarations, which are implicit, are visible to layouts but not to other components.

Each layout is also a separate scope. AIL code regions, options, declarations, and embeddings are not visible to other layouts. They are, however, visible within referenced components, which are considered to be nested scopes.

During inheritance, the contents of the parent are incorporated into the child's scope. Hence, all parts of a module are visible to its children.

4.8 Qualified Names

On occasion, it may be desirable to divide a large set of MetaLexer files into folders and subfolders. This might be because two similar modules (i.e. components/layouts) have the same name or because there are simply too many files. Rearranging the files is simple enough, but we also need to ensure that MetaLexer can find them. The simplest option would be to provide MetaLexer with a list of the directories we are using. However, this will not suffice if we want to use two modules with the same name but different paths. For this reason, MetaLexer allows qualification of module names. That is, a module name may incorporate path information that tells MetaLexer where to look for it.

For example, suppose that while parsing a specification, MetaLexer encountered the component name *dir1.dir2.comp*. Then, for each directory on its path, it would look in the subdirectory *dir1/dir2/* for the file *comp.mlc*. Upon finding the module in question, it would verify that it was named *dir1.dir2.comp*.

Organizing MetaLexer files into folders does not introduce any scopes. Modules within the same directory relate to each other in exactly the same way as modules within different directories and names must always be fully qualified.

4.9 Append Components

Append components are used to construct tokens containing blobs of validated text. For example, consider string literals. The resulting token will contain the text as it originally appeared in the input stream (i.e. without modification), but some validation is required before the token can be accepted. One might wish to weed out invalid escape sequences, for instance. In an append component, successfully matched rules append to a shared buffer rather than returning individually. Then, when the component is complete (i.e. when the end meta-pattern of the embedding of which it is the guest is observed), the buffer is wrapped in a token and returned.

A component is flagged as an append component by including an **%append** or

`%appendWithStartDelim` region. The AIL contents of the region will be treated as a method to be called when the component is complete. This method has parameters *int startLine*, *int startCol*, *int endLine*, *int endCol*, *String text* indicating the position and contents of the buffer built up by the component. They should be used to construct and return a token object.

The presence of an **`%append`** or **`%appendWithStartDelim`** region also makes available an *append(String)* method, which individual actions can use to append the text they match to the buffer. It is possible to append a string other than the matched input, but the position arguments passed to the **`%append`** or **`%appendWithStartDelim`** region will only reflect the size and location of the original text.

Listing 4.9 shows how an append region can be used to build up a string literal, validating and evaluating escape sequences along the way. Whenever an escape sequence is encountered, it is evaluated and appended to the buffer. Everything else is appended directly. Then, when the closing quote is encountered, the append action method is called and a new string literal token is returned, containing the validated and expanded text. Note that the position of the token will reflect the size and location of the original text, rather than the expanded text.

A component that inherits from an append component is also an append component. By default, it will inherit its parent's **`%append`** or **`%appendWithStartDelim`** region but this can be overridden with a local version.

Sometimes, append components interact with each other. This interaction is captured in *Table 4.1*.

	To Normal	To Append
From Normal	N/A	New Buffer
From Append	Call Append Region	Retain Buffer

Table 4.1 Interaction of append components

Note that the append buffer is preserved when transitioning between append components. That is, consecutive append components append to the same buffer, building up a single string.

```
1  %component string
2  %append{
3     return new StringLiteral(text, startLine, startColumn, endLine,
          endColumn);
4  %append}
5
6  HexDigit = [0-9A-Fa-f]
7
8  %%
9
10 "\"" {: :} END_STRING
11
12 (\\u{HexDigit}{1, 4}) | (\\x{HexDigit}{2}) {:
13    String hexString = yytext().substring(2);
14    int hexNum = Integer.parseInt(hexString, 16);
15    append((char) hexNum);
16 :}
17
18 \\[0-3]?[0-7]?[0-7] {:
19    String octalString = yytext().substring(1);
20    int octalNum = Integer.parseInt(octalString, 8);
21    append((char) octalNum);
22 :}
23
24 \\ (n|\n) {: append("\n"); :}
25 \\ (r|\r) {: append("\r"); :}
26 \\t {: append("\t"); :}
27 \\f {: append("\f"); :}
28 \\b {: append("\b"); :}
29
30 \\ (. | \n) {: append(yytext().substring(1)); :}
31
32 %:
33 %:
34
35 <<ANY>> {: append(yytext()); :}
36 <<EOF>> {: error("Unterminated string literal"); :}
```

Listing 4.9 Append Component Example – String Literal

There is one exception to these general rules. As mentioned in *Section 4.3*, on an end-transition, a special region meta-pattern is triggered. For example, after we complete a *string_literal* component, we see a *%string_literal%* region. In rare cases, this region might actually be the end meta-pattern of the next embedding on the stack. If it is, then another

embedding will end and another region will be generated. Since this effect can cascade, we thought it prudent to suppress all but the first append action. After all, only the first append action can receive a non-empty string since the buffer is cleared as soon as it is called.

4.9.1 Start Delimiters

Sometimes, it may be desirable to retain the delimiters of an append region. For example, if we were building up a multiline comment in Java, we might want to retain the '/*' and '*/'. Unfortunately, the start delimiter comes before the transition to the append component.

MetaLexer solves this problem with a two part construct. First, the component that sees the start delimiter calls *appendToStartDelim(String)* (which is available to all components, whether or not they are append components) to store it in a special delimiter buffer. Then, the component that wants the start delimiter (i.e. the append component) can use an **`%appendWithStartDelim`** region (rather than an **`%append`** region) to request that the contents of the delimiter buffer be prepended to the append buffer (and positions) that it has built up.

For example, *Listings 4.10 & 4.11* show how we might build up a multiline comment in Java, retaining the delimiters. In *Listing 4.10* we call *appendToStartDelim(String)* when we encounter the start delimiter and in *Listing 4.11* we use an **`%appendWithStartDelim`** region rather than the usual **`%append`** region. We also call *append(String)* when we see the end delimiter.

```
1 %component java
2 %%
3 "/*" {: appendToStartDelim(yytext()); :} START_COMMENT
```

Listing 4.10 Start Delimiter Example – Java

```
1 %component comment
2 %appendWithStartDelim{
3    return new Comment(text, startLine, startColumn, endLine, endColumn);
4 %appendWithStartDelim}
5 %%
6 "*/" {: append(yytext()); :} END_COMMENT
7 %:
8 %:
9 <<ANY>> {: append(yytext()); :}
```

Listing 4.11 Start Delimiter Example – Java Comment

4.10 Conditional Meta-Tokens

Though MetaLexer is intended to abstract all of the state transition logic out of the action code, it is sometimes necessary to violate this abstraction. For example, the transition logic may depend on an externally set runtime property.

Listing 4.12 shows an example of how to conditionally send a meta-token to the meta-lexer (presumably triggering a transition). A character is pushed back into the input stream and an action code transition (unsafe, but occasionally necessary) determines which rule will reconsume the character. After the character is reconsumed and the meta-token is or is not sent, an action code transition restores the original state. The unsafe logic is entirely encapsulated within the module – no other parts of the lexer need to be aware of the ugliness.

This pattern makes a lexer harder to read and understand so it should be used sparingly.

4.10.1 Indentation-Based Languages

Some languages use indentation rather than symbols to indicate scoping levels. Python[4], for example, creates a new scope whenever a line of code is indented more than the previous line and closes a scope whenever a line of code is indented less than the previous line. In *Listing 4.13*, lines 3, 4, and 7 are all in new scopes. Furthermore, line 9 is actually in the

[4]http://www.python.org/

```
%component conditional
%xstate COND_TRUE, COND_FALSE
%%
rule {:
   yypushback(1);
   if(external condition) {
      yybegin(COND_TRUE);
   } else {
      yybegin(COND_FALSE);
   }
:} //NB: no meta-token

%:
%:

<COND_TRUE> {
   <<ANY>> {: yybegin(YYINITIAL); :} META_TOKEN //NB: meta-token
}

<COND_FALSE> {
   <<ANY>> {: yybegin(YYINITIAL); :} //NB: no meta-token
}
```

Listing 4.12 Conditional Meta-Token Pattern Example

outermost scope, since it is dedented twice from the preceding line.

```
i = 99
while 1:
   if i == 1:
      print "1 bottle of beer on the wall."
      break
   else:
      print "%s bottles of beer on the wall." % i
      i = i - 1
print "No more beer on the wall."
```

Listing 4.13 Python Indentation Example

While this scheme does revolve around a stack of indentation levels, it is difficult or impossible to simulate this behaviour with MetaLexer's stack of embeddings. Instead, the conditional meta-tokens described above can be used to determine when an indentation should trigger one or more transitions.

Listing 4.14 outlines a solution in MetaLexer pseudocode (with a Java-like AIL). The indentation level is tracked using a stack, just as in the default implementation of Python. *START_SCOPE* and *END_SCOPE* meta-tokens are generated at the beginning and end of each scope so that they can be used in embedding start and end patterns. *INDENT* and *DEDENT* tokens are also returned for the benefit of the parser. All of this custom logic can be encapsulated in a helper component and hidden from the rest of the specification.

```
%{
   //indentation levels of scopes enclosing current scope
   Stack<Integer> indentationLevels = new Stack<Integer>();
   //number of dedents indicate by single decrease in indentation
   int numDedents = 0;
%}

{Whitespace} {:
   int existing = indentationLevels.peek();
   int current = yylength();
   if(current > existing) {
      //indentation up - push level and start scope
      indentationLevels.push(current);
      yybegin(INDENT); //helper state handles token, meta-token
      yypushback(1); //pushback so that helper state can re-consume
   } else {
      //indentation down or same
      //pop, looking for current indentation level
      while(current < existing) {
         numDedents++;
         indentationLevels.pop();
         existing = indentationLevels.peek();
      }
      if(current != existing) {
         //didn't find current level in stack - error
         error("Invalid dedent - didn't match any previous level");
      } else if(numDedents > 0) {
         //found current level in stack - valid
         yybegin(DEDENT); //helper state handles tokens, meta-tokens
         yypushback(1); //pushback so that helper state can re-consume
```

```
31          }
32       }
33    :}
34
35    <INDENT> {
36       <<ANY>> {:
37          //generate token and meta-token, then return to default state
38          yybegin(YYINITIAL);
39          return token(INDENT);
40       :} START_SCOPE
41    }
42
43    <DEDENT> {
44       <<ANY>> {:
45          //numDedents == num tokens/meta-tokens to generate
46          if(numDedents > 0) {
47             yypushback(1); //pushback for re-consumption by self
48             numDedents--;
49             return token(DEDENT);
50          } else {
51             yybegin(YYINITIAL);
52          }
53       :} END_SCOPE
54    }
```

Listing 4.14 Indentation-Based Languages in MetaLexer

Chapter 5

Tool Execution

This chapter describes the tools provided in the MetaLexer distribution: the MetaLexer-to-JFlex, MetaLexer-to-MetaLexer, and JFlex-to-MetaLexer translators.

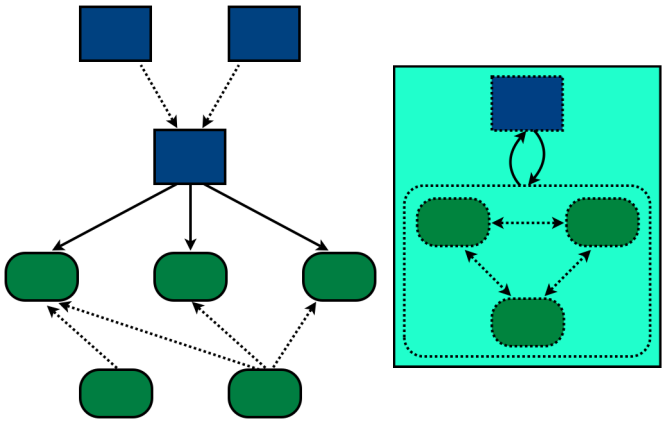
5.1 MetaLexer-to-JFlex Translator

The MetaLexer-to-JFlex translator is the most important of the provided tools because it creates executable lexers. As shown in *Figure 5.1*, it reads MetaLexer specifications (*Figure 5.1a*) and produces JFlex specifications (*Figure 5.1b*). The resulting JFlex specifications can be compiled to Java classes without any external dependencies (e.g. runtime libraries).

The translator is most easily executed via metalexer-jflex.jar. The program accepts three arguments: the name of a layout (without file extension), the directory in which to look for the layout, and the directory in which to write the new JFlex files.

For example, one might run *java -jar metalexer-jflex.jar natlab /home/userA/src /tmp*.

Otherwise, you can execute the main class, *metalexer.jflex.ML2JFlex*, directly.



(a) Input Files – MetaLexer **(b)** Output Files – Structured JFlex

Figure 5.1 MetaLexer-to-JFlex Translator

5.1.1 Tracing

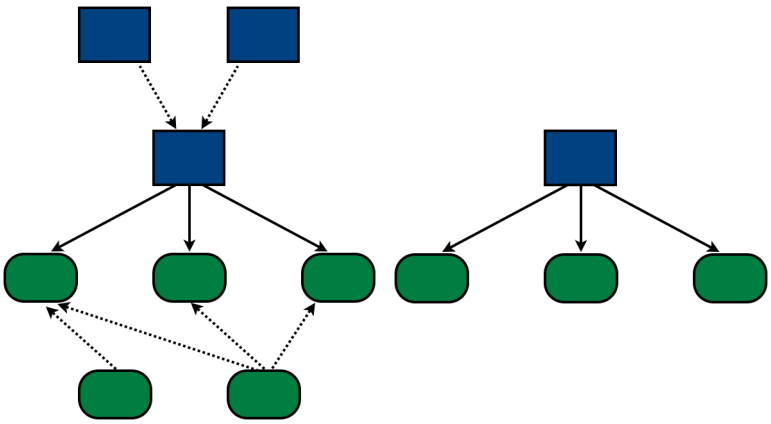
If *-t* is passed before the first argument to the jar (or to ML2JFlex), then the generated lexer will support tracing. To see a high-level meta-lexing trace, call *setTracingEnabled(true)* on the generated lexer before using it.

Furthermore, when the *-t* is passed to MetaLexer, the generated lexer will provide functions *getCurrentComponent()* and *getCurrentEmbedding()*. These functions return the names of the current component and embedding, respectively. They should never be used to affect the control flow of the lexer but they are useful for debugging.

5.2 MetaLexer-to-MetaLexer Translator

The MetaLexer-to-MetaLexer translator reads in a MetaLexer specification, performs syntactic and semantic checks, processes inheritance directives, and then prints out a flattened (i.e. inheritance-free) MetaLexer specification (or a list of errors). *Figure 5.2* shows an example of this transformation. In *Figure 5.2a*, the original specification consists of multiple layouts (blue) and components (green). Dotted arrows indicate inheritance (from child to parent) and solid arrows indicate composition (from layout to component). As a result of

inheritance, the children are merged into their parents and only the composition relationships remain. *Figure 5.2b* shows the result – a single flat layout referring to a number of flat components.



(a) Input Files – MetaLexer **(b)** Output Files – Flat MetaLexer

Figure 5.2 MetaLexer-to-MetaLexer Translator

The MetaLexer-to-MetaLexer translator exists primarily as a proof-of-concept – it demonstrates that the MetaLexer architecture can support multiple code generation engines. The translator is, however, useful in its own right. During the development process, it can be used to check a specification (syntactically and semantically) without considering AIL fragments. Furthermore, it shows the effects of inheritance without requiring the developer to read through lower-level LSL code.

The process for running the translator is exactly as above, except with metalexer-metalexer.jar and metalexer.metalexer.ML2ML in place of metalexer-jflex.jar and metalexer.jflex.ML2JFlex, respectively. That is . . .

The translator is most easily executed via metalexer-metalexer.jar. The program accepts three arguments: the name of a layout (without file extension), the directories in which to look for the layout (semicolon-separated list), and the directory in which to write the new MetaLexer files.

For example, one might run *java -jar metalexer-metalexer.jar natlab /home/userA/src /tmp*.

Otherwise, you can execute the main class, *metalexer.jflex.ML2ML*, directly.

5.3 JFlex-to-MetaLexer Translator

The JFlex-to-MetaLexer translator also serves as a proof-of-concept. By converting (nearly) any valid JFlex specification into a valid MetaLexer specification, the translator shows that MetaLexer is no less powerful than JFlex (as a LSL). We can be sure that it will accept any valid JFlex specification because the translator is actually a modification of JFlex 1.4.1[1].

Unfortunately, the JFlex-to-MetaLexer translator is not very useful as a tool for porting existing lexical specifications from JFlex to MetaLexer. The generated files are not written in proper MetaLexer style; rather, they are the simplest possible MetaLexer that will exhibit the same behaviour as the original JFlex files. Furthermore, the translator is subject to the limitations described in *Section 5.3.3*. A more useful tool is described in *Section 11.6*.

Figure 5.3 shows the transformation performed by the JFlex-to-MetaLexer translator. *Figure 5.3a* shows the original specification, a black box of JFlex code, and *Figure 5.3b* shows the output specification, naive MetaLexer. The output specification will always consist of a single inheritance-free layout referring to a single inheritance-free component.

(a) Input Files – JFlex **(b)** Output Files – Naive MetaLexer

Figure 5.3 JFlex-to-MetaLexer Translator

5.3.1 Functionality

Listings 5.1-5.3 illustrate the translation process. Starting with the original specification, *Listing 5.1*, the translator produces two MetaLexer files: a layout, *Listing 5.2*, and a component, *Listing 5.3*.

[1] http://jflex.de/

```
1  package pkg;
2  %%
3  %class Class
4  %{ /* action code */ %}
5  %init{ /* init code */ %}
6  %xstate state1, state2
7  identifier = [a-zA-Z]+
8  %%
9  <state1> {
10    keyword1 { kw1_action(); }
11    {identifier} { id_action(); }
12    (.|\n) { catchall_action(); }
13 }
14 <state2> {
15    keyword2 { kw2_action(); }
16    (.|\n) { catchall_action(); }
17 }
18 <<EOF>> { eof_action(); }
```

Listing 5.1 JFlex-to-MetaLexer Example – Original JFlex

```
1  package pkg;
2  %%
3  %%
4  %option class "%class Class"
5  %{ /* action code */ %}
6  %init{ /* init code */ %}
7  %xstate state1, state2
```

Listing 5.2 JFlex-to-MetaLexer Example – Generated Layout

The layout is constructed from the first part of the JFlex specification. First, the translator copies the JFlex header into the MetaLexer local header and leaves the MetaLexer inherited header blank. Then it adds the options: *%name value* becomes *%option name "%name value"*. Finally, the translator copies code regions and state declarations directly from JFlex. The generated layout has no embeddings section because all transition logic is contained in AIL code.

The component is constructed from the remainder of the JFlex specification. The translator copies macro declarations directly from JFlex to MetaLexer. Then it copies the rules, but not before they are modified and reordered. The AIL code delimiters have to be changed

```
identifier = [a-zA-Z]+
%%
<state1> {
   keyword1 {: kw1_action(); :}
}
<state2> {
   keyword2 {: kw2_action(); :}
}
%:
<state1> {
   {identifier} {: id_action(); :}
}
%:
<state1> {
   <<ANY>> {: catchall_action(); :}
}
<state2> {
   <<ANY>> {: catchall_action(); :}
}
<state1, state2> {
   <<EOF>> {: eof_action(); :}
}
```

Listing 5.3 JFlex-to-MetaLexer Example – Generated Component

– from '{ }' to '{: :}' – and the rules have to be divided into acyclic, cyclic, and cleanup categories. The translator will also attempt to find catchall patterns (e.g. *(.|\n)*) and convert them into **<<ANY>>** rules. Finally, the translator explicitly associates loose **<<EOF>>** rules with all declared states.

Adding explicit state lists to **<<EOF>>** rules is necessary because JFlex and MetaLexer handle implicit states on **<<EOF>>** rules slightly differently. Whereas MetaLexer treats **<<EOF>>** rules the same as other rules (i.e. no explicit states implies all inclusive states), JFlex treats them differently. In JFlex, **<<EOF>>** rules with no explicit states are considered to belong to *all* states. The translator makes this assumption explicit so that the resulting MetaLexer specification has the same meaning as the original JFlex specification.

5.3.2 Execution

The translator accepts the same arguments as the original JFlex executable: a list of JFlex files, *-d outdir*, etc. JFlex options related to the type of finite automaton to be generated will be ignored. If no arguments are specified, then a graphical interface will be displayed.

If you have metalexer-jflex.jar, you can execute the translator directly. For example, one might run *java jflex.metalexer.JFlex2ML lang.flex -d /gen*.

Otherwise, you will have to execute the main class, *jflex.metalexer.JFlex2ML*.

5.3.3 Limitations

Though the translator will generate a correct lexer most of the time, there are circumstances under which it does not perform as one might desire.

First, the translator discards all comments at the specification level (as opposed to within actions). This is part of JFlex's behaviour and the translator is a modified version of JFlex.

Second, the translator does not support JFlex's (infrequently used) **`%eof`** directive. This is because MetaLexer lacks an comparable directive and simulating the behaviour would require too much analysis to avoid name conflicts with the input specification.

Third, the MetaLexer-to-JFlex and JFlex-to-MetaLexer tools, alternately applied, will never achieve a steady state. The JFlex-to-MetaLexer translator does not look for code generated by the MetaLexer-to-JFlex translator so it is converted into AIL blocks in the resulting MetaLexer specification. Consequently, when the MetaLexer-to-JFlex translator is re-run, it will generate the same code. In the present implementation, this results in name conflicts (see *Section 7.2.3* for details). However, even if the name conflicts were resolved, the MetaLexer-to-JFlex translator would still re-add the same constructs every iteration, preventing a steady state.

Finally, if any of the action code in the JFlex specification refers explicitly to a lexical state, then the generated MetaLexer specification will be incorrect. This is because, in the generated MetaLexer specification, the lexical state will be declared at the component level

whereas the action code will be inserted at the layout level (i.e. scoping issue).

Chapter 6

Language Design

The previous chapters have described how MetaLexer works. This chapter explains why it works the way it does. By considering the most fundamental and contentious MetaLexer design decisions, we illustrate its underlying philosophy. Each section explores a single design decision and its consequences.

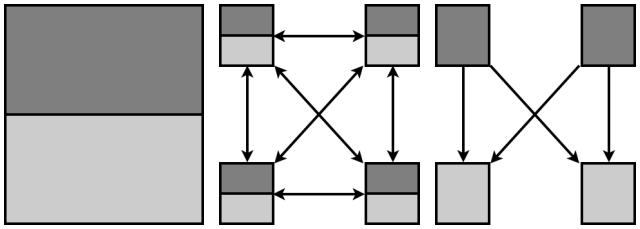
6.1 Language Division

A specification that must be contained in a single file is not very modular, so some sort of division is necessary. Now, in a lexer, we have two types of information to specify: lexical states and the interactions (i.e. transitions) between those states. As a result, there are essentially two ways in which we can divide the specification. Either we can mix the two types of information or we can keep them separate.[1] In designing MetaLexer, we decided to keep them separate: components describe lexical states and layouts describe the transitions between them.[2]

[1] Those familiar with aspect-oriented languages may recognize these setups as symmetric and asymmetric, respectively.

[2] MetaLexer further subdivides files using inheritance. This takes place after the more fundamental separation (i.e. of layouts from components) discussed here.

Figure 6.1 illustrates the two types of division. In *Figure 6.1a*, we see a single monolithic specification containing two types of information. *Figure 6.1b* shows a symmetric division of this specification. Each new file contains information of both types and any file can refer to any other file. This would be like having several little lexical specifications, each containing both lexical states and transitions. On the other hand, *Figure 6.1b* shows an asymmetric division of the specification. Each new file contains only a single type of information and references are unidirectional. This is like putting transitions in layouts and lexical states in components and then creating references from layouts to components.



(a) Monolithic **(b)** Symmetric **(c)** Asymmetric

Figure 6.1 Dividing a monolithic specification into smaller files

The separation of components from layouts makes specifications clearer and easier to read since all of the transition logic is in one place. Furthermore, it makes both layouts and components more reusable. After all, it is frequently the case that two languages lex certain constructs the same way. For example, many languages have nearly identical rules for lexing string literals. In MetaLexer, these rules are contained in a single component. Now, if this component contained the rules for transitioning into or out of the string literal lexical state (i.e. itself), then it would necessarily be coupled to another component. Unfortunately, this other component would almost certainly be language-specific and so reusability would be greatly diminished.

6.2 Types of Extension

Extensibility was a primary design goal of MetaLexer, so we had to decide what types of extension to allow. The most general possible system would allow addition, removal, or

replacement of any element of a specification. We decided to stop just short of this level of extensibility because we wanted to prevent some potentially dangerous operations. As a result, MetaLexer supports addition and replacement of all elements of a specification but deletion of only rules, options, and embeddings.

Within a component, the most obvious candidates for modification are lexical rules. When inheriting a component, these can be deleted, added, or overridden (i.e. replaced). Deletion is accomplished using the **`%delete`** directive and addition and overriding are accomplished by inserting new rules before the inherited component. Header items can be added or replaced but not deleted. Code regions, exceptions, and macros are so integral to a component that removing them would be quite unsafe.

When inheriting a layout, embeddings can be deleted, added, or overridden (i.e. replaced). Deletion is accomplished using the **`%unembed`** directive. Addition and overriding are accomplished by inserting new embeddings before the inherited layout. Options are also fully modifiable since they are likely to change from one lexer to the next, even if the languages are similar. They can be deleted using the **`%unoption`** directive and added or overridden by inserting new options before the inherited layout. Code regions, exceptions, and declarations are not removable because they are so integral to a layout. Similarly, the inherited header can be supplemented but not reduced.

6.3 Component Replacement

Replacement of component references in layouts carries certain risks – as does any sort of global replacement – but we decided to allow it because it substantially reduces development time and specification fragility. We did, however, apply certain restrictions to mitigate some of the risks.

We could have omitted this feature since the same effect can be achieved using the normal inheritance mechanisms. That is, existing embeddings (and other elements that refer to components) can be explicitly deleted and re-added referring to the new component. However, this approach is both labour-intensive and fragile since the inheriting and inherited

layouts must be kept synchronized manually.

We restricted replacement of component references in two ways. First, we decided that all replacements would be perfomed in a single pass, so that developers would not have to worry about cumulative effects. Second, we decided not to delve into components and replace the component references therein.

To make the replacement process intuitive, we decided to combine all replacements (for a single inherited layout) into a single translation map and make a single pass through the affected layout. Whenever a component name is encountered, the translation map is consulted and a replacement is made, if necessary. Since there is only one pass, there is no chance of transforming a single component name more than once. For example, if the replacement list consisted of *%replace A, B* and *%replace B, C,* the translation map would look like $\{A \mapsto B, B \mapsto C\}$. Note that occurrences of the component name *A* would be replaced by *B*, rather than by *C* since the translation map is only applied once.

To limit unintended consequences of replacements – especially to the inheritance hierarchy of components – we decided not to have replacement affect components. That is, component references in components referred to by the layout are not modified by a replacement. Hence, there is no need to worry that the component inheritance hierarchy will be modified by a replacement. For example, if a layout uses a component *C* that inherits a component *D*, and *D* is replaced with *E*, then component *C* will be unaffected even though it refers to a component that has been replaced.

6.4 Inheritance

In designing MetaLexer's inheritance mechanism, we considered two fundamentally different approaches. On the one hand, there was object-oriented (OO) inheritance, in which children delegate to parents. On the other, there was textual inclusion, in which parents are copied directly into children. We chose to use an extended form of textual inclusion, primarily because we found it to be substantially more intuitive. The reasons are threefold.

First, MetaLexer inheritance can always be mimicked by manually merging a component

or layout and its ancestors into a single file. This is possible because descendents of a common ancestor share nothing in common – each has its own copy of whichever parts of the ancestor remain.

Second, this approach is more consistent with existing LSLs. Since developers are used to working with a single specification file (and since ultimately, the implementation will generally output a single specification file), having a clear way to process inheritance and visualize the result is very helpful.

Third, as discussed in *Section 6.2*, we wanted inheriting modules to be able to delete elements of their ancestors. Speaking loosely, this means that MetaLexer modules are not subtypes of their ancestors. While this does not technically violate the definition of inheritance, popular OO languages so often conflate inheritance and subtyping that this discrepancy could cause confusion. That is, we worried that being partially, but not totally consistent with familiar OO systems would be counter-intuitive.

Unfortunately, this decision is not without consequence. Under this scheme, a naive implementation will likely produce output containing a substantial amount of duplicated code. For example, if all of the macros are extracted into their own helper component and then inherited by all components that use macros, then each component will end up with copies of all macros. A more advanced implementation would recognize and eliminate identical sections of inherited code, especially those that are inherited from the same source.

6.5 Finalization

We decided early on that we wanted MetaLexer error messages to have very specific positions in the source code. This had important consequences for the inheritance mechanism. In particular, it meant that each module had to be treated as a self-contained unit, capable of being error checked. Hence, each module is finalized – made self-sufficient – before it is inherited.

Finalizing each module before inheritance makes it possible to check for errors at each step in the inheritance process, rather than waiting until the end when everything has been

flattened. For example, suppose a component refers to a macro that it has not declared. An error should be reported and it should refer to the specific component and rule. However, if error checking is delayed until after all inheritance has been performed, then the macro might be defined in a component that inherits the invalid component and the error might remain hidden. This is why each module (i.e. component or layout) is evaluated to an independent unit and error checked before it is integrated into an inheriting module.

As an added benefit, this type of inheritance is very intuitive, because the intermediate files can actually be constructed and examined independently. There is no need to visualize sharing of data in memory.

The alternative would have been to perform error-checking after processing all inheritance. This would have resulted in confusing situations where gaps in specifications (i.e. errors) were inadvertently filled by inheriting modules. While this sort of behaviour is sometimes useful, even necessary, we decided that we would prefer to make it explicit. To this end, modules can be flagged as **`%helper`** and some checks will be deferred (see *Section 4.6.2* for details).

6.6 Order and Duplication

The chief problem when combining multiple modules into a single specification, especially using multiple inheritance, is how to resolve conflicts. That is, if two different modules provide the identical (or overlapping) elements, then somehow one must be chosen. One option would be to raise an error for each conflict, but this approach tends to be too restrictive. Better options are to provide a general rule for resolving conflicts or to allow the developer to resolve conflicts explicitly.

Fortunately, when designing MetaLexer, we had some precedent to rely on. In JFlex (and other LSLs), if two rules match the same input string, then the first rule is chosen. For example, if a lexer containing rules *a(aa)** and *a*b?* (in that order) was executed on input *aaa*, then both rules would match but *a(aa)** would be chosen because it appears first textually. We decided to take the same approach. Within a MetaLexer specification, order

matters. If two rules (or directives) conflict/overlap, then the first is chosen.

Seeing how well this worked, we decided to extend this philosophy to the rest of MetaLexer: if two options have the same name then the second will be ignored; if there are two start states or start components, then the second will be ignored; if there are two **%append** regions, then the second will be ignored; etc. This eliminates a lot of errors and makes it easier to combine modules that were developed separately. Furthermore, we decided to apply this policy to individual files to reinforce the idea that inheritance can always be simulated by manually merging files.

In general, duplicating part of a MetaLexer specification will not result in an error. If something is obviously redundant, then MetaLexer may issue a warning but it will have no effect on the behaviour of the generated lexer.

To some extent, MetaLexer also allows the developer to manually resolve conflicts. For example, when inheriting a module, they can choose to delete an element that would have caused a conflict. This is primarily useful for eliminating warnings.

6.7 Rule Organization

Perhaps our most controversial design decision was to divide component rules into three categories: acyclic, cyclic, and cleanup. We chose to do so because insertion points are required for new rules and the boundaries between these categories are both natural and (in practice) sufficient. Furthermore, the restrictions this division imposes are not as severe as they initially appear.

We chose this particular division based on our observations concerning frequently used regular expressions. The three categories correspond neatly to the most commonly used types of regular expressions: acyclic regular expressions are used to represent keywords and symbols; cyclic regular expressions are used to represent identifiers and numeric literals; and cleanup regular expressions generally perform error handling and other administration. Furthermore, the order in which these categories are arranged is natural – keywords usually precede identifiers, which usually precede cleanup rules.

The boundaries between these categories are (almost) totally sufficient as insertion points for new rules. That is, given a new rule and an arbitrary insertion point into an existing list of rules, the same effect can (nearly) always be achieved by inserting the new rule at one of the boundaries. As *Figure 6.2* shows, new keywords and symbols should be inserted before the existing acyclic section; new identifiers and numeric literals should be inserted after the existing acyclic section but before the existing cyclic section; and new cleanup code should be inserted after the existing acyclic and cyclic sections but before the existing cleanup section.

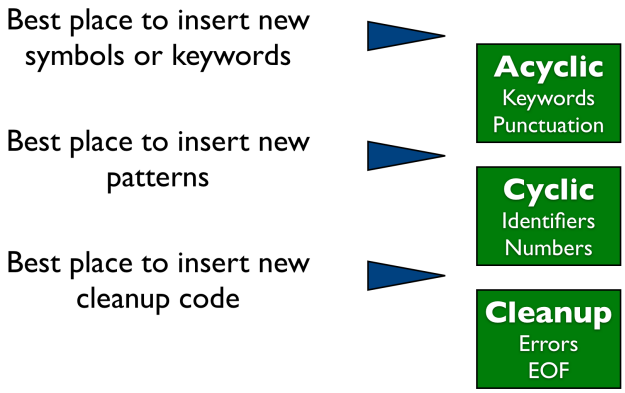


Figure 6.2 Rule type boundaries as insertion points

Exceptions do exist. For example, suppose that an existing specification contained the rules *(aa)*+ and *(aaaaa)*+. Then, since 2, 3, and 5 are coprime, inserting *(aaa)*+ between them would change the behaviour of the lexer. For example, before the insertion, *(a)*$\{15\}$ would have matched the *(aaaaa)*+, whereas afterwards it would match *(aaa)*+. This problem can arise in any specification where two rules overlap, but neither subsumes the other. However, this rarely occurs in practice so the following workaround suffices: simply delete the rules in the inheriting component and reinsert them in the correct order.

As for the restrictions that this system seems to impose, observe that, given a list of rules such that no rule is (partially) subsumed by a preceding rule, the list can be rearranged into this order without changing its behaviour.

6.8 Append Components

Another early consideration in the design of MetaLexer was the goal of eliminating boilerplate code for input validation lexical states. These states gather up the input as they validate it and return a single token at the end. Each one requires a lexical state, a string buffer, and possibly position variables plus code to coordinate them. To eliminate this boilerplate code, we decided to supplement MetaLexer with features making this type of lexical state easy to specify.

Before adding the new feature, we had to decide whether or not it was worth the extra complication. After all, it is frequently possible to eliminate these validation lexical states in favour of regular expressions. For example, a regular expression can be used to validate string literals (e.g. all escapes are valid, no newlines, etc). However, regular expressions do not produce very good error messages. Rather than indicating which part of a string fails to match the regular expression, they simply fail to match at all. The best one can hope for is an 'unexpected character' message indicating that no other rule has matched the beginning of the string literal. This is why lexer writers create lexical states to verify string literals character-by-character as they are appended to a buffer. Consequently, we decided that a new language construct would be worthwhile.

MetaLexer eliminates validation lexical state boilerplate by introducing append components (see *Section 4.9* for an explanation of their behaviour). In common cases, like string literals and multi-line comments, no state information is required at all – MetaLexer maintains it all behind the scenes. If some variables are required it is because they pertain to the specific component and are, therefore, not boilerplate.

Unfortunately, append components are not very good at handling start delimiters. Since start delimiters occur before the transition to an append component, they are unavailable to that component. As such we had to add another pair of constructs: the *appendToStartDelim(String)* method and appendWithStartDelim regions (see *Section 4.9.1* for an explanation of their behaviour). The *appendToStartDelim(String)* method passes information to the next component and an appendWithStartDelim region receives it.

Notice that this design does not couple the source and destination components. If the source component is used with a different, non-append destination component, then the start delimiter will be ignored. Similarly, if the destination component is used with a different source, it will use the start delimiter, if any, from that component.

We append to the start delimiter rather than setting it, because it could be build up across several rules (since a start meta-pattern can be built up across several rules).

6.9 Meta-Pattern Restrictions

When we designed the meta-patterns in MetaLexer, we had a specific implementation in mind. We expected to have a pair of lexers: one for processing the input stream and one for processing the meta-token stream (see *Section 7.2.3* for details of an example implementation). Since we expected both to be full-scale lexers, we had the option of making meta-patterns every bit as complicated as normal patterns (i.e. regular expressions). However, to keep meta-patterns intuitive and to avoid forcing future implementors to follow this implementation pattern, we decided to restrict meta-patterns more than normal patterns.

We did, however, give meta-patterns one feature that normal patterns lack. There is a single meta-pattern that matches an empty input sequence: **`<BOF>`**. Because of the risk of infinite loops and the difficulty of matching an empty string with a normal pattern, this particular pattern, called a *pure BOF* is supported separately. MetaLexer determines analytically all pure BOF transitions that will take place at the beginning of the stream and performs them as a single step. It also detects cycles ahead of time so that errors can be raised at compilation time.

We included this feature mostly for consistency – it would be strange if it could be used in combination with other meta-tokens, but not on its own. Hypothetically, it also allows developers to initialize the embedding stack. Using pure BOFs, it is possible to move a number of embeddings onto the stack before lexing starts so that they can be popped off at appropriate times. While this is unlikely to be useful in practice, the meaning is sensible and the behaviour would be very difficult to simulate without pure BOFs.

The first restriction we imposed on meta-patterns was the elimination of ranges. Since meta-tokens and regions (i.e. references to entire components) are unordered, ranges have no intuitive meaning. We could have implemented them quite easily, they just would not have had predictable behaviour.

Next, we eliminated general negation. While it is frequently useful to specify the negation of a single meta-token (e.g. anything but a newline), it is less commonly necessary to specify the negation of a meta-pattern (e.g. anything but four keywords in a row). Furthermore, it is often difficult to predict at what point such a meta-pattern will match. Consequently, we decided to limit negation to classes. That is, classes of meta-tokens and regions can be negated, but full meta-patterns cannot.

We decided not to implement meta-pattern macros since meta-patterns are not generally repeated. Furthermore, in order to be checkable, macros would have to be tied to specific components and there are relatively few meta-patterns referring to any one component. There are, however, no obstacles to adding support for meta-pattern macros in the future.

We also insisted that **<BOF>** and regions only appear at the beginning of meta-patterns. The reason for restricting the position of **<BOF>** is obvious – nothing can precede the beginning of the meta-token stream. Regions are a bit trickier – we restricted their positioning for efficiency reasons. Regions can only appear at the beginning of a meta-pattern because, if something preceded them, then a portion of the meta-token stream would have to be matched more than once. That is, since the region is present in the stream, it must be the case that the preceding meta-tokens triggered a transition to the corresponding component. Therefore, they have already been matched. Allowing them to be rematched, perhaps a large number of times, would substantially increase the worst-case runtime of the lexer.

Unfortunately, meta-patterns in which regions appear after the first position can be quite useful. For example, suppose we wanted to switch contexts upon seeing parenthesized strings; we might write a meta-pattern like *LPAREN %STRING% RPAREN*. This seems like a perfectly reasonable thing to do, but MetaLexer forbids it because the region, *%STRING%*, is not in the first position. If we allowed this meta-pattern, then we would have to re-process the *LPAREN* meta-token, possibly a large number of times.

This might restriction might seem to impose quite a deficiency, but recall the context – we are still in the lexer. This sort of meta-pattern is properly the domain of the parser. The absence of this functionality is no more significant than the absence of any other context-free construct.

6.10 Cross-Platform Support

Our decision to make MetaLexer cross-platform – independent of AIL, PSL, and (backing) LSL – substantially increased the complexity of the project. It affected all aspects of the design and is responsible for many of the syntactic differences between MetaLexer and JFlex. However, the careful re-examination of all elements of the design that cross-platform support required was ultimately beneficial.

6.10.1 Action Implementation Language

To keep MetaLexer independent of AIL, we decided to treat all occurrences of AIL code in MetaLexer specifications as free-form strings. For example, when declaring exceptions, the exception names are enclosed in quotation marks so that they can contain whitespace or other non-identifier characters, depending on the AIL.

We provided escape sequences for all closing delimiters to resolved the problem of allowing closing delimiters within these free-form strings. For example, an action may contain the character sequence ‘:}’ if it is escaped as ‘%:}’. Similarly, an **`%init`** code region may contain the character sequence ‘%init}’ if it is escaped as ‘%%init}’. As a result, free-form strings are totally unrestricted.

Inheritance of header sections would have been simpler if they had been converted into a series of directives (e.g. **`%package`**, **`%import`**, etc), but this would have tied MetaLexer to the structure of one AIL. Instead, we decided to retain the free-form header of JFlex and other LSLs, merging them using simple concatenation. Unfortunately, this meant giving up support for deletion and replacement of parts of the header.

After rule categorization (see *Section 6.7*) our most controversial decision was to change the brackets surrounding action code in component rules. Unlike JFlex (and JLex, Flex, etc), MetaLexer uses '{: :}' rather than '{ }'. We made this breaking change because MetaLexer might conceivably be used with an AIL in which a single identifier is a valid action. For example, in Ruby, a single identifier can be a function call because parentheses are optional in Ruby. In such cases, an action containing a single identifier would be indistinguishable from a macro invocation at the end of the preceding regular expression. Modifying the delimiters solves this problem.

Finally, we attempted to minimize the number of API calls guaranteed to specification writers (e.g. *append(String)*) and lexer users (e.g. *stop()*) in order to remain implementable in the largest possible number of AILs.

6.10.2 Parsing Specification Language

As originally conceived, MetaLexer was to have monitored the stream of tokens returned by the lexer and used the tokens themselves to make decisions about transitions between lexical states. This would have eliminated both the need for meta-tokens and the need for special start delimiter behaviour (see *Section 6.8*). However, it would also have tied MetaLexer not only to a specific AIL but to a specific PSL.

To keep MetaLexer cross-platform, we decided to introduce meta-tokens. They require some extra work on the part of the specification writer, but they also simplify transition logic.

First, meta-tokens eliminate the need to examine all tokens returned by the lexer. By leaving some rules unlabelled with meta-tokens, it is possible to drastically simplify most multi-meta-token meta-patterns. In particular, they no longer have to account for uninteresting tokens in the middle of desirable patterns.

Second, meta-tokens can be generated by rules that do not return tokens. This makes it possible to make transition decisions without cluttering the token stream received by the parser.

Finally, meta-tokens do not have to be distinct. Assigning to rules the same transition effect is easy with meta-tokens – just label them with the same meta-token. It does not matter if they do not return the same type of token or even any tokens at all.

We also declined to provide a specific API for generated lexers. This makes it easier for specification writers to adapt the public APIs of their lexers to suit their preferred PSLs.

6.10.3 Lexer Specification Language

Lexer specifications contain a lot of information in options/directives. However, these vary substantially from language to language. To accomodate this, we made options free-form but gave them names. This allows them to be deleted or replaced without tying them to a specific LSL.

Chapter 7

Architecture

The previous chapter described the design decisions behind MetaLexer. This chapter discusses some of the specific implementation choices that were made and the issues that were encountered while constructing MetaLexer. It is broken down into two parts: first we discuss the tools that are used to build and execute MetaLexer and then we explore the more interesting/involved parts of MetaLexer's backend implementation.

7.1 Tools Used

This section describes the tools that we use to build, execute, and test MetaLexer.

7.1.1 Ant and Eclipse

MetaLexer can be built from source using Ant[1] alone or a combination of Ant and Eclipse[2]. This should satisfy the majority of Java developers.

Presently, the two most popular ways to build a Java project from source are to use an Ant script or to create an Eclipse project and let Eclipse do the work. To accomodate a wide

[1] http://ant.apache.org/
[2] http://www.eclipse.org/

range of developers, MetaLexer supports both approaches. The source distribution includes both Eclipse project files and Ant build scripts. In both cases, the Ant build scripts are used for all non-Java compilation, but if Eclipse is used, then the *eclipse.running* property must be set to prevent the Ant build scripts from calling javac to build class files.

The two approaches – Ant and Eclipse – are basically equivalent, but Ant should be used to build jar files because otherwise Eclipse will compile and include Java classes that may not be necessary for a specific configuration.

Ant and Eclipse are not distributed with MetaLexer. They are considered to be basic programming tools and it is assumed that users of MetaLexer already have or can easily obtain them.

7.1.2 JFlex

MetaLexer uses JFlex in a number of ways, but the two projects cannot be distributed together because of their incompatible licenses. Consequently, care was taken to ensure that MetaLexer would work in JFlex's absence.

JFlex specifications play several important roles in MetaLexer. First, the initial specifications for the component and layout language lexers were written in JFlex. Even after bootstrapping MetaLexer (i.e. rewriting these specifications in MetaLexer), the lexer specifications are still translated to JFlex and compiled from there. Second, MetaLexer uses a number of small lexers for string processing. Since they are so simple, these lexers are specified in JFlex (rather than MetaLexer). Third, the reference implementation – the MetaLexer-to-JFlex translator – produces lexer specifications written in JFlex. If JFlex is absent, these generated JFlex files cannot be compiled and so a number of the backend tests cannot be run (those that test the behaviour of the generated lexers).

Unfortunately, JFlex is covered by the GNU General Public License (GPL)[3], so it cannot be released under MetaLexer's modified-BSD[4] license. As such, MetaLexer has been designed to run in a (slightly) reduced-capacity mode when JFlex is absent. This is accom-

[3] http://www.gnu.org/copyleft/gpl.html
[4] http://www.opensource.org/licenses/bsd-license.php

plished by loading all JFlex classes reflectively and failing gracefully if they are missing.

JFlex lexers (i.e. the programs output by the JFlex tool), on the other hand, are not covered by the GPL. Hence, a number of pre-compiled lexers are included with the distribution. Using a special Ant target, these lexers are generated before creating the release jar. Then, at build-time, the build script searches for JFlex on the classpath. If it is not found, the JFlex compilation steps are skipped and the pre-generated files are copied from the *permgen* directory to the directory in which they would have been generated (i.e. the *gen* directory). If a user does not have access to JFlex, then these files cannot be regenerated but MetaLexer will still be useable.

The JFlex-to-MetaLexer translator, which is distributed separately, is covered by the GPL and so does not have this problem.

7.1.3 MetaLexer

The lexers for the MetaLexer component and layout languages are specified in MetaLexer itself. They are compiled into JFlex specifications by a binary version of the MetaLexer-to-JFlex translator and are then compiled into Java classes using JFlex.

Establishing this circular dependence was straightforward. Once we had a working MetaLexer-to-JFlex translator, we created a jar file of the binaries. Then we re-specified the component and layout languages in MetaLexer and used the jar to compile them, as we would any other MetaLexer specification. To complete the process, we created a new jar containing the lexer classes generated from the MetaLexer specifications. From then on, development of the lexer was done in MetaLexer.

7.1.4 Beaver

The parsers for the MetaLexer component and layout languages are generated by the Beaver parser generator[5]. Beaver has two main advantages over its popular competitor CUP[6].

[5] http://beaver.sourceforge.net/
[6] http://www2.cs.tum.edu/projects/cup/

First, Beaver allows extended Backus-Naur form (EBNF) operators (i.e. '+', '*', '?') to be applied to parenthesized subexpressions, effectively creating anonymous non-terminals. This eliminates the need for a large number of temporary non-terminals and makes the grammar more readable. Second, Beaver is speed-oriented. It claims to provide the fastest possible dispatching of actions (within the LALR framework)[7].

Beaver is covered by a modified-BSD license and so is distributed with MetaLexer. Files generated by Beaver are not covered by any license and so they too are distributed with MetaLexer.

7.1.5 JastAdd

JastAdd [EH07b] is an extensible attribute grammar framework. It provides a particularly nice way to specify, build, and transform abstract syntax trees (ASTs). JastAdd also provides lightweight support for aspects. It allows intertype declarations into specific generated classes without forcing developers to deal with the complexity of an entirely aspect-oriented project. Furthermore, since JastAdd is the sort of extensible framework that MetaLexer should eventually be used with, its use also serves as a sort of proof of concept.

JastAdd is covered by a modified-BSD license and so is distributed with MetaLexer. Files generated by JastAdd are not covered by any license and so they too are distributed with MetaLexer.

7.1.6 JUnit

MetaLexer uses the infrastructure provided by the JUnit[8] tool to manage its test suites. Though this violates design principles of the JUnit team[9], it is quite effective[10].

JUnit has become the de-facto standard for testing Java programs. Even when the tests are

[7] "The inner workings of Beaver's parsing engine use some interesting techniques which make it really fast, probably as fast as a LARL [sic] parser can get" – http://beaver.sourceforge.net/

[8] http://www.junit.org/

[9] http://junit.sourceforge.net/doc/faq/faq.htm#tests_12

[10] For example, JUnit has actually been extended to handle functional tests: http://jfunc.sourceforge.net/

not strictly unit tests, JUnit is a useful tool for organizing and running tests because of the infrastructure it provides.

MetaLexer uses JUnit for integration and functional testing. There are test suites for the various phases of compilation: scanning, parsing, inheritance processing, error checking, and code generation. Unusually, the test suite classes are actually generated from simpler specifications.

Each test suite consists of a number of pairs of input and output files. For example, for the scanning test suite the input files contain MetaLexer fragments and the output files contain lists of tokens (type, position, and contents) and errors. These pairs are listed in another file (in order to give the programmer explicit control over which tests will be run and in what order). Finally, a JUnit 3 test suite class is generated from the list of test cases – one method for each test case. This approach is preferable to looping over a list of files for a number of reasons. First, it keeps the tests independent – a failure or exception in one will not disrupt the others. Second, it makes it possible to run a single test case. Third, it results in much better reports from the standard JUnit tools – the failure count is more accurate and failing test cases are more easily located. (More implementation details are available in *Appendix B*.)

JUnit is covered by a modified-BSD license and so is distributed with MetaLexer.

7.2 Multiple Backends

MetaLexer is divided into frontend and backend components so that it can easily be extended to support other AILs, PSLs, and (backing) LSLs.

In addition to being cross-platform (see *Section 6.10*), MetaLexer is designed to support multiple code generation targets without substantial reimplementation. To this end, it is broken into frontend and backend components. The frontend contains all of the shared functionality. The backends depend on the frontend and extend it with implementation-specific transformations and code generation.

As a proof of concept, the reference implementation contains two different backends (see *Figure 7.1*. The first is a simple pretty printer. It simply invokes the transformations of the frontend and then generates MetaLexer files from the resulting AST. The second is a more functional JFlex code generator. After invoking the frontend, it performs additional transformations and then generates a JFlex lexer that implements the MetaLexer specification.

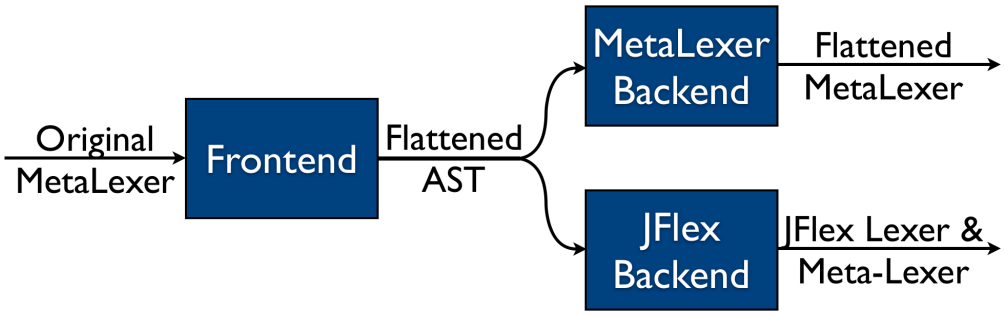


Figure 7.1 The flow of information through the MetaLexer front- and backends.

Adding a new backend implementing MetaLexer with JLex or Flex, for example, would be quite straightforward. The design would closely follow the pattern established by the JFlex backend because the lexical specification and programming languages are so similar.

Note that these backends are quite independent. They all depend on the frontend, but otherwise they do not interact. This makes it straightforward to create separate jars for each one without including a lot of extraneous code.

7.2.1 Frontend

The shared frontend accomplishes everything up to and including error checking in the traditional compiler workflow (though, of course, a backend can add additional transformations and error checking). It searches a provided list of directories (i.e. path), loads the input files, scans them, parses them, builds an AST, builds a symbol table (in attributes, since MetaLexer use JastAdd), processes all inheritance of layouts and components, and performs error checking (which generates both warnings and errors).

When the frontend is finished, the result is either a sorted list of warnings and errors (sorted

by file, position, and message) or an AST. If the result is an AST, then it consists of a single inheritance-free layout referring to a number of inheritance-free components.

Error Checking

As discussed in *Section 6.5*, each module is finalized – made complete in itself – before it is inherited so that error checking can be performed on each individual module rather than being limited to the specification as a whole.

However, to account for the fact that some modules are intentionally incomplete because they are only intended to be used through inheritance, MetaLexer includes the **`%helper`** directive. Layouts and components marked as **`%helper`** will be subject to a subset of the normal tests so that their deficiencies can be remedied in inheriting modules.

The frontend issues errors for missing declarations (macros, states, components, meta-tokens, etc), misclassified lexical rules (see *Section 4.4.2*), empty components, empty ranges in regular expressions, missing modules (i.e. not on the path), misnamed modules (i.e. with names not corresponding to filenames), unsatisfied **`%extern`**'s, circular dependencies, etc.

MetaLexer allows a lot of things that are unproductive but are not actually incorrect. To notify the programmer of potential problems with their specification, MetaLexer raises non-fatal warnings. Warnings are not severe enough to merit aborting the compilation process, but they indicate areas of concern that deserve the programmer's attention.

The frontend issues warnings for deletions that have no effect (deletions of rules, options, embeddings, etc), unused declarations (macros, states, etc), clobbering of states and macros in inheriting components, clobbering of options in inheriting components, etc.

Some of these problems cannot possibly be remedied by inheritance (e.g. circular dependencies) and so are unconditional. Others are almost unavoidable in modules that are intentionally incomplete to maximize reusability (e.g. missing declaration). The checks for those that could be corrected by inheriting modules are deferred.

7.2.2 MetaLexer Backend

The structure of MetaLexer backend is very simple. Since it does not perform any transformations of its own, it simply uses intertype declarations to add a pretty-printing function to each AST node type. Then it creates a public method in the Layout class that creates appropriate layout and component files and fills them with the pretty-printing output of the corresponding nodes.

7.2.3 JFlex Backend

The JFlex backend is more complicated because it has to transform a high-level MetaLexer specification into a lower-level JFlex specification. Many things that are implicit and hidden in MetaLexer must be made explicit in JFlex. In addition, care must be taken to compensate for the subtle semantic differences between MetaLexer and JFlex (for an example, see *Section 5.3.1*).

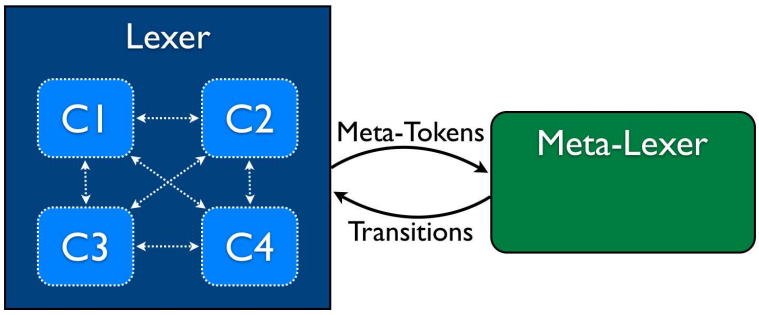


Figure 7.2 A high-level view of the organization of a lexer generated JFlex backend.

Figure 7.2 shows, at a high-level, the structure of the lexer generated by the JFlex backend. On the left is the main lexer, which contains the components. On the right is the meta-lexer, which controls the transitions amongst the components. As input is matched, the lexer generates meta-tokens and sends them to the meta-lexer. The meta-lexer processes meta-tokens and optionally signals transitions in the lexer.

The most interesting aspects of the implementation pattern demonstrated by the JFlex backend are described below.

Scoping

Creating an AIL-level scope at the level of each component and layout is a non-trivial task. The MetaLexer-to-JFlex translator illustrates a possible solution.

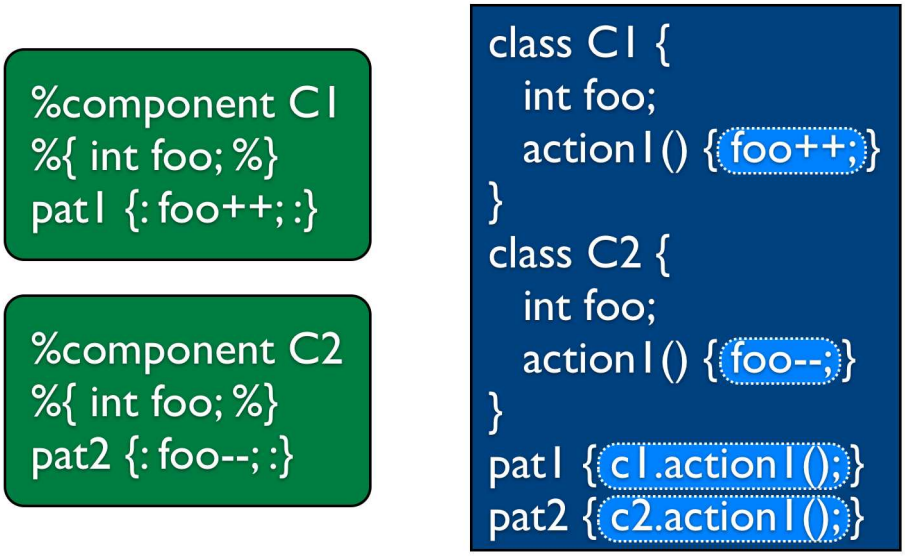
Components have their own state whereas lexical states do not. Therefore, in converting from one to the other, care must be taken to give each component its own namespace so that it does not interfere with any other components. On the other hand, all components share state that is declared at the layout level and must add tokens to the same token stream.

The MetaLexer-to-JFlex translator solves this problem by creating a non-static inner class for each component. This gives each component its own namespace and state plus access to the shared layout state, which is declared in the top-level lexer class. Each component class is instantiated exactly once as a field of the lexer class. *Figure 7.3* shows a high-level example of this process. *Figure 7.3a* shows the original MetaLexer specification. Notice that both components have a field called *foo*. These fields are unrelated because each component has a separate scope. *Figure 7.3b* shows the resulting JFlex specification, which contains an inner class for each component. The highlighted regions are oversimplified and will be expanded upon later.

Layout state must be stored in the top-level class because it is part of the API of the lexer. If it was wrapped in an inner class, then clients of the lexer would have to access them differently.

One limitation of this inner-class approach is that components cannot contain any static fields or methods. This is because Java forbids static members in non-static inner classes. This is a relatively minor problem that can be solved either by removing the static modifier or by promoting the member to the layout-level and adding a declare-extern pair.

Boilerplate code generated by the MetaLexer-to-JFlex translator (which is required since JFlex does not support the same abstractions) is wrapped in another inner class to prevent name conflicts with programmer-defined state. Much of it is completely static and could easily be moved into a top-level class but this would introduce a requirement for a runtime jar. Depending on a separate jar for execution is undesirable because it places an added



(a) MetaLexer – Components **(b)** JFlex – Inner Classes

Figure 7.3 Translating components to JFlex code

burden on the clients of the jar.

Generated Names

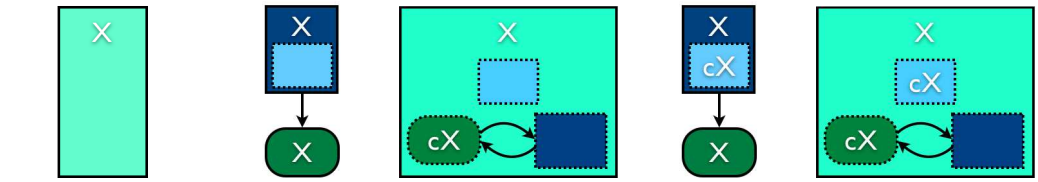
Creating a separate inner class for each component solves one problem and introduces another. Every time a new name is used, there is a potential for conflict with user-specified names. For example, if the original specification declared an inner class called *CompClass* and a component *Comp*, then calling the inner class for the component *CompClass* will cause a conflict.

Following JFlex's precedent, we have largely ignored this problem. That is, we generate fairly complex names that are unlikely to conflict with user-specified names, but take no action to eliminate or even flag conflicts. The resulting lexer will simply fail to compile and the user will have to choose another name. A more ambitious implementation could inspect identifiers in user-specified AIL code and modify generated names to eliminate conflicts.

In order to be able to use reasonable identifiers in our generated code, we do wrap most of

the state information and helper functions in another inner class. This way, the user only has to worry about conflicting with the name of a single class and its instantiation, rather than all generated state variables and functions.

This problem is most noticeable when alternately apply the JFlex-to-MetaLexer and MetaLexer-to-JFlex translators. For example, *Figure 7.4* illustrates one such problem at a high level. *Figure 7.4a* shows an initial JFlex specification. It is a black-box – we do not know anything other than its name. *Figure 7.4b* shows the result of translating this specification to MetaLexer. It consists of a single layout and a single component, both named for the original specification. The pale blue box in the layout indicates that there is not presently any AIL code of interest. *Figure 7.4c* shows the translation back to JFlex. Now the JFlex specification contains an inner class corresponding to the component in *Figure 7.4b*. *Figure 7.4d* shows the second translation to MetaLexer. Once again, there is one layout and one component, both named for the JFlex specification. However, now the inner class has been moved into an AIL code region (pale blue) because it is not recognized as a component. Finally, in *Figure 7.4e*, we see a conflict between the inner class corresponding to the component (green) and the inner class from the AIL code region (pale blue), which formerly corresponded to a component.



(a) Original JFlex **(b)** Generated MetaLexer 1 **(c)** Generated JFlex 1 **(d)** Generated MetaLexer 2 **(e)** Generated JFlex 2

Figure 7.4 Collisions of generated names in repeatedly translated code

However, even if this problem was resolved (e.g. by examining user-specified names), the JFlex-to-MetaLexer and MetaLexer-to-JFlex translators would still not interact nicely (see *Section 5.3.3*).

Actions

Actions are an important part of the scoping problem. They must be triggered by the rules of the top-level lexer, but they must have access to the state of their containing components.

MetaLexer accomplishes this by turning actions into methods of the component inner classes and then calling these methods from the actual rule actions. Unfortunately, this is less straightforward than it seems. Actions are not actually valid method bodies because they may or may not return a value. MetaLexer addresses this issue by having action methods *always* return *potential* values (rather than *potentially* returning a *definite* value). That is, each action method always returns a value, but that value may be either a true return or empty (this maybe-value pattern should be familiar to users of Haskell).

Unfortunately, this introduces a new problem. It is, in general, impossible to determine whether or not a given return statement will be executed. Thus, the highlighted method bodies in *Figure 7.3b* are insufficient. MetaLexer must somehow provide a fallback that returns a non-value, (*nothing*), if and only if the original action does not return a value.

Fortunately, §*14.21* of the Java Language Specification [GJSB05] tells us that code following an if-statement with the condition *true* is never considered to be unreachable. That is, even if the unconditionally executed body is known to leave the method (e.g. by returning or raising an exception), the code after the block will never be the subject of an "unreachable" code error. This means that if the original action code is wrapped in an if-statement with the condition *true* and followed by a non-value return, then a value will always be returned. The pseudo-code in *Listing 7.1* illustrates our solution.

```
1 if(true) {
2    //original action code
3 }
4 return Maybe.Nothing();
```

Listing 7.1 Pseudo-Code for an Action Method

With this accomplished, it remains only to determine what should go in the rule actions of the generated JFlex specification (highlighted in *Figure 7.3b*). Each action must accomplish the following, in order: the original action must be evaluated, the associated meta-token

must be generated, any exceptions raised by the original action must be raised, and (if no exceptions have been raised) any return value from the original action must be returned. The pseudo-code in *Listing 7.2* illustrates one way of accomplishing this.

```
1 Maybe<? extends TokenType> maybeResult;
2 try {
3    maybeResult = compInstance.actionMethod();
4 } finally {
5    generateMetaToken();
6 }
7 if(maybeResult.isJust()) {
8    return maybeResult.fromJust();
9 }
```

Listing 7.2 Pseudo-Code for an Action

Abstraction Violation

Despite determined efforts to avoid it, it was found to be necessary to violate the opacity of the AIL. That is, to provide JFlex with certain information that it requires but which is not general enough to expose in MetaLexer itself, it is sometimes necessary to read and even modify some free-form AIL strings.

The required reading is fairly innocuous. First, in order to give the meta-lexer the correct package declaration, it is necessary to search the layout headers for the package declaration that will be included in the JFlex specification. Second, to avoid constantly casting to and from *java.lang.Object*, the JFlex backend locates the JFlex **`%type`** directive amongst the layout options and uses that type in a variety of method signatures. Finally, component code regions are searched for the 'static' keyword so that appropriate errors can be raised.

The modification is a little bit more complicated. While reading *Section 7.2.3*, astute readers will have noticed that the value returned by an original action and the non-value returned by the action method do not have the same type. MetaLexer corrects this problem by wrapping the true return value from the action in an object of the correct type. Unfortunately, this cannot be accomplished without modifying the original action code – each occurrence of *return x;*, where *x* is an arbitrary expression, is replaced by *return Maybe.just(x);*. This

transformation will not fail very gracefully if the Java in the action code is malformed, but in that case the lexer would not have worked anyway.

Meta-Lexer

Once the components have been translated to JFlex, the layout(s) must be dealt with. As we saw in *Section 4.2*, the layout resembles another lexer with meta-patterns as regular expressions over an alphabet of symbols (i.e. meta-tokens and regions). By associating a distinct integer with each symbol, it becomes possible to express each start and end meta-pattern as a regular expression (over integers). Each regular expression is combined with a transition action to form a rule in the 'meta-lexer'[11].

Since the meta-lexer is just another lexer, it is tempting to specify it in JFlex as well. Superficially, the JFlex LSL appears to be ideal for describing the behaviour of the meta-lexer. Unfortunately, it is unsuitable because of one small but important semantic difference. Whereas JFlex (and other traditional lexers) search for the 'longest match', the meta-lexer searches for the 'shortest match'. That is, in the meta-lexer we want to match and perform a transition as soon as *any* meta-pattern has been observed. In contrast, a JFlex lexer would note the observation of the meta-pattern and then continue processing meta-tokens to ensure that no longer match was possible.

In spite of this semantic difference, we use the syntax of JFlex to illustrate the organization of the meta-lexer. Note, however, that no such syntax is ever produced by the backend. In fact, we actually build and generate code for finite automata (see later in *Section 7.2.3*).

States

The meta-lexer has a separate lexical state for each embedding to ensure that meta-patterns are not matched unless the system is in the correct embedding. The lexical state for a given embedding contains two types of rules.

First, the lexical state contains the start patterns of all embeddings hosted by the current

[11] Distinguish: *MetaLexer* – the entire LSL – versus *meta-lexer* – the lexer of meta-tokens.

embedding's guest component. For example, if embedding E_1 has host H_1 and guest G_1 and embedding E_2 has host H_2 and guest G_2, and if $G_1 = H_2$, then the lexical state for E_1 contains the start meta-pattern for E_2.

Second, the lexical state contains all end patterns for the corresponding embedding.

Within these groups (i.e. start and end meta-patterns), rules are in the same order as in the original MetaLexer layout specification.

Additionally, there is an initial lexical state containing the start meta-patterns of all embeddings hosted by the start component of the lexer.

Listings 7.3 & 7.4 show an example of this translation. *Listing 7.3* shows part of a layout – a start component and two embeddings. *Listing 7.4* shows the corresponding lexical states in the meta-lexer (in simulated JFlex). First, there is a *BASE* state containing rules for the start meta-patterns of all embeddings hosted by the start component. Then, for each embedding, there are rules for the start meta-patterns of all embeddings hosted by the embedding's guest and a rule for the embedding's end meta-pattern.

```
1  %start c1
2  %%
3
4  %%embed
5  %name embed1
6  %host c1
7  %guest c2
8  %start START_E1
9  %end END_E1
10
11 %%embed
12 %name embed2
13 %host c2
14 %guest c3
15 %start START_E2
16 %end END_E2
```

Listing 7.3 Meta-Lexer Lexical States Example – MetaLexer

```
<BASE> {
   //start meta-patterns from embeddings with host 'c1' (start
      component)
   {START_E1} { /* trigger embed1 */ }

   //no end meta-patterns since this is not a real embedding
}

<embed1> {
   //start meta-patterns from embeddings with host 'c2' (guest of
      embed1)
   {START_E2} { /* trigger embed2 */ }

   //end meta-pattern of embed1
   {END_E1} { /* trigger restore */ }
}
<embed2> {
   //start meta-patterns from embeddings with host 'c3' (guest of
      embed2)
   //N/A

   //end meta-pattern of embed2
   {END_E2} { /* trigger restore */ }
}
```

Listing 7.4 Meta-Lexer Lexical States Example – Simulated JFlex

Transitions

The meta-lexer keeps track of the current state of the lexer as a whole by transitioning amongst its lexical states. The current lexical state always corresponds to the most recently triggered embedding. Hence, the current component is always the guest of the embedding corresponding to the current lexical state.

The transitions amongst the lexical states are tracked on a stack so that the meta-lexer can return to the correct lexical state when an embedding ends (i.e. when an end meta-pattern is observed).

Extraneous Symbols

Obviously, not all meta-tokens and regions will be part of a start or end meta-pattern. If an extraneous symbol is observed, partially-matched patterns that cannot accept it are discarded, as is the symbol itself.

<BOF>

The **<BOF>** meta-pattern is treated like a normal symbol (i.e. meta-token or region). It is assigned a distinct integer, which is matched in the same way as any other integer. It is generated exactly once at the beginning of the meta-token stream. There is no reason to generate more than one **<BOF>** because pure BOFs are handled elsewhere (*Section 6.9*). That is, the first time **<BOF>** is consumed, some other symbol(s) will also be consumed (since the pattern is not a pure BOF) and the stream will no longer be un-started.

Communication

The lexer communicates with the meta-lexer via the meta-lexer's *processSymbol* method. The method accepts a single *int* (representing a meta-token, a region, or **<BOF>**) and returns a *Transition* object, possibly null. If the transition object is non-null, it contains the sequence of integers matched and an embedding index. If the embedding index is non-negative, then it indicates the embedding that has begun (i.e. a start meta-pattern match). Otherwise, it indicates that the current embedding has ended (i.e. an end meta-pattern match).

Dependencies

Since the integer values of the symbols are assigned at compile-time, they can be included directly in the specification of the meta-lexer. As a result, at runtime, the meta-lexer is completely independent of the lexer proper. It simply accepts integers and outputs transi-

tions. In particular, there is no runtime mapping between symbols and integers – they are all hard-coded in the generated class.

Finite Automata

Since JFlex was not suitable for the implementation of the meta-lexer, we had to create our own lexer. We took the fairly standard approach of constructing a discrete finite automaton (DFA) for each lexical state. However, instead of looking for longest matches, we simply match as soon as an accepting state is reached.

To obtain a DFA, we construct a non-deterministic finite automaton with ε-transitions (an ε-NFA) from the meta-patterns of the lexical state, eliminate the ε-transitions to form a non-deterministic finite automaton without ε-transitions (an NFA), and then apply the subset construction algorithm to produce a DFA. This is a standard process, described in detail in many texts (e.g. [App98], [Mar03]).

Figure 7.5 shows the structure of the ε-NFA for a lexical state. It consists of a distinguished start state and the ε-NFAs corresponding to the individual meta-patterns in the lexical state (shown with dotted borders). The start state has an edge that loops back to itself on any possible input. This loop handles extraneous characters. Basically, the machine will accept any amount of nonsense before it starts matching an actual meta-pattern. Once the nonsense has been consumed by the self-loop, the machine follows an ε-transition to the appropriate meta-pattern sub-machine and completes the match. The accepting state of the sub-machine (shown with a double border) contains a unique integer identifying the meta-pattern matched.

The generated DFA is minimized to save space and execution time. We followed the fixed-point algorithm described in section 5.3 of [Mar03], but we choose a different initial condition, also distinguishing accepting states with different meta-pattern identifiers. The reasons for this are discussed below.

Unfortunately, when the DFA indicates a match and a sequence of integers is recovered, there is no way to determine which prefix of the match came from the self-loop (i.e. which

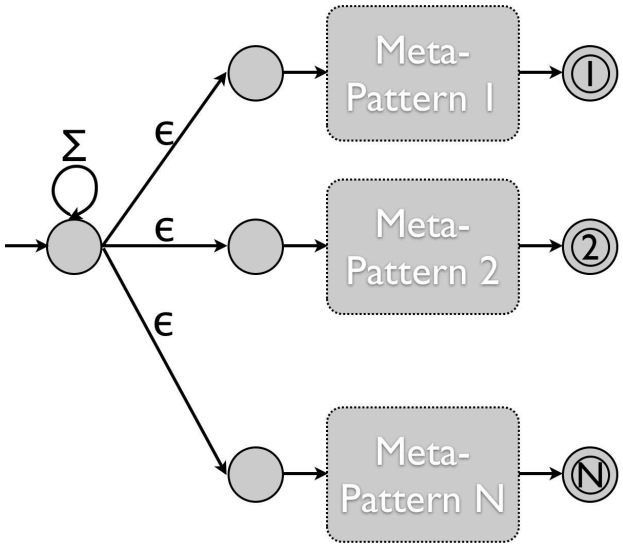


Figure 7.5 A high-level view of the ε-NFA generated for a lexical state of the meta-lexer.

are extraneous). This is why each accepting state stores the identifier of its corresponding meta-pattern; when a match is made, we can work backwards through the meta-pattern to determine which suffix of the match comes from the meta-pattern itself. Everything not in the suffix is discarded as extraneous. Since the meta-lexer uses shortest match semantics, there is no danger that this suffix will be shorter than the actual match.

For example, suppose we have just transitioned into a component that represents a Java class. Then, barring intervening start meta-patterns, the next thing we are looking for is the closing brace that will end the component. Hence, we will be in an ε-NFA that looks something like *Figure 7.6*. However, we are likely to see a lot of extraneous symbols before we reach the end of the component – keywords, parentheses, dots, etc. Our raw match might look like `IF DOT WHILE RETURN RBRACE`[12]. While this is indeed the sequence of symbols that we have matched, only the `RBRACE` was actually matched by the meta-pattern. When we work backwards through the match, we match against an ε-NFA like *Figure 7.7*, which picks out just the `RBRACE`.

The backwards matching of meta-patterns is accomplished by building a DFA for the reverse of each meta-pattern. The process is the same as for lexical states except that there is

[12]Clearly, this is not a realistic trace of a Java class.

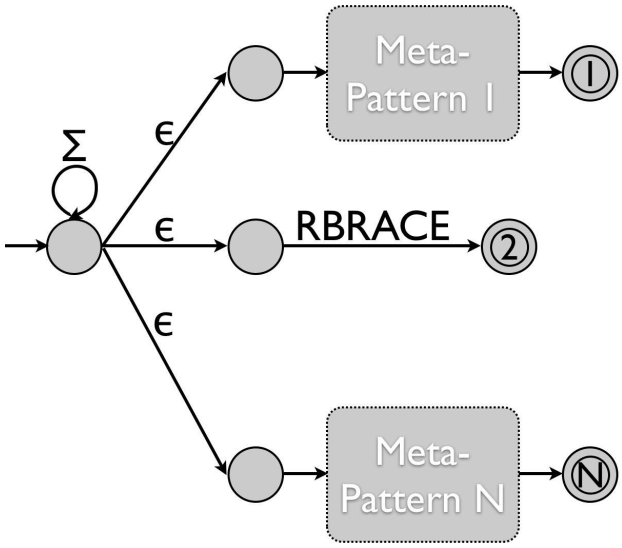


Figure 7.6 A high-level view of the ε-NFA generated for a lexical state of the meta-lexer.

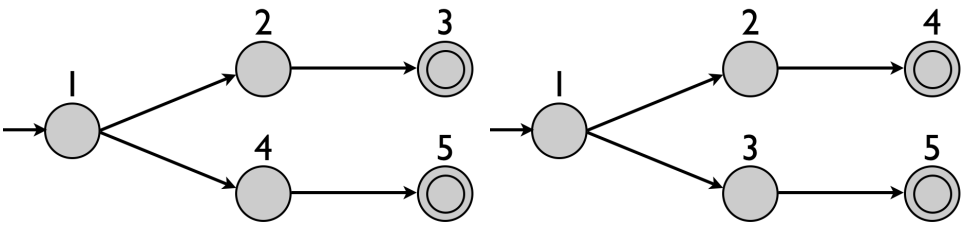
Figure 7.7 A high-level view of the reverse ε-NFA generated for a single meta-pattern.

only one meta-pattern in each ε-NFA and the self-loop is omitted.

Each DFA can be represented by two arrays: one for transitions and one for actions. The transitions array is two-dimensional with states on one axis and symbols on the other. The actions array is one-dimensional with an element for each state. Hence, we can encode each DFA as a pair of statically initialized *Integer* arrays. These are both compact (especially when accepting-state transitions are omitted) and quick to initialize (since no string parsing is required).

We perform a small optimization that is very effective in practice. Since the numbering of the DFA states is arbitrary, we can shuffle all of the accepting states to the end (i.e. give them the highest numbers). Then, when we print out the transition table for the DFA, we can omit those rows (since we are done as soon as we reach an accepting state) and still have a contiguous matrix. *Figure 7.8* shows a sample renumbering.

Since most meta-patterns consist of a single symbol, most of the minimized DFAs consist



(a) Original Numbering **(b)** Revised Numbering

Figure 7.8 Renumbering DFA states to move accepting states to the end.

of a single start state with transitions to a variety of accepting states. That is, in practice, the meta-pattern DFAs tend to have only one non-accepting state. As a result, most meta-pattern DFAs have a single-row transition table.

Chapter 8

Case Studies

This chapter describes the experience of using MetaLexer to specify lexers for three real programming languages: McLab, abc, and MetaLexer itself. We first describe the process of developing the specifications and the improvements that resulted. We then present our experimental findings about the performance of these specifications.

8.1 McLab

The Sable Lab[1] at McGill University is developing an optimizing compiler framework for scientific programming languages called McLab[2]. In its first incarnation, McLab will be a compiler for Matlab[3].

Unfortunately, the syntax of Matlab is rather convoluted, apparently having grown organically over the course of decades. As a result, some features are not amenable to normal lexing and LALR parsing techniques. For this reason, the McLab team has defined a functionally equivalent subset of the language, called Natlab, that omits some of the more troublesome syntax (e.g. command-style function calls).

[1] http://www.sable.mcgill.ca/
[2] http://www.sable.mcgill.ca/mclab/
[3] http://www.mathworks.com/products/matlab/

Originally, the Natlab lexer was specified using JFlex[4]. However, since the Natlab language is intended to be the foundation of many language extensions, the Natlab lexer has been re-specified in MetaLexer.

8.1.1 Improvements

Re-specifying Natlab in MetaLexer resulted in three substantial improvements. First, the new lexer is extensible. Second, nearly all of the action code in the JFlex lexer was eliminated in favour of MetaLexer language constructs. Third, all lexical states were replaced by components. These improvements are particularly gratifying in light of Natlab's inherent complexity.

Extensibility

Eventually, McLab will support type inference for Matlab programs. For now, however, types are specified manually in specially formatted comments called annotations. Support for annotations was added before the lexer was converted for MetaLexer. In the original JFlex specification, however, there was no way to separate the extension from the rest of the language. Instead, the extended language replaced the original language. With MetaLexer, the two languages – extended and unextended – can co-exist.

Given the MetaLexer specification for Natlab, creating an extension for annotations was easy. First, lexical rules for annotations were specified in a new component. Then, components were created for the start and end delimiters of annotations. For each component that needed to use one of the new delimiters (i.e. anywhere an annotation can occur), a new component was created inheriting both the original component and the delimiter component. Finally, a new layout was created. The new layout extends the original layout, introducing a single new embedding for annotations, and replacing all components with their new annotation counterparts.[5]

[4]Disclosure: the JFlex lexer for Natlab was built by the creator of MetaLexer.

[5]Since Natlab does not presently use an extensible parser, annotations are still treated as opaque blobs and handed off to a separate parser.

The extension required eight new files: the annotation lexical rules, the annotation start delimiter, the annotation end delimiter, the extended layout, and four components that combine an existing component with an annotation delimiter component (four lines each). It could be done with fewer, but this solution is clean and easy to read.

A colleague, Toheed Aslam, is presently working on the first major extension of Natlab, AspectMcLab[6]. It will add aspects to the Matlab programming language. Toheed's initial experiences have been positive – extension is straightforward and the specification is clean and modular. He found pair filters to be the most difficult feature of MetaLexer to understand, so we have made an effort to explain them in greater detail and provide examples (*Section 4.2.1*).

Action Code Elimination

The JFlex specification for Natlab required a lot of embedded Java code to keep track of state and accomplish lexical state transitions. In the MetaLexer specification, virtually all Java code has been eliminated. Simple methods for constructing symbols, throwing errors, parsing numeric literals, and passing comments directly to the parser remain, but code for tracking position and maintaining a stack of lexical states has been replaced by normal MetaLexer control flow. Nearly all lexical rule actions consist of a single statement – an append, a return, or an error.

Lexical State Elimination

The MetaLexer specification for Natlab does not declare *any* lexical states. All transitions are controlled by the layout. As a result, the interaction of the components can be understood without reading any Java code. Furthermore, the lexer will be much easier to port to another AIL/LSL because none of the transition logic will have to be modified.

[6] http://www.sable.mcgill.ca/mclab/

8.1.2 Difficulties

In most cases, it was straightforward to replace transition logic written in Java with simple embeddings. However, certain features of Natlab required special handling.

Transpose

In Natlab, a single-quote can indicate either a string literal delimiter (i.e. opening or closing a string) or the transpose of a matrix. The two cases are distinguished by the token immediately preceding the single-quote.

In the JFlex implementation of the lexer, a flag was set after each token that could precede a transpose operator and cleared after each token that could not. This process was simplified slightly by filtering all symbol returns through a common method, but rules that did not return tokens still had to explicitly clear the flag. Obviously, this system was quite fragile since it required each new rule and token type to correctly update the logic.

In the MetaLexer implementation, we created a component for the transpose. Any rule matching text that can precede a transpose operator triggers a transition to the *transpose* component. The component consumes the operator and transitions back. To limit the number of spurious transitions, the meta-token is generated only if the lexer's lookahead indicates that a single-quote will follow.

The MetaLexer solution is much easier to understand because no code is required for rules that do not immediately precede transpose operators. In hindsight, the MetaLexer solution could have been applied in the JFlex lexer. However, the solution only presented itself after reframing the problem in terms of components and meta-tokens.

Field Names

In Natlab, it is legal to use keywords as names for structure fields. Since structure field names are accessed using the dot operator, keywords following the dot operator should be treated as normal identifiers.

The JFlex implementation of the lexer handled this by switching into a special keyword-less state after each dot operator. Unfortunately, many of the rules of this state were shared with other lexical states (since it was inclusive) so special logic was required to leave that state after returning any symbol.

The MetaLexer implementation simply transitions to a component that only accepts identifiers. If it sees anything else, it pushes it back into the lexer buffer and returns to the previous component. Clearly, this same approach was possible in JFlex but, as above, it was not obvious until the problem was reframed by MetaLexer.

Matrix Row Separators

Natlab has special syntax for constructing two-dimensional matrices – elements are separated by commas and rows are separated by semicolons or line terminators. However, if the end of a row is indicated by a line terminator, Natlab allows a comma or semicolon, whitespace, and a comment to appear after the last element in the row. *Listing 8.1* shows an example of such a matrix. To avoid grammar conflicts, the parser requires that the comma/semicolon and line terminator be returned as a single token.

```
1 a = [1, 2, 3, %this is the end of the first row
2     4, 5, 6]
```

Listing 8.1 Example – Natlab Matrix Syntax

The MetaLexer implementation handles this in more or less the same way as the original JFlex implementation. When a comma or semicolon is encountered, the lexer switches to a component/lexical state in which the line terminator is sought. If it is found, a single large token is returned. Otherwise, only the comma or semicolon is returned. Unfortunately, in MetaLexer, there is no good way to keep track of the position of the original comma or semicolon so it must be stored in a (lexer-)global variable shared by the two components (i.e. the one that sees the comma or semicolon and the one that looks for the line-terminator).

End Expression

Natlab classes use a number of keywords that are not required by non-OO programs. To limit the impact on the programmer, Natlab allows these keywords to be used as identifiers outside of class bodies. Unfortunately, this means that the lexer has to keep track of whether or not it is in a class body. Superficially, it appears that this can be accomplished by matching *end* keywords with the beginnings of the corresponding blocks until the end of the class is found. Unfortunately, within index expressions (i.e. expressions indicating where to index into an array), the *end* keyword has another meaning – it evaluates to the last index of the array.

The JFlex implementation addressed this problem by keeping track of the bracketing level. An *end* keyword ends a block if-and-only-if it is not inside any brackets (round, curly, or square). This requires a global counter plus appropriate increments, decrements, and checks.

In MetaLexer, we eliminated the counter by duplicating the *class* component[7]. The *class_bracketed* component is exactly the same as the *class* component except that the *end* keyword does not generated a meta-token in *class_bracketed* so it never gets paired with a block opening. This component starts at an open-bracket and ends whenever an unpaired close-bracket is encountered.

Multiple Meta-Tokens

Occassionally it seems desirable to label a single rule with two meta-tokens. For example, an identifier can indicate both that a field name has been seen and that a transpose operator could follow. We found that these cases are easily accomodated by introducing a new meta-token with both meanings and then using meta-pattern classes to allow it in both situations (e.g. *END_FIELD_NAME_START_TRANSPOSE* in *Listing 8.2*).

[7] The use of inheritance and helper components significantly reduces the amount of duplicate code.

```
1  %%embed
2  %name field_name
3  %host base, class
4  %guest field_name
5  %start START_FIELD_NAME
6  %end [END_FIELD_NAME END_FIELD_NAME_START_TRANSPOSE]
7
8  %%embed
9  %name field_name_transpose
10 %host field_name
11 %guest transpose
12 %start END_FIELD_NAME_START_TRANSPOSE
13 %end END_TRANSPOSE
```

Listing 8.2 Extract – Multiple Meta-Tokens

Error at End-of-File

Natlab uses the Beaver PSL, which requires that a special end-of-file (EOF) token be returned, even if an error has occurred. This can be problematic if the EOF causes a lexical error.

For example, *Listing 8.3* shows a typical string literal component (the one from Natlab, actually). The problem is how to handle **<<EOF>>**. We would like to throw an exception to indicate that the string literal is unterminated, but if we do so, then we cannot return the EOF symbol required by the parser. If we generated a meta-token that will take us to another component that will generated the EOF symbol, then the append block will trigger and return the string literal token, even though it is incomplete[8].

To avoid unwinding the entire embedding stack, we transition forward rather than back on an EOF error. A dedicated component returns the EOF token required by the parser and then the parser stops requesting tokens (i.e. lexing ends).

This solution is not ideal, but it is acceptable.

[8] This could be avoided with a flag, but the extra token makes little difference after the error.

```
1  %component string
2
3  %extern "Symbol symbol(short, Object, int, int, int, int)"
4  %extern "void error(String) throws Scanner.Exception"
5
6  %appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
       endCol, String text)*/
7     return symbol(STRING, text, startLine + 1, startCol + 1, endLine +
          1, endCol + 1);
8  %appendWithStartDelim}
9
10 %%
11 %%inherit macros
12
13 "'" {: append(yytext()); :}
14 "'" {: /*just end string - %append will handle token*/ :} END_STRING
15 {ValidEscape} {: append(yytext()); :}
16 \\ {: error("Invalid escape sequence"); :}
17 {LineTerminator} {: error("Unterminated string literal"); :}
18 %:
19 %:
20 <<ANY>> {: append(yytext()); :}
21 <<EOF>> {: error("Unterminated string literal"); :} EOF_ERROR
```

Listing 8.3 Extract – Error at End-of-File

8.2 abc

abc [ACH+05] is an extensible research compiler for AspectJ [KHH+01]. It has two different implementations: one using Polyglot [NCM03] and the other using JastAdd [EH07b]. In this chapter, we will focus on the Polyglot implementation. Logically enough, the Polyglot implementation uses the Polyglot Parser Generator (PPG) for parsing. Since, Polyglot does not include a corresponding extensible lexer, abc uses an ad-hoc approach [HdMC04].

Interestingly, excluding the extension mechanism, the abc lexer is remarkably similar in structure to a MetaLexer lexer[9]. The abc lexer breaks AspectJ into four sub-languages: java, aspect, pointcut, and pointcut-if-expression. The first three are self-explanatory. The last refers to the bits of aspect syntax that appear within if pointcuts. The nesting structure

[9] We discovered this after the fact.

of these sub-languages is tracked on a stack which is pushed and popped when certain tokens are observed.

The extension mechanism of the abc lexer is fairly intricate. Each extension provides a class containing a dynamic list of keywords, each of which may have an associated lexical state transition. The lexer itself matches all keywords as identifiers and then checks them against its list at runtime. If the identifier is found to be a keyword, then an appropriate keyword token is returned and the associated transition, if any, is performed.

As an experiment, we tried replacing the abc lexer with our own version written in MetaLexer. We also replaced two extensions: eaj and tm[10]. eaj (Extended AspectJ) extends AspectJ with experimental new join points and global pointcuts. This requires adding a few new keywords plus a couple of extra transitions (*let* has the same syntax as *if* and *global* has the same syntax as *declare*). tm (TraceMatches) allow users to create more complicated pointcuts using temporal logic. This requires a few new keywords plus pointcuts that are terminated by semicolons, rather than advice.

In the end, our modified abc compiler (i.e. with the new lexer) passed all but one of the (roughly 1200) original abc test cases (for the aspectj, eaj, and tm languages). In the case where our compiler differed from the original, the divergence occurred only after a significant lexical error and could easily have been eliminated (see *Section 8.2.2*).

8.2.1 Improvements

MetaLexer is an excellent choice for specifying the syntax of AspectJ. It has many advantages over the existing implementation.

Pointcut-If-Expression

The biggest improvement made possible by MetaLexer was the elimination of the pointcut-if-expression sub-language. In the original implementation, the aspect and pointcut-if-expression languages were identical except that the pointcut-if-expression would return

[10]For descriptions, see http://abc.comlab.ox.ac.uk/extensions

to the previous sub-language upon balancing the opening parenthesis of the enclosing if expression. Since this balancing is specified at the embedding level, rather than at the component level, in MetaLexer, there was no reason to keep the sub-languages separate.

Clarity

It took us quite some time to figure out how the original lexer worked. The JFlex lexer declares lexical states for the four sub-languages but never transitions amongst them. Eventually, we realized that the transitions were attached to the dynamically-defined keywords. In MetaLexer, the transitions are where they always are – in the layout.

Furthermore, the embeddings for the aspectj, eaj, and tm languages are extremely easy to read. In particular, the embeddings that transition amongst the sub-languages read almost like the English descriptions in [HdMC04]. *Listing 8.4* shows a few examples.

```
1  %%embed
2  %name perclause
3  %host aspect_decl
4  %guest pointcut
5  %start [PERCFLOW PERCFLOWBELOW PERTARGET PERTHIS] LPAREN
6  %end RPAREN
7  %pair LPAREN, RPAREN
8
9  %%embed
10 %name declare
11 %host aspect
12 %guest pointcut
13 %start DECLARE
14 %end SEMICOLON
15
16 %%embed
17 %name pointcut
18 %host java, aspect
19 %guest pointcut
20 %start POINTCUT
21 %end SEMICOLON
```

Listing 8.4 Extract – Embeddings from aspectj.mll

Extensibility

Extending the base aspectj layout to eaj and thence to tm was very easy.

eaj adds new global keywords (i.e. affecting all sub-languages), pointcut keywords (i.e. affecting only the pointcut sub-language), and transitions. The new keywords were added by wrapping them in new components and then inheriting them in extensions of the original components. *Listing 8.5* shows an example – the new global keywords are inherited into a component extending the original *aspect* component. This was done for each component that needed the new keywords, then the extended layout performed the necessary replacements (e.g. *Listing 8.6*). Finally, the new embeddings were added to the extended layout.

```
1 %component eaj_aspect
2 %%
3 %%inherit eaj_global_keywords
4 %%inherit aspect
```

Listing 8.5 Extract – Adding New Global Keywords

```
1 %%inherit aspectj
2 %replace aspect, eaj_aspect
3 %replace aspect_decl, eaj_aspect_decl
4 %replace java, eaj_java
5 %replace java_decl, eaj_java_decl
6 %replace pointcut, eaj_pointcut
7 %replace pointcut2, eaj_pointcut2
```

Listing 8.6 Extract – Replacing Components

tm is similar – it extends eaj with a few new keywords and a new embedding. This is accomplished in exactly the same way (though inheriting from eaj rather than aspectj).

Since tm introduces substantially different language features, it might have benefitted from new lexical rules as well. However, since the existing lexer cannot add new sub-languages, there was no way implement this. With MetaLexer, on the other hand, adding a new sub-language would have been easy – particularly since trace matches have clear start- and end-delimiters.

Compile-Time Keywords

In the MetaLexer specification, keywords are defined in the lexer itself, rather than in an auxiliary class. They can be compiled into the lexer's finite automata, rather than being retrieved at runtime.

8.2.2 Difficulties

Though, on the whole, the MetaLexer solution was very elegant, a few blemishes remain.

Failed Test Case

Our MetaLexer specifications for aspectj, eaj, and tm passed all but one of over 1200 tests. In that test, a declare statement is not terminated. The two lexers agree up to the point of the error, but differ afterwards. In the existing lexer, the class body that follows the declare statement is lexed normally, whereas in the MetaLexer lexer, it is lexed as if it were a pointcut (which leads to further problems). It would be straightforward to handle this case in the MetaLexer specification – it is just a matter of allowing a transition from the pointcut sub-language to the java sub-language (see *Listing 8.7*) – but we decided that the behaviour was essentially inadvertent in the original lexer and preferred not to introduce a confusing new embedding purely for consistency in error situations.

New Keywords

The most straightforward method for adding new keywords to a MetaLexer lexer requires a surprising number of new files. Since this is the only kind of extension present in abc, the new specification looks rather verbose in this area. If this turns out to be a particularly common example, then a boilerplate-eliminating language construct might be worthwhile.

Superficially, it appears that the nicest solution would be to allow insertion of the new keywords directly into the old keyword helper components. However, it is frequently difficult

```
%%embed
%name pointcut_java_decl
%host pointcut
%guest java_decl
%start [CLASS INTERFACE]
%end LBRACE

%%embed
%name pointcut_java
%host pointcut
%guest java
%start %java_decl%
%end RBRACE
%pair LBRACE, RBRACE
```

Listing 8.7 Example – Unterminated Declare

to see all the implications of such a change. For this reason, we decided that modification of inherited components was too dangerous.

Runtime Behaviour

abc can change its lexical behaviour at runtime. In particular, the options singleton (*abc.main.options.OptionsParser.v()*) is used to determine whether multi-line comments should be nestable and the debug singleton (*abc.main.options.OptionsParser.v()*) is used to enable and disable keywords (e.g. *assert*). In the existing lexer, changing this behaviour at runtime is easy because all keywords are defined at runtime. In MetaLexer, however, special care must be taken. This is especially true for keywords that generate meta-tokens. It is simple enough to pushback an incorrectly matched keyword, but preventing generation of a spurious meta-token requires the conditional meta-token pattern described in *Section 4.10*.

Duplicate Components

Unfortunately, we found that some embeddings needed to be conditional. For example, a pointcut ends at a semicolon in a declare statement, but at a left brace if it is defining *before*, *after*, or *around* advice. We solved this problem by duplicating components and

then giving the duplicates different behaviours. For example, upon seeing *declare*, we transition to *poincut* but, upon seeing *before*/*after*/*around* we transition to *pointcut2* (see *Listing 8.8*).

```
1  %%embed
2  %name pointcut
3  %host java, aspect
4  %guest pointcut
5  %start POINTCUT
6  %end SEMICOLON
7
8  %%embed
9  %name advice
10 %host aspect
11 %guest pointcut2
12 %start [BEFORE AFTER AROUND]
13 %end LBRACE
14
15 %decl states
16 %duplicate components
```

Listing 8.8 Extract – Duplicate Pointcut Component

This does not cause any code duplication because the copy consists of a single inherit directive, inheriting the original. However, in the present implementation of MetaLexer, it does create duplicate code in the generated lexer. If this turns out to be commonly necessary, then it may be worthwhile to create an explicit duplication construct so that the back-end can do something more intellegent when duplication occurs.

Pre-Defined Character Classes

Since MetaLexer was designed to be cross-platform, we decided not to include the same language-specific predefined character classes as JFlex. In particular, MetaLexer lacks pre-defined character classes for Java identifier characters. Unfortunately, the explicit version of this character class is quite long and unpleasant to define.

8.3 MetaLexer

MetaLexer actually consists of two languages: one for components and one for layouts. The lexers for both are specified in MetaLexer. Originally, they were written in JFlex, but we wanted to show that we believe in our tool. The full specification can be found in *Appendix C*.

8.3.1 Improvements

Most of the benefits of re-implementing the MetaLexer lexer in MetaLexer have been discussed above. The MetaLexer version has no lexical states – all transitions are performed using embeddings; the action code that remains is mostly limited to append, return, and error; and the Java code for maintaining stacks of states and positions as well as text buffers is gone. However, there are a few other noteworthy improvements.

Macro Definition State

In the original JFlex specification, there is some trickiness involved when defining macros. Macros are defined by regular expressions. Unfortunately, regular expressions can contain elements that look like identifiers and vice versa. This problem was handled by matching ambiguous strings as identifiers if they appeared at the beginning of a line (modulo whitespace) and as regular expression elements otherwise.

When we ported these rules to MetaLexer, we found that we had to move the regular expression element rule (which is acyclic) ahead of the identifier rule (which is cyclic) to satisfy MetaLexer's rule group constraints. At first, this seemed like a significant problem. However, we realized that we could move the regular expression rules into a separate component. The macro definition component begins at an equals sign (in the option section of a component) and ends at the end of the line. This eliminated the ambiguity. Furthermore, after this change, all elements of the component option section ended with line breaks so we could always assume that they started at the beginning of lines. This eliminated a lot of

beginning-of-line-followed-by-whitespace patterns that were causing JFlex warnings.

This same solution was possible in JFlex, but we did not see it because we were not thinking about the problem in the right way.

Shared Code

The component and layout languages provide many excellent examples of the reusability of MetaLexer components. Since the two share so many lexical rules in common, nearly a third of the modules in their definitions are shared ('modules' since there are also shared helper components).

Merged Lexical States

As with the aspect and pointcut-if-expression sub-languages in abc (*Section 8.2.1*), the *INSIDE_ANGLE_BRACKETS* and *INSIDE_DELETE_ANGLE_BRACKETS* lexical states of the JFlex lexer differed only in their transition behaviour. By encoding this difference in embeddings rather than in components, we were able to merge the two (i.e. eliminate one).

8.3.2 Difficulties

Since MetaLexer was one of the use cases we have had in mind since we began the project, there were relatively few difficulties in creating its MetaLexer specification.

Error at End-of-File

We had to deal with the end-of-file error problem described in *Section 8.1.2*. The solution was the same.

Start Delimiter Position

The original JFlex specification dropped delimiters (e.g. quotes around string literals) in token values but counted them when determining position information. To achieve this same behaviour in the MetaLexer implementation we had to append empty start delimiters to all of the affected components. This was not particularly difficult, but it was a case that we had not considered when designing the start delimiter mechanism.

8.4 Performance

We compared the performance of MetaLexer and JFlex in a number of different areas: specification length, generated lexer length, compilation time, and execution time.

8.4.1 Testing Setup

Table 8.1 describes our testing environment.

Computer	MacBook Pro
Operating System	OS X 10.6.0
Processor Type	Intel Core 2 Duo
Processor Speed	2.33 GHz
Memory	2 GB
Java	1.6.0_15
Ant	1.7.0
JavaNCSS	32.53
MetaLexer	20090912
JFlex	1.4.1

Table 8.1 Testing Environment

All times were measured using Java's *System.currentTimeMillis()*. The numbers in the tables below reflect the averages of 11 runs each, excluding the first (warm-up), the best, and the worst[11].

[11]The entire suite can be found at http://www.cs.mcgill.ca/metalexer/

8.4.2 Code Size

For each of our six MetaLexer specifications – Natlab, aspectj, eaj, tm, component, and layout – we measured the number of files in the specification, the total length of the specification, and the size of the generated Java lexer class. The results are shown in *Tables 8.2-8.7*.

We defined the length of a specification file (JFlex or MetaLexer) to be the output of *wc -l*[12].

We defined the length of a Java file to be the total number of non-comment source statements (NCSS) as reported by the JavaNCSS tool[13]. This measure ignores whitespace and comments, making the comparison more accurate that a simple line count.

Superficially, it appears that the MetaLexer specification for Natlab is longer than the JFlex specification (*Table 8.2*). However, 122 of those lines (and 8 of those files) are for lexing annotations, something that the JFlex specification does with a single regular expression. That is, the original specification, having no capacity for extension, simply lexed annotations as opaque text regions. In contrast, the MetaLexer specification actually validates them. The generated lexer class is roughly 5 times as large, but a lot of that comes from the layers of abstraction around actions (see *Section 7.2.3*).

	JFlex	MetaLexer
Number of Specification Files	1	27[14]
Specification Size (LoC)	668	767[15]
Generated Class Size (NCSS)	859	4582

Table 8.2 Code Size for Natlab

The existing abc lexer specifications (*Tables 8.3-8.5*) are the only ones that contain both Java and specification files. As described in *Section 8.2*, the extension is accomplished using auxiliary Java classes. We included all such code in the specification size of the

[12] This is a conservative metric – in general, MetaLexer specifications contains more blank lines and/or comments.

[13] http://www.kclee.de/clemens/java/javancss/

[14] 8 are for lexing annotations

[15] 122 are for lexing annotations

lexers (each cell in the row has two values: one for the length combined length of the lexical specification files and one for the combined length of the separate Java files), but we excluded them from the file count unless they were absent from the MetaLexer version. For example, the *AbcExtension* classes used to initialize the keyword lists in the original lexer are still present in the MetaLexer implementation because they perform other functions as well. As a result, the *AbcExtension* classes are included in the specification size of the original JFlex lexer but not in the file count of either lexer.

The MetaLexer specification for the (abc) aspectj language is slightly shorter than the existing JFlex and Java specification, but the generated lexer class is nearly 11 times as long (*Table 8.3*). This is due in part to the layers of abstraction around actions and in part to duplicated code in the MetaLexer output. That is, JFlex treats rules that appear in multiple lexical states as shared whereas MetaLexer treats rules that appear in multiple components as copies. If the MetaLexer backend merged the rules as well (using information it already has available), its output size would shrink dramatically.

	JFlex	MetaLexer
Number of Specification Files	5	20
Specification Size (LoC/NCSS)	860/143	906/0
Generated Class Size (NCSS)	912	9852

Table 8.3 Code Size for abc – aspectj

The figures in *Tables 8.4-8.5* represent differences from those in *Table 8.3*. For example, the file count is represents the number of files added to the system to extend the lexer. (Files like *AbcExtension* that are not lexer-specific were not counted.) The exception is the MetaLexer generated class size. Since a separate Java class is generated for each layout, the file size is a total rather than a difference.

These figures are interesting because of the zeroes on the JFlex side. In the existing abc lexer, extensions require only a few lines of additional Java code – just adding some new keywords to the list. Since it is specialized to handle only this one type of extension, it does so very efficiently.

The most interesting thing about *Tables 8.6-8.7* is the amount of shared code (see *Sec-*

	JFlex	MetaLexer
Number of Specification Files	0	9
Specification Size (LoC/NCSS)	0/22	172/0
Generated Class Size (NCSS)	0	10388[16]

Table 8.4 Code Size for abc – eaj

	JFlex	MetaLexer
Number of Specification Files	0	4
Specification Size (LoC/NCSS)	0/8	71/0
Generated Class Size (NCSS)	0	10544[16]

Table 8.5 Code Size for abc – tm

tion 8.3.1). Considering the languages separately, the JFlex and MetaLexer specification sizes look very similar. However, the JFlex specifications for the two languages are independent, whereas the MetaLexer specifications overlap. Considering the languages together, the MetaLexer specification is shorter. Furthermore, the generated lexers are only 3-4 times as large.

	JFlex	MetaLexer
Number of Specification Files	1	26[17]
Specification Size (LoC)	837	880[18]
Generated Class Size (NCSS)	875	3199

Table 8.6 Code Size for MetaLexer – Component

For all of these languages, we see that the MetaLexer specification requires many more files. This is simply the result of a different design that places a greater emphasis on encapsulation. It does not result in longer specifications.

[16] Total – independent of previous generated classes.
[17] 10 are shared
[18] 243 are shared

	JFlex	MetaLexer
Number of Specification Files	1	18[19]
Specification Size (LoC)	628	594[20]
Generated Class Size (NCSS)	702	1937

Table 8.7 Code Size for MetaLexer – Layout

8.4.3 Compilation Time

Figure 8.1 shows how long it took to convert the specifications for our six languages into Java lexer classes. Two different values are shown for each MetaLexer specification – the time taken for the MetaLexer-to-JFlex translator to run and the time taken for the entire translation process (MetaLexer-to-JFlex plus JFlex-to-Java).

As we expected, compiling a MetaLexer specification takes quite a bit longer than compiling a JFlex specification (even ignoring the fact that MetaLexer compilation includes JFlex compilation). This makes sense because MetaLexer performs a lot of processing to handle multiple inheritance and performs much more validation than JFlex.

It is interesting to note that the MetaLexer-to-JFlex translation generally took only about half of the MetaLexer compilation time. This suggests that streamlining the JFlex output by the translator would substantially speed up compilation time.

The other interesting observation we can make about *Figure 8.1* is that, even though eaj and tm are tiny extensions, their presence slows down the translator substantially. This is because inherited code frequently needs to be re-checked in the context of the inheriting module. It might be able to reduce the slowdown by optimizing away some of these checks, but some duplication will always be necessary.

[19] 10 are shared
[20] 243 are shared

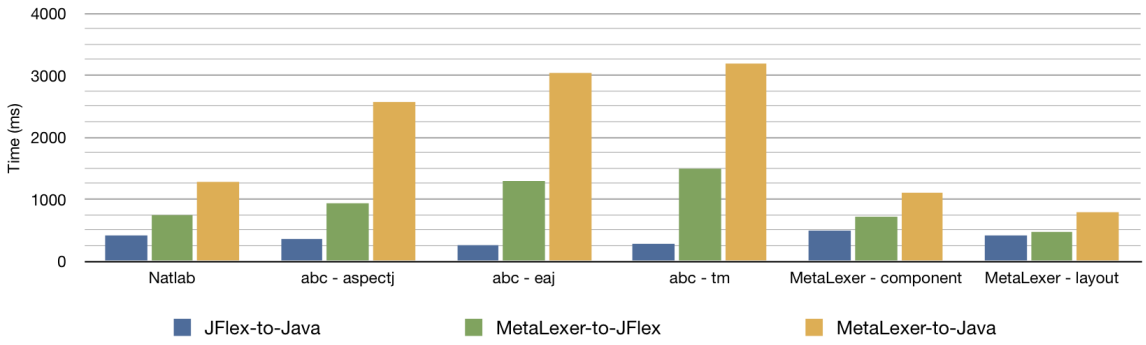


Figure 8.1 Compilation Times

8.4.4 Execution Time

For each of our six languages, we chose a variety of real-world benchmarks (i.e. files that are actually in use in real projects) and measured four execution times: the time taken to the lex the file with the original JFlex lexer, the time taken to lex the file with the new MetaLexer lexer, the time taken to parse the file using the the existing parser and the JFlex lexer, and the time taken to parse the file using the existing parser and the MetaLexer lexer. We measured the execution times of the lexers so that we could compare them directly and the execution times of the parsers to get a sense of how much of the overall runtime the lexer represents. *Figures 8.2-8.7* show the results.

Natlab

We drew our Natlab benchmarks from the suite used by the McLab group. Two of the four benchmarks – *benchmark2* and *reduction* – perform computations and the other two are drivers – *drv_edit* and *drv_svd*. *benchmark2* (308 lines) is a numerical computation benchmark for Matlab created by Philippe Grosjean[21]; *reduction* (141 lines) computes the LLL-QRZ factorization of a matrix (created by Xiao-Wen Chang and Tianyang Zhou); *drv_edit* (292 lines) is a test driver for an edit-distance calculator; and *drv_svd* (6494 lines) is a test driver for a function that computes the singular value decomposition of a matrix.

[21] http://www.sciviews.org/

Only small modifications were made to the files. First, the files were in normal Matlab syntax. We used a tool provided by the McLab project to convert them to Natlab. Second, we corrected an unescaped backslash in *benchmark2*.

We see from *Figure 8.2* that MetaLexer is slower than JFlex (which certainly makes sense, in light of the sizes of the generated classes) but we cannot say how much slower because most of the differences are below the error threshold of the timing mechanism. A rough estimate would be that MetaLexer is generally about 3 times slower (thought it may spike to 8 times).

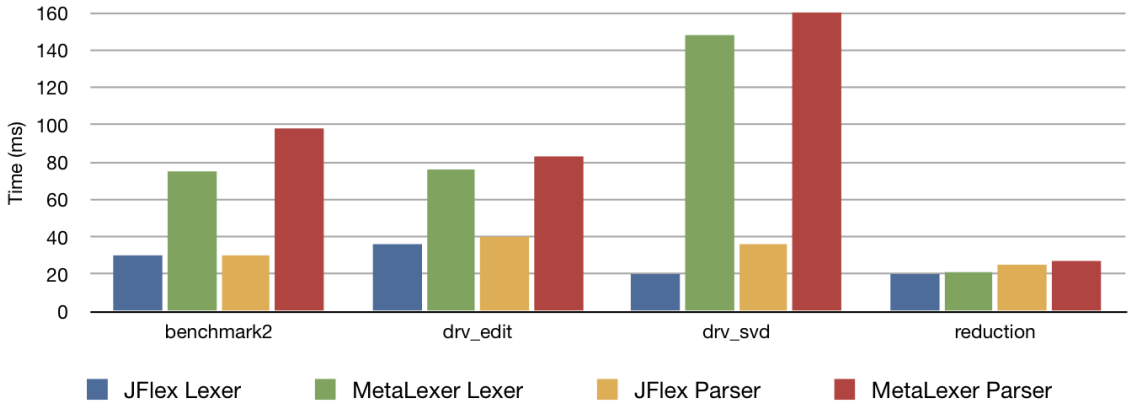


Figure 8.2 Execution Times for Natlab

abc

We drew our abc benchmarks from AspectBench[22] suites. This seemed prudent as the abc implementation of AspectJ differs slightly from the original ajc implementation.

Since we planned to re-test the aspectj benchmarks in each of the extensions (i.e. eaj and tm), we chose only three. *EnforceCodingStandards* (86 lines) is an aspect that logs all *null* returns from non-void functions; *Metrics* (134 lines) computes metrics of a running program (i.e. profiling data); and *MSTPrim* (212 lines) adds a strongly-connected-components method to a graph class.

[22] http://www.aspectbench.org/

These files are all relatively short, so it is hard to draw any conclusions from the execution times (see *Figure 8.3*). It does, however, seem likely that MetaLexer is roughly as fast as JFlex.

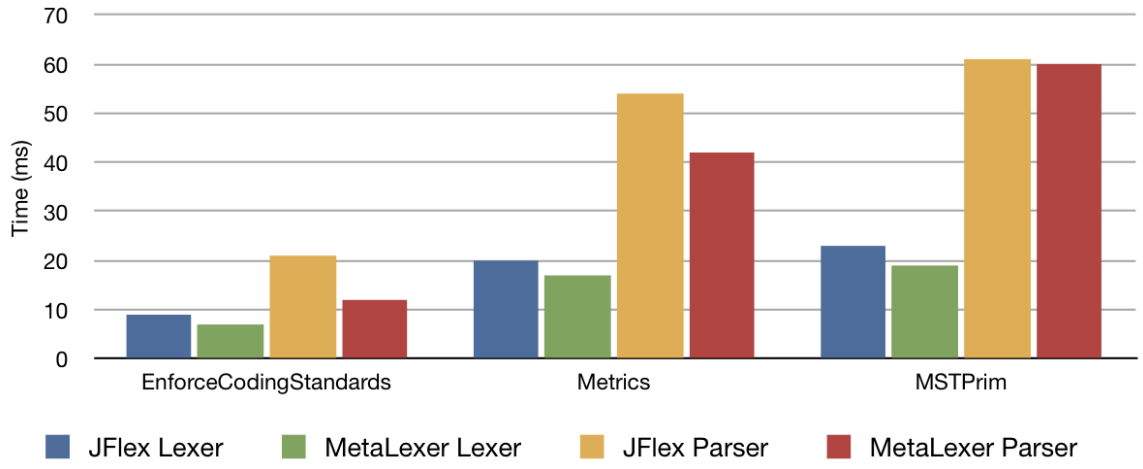


Figure 8.3 Execution Times for abc – aspectj

Since eaj is a testbed for experimental features, we were unable to find any real-world files that made use of the extension. Consequently, *Figure 8.4* shows only the runtimes for the aspectj benchmarks. As expected, it strongly resembles *Figure 8.3*. Some slowdown is evident, but it is difficult to quantify. It likely stems from the increased size of the generated class (see *Tables 8.3-8.5*).

Several papers have been published about tracematches and their various applications so we were able to find tm-specific benchmarks. *FailSafeEnumThread* (68 lines) verifies that enumerations are not modified between reads; *FailSafeIter* (57 lines) does the same for iterators; and *HashMapTest* (66 lines) verifies that objects in hashmaps are not modified in ways that change their hashcodes.

In *Figure 8.5* see a little bit more slowdown in the original aspectj benchmarks, but the tm-specific benchmarks are all very fast.

Once again, all we can conclude is that MetaLexer is slower than JFlex – we cannot say by how much.

The MetaLexer implementations exhibited their greatest slowdowns on the abc bench-

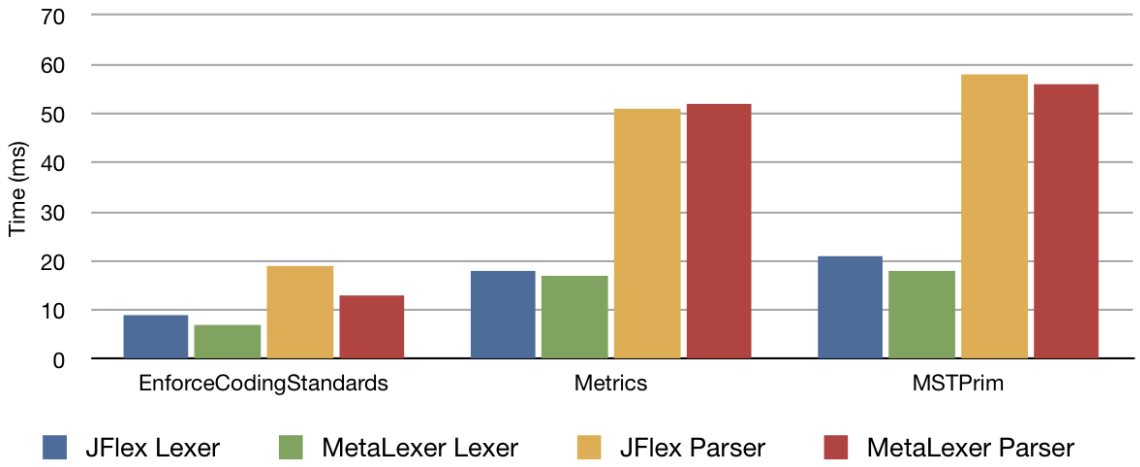


Figure 8.4 Execution Times for abc – eaj

marks. This is consistent with our findings for code size and compilation time, which suggests that the large amount of duplication (see *Section 8.2.2*) is to blame. The problem can probably be addressed by finding a way to share this code. Doing so should speed up inheritance and reduce the size of the generated code, reducing the runtime.

MetaLexer

Our MetaLexer benchmarks were easy to choose. We simply chose the largest layouts (120-310 lines) and components (60-140 lines) in the only existing real-world MetaLexer specifications – those of our six example languages.

The MetaLexer syntax is relatively simple and so, as expected, *Figures 8.6-8.7* show a relatively small slowdown (roughly 1.5 times for the component language and 1.25 times for the layout language). We also see that the lexer takes up a large percentage of the parser's total runtime because the parser proper is so simple.

8.4.5 Summary

We compared MetaLexer's performance to that of JFlex in four areas: specification size, generated lexer size, compilation time, and execution time. MetaLexer generally has

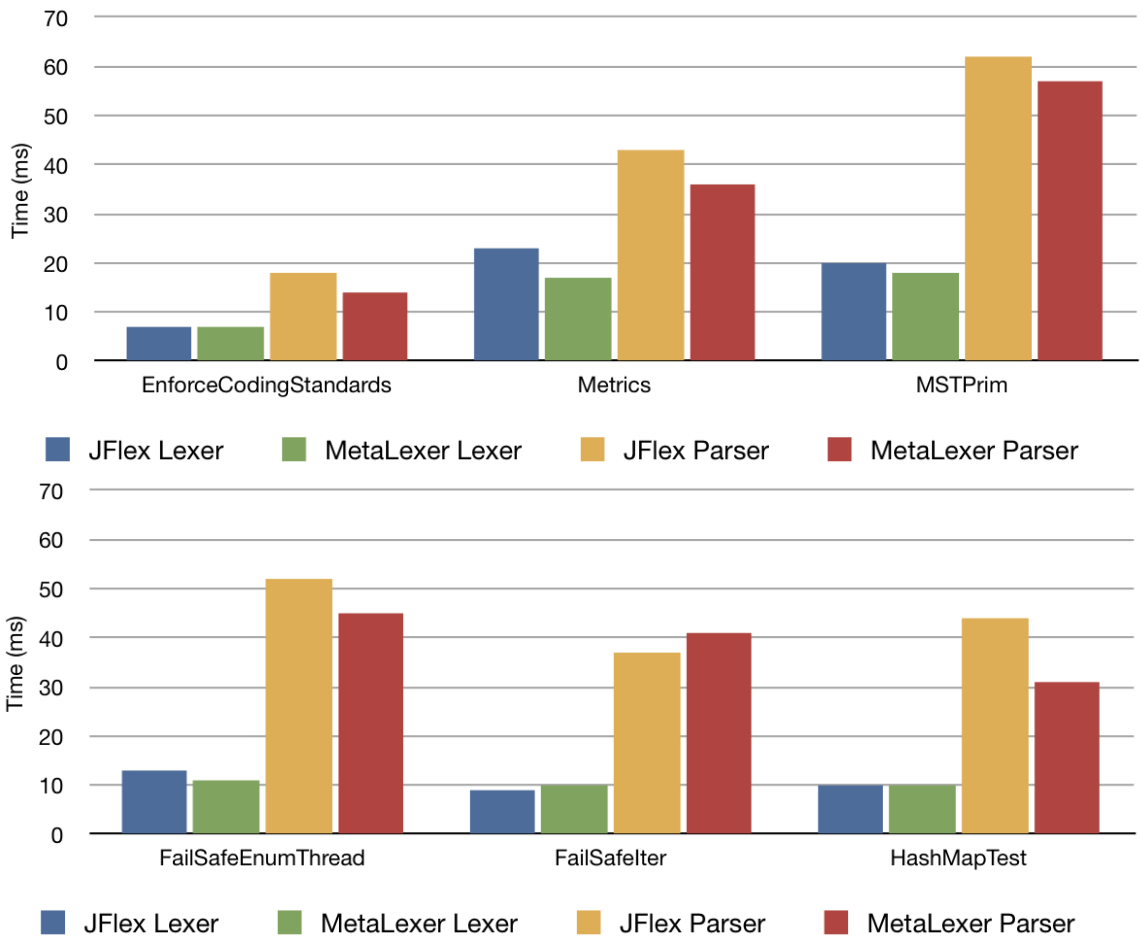


Figure 8.5 Execution Times for abc – tm

shorter and clearer specifications than JFlex and the other metrics are all within an order of magnitude. The increased clarity of the specifications makes this tradeoff worthwhile, especially since our initial implementation is untuned and unoptimized. Furthermore, new improvements frequently present themselves when lexers are rewritten in MetaLexer.

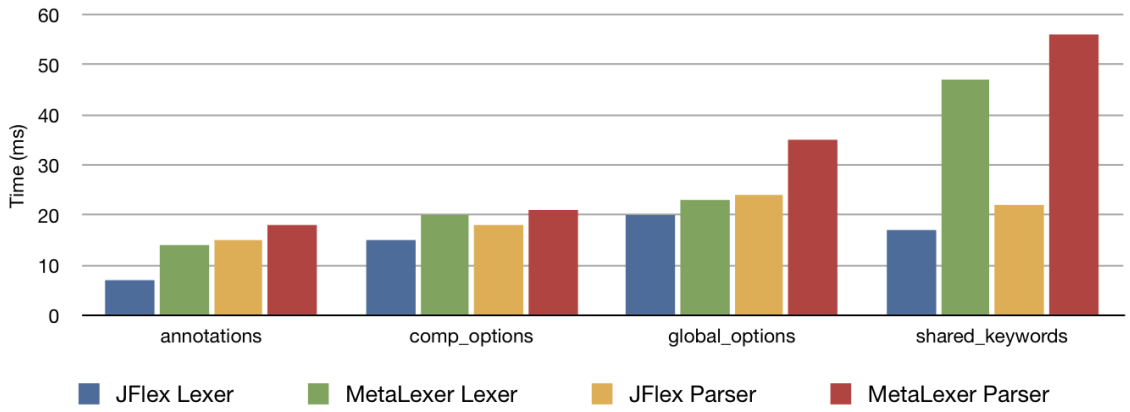


Figure 8.6 Execution Times for MetaLexer – Component

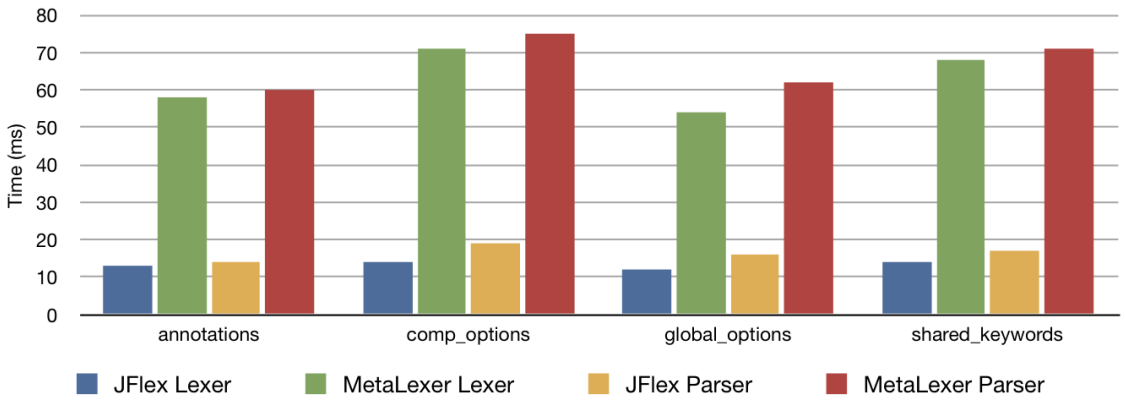


Figure 8.7 Execution Times for MetaLexer – Layout

Chapter 9

Related Work

We were not the first to explore the area of modular, extensible compilers. This chapter describes research that shows the demand for such compilers and the work that has been done to satisfy the demand.

9.1 Demand

There are many applications for modular, extensible compilers. One of the most rapidly growing is mixed language programming (MLP)[1]. In MLP, multiple programming languages are combined not only in the same program, but in the same file. This allows programmers to use the most suitable language for each programming task at a finer granularity than the program level.

During the development process, integrated development environments (IDEs) such as Eclipse[2] are invaluable tools. However, most IDEs provide assistance with only a single language. Even more advanced IDEs with plugins for multiple languages provide assistance with only a single language in each file. (Usually, there is a separate editor for each

[1] Since the field is not yet established, there is no standard terminology. Sometimes it is referred to as 'multi-language programming' or programming with 'embedded languages'.

[2] http://eclipse.org/

language and so a single language must be chosen when the file is opened.) However, Kats et al ([KKV08]) have done work to provide MLP support in the Eclipse IDE Meta-tooling Platform (IMP) [3]. Using their extended IMP, it is possible to create an MLP editor that supports syntax checking, syntax highlighting, outline view, and code folding.

Since such MLP editors are not widely available, some researchers have used libraries to simulate MLP within a single language. This approach is most commonly used in the functional programming (FP) community (e.g. Haskell [Hud96] and Lisp [EH80]). In most cases, this places much stronger limits on the new language than true MLP would.

MLP can also be applied to improve programs with modules written in different languages. For example, the Jeannie tool [HG07] created by Hirzel and Grimm simplifies programs written using the Java Native Interface (JNI). Rather than separating C and Java code into separate files and then having them call each other through an interface, Jeannie mixes both languages in every file. This makes JNI programs much easier to read and write. They accomplish this by introducing new delimiters that switch from one language to the other. When the MLP code is compiled, it is separated into separate files in the traditional JNI style.

Other calls for mixed language functionality, whether at the file level or at the program level, can be found in [Vol05] and [Bur95].

9.2 Approaches using LR Parsers

The most commonly used parser generators all accept some variation on LR grammars, usually LALR but occasionally SLR or full LR(1). As a result, these classes of grammars are familiar and well understood and there are mature tools for developing them. Naturally then, work has been done to make such grammars modular and/or extensible.

The Polyglot Parser Generator [NCM03], developed by Brukman and Myers, is an extension of the popular CUP[4] parser generator that adds extensibility. Existing grammars can

[3] http://eclipse-imp.sourceforge.net/

[4] http://www2.cs.tum.edu/projects/cup/

be extended by new grammars that add, delete, or replace their productions. The Polyglot Parser Generator is only one element of the larger Polyglot extensible compiler framework. Unfortunately, Polyglot does not address the problem of extensible lexing. Instead, each project must develop its own solution, in the worst case developing a new lexer for each extension of the parser.

Going to the next level, Ekman et al created JastAdd [EH07b], an extensible, modular attribute grammar system that can be used to build entire extensible compilers. JastAdd is mostly indifferent to how its input is parsed, as long as the parser builds up an abstract syntax tree (AST) using its generated AST classes. However, for their own project, the JastAddJ extensible Java compiler [EH07a], they created a parsing tool that compiles a new specification language (Beaver, slightly modified to improve modularity) to Beaver[5]. It moves rule type information and token declarations out of the parser header so that separate files can be merged by simple concatenation. Then each extension concatenates an appropriate subset of the parser files to form its own parser. Extensible lexing is similarly handled by concatenating lexer specification fragments (written in JFlex). Of course, concatenation is blind – no checks are performed. Furthermore, concatenation is purely constructive – deletion of (lexer or parser) rules is impossible.

The abc extensible AspectJ compiler [ACH+05], developed by Avgustinov et al, combines all of these ideas. At present, it has two front-ends, one written using Polyglot and the other using JastAdd. Both, however, use an ad-hoc extensible lexer written in JFlex. Interestingly, the behaviour of the manually written abc lexer is very similar to the behaviour of the generated JFlex produced by MetaLexer. A detailed description of the abc lexer can be found in *Section 8.2*.

9.3 Approaches using Other Classes of Parsers

Since LR(1)/SLR/LALR grammars are not composable, they are not particularly well suited to modular parsing. With this in mind, some researchers have explored approaches

[5] http://beaver.sourceforge.net/

based on other classes of grammars. All are slower, but more powerful, than LR(1)/SLR/LALR grammars.

Some approaches, having already accepted a reduction in performance, go a step further and eliminate the lexer. Obviously, once a full-scale parser is (effectively) handling the lexing of a language, MLP becomes very straightforward.

9.3.1 Antlr

The Antlr parser generator [Par07], created by Terrence Parr, aims to be a declarative way to specify the sort of recursive descent parser that one would ordinarily build by hand. It uses an extension of LL(k) parsing called LL(*). In LL(*), unless a restriction is imposed by the grammar writer, an arbitrary amount of lookahead is available when resolving ambiguity. As a result, Antlr is powerful enough to be able to specify the syntax of C++[6].

Since Antlr was created primarily as a tool for practicing compiler writers (rather than a proof-of-concept or an academic research project), it has many features that make common parsing tasks easier. It has a nice IDE for creating and debugging grammars as well as special syntax for building ASTs and performing source-to-source translations. It also supports extended Bachus-Naur form (EBNF) syntax, which alleviates much of the pain of being unable to use left-recursion.

With all of its features, lookahead, and backtracking, Antlr is decidedly slower than an LR tool like Beaver. It also generates a much larger parser class (since the code for a recursive descent parser is much larger than the binary representation of a few LR parsing tables).

9.3.2 Rats!

Another particularly interesting system is Rats!, created by Robert Grimm [Gri06]. Rats! discards context free grammars (CFGs) in favour of parsing expression grammars (PEGs). The specification for a PEG looks like a normal CFG, but it is interpreted differently. If a

[6]Since LR approaches cannot specify the syntax of C++, they specify a slightly larger language. Subsequent phases of the compiler then perform weeding and disambiguation.

non-terminal has multiple productions, then they will be tested in order until one matches. If no matching production is found, then the parser backtracks in the derivation.

Unfortunately, the frequent backtracking required by a PEG parser can easily lead to an exponential runtime (in the size of the input text). To avoid this problem, PEG parsers memoize all intermediate results (i.e. matches of non-terminals). For this reason, they are also referred to as 'packrat' parsers. With memoization, PEG parsers run in linear time but require linear additional space (both linear in the size of the input text).

Another consequence of frequent backtracking is that Rats! grammar actions with side effects must be performed in nested transactions so that they can be undone. This can be quite cumbersome, especially if the parser needs to maintain some sort of global state (e.g. a counter of some sort).

Rats! does not use a separate lexer. Instead it uses PEG specifications all the way down to the character level. As a result, it is very straightforward to lex different regions of the input according to different rules. Furthermore, Rats! allows developers to integrate their own, hand-code lexical analysis methods into the grammar. That is, characters are not the only terminals in the grammar – developers can create their own terminals using customized functions.

Rats! is implemented as a recursive descent parser (it is the recursive calls that are memoized). With its backtracking, transactions, and lack of separate lexer, Rats! tends to generate very large parser classes.

In spite of its drawbacks, Rats! is immensely powerful. It is expressive enough to specify complex languages like C and Matlab (which is notable for its command-style function calls). Furthermore, since the class of PEGs is composable, Rats! is very modular. An excellent example of this is the Jeannie tool [HG07] which combines Java and C in a single file. Using special delimiters, C code can contains blocks of Java code, which can contain blocks of C code, ad infinitum. The system was actually constructed by combining existing Rats! parsers for C and Java.

9.3.3 GLR

Generalized LR (GLR) parsing is an extension of LR parsing that accepts the full class of CFGs. Unlike a normal LR parser, a GLR parser accepts grammars with shift-reduce or reduce-reduce conflicts. It handles conflicts at runtime by branching its execution and following both paths (i.e. building both CSTs). Some GLR parsers simply return all CSTs constructed in this way. Others use heuristics or user specifications to choose the 'correct' CST before proceeding. Elkhound [7], SDF [8], and Bison[9] (which is actually an LALR parser generator with a GLR mode) are the most popular GLR parser generators.

If the lexing is done separately, GLR does not address the problem of how to handle different regions of a program differently. As a result, some have proposed using scannerless GLR (SGLR) parsers. SGLR parsers have characters, rather than tokens as their terminals. This makes it very straightforward to handle MLP.

For example, Bravenboer and Visser recommend SGLR for embedding domain-specific languages (DSLs) in general-purpose programming languages [BV04]. Along similar lines, Kats et al have used SGLR to support create rich editors for MLP in Eclipse [KKV08]. Even more relevantly, Bravenboer et al have recommended using SGLR in the abc frontend [BETV06].

Since (S)GLR grammars can capture any CFG, the class is closed under composition. As a result, it is possible to create modular (S)GLR parser generators.

GLR is slower than Rats!, which is slower than Antlr [Gri06], but work has been done on improving its performance by isolating ambiguities (e.g. [WS95]).

9.3.4 metafront

Developed by Brabrand et al, the metafront system [BSV03], serves a twofold purpose. First, it is a declarative language for transforming CSTs for one grammar into CSTs for

[7] http://www.cs.berkeley.edu/ smcpeak/elkhound/
[8] http://www.program-transformation.org/Sdf/SGLR/
[9] http://www.gnu.org/software/bison/

another grammar. Second, and more interestingly, it is an extensible parsing system.

metafront achieves extensibility by using specificity parsing, rather than context free parsing. Intuitively, specificity parsing is a scannerless parsing technique that resolves ambiguity in favour of the 'most specific' alternative. This occurs at both the lexical and syntactic levels[10].

Specificity parsing is highly unconventional and presents no clear benefits over PEG or (S)GLR parsing. It is, however, more extensible than LR(1)/SLR/LALR systems.

9.4 Approaches specific to Domain-Specific Languages

Finally, some approaches are specifically tailored to rapid development of domain-specific languages (DSLs). Systems like MontiCore [KRV07], [GKR+08] and MPS[11] (the Meta Programming System) allow developers to quickly plug together lexer, parser, and semantic modules to create new DSLs from libraries of available behaviours. While these tools are well suited to creating languages and editors for DSLs with limited syntax, they cannot be used to parse more complicated, general purpose languages. That is, they are less expressive than traditional L/PSLs and are not suitable for lexing/parsing complex languages like C or Java.

[10] Though there is no lexical pre-processor, the terminals of the grammar are still tokens formed by matching regular expressions.

[11] http://www.onboard.jetbrains.com/is1/articles/04/10/lop/

Chapter 10

Conclusions

The idea of creating compilers for languages with extensible syntax and compilers for mixed language programming (MLP) is growing in popularity. Numerous tools have sprung up for extensible and composable parsing, attribute grammars, and analyses, but still there is a gap. None of these tools provides a system for handling extensible and composable lexing.

To fill this gap, we presented the MetaLexer lexical specification language. It has three key features. First, it abstracts lexical state transitions out of semantic actions. This makes specifications clearer, easier to read, and more modular. Second, it introduces multiple inheritance. This is useful for both extension and code sharing. Third, it provides cross-platform support for a variety of programming languages and compiler toolchains.

We implemented three translators for MetaLexer. The most important translates MetaLexer specifications into JFlex specifications so that they can be realized as Java classes. The others provide help with debugging of MetaLexer specifications and porting of existing JFlex specifications.

Using these translators, we implemented lexers for three different programming systems: the Natlab language of the McLab project, the aspectj language of the abc project (with its eaj and tm extensions), and the component and layout languages of MetaLexer itself. We compared these specifications to the original JFlex implementations and found them

to be much simpler and clearer. In particular, nearly all of the supporting Java code was eliminated in favour of standard MetaLexer constructs. Furthermore, rewriting JFlex specifications in MetaLexer enabled us to see new solutions to existing lexer problems.

We compared MetaLexer's performance to that of JFlex in four areas: specification size, generated lexer size, compilation time, and execution time. MetaLexer generally has shorter specifications than JFlex and the other metrics are all within an order of magnitude. The increased clarity of the specifications makes this tradeoff worthwhile, especially since our initial implementation is untuned and unoptimized.

Chapter 11

Future Work

In its current state, MetaLexer is already a useful tool. However, there is always room for improvement. This chapter describes directions for subsequent development of MetaLexer.

11.1 Optimizations

During the initial development of MetaLexer, relatively little has been done in the way of optimization – neither in the compiler itself, nor in the generated code. Clearly, however, there is substantial opportunity to do so in the future.

11.1.1 Compilation Time

The most straightforward way to improve the execution time of the MetaLexer compiler(s) would be to make more efficient use of JastAdd attributes. In particular, many attributes need only be calculated once because they will never change. Such attributes can be flagged as *lazy* so that their values will be memoized. Even the attributes that do need to be recomputed generally only have be recomputed when the structure of the AST changes. Making these attributes lazy as well and then flushing them manually during transformations might also improve performance.

Of course, since MetaLexer performs more elaborate checks than traditional lexer generators, it can never be expected to compile specifications as quickly as they do.

11.1.2 Code Generation

At present, the JFlex specifications generated by MetaLexer are much longer than the corresponding hand-written specifications (see *Section 8.4.2*). Several improvements to the JFlex code generator are possible and could help close the gap.

First, the DFAs of the MetaLexer are stored naively. Unlike JFlex, which compresses its transition tables, MetaLexer generates arrays of integers. Binary representations of these tables would be much more compact.

Second, the DFA transition tables could be shrunk by using component-specific alphabets. Observe that, for a given components, the only symbols that can be seen by the meta-lexer are the meta-tokens declared in that component, **<BOF>**, and all possible regions.[1] By sharing the same alphabet across all components, we are adding unreachable columns to all of the transition tables.

Third, the unconditional if-statements described in *Section 7.2.3* are often unnecessary. Frequently, we can determine statically whether *Nothing* or *Just* will be returned. For example, many actions are either empty or contain only a return statement. Obviously, there is no need to consider both *Nothing* and *Just* cases in these instances. Going a step further, if we know that the state (i.e. fields) of the component will not be accessed or modified (as is frequently the case), then we can inline the action body in the generated action rather than wrapping it in a method of the component class.

Finally, the generated code contains substantial duplication. The simplest example is macros. If two components inherit the same macro (and both make use of it), then both get a copy. It would be much better to recognize that the macro has been inherited and use the same one in both cases. Similarly, it might be possible to move inherited code regions into shared superclasses. For example, if component *A* and *B* both inherit helper component *H*,

[1] We can actually go a step further if we observe that not all regions are possible – we can only see those that are guests of the current component in some embedding.

then perhaps the corresponding classes for *A* and *B* could both inherit functionality from the corresponding class for *H*. Of course, substantial thought and, perhaps, analysis is required to ensure that this is done soundly (i.e. without affecting the semantics).

11.1.3 Execution Time

If the present implementation of the JFlex backend is retained, then most execution time improvements will come from performance tuning of common cases and the elimination of layers of abstraction described above (see *Section 11.1.2*). Alternatively, there may be another way to organize the backend that results in faster lexing.

Fundamentally, the execution time is limited by the fact that a meta-token (and, potentially, a region) may be generated for each character in the input. In such cases, a MetaLexer lexer must process two (or three) times as many symbols as a comparable JFlex lexer. However, there is hope because the transition logic in the JFlex backend is handled entirely by DFAs, whereas a JFlex lexer may use a slower ad-hoc solution.

11.2 Analysis

Another area that is ripe for examination is the new analyses that become possible once lexical state transitions are abstracted out of semantic actions. If no lexical states are declared in a MetaLexer specification, then the compiler knows for certain that all transitions are specified in the top-level layout. As a result, all transitions are available to the lexer – the interaction of the various components is perfectly known. It seems likely that this information could facilitate optimizations of the generated lexer. Even if it does not, it makes possible a variety of verification and visualization tools.

11.3 Known Issues

Unfortunately, the present implementation of MetaLexer is not without blemish. A few issues remain.

11.3.1 Frontend

Some issues affect the frontend, and thus affect all backends.

First, for convenience and to eliminate duplication, error messages are sorted. They are arranged by file, by position, and then by message. Unfortunately, this means that if two messages occur at the same position in the same file – perhaps because they are related – then they may be reordered. A better solution might be to order only by file and by position and eliminate duplicates in some other way.

Second, the append buffer (see *Section 4.9*) is unavailable to rules generating error messages. Keeping the buffer hidden was a design decision intended to prevent developers from using it to affect control flow. However, it may ultimately prove worthwhile to expose it.

Third, if a lexical state is declared at the component level, then there is no way to refer to it at the layout level[2]. In general, this is a good thing because it encourages encapsulation. However, it does make it harder to port some older JFlex (or Flex) specifications that refer to specific lexical states in helper methods.

11.3.2 JFlex Backend

Other issues affect the JFlex backend. They may or may not affect other LSL backends, depending on the features of the LSL and the underlying AIL.

First, JFlex uses Java as an AIL and Java does not allow non-static inner classes to contain static members or fields. As a result, AIL code regions in components (which are wrapped

[2]Unless one cheats and makes assumptions about the name mangling.

in inner classes) may not contain static members or functions. While not a major limitation, this is quite frustrating. Any backend with Java as an AIL using the same implementation pattern will encounter this problem.

Second, when tracing code is embedded in the generated lexer it is enabled by a static method, *setTracingEnabled()*. It would be much nicer to pass this as a flag to the constructor. Unfortunately, only the very latest version of JFlex supports adding arguments to the lexer's constructor and we felt that this would limit its usefulness. Since some tracing occurs in the constructor itself, the flag must be set before the constructor is executed, hence the static method.

Third, the pair filter does not interact nicely with start meta-patterns. It is frequently the case that a start meta-pattern contains a meta-token that is intended to be paired with a meta-token in the end meta-pattern. For example, a Java class component might have an open brace in its start meta-pattern and a close brace in its end meta-pattern. Obviously, these braces are intended to be paired. Unfortunately, the start meta-pattern occurs in the host component and the end meta-pattern occurs in the guest meta-pattern, making it impossible to pair them. To resolve this issue, open-items contained in the start meta-pattern are cleaned out of the pair filter. This works well in practice, but it is not very nice in principle. In particular, close-items in the start meta-pattern are not cleaned out of the pair filter because there is no good way to restore the open-items they have already cancelled. As a result, the pair filter recognizes close-items but not open-items in start meta-patterns. This is not very intuitive.

11.4 Qualified Names

At present, qualified names in MetaLexer are not particularly useful. Instead of qualifying names with a dot, one could just as easily put everything in one directory and group files be prepending prefixes. Qualified names would be much more useful if there were circumstances in which names could be used without qualification – perhaps within the same directory or when explicitly imported, as in Java. Alternatively, an aliasing mechanism

could be introduced to allow components to be referred to by shorter names.

11.5 Other Platforms

As discussed in *Section 7.2*, it should be quite straightforward to implement additional code generation engines for MetaLexer. Two, in particular, would be especially helpful.

First, a JLex backend would serve as a useful starting point. Though it is less powerful and modern than JFlex, it has the advantage of being released under a modified-BSD license. This means that it could be freely distributed with MetaLexer, making the project more self-contained.

Second, a Flex backend would be useful because it would give C and C++ programmers access to the power of MetaLexer. Some of the implementation details would not translate directly – inner classes differ slightly between Java and C/C++ – but there are no fundamental obstacles.

11.6 JFlex Porting

As discussed in *Section 5.3*, the JFlex-to-MetaLexer translator exists primarily to demonstrate that MetaLexer is as powerful as JFlex. Unfortunately, it is not very useful as a tool. A more practical tool would disregard the validity of its output and focus instead on providing useful stubs for the developer porting from JFlex to MetaLexer. Basically, it would perform the grunt work of splitting the specification up into smaller files and replacing all action delimiters (i.e. '{ }' to '{: :}').

Given a JFlex specification, the tool would create a layout with the same name containing all AIL helper code and imports for the components described next. It would create a helper component containing all macros and a non-helper component for each lexical state, inheriting the macro component and containing all the rules of the lexical state. It would be up to the developer to port the transition logic to MetaLexer. While the output would not

be remotely complete, it would serve as a much more useful starting point than the valid MetaLexer produced by the existing translator.

11.7 Comparison with Lexerless Techniques

Lexerless parsing is another good way to handle MLP and other tasks that require extensible, modular compiler frontends. However, techniques like PEG and SGLR parsing are slower than traditional LALR parsing and frequently their full power is not needed. For example, it seems likely that MetaLexer and LALR could have been used to generate MLP editors for Eclipse ([KKV08]) or to parse Jeannie ([HG07]).

We built MetaLexer because we believe that LALR is frequently 'good enough' and that, when it is, the performance benefit of using it is substantial. It would interesting to compare these approaches directly. The work of Bravenboer et al, expressing the syntax of abc in SGLR [BETV06], presents an excellent opportunity for comparison. Though both approaches are relatively new and unoptimized, it would be interesting to see how they compare. Furthermore, additional work will be required to determine how often the combination of MetaLexer and LALR is 'good enough'.

11.8 Parser Specification Language

When we began, our goal was to create not an LSL but a PSL. Having been dissatisfied with a number of such tools, we decided to create a composable PSL. However, we realized that before we could begin we would need a composable LSL. Thus was born MetaLexer. However, our goal remains. We record below the fruits of our initial research in the hope that they may be useful to a future implementer.

The chief problem when creating a composable PSL is that the classes of grammars traditionally used for parsing (i.e. LR(1)/SLR/LALR) are not composable. In particular, any new productions added during composition have a chance of conflicting with existing productions (either shift-reduce or reduce-reduce). Consequently, it is necessary to use another

class of grammar. Antlr[3] uses LL(*) grammars, which extend LL(k) grammars with infinite lookahead. SGLR systems like SDF[4] use full context-free grammars, which are closed under composition. Rats! [Gri06] uses PEGs, which eliminate ambiguity by testing rules in order and backtracking upon failure.

All of these classes of grammars are composable, but all are slower than LR(1)/SLR/LALR grammars. Furthermore, they are less established and standardized, so their are fewer tools designed to work with them. Without tables, Antlr generates an enormous amount of decision-making code. It also lacks left-recursion and produces slower parsers than Beaver. SGLR is slower than LR(1)/SLR/LALR and does not work nicely with existing tools (e.g. [KKV08]). Rats! has to be able to backtrack, so actions cannot have side-effects. It also produces slower parsers than Antlr.

The Polyglot Parser Generator [NCM03] adds extensibility to the popular LALR CUP[5] parser but it is not composable.

We propose returning to a LR(1)/SLR/LALR approach, but restricting composition to subgrammars with different alphabets. If two subgrammars share no tokens in common, then they can never conflict with each other[6]. Of course, MetaLexer is ideal for specifying the lexer for each subgrammar separately.

At a high-level, our composable PSL would have many features in common with MetaLexer. First, it would allow rules to be added, removed, and replaced. This would make specifications extensible. Second, it would serve primarily as a preprocessor, compiling high-level specifications down to the syntax of existing PSLs such as beaver and bison. In this way, it could provide a standard feature set across different platforms. As a preprocessor, it could provide syntactic sugar for full EBNF syntax and left-recursion, even if the underlying PSL lacked support.

As an additional nicety, our composable PSL would probably separate the enumeration of tokens from the parser proper to eliminate the dependence of the lexer on the parser.

[3] http://www.antlr.org/
[4] http://www.program-transformation.org/Sdf/SGLR/
[5] http://www2.cs.tum.edu/projects/cup/
[6] Special handling may be required for nullable rules.

Appendix A

Acronyms

ε-NFA: Epsilon Non-deterministic Finite Automaton – a special NFA in which some transitions (i.e. ε-transitions) can be made without consuming input.

abc: AspectBench Compiler – an open source implementation of the AspectJ programming language.

AIL: Action Implementation Language – the language in which lexer actions are specified. For example, JFlex uses Java as its AIL.

AST: Abstract Syntax Tree – a refined and simplified CST.

CFG: Context Free Grammar – a succinct way of describing a context free language.

CST: Concrete Syntax Tree – the raw parse tree constructed by a parser.

DFA: Deterministic Finite Automaton – an FSM in which for a given state and input, there is precisely one next state.

DSL: Domain Specific Language – a programming language that is tailored to a specific field of inquiry.

eaj: Extended AspectJ – an extension of the AspectJ language created using abc. Includes several new pointcuts.

EBNF: Extended Bachus-Naur Form – a canonical syntax for specifying context free grammars.

FSM: Finite State Machine – an automaton consisting of a finite number of states connected by transition edges.

GPL: General Public License – an open-source copyleft license from the GNU foundation.

GLR: Generalized LR – an extension of LR parsing that handles shift-reduce and reduce-reduce conflicts by building both possible CSTs.

IDE: Integrated Development Environment – a feature-rich application for developing software.

IMP: IDE Meta-tooling Platform – a platform for developing rich-editors for the Eclipse IDE.

JNI: Java Native Interface – Java's foreign function interface (with C).

LSL: Lexer Specification Language – a language in which lexer specifications are written. For example, the JFlex system calls its LSL 'JFlex'. Note the distinction between the language for specifying lexers and the system for compiling specifications into lexers.

MLP: Mixed Language Programming – when multiple programming languages are mixed within a single source file.

NFA: Non-deterministic Finite Automaton – an FSM in which for a given state and input, there are zero or more next states.

PEG: Parsing Expression Grammar – a CFG-like grammar in which productions are tested in order.

PSL: Parser Specification Language – a language in which parser specifications are written. For example, the Beaver parser generator calls its PSL 'Beaver'. Note the distinction between the language for specifying parsers and the system for compiling specifications into parsers.

SGLR: Scannerless GLR – an extension of GLR that handles lexing within the parser.

tm: Tracematches – an extension of the AspectJ language (actually, of the eaj extension of that language) that extends pointcuts with temporal logic constructs.

Appendix B

Developer Manual

This appendix describes the organization of the MetaLexer project. In order to postpone the inevitable obsolescence of this appendix, we focus on highlights rather than providing an exhaustive listing.

B.1 Organization

The files of the MetaLexer project are organized as follows.

B.1.1 metalexer/

.classpath & .project & .settings/ Eclipse project files.

common.properties & .xml The Ant build file for the frontend. Shared by all backends.

metalexer.properties & .xml The Ant build file for the metalexer backend.

jflex.properties & .xml The Ant build file for the jflex backend.

bin/ The directory containing the Java class files produced the build process (either Eclipse or CLI). If jars are produced, then they will be stored elsewhere.

gen/ The directory containing source files produced by the build process (scanners produced by JFlex, parsers produced by Beaver, AST node classes, some JUnit test classes, etc).

permgen/ The directory containing files that will be copied to **gen** if for some reason generation is impossible (e.g. if JFlex is missing).

lib/ The directory containing external jars depended upon by the build process. Note that these jars are not required by the produced jars.

run_targets/ Useful eclipse run targets (mostly for running tests, but also for the frontend).

src/ Handwritten (vs generated) source code (JFlex, Beaver, JastAdd, Java, etc), not related to testing. Must not depend on anything in **test**.

test/ Handwritten (vs generated) testing source code. Not included in jars.

... Anything else is unofficial and is not required for building MetaLexer. It is not considered to be part of the system and may not be depended upon.

B.1.2 metalexer/src/ & metalexer/test/

frontend Files related to the frontend or shared by all backends.

backend-metalexer Files related to the MetaLexer backend (i.e. for the MetaLexer-to-MetaLexer translator).

backend-jflex Files related to the JFlex backend (i.e. for the MetaLexer-to-JFlex translator).

B.1.3 metalexer/src/frontend/

metalexer Java package.

lexer Directory containing the specification of the lexers.

component.grammar & .ast Specification of the component language parser.

layout.grammar & .ast Specification of the layout language parser.

Component & LayoutInherit.jadd The specifications of processInheritance for components and layouts, respectively. These are the methods that trigger processing of the raw AST from the parser.

Component & LayoutErrors.jrag The specifications of getErrors for components and layouts, respectively. These files delineate the possible (frontend) errors that can be raised. (Note that some errors are raised in the parser, the file loader, etc).

Component & LayoutWarnings.jrag The specifications of getWarnings for components and layouts, respectively. These files delineate the possible (frontend) errors that can be raised. (Note that some errors are raised in the parser, the inheritance mechanism, etc).

... The other files provide support methods for processInheritance and getErrors.

B.1.4 metalexer/src/frontend/metalexer

CompilationProblem & Error & Warning CompilationProblem is the base type of all errors and warning encountered during compilation.

Constants Some helpful constants.

FileLoader Handles loading of components and layouts from files.

PatternType An enumeration of possible pattern types (i.e. acyclic, cyclic, cleanup).

B.1.5 metalexer/src/frontend/lexer

component Components and layouts specific to the component lexer.

layout Components and layouts specific to the layout lexer.

shared Components and layouts shared by the component and layout lexers.

B.1.6 metalexer/src/backend-metalexer/

Component & LayoutGeneration.jadd The printers that turn a processed AST into MetaLexer files.

metalexer/metalexer/ Java package containing **ML2ML**, the entry point for the MetaLexer-to-MetaLexer translator.

B.1.7 metalexer/src/backend-jflex/

Component & LayoutGeneration.jadd The printers that turn a processed AST into JFlex files.

_Generation.jadd Hierarchically called from **Component & LayoutGeneration.jadd**. Generate the .flex file output by the translator (i.e. the actual lexer).

_DFAGeneration.jadd Hierarchically called from **LayoutGeneration.jadd** via **LayoutDFAGeneration**. Generate the meta-lexer class for the specification, most notably the automata used to match meta-patterns.

JFlexErrors.jrag The specification of JFlex-specific component and layout errors. Contributes to the same collections as **Component & LayoutErrors.jrag**.

ReturnWrap.flex Custom lexer for translating return statements of the form `return X;` into statements of the form `return Maybe.Just(X);`. This is handled with a lexer so that comments and strings can be handled properly.

PackageFind.flex Custom lexer for finding package statements of the form `package X;` and returning `X`. This is handled with a lexer so that comments and strings can be handled properly.

StaticFind.flex Custom lexer for finding all occurrences of the keyword static. This is handled with a lexer so that comments and strings can be handled properly.

metalexer/jflex/ Java package containing **ML2JFlex**, the entry point for the MetaLexer-to-JFlex translator.

metalexer/jflex/fsm/ Java package containing the classes for constructing and manipulating finite state machines, namely ε-NFAs, NFAs, and DFAs.

... Files that support generation and meta-generation.

B.1.8 metalexer/test/frontend/

c_scanner.testlist List of component language scanner tests (input in .in, expected output in .out). Output files contain a list of tokens up to either EOF or the first error. (No warnings are possible.)

c_parserpass.testlist List of component language parser tests (input in .in, expected output in .out). Output files contain pretty printed versions of input files. (Warnings are ignored.)

c_parserfail.testlist List of component language parser tests (input in .in, expected output in .out). Output files contain lists of errors. (Warnings are ignored.)

c_inheritancepass.testlist List of component language inheritance tests (input in .mlc, expected output in .out). Output files contain collapsed (i.e. post-inheritance), pretty printed versions of input files. (Warnings are ignored.)

c_inheritancefail.testlist List of component language inheritance tests (input in .mlc, expected output in .out). Output files contain lists of errors. (Warnings are ignored.)

c_error.testlist List of component language error tests (input in .mlc, expected output in .out). Output files contain lists of errors. (Warnings are ignored.)

c_warning.testlist List of component language warning tests (input in .mlc, expected output in .out). Output files contain lists of warnings (errors are ignored).

l_scanner.testlist List of layout language scanner tests (input in .in, expected output in .out). Output files contain a list of tokens up to either EOF or the first error.

l_parserpass.testlist List of layout language parser tests (input in .in, expected output in .out). Output files contain pretty printed versions of input files. (Warnings are ignored.)

l_parserfail.testlist List of layout language parser tests (input in .in, expected output in .out). Output files contain lists of errors. (Warnings are ignored.)

l_inheritancepass.testlist List of layout language inheritance tests (input in .mll, expected output in .out). Output files contain collapsed (i.e. post-inheritance), pretty printed versions of input files. (Warnings are ignored.)

l_inheritancefail.testlist List of layout language inheritance tests (input in .mll, expected output in .out). Output files contain lists of errors. (Warnings are ignored.)

l_error.testlist List of layout language error tests (input in .mll, expected output in .out). Output files contain lists of errors. Note: l_error_helper_ tests check the effect of the %helper directive on errors. (Warnings are ignored.)

l_warning.testlist List of layout language error tests (input in .mll, expected output in .out). Output files contain lists of warnings. (Errors are ignored.)

metalexer/ Java package. Contains JastAdd files (in contrast to **src**) to keep them separate from the test input and output files.

... Test specifications (as listed in .testlist files).

B.1.9 metalexer/test/frontend/metalexer/

Component & LayoutPrint.jrag Pretty printers for testing. Produce an output that is useful for testing. Not guaranteed to produce correct MetaLexer (e.g. may include helpful, but illegal annotations).

FrontendTests The top-level JUnit test suite.

_**TestBase** Class to be extended by generated test file.

_**TestGenerator** Class that generates a test file from a .testlist file.

_**TestTool** Class for generating .out files automatically.

... Support files for the Base, Generator, and Tool classes.

B.1.10 metalexer/test/backend-metalexer/

metalexer/metalexer/ Java package containing the top-level JUnit test suite, **BackendMetalexerTests**.

in/ Metalexer specifications.

out1/ Files from **in** that have been run through the pretty printer.

out2/ Files from **out1** that have been run through the pretty printer. Contents should be identical to those of **out1**.

B.1.11 metalexer/test/backend-jflex/

c_error.testlist List of component language (JFlex-specific) error tests (input in .mll, expected output in .out). Output files contain lists of errors.

c_error.testlist List of layout language (JFlex-specific) error tests (input in .mll, expected output in .out). Output files contain lists of errors.

m_c_meta.testlist List of tests. Each test has a .in file list of meta-tokens and regions to be passed to the generated meta-lexer for the component language specification and a .out file listing the expected embedding transitions returned by the meta-lexer.

n_scanner.testlist List of natlab language scanner tests. Copied from McLab but modified by removing lines passed through the comment buffer (i.e. preceded by '#' in the original).

in/ Specifications of various languages given in MetaLexer.

out/ The output from translating the specifications in **in/** to JFlex: lexer and meta-lexer JFlex files, properties files recording the characters assigned to regions and meta-tokens and the numbers assigned to embeddings, and Java scanners output by JFlex.

out/bin/ The class files that result from compiling the Java scanners output by JFlex. Note: this directory is on the classpath so that these scanners can be tested by the JUnit test suite.

placeholders/ Stubs of Java classes that contain just enough to make the generated scanners work. Most importantly, the parser stubs contain the list of tokens on which the scanners

depend. Note: subdirectories are all Java packages and are on the classpath.

metalexer/jflex/ Java package. Contains JastAdd files to keep them separate from test intput files.

B.1.12 metalexer/test/backend-jflex/metalexer/jflex/

BackendJFlexTests The top-level JUnit test suite.

CompilationTests Arguably the most important test cases. Compiles the MetaLexer specifications in **in** to JFlex, to Java, to class files. Depended upon by the tests that exercise the generated scanners. Note: when running in Eclipse, it is necessary to run the tests more than once, refreshing in between, because Eclipse will not pick up the changes made by this test during a single run.

_**Tests** Handwritten (vs generated) test cases.

_**TestBase** Class to be extended by generated test file.

_**TestGenerator** Class that generates a test file from a .testlist file.

_**TestTool** Class for generating .out files automatically.

JFlexHelper A helper class that loads the JFlex jar at runtime if it is present.

ReflectionHelper A helper class that loads the generated scanner files at runtime if they have been generated (i.e. if JFlex is available).

... Support files for the Base, Generator, and Tool classes.

B.2 JFlex

Unfortunately, JFlex is covered by a GPL license. Since the MetaLexer is covered by a modified BSD-style license, the MetaLexer distribution cannot include JFlex.

MetaLexer will function properly without JFlex, but you will be unable to rebuild any modified .flex files and the compilation-based tests in the JFlex backend will be unavailable.

If you wish to download JFlex on your own, it is available at *http://jflex.de/*. Just unzip the archive under the lib/ directory and point `jflex.jar.path.prop` in *common.properties* at the jar file (path should be relative to lib/).

B.3 Configurations

The MetaLexer compiler supports multiple backends with a shared frontend. The current implementation supports MetaLexer-to-JFlex and MetaLexer-to-MetaLexer.

Unfortunately, since the project uses aspects, there is interference between the backends. As a result, only one can be built and developed at a time. One benefit of this approach is that it keeps the release jars small by excluding code required for other backends.

To work solely on the frontend, use the **common.xml** build file. To work on the MetaLexer backend and the frontend, use the **metalexer.xml** build file. To work on the JFlex backend and the frontend, use the **jflex.xml** build file.

B.4 Building MetaLexer

MetaLexer developers can use either the command line or the Eclipse IDE.

B.4.1 Command Line

When executing the build files from the command line, ensure that the *eclipse.running* property is not set. If it is, then the Java compilation steps will be skipped.

The Ant build file takes care of all classpath issues. No configuration should be required.

B.4.2 Eclipse

To build the project in Eclipse, use the same Ant targets as from the command line, but ensure that the *eclipse.running* property is set. This will allow Eclipse to build Java files that are in source path folders in the Eclipse build path.

The provided **.classpath** file takes care of all source- and classpath issues. No configuration should be required.

When running the JUnit tests in the JUnit view, note that the frontend and backend tests must be run separately. (In contrast, the *test* Ant target runs all appropriate tests in a single pass.)

If you want a really minimal release jar, you have to build it from the command line because otherwise Eclipse will compile some extraneous Java files and they will be included.

Appendix C

Language Specification

```
The MetaLexer tool is covered by the (modified) BSD License.

Copyright (c) 2009, Andrew Casey (McGill University)
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimer in the documentation
      and/or other materials provided with the distribution.
    * Neither the name of McGill University nor the names of its contributors
      may be used to endorse or promote products derived from this software
      without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.1 Component

```
1  package metalexer;
2  %%
3  //based on http://jflex.de/manual.html
4
5  import static metalexer.ComponentParser.Terminals.*;
6  %%
7
8  %layout component
9
10 %option visibility "%public"
11 %option finality "%final"
12 %option class_name "%class ComponentScanner"
13
14 %component action
15 %component append_delim_region
16 %component append_region
17 %component base
18 %component bracket_comment
19 %component char_class
20 %component comp_options
21 %component comp_rules
22 %component decl_region
23 %component delete_directive
24 %component init_region
25 %component macro_defn
26 %component macro_invoc
27 %component mtok_decl
28 %component open_rule_group
29 %component repetition_spec
30 %component state_list
31 %component string_id_directive
32 %component string
33
34 %start base
```

```
35
36 %%
37
38 %%inherit beaver
39 %%inherit helper
40
41 %%embed
42 %name header
43 %host base
44 %guest comp_options
45 %start <BOF>
46 %end END_HEADER_SECTION
47
48 %%embed
49 %name rules
50 %host base
51 %guest comp_rules
52 %start %comp_options%
53 %end END_RULE_SECTION
54
55 %%embed
56 %name action
57 %host comp_rules
58 %guest action
59 %start START_ACTION
60 %end END_ACTION
61
62 %%embed
63 %name init_region
64 %host comp_options
65 %guest init_region
66 %start START_INIT_REGION
67 %end END_INIT_REGION
68
69 %%embed
70 %name decl_region
71 %host comp_options
```

```
%guest decl_region
%start START_DECL_REGION
%end END_DECL_REGION

%%embed
%name append_region
%host comp_options
%guest append_region
%start START_APPEND_REGION
%end END_APPEND_REGION

%%embed
%name append_delim_region
%host comp_options
%guest append_delim_region
%start START_APPEND_DELIM_REGION
%end END_APPEND_DELIM_REGION

%%embed
%name macro_defn
%host comp_options
%guest macro_defn
%start START_MACRO_DEFN
%end END_MACRO_DEFN

%%embed
%name bracket_comment
%host comp_options, macro_defn, comp_rules, mtok_decl, state_list,
    string_id_directive, delete_directive, open_rule_group
%host bracket_comment //NB: host and guest => nestable
%guest bracket_comment
%start START_BRACKET_COMMENT
%end END_BRACKET_COMMENT

%%embed
%name char_class
%host macro_defn, comp_rules, delete_directive
```

```
%guest char_class
%start START_CHAR_CLASS
%end END_CHAR_CLASS

%%embed
%name delete_directive
%host comp_rules
%guest delete_directive
%start START_DELETE_DIRECTIVE
%end END_DELETE_DIRECTIVE

%%embed
%name macro_invoc
%host macro_defn, comp_rules, delete_directive
%guest macro_invoc
%start START_MACRO_INVOC
%end END_MACRO_INVOC

%%embed
%name repetition_spec
%host macro_defn, comp_rules, delete_directive
%guest repetition_spec
%start START_REP_SPEC
%end END_REP_SPEC

%%embed
%name mtok_decl
%host comp_rules
%guest mtok_decl
%start %action%
%end END_MTOK_DECL

%%embed
%name string_id_directive
%host comp_options, comp_rules
%guest string_id_directive
%start START_STRING_ID_DIRECTIVE
```

```
145 %end END_STRING_ID_DIRECTIVE
146
147 %%embed
148 %name string
149 %host macro_defn, comp_rules, delete_directive, string_id_directive
150 %guest string
151 %start START_STRING
152 %end END_STRING
153
154 %%embed
155 %name state_list
156 %host comp_rules, delete_directive
157 %guest state_list
158 %start START_STATE_LIST
159 %end END_STATE_LIST
160
161 %%embed
162 %name delete_state_list
163 %host delete_directive
164 %guest state_list
165 %start START_STATE_LIST
166 %end END_STATE_LIST
167
168 %%embed
169 %name open_rule_group
170 %host comp_rules
171 %guest open_rule_group
172 %start %state_list%
173 %end END_OPEN_RULE_GROUP
174
175 %%embed
176 %name eof_error
177 %host action, append_delim_region, append_region, bracket_comment,
       char_class
178 %host decl_region, init_region, macro_invoc, open_rule_group,
       repetition_spec
179 %host state_list, string
```

```
%guest base
%start EOF_ERROR
%end <ANY> //NB: will never happen
```

lexer/component/component.mll

```
%component action

%extern "private Symbol symbol(short type, Object value, int startLine,
    int startCol, int endLine, int endCol)"
%extern "private void error(String msg) throws Scanner.Exception"

%appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
    endCol, String text)*/
    return symbol(ACTION, text, startLine + 1, startCol + 1, endLine +
        1, endCol + 1);
%appendWithStartDelim}

%%

%%inherit comp_macros

%{CloseAction} {: append(yytext().substring(1)); :}
{CloseAction} {: :} END_ACTION

%:
%:

<<ANY>> {: append(yytext()); :}
<<EOF>> {: error("Unterminated action"); :} EOF_ERROR
```

lexer/component/action.mlc

```
%component append_delim_region

%extern "private Symbol symbol(short type, Object value, int startLine,
    int startCol, int endLine, int endCol)"
%extern "private void error(String msg) throws Scanner.Exception"
```

```
5
6  %appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
       endCol, String text)*/
7      return symbol(APPEND_WITH_START_DELIM_REGION, text, startLine + 1,
           startCol + 1, endLine + 1, endCol + 1);
8  %appendWithStartDelim}
9
10 CloseAppendDelimRegion = "%appendWithStartDelim}"
11
12 %%
13
14 %{CloseAppendDelimRegion} {: append(yytext().substring(1)); :}
15 {CloseAppendDelimRegion} {: :} END_APPEND_DELIM_REGION
16
17 %:
18 %:
19
20 <<ANY>> {: append(yytext()); :}
21 <<EOF>> {: error("Unterminated append region"); :} EOF_ERROR
```

lexer/component/append_delim_region.mlc

```
1  %component append_region
2
3  %extern "private Symbol symbol(short type, Object value, int startLine,
       int startCol, int endLine, int endCol)"
4  %extern "private void error(String msg) throws Scanner.Exception"
5
6  %appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
       endCol, String text)*/
7      return symbol(APPEND_REGION, text, startLine + 1, startCol + 1,
           endLine + 1, endCol + 1);
8  %appendWithStartDelim}
9
10 CloseAppendRegion = "%append}"
11
12 %%
13
```

```
14 %{CloseAppendRegion} {: append(yytext().substring(1)); :}
15 {CloseAppendRegion} {: :} END_APPEND_REGION
16
17 %:
18 %:
19
20 <<ANY>> {: append(yytext()); :}
21 <<EOF>> {: error("Unterminated append region"); :} EOF_ERROR
```

lexer/component/append_region.mlc

```
1  %component char_class
2
3  %extern "private Symbol symbol(short type)"
4  %extern "private Symbol symbol(short type, Object value)"
5  %extern "private void error(String msg) throws Scanner.Exception"
6
7  %%
8
9  %%inherit comp_macros
10
11 \^ {: return symbol(CHAR_CLASS_NEGATE); :}
12 \- {: return symbol(DASH); :}
13 \] {: return symbol(RSQUARE); :} END_CHAR_CLASS
14
15 {EscapeSequence} {: return symbol(ESCAPE_SEQUENCE, yytext()); :}
16 \\{Any} {: return symbol(CHAR_CLASS_CHAR, yytext().substring(1)); :}
17 \\ {: error("Incomplete escape sequence"); :}
18
19 %:
20 %:
21
22 <<ANY>> {: return symbol(CHAR_CLASS_CHAR, yytext()); :}
23 <<EOF>> {: error("Unterminated character class"); :} EOF_ERROR
```

lexer/component/char_class.mlc

```
1 %component comp_macros
```

```
2  %helper
3
4  OpenAppendRegion = "%append{"
5  OpenAppendWithStartDelimRegion = "%appendWithStartDelim{"
6
7  OpenAction = "{:"
8  CloseAction = ":}"
9
10 OpenCurlyBracket = \{
11 CloseCurlyBracket = \}
12
13 OpenAngleBracket = \<
14 CloseAngleBracket = \>
15
16 NonMeta = [^\|\(\)\{\}\[\]\<\>\\\.\*\+\?\^\$\/\.\"\~\!\-]
17
18 //ok to not handle comments - JFlex doesn't either
19 MacroLookahead = {OtherWhiteSpace}* {Identifier} {OtherWhiteSpace}*
       {CloseCurlyBracket}
20 RepetitionLookahead = {OtherWhiteSpace}* {Number} ({OtherWhiteSpace}*
       "," {OtherWhiteSpace}* {Number})? {OtherWhiteSpace}*
       {CloseCurlyBracket}
21
22 GroupSeparator = "%:"
23
24 %%
25
26 %%inherit shared_macros
```

lexer/component/comp_macros.mlc

```
1  %component comp_options
2
3  %extern "private Symbol symbol(short type)"
4  %extern "private Symbol symbol(short type, Object value)"
5  %extern "private void error(String msg) throws Scanner.Exception"
6
7  %%
```

```
8
9  %%inherit comp_macros
10 %%inherit comment_start
11
12 //whitespace
13 {LineTerminator} {: /* ignore */ :}
14 {OtherWhiteSpace} {: /* ignore */ :}
15
16 {OpenDeclRegion} {: appendToStartDelim(""); /*tweak start pos*/ :}
       START_DECL_REGION
17 {OpenInitRegion} {: appendToStartDelim(""); /*tweak start pos*/ :}
       START_INIT_REGION
18 {OpenAppendRegion} {: appendToStartDelim(""); /*tweak start pos*/ :}
       START_APPEND_REGION
19 {OpenAppendWithStartDelimRegion} {: appendToStartDelim(""); /*tweak
       start pos*/ :} START_APPEND_DELIM_REGION
20
21 //no-arg directives
22 "%helper" / {DirectiveLookahead} {:
23    return symbol(HELPER_DIRECTIVE);
24 :} START_STRING_ID_DIRECTIVE
25
26 //identifier directives
27 "%component" / {DirectiveLookahead} {:
28    return symbol(COMPONENT_DIRECTIVE);
29 :} START_STRING_ID_DIRECTIVE
30 "%state" / {DirectiveLookahead} {:
31    return symbol(STATE_DIRECTIVE);
32 :} START_STRING_ID_DIRECTIVE
33 "%xstate" / {DirectiveLookahead} {:
34    return symbol(XSTATE_DIRECTIVE);
35 :} START_STRING_ID_DIRECTIVE
36 "%start" / {DirectiveLookahead} {:
37    return symbol(START_DIRECTIVE);
38 :} START_STRING_ID_DIRECTIVE
39
40 //string directives
```

```
41 "%extern" / {DirectiveLookahead} {:
42    return symbol(EXTERN_DIRECTIVE);
43 :} START_STRING_ID_DIRECTIVE
44 "%import" / {DirectiveLookahead} {:
45    return symbol(IMPORT_DIRECTIVE);
46 :} START_STRING_ID_DIRECTIVE
47 "%initthrow" / {DirectiveLookahead} {:
48    return symbol(INITTHROW_DIRECTIVE);
49 :} START_STRING_ID_DIRECTIVE
50 "%lexthrow" / {DirectiveLookahead} {:
51    return symbol(LEXTHROW_DIRECTIVE);
52 :} START_STRING_ID_DIRECTIVE
53
54 //invalid directives
55 "%" {: error("Invalid directive"); :}
56
57 //end of section
58 {SectionSeparator} {:
59    return symbol(SECTION_SEPARATOR);
60 :} END_HEADER_SECTION
61
62 %:
63
64 //for macro declarations
65 //NB: beginning of line so that it doesn't interfere with patterns
66 {Identifier} {: return symbol(IDENTIFIER, yytext().trim()); :}
      START_MACRO_DEFN
67
68 %:
69 <<ANY>> {: error("Unexpected character: " + yytext()); :}
70 <<EOF>> {: :} END_HEADER_SECTION
```

lexer/component/comp_options.mlc

```
1 %component comp_rules
2
3 %extern "private Symbol symbol(short type)"
4
```

```
%%

%%inherit comp_macros
%%inherit comment_start

//whitespace
{LineTerminator} {: /* ignore */ :}
{OtherWhiteSpace} {: /* ignore */ :}

{OpenAngleBracket} {:
   return symbol(LANGLE);
:} START_STATE_LIST

{OpenCurlyBracket} / {MacroLookahead} {:
   return symbol(OPEN_MACRO);
:} START_MACRO_INVOC

{OpenCurlyBracket} / {RepetitionLookahead} {:
   return symbol(OPEN_REPETITION_SPEC);
:} START_REP_SPEC

{OpenAction} {: appendToStartDelim(""); /*tweak start pos*/ :}
   START_ACTION

//for ending state rule-groups
{CloseCurlyBracket} {: return symbol(CLOSE_RULE_GROUP); :}

{GroupSeparator} {: return symbol(GROUP_SEPARATOR); :}
{SectionSeparator}inherit / {DirectiveLookahead} {:
   return symbol(INHERIT_SECTION_SEPARATOR);
:} START_STRING_ID_DIRECTIVE

//lookahead ensures that "%delete" isn't a prefix (e.g. "%deleted") and
   that we're not in a regex (since no <)
"%delete" / {DirectiveLookahead} {:
   return symbol(DELETE_DIRECTIVE);
:} START_DELETE_DIRECTIVE
```

```
40
41 %:
42 %:
43
44 <<EOF>> {: :} END_RULE_SECTION
45
46 %%inherit comp_symbols
```

lexer/component/comp_rules.mlc

```
1  %component comp_symbols
2  %helper
3
4  %extern "private Symbol symbol(short type)"
5  %extern "private Symbol symbol(short type, Object value)"
6  %extern "private void error(String msg) throws Scanner.Exception"
7
8  %%
9
10 %%inherit comp_macros
11
12 \( {: return symbol(LPAREN); :}
13 \) {: return symbol(RPAREN); :}
14 \[ {: return symbol(LSQUARE); :} START_CHAR_CLASS
15 \] {: return symbol(RSQUARE); :}
16
17 \^ {: return symbol(BEGINNING_OF_LINE); :}
18 \$ {: return symbol(END_OF_LINE); :}
19
20 \! {: return symbol(NOT); :}
21 \~ {: return symbol(UPTO); :}
22
23 \* {: return symbol(STAR); :}
24 \+ {: return symbol(PLUS); :}
25 \? {: return symbol(OPT); :}
26
27 \- {: return symbol(DASH); :}
28
```

```
29 \/ {: return symbol(SLASH); :}
30 \| {: return symbol(ALT); :}
31
32 \. {: return symbol(DOT); :}
33
34 "=" {: return symbol(ASSIGN); :}
35
36 "<<ANY>>" {: return symbol(ANY_PATTERN); :}
37 "<<EOF>>" {: return symbol(EOF_PATTERN); :}
38
39 {EscapeSequence} {: return symbol(ESCAPE_SEQUENCE, yytext()); :}
40 \\{Any} {: return symbol(NON_META, yytext().substring(1)); :}
41 \\ {: error("Incomplete escape sequence"); :}
42
43 {Quote} {: appendToStartDelim(""); /*tweak start pos*/ :} START_STRING
44
45 //safe fallback for patterns
46 {NonMeta} {: return symbol(NON_META, yytext()); :}
47
48 %:
49 %:
50
51 <<ANY>> {: error("Unexpected character: " + yytext()); :}
```

lexer/component/comp_symbols.mlc

```
1 %component delete_directive
2
3 %extern "private Symbol symbol(short type)"
4
5 %%
6
7 %%inherit comment_start
8
9 {LineTerminator} {: return symbol(DELETE_TERMINATOR); :}
     END_DELETE_DIRECTIVE
10 {OtherWhiteSpace} {: /* ignore */ :}
11
```

```
{OpenAngleBracket} {: return symbol(LANGLE); :} START_STATE_LIST

{OpenCurlyBracket} / {MacroLookahead} {:
   return symbol(OPEN_MACRO);
:} START_MACRO_INVOC

{OpenCurlyBracket} / {RepetitionLookahead} {:
   return symbol(OPEN_REPETITION_SPEC);
:} START_REP_SPEC

%%inherit comp_symbols

%:
%:

<<EOF>> {: return symbol(DELETE_TERMINATOR); :} END_DELETE_DIRECTIVE
```

lexer/component/delete_directive.mlc

```
%component macro_defn

%extern "private Symbol symbol(short type)"

%%

%%inherit comp_macros
%%inherit comment_start

//whitespace
{LineTerminator} {: :} END_MACRO_DEFN
{OtherWhiteSpace} {: /* ignore */ :}

{OpenCurlyBracket} / {MacroLookahead} {:
   return symbol(OPEN_MACRO);
:} START_MACRO_INVOC

{OpenCurlyBracket} / {RepetitionLookahead} {:
   return symbol(OPEN_REPETITION_SPEC);
```

```
:} START_REP_SPEC

%:
%:

<<EOF>> {: :} END_MACRO_DEFN

%%inherit comp_symbols
```

lexer/component/macro_defn.mlc

```
%component macro_invoc

%extern "private Symbol symbol(short type)"
%extern "private Symbol symbol(short type, Object value)"
%extern "private void error(String msg) throws Scanner.Exception"

%%

%%inherit comp_macros

//whitespace
{LineTerminator} {: /*ignore*/ :}
{OtherWhiteSpace} {: /*ignore*/ :}

{CloseCurlyBracket} {: return symbol(CLOSE_MACRO); :} END_MACRO_INVOC

%:

//for macro names
{Identifier} {: return symbol(IDENTIFIER, yytext()); :}

%:

<<ANY>> {: error("Unexpected character in macro invocation: " +
   yytext()); :}
<<EOF>> {: error("Unterminated macro invocation"); :} EOF_ERROR
```

lexer/component/macro_invoc.mlc

```
1  %component mtok_decl
2
3  %extern "private Symbol symbol(short type, Object value)"
4  %extern "private void error(String msg) throws Scanner.Exception"
5
6  %%
7
8  %%inherit comp_macros
9  %%inherit comment_start
10
11 //whitespace
12 {LineTerminator} {: :} END_MTOK_DECL
13 {OtherWhiteSpace} {: /* ignore */ :}
14
15 %:
16
17 //for meta-token types
18 {Identifier} {: return symbol(IDENTIFIER, yytext()); :}
19
20 %:
21
22 <<ANY>> {: error("Unexpected character in meta token specification: " +
       yytext()); :}
23 <<EOF>> {: :} END_MTOK_DECL
```

lexer/component/mtok_decl.mlc

```
1  %component open_rule_group
2
3  %extern "private Symbol symbol(short type)"
4  %extern "private void error(String msg) throws Scanner.Exception"
5
6  %%
7
8  %%inherit comp_macros
```

```
%%inherit comment_start

//whitespace
{LineTerminator} {: /* ignore */ :}
{OtherWhiteSpace} {: /* ignore */ :}

{OpenCurlyBracket} {: return symbol(OPEN_RULE_GROUP); :}
    END_OPEN_RULE_GROUP

%:
%:

<<ANY>> {: error("Expecting '{', found: " + yytext()); :}
<<EOF>> {: error("No group associated with state list"); :} EOF_ERROR
```

lexer/component/open_rule_group.mlc

```
%component repetition_spec

%extern "private Symbol symbol(short type)"
%extern "private Symbol symbol(short type, Object value)"
%extern "private void error(String msg) throws Scanner.Exception"

%%

%%inherit comp_macros

//whitespace
{LineTerminator} {: /*ignore*/ :}
{OtherWhiteSpace} {: /*ignore*/ :}

//for separating repetition quantities
, {: return symbol(COMMA); :}

{CloseCurlyBracket} {: return symbol(CLOSE_REPETITION_SPEC); :}
    END_REP_SPEC

%:
```

```
21
22 //for repetition quantities
23 {Number} {: return symbol(NUMBER, yytext()); :}
24
25 %:
26
27 <<ANY>> {: error("Unexpected character in repetition specification: " +
       yytext()); :}
28 <<EOF>> {: error("Unterminated repetition specification"); :} EOF_ERROR
```

lexer/component/repetition_spec.mlc

```
1  %component state_list
2
3  %extern "private Symbol symbol(short type)"
4  %extern "private Symbol symbol(short type, Object value)"
5  %extern "private void error(String msg) throws Scanner.Exception"
6
7  %%
8
9  %%inherit comp_macros
10 %%inherit comment_start
11
12 {CloseAngleBracket} {: return symbol(RANGLE); :} END_STATE_LIST
13
14 //whitespace
15 {LineTerminator} {: /* ignore */ :}
16 {OtherWhiteSpace} {: /* ignore */ :}
17
18 //for separating state names
19 , {: return symbol(COMMA); :}
20
21 %:
22
23
24 //for state names
25 {Identifier} {: return symbol(IDENTIFIER, yytext()); :}
26
```

```
%:

//catchall - error
<<ANY>> {: error("Unexpected character in state list: " + yytext()); :}
<<EOF>> {: error("Unterminated state list"); :} EOF_ERROR
```

lexer/component/state_list.mlc

C.2 Layout

```
package metalexer;
%%
//based on http://jflex.de/manual.html

import static metalexer.LayoutParser.Terminals.*;
%%

%layout layout

%option visibility "%public"
%option finality "%final"
%option class_name "%class LayoutScanner"

%component base
%component bracket_comment
%component decl_region
%component init_region
%component layout_inherited_header
%component layout_local_header
%component layout_options
%component layout_rules
%component mpat_directive
%component pair_directive
%component string_id_directive
%component string

```

```
%start base

%%

%%inherit beaver
%%inherit helper

%%embed
%name local_header
%host base
%guest layout_local_header
%start <BOF>
%end END_LOCAL_HEADER

%%embed
%name inherited_header
%host base
%guest layout_inherited_header
%start %layout_local_header%
%end END_INHERITED_HEADER

%%embed
%name options_section
%host base
%guest layout_options
%start %layout_inherited_header%
%end END_OPTION_SECTION

%%embed
%name rules_section
%host base
%guest layout_rules
%start %layout_options%
%end END_RULES_SECTION

%%embed
%name bracket_comments
```

```
64 %host layout_options, layout_rules, string_id_directive,
       mpat_directive, pair_directive
65 %host bracket_comment //NB: host and guest => nestable
66 %guest bracket_comment
67 %start START_BRACKET_COMMENT
68 %end END_BRACKET_COMMENT
69
70 %%embed
71 %name init_region
72 %host layout_options
73 %guest init_region
74 %start START_INIT_REGION
75 %end END_INIT_REGION
76
77 %%embed
78 %name decl_region
79 %host layout_options
80 %guest decl_region
81 %start START_DECL_REGION
82 %end END_DECL_REGION
83
84 %%embed
85 %name string_id_directive
86 %host layout_options, layout_rules
87 %guest string_id_directive
88 %start START_STRING_ID_DIRECTIVE
89 %end END_STRING_ID_DIRECTIVE
90
91 %%embed
92 %name pair_directive
93 %host layout_rules
94 %guest pair_directive
95 %start START_PAIR_DIRECTIVE
96 %end END_PAIR_DIRECTIVE
97
98 %%embed
99 %name mpat_directive
```

```
100 %host layout_rules
101 %guest mpat_directive
102 %start START_MPAT_DIRECTIVE
103 %end END_MPAT_DIRECTIVE
104
105 %%embed
106 %name string
107 %host string_id_directive
108 %guest string
109 %start START_STRING
110 %end END_STRING
111
112 %%embed
113 %name eof_error
114 %host bracket_comment, decl_region, init_region,
        layout_inherited_header, layout_local_header, string
115 %guest base
116 %start EOF_ERROR
117 %end <ANY> //NB: will never happen
```

lexer/layout/layout.mll

```
1  %component layout_inherited_header
2
3  %extern "private Symbol symbol(short type, Object value, int startLine,
       int startCol, int endLine, int endCol)"
4  %extern "private void error(String msg) throws Scanner.Exception"
5
6  %append{ /*(int startLine, int startCol, int endLine, int endCol,
       String text)*/
7     if(text.startsWith("\r\n")) {
8        text = text.substring(2);
9     } else if(text.startsWith("\n") || text.startsWith("\r")) {
10       text = text.substring(1);
11    }
12    return symbol(INHERITED_HEADER, text, startLine + 1, startCol + 1,
          endLine + 1, endCol + 1);
13 %append}
```

```
14
15 %%
16
17 %%inherit layout_macros
18
19 //end of section
20 {SectionSeparator} {: :} END_INHERITED_HEADER
21
22 %:
23 %:
24
25 <<ANY>> {: append(yytext()); :}
26 <<EOF>> {: error("Unterminated inherited header section."); :} EOF_ERROR
```

lexer/layout/layout_inherited_header.mlc

```
1  %component layout_local_header
2
3  %extern "private Symbol symbol(short type, Object value, int startLine,
       int startCol, int endLine, int endCol)"
4  %extern "private void error(String msg) throws Scanner.Exception"
5
6  %append{ /*(int startLine, int startCol, int endLine, int endCol,
       String text)*/
7     return symbol(LOCAL_HEADER, text, startLine + 1, startCol + 1,
          endLine + 1, endCol + 1);
8  %append}
9
10 %%
11
12 %%inherit layout_macros
13
14 //end of section
15 {SectionSeparator} {: :} END_LOCAL_HEADER
16
17 %:
18 %:
19
```

```
20 <<ANY>> {: append(yytext()); :}
21 <<EOF>> {: error("Unterminated local header section."); :} EOF_ERROR
```

lexer/layout/layout_local_header.mlc

```
1 %component layout_macros
2 %helper
3
4 %%
5
6 %%inherit shared_macros
```

lexer/layout/layout_macros.mlc

```
1 %component layout_options
2
3 %extern "private Symbol symbol(short type)"
4 %extern "private void error(String msg) throws Scanner.Exception"
5
6 %%
7
8 %%inherit layout_macros
9 %%inherit comment_start
10
11 //whitespace
12 {LineTerminator} {: /* ignore */ :}
13 {OtherWhiteSpace} {: /* ignore */ :}
14
15 {OpenDeclRegion} {: appendToStartDelim(""); /*tweak start pos*/ :}
      START_DECL_REGION
16 {OpenInitRegion} {: appendToStartDelim(""); /*tweak start pos*/ :}
      START_INIT_REGION
17
18 //no-arg directives
19 "%helper" / {DirectiveLookahead} {:
20    return symbol(HELPER_DIRECTIVE);
21 :} START_STRING_ID_DIRECTIVE
22
```

```
23 //identifier directives
24 "%layout" / {DirectiveLookahead} {:
25    return symbol(LAYOUT_DIRECTIVE);
26 :} START_STRING_ID_DIRECTIVE
27 "%start" / {DirectiveLookahead} {:
28    return symbol(START_DIRECTIVE);
29 :} START_STRING_ID_DIRECTIVE
30 "%component" / {DirectiveLookahead} {:
31    return symbol(COMPONENT_DIRECTIVE);
32 :} START_STRING_ID_DIRECTIVE
33
34 //string directives
35 "%declare" / {DirectiveLookahead} {:
36    return symbol(DECLARE_DIRECTIVE);
37 :} START_STRING_ID_DIRECTIVE
38 "%initthrow" / {DirectiveLookahead} {:
39    return symbol(INITTHROW_DIRECTIVE);
40 :} START_STRING_ID_DIRECTIVE
41 "%lexthrow" / {DirectiveLookahead} {:
42    return symbol(LEXTHROW_DIRECTIVE);
43 :} START_STRING_ID_DIRECTIVE
44
45 //mixed directives
46 "%option" / {DirectiveLookahead} {:
47    return symbol(OPTION_DIRECTIVE);
48 :} START_STRING_ID_DIRECTIVE
49
50 //invalid directives
51 "%" {: error("Invalid directive"); :}
52
53 //end of section
54 {SectionSeparator} {: :} END_OPTION_SECTION
55
56 %:
57 %:
58
59 <<ANY>> {: error("Unexpected character in option section: " +
```

```
    yytext()); :}
<<EOF>> {: :} END_OPTION_SECTION
```

lexer/layout/layout_options.mlc

```
%component layout_rules

%extern "private Symbol symbol(short type)"
%extern "private void error(String msg) throws Scanner.Exception"

%%

%%inherit layout_macros
%%inherit comment_start

//whitespace
{LineTerminator} {: /* ignore */ :}
{OtherWhiteSpace} {: /* ignore */ :}

"%%embed" / {DirectiveLookahead} {:
   return symbol(START_EMBED_GROUP);
:}

"%name" / {DirectiveLookahead} {:
   return symbol(EMBEDDING_NAME);
:} START_STRING_ID_DIRECTIVE
"%host" / {DirectiveLookahead} {:
   return symbol(EMBEDDING_HOST);
:} START_STRING_ID_DIRECTIVE
"%guest" / {DirectiveLookahead} {:
   return symbol(EMBEDDING_GUEST);
:} START_STRING_ID_DIRECTIVE
"%pair" / {DirectiveLookahead} {:
   return symbol(EMBEDDING_PAIR);
:} START_PAIR_DIRECTIVE

"%start" / {DirectiveLookahead} {:
   return symbol(EMBEDDING_START);
```

```
:} START_MPAT_DIRECTIVE
"%end" / {DirectiveLookahead} {:
   return symbol(EMBEDDING_END);
:} START_MPAT_DIRECTIVE

"%%inherit" / {DirectiveLookahead} {:
   return symbol(START_INHERIT_GROUP);
:} START_STRING_ID_DIRECTIVE
"%unembed" / {DirectiveLookahead} {:
   return symbol(INHERIT_UNEMBED);
:} START_STRING_ID_DIRECTIVE
"%replace" / {DirectiveLookahead} {:
   return symbol(INHERIT_REPLACE);
:} START_STRING_ID_DIRECTIVE
"%unoption" / {DirectiveLookahead} {:
   return symbol(INHERIT_UNOPTION);
:} START_STRING_ID_DIRECTIVE

%:
%:

<<ANY>> {: error("Unexpected character in rule section: " + yytext());
   :}
<<EOF>> {: :} END_RULES_SECTION
```

lexer/layout/layout_rules.mlc

```
%component mpat_directive

%extern "private Symbol symbol(short type)"
%extern "private Symbol symbol(short type, Object value)"
%extern "private void error(String msg) throws Scanner.Exception"

%%

%%inherit layout_macros
%%inherit comment_start

```

```
12 //whitespace
13 {LineTerminator} {: :} END_MPAT_DIRECTIVE
14 {OtherWhiteSpace} {: /* ignore */ :}
15
16 "<ANY>" {: return symbol(MP_ANY); :}
17 "<BOF>" {: return symbol(MP_BOF); :}
18 "(" {: return symbol(MP_LPAREN); :}
19 ")" {: return symbol(MP_RPAREN); :}
20 "[" {: return symbol(MP_LSQUARE); :}
21 "]" {: return symbol(MP_RSQUARE); :}
22 "^" {: return symbol(MP_CARET); :}
23 "*" {: return symbol(MP_STAR); :}
24 "+" {: return symbol(MP_PLUS); :}
25 "?" {: return symbol(MP_OPT); :}
26 "|" {: return symbol(MP_OR); :}
27
28 %:
29
30 "%"{Identifier}"%" {: return symbol(MP_REGION, yytext().substring(1,
      yylength() - 1)); :}
31 "%"{QualifiedIdentifier}"%" {: return symbol(MP_REGION,
      yytext().substring(1, yylength() - 1)); :}
32 {Identifier} {: return symbol(MP_SYM, yytext()); :}
33
34 %:
35
36 //catchall - error
37 <<ANY>> {: error("Unexpected character in meta-pattern: " + yytext());
      :}
38 <<EOF>> {: :} END_MPAT_DIRECTIVE
```

lexer/layout/mpat_directive.mlc

```
1 %component pair_directive
2
3 %extern "private Symbol symbol(short type)"
4 %extern "private Symbol symbol(short type, Object value)"
5 %extern "private void error(String msg) throws Scanner.Exception"
```

```
6
7  %%
8
9  %%inherit layout_macros
10 %%inherit comment_start
11
12 //whitespace
13 {LineTerminator} {: :} END_PAIR_DIRECTIVE
14 {OtherWhiteSpace} {: /* ignore */ :}
15
16 , {: return symbol(COMMA); :}
17
18 %:
19
20 "%"{Identifier}"%" {: return symbol(MP_REGION, yytext().substring(1,
       yylength() - 1)); :}
21 "%"{QualifiedIdentifier}"%" {: return symbol(MP_REGION,
       yytext().substring(1, yylength() - 1)); :}
22 {Identifier} {: return symbol(MP_SYM, yytext()); :}
23
24 %:
25
26 //catchall - error
27 <<ANY>> {: error("Unexpected character in pair element: " + yytext());
       :}
28 <<EOF>> {: :} END_PAIR_DIRECTIVE
```

lexer/layout/pair_directive.mlc

C.3 Shared

```
1  %%
2  %%
3
4  %layout helper
5  %helper
```

```
6
7  %declare "private Symbol symbol(short type)"
8  %declare "private Symbol symbol(short type, Object value)"
9  %declare "private Symbol symbol(short type, Object value, int
       startLine, int startCol, int endLine, int endCol)"
10 %{
11   //// Returning symbols
       ////////////////////////////////////////////////////
12
13   //Create a symbol using the current line and column number, as
       computed by JFlex
14   //No attached value
15   //Symbol is assumed to start and end on the same line
16   //e.g. symbol(SEMICOLON)
17   private Symbol symbol(short type) {
18     return symbol(type, null);
19   }
20
21   //Create a symbol using the current line and column number, as
       computed by JFlex
22   //Attached value gives content information
23   //Symbol is assumed to start and end on the same line
24   //e.g. symbol(IDENTIFIER, "x")
25   private Symbol symbol(short type, Object value) {
26     //NB: JFlex is zero-indexed, but we want one-indexed
27     int startLine = yyline + 1;
28     int startCol = yycolumn + 1;
29     int endLine = startLine;
30     int endCol = startCol + yylength() - 1;
31     return symbol(type, value, startLine, startCol, endLine, endCol);
32   }
33
34   //Create a symbol using explicit position information (input is
       one-indexed)
35   private Symbol symbol(short type, Object value, int startLine, int
       startCol, int endLine, int endCol) {
36     int startPos = Symbol.makePosition(startLine, startCol);
```

```
37    int endPos = Symbol.makePosition(endLine, endCol);
38    return new Symbol(type, startPos, endPos, value);
39   }
40 %}
41
42 %declare "private void error(String msg) throws Scanner.Exception"
43 %declare "private void error(String msg, int columnOffset) throws
       Scanner.Exception"
44 %{
45   //// Errors
       ////////////////////////////////////////////////////////////////
46
47   //throw an exceptions with position information from JFlex
48   private void error(String msg) throws Scanner.Exception {
49    //correct to one-indexed
50    throw new Scanner.Exception(yyline + 1, yycolumn + 1, msg);
51   }
52
53   //throw an exceptions with position information from JFlex
54   //columnOffset is added to the column
55   private void error(String msg, int columnOffset) throws
        Scanner.Exception {
56   //correct to one-indexed
57    throw new Scanner.Exception(yyline + 1, yycolumn + 1 + columnOffset,
         msg);
58   }
59 %}
```

lexer/shared/helper.mll

```
1 %%
2 import beaver.Symbol;
3 import beaver.Scanner;
4 %%
5
6 %layout beaver
7 %helper
8
```

```
//required for beaver compatibility
%option parent_class "%extends Scanner"
%option encoding "%unicode"
%option function_name "%function nextToken"
%option token_type "%type Symbol"
%lexthrow "Scanner.Exception"

%option line "%line"
%option column "%column"
```

lexer/shared/beaver.mll

```
%component base

%extern "private Symbol symbol(short type)"
%extern "private void error(String msg) throws Scanner.Exception"

%%

%:
%:

<<ANY>> {: error("Unexpected character: " + yytext()); :}
<<EOF>> {: return symbol(EOF); :}
```

lexer/shared/base.mlc

```
%component bracket_comment

%extern "private void error(String msg) throws Scanner.Exception"

%%

%%inherit shared_macros

{OpenBracketComment} {: :} START_BRACKET_COMMENT
{CloseBracketComment} {: :} END_BRACKET_COMMENT

```

```
%:
%:

<<ANY>> {: /* ignore */ :}
<<EOF>> {: error("Unterminated bracket comment"); :} EOF_ERROR
```

lexer/shared/bracket_comment.mlc

```
%component comment_start
%helper

%%

%%inherit shared_macros

{OpenBracketComment} {: :} START_BRACKET_COMMENT

%:

{Comment} {: /* ignore */ :}
```

lexer/shared/comment_start.mlc

```
%component decl_region

%extern "private Symbol symbol(short type, Object value, int startLine,
    int startCol, int endLine, int endCol)"
%extern "private void error(String msg) throws Scanner.Exception"

%appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
    endCol, String text)*/
    return symbol(DECL_REGION, text, startLine + 1, startCol + 1,
        endLine + 1, endCol + 1);
%appendWithStartDelim}

%%

%%inherit shared_macros
```

```
13
14 %{CloseDeclRegion} {: append(yytext().substring(1)); :}
15 {CloseDeclRegion} {: :} END_DECL_REGION
16
17 %:
18 %:
19
20 <<ANY>> {: append(yytext()); :}
21 <<EOF>> {: error("Unterminated declaration region"); :} EOF_ERROR
```

lexer/shared/decl_region.mlc

```
1  %component init_region
2
3  %extern "private Symbol symbol(short type, Object value, int startLine,
       int startCol, int endLine, int endCol)"
4  %extern "private void error(String msg) throws Scanner.Exception"
5
6  %appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
       endCol, String text)*/
7     return symbol(INIT_REGION, text, startLine + 1, startCol + 1,
          endLine + 1, endCol + 1);
8  %appendWithStartDelim}
9
10 %%
11
12 %%inherit shared_macros
13
14 %{CloseInitRegion} {: append(yytext().substring(1)); :}
15 {CloseInitRegion} {: :} END_INIT_REGION
16
17 %:
18 %:
19
20 <<ANY>> {: append(yytext()); :}
21 <<EOF>> {: error("Unterminated initialization region"); :} EOF_ERROR
```

lexer/shared/init_region.mlc

```
%component shared_macros
%helper

LineTerminator = \r|\n|\r\n
OtherWhiteSpace = [ \t\f]

Letter = [a-zA-Z]
Digit = [0-9]
Identifier = {Letter} (_ | {Letter} | {Digit})*
QualifiedIdentifier = {Identifier} (\. {Identifier})+
Number = 0 | [1-9] {Digit}*

Quote = \"

HexDigit = {Digit} | [a-fA-F]
EscapeSequence = \\ ( [nrtfb] | x {HexDigit}{2} | u {HexDigit}{1,4} |
    [0-3]? [0-7]? [0-7] |
    [\|\(\)\{\}\[\]\<\>\\\.\*\+\?\^\$\/\.\"\~\!\-])

Any = . | \n

Comment = "//"[^\r\n]*
OpenBracketComment = "/*"
CloseBracketComment = "*/"

SectionSeparator = "%%"

OpenDeclRegion = "%{"
CloseDeclRegion = "%}"

OpenInitRegion = "%init{"
CloseInitRegion = "%init}"

//handles comments
DirectiveLookahead = !({Letter} | {Digit} | _) //NB: use not (instead
    of a class) because it also catches EOF
```

lexer/shared/shared_macros.mlc

```
%component string

%extern "private Symbol symbol(short type, Object value, int startLine,
    int startCol, int endLine, int endCol)"
%extern "private void error(String msg) throws Scanner.Exception"

%appendWithStartDelim{ /*(int startLine, int startCol, int endLine, int
    endCol, String text)*/
   return symbol(STRING, text, startLine + 1, startCol + 1, endLine +
       1, endCol + 1);
%appendWithStartDelim}

%%

%%inherit shared_macros

{Quote} {: :} END_STRING
{EscapeSequence} {: append(yytext()); :}
\\{Any} {: append(yytext().substring(1)); :}
\\ {: error("Incomplete escape sequence"); :}

{LineTerminator} {: error("Unterminated string literal"); :}

%:
%:

<<ANY>> {: append(yytext()); :}
<<EOF>> {: error("Unterminated string literal"); :} EOF_ERROR
```

lexer/shared/string.mlc

```
%component string_id_directive

%extern "private Symbol symbol(short type)"
%extern "private Symbol symbol(short type, Object value)"
```

```
5  %extern "private void error(String msg) throws Scanner.Exception"
6
7  %%
8
9  %%inherit shared_macros
10 %%inherit comment_start
11
12 //whitespace
13 {LineTerminator} {: :} END_STRING_ID_DIRECTIVE
14 {OtherWhiteSpace} {: /* ignore */ :}
15
16 //for strings
17 {Quote} {: appendToStartDelim(""); /*tweak start pos*/ :} START_STRING
18
19 , {: return symbol(COMMA); :}
20
21 %:
22
23 //for state names
24 {Identifier} {: return symbol(IDENTIFIER, yytext()); :}
25 {QualifiedIdentifier} {: return symbol(QUALIFIED_IDENTIFIER, yytext());
       :}
26
27 %:
28
29 <<ANY>> {: error("Unexpected character in string/identifier directive:
       " + yytext()); :}
30 <<EOF>> {: :} END_STRING_ID_DIRECTIVE
```

lexer/shared/string_id_directive.mlc

Bibliography

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc : An extensible AspectJ compiler. In *AOSD 05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, Chicago, Illinois, 2005, pages 87–98. ACM, New York, NY, USA.

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 1998.

[BETV06] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, 2006, pages 209–228. ACM, New York, NY, USA.

[BSV03] Claus Brabrand, Michael I. Schwartzbach, and Mads Vanggaard. The metafront system: Extensible parsing and transformation. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications, LDTA '03*, 2003, volume 82(3), pages 592–611.

[Bur95] BD Burow. Mixed language programming. In *Proceedings of Computing in High Energy Physics*, 1995. http://www.hep.net/chep95/html/papers/p69/p69.pdf.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, Canada, 2004, pages 365–383. ACM, New York, NY, USA.

[EH80] Pär Emanuelson and Anders Haraldsson. On compiling embedded languages in LISP. In *LFP '80: Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, Stanford University, California, United States, 1980, pages 208–215. ACM, New York, NY, USA.

[EH07a] Torbjörn Ekman and Görel Hedin. The Jastadd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[EH07b] Torbjörn Ekman and Görel Hedin. The JastAdd system: Modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, Dec 2007.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, Jan 2005.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A framework for the development of textual domain specific languages. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008, pages 925–926. ACM, New York, NY, USA.

[Gri06] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Ontario, Canada, 2006, pages 38–51. ACM, New York, NY, USA.

[HdMC04] Laurie Hendren, Oege de Moor, and Aske Simon Christensen. The abc scanner and parser. Technical report, Programming Tools Group, Oxford University and the Sable research group, McGill University, Sep 2004. http://abc.comlab.ox.ac.uk/documents/scannerandparser.pdf.

[HG07] Martin Hirzel and Robert Grimm. Jeannie: Granting Java Native Interface developers their wishes. *ACM SIGPLAN Notices*, 42(10):19–38, 2007.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Comp. Sci. Tech. Rep. 32, Bell Laboratories, Jan 1975.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001 Object-Oriented Programming*, 2001, pages 327–353. Springer-Verlag.

[KKV08] Lennart Kats, Karl Trygve Kalleberg, and Eelco Visser. Generating editors for embedded languages. Technical Report Series TUD-SERG-2008-006, Delft University of Technology, Software Engineering Research Group, 2008. http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-006.pdf.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[LS75] M.E. Lesk and Eric Schmidt. Lex: A lexical analyzer generator. Comp. Sci. Tech. Rep. 39, Bell Laboratories, Oct 1975.

[Mar03] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Higher Education, New York, NY, USA, 2003.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, 2003, pages 138–152. Springer-Verlag.

[Par07] Terrence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, Raleigh, NC, 2007.

[Vol05] Elias Volanakis. Mixed-language usage scenarios in Eclipse. Position Paper for the Eclipse Languages Symposium, Oct 2005. <http://www.eclipse.org/org/langsymp/Volanakis - Mixed-language usage scenarios.pdf> .

[WS95] Fuliang Weng and Andreas Stolcke. Partitioning grammars and composing parsers. In *Proceedings of the 4th International Workshop on Parsing Technologies*, 1995.