MetaLexer: A Modular Lexical Specification Language

Andrew Casey and Laurie Hendren
School of Computer Science
McGill University
3480 University Street
Montreal, Quebec, H3A 2A7
[acasey,hendren]@cs.mcgill.ca

ABSTRACT

Compiler toolkits make it possible to rapidly develop compilers and translators for new programming languages. Although there exist elegant toolkits for modular and extensible parsers, compiler developers must often resort to ad-hoc solutions when extending or composing lexers. This paper presents MetaLexer, a new modular lexical specification language and associated tool.

MetaLexer allows programmers to define lexers in a modular fashion. MetaLexer modules can be used to break the lexical specification of a language into a collection smaller modular lexical specifications. Control is passed between the modules using the concept of meta-tokens and meta-lexing. MetaLexer modules are also extensible.

MetaLexer has two key features: it abstracts lexical state transitions out of semantic actions and it makes modules extensible by introducing multiple inheritance.

We have constructed a MetaLexer tool which converts MetaLexer specifications to the popular JFlex lexical specification language and we have used our tool to create lexers for three real programming languages and their extensions: AspectJ (and two AspectJ extensions), MATLAB (and the AspectMatlab extension), and MetaLexer itself. The new specifications are easier to read, are extensible, and require much less action code than the originals.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*processors*

General Terms

Languages

1. INTRODUCTION

Compiler-generator toolkits enable rapid development of compilers by generating front-ends from lexical and parser specifications. Lexers are intended to recognize simple regular languages and are typically used to perform tokenization for parsers. In practice, however, the languages handled by lexers often consist of multiple sub-languages. For example, in Java there are separate lexing rules for strings, comments, and Javadoc specifications. These different sub-languages are often dealt with by allowing a lexer to have different states, each of which lexes a particular sublanguage.

Mixed-language programming is another area that requires lexing of different sub-languages. Although this is not a new concept[1], mixed-language programming is growing in popularity, especially in the web development community. HTML documents often contain embedded JavaScript [33] and CSS [32]. Languages like ASP [25] and JSP [30] go a step further and mix general purpose languages with HTML.

Languages are often extended with new functionality. For example, in aspect-oriented languages like AspectJ, new constructs are added to an existing programming language base. In many cases these extensions have a very different lexical structure from the base. For example, if the string "`if*1`" is part of Java code, it should be tokenized as three tokens: the keyword `if`, the operator `*` and the integer constant `1`. If this same string occurred in an AspectJ pointcut, it should be be interpreted as a pattern that matches all identifiers beginning with the substring "`if`" and ending with the substring "`1`". Clearly in this case the "`if`" is not a keyword token.

Though there are some very nice tools for developing modular and extensible parsers, there appears to be a real lack of tools for modular and extensible lexers that are separate from the parser. As just one example, the *abc* AspectJ compiler[2] used two parser toolkits, Polyglot[7] and JastAdd[12], to implement the AspectJ extensions to Java and also to enable further AspectJ extensions. Building the parser extensions using these toolkits was very natural. However, it turned out that the lexical structure of the AspectJ-specific grammar is quite different from that of Java and considerable work was needed to create an ad-hoc solution for the lexer. Problems like this, plus the increasing popularity of defining mixed languages inspired us to develop a solution for modular and extensible lexers.

This paper introduces MetaLexer, a new lexical specification language and associated tool, which provides a clean, extensible and modular approach to lexing co-existing sublanguages. MetaLexer is intended to be a more advanced alternative to traditional tools, such as JFlex[21], and it has been designed to work well with traditional parser-generator

[1]Since the early days of C, programmers have been inserting blocks of assembly with *asm* regions [19]. Around the same time, C was being embedded in Lex specifications [23].

AOSD'11, March 21–25, 2011, Pernambuco, Brazil.

tools. Thus, a MetaLexer-based lexer could, for example, be used in existing or new projects with traditional parser-generators such as yacc/bison or with more complete compiler systems such as JastAdd and Polyglot.

One of the main challenges was to find a mechanism to naturally compose lexers for different sub-languages. Such a composable specification allows one to: (1) define each sub-language separately; (2) easily reuse previously defined sub-language lexers; and (3) be able to easily extend existing specifications. Normally, the lexical rules in a traditional lexer-specification language are tightly coupled by the lexer's transition logic. Thus, in order to be able to decompose these specifications, such that there is one modular sub-specification for each sub-language, one must invent a way to lift out and abstract the state transition logic.

Solving this problem led to the main idea of our new approach, *meta-lexing*. The fundamental idea of meta-lexing is that each pattern (rule) in an ordinary lexer can have an associated meta-token. Then, sequences of meta-tokens can be used to guide control flow. We call it meta-lexing because recognizing the patterns of meta-tokens is a form of lexing at the meta-token level. One could think of this as having two levels of lexing, which are cooperating with each other. Conceptually, there is a base-level lexer for each sub-language. At a particular point in time, one of those sub-language lexers will be active and will process the incoming character stream and produce two output streams - a stream of ordinary tokens as well as a stream of meta-tokens. As lexing proceeds, the stream of meta-tokens is meta-lexed to recognize when transitions from one sub-language to another should occur. In particular, we use the concept of an embedding to specify when the metalexer should shift to a new state and when it should shift back to the previous state.

Two key features distinguish MetaLexer from its predecessors:

1. Lexical state transitions are lifted out of semantic actions (Section 2.1) and

2. modules support multiple inheritance (Section 2.2).

We have implemented a translator from MetaLexer to the popular JFlex lexical-specification language and used it to build lexers for three real programming languages: AspectJ (and two extensions), MATLAB (and the AspectMatlab extension), and MetaLexer itself. The new specifications are easier to read and require much less lexer action code (e.g. in C or Java) than the originals.[2]

The main contributions of this paper are:

- the definition of a new paradigm for modularly specifying lexers based on the concept of meta-lexing;

- a tool implementing the new paradigm which translates metalexer specifications to a standard JFlex specification; and

- applications of the tool to three real-world programming languages and their extensions.

The remainder of the paper is organized as follows. Section 2 describes the key features of MetaLexer in greater detail. Section 3 introduces MetaLexer syntax using a practical example. Section 4 outlines the implementation of the MetaLexer-to-JFlex translator. Section 5 describes our experience building lexers for AspectJ, MATLAB, and MetaLexer. Section 6 discusses related work. Finally, Section 7 summarizes our conclusions and suggests opportunities for future work.

[2]See `http://www.sable.mcgill.ca/metalexer` for the MetaLexer implementation, documentation, and examples.

2. KEY FEATURES

In this section we describe the key features that distinguish MetaLexer from its predecessors, and at the same time introduce some high-level examples to illustrate the ideas. In Section 2.1 we introduce our notion of meta-tokens/meta-lexing and the concept of two kinds of specifications: *components* to specify lexers for sub-languages and *layouts* to specify the transitions between sub-languages. As one of our objectives was to support extensible specifications, MetaLexer also supports inheritance, which is introduced in Section 2.2.

2.1 Key Feature: State Transitions

Practical lexers 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 tool-dependent.

For example, Listing 1 shows a JFlex 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. It remains in the string state until a closing quotation mark is seen. The previous state must be stored in a variable so that the lexer can return once the closing quote has been seen.

```
<YYINITIAL> {
        \" { yybegin(STRING_STATE); prev =
             YYINITIAL; }
        /* other rules related to lexing in the
             initial state */
}
<CLASS> {
        \" { yybegin(STRING_STATE); prev = CLASS; }
        /* other rules related to lexing within a
             class */
}
<STRING_STATE> {
        \" { yybegin(prev); return STRING(text); }
        /* other rules building up text buffer */
}
```

Listing 1: JFlex State Transitions

In other words, *when in state CLASS, transition to state STRING_STATE upon seeing the " token; transition back upon seeing the matching " token.* As in this example, 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.*

In our simple example, it is just one token that signals entering and leaving a state, but 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.

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 shows a possible organization of a MATLAB lexer. A (square) layout – *Matlab* – refers to three (rounded) components – *Base*, *String*, and *Comment*. Each of the components describes a lexical state and the layout describes their interaction. Note that we illustrate the actual syntax used within layouts and components in the next section (Section 3).

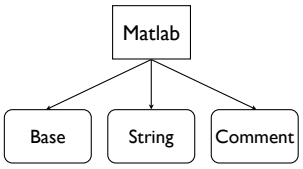


Figure 1: Layout (square) and components (rounded) for MATLAB

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 2 extends Figure 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.

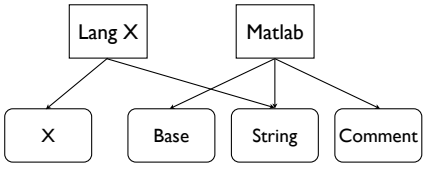


Figure 2: Two layouts sharing components

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 5.3). Additionally, the components of the *abc* language inherit many of the same helper components (Section 5.1).

2.2 Key Feature: Inheritance

MetaLexer uses inheritance to achieve extensibility and modularity. We have used these features extensively in our own applications of MetaLexer. The following small examples illustrate the key ideas.

Figure 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 (normal MATLAB strings support very few escape sequences). One could do this by inheriting the *String* component in a new *String++* component that 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.

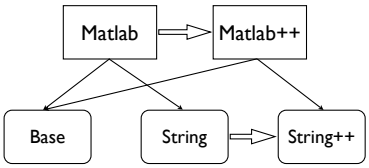


Figure 3: Using inheritance to extend the syntax of MATLAB strings

On the other hand, Figure 4 shows how inheritance can improve modularity by factoring out useful "helper" fragments into separate layouts/components. In this case, since the 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.

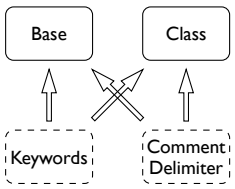


Figure 4: Using inheritance to improve modularity

The inheritance mechanism in MetaLexer is specific to the problem of lexing, and thus a key design goal was designing a lightweight, but sufficiently expressive, mechanism for specifying the order in which newly inherited lexing rules are combined with existing rules, and which rules take precedence in the case of a conflict. We return to these details in Section 3.2.

3. LANGUAGE

As we introduced in the previous section, 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 section introduces the syntax of both languages, and provides illustrative examples.[3]

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

[3]The complete specification and longer examples may be found at `www.sable.mcgill.ca/metalexer`.

line contains a key-value pair, a comment (from '#' to end-of-line), or whitespace. Listing 2 shows a sample properties file. It specifies three key-value pairs: (name, 'properties'), (date, '2009/09/21'), and (owner, 'root'). Everything else is ignored.

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

#some more properties
owner=root
```

Listing 2: Syntax Example – A Properties File

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. The overall structure will be two sub-language components, *key* and *value*, and a helper component *util_patterns*. The main idea is that the *key* component will handle most of the language, but the small *value* sub-language will be used to recognize the values that come after the `"="` token. The top-level specification, which also specifies the transitions between the components will be given in the *properties* layout file.

Let's first examine the *key* component (Listing 3) 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` specifies a point at which other rules should be inherited, in this case *util_patterns.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 util_patterns
{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: Syntax Example – key.mlc

Readers familiar with JFlex will note three main differences in the syntax of MetaLexer's rules. First, MetaLexer introduces a new (top-level) `<<ANY>>` pattern which is used to designate the catchall rule (described below).[4] 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 [8] for an explanation).

Now let us examine the small *value* component for lexing the values that occur after the `"="` token (Listing 4). It has many of the same features as Listing 3 – a component name, external declarations, inheritance of *util_patterns*, meta-tokens – but it also illustrates another MetaLexer 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. In this case, we are using an append block to gather up all of the characters that appear between the equals sign and the end of the line (or file).

```
%component value

%extern "Token symbol(int, String, int, int, int,
     int)"

%append{
     return symbol(VALUE, text, startLine, startCol,
          endLine, endCol);
%append}

%%

%%inherit util_patterns

{lineTerminator} {: :} LINE_TERMINATOR
%:
%:
<<ANY>> {: append(yytext()); :}
<<EOF>> {: :} LINE_TERMINATOR
```

Listing 4: Syntax Example – value.mlc

Listing 5 shows the *util_patterns* 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 pattern definitions themselves are just as in JFlex.

```
%component util_patterns
%helper

lineTerminator = [\r\n] | "\r\n"
otherWhitespace = [ \t\f\b]
identifier = [a-zA-Z][a-zA-Z0-9_]*
comment = #[^\r\n]*
```

Listing 5: Syntax Example – util_patterns.mlc

Finally, Listing 6 shows the *properties* layout that joins everything together. It is the layout that we will compile into a working lexer. Like a normal lexical 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.

[4] We intentionally made the syntax of MetaLexer look like the syntax of JFlex in order to make it easy for JFlex/Flex programmers to adapt to MetaLexer.

```
package properties;
%%
import static properties.TokenTypes.*;
%%
%layout properties

%option public "%public"
%option final "%final"
%option class "%class PropertiesLexer"
%option unicode "%unicode"
%option function "%function getNext"
%option type "%type Token"
%option pos_line "%line"
%option pos_column "%column"

%declare "Token symbol(int)"
%declare "Token symbol(int, String)"
%declare "Token symbol(int, String, int, int, int,
    int)"
%declare "void error(String) throws LexerException"
%{
    private Token symbol(int symbolType) { ... }
    private Token symbol(int symbolType, String
        text) { ... }
    private Token symbol(int symbolType, String
        text, int startLine, int startCol, int
        endLine, int endCol) { ...
    private void error(String msg) throws
        LexerException { ... }
%}

%lexthrow "LexerException"
%component key
%component value
%start key

%%

%%embed
%name key_value
%host key
%guest value
%start ASSIGN
%end LINE_TERMINATOR
```

Listing 6: Syntax Example – properties.mll

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 layout. The string part is passed directly to the underlying lexical specification language. Following the options are declarations in the underlying action implementation language (for example, Java) (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. Note that embeddings naturally express stack-based transitions between states.

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

This simple example shows the important syntax of MetaLexer. For a complete description of the syntax see [8].

3.2 Inheritance

Inheritance in MetaLexer must provide a simple and principled way of adding functionality to an existing MetaLexer specification for layouts or components.[5]

An `%%inherit` directive instructs MetaLexer to weave the contents of the referenced module into the current module. However, since the lexical specifications have internal structure and because the order of lexical rules is important, this weaving process must conform to that structure. MetaLexer splits up the inherited file and weaves its elements to the corresponding sections 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. However, the order of embeddings and lexical rules matters, and so MetaLexer supports well-defined mechanisms for specifying where insertions should occur.

3.2.1 Inheritance for layouts

An inheritance directive in a layout indicates that another layout should be inherited. It is of the form, `%%inherit` *layout*.

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.

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 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 the example in Figure 8 in Section 5). Replace directives are of the form, `%replace` *component, component*.

Unembed directives filter out embeddings from inherited layouts. They are of the form, `%unembed` *name*.

3.2.2 Inheritance for components

When a component inherits another component, the process is more complicated. The key observation is that the order of lexical rules matters, and we must have a way of weaving in the inherited rules at the appropriate places. We considered forcing the programmer to give a name to each rule, thus exposing explicit named weaving points. However, we discovered a much less burdensome approach based

[5]We distinguish here between inheritance and subtyping, though the two are often conflated. By inheritance, we mean a mechanism for sharing code, as opposed to a way to facilitate polymorphism (by substituting a child instance for a parent instance).

on structuring the rules into three categories and using a separator to delineate the boundary between categories.

In the preceding example, the components contained an additional separator – `%:`. This separator actually indicates a boundary between different categories of rules. MetaLexer has three such categories:

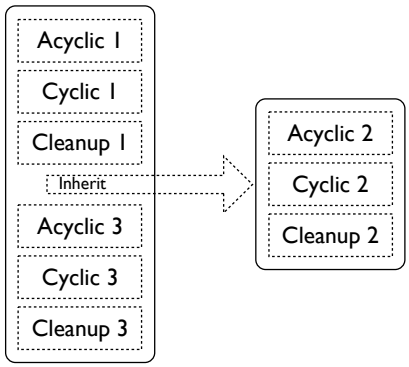
1. **Acyclic** rules can match only finitely many strings. Conceptually, their minimal DFAs are acyclic.

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

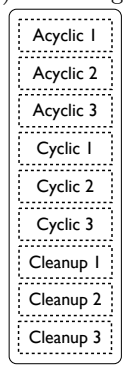
3. **Cleanup** rules are either catchall – `<<ANY>>` – or end-of-file – `<<EOF>>` – rules.

These categories always appear together and in order. A new Acyclic-Cyclic-Cleanup group begins after the section separator – `%%` – and after each `%%inherit` directive. We instituted this separation because insertion points are required for new rules and the boundaries between these categories are both natural and (in practice) sufficient.

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.



(a) a component (left) inheriting from a parent (right)



(b) ordering after weaving due to inheritance

Figure 5: Ordering of inherited rules for components

The conceptual rearrangement, which takes place during the inheritance process, is illustrated by Figure 5. Figure 5(a) shows a component (left) inheriting rules from its parent (right). The arrow from the left indicates the location of the inherit statement. Figure 5(b) shows the order of the rules in the flattened component (i.e. after inheritance). Note that the woven version retains the three groups (acyclic, cyclic and cleanup), and within each group the rules are listed in the order they would appear by inlining the inherited rules at the point of the inherit directive.

To make this more concrete, consider an explicit example of component inheritance. Figure 6(a) shows a component that extends the component defined on the right (`parent_comp`). The parent component has rules for some keywords and a number token, but the new component adds some new keywords and an identifier. Figure 6(b) shows the result. Note the order of the keywords in Figure 6(b) – 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_keywd1 {: action5 :}
new_keywd2 {: action6 :}
%:
{identifier} {: action7 :}
%:
<<ANY>> {: action8 :}
<<EOF>> {: action9 :}

%%inherit parent_comp

new_keywd3 {: action10 :}
```

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

(a) a component (left) inheriting from a parent (right)

```
%%
new_keywd1 {: action5 :}
new_keywd2 {: action6 :}
keyword1 {: action1 :}
keyword2 {: action2 :}
keyword3 {: action3 :}
new_keywd3 {: action10 :}
%:
{identifier} {: action7 :}
{number} {: action4 :}
%:
<<ANY>> {: action8 :}
<<EOF>> {: action9 :}
```

(b) ordering after weaving due to inheritance

Figure 6: Explicit example of inheritance

In our experience we have found the boundaries between these categories to be 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 usually be achieved by inserting the new rule at one of the boundaries. In general, 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.

3.2.3 Multiple Inheritance

Since MetaLexer allows modules to inherit from multiple parents, there may be conflicts. For example, pattern definitions, 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.

MetaLexer will warn when such a conflict occurs and resolve the conflict by choosing the first occurrence, given the ordering defined above. This default behaviour is consistent with JFlex and other traditional lexer systems.

4. IMPLEMENTATION

MetaLexer has been completely implemented using Java-based tools including JFlex, Beaver, and JastAdd. The lexing of MetaLexer's component and layout languages was originally done using JFlex, but it has since been bootstrapped, so the current implementation is written in MetaLexer itself. This section gives an overview of the organization of a MetaLexer lexer.

Figure 7 shows the usual model for a lexer (e.g. JFlex). Externally, the lexer reads a stream of characters and produces a stream of tokens. Internally, the lexer moves amongst a number of lexical states that determine which set of lexical rules will be used.

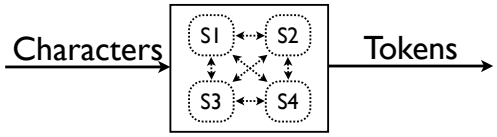


Figure 7: Data flow of a JFlex lexer

Figure 8 shows a MetaLexer lexer. Externally, it is very similar to the usual model. Internally, however, it uses a different mechanism to determine which set of lexical 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 – we have called *meta-lexing*.

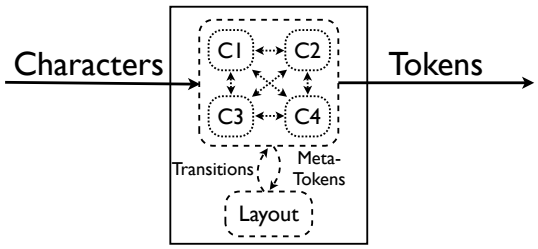


Figure 8: Data flow of a MetaLexer lexer

Our system compiles each MetaLexer component into JFlex code.[6] Perhaps the most difficult part of superimposing the component abstraction onto an existing lexical system like JFlex was introducing a new scope at the component level. This was crucial to ensure that components would remain independent and composable. We accomplished this by creating a non-static inner class within the lexer for each component. This allowed each component to access its own variables without having to worry about conflicts and also allowed components to share access to variables declared at the layout level.

Although components naturally translate to JFlex, the meta-lexing performed by layouts required a different approach. The key difference is that unlike traditional JFLex lexers, which match the longest-match, the semantics for the meta-lexer call for the shortest-match (i.e. transition as soon as *any* meta-pattern is matched). If longest-match semantics were used, the meta-lexer would have to wait for an additional meta-token before transitioning, to determine whether a longer match was possible. If not, the lexer would have to be rolled back because characters following the original meta-token would have been lexed by the wrong component.

To create a meta-lexer, we construct an NFA that starts with a self-loop that matches any extraneous symbols and then has ϵ-transitions to NFAs for each of the meta-patterns (Figure 9). We then apply standard techniques to convert it to a minimized DFA. One key point is that we must annotate each accepting state with its associated meta-pattern identifier to enable the backwards matching phase described below.

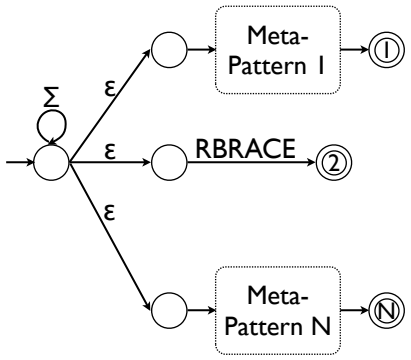


Figure 9: A high-level view of the ϵ-NFA generated for a lexical state of the meta-lexer.

One interesting twist is that after a match we need to determine which part of the matched string was the extraneous prefix recognized the initial self-loop and which was the suffix part matched by the meta-pattern. We do this using a backwards DFA which recognizes the meta-pattern suffix.

Let us demonstrate the meta-lexing process with an 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

[6]We have taken care not to rely on any JFlex-specific internals or syntax so that MetaLexer should be easy to retarget to other lexer tools.

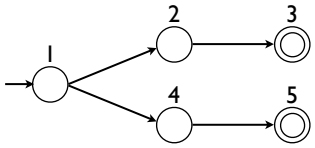
in an ϵ-NFA that looks something like Figure 9. However, we are likely to see a lot of extraneous symbols before we reach the end of the component – keywords, parentheses, dots, etc. Our raw match might look like `IF DOT WHILE RETURN RBRACE`[7]. While this is indeed the sequence of symbols that we have 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 10, which picks out just the `RBRACE`.

Figure 10: A high-level view of the reverse ϵ-NFA generated for a single meta-pattern.

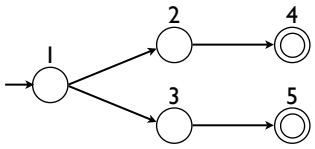
The backwards matching of meta-patterns is accomplished by building a DFA for the reverse of each meta-pattern. The process is the same as for lexical states except that there is only one meta-pattern in each ϵ-NFA and the self-loop is omitted.

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 11 shows a sample renumbering.



(a) Original Numbering



(b) Revised Numbering

Figure 11: Renumbering DFA states to move accepting states to the end.

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

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

5. CASE STUDIES

After we had built the first version of MetaLexer, we used it to specify lexers for three real programming languages: AspectJ (and extensions), MATLAB (and the AspectMatlab extension), and MetaLexer itself. We first did AspectJ and MATLAB, in order to refine and improve the MetaLexer specification, based on our experiences. Then, after we had finished improving MetaLexer, we implemented MetaLexer in itself. After completion, the AspectMatlab team used MetaLexer to create the lexer for AspectMatlab. These are all non-trivial systems, and we were very encouraged with our experiences and we hope that others will find MetaLexer equally useful.

5.1 AspectJ

The AspectBench Compiler, *abc* [2], is an extensible research compiler for AspectJ [20]. We used MetaLexer to construct a new lexer for the core AspectJ language of *abc* and two of its extensions: eaj (Extended AspectJ) and tm (Tracematches) (see [1] for details on these extensions).

The existing *abc* lexer [16] (written in JFlex) breaks AspectJ into four sub-languages: java, aspect, pointcut, and pointcut-if-expression. The first three correspond straightforwardly to parts of the AspectJ syntax. The last refers to the bits of aspect syntax that appear within if pointcuts. The nesting structure of these sub-languages is tracked on a stack, which is pushed and popped when certain tokens are observed.

In our MetaLexer implementation, we mimicked this approach. However, since the aspect and pointcut-if-expression sub-languages differed only in their transitions, we were able to capture their differences in the layout and eliminate the pointcut-if-expression component. The resulting specification for state transitions was very clear, and the key meta-lexer transitions are given in Listing 7. In this listing, three MetaLexer transitions (embeddings) are specified, named `perclause`, `declare` and `pointcut`.

Note that the syntax for the transition (embedding) rules is very simple – each embedding is just a listing of the following elements: `name`, `host`, `guest`, `start` and `end`. Optionally, a `pair` element can be specified when proper nesting of `start` and `end` symbols needs to be enforced.

To illustrate, let us consider the example of the `perclause` transition (embedding) defined in Listing 7. This transition specifies when the meta-lexer should transition from the `aspect_decl` sublanguage (the host) to the `perclause` sublanguage (the guest). The transition should trigger (start) when a sequence of meta-tokens composed of one of the "PER" tokens, followed by an LPAREN is recognized. The transition should go back to the host sub-language (end) when the matching RPAREN meta-token is recognized.

The `perclause` transition rule also illustrates the optional `pair` element. Many lexing situations actually require some notion of nesting. In this case, we only want to transition back when we find the matching RPAREN (which is not necessarily the first one). MetaLexer supports this through the `pair` directive.[8]

[8]Another common case for the use of pairs is the proper matching of begin comments and end comments. If one

```
%%embed
%name perclause
%host aspect_decl
%guest pointcut
%start [PERCFLOW PERCFLOWBELOW PERTARGET PERTHIS]
    LPAREN
%end RPAREN
%pair LPAREN, RPAREN

%%embed
%name declare
%host aspect
%guest pointcut
%start DECLARE
%end SEMICOLON

%%embed
%name pointcut
%host java, aspect
%guest pointcut
%start POINTCUT
%end SEMICOLON
```

Listing 7: Extract – Embeddings from aspectj.mll

Another significant advantage of the MetaLexer implementation was that we were able to replace the ad-hoc extensibility mechanism implemented in *abc*, which had been grafted onto the original lexer, with a more general and systematic approach.

For example, both the eaj and tm (tracematches) extensions were very easy to implement with MetaLexer. The eaj extension 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 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 9). Finally, the new embeddings were added to the extended layout.

```
%component eaj_aspect
%%
%%inherit eaj_global_keywords
%%inherit aspect
```

Listing 8: Extract – Adding New Global Keywords

```
%%inherit aspectj
%replace aspect, eaj_aspect
%replace aspect_decl, eaj_aspect_decl
%replace java, eaj_java
%replace java_decl, eaj_java_decl
%replace pointcut, eaj_pointcut
%replace pointcut2, eaj_pointcut2
```

Listing 9: Extract – Replacing Components

The tm extension 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 benefited from new lexical rules as well. However, since the previous lexer could not add new sub-languages, there was no way to 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.

We did encounter one noteworthy obstacle when specifying the *abc* lexer. Some of the embeddings (i.e. transitions) in *abc* need 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 some components and then giving the duplicates different transition rules.

5.2 MATLAB

The Sable Lab at McGill University is developing an optimizing compiler framework for scientific programming languages called McLab [28]. In its first incarnation, McLab is a compiler and virtual machine toolkit for MATLAB [24] and extensions of MATLAB.

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 LR parsing techniques. For this reason, the McLab team has defined a cleaner and functionally equivalent subset of the language, called Natlab.[9]

Originally, we specified the Natlab lexer using JFlex. However, the JFlex solution was both complex and not easy to extend. Thus, we respecified the lexer using MetaLexer, in order to get a cleaner and extensible solution for our McLab toolkit.

Re-specifying Natlab in MetaLexer resulted in three substantial improvements. First, the new lexer is extensible. Since the original development of the Natlab lexer, this extensibility has successfully been used by the AspectMatlab team. Second, nearly all of the action code in the JFlex lexer was eliminated in favour of MetaLexer language constructs. In fact, nearly all remaining code is for returning tokens, appending to string buffers, and raising lexical errors. Third, all lexical states were replaced by components. These improvements are particularly gratifying in light of Natlab's inherent complexity.

5.3 MetaLexer

To show our confidence in MetaLexer, we decided to specify the lexers for MetaLexer itself (i.e. the lexers for the layout and component languages) in MetaLexer. The benefits were largely as described above: the specification is clearer; all lexical states were eliminated; and the remaining action code is limited to return, append, and error. Additionally, rewriting the layout and component lexers in MetaLexer allowed the two languages to share many of their components and gave us ideas for improvements that were hard to see in the complexity of the original.

5.4 Experience Summary

Our experiences have shown that MetaLexer has captured the right abstractions for modular and clean specifications for three real-world applications, AspectJ, Matlab and MetaLexer itself. Both the AspectJ and Matlab examples have

wants to support nested comments, then a `pair` directive can be used. The `pair` directive is more expressive than pure lexical analysis, but of course not more expressive than the arbitrary action code that one routinely finds in lexers to handle precisely this kind of matching.

[9]In fact, it was the exercise of trying to define a clean lexer for MATLAB/Natlab that motivated this whole MetaLexer project.

also shown that MetaLexer enables modular specifications for language extensions. The MetaLexer example demonstrates reuse through the modular extensions, with component sharing between the two sub-languages.

6. RELATED WORK

There has been a lot of work on compiler-generator tools and many excellent approaches, each with its own strengths, have been proposed. If we concentrate on work related to extensible, modular, and composable lexing, this can be divided into two main categories: those with separate lexers and those without (i.e. with lexing and parsing integrated).

6.1 With a Separate Lexer

Traditionally, lexical analysis (i.e. tokenization) has been performed separately from parsing. The two primary motivations for the separation are performance and clarity. Performance is improved because characters can be grouped into tokens more quickly by the regular-expression-driven lexer than by the CFG-driven parser. Clarity is improved because separate tokenization allows the parser specification to be written at a much higher level – without comments or whitespace and with nice names for classes of tokens (e.g. 'number' or 'identifier').

At present, most extensible, modular, composable parsing systems with separate lexers use ad-hoc techniques for lexical analysis. There are, however, some exceptions.

6.1.1 Ad-Hoc

The Polyglot Parser Generator [7], developed by Brukman and Myers, is an extension of the popular CUP parser generator [31] that adds extensibility. Unfortunately, it provides neither composability, nor a corresponding lexer.

Ekman et al created JastAdd [12], an extensible, modular attribute grammar system. JastAdd can be used to create modular abstract syntax trees (ASTs) using any parser that can construct an AST from JastAdd-generated classes. Their JastAddJ extensible Java compiler [11] achieves some level of extensibility and modularity by breaking existing (lexer and parser) specifications into small fragments and then selectively concatenating them back together for each extension. Of course, the concatenation is not subject to any checks and there is no way to delete existing rules.

The *abc* extensible AspectJ compiler [2], developed by Avgustinov et al, uses an ad-hoc extensible lexer written in JFlex [16]. It handles the multi-language structure of AspectJ by breaking the language into four sub-languages: java, aspect, pointcut, and pointcut-if-expression. Actions attached to keywords tell the lexer when to switch sub-languages. Extensions of the lexer can only add or remove keywords and keyword actions.

6.1.2 ANTLR

The ANTLR parser generator [27], 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. Since hand-written lexers frequently match tokens that cannot be captured by regular expressions (especially bracketing), the Antlr lexer supports CFGs. Of course, this eliminates the performance benefit of using a regular-expression-based lexer.

Instead of lexical states, ANTLR uses semantic predicates – boolean expressions that enable and disable rules. This gives specification writers a powerful (if non-declarative) way to apply lexical rules contextually.

ANTLR allows both lexer and parser extension using delegation [26]. A lexer can inherit one or more existing lexers and will delegate to them, if none of its own tokens match. There does not appear to be a way to manipulate the relative priority of the new and inherited lexer rules. Xtext is a complete Eclipse-based system for generating editors and IDEs based upon ANTLR 3 [3].

6.1.3 DSL-Specific

Some approaches are specifically tailored to rapid development of domain-specific languages (DSLs). Systems like MontiCore [22], [15] and MPS (the Meta Programming System) [10] 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 general multi-language programming like *asm* blocks or JSP.

6.2 Integrated Solutions with no Separate Lexer

Of course, the easiest way to make lexical analysis context-sensitive is to integrate it with the (context-sensitive) parser. If the parser is top-down, then this is accomplished by specifying lexical rules for terminals. If the parser is bottom-up, then all terminals are characters (which amounts to returning a token for each character).

6.2.1 Top-Down

The Rats! parser generator [14], created by Robert Grimm, is a lexerless approach to extensible, modular, and composable parsing (e.g. Jeannie [17]). Rats! discards CFGs in favour of parsing expression grammars (PEGs). PEG specifications look like normal CFG specifications, but productions are tested in order and the parser backtracks over mismatches as needed. Rats! terminals are either individual characters or strings matched by hand-written recognition methods. There is no need for lexical states as tokenization is handled separately for each non-terminal. Rats! has a powerful module system that takes advantage of the composability of PEGs. Unfortunately, all this power comes at the expense of performance – Rats! is slower than ANTLR, which is slower than LALR parsers [14].

Another system based upon PEGs is Neverlang, which supports a very modular specification of individual language components and support code [9].

The metafront system [4], developed by Brabrand et al, is another top-down lexerless parsing system. It accomplishes context-dependent lexical analysis by associating a regular expression with each terminal. In place of CFGs, metafront uses "specificity grammars", which resolve ambiguity in favour of the "most specific" alternative. Since specificity can be used to resolve any ambiguity, metafront parsers are both extensible and composable.

6.2.2 Bottom-Up

The Copper lexer and parser generator [34], created by Van Wyk and Schwerdfeger, uses a modified LALR parser to pass context information to a DFA-based lexer. (We consider it to be a lexerless approach since the lexer does not operate independently of the parser.) Basically, the lookahead information in the parser is used to tell the lexer what

subset of its lexical rules it should attempt to match. While this approach retains the benefits of DFA tokenization, its lexing is no longer constant with respect to the size of the grammar because it depends on the sizes of the grammar's lookahead sets. Furthermore, special care must be taken with lookahead tokens as, under certain circumstances, they may need to be re-lexed after an LALR reduction.

The Copper system has recently been extended to handle syntactic composition in a deterministic way [29].

Generalized LR (GLR) parsing is an alternative approach that provides a graceful way to handle conflicts in LR grammars (i.e. shift-reduce or reduce-reduce). Instead of raising an error like a traditional LR parser, a GLR parser constructs all possible parse trees. It may return all of them or it may apply a heuristic or a hand-written decision function to weed out undesirable parse trees. Unlike traditional general CFG parsing, which is cubic in the size of the input, GLR has linear execution time for conflict-free grammars – it only slows down around ambiguities.

On its own, GLR does not address the problem of composable parsing. However, it can be extended to scannerless GLR (SGLR), in which the parser uses characters rather than tokens as terminals. Since the class of CFGs is closed under composition, it is easy to create extensible, composable SGLR grammars. Furthermore, SGLR makes it very straightforward to perform context-sensitive lexical analysis. For example, Bravenboer and Visser recommend SGLR for embedding domain-specific languages (DSLs) in general-purpose programming languages [6]. Along similar lines, Kats et al have used SGLR to support create rich editors for multi-language programming in Eclipse [18]. Even more relevantly, Bravenboer et al have recommended using SGLR in the *abc* frontend [5].

6.3 The Position of MetaLexer

Although related techniques have many strengths, we believe that MetaLexer provides an elegant solution to a very common situation. MetaLexer is a good fit for projects built using a standard LR parsing tool, and which could benefit from a more modular and extensible lexer than is easily specified with standard tools like JFlex. This is particularly true for lexers which support several sub-languages. Since MetaLexer uses much of the same notation as JFlex, users who are already comfortable with JFlex should be able to easily adapt and make use of the MetaLexer features.

7. CONCLUSIONS AND FUTURE WORK

The idea of creating compilers for languages with extensible syntax and compilers for mixed language programming is growing in popularity. Numerous tools have sprung up for extensible and composable parsing, attribute grammars, and analyses, but still there is a gap. For developers using traditional parser generators like yacc/bison or extensible compiler frameworks like Polyglot and JastAdd, there was no existing tool supporting modular and extensible lexers. We were faced with this problem both in the development of the *abc* framework for AspectJ and in the development of our new extensible McLab system for MATLAB. In both cases we had elegant solutions for extensible parser and semantic analyses through the use of Polyglot or JastAdd, but we did not have elegant solutions for lexing.

Without MetaLexer the developer would either have to develop a project-specific extensibility mechanism for the lexer (which is eventually what the *abc* project did) or each extension would have to have a complete copy of the lexer rules (the initial solution for *abc*), with lexing changes and additions made to that copy. Clearly such a copy and edit process is not good software development practice.

To fill this gap, we developed the MetaLexer lexical specification language. It has two 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. We found code sharing to be quite useful when we implemented MetaLexer in itself, since approximated a third of the modules could be shared between the specifications for the component and layout languages.

We implemented a translator from MetaLexer to the popular JFlex lexical specification language and used it to create lexers for three real programming languages: AspectJ (and two extensions), MATLAB and the AspectMatlab extension, and MetaLexer itself. The new specifications are easier to read and require much less action code than the originals. Furthermore, rewriting JFlex specifications in MetaLexer enabled us to see new solutions to existing lexer problems.

In its current state, MetaLexer is already a useful tool. However, there is always room for improvement. Given that we now have a fully functional version, we think it would be worthwhile to optimize the MetaLexer compiler itself, as well as the generated code. It would also be worthwhile to create creating a second code generation engine that produces Flex [13] specifications so that C++ programs can also make use of MetaLexer. The similarity between Java and C++ and between JFlex and Flex should make this straightforward.

Now that we have a composable lexer, it would be interesting to investigate a corresponding composable LR parser generation approach. Ideally, it would, like MetaLexer, be a preprocessor for existing tools. With or without a composable LR parser generator, it would be interesting to compare the combination of MetaLexer and LR to scannerless approaches like SGLR and PEGs. 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. The work of Bravenboer et al, expressing the syntax of abc in SGLR [5], presents an excellent opportunity for comparison. Furthermore, additional work will be required to determine how often the combination of MetaLexer and LALR is "good enough".

Acknowledgments.

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

8. REFERENCES

[1] AspectBench Group. abc extensions. `http://www.aspectbench.org/extensions`.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc : An

extensible AspectJ compiler. In *AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM.

[3] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhlein, K. Wannheden, S. Zarnekow, and contributors. XText user guide. `http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf`, 2010.

[4] C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The metafront system: Extensible parsing and transformation. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications, LDTA 2003*, volume 82(3), pages 592–611, 2003.

[5] M. Bravenboer, Éric Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 209–228, New York, NY, USA, 2006. ACM.

[6] M. Bravenboer and E. Visser. Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In *OOPSLA 2004: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, New York, NY, USA, 2004. ACM.

[7] M. Brukman and A. C. Myers. PPG: a parser generator for extensible grammars. `http://www.cs.cornell.edu/Projects/polyglot/ppg.html`.

[8] A. Casey. The MetaLexer lexer specification language. Master's thesis, McGill University, Montreal, QC, CA, 2009.

[9] W. Cazzola and D. Poletti. Dsl evolution through composition. In *Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, page to appear, 2010.

[10] S. Dmitriev. Language oriented programming: The next programming paradigm. `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`.

[11] T. Ekman and G. Hedin. The Jastadd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[12] T. Ekman and G. Hedin. The JastAdd system: Modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, Dec 2007.

[13] GNU. Flex: The fast lexical analyzer. `http://www.gnu.org/software/flex/`.

[14] R. Grimm. Better extensibility through modular syntax. In *PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[15] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. MontiCore: A framework for the development of textual domain specific languages. In *ICSE Companion 2008: Companion of the 30th International Conference on Software Engineering*, pages 925–926, New York, NY, USA, 2008. ACM.

[16] L. Hendren, O. de Moor, and A. S. 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`.

[17] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. *ACM SIGPLAN Notices*, 42(10):19–38, 2007.

[18] L. Kats, K. T. Kalleberg, and E. 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`.

[19] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001: Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[21] G. Klein. JFlex: The fast scanner generator for Java. `http://jflex.de/`.

[22] H. Krahn, B. Rumpe, and S. Völkel. Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[23] M. Lesk and E. Schmidt. Lex: A lexical analyzer generator. Comp. Sci. Tech. Rep. 39, Bell Laboratories, Oct 1975.

[24] MathWorks. Matlab. `http://www.mathworks.com/products/matlab/`.

[25] Microsoft. ASP. `http://www.asp.net/`.

[26] T. Parr. Composite grammars. `http://www.antlr.org/wiki/display/ANTLR3/Composite+Grammars`.

[27] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, Raleigh, NC, 2007.

[28] Sable Lab. Mclab. `http://www.sable.mcgill.ca/mclab/`.

[29] A. C. Schwerdfeger and E. R. V. Wyk. Verifiable composition of deterministic grammars. In *PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–210, New York, NY, USA, 2009. ACM.

[30] Sun Microsystems. JSP. `http://java.sun.com/products/jsp/`.

[31] Technische Universität München. CUP: LALR parser generator in Java. `http://www2.cs.tum.edu/projects/cup/`.

[32] W3C. Cascading Style Sheets. `http://www.w3.org/Style/CSS/`.

[33] W3C. Scripts in HTML4. `http://www.w3.org/TR/html4/interact/scripts.html`.

[34] E. R. V. Wyk and A. C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 63–72, New York, NY, USA, 2007. ACM.