MetaLexer User Manual

Andrew Casey – 260 260 558

February 6, 2009

Contents

List of Figures

Listings

Before Beginning

This manual is intended for users who have some familiarity with JFlex or with a similar system (e.g. JLex, lex, flex, etc). Those without such a background are encouraged to visit *http://jflex.de/manual.html* before proceeding.

1 Concepts

MetaLexer is a new lexer specification language based on JFlex. It aims to be more extensible and modular than traditional lexer specification languages.

Metalexer has two key features:

1. Lexer state transitions are lifted out of semantic actions.
2. Modules support multiple inheritance.

1.1 State Transitions

Lexers for non-trivial languages nearly always make use of lexer 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* shows a 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. The previous state is stored so that the lexer can return once the closing quote has been seen.

Listing 1: JFlex State Transitions

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

As in *Listing 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 lexer state is stored so that it can be returned to 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.

For example,

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

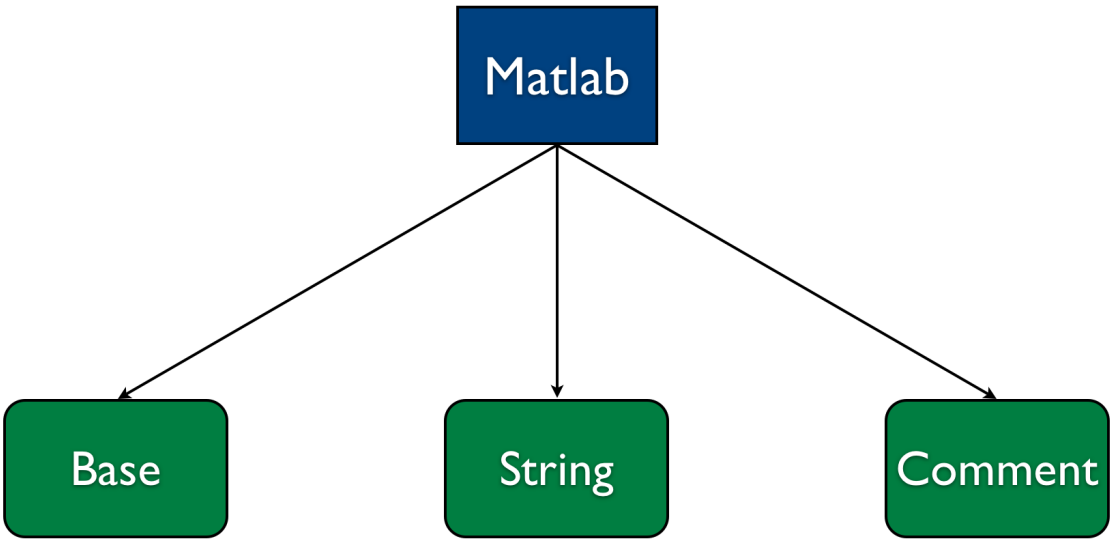
MetaLexer makes these rules explicit by associating "meta-tokens" with rules and then using a "meta-lexer" to match patterns of meta-tokens and trigger corresponding transitions. This organization gives rise to two different types of modules: *components* and *layouts*.

A *component* contains rules for matching tokens. It corresponds to a single lexer 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 (blue) layout – *Matlab* – refers to three (green) components – *Base*, *String*, and *Comment*. Each of the components describes a lexer state and the layout describes their interaction.

Figure 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 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.

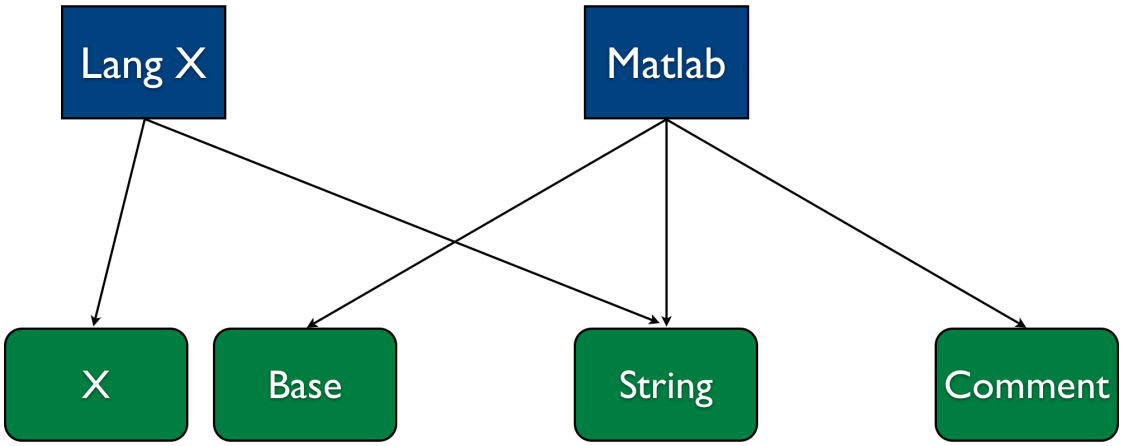
1.2 Inheritance

MetaLexer uses multiple inheritance to achieve extensibility and modularity.

For example, *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. 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 4* shows how inheritance can improve modularity by factoring out useful

Figure 2: Two layouts sharing components



"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.

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 re-arrangement of existing files. In particular, common ancestors are not shared but duplicated.

Both layouts and components support multiple inheritance.

2 Syntax

MetaLexer actually consists of two specification languages – one for components and one for layouts.

2.1 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.

See *Listing 2* for the text of a sample component.

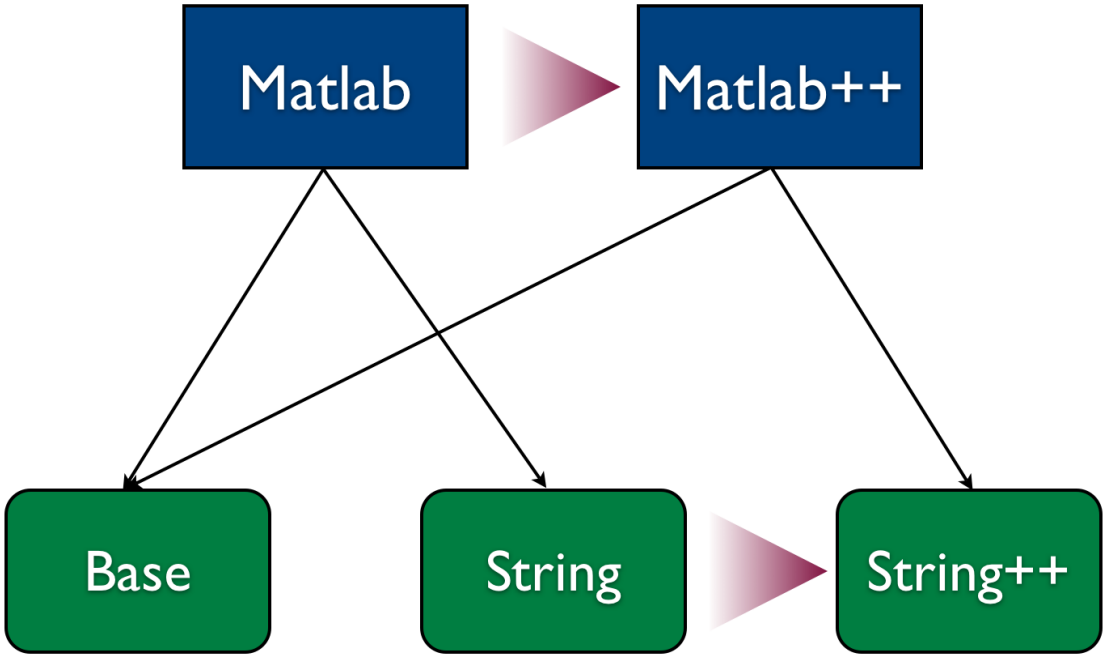
2.1.1 Option Section

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.

%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).

%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.

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



Listing 2: Component Syntax Example – sample.mlc

```
1  %component sample
2  %extern "void error(String) throws ScannerException"
3  %extern "Symbol symbol(short, String)"
4  identifier = [a-zA-Z][a-zA-Z0-9]*
5  %%
6  keyword1 {: return symbol(KEYWORD1, ""); :} KEY1
7  keyword2 {: return symbol(KEYWORD2, ""); :} KEY2
8  %:
9  {identifier} {: return symbol(ID, yytext()); :} ID
10 %:
11 <<ANY>> {: error("Unexpected char: " + yytext()); :}
```

%state *name, name, ...* – ANY NUMBER – This directive declares a number of inclusive states. These states will include all rules tagged with the state name as well as all untagged rules. An inclusive state called *YYINITIAL* is declared by default. **This is an advanced directive and should not be used under normal circumstances.**

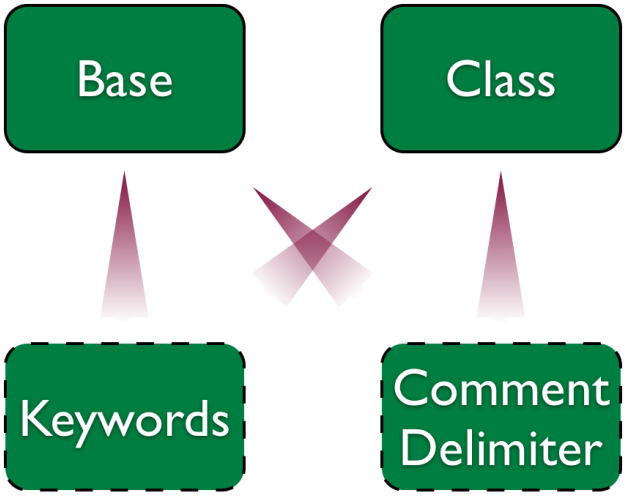
%xstate *name, name, ...* – ANY NUMBER – This directive declares a number of exclusive states. These states will include only rules tagged with the state name. **This is an advanced directive and should not be used under normal circumstances.**

%start *name* – AT MOST 1 – 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. **This is an advanced directive and should not be used under normal circumstances.**

%import *"class"* – ANY NUMBER – This directive indicates that the top-level lexer should import/include/require (depending on the action language; e.g. C for Flex, Java for JFlex, etc) the specified class/module/file.

%extern *"signature"* – ANY NUMBER – This directive indicates that the component expects any

Figure 4: Using inheritance to improve modularity



layout making use of it to provide an entity with the specified signature. In particular, the layout must include **%declare** *"signature"*.

%{ *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.

%append{ *method code* **%append}** – AT MOST 1 – An append block is both a directive and a code block. 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 3.6* 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)*.

%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.

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

2.1.2 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*

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.

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

%%inherit *component*

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 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, indicating that the pattern should be sought in the default *YYINITIAL* 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 rule 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 3.4* for the importance of and the rationale behind this distinction between different types of rules.

2.2 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.

See *Listing 3* for the text of a sample layout.

2.2.1 Local Header

The local header is a block of freeform text that will be inserted at the top of the generated lexer class (i.e. the file generated by the underlying lexer (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.

Listing 3: Layout Syntax Example – sample.mll

```
package sample;
%%
import beaver.*;
%%
%layout sample
%option "%line"
%option "%column"
%option "%class SampleScanner"
%option "%function nextToken"
%component base
%component string
%start base
%declare "Symbol symbol(short, Object)"
%{
private Symbol symbol(short type, Object value) {...}
%}
%%
%%embed
%name string_embed
%host base
%guest string
%start START_STRING
%end END_STRING
```

2.2.2 Inherited Header

The inherited header is another block of freeform 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.

2.2.3 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).

%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).

%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.

%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.

%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"*.

%{ *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.

%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.

2.2.4 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*
%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 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 *layout*

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.

2.3 Comments

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

3 Semantics

MetaLexer handles matching of component rules and execution of actions in the same way as JFlex. The new features, relating to inheritance and abstracted state transitions are described below.

3.1 Scoping

Each component is a separate scope. Code regions, macros, and rules are invisible to other components, so there is no possibility of a name collision.

The code regions in a layout are visible to all components.

MetaLexer does not have visibility modifiers. An inheriting module (i.e. component or layout) has access to everything in its parent.

3.2 Inheritance

Inheritance in MetaLexer is governed by two key concepts.

1. A module is finalized before it is inherited.
2. If anything is duplicated, then the first occurrence is dominant.

A module is finalized if it has incorporated code from all parents (which are, in turn, finalized) and performed all error checks.

Duplicates occur frequently as a result of inheritance. Macros, invocations, component rules, embeddings, options, and other constructs can be duplicated as a result of inheritance. If this occurs, then the first definition will dominate. For example, if a layout has an embedding E and it inherits from a layout that also has an embedding E then there will be a conflict. The conflict is resolved by checking which occurs first, textually. If the **%%inherit** directive comes before the embedding, then the embedding from the parent replaces the embedding from the child. Otherwise, the embedding from the child replaces the embedding from the parent.

It follows that the locations of **%%inherit** directives are quite important. The contents of the inherited module are inserted after anything the directive follows textually and before anything it precedes textually, including the contents of other inherited modules. Note that this applies to headers and options as well. The header and option sections are concatenated in according to textual order – the local module is first, followed by the inherited modules.

The final result of the inheritance process is a single flat layout (i.e. without inheritance) referring to one or more flat components.

3.3 %helper

If a module is flagged **%helper**, then some of its error 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.

Note that the **%helper** directive is not inherited.

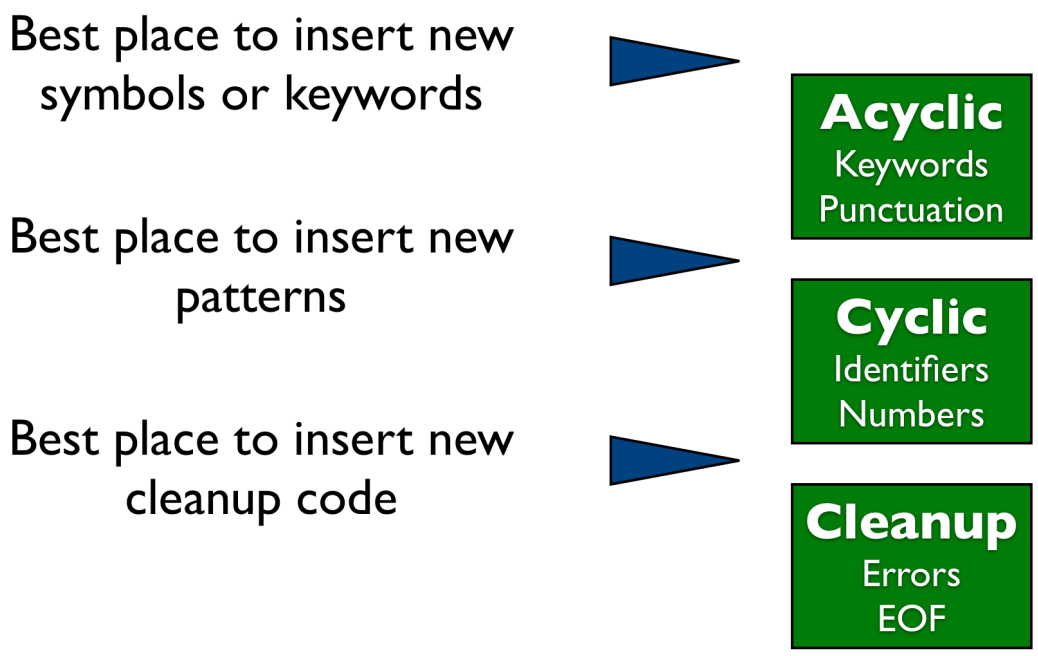
3.4 Component Rule Types

At first glance, it is not clear why component rules should be broken into categories: acyclic, cyclic, and cleanup. There are two main reasons.

First, these categories correspond neatly to the most commonly used 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.

Second, 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 5* 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 5: Rule type boundaries as insertion points



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.

The rules in a component file should not be read in textual order. Rather, they should be regarded as appearing in the following order.

1. Local acyclic rules preceding the first **%%inherit** directive.
2. Acyclic rules from the first inherited component.
3. Local acyclic rules preceding the second **%%inherit** directive.
4. Acyclic rules from the second inherited component.
5. etc
6. Local cyclic rules preceding the first **%%inherit** directive.
7. Cyclic rules from the first inherited component.
8. etc
9. Local cleanup rules preceding the first **%%inherit** directive.
10. Cleanup rules from the first inherited component.
11. etc
12. Local acyclic rules following the last **%%inherit** directive.
13. Local cyclic rules following the last **%%inherit** directive.
14. Local cleanup rules following the last **%%inherit** directive.

This conceptual rearrangement is illustrated by *Figures 6 & 7*. *Figures 6* shows a component file with rules before and after an inherited component. *Figures 7* shows the conceptual order of the rules after inheritance.

3.5 Meta-Lexing

Meta-lexing is straightforward. At any given time, the lexer is in both a component and an embedding.

Initially, the lexer is in the start component (as indicated by the layout) and the degenerate initial embedding (which has no end meta-pattern or pair filter).

When a component rule with a corresponding meta-token is matched, the meta-token is passed to the pair filter for the current embedding. If it 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.

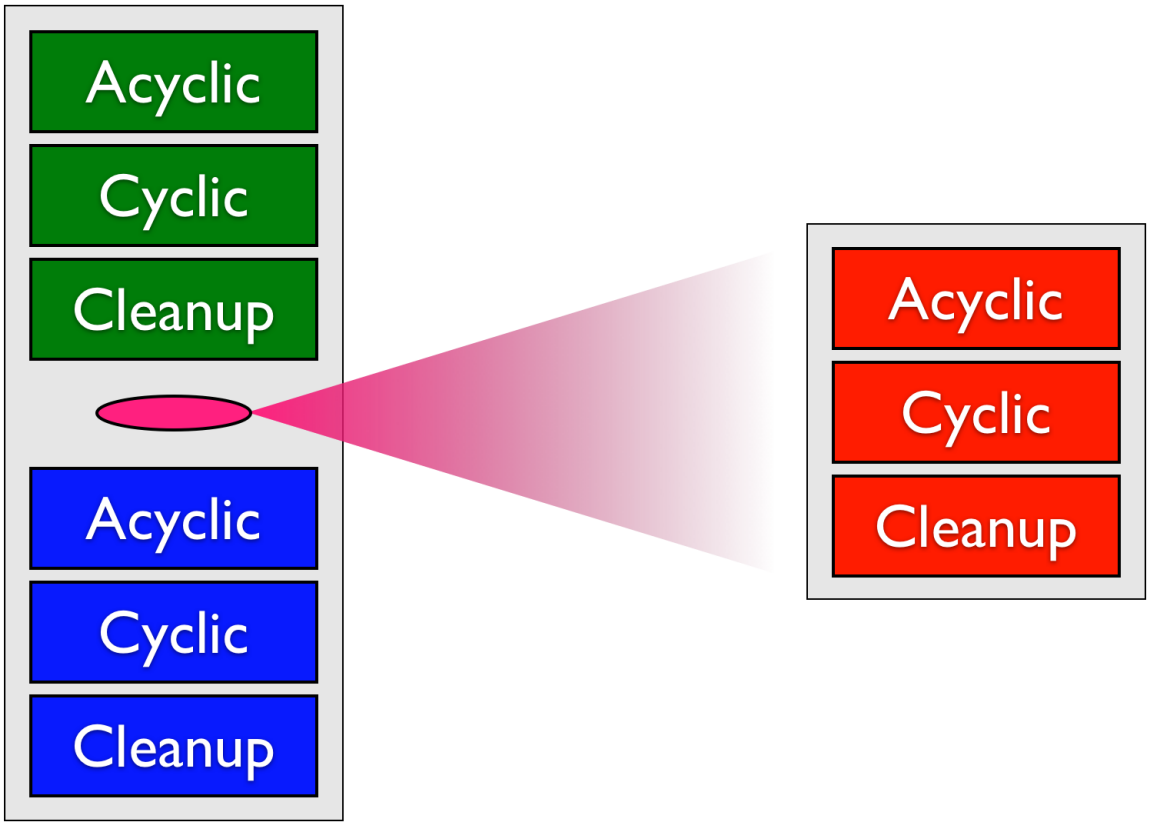
In the layout, the meta-token is tested against **%start** sequences of embeddings hosted by the current component and the **%end** sequence of the current embedding. As in the lexer, the longest match wins. If there is a tie, then the **%start** sequence is preferred. If there is still a tie, then the textually-first **%start** sequence is preferred.

When a **%start** sequence is matched, the meta-lexer transitions to the guest component of the containing embedding.

When an **%end** sequence is matched, the meta-lexer transitions back to the host component of the containing embedding. This generates a region meta-pattern for the completed component, which is processed in the same way as a meta-token, possibly starting or ending additional embeddings.

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

Figure 6: Rules in textual order



3.6 %append

As mentioned above, **%append** has two meanings. First, it makes available an *append(String)* method. Second, it triggers calls to a special append action function when appropriate.

A component that inherits from an append component is also an append component. As with other inherited items, a local version is preferred, followed by the inherited versions in textual order.

The append buffer (i.e. the buffer built up by *append(String)*) is reset whenever the lexer transitions from an append component to a non-append component.

The append action method is called whenever the lexer performs an end-transition from an append component guest back to a non-append component host.

Note that if both the host and the guest of an embedding are append components, then the guest will continue to append to the same buffer and the append action method will not be called.

There is one exception to these general rules. As mentioned above, on an end-transition, a special region meta-pattern is triggered. This may trigger another end-transition, followed by another region meta-pattern, etc. Though the rules above might require many buffer resets and append action calls as a result of all these transitions, it is clear that only the first call to the append action method can receive a non-empty buffer. As such, only the first call will be made - the others will be suppressed.

4 Execution

At present, the MetaLexer compiler supports two output languages: MetaLexer and JFlex.

4.1 MetaLexer-to-MetaLexer Translator

The MetaLexer-to-MetaLexer translator is useful for testing specifications. It will perform all syntactic and semantic checks as well as processing all component and layout inheritance before re-outputting the specification in new MetaLexer files. The output will consist of exactly one layout and a number of independent components which it includes.

If you have a copy of metalexer-metalexer.jar, you can execute the jar directly with 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 MetaLexer files.

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

If you have an unjarred copy of the compiler, you can run the main class metalexer.metalexer.ML2ML with the same arguments.

For example, one might run `java metalexer.metalexer.ML2ML natlab /home/userA/src /tmp`.

This approach is quite a bit more complicated as it requires manual configuration of the classpath.

4.2 MetaLexer-to-JFlex Translator

The MetaLexer-to-JFlex translator converts a MetaLexer specification into a pair of JFlex specifications that can be compiled, without external dependencies, into a JFlex scanner for the specified language.

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

If you have a copy of metalexer-jflex.jar, you can execute the jar directly with 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`.

If you have an unjarred copy of the compiler, you can run the main class metalexer.jflex.ML2JFlex with the same arguments.

For example, one might run `java metalexer.metalexer. ML2JFlex natlab /home/userA/src /tmp`.

This approach is quite a bit more complicated as it requires manual configuration of the classpath.

Figure 7: Rules in conceptual order