MetaLexer Developer Manual

Andrew Casey - 260 260 558

January 20, 2009

1 Organization

The files of the MetaLexer project are organized as follows.

1.1 metalexer/

.classpath & .project Eclipse project files.

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

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

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

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

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

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

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

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

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

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

1.2 metalexer/src/ & metalexer/test/

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

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

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

1.3 metalexer/src/frontend/

metalexer Java package.

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

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

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

Component & LayoutErrors.jrag The specifications of getErrors for components and layouts, respectively. These files delineate all the possible (frontend) errors that can be raised.

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

1.4 metalexer/src/frontend/metalexer

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

Constants Some helpful constants.

FileLoader Handles loading of components and layouts from files.

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

1.5 metalexer/src/backend-metalexer/

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

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

1.6 metalexer/src/backend-jflex/

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

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

_MetaGeneration.jadd Hierarchically called from **LayoutGeneration.jadd**, under **LayoutMetaGeneration**. Generate the .macro.flex file output by the translator (i.e. the meta-lexer).

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

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

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

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

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

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

1.7 metalexer/test/frontend/

c_scanner.testlist List of component language scanner tests (input in .in, expected output in .out). Output files contain a list of tokens up to either EOF or the first error.

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

c_parserfail.testlist List of component language parser tests (input in .in, expected output in .out). Output files contain lists of errors.

c_inheritancepass.testlist List of component language inheritance tests (input in .mlc, expected output in .out). Output files contain collapsed (i.e. post-inheritance), pretty printed versions of input files.

c_inheritancefail.testlist List of component language inheritance tests (input in .mlc, expected output in .out). Output files contain lists of errors.

c_error.testlist List of component language error tests (input in .mlc, expected output in .out). Output files contain lists of errors.

l_scanner.testlist List of layout language scanner tests (input in .in, expected output in .out). Output files contain a list of tokens up to either EOF or the first error.

l_parserpass.testlist List of layout language parser tests (input in .in, expected output in .out). Output files contain pretty printed versions of input files.

l_parserfail.testlist List of layout language parser tests (input in .in, expected output in .out). Output files contain lists of errors.

l_inheritancepass.testlist List of layout language inheritance tests (input in .mll, expected output in .out). Output files contain collapsed (i.e. post-inheritance), pretty printed versions of input files.

l_inheritancefail.testlist List of layout language inheritance tests (input in .mll, expected output in .out). Output files contain lists of errors.

l_error.testlist List of layout language error tests (input in .mll, expected output in .out). Output files contain lists of errors. Note: l_error_helper_ tests check the effect of the %helper directive on errors.

metalexer/ Java package. Contains JastAdd files (in contrast to **src**) to keep them separate from the test input and output files.

... Test specifications (as listed in .testlist files).

1.8 metalexer/test/frontend/metalexer/

Component & LayoutPrint.jrag Pretty printers for testing. Produce an output that is useful for testing. Not guaranteed to produce correct MetaLexer (e.g. may include helpful, but illegal annotations).

FrontendTests The top-level JUnit test suite.

_TestBase Class to be extended by generated test file.

_TestGenerator Class that generates a test file from a .testlist file.

_TestTool Class for generating .out files automatically.

... Support files for the Base, Generator, and Tool classes.

1.9 metalexer/test/backend-metalexer/

metalexer/metalexer/ Java package containing the top-level JUnit test suite, **BackendMetalexerTests**.

in/ Metalexer specifications.

out1/ Files from **in** that have been run through the pretty printer.

out2/ Files from **out1** that have been run through the pretty printer. Contents should be identical to those of **out1**.

1.10 metalexer/test/backend-jflex/

c_error.testlist List of component language (JFlex-specific) error tests (input in .mll, expected output in .out). Output files contain lists of errors.

c_error.testlist List of layout language (JFlex-specific) error tests (input in .mll, expected output in .out). Output files contain lists of errors.

m_c_meta.testlist List of tests. Each test has a .in file list of meta-tokens and regions to be passed to the generated meta-lexer for the component language specification and a .out file listing the expected embedding transitions returned by the meta-lexer.

n_scanner.testlist List of natlab language scanner tests. Copied from McLab but modified by removing lines passed through the comment buffer (i.e. preceded by '#' in the original).

in/ Specifications of various languages given in MetaLexer.

out/ The output from translating the specifications in **in/** to JFlex: lexer and meta-lexer JFlex files, properties files recording the characters assigned to regions and meta-tokens and the numbers assigned to embeddings, and Java scanners output by JFlex.

out/bin/ The class files that result from compiling the Java scanners output by JFlex. Note: this directory is on the classpath so that these scanners can be tested by the JUnit test suite.

placeholders/ Stubs of Java classes that contain just enough to make the generated scanners work. Most importantly, the parser stubs contain the list of tokens on which the scanners depend. Note: subdirectories are all Java packages and are on the classpath.

metalexer/jflex/ Java package. Contains JastAdd files to keep them separate from test intput files.

1.11 metalexer/test/backend-jflex/metalexer/jflex/

BackendJFlexTests The top-level JUnit test suite.

CompilationTests Arguably the most important test cases. Compiles the MetaLexer specifications in **in** to JFlex, to Java, to class files. Depended upon by the tests that exercise the generated scanners. Note: when running in Eclipse, it is necessary to run the tests more than once, refreshing in between, because Eclipse won't pick up the changes made by this test during a single run.

_Tests Handwritten (vs generated) test cases.

_TestBase Class to be extended by generated test file.

_TestGenerator Class that generates a test file from a .testlist file.

_TestTool Class for generating .out files automatically.

JFlexHelper A helper class that loads the JFlex jar at runtime if it is present.

ReflectionHelper A helper class that loads the generated scanner files at runtime if they have been generated (i.e. if JFlex is available).

... Support files for the Base, Generator, and Tool classes.

2 JFlex

Unfortunately, JFlex is covered by a GPL license. Since the MetaLexer is covered by a BSD-style license, the MetaLexer distribution cannot contain JFlex.

MetaLexer will function properly without JFlex, but you will be unable to rebuild any modified .flex files and the compilation-based tests in the JFlex backend will be unavailable.

If you wish to download JFlex on your own, it is available at *http://jflex.de/*. Just unzip the archive under the lib/ directory and point `jflex.jar.path.prop` in *common.properties* at the jar file (path should be relative to lib/).

3 Configurations

The MetaLexer compiler supports multiple backend with a shared frontend. The current implementation supports MetaLexer-to-MetaLexer and MetaLexer-to-JFlex.

Unfortunately, since the project uses aspects, there is interference between the backends. As a result, only one can be built and developed at a time. One benefit of this approach is that it keeps the release jars small by excluding code required for other backends.

To work solely on the frontend, use the **common.xml** build file. To work on the MetaLexer backend and the frontend, use the **metalexer.xml** build file. To work on the JFlex backend and the frontend, use the **jflex.xml** build file.

4 Building MetaLexer

MetaLexer developers can use either the command line or the Eclipse IDE.

4.1 Command Line

When executing the build files from the command line, ensure that the *eclipse.running* property is not set. If it is, then the Java compilation steps will be skipped.

The Ant build file takes care of all classpath issues. No configuration should be required.

4.2 Eclipse

To build the project in Eclipse, use the same Ant targets as from the command line, but ensure that the *eclipse.running* property is set. This will allow Eclipse to build Java files that are in source path folders in the Eclipse build path.

The provided **.classpath** file takes care of all source- and classpath issues. No configuration should be required.

When running the JUnit tests in the JUnit view, note that the frontend and backend tests must be run separately. (In contrast, the *test* Ant target runs all appropriate tests in a single pass.)

If you want a really minimal release jar, you have to build it from the command line because otherwise Eclipse will compile some extraneous Java files and they will be included.