Creating MaSH Programming Environments

Andrew Rock
School of Information and Communication Technology
Griffith University
Nathan, Queensland, 4111, Australia
a.rock@griffith.edu.au

November 29, 2010

1 Introduction

This document defines the MaSH environment creation language.

The MaSH compiler `mashc` works by parsing a MaSH program, in a `.mash` file, and rewriting it as a `.java` file. To create a valid Java source, the code must be encapsulated in a class. If the MaSH program is written as just statements, not encapsulated in methods, then all those statements will need to be wrapped in a method, usually `main`. A MaSH environment must define how to wrap the code in a class, and how to wrap statements in a main method. As well, the environment presents the complete, non-object-oriented API available for MaSH programs using this environment.

The MaSH compiler has only one option that affects code generation: `+debug`. This option injects extra calls into the output Java code. An environment may optionally include a debugger object that accepts these calls and presents an interactive debugger interface.

The best way to learn how to write environments is to look at the source code for existing environments.

2 Lexical syntax

The MaSH environments lexical syntax describes how to make up the tokens that make up the MaSH language for specifying environments.

2.1 Kinds of characters

As per MaSH programs.

2.2 Comments

As per MaSH programs.

2.3 Identifier and keywords

2.3.1 Identifiers

As per MaSH programs.

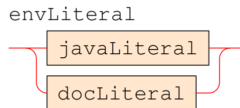
2.3.2 Keywords

The following additional keywords appear in MaSH environments only: `environment`; `rewrite`; `member`; `prelude`; `postlude`; `inline`; `purpose`; `precondition`; `section`.

2.4 Literals

There are only two types of literals in MaSH environments.

```
envLiteral ::= javaLiteral | docLiteral.
```

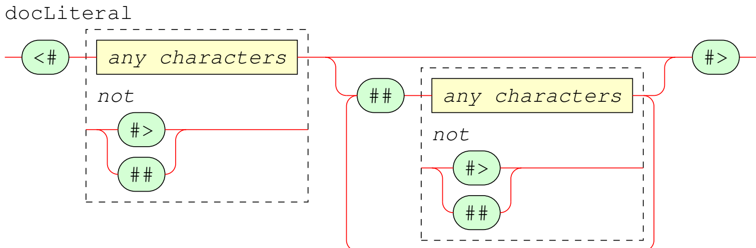


A documentation literal is some text that is to be output as is to the LaTeX documentation files produced by `mashdoc`. It is delimited by `<#` and `#>`. within it are parts separated by `##` sequences. The first part should contain plain text marked up with LaTeX commands. The second part is for plain text marked up with HTML. `mashdoc` will use the second part to generate HTML output instead of the first part. If there is no second part it will translate the LaTeX part into HTML. The translator is not complete, but is adequate for most of the markup that occurs in typical program documentation. It is preferable to use only one part if you can. Any more parts after the first two are currently ignored.

```
docLiteral ::=
   "<#" <$any characters$ ! ("#>" | "##")>
   {"##" <$any characters$ ! ("#>" | "##")>} "#>".
```

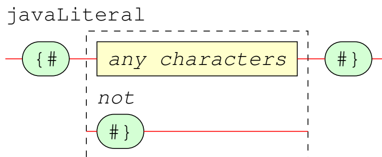
pattern	*option*	*replaced with*
`#PROGRAM_NAME#`		the base file name of the MaSH program, which is intended to be the name of the output Java class
`#ENVIRONMENT_NAME#`		the name of the MaSH environment that has been imported by the MaSH program
`#OPEN_DEBUGGER#`	`-debug`	
	`+debug`	`mash_debugger.open();`
`#CLOSE_DEBUGGER#`	`-debug`	
	`+debug`	`mash_debugger.close();`
`#SUSPEND_DEBUGGER#`	`-debug`	
	`+debug`	`mash_debugger.suspend();`
`#RESUME_DEBUGGER#`	`-debug`	
	`+debug`	`mash_debugger.resume();`
`#UPDATE_DEBUGGER#`	`-debug`	
	`+debug`	`mash_debugger.update();`

Table 1: Textual substitutions performed on all java literals.



A java literal is a fragment of Java code that is to be included as is (after some textual substitutions) to the Java target output by `mashc`. It is delimited by `{#` and `#}`. The textual substitutions are summarised in table 1.

```
javaLiteral ::= "{#" < $any characters$ ! "#}"> "#}".
```



2.5 Separators

As per MaSH programs.

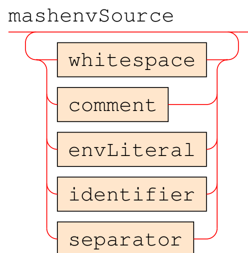
2.6 Operators

None. All operators are embedded in Java literals.

2.7 A MaSH environment input source

A MaSH environment source is made up of a sequence of these elements in any order (as far as the lexical syntax is concerned): whitespace; comments; literals; identifiers (including keywords); and separators.

```
mashenvSource ::= {   whitespace | comment | envLiteral
                    | identifier | separator}.
```



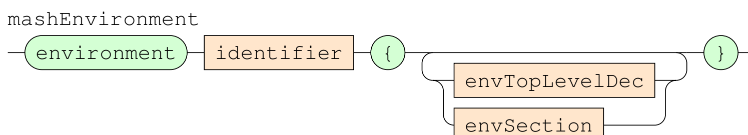
3 Context-free grammar

This is the context-free grammar for MaSH environments.

3.0.1 MaSH environments

A MaSH environment is headed by the keyword **`environment`** and the name of the environment. The name should start with a lower case letter (like Java packages, because they get imported with the same word) and be the same as the base file name. The sequence of top level declarations and sections in the environment is enclosed by braces.

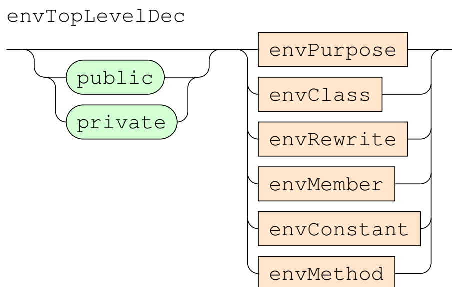
```
mashEnvironment ::= "environment" identifier
   "{" {envTopLevelDec | envSection} "}".
```



Top level declarations

A top level declaration is optionally preceded by either of the visibility keywords `public` or `private`. These only control which top level declaration will be exposed by `mashdoc`. There is an appropriate visibility default for each kind of top level declaration. Each top level declaration is of either a purpose documentation insert or one of the kinds of environment elements.

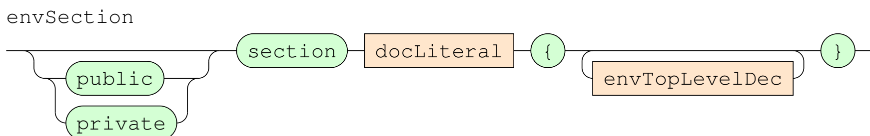
```
envTopLevelDec ::= ["public" | "private"]
   ( envPurpose | envClass | envRewrite | envMember
     | envConstant | envMethod).
```



Sections

Top level declarations may optionally be partitioned into sections. This only affects the organization of the documentation. Sections may be `public` or `private`. They have a heading followed by top level declarations enclosed in braces.

```
envSection ::= ["public" | "private"] "section"
   docLiteral "{" {envTopLevelDec} "}".
```



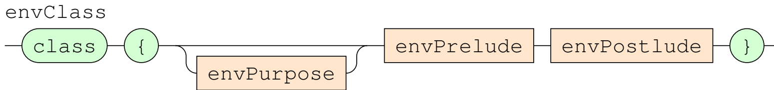
3.0.2 Environment elements

The environment elements are things that produce actual Java code. All have the possibility to include a purpose documentation insert. Where the element is normally exposed by `mashdoc` it is not optional.

Environment classes

The environment class specifies the top and the tail of the output Java program. At a minimum the prelude will contain the class heading and the postlude will contain a closing brace. There must only be one environment class per environment.

```
envClass ::= "class" "{" [envPurpose]
             envPrelude envPostlude "}".
```



The class prelude *may* include Java `import` directives, but there is a good reason not to. Students *will* choose names for their MaSH programs that collide with names of classes that are in the imported packages. This causes mysterious errors to be generated by `javac`. Instead of using `import` directives, fully qualify all Java API class names in the environment.

Similarly, the class may extend some other class, at the risk that students create members with signatures that collide with members of the superclass.

Extra, private classes to support the main class may be defined in the postlude. A useful pattern: Make the main class a subclass of a private class that provides default behaviours that can be optionally overridden by students when they implement a method.

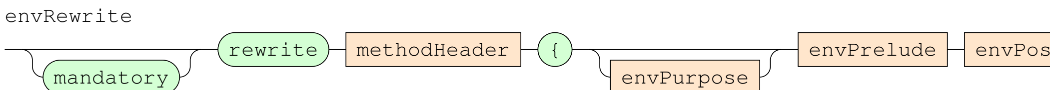
Rewrites

A rewrite specifies how to rewrite the top and the tail of a method that is declared in a MaSH program. The first and foremost use of a rewrite is to define the main method for a Java program, but it could be used for `run()` methods in multi-threaded environments, event handlers, repaints, or any method that is called by the JVM and not directly by method invocations within the MaSH program itself.

A rewrite declaration begins with the keyword `rewrite` and then a method header. The method header matches the one that will be declared within the MaSH program. Then within braces specify the prelude and postlude that replace the MaSH method's top and tail.

The prelude and postlude of a rewrite are not written as is. Any instance of a pattern of the form `#`*`arg`*`#` where *`arg`* is one of the formal parameter names is replaced by the corresponsing formal parameter name in the MaSH program.

```
envRewrite ::= ["mandatory"] "rewrite" methodHeader
               "{" [envPurpose]
               envPrelude envPostlude "}".
```



For the `main` method that automatically encloses the statements in a no-method MaSH program use the header `void main()`. Most environments will

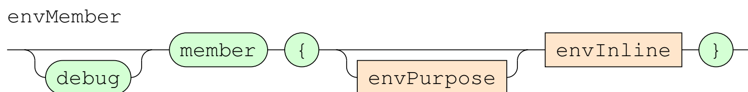
choose to rewrite this method with the standard, usually useless, `args` parameter. If the environment is to providing a debugger, then add `#OPEN_DEBUGGER#` to the prelude and `#CLOSE_DEBUGGER#` to the postlude.

Methods that you do not want a debugger to be able to single step through (recommended if they run in the event dispatcher thread, e.g. `paintComponent`), should include `#SUSPEND_DEBUGGER#` in the prelude and `#RESUME_DEBUGGER#` in the postlude.

Members

A member is any arbitrary text that is to included in the output Java class. Use it to define global variables, private constants, inner classes... The optional keyword `debug` declares that this member is only to be included in the output when the `+debug` option is applied to the `mashc` command.

```
envMember ::= ["debug"] "member"
              "{" [envPurpose] envInline "}".
```

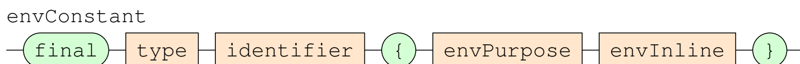


Environment constants

A environment constant is a constant to be defined as a `static` within the output class and available to be used in the MaSH program and documented by `mashdoc`.

It is declared with `final`, a type and a name as per Java, but then the value of the constant is hidden in an inline.

```
envConstant ::= "final" type identifier
                "{" envPurpose envInline "}".
```



Environment methods

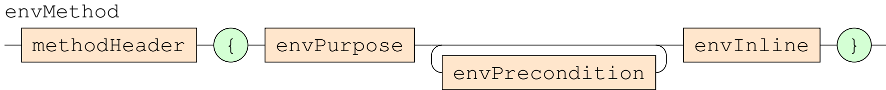
An environment method is similar to a rewrite, but rather than rewriting the definition of a method it rewrites a method invocation.

The declaration of an environment method begins with a method header. Invocations in the MaSH program that match this header are replaced by the inline.

The inline of a method is not written as is. Any instance of a pattern of the form *`#arg#`* where *`arg`* is one of the formal parameter names is replaced by the actual parameter in the MaSH program's invocation.

Methods must have a purpose and should also have preconditions that document the ranges permissible for the arguments.

```
envMethod ::= methodHeader "{" envPurpose
              {envPrecondition} envInline "}".
```

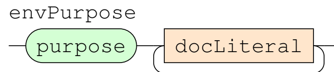


3.0.3 Documentation inserts

Documentation inserts contain LaTeX (and optionally HTML) formatted text to be included in the `mashdoc` documentation. Each consists of a keyword to indicate the kind of insert it is, and then a sequence of documentation literals. Each documentation literal in the sequence is typeset as a separate paragraph.

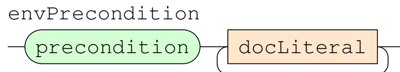
Purposes

```
envPurpose ::= "purpose" {docLiteral}+.
```



Preconditions

```
envPrecondition ::= "precondition" {docLiteral}+.
```

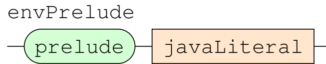


3.0.4 Java inserts

Java inserts contain Java code to be included in Java program output by `mashc`. Each of the environment elements will contain a prelude and a postlude, or only an inline. Each consists of a keyword to indicate the kind of insert it is, and then a java literal.

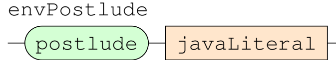
Preludes

```
envPrelude ::= "prelude" javaLiteral.
```



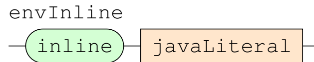
Postludes

```
envPostlude ::= "postlude" javaLiteral.
```



Inlines

```
envInline ::= "inline" javaLiteral.
```



4 Providing a debugger

An environment that wants to provide a debugger needs to provide an object called `mash_debugger` that (effectively, but not explicitly) implements the fake interface below. It also needs to inject the calls to open, close, suspend, resume, and update. Do these things *only* by using the substitutions listed in table 1, so that the calls are not injected unless the `+debug` option is applied to the compiler. The push, pop, checkpoint and set calls are injected by the compiler.

```
interface MaSHDebugger {

   // called first to initialise the debugger when the program
   // is about to start.
   public void open();

   // called last to inform the debugger that the program is
   // finished
   public void close();

   // called when a new method is entered
   public void push();

   // called to inform debugger of a new variable coming into
   // scope. The value is usually "undefined" in this call.
   // The kind identifies the scope of the variable, but
   // don't rely on the format given. Deduce that from
```

```
// the order of push calls.
public void push(String name, String kind, String type,
                 String value);

// called when a block (not a method's block) is exited.
// n variables are being deallocated
public void pop(int n)

// called when a method is exited, all the variables for
// that method are being deallocated.
public void pop()

// called when a previously pushed variable is assigned a
// new value. The variable is either in the current method
// or a global. It's up to the debugger to resolve that
// question.
public void set(String name, value)

// called at some point in the program, given the debugger
// the opportunity to take control, pause, display the source
// code, where the program is up to (line, column) (counting
// from 0), and the contents of variables.
public void checkpoint(int line, int column)

// same as checkpoint(int, int), but the location is beyond
// the end of a no-method program.
public void checkpoint()

// called to convert a value to a String. Implement this for
// all MaSH types. Since MaSH has arrays, implement for as
// many dimensioned arrays as you think necessary.
public String toString(value)

// called to inform the debugger that it must not delay
// or block on checkpoint calls until resumed.
public void suspend()

// called after suspend() to resume the normal operation
// on checkpoint calls.
public void suspend()

// called (usually between suspend() and resume()) to
// request the debugger update its display of variable
// contents, without delaying or blocking.
public void update()
```

```
}
```