JSE Reference Manual

for release 0.4

Jonathan Bachrach
MIT AI Lab

0.1 Introduction

We have written a java implementation of this sytem called JSE (for Java Syntactic Extender). It is a preprocessor which takes as input files with Java macro definitions and uses and produces pure Java. It uses a parser generator called ANTLR and an associated extended ANTLR Java grammar to produce a Java text to code quote translator.

0.2 Installation

You need to make sure your CLASSPATH includes directories with ANTLR, JSE, and any macros you might be needing.

0.3 Usage

You of course must compile macros that you use before you use them. From the command line you can preprocess a JSE file with the following command:

```
java JSE file.jse ...
```

which will translate `file.jse` to `file.java`. Then you just run your Java compiler on the output as follows:

```
javac file.java ...
```

which will produce file.class.

JSE takes a few flags:

`-pretty`	`off`
	pretty print output
`-lineup`	`on`
	maintain source line positions
`-stdio`	`off`
	take input from stdin and output to stdout
`-deep`	`on`
	macro expands recursively
`-one`	`off`
	macro expands only one level
`-trace`	`off`
	turn on very verbose tracing

0.3.1 Makefiles

You can write Makefiles to do the JSE preprocessing and Java compilation automatically by including implicit rules for JSE and java suffixed files. For example, in GNU's Make you include the following at the top of a Makefile:

```
.SUFFIXES: .java .class .jse
```

to teach it about your new implicit rules and your actual implicit rules at the bottom of your Makefile:

```
.java.class:
$(COMPILER) $(COPTIONS) -classpath ''$(CLASSPATH)''  $?
.jse.java:
java JSE $?
```

0.4 Status

Information on the system will be made available on `http://www.ai.mit.edu/ jrb/jse`. Currently the following are not implemented:

`tracing`	`-trace`	**Workar**
`error trailing`	`none`	**Workar**
`hygiene`	`genSym`	**Workar**
`default in syntaxSwitch`	`# ...`	**Workar**
`use of ...  outside of codeQuotes`	`* pattern variable`	**Workar**

0.5 API

0.5.1 Hierarchy

```
Fragment
  CompoundFragment
    NestedFragment
      BracesFragment
      BracketsFragment
      ParensFragment
    PatternVariableFragment
      DotDotDotFragment
    SequenceFragment
    Expansion
    Template
  LeafFragment
    IdentifierFragment
    LiteralFragment
      CharacterFragment
      StringFragment
      IntegerFragment
      FloatFragment
    PunctuationFragment
      QuestionFragment
      DotFragment
      SeparatorFragment
        CommaFragment
        SemicolonFragment
SyntaxException
  SyntaxMatchFailure
SyntaxConstraint
```

```
  GrammarSyntaxConstraint
List
  FragmentList
LooseFragmentParser
MacroExpander
PrettyStream
```

0.5.2 Methods

Fragment Methods

`out`	`PrettyStream ()`	**Static M**
	pretty printer	
`getInt`	`int ()`	**M**
	converts to an integer if possible	
`getString`	`String ()`	**M**
	converts to a string if possible	
`getFloat`	`float ()`	**M**
	converts to a float if possible	
`getValue`	`Object ()`	**M**
	converts to a value if possible	
`tokens`	`List ()`	**M**
	returns token representation	
`collectBoundVariables`	`FragmentList ()`	**M**
	finds all pattern variables	

CompoundFragment Methods

`getInsideFragments`	`FragmentList ()`	**Method**

SequenceFragment Methods

`FragmentList`	`SequenceFragment ()`	**Static Method**

IdentifierFragment Methods

`IdentifierFragment`	`IndentifierFragment(String)`	**Static Method**
`genSym`	`IdentifierFragment (String name)`	**Static Method**
`capitalize`	`String ()`	**Method**
`equals`	`boolean (IdentifierFragment)`	**Method**
`equals`	`boolean (String)`	**Method**

FragmentList Methods

`fnil`	`FragmentList`	**Constant**
	empty list	
`flist`	`FragmentList (Fragment)`	**Static Method**
	creates a one fragment list	
`felt`	`Fragment (int)`	**Method**
	gets nth element	
`fpush`	`FragmentList (Fragment)`	**Method**
	pushes arg onto front of list	
`fpush`	`FragmentList (int)`	**Method**
	fpush(new IntegerFragment(x))	
`fpush`	`FragmentList (float)`	**Method**
	fpush(new FloatFragment(x))	
`fpush`	`FragmentList (String)`	**Method**
	fpush(new StringFragment(x))	
`fpush`	`FragmentList (char)`	**Method**
	fpush(new CharacterFragment(x))	
`fhead`	`Fragment ()`	**Method**
	gets head of list	
`ftail`	`FragmentList ()`	**Method**
	gets tail of list	
`fappend`	`FragmentList (FragmentList)`	**Method**
	concatenation of two lists	
`freverse`	`FragmentList ()`	**Method**
	reverses list	
`tokens`	`List ()`	**Method**
	returns token representation	
`expandMacros`	`void (MacroExpander, boolean isRecursive)`	**Method**
`expandTemplates`	`void ()`	**Method**
`pprint`	`void (PrettyStream)`	**Method**
`lprint`	`void (PrettyStream)`	**Method**

LooseFragmentParser Methods

`LooseFragmentParser`	`LooseFragmentParser (String filename, boolean isTraced)`	**Static Method**
`parse`	`FragmentList ()`	**Method**

PrettyStream Methods

`PrettyStream`	`PrettyStream (PrintStream)`	**Static Method**
`getLine`	`int ()`	**Method**
`println`	`void ()`	**Method**
`println`	`void (String)`	**Method**
`print`	`void (Object)`	**Method**
`close`	`void ()`	**Method**
`println`	`void (Object)`	**Method**