The Bali Language

Jacob ("Jack") Neal Sarvela

May 1, 2003

Abstract. Bali is an LL(k) grammar-specification language designed to produce parsers for the **AHEAD** tool suite. Unlike traditional grammar specifications using, for example, lex/yacc or JavaCC, Bali supports composition of specifications so that sub-grammars can be shared and re-used. This document describes Bali grammar specifications and it summarizes the construction of a parser from a Bali specification.

This is a preliminary version of this document that describes the new set of Bali tools (yes, there was an *old* set which is no longer being used). It is being merged with the older documentation in *How to Write a Compiler Using **AHEAD** Tools* (see HowTo.html), in *Internal Representation of AHEAD Abstract Syntax Trees/Parse Trees* (see AST.html), and in *AST Cursors* (see AST_Cursors.html), but the merge isn't complete. For now, the reader should read all these documents. In addition, the new tool-specific documentation for bali2jak (see bali2jak.html), bali2javacc (see bali2javacc.html) and balicomposer (see balicomposer.html) should be helpful.

Table of Contents

§1. Introduction to Bali 2

§2. Building and Using a Parser 3
- 2.1. Phase 1: Generating a Bali Parser 3
- 2.2. Phase 2: Composing the **AHEAD** Tool Layers 5
- 2.3. Phase 3: Compiling the Generated Source Code 6

§3. Bali Language Reference 6
- 3.1. The Options Section 6
- 3.2. The Parser Code Section 6
- 3.3. require Statement 7
- 3.4. Bali Token Definitions 7
- 3.5. Regular Expression Token Definitions 7
- 3.6. Bali Rules 8
- 3.7. JAVACODE Productions 9
- 3.8. Token Manager Declarations 9

§4. The Bali Grammar for Bali 9

1 Introduction to Bali

A Bali grammar specification combines modified LL(k) BNF grammar rules with lexical definitions and embedded Java code. Grammars written in Bali are compositional — they may be composed to specify language combinations. This section describes the Bali language by using examples from the Bali grammar, which itself is written in Bali.

Bali Grammar Rules. We'll introduce Bali with the first BNF rule in the Bali grammar:

```
BaliGrammar : [Options] [ParserCode] [Statements] :: BaliGrammarNode ;
```

This rule, like all BNF rules in Bali, is labeled with a *non-terminal symbol* followed by a colon (":"). In this case, the label is BaliGrammar and, since this is the first BNF rule in the grammar, the label BaliGrammar is taken as the *start symbol* for the grammar. Non-terminal symbols must be *identifiers* and, by convention, they are written in *mixed case* with the beginning of each word capitalized (aka, *Pascal case* or *studly caps*).

After a rule's label, there is a set of one or more *productions* separated by a vertical bar ("|") and terminated by a semi-colon (";"). Here, BaliGrammar has only one production as shown below:

```
[Options] [ParserCode] [Statements] :: BaliGrammarNode
```

The final phrase of this production is ":: BaliGrammarNode". The double-colon ("::") denotes a *named production* where the name is BaliGrammarNode, the identifier following the double-colon. This name is used as the *class name* for a sub-parse matching the syntactic elements specified in the production.

The production itself is a sequence of three symbols: Options, ParserCode and Statements. Each symbol is optional as is denoted by the brackets ("[]") surrounding each symbol. Semantically, this single rule also defines the *outline* of a Bali grammar specification. A Bali grammar begins with an optional *options* section. That is followed by an optional *parser code* section. The last section, also optional, is the *statements* section.

The meat of a Bali grammar is in the statements section which is defined in the Bali language as follows:

```
Statements : (Statement)+ ;

Statement  : RequireStatement
           | BaliGrammarRule
           | BaliTokenDefinition
           | JavacodeProduction
           | RegexTokenDefinition
           | TokenManagerDeclarations
           ;
```

This example shows two more BNF rules named Statements and Statement, respectively. The first of these, Statements, again has only one production and, this time, the production is a *simple list* of Statement elements. A simple list is always indicated by parentheses ("()") surrounding a symbol, followed by a plus sign ("+"). During parsing, a simple list matches one or more sub-parses matching the enclosed symbol. In this example, a parse of Statements matches one or more Statement parses.

The `Statement` rule, on the other hand, consists of six productions, each of which has exactly one non-terminal symbol. During `Bali`'s early history, such productions were called *unnamed productions* or, less accurately, *unnamed rules*. It's better to call them *sub-productions* because the symbol in the production body names a sub-type of the rule. For example, a `RequireStatement` is a sub-type of `Statement` which means that any occurrence of a parse matching a `RequireStatement` can be used as a parse that matches `Statement`.

Semantically, these two rules mean that the statements section of a `Bali` grammar, if present, must contain one or more statements and each statement can be a *require statement*, a ***`Bali`*** *grammar rule*, a ***`Bali`*** *token definition*, a *Java code production*, a *regular-expression token definition* or a *token manager declaration*.

Summary. So far, we've seen three examples of `Bali` grammar rules, `BaliGrammar`, `Statements` and `Statement`. More detailed information about the `Bali` grammar is provided in the `Bali` language reference (§3, p. 6). First, though, we'll describe how a parser can be built from a set of `Bali` grammar specifications and, in the process, we'll see some other aspects of `Bali` grammars.

2 Building and Using a Parser

A `Bali` parser is generated from one or `Bali` grammar specifications and the resulting parser is usually integrated into an **AHEAD** tool such as a translator. Figure 1 shows the three phases to constructing such an **AHEAD** tool. The first phase uses the tools `balicomposer`, `bali2jak` and `bali2javacc` to generate source code for the `Bali` parser. The outputs of phase one are a *parse-tree layer* in Jak source code and a *JavaCC parser*. Phase two composes the generated parse-tree layer with the other layers of the **AHEAD** tool This step is the same as the composition of hand-written Jak layers and it generates a complete composition of Jak source code. Finally, phase three translates the generated source code into, ultimately, Java `.class` files.

2.1 Phase 1: Generating a `Bali` Parser

This phase generates a `Bali` parser from a set of `Bali` grammar specifications. For example, suppose there are two `Bali` grammar specifications. The first is `infix.b`, which specifies the grammar for infix expressions such as those used in languages like `C` and Java. Let's suppose that `Expression` is the primary non-terminal symbol in `infix.b`

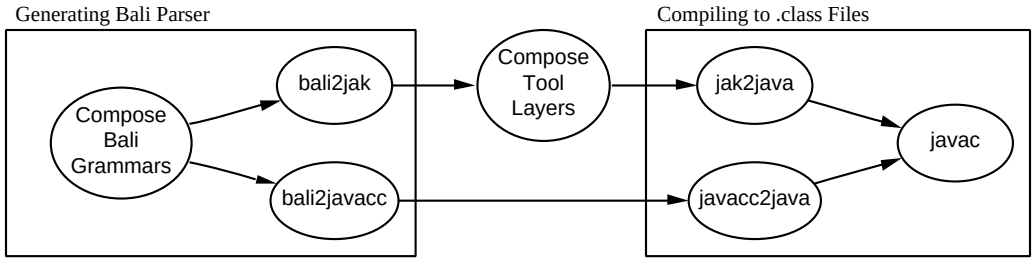

Figure 1: *Constructing an **AHEAD** Tool that Uses a `Bali` Parser.* Three phases of construction are shown: (1) Generating the `Bali` parser; (2) Compose the **AHEAD** tool layers with the `Bali` parse-tree layer; and (3) Compiling the generated source code to `.class` files.

and that it also contains secondary non-terminals such as `BooleanExpression`. The second grammar specification is `statement.b` which specifies the grammar of imperative programming statements such as `if-then-else` and `while-do`. Such statements are typically defined to use the values of boolean expressions for conditional tests, so it's natural that these two grammar specifications should be composed to yield a complete specification of statements. Here's a partial definition of `statement.b`:

```
require BooleanExpression ;

Statement = IfStatement | WhileStatement | ... ;
IfStatement = "if" "(" BooleanExpression ")" "then" ... ;
WhileStatement = "while" "(" BooleanExpression ")" "do" ... ;
```

The first line of this example is a `Bali` *require statement*. It specifies a non-terminal symbol that is defined externally in another grammar specification. In this case, the non-terminal is `BooleanExpression` which is referenced in the definitions of `IfStatement` and `WhileStatement`. When `infix.b` is composed with `statement.b`, the result is a grammar specification that defines an infix form for `BooleanExpression`. The composition is achieved with the **`composer`** tool as follows:

```
composer --target=grammar.b infix.b statement.b
```

The **`composer`** recognizes the `.b` file extension and invokes the `balicomposer` tool to perform the composition. If preferred, the user can invoke `balicomposer` directly:

```
balicomposer -output grammar.b infix.b statement.b
```

In these examples, the order of the composition is important since the definition of `BooleanExpression` must occur before its first use in a require statement.

Now, suppose that another syntax was desired for `BooleanExpression` and `Expression`. If a functional syntactic form was defined in `functional.b`, then this form could be composed with `statement.b` using either of the following two commands:

```
composer --target=grammar.b functional.b statement.b
balicomposer -output grammar.b functional.b statement.b
```

Again, the order of composition is important.

Generating the parse-tree layer. Given the composed grammar in `grammar.b`, the `bali2jak` tool is used to produce a *directory hierarchy* of Jak source files, each containing a class definition of a parse-tree node. Here's how the `bali2jak` command is used:

```
bali2jak grammar.b -directory dsl/tool/parse-tree
```

This example places the generated parse-tree nodes into a directory named `dsl/tool/parse-tree`.[1] By default, the name of the generated layer is the same as the base name of the destination directory, but this can be overridden by specifying a value to the optional `-layer` option of the `bali2jak` command.

For a specification of the generated parse-tree nodes, please refer to the `bali2jak` document.[2]

[1] This example uses UNIX-style file names. On Windows systems, a Windows-style file name should be used instead.

[2] Fix this: Refers to `bali2jak` document, so write one.

Generating the JavaCC parser. The next and final step in generating a `Bali` parser is to generate a JavaCC[3] parser that drives the generation of a `Bali` parse tree. Here's how the `bali2javacc` command can be invoked to use the `grammar.b` file as input:

```
bali2javacc grammar.b -output grammar.jj -package parser
```

The output of this step is a single file (named `grammar.jj` in this example) containing a JavaCC parser. By default, the parser is placed into Java's default package, but this can be overridden by specifying a value to `bali2javacc`'s `-package` option.

2.2 Phase 2: Composing the AHEAD Tool Layers

Given the generated layer `dsl/tool/parse-tree` from §2.1, this phase composes that layer with the additional layers that define the **AHEAD** tool being built. Typically, the additional layers define parse-tree processing methods and semantic analysis as well as a driver such as a `main` program. Collectively, we'll call these the *tool layers*. We'll suppose, for example, that the there are two tool layers named `dsl/tool/semantics` and `dsl/tool/driver`.

In addition to the parse-tree layer and the tool layers, it's also necessary to include a *kernel layer* that defines the base classes for the parse-tree layer. All in all, the layers to be composed are:

`dsl/kernel` ..... *standard kernel layer defined in* ***AHEAD***
`dsl/tool/parse-tree` ..... *generated parse-tree layer*
`dsl/tool/semantics` ..... *semantic refinements to parse-tree nodes*
`dsl/tool/driver` ..... *driver program with argument processing, etc.*

These layers form the *equation* to be composed. There are some order dependencies in this equation. For example, the parse-tree layer, by construction, refines the kernel layer so it must come after the kernel layer. Typically, the semantics layer refines the parse-tree layer, so that comes next. On the other hand, a driver layer can be written to be independent of the other layers by, e.g., using reflection but, for this example, we assume that the driver layer refines the semantics layer. Given these order dependencies, an equations file defining the tool can be created to contain the following single line:

```
this = dsl/kernel dsl/tool/parse-tree dsl/tool/semantics dsl/tool/driver
```

Supposing that the equations file is named `tool.equations`, the tool can be composed with the following **`composer`** command:

```
composer --equation=tool.equations
```

By default, the **`composer`** writes the resulting source code into a directory named with the base name of the equations file, `tool` in this case. To specify an alternative destination, the **`composer`**'s `--target` option could be specified. Refer to the **`composer`** document for more detail.[4]

[3] Fix this: Reference JavaCC documentation.
[4] Fix this: Include reference to **`composer`** document.

2.3 Phase 3: Compiling the Generated Source Code

Finally, the generated code in the `tool` directory can be compiled. Here, the Jak source code and the JavaCC parser must first be translated to Java. This can be accomplished with the following command sequence, again using Unix-style syntax:

```
cd tool
jak2java *.jak
javacc ../grammar.jj
```

The resulting Java code can then be compiled to class files by using a standard Java compiler.

3 Bali Language Reference

In §1, some of the basic elements of a `Bali` grammar were introduced, but more complex parts of `Bali` was not described. This section summarizes all major parts of a `Bali` grammar. Some parts of a `Bali` grammar are taken directly from JavaCC[5] grammars and those parts will be described by reference to the Web-based JavaCC documentation.

Whitespace and comments. The `Bali` lexical analyzer will scan and ignore any unquoted whitespace, including spaces, tabs, linefeeds and carriage returns. Further, `C`-style and `C++`-style comments can appear anywhere that a whitespace character can appear. `C`-style comments begin with a `/*` and continue until a closing `*/` is encountered. On the other hand, `C++`-style comments begin with a `//` and continue only to the next end-of-line.

3.1 The Options Section

The options section is the first non-comment region in a `Bali` grammar. It begins with "`options {`", it ends with "`} options`" and it can include any JavaCC *option binding* as defined in the JavaCC documentation (`http://www.experimentalstuff.com/Technologies/JavaCC/javaccgrm.html#prod6`).

3.2 The Parser Code Section

The parser code section is a block of Java code, quoted within "`code {`" and "`} code`" delimiters, that follows the options section. The entire block, whitespace and all, is placed verbatim into the parser class generated by JavaCC. When building a JavaCC parser by way of `Bali`, the parser class is always named `BaliParser`.

[5] `http://www.experimentalstuff.com/Technologies/JavaCC/`

3.3 `require` Statement

One example of a `require` statement occurred in §2.1. These statements are used to specify non-terminal symbols that are defined externally to the current `Bali` grammar specification. The general syntax of a `require` statement, written in `Bali`, is as follows:

```
RequireStatement : "require" RequireRules ";" ;
RequireRules     : RequireRule ("," RequireRule)* ;
RequireRule      : IDENTIFIER [RequireType] ;
RequireType      : "->" IDENTIFIER ;
```

The keyword `require` can be followed by one or more symbol specifications which are separated by a comma (“`,`”). Each symbol specification is a non-terminal symbol, optionally followed by a type name. The type name may be the name of any valid syntax-tree class and, when not present, it defaults to the class that corresponds to the non-terminal symbol itself.

3.4 `Bali` Token Definitions

There are two types of token definitions in `Bali` grammars. The simplest type is most suitable for constant tokens such as keywords and punctuation characters. Here are a few examples of `Bali` token definitions:

```
"<"                   OPENANGLE
">"                   CLOSEANGLE
"("                   OPENPAREN
")"                   CLOSEPAREN
"code"                CODE
"options"             OPTIONS
```

For each `Bali` token, the constant string value of the token is first given as a quoted string and that is followed by the token name. Once a token name is defined in this way, it may be used as a terminal symbol in any `Bali` rule. By convention, token names are always upper-case identifiers.

3.5 Regular Expression Token Definitions

More complex token definitions are defined by using regular expressions. Such definitions are exactly the same as JavaCC *regular expression productions* which are described at `http://www.experimentalstuff.com/Technologies/JavaCC/javaccgrm.html\#prod10`. Examples can be found in the `Bali` grammar for `Bali` in §4.

3.6 Bali Rules

Bali grammar rules were introduced in §1 and we won't repeat everything from the introduction. Here, we briefly describe the four types of productions that can be included in a rule: (1) Named productions; (2) Sub-productions; (3) Simple lists; and (4) Complex lists. In this version of Bali, using some production types places a restriction on Bali rules and these restrictions are noted in longer descriptions below.

Generally, a production is defined in terms of of *primitive tokens*. These can be terminal symbols, non-terminal symbols or quoted strings. Primitives may be *optional*, in which case they are surrounded by brackets ("[]"). Any production can also begin with an optional *lookahead clause* as specified by JavaCC http://www.experimentalstuff.com/Technologies/JavaCC/javaccgrm.html#prod21. Depending on the type of production, other elements may also be present.

Named productions. A named production is any arbitrary sequence of primitives followed by a double-colon ("::") and a name. Any Bali grammar rule can have an arbitrary number of named productions. Here's an example of a single named production within a rule:

```
StateSet : "<" States ">" :: StatesNode ;
```

The name is used as the name of a generated class representing a parse-tree node matching the production.

Sub-productions. A sub-production has exactly one non-terminal symbol and a rule can have any number of sub-productions. Two sub-productions are shown in the rule below:

```
Statement : IfStatement | WhileStatement ;
```

This specifies that non-terminals IfStatement and WhileStatement are sub-types of non-terminal Statement. This is reflected in the inheritance hierarchy generated by bali2jak.

Simple lists. List productions in Bali currently are used to define several different parse-tree classes in their own inheritance hierarchy. As a result, if a list production occurs in a Bali grammar rule, the rule must have *no other productions*. A simple list has the following Bali grammar specification:

```
SimpleList : "(" [Lookahead] Primitive ")" "+" ;
```

where Lookahead represents a lookahead clause and Primitive represents any primitive element as described above. The semantic meaning of a simple list is as a whitespace-separated sequence of one or matches of the primitive element.

Complex lists. Complex list productions suffer from the same restriction as simple list productions — a Bali grammar rule that contains a complex list production can contain no other productions. However, complex lists do allow a slightly more general type of list as specified in the Bali grammar rule below:

```
ComplexList : Primitive "(" [Lookahead] Primitive Primitive ")" "*" ;
```

The purpose of complex lists is to specify a sequence of parse elements with an arbitrary separator. For example, it's typical to parse function arguments as a comma-separated list of expressions. The example below shows a complex list production for function arguments:

```
Arguments : Argument ("," Argument)+ ;
```

However, the definition of ComplexList previous allows more general lists, including lists where the separator is specified by a non-terminal symbol.

3.7 JAVACODE Productions

A JAVACODE production is another statement type taken from JavaCC. They begin with the keyword JAVACODE preceding a Java method definition. The accompanying Java method is copied verbatim into the generated JavaCC parser. Detailed documentation is available at http://www.experimentalstuff.com/Technologies/JavaCC/javaccgrm.html#prod9.

3.8 Token Manager Declarations

The last statement type specifies *token manager declarations* which are copied verbatim into the lexical analyzer generated by JavaCC. Detailed documentation is available at http://www.experimentalstuff.com/Technologies/JavaCC/javaccgrm.html#TOKEN_MGR_DECLS.

4 The Bali Grammar for Bali

This section contains the complete grammar for the Bali language as specified in the Bali language itself. This is the composed Bali specification used in the actual build process.

```
// Automatically generated Bali code.  Edit at your own risk!
// Generated by "balicomposer" v2003.02.17.

//-----------------------------------//
// Option block:
//-----------------------------------//

options {
    CACHE_TOKENS = true ;
    JAVA_UNICODE_ESCAPE = true ;
    OPTIMIZE_TOKEN_MANAGER = true ;
    STATIC = false ;
} options

//-----------------------------------//
// Parser code block:
//-----------------------------------//

code {
    //*********************************************************************
    // Code inserted from "bali.b" source grammar:
    //*********************************************************************

    /**
     * Append the given {@link Token} and any preceding special tokens to a
```

```
     * given {@link StringBuffer}.
     *
     * @param token the given JavaCC {@link Token} object
     * @param buffer the buffer to which to append <code>token</code>
     **/
    final private static void accumulate (Token token, StringBuffer buffer) {

        // Append preceding special tokens to <code>buffer</code>:
        //
        Token special = firstSpecial (token) ;
        if (special != token)
            while (special != null) {
                buffer.append (special.toString ()) ;
                special = special.next ;
            }

        // Finally, append the token itself:
        //
        buffer.append (token.toString ()) ;
    }

    /**
     * Accumulate {@list Token} objects from the token stream, respecting
     * nested code inside <code>open</code> and <code>close</code> pairs,
     * until an unmatched <code>close</code> is the next token in the stream.
     * This method assumes that an <code>open</code> token has just been read
     * from the stream so the initial nesting level is 1.  The method returns
     * when a matching <code>close</code> token is the next token in the token
     * stream.  <em>The <code>close</code> token is left in the stream!</em>
     *
     * @return the accumulated tokens as a {@link String}.
     *
     * @throws ParseException
     * if an end-of-file is found before an unmatched <code>close</code> token.
     **/
    final private Token accumulateNestedRegion (int open, int close)
    throws ParseException {

        StringBuffer buffer = new StringBuffer () ;

        // Initialize result with known information (starting position, etc.):
        //
        Token result = Token.newToken (OTHER) ;
        result.specialToken = null ;

        Token startToken = firstSpecial (getToken (1)) ;
        result.beginColumn = startToken.beginColumn ;
```

```
        result.beginLine = startToken.beginLine ;

        // Accumulate tokens until a <code>close</code> token is found:
        //
        for (int nesting = 1 ; nesting > 0 ; ) {

            token = getToken (1) ;

            // Update information in result:
            //
            result.endColumn = token.endColumn ;
            result.endLine = token.endLine ;
            result.next = token.next ;

            if (token.kind == EOF)
                throw new ParseException (
                    "accumulating from line "
                    + result.beginLine
                    + " at column "
                    + result.beginColumn
                    + ": EOF reached before ending "
                    + tokenImage [close]
                    + " found"
                ) ;

            if (token.kind == open)
                ++ nesting ;
            else if (token.kind == close) {
                if (nesting == 1)
                    break ;
                -- nesting ;
            }

            accumulate (token, buffer) ;
            getNextToken () ;
        }

        result.image = buffer.toString () ;
        return result ;
    }

    /**
     * Accumulate {@link Token} objects from the token stream until a token
     * matching <code>tokenKind</code> is consumed from the stream.  The
     * tokens are accumulated in <code>buffer</code>, including the terminating
     * token.
     *
```

```
     * @return a {@link Token}
     * formed by concatenating all intervening tokens and special tokens.
     **/
    final private Token accumulateUntilToken (int tokenKind)
    throws ParseException {

        StringBuffer buffer = new StringBuffer () ;
        Token token = getNextToken () ;

        // Initialize result with known information (starting position, etc.):
        //
        Token result = Token.newToken (OTHER) ;
        result.specialToken = null ;

        Token startToken = firstSpecial (token) ;
        result.beginColumn = startToken.beginColumn ;
        result.beginLine = startToken.beginLine ;

        // Accumulate tokens until a <code>tokenKind</code> token is found:
        //
        while (token.kind != tokenKind) {

            // Update information in result:
            //
            result.endColumn = token.endColumn ;
            result.endLine = token.endLine ;
            result.next = token.next ;

            if (token.kind == EOF)
                throw new ParseException (
                    "from line "
                    + result.beginLine
                    + " at column "
                    + result.beginColumn
                    + ": EOF reached before "
                    + tokenImage [tokenKind]
                    + " found"
                ) ;

            accumulate (token, buffer) ;
            token = getNextToken () ;
        }

        accumulate (token, buffer) ;

        result.image = buffer.toString () ;
        return result ;
```

```
    }

    /**
     * Finds the first token, special or otherwise, in the list of special
     * tokens preceding this {@link Token}.  If this list is non-empty, the
     * result will be a special token.  Otherwise, it will be the starting
     * token.
     *
     * @param token the given {@link Token}.
     * @return the first special token preceding <code>token</code>.
     **/
    final private static Token firstSpecial (Token token) {

        while (token.specialToken != null)
            token = token.specialToken ;

        return token ;
    }
} code

//-----------------------------------//
// Token manager declarations:
//-----------------------------------//

// No TOKEN_MGR_DECLS defined in Bali grammar.

//-----------------------------------//
// Bali tokens:
//-----------------------------------//

">"                     CLOSEANGLE
")"                     CLOSEPAREN
"{"                     LBRACE
"<"                     OPENANGLE
"("                     OPENPAREN
"}"                     RBRACE
"code"                  _CODE
"EOF"                   _EOF
"IGNORE_CASE"           _IGNORE_CASE
"JAVACODE"              _JAVACODE
"LOOKAHEAD"             _LOOKAHEAD
"MORE"                  _MORE
"options"               _OPTIONS
"PARSER_BEGIN"          _PARSER_BEGIN
"PARSER_END"            _PARSER_END
"require"               _REQUIRE
"SKIP"                  _SKIP
```

```
"SPECIAL_TOKEN"         _SPECIAL_TOKEN
"TOKEN"                 _TOKEN
"TOKEN_MGR_DECLS"       _TOKEN_MGR_DECLS


//-----------------------------------//
// Regular expression tokens:
//-----------------------------------//

TOKEN: {
        <BALI_TOKEN: <UPPERCASE> (<UPPERCASE> | <DIGIT>)*> |
        <#UPPERCASE: ["A"-"Z", "_", "$"]> |
        <STRING:
                "\""
                ( (~["\"","\\","\n","\r"])
                | ("\\"
                    ( ["n","t","b","r","f","\\","'","\""]
                    | ["0"-"7"] ( ["0"-"7"] )?
                    | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                    )
                  )
                )*
                "\""
        > |
        <INTEGER: (<DIGIT>)+>
}


//-----------------------------------//
// Java code blocks:
//-----------------------------------//

JAVACODE
CodeBlockNode codeBlockNode (Token token) {
    return (new CodeBlockNode ()) . setParms (t2at (token)) ;
}

JAVACODE
CodeBlockNode findBlockBegin () {
    return codeBlockNode (accumulateUntilToken (LBRACE)) ;
}

JAVACODE
CodeBlockNode findBlockEnd () {
    return codeBlockNode (accumulateNestedRegion (LBRACE, RBRACE)) ;
}
```

```
JAVACODE
CodeBlockNode findCloseAngle () {
    return codeBlockNode (accumulateNestedRegion (OPENANGLE, CLOSEANGLE)) ;
}

JAVACODE
CodeBlockNode findCloseParen () {
    return codeBlockNode (accumulateNestedRegion (OPENPAREN, CLOSEPAREN)) ;
}

//-----------------------------------//
// Bali productions:
//-----------------------------------//

require findBlockBegin -> CodeBlockNode ;
require findBlockEnd -> CodeBlockNode ;
require findCloseAngle -> CodeBlockNode ;
require findCloseParen -> CodeBlockNode ;

BaliParse
        : [Options] [ParserCode] [Statements]           :: BaliParseNode
        ;

AngleRegex
        : LOOKAHEAD(2) BALI_TOKEN ">"                   :: BaliRegexNode
        | LOOKAHEAD(2) [Label] ComplexRegex             :: ComplexRegexNode
        ;

BaliGrammarRule
        : IDENTIFIER ":" Productions ";"                :: BaliGrammarNode
        ;

BaliTokenDefinition
        : STRING BALI_TOKEN                             :: BaliTokenDefineNode
        ;

Block
        : "{" findBlockEnd "}"                          :: BlockNode
        ;

CaseFlag
        : "[" _IGNORE_CASE "]"                          :: CaseFlagNode
        ;

ClassName
        : "::" IDENTIFIER                               :: ClassNameNode
        ;
```

```
ComplexRegex
        : LOOKAHEAD(2) STRING ">"                       :: StringComplexNode
        | findCloseAngle ">"                            :: AngleComplexNode
        ;

JavacodeProduction
        : _JAVACODE ScanBlock                           :: JavacodeNode
        ;

Label
        : ["#"] BALI_TOKEN ":"                          :: LabelNode
        ;

Lookahead
        : _LOOKAHEAD "(" findCloseParen ")"             :: LookaheadNode
        ;

NextState
        : ":" BALI_TOKEN                                :: NextStateNode
        ;

Options
        : _OPTIONS Block _OPTIONS                       :: OptionsNode
        ;

ParserCode
        : _CODE Block _CODE                             :: ParserCodeNode
        ;

Pattern
        : (Primitive)+
        ;

Primitive
        : "[" [Lookahead] Terminal "]"                  :: OptionalNode
        | Terminal
        ;

PrimitiveRewrite
        : "(" [Lookahead] Primitive Primitive ")" "*"   :: ComplexListNode
        | [Pattern] [ClassName]                         :: PatternNode
        ;

Production
        : [Lookahead] Rewrite                           :: ProductionNode
        ;
```

```
Productions
        : Production ("|" Production)*
        ;

REKind
        : _TOKEN                                        :: TokenKindNode
        | _SPECIAL_TOKEN                                :: SpecialKindNode
        | _SKIP                                         :: SkipKindNode
        | _MORE                                         :: MoreKindNode
        ;

RElist
        : RegexBlock ("|" RegexBlock)*
        ;

Regex
        : STRING                                        :: StringRegexNode
        | "<" AngleRegex                                :: AngleRegexNode
        ;

RegexBlock
        : Regex [Block] [NextState]                     :: RegexBlockNode
        ;

RegexTokenDefinition
        : [StateSet] REKind [CaseFlag] ":" "{" RElist "}"
                                                        :: RegexDefinitionNode
        ;

RequireRule
        : IDENTIFIER [RequireType]                      :: RequireRuleNode
        ;

RequireRules
        : RequireRule ("," RequireRule)*
        ;

RequireStatement
        : _REQUIRE RequireRules ";"                     :: RequireStatementNode
        ;

RequireType
        : "->" IDENTIFIER                               :: RequireTypeNode
        ;

Rewrite
        : "(" [Lookahead] Primitive ")" "+"             :: SimpleListNode
```

```
        | Primitive PrimitiveRewrite                    :: PrimitiveRewriteNode
        ;

ScanBlock
        : findBlockBegin findBlockEnd "}"               :: ScanBlockNode
        ;

StateName
        : BALI_TOKEN                                    :: StateNameNode
        ;

StateSet
        : "<" StatesSpecifier ">"                       :: StatesNode
        ;

Statement
        : RequireStatement
        | BaliGrammarRule
        | BaliTokenDefinition
        | JavacodeProduction
        | RegexTokenDefinition
        | TokenManagerDeclarations
        ;

Statements
        : (Statement)+
        ;

StatesList
        : StateName ("," StateName)*
        ;

StatesSpecifier
        : "*"                                           :: StarStatesNode
        | StatesList                                    :: ListStatesNode
        ;

Terminal
        : BALI_TOKEN                                    :: BaliTokenNode
        | IDENTIFIER                                    :: IdentifierNode
        | STRING                                        :: StringNode
        ;

TokenManagerDeclarations
        : _TOKEN_MGR_DECLS ":" ScanBlock                :: TokenManagerNode
        ;
```