

## B. j-- Syntax

### B.1 Introduction and Notation

This appendix describes the lexical and syntactic grammars for the *j--* programming language; the former specifies how individual tokens in the language are composed, and the latter specifies how language constructs are formed.

We employ the following notation to describe the grammars.

- *//* indicates comments;
- Non-terminals are written in the form of Java (mixed-case) identifiers;
- Terminals are written in **bold**;
- Token representations are enclosed in **<>**;
- [x] indicates zero or one occurrence of x;
- {x} indicates zero or more occurrences of x;
- x | y indicates x or y;
- ~x indicates negation of x;
- Parentheses are used for grouping;
- Level numbers in expressions indicate precedence

### B.2 Lexical Grammar

#### B.2.1 White Space

White space in *j--* is defined as the ASCII space (**SP**), horizontal tab (**HT**), and form feed (**FF**) characters, as well as line terminators: line feed (**LF**), carriage return (**CR**), and carriage return (**CR**) followed by line feed (**LF**).

#### B.2.2 Comments

*j--* supports single-line comments; all the text from the ASCII characters *//* to the end of the line is ignored.

#### B.2.3 Reserved Words

The following tokens are reserved for use as keywords in *j--* and cannot be used as identifiers.

Appendix B - 1

<b>abstract</b>	<b>boolean</b>	<b>char</b>	<b>class</b>	<b>else</b>	<b>extends</b>	<b>false</b>
<b>import</b>	<b>instanceof</b>	<b>int</b>	<b>new</b>	<b>null</b>	<b>package</b>	<b>private</b>
<b>protected</b>	<b>public</b>	<b>return</b>	<b>static</b>	<b>super</b>	<b>this</b>	<b>true</b>
<b>void</b>	<b>while</b>					

## B.2.4 Operators

The following tokens serve as operators in *j--*.

**=    ==    >    ++    &&    <=    !    -    +    +=    \***

## B.2.5 Separators

The following tokens serve as separators in *j--*.

**,    .    [    {    (    )    }    ]    ;**

## B.2.6 Identifiers

The following regular expression describes identifiers in *j--*.

**<identifier> → (a-z | A-Z | \_ | \$) {a-z | A-Z | \_ | 0-9 | \$}**

## B.2.7 int, char and String Literals

An escape (**ESC**) character in *j--* is a **\** followed **n**, **r**, **t**, **b**, **f**, **'**, **"**, or **\**. Besides the **true**, **false**, and **null** literals, *j--* supports **int**, **char** and **String** literals as described by the following regular expressions.

**<int\_literal> → 0 | (1-9) {0-9}**  
**<string\_literal> → " {ESC | ~(" | \ | LF | CR)} "**  
**<char\_literal> → ' (ESC | ~(' | \ | LF | CR)) '**

## B.3 Syntactic Grammar

**compilationUnit** → [**package** qualifiedIdentifier ;]  
                  {**import** qualifiedIdentifier ;}  
                  {typeDeclaration} **<eof>**

Appendix B - 2

qualifiedIdentifier → **<identifier>** { . **<identifier>** }

typeDeclaration → modifiers classDeclaration

modifiers → { **public** | **protected** | **private** | **static** | **abstract** }

classDeclaration → **class** **<identifier>** [ **extends** qualifiedIdentifier ] classBody

classBody → { { modifiers memberDecl } }

memberDecl → **<identifier>** // constructor  
                  formalParameters block  
                  | ( **void** | type ) **<identifier>** // method  
                  formalParameters ( block | ; )  
                  | type variableDeclarators ; // field

block → { { blockStatement } }

blockStatement → localVariableDeclarationStatement  
                  | statement

statement → block  
            | **<identifier>** : statement  
            | **if** parExpression statement [ **else** statement ]  
            | **for** ( forInit ; forCond ; forIter ) statement  
            | **while** parExpression statement  
            | **do** statement **while** parExpression ;  
            | **break** [ **<identifier>** ] ;  
            | **continue** [ **<identifier>** ] ;  
            | **return** [ expression ] ;  
            | ;  
            | statementExpression ;

formalParameters → ( [ formalParameter { , formalParameter } ] )

formalParameter → type **<identifier>**

parExpression → ( expression )

localVariableDeclarationStatement → type variableDeclarators ;

variableDeclarators → variableDeclarator { , variableDeclarator }

variableDeclarator → **<identifier>** [= variableInitializer]

Appendix B - 3

variableInitializer → arrayInitializer | expression

arrayInitializer → { [variableInitializer {, variableInitializer}] }

arguments → ( [expression {, expression}] )

type → referenceType | basicType

basicType → **boolean** | **char** | **int**

referenceType → basicType [ ] { [ ] }  
                   | qualifiedIdentifier { [ ] }

statementExpression → expression // but must have side-effect, eg i++

expression → assignmentExpression

assignmentExpression → conditionalAndExpression // must be a valid lhs  
                           [(= | +=) assignmentExpression]

conditionalAndExpression → equalityExpression // level 10  
                               {&& equalityExpression}

equalityExpression → relationalExpression // level 6  
                       {== relationalExpression}

relationalExpression → additiveExpression // level 5  
                       [(> | <=) additiveExpression | **instanceof** referenceType]

additiveExpression → multiplicativeExpression // level 3  
                       {(+ | -) multiplicativeExpression}

multiplicativeExpression → unaryExpression // level 2  
                           {\* unaryExpression}

unaryExpression → ++ unaryExpression // level 1  
                   | - unaryExpression  
                   | simpleUnaryExpression

simpleUnaryExpression → ! unaryExpression  
                       | ( basicType ) unaryExpression //cast  
                       | ( referenceType ) simpleUnaryExpression // cast  
                       | postfixExpression

postfixExpression → primary {selector} {--}

Appendix B - 4

selector → . qualifiedIdentifier [arguments]  
| [ expression ]

primary → parExpression  
| **this** [arguments]  
| **super** (arguments | . <identifier> [arguments])  
| literal  
| **new** creator  
| qualifiedIdentifier [arguments]

creator → (basicType | qualifiedIdentifier)  
( arguments  
| [ ] { [ ] } [arrayInitializer]  
| newArrayDeclarator )

newArrayDeclarator → [ expression ] { [ expression ] } { [ ] }

literal → <int\_literal> | <char\_literal> | <string\_literal> | true | false | null

Appendix B - 6

Draft, © 2008-2009 by Bill Campbell, Swami Iyer and Bahar Akbal