Java ++ Language Reference

Introduction to Java++

Java++ is an amalgamation of C, C++, Java, and personal design choices created by Ben Ebadinia and assisted by Dr. Arthur Hanna. Java++ is a simple programming language, but is still in development. A Java++ program must be contained in a single source file named with the extension .java++. As of December 2023, Java++ is currently designed to support:

- Single-Line and Multi-Line (block) comments (two different syntaxes)
- Literals for scalar data types integer, boolean, and string
- Explicitly defined scalar variables and named constants (named final in Java++) with scalar data types integer and boolean
- Statically allocated program module local scope variables and finals
- A list of unary and binary operators
- Ability to assign literals within the definition statement
- A multiple assignment statement
- Two "standard" structured flow-of-control statements: IF-statement and REPEATWHILE-statement
- STDIN-statement and EITHER-statement for console formatted text input/output;
- EITHER-statement where user can use "cout" or "coutln" which can automatically go to the next line

BNF for sample Java++

<Java++Program> ::= <STARTDefinition> EOPC

<STARTDefinition> ::= START
{ <Statement> | <DataDefinitions> }*
FIN

<DataDefinitions> ::= <VIDDefinitions> | <FINALDefinitions>

<VIDDefinitions> ::= VARIABLE <Datatype> <VID> { ; <Datatype> <VID> }*.
| VARIABLE <Datatype> <VID> <INTO> <literal> { ; <Datatype> <INTO> <literal> }*.

<FINALDefinitions> ::= FINAL <Datatype> <INTO> <literal>
{ ; <Datatype> <INTO> <literal> }.

<Datatype> ::= ((NUM | BOOLEAN))

<Statement> ::= <EITHERStatement>
| <STDINStatement>
| <DataDefinitions>
| <AssignmentStatement>
| <IFStatement>
| <REPEATWHILEStatement>

<EITHERStatement> ::= (cout | coutln) ([<string> | <ORExpression>]
{ ; (< string > | <ORExpression>) }*).

<STDINStatement> ::= STDIN [<string>] ; <VID>.

<AssignmentStatement> ::= <VID> { ; <VID> }* <INTO> <literal> .

<IFStatement> ::= IF [<ORExpression>]
(({))
{ <Statement> }*
{ OTHERWISEIF [<ORExpression>] EXECUTE
{ <Statement> }* }*
[OTHERWISE
{ <Statement> }*] ((}))

<REPEATWHILEStatement> ::= REPEAT
{ <Statement> }*
WHILE [<ORExpression>] (({))
{ <Statement> }* ((}))

<ORExpression> ::= <ANDExpression> { ((OR | NOR | XOR)) <ANDExpression> }*

<ANDExpression> ::= <NOTExpression> { ((AND | NAND)) <NOTExpression> }*

<NOTExpression> ::= [NOT] <COMPAREExpression>

<COMPAREExpression> ::= <BASICExpression>
{ ((< | <= | = | > | >= | !=)) <BASICExpression> }*

<BASICExpression> ::= <ADVANCEDExpression>
{ ((+ | -)) <ADVANCEDExpression> }*

<ADVANCEDExpression> ::= <FACTORExpression>
{ ((* | / | %)) <FACTORExpression> }*

<FACTORExpression> ::= [((ABSOLUTE | + | -))] <POWERExpression>

<POWERExpression> ::= <Increments> [((^)) <Increments>]

<Increments> ::= <PRIMARYExpression> | ((++ | --)) <Var>

<PRIMARYExpression> ::= (<ORExpression>) | <literal> | <VID>

<VID> ::= <letter> { ((<letter> | <num> | _)) }*

<literal> ::= <int> | <bool> | <string>

<string> ::= "{ <ASCIICharacters> }*"

<int> ::= <num> { <num> }*

<bool> ::= TRUE | FALSE

<ASCIICharacters> ::= … ||Every printable ASCII character

<letter> ::= a | A | b | B | … ||Every letter uppercase and lowercase

<num> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<INTO> ::= ((<<))

<comment> ::= ** { <ASCIICharacters> }* EOLC ||Single-line comment
| *** { <ASCIICharacters> }* ;** ||Multi-line comment

Description of the syntax and the dynamic semantics of Java++ (Assisted by Dr.Hanna)

Java++ is a free-format language; that is, any amount of white space characters (blanks, tabs, and end-of-lines) is allowed between source code tokens. An **<VID>** (Variable Identifier) like xyz or x1 are not case sensitive, however reserved words like START and FIN are case sensitive. *Java++* uses the ASCII character set.

Notice, *Java++* <**EITHERStatement**>, <**STDINStatement**>, <**AssignmentStatement**>, <**VIDDefinitions**>, <**FINALDefinitions**> are always terminated by a period, while <**IFStatement**> and <**REPEATWHILEStatement**> are terminated by a closed curly brace (}).

Notice, *Java++* <**STARTDefinition**> begins with the reserved word START and is always terminated by the reserved word FIN.

A single-line <**comment**> may be appended to the end of any source line; that is, a single-line comment extends from the double asterisk (**) prefix (used to mark the beginning of the single-line comment) to the end of the line containing the double asterisk. Single-line comments are treated like white space at the end of the source line in which the comment appears.

<**comment**> ::= ** { <**ASCIICharacters**> }* EOLC ||Single-line comment

For example,

```
**--------------------
** Print out "Hello, world!"
**--------------------
coutln("Hello, world!").      **Print out Hello, world!
```

A multi-line (block) <**comment**> may appear any place white space may appear. The block comment extends from the triple asterisk (***) prefix (used to mark the beginning of the block comment) to the semi-colon double asterisk (;**) suffix (used to mark the ending of the block comment).

<**comment**> ::= *** { <**ASCIICharacters**> }* ;** ||Multi-line comment

For example,

```
***
  Print out "Hello, world!"
;**
coutln("Hello, world!").      *** Print out
                                      Hello, world! ;**
```

<**literal**> constants may be <**int**>, <**bool**>, or <**string**>. An int literal must be unsigned and must be an element of the range [0, 2^{15}-1 = 32767]; the reserved words true and false are the only 2 bool literals; and a string literal is delimited by double quotes (") and may contain 0 or more

printable characters. (Note String literals may contain a double quote by coding it using the escape sequence \"; String literals may contain a backslash character \ by coding it using the escape sequence \\).

<**literal**> ::= <**int**> | <**bool**> | <**string**>
<**int**> ::= <**num**> { <**num**> }*
<**bool**> ::= TRUE | FALSE
<**string**> ::= "{ <**ASCIICharacters**> }*"

For example, the coutln-statement shown below displays the line "Howdy", he exclaimed!
coutln("\"Howdy\", he exclaimed!").

An **<VID>** (Variable Identifier) is used to name constants which can be program module (local) variables and finals; A variable is defined when it is specified in a data definitions section and can even be assigned a literal from within the data definition statement itself. A variable or final must be defined with a data type (int or bool).

<**VID**> ::= <**letter**> { ((<**letter**> | <**num**> | _)) }*

The scope of the START module and constants is local scope and extends from the point of definition to the end of the module body.

The lifetime (or duration) start module local variables and finals is the entire duration of the program's execution.

A **<Java++Program>** is a single program module definition: Each *Java++* program must consist of at least 1 module, the start module. When a *Java++* program begins execution, flow-of-control begins with the first statement in the program module's list of statements and flow-of-control continues until it terminates when the statement that ends the list of program module statements completes execution.

<**Java++Program**> ::= <**STARTDefinition**> EOPC

A **<DataDefinitions>** is a possibly empty list of variable and/or final definitions. A variable is named-with-<**VID**>, must be scalar, has an explicit data type, and may be given an initial literal value. A final is named-with-<**VID**>, must be scalar, has an explicit data type, and must be given an initial literal value. Every Java++ variable or final can be defined anywhere within the start module and must be defined before it can be legally referenced.

Variables and finals can not be defined in the <**DataDefinitions**> block that precedes all module definitions as Java++ currently does not offer global variables and finals. The scope of variable or final identifier is local scope, a scope that extends from the point of definition to the last statement in the body of the start. It is a static semantic error to reference an undefined identifier. It is a static semantic error to multiply-define an identifier.

<**DataDefinitions**> ::= <**VIDDefinitions**> | <**FINALDefinitions**>
<**VIDDefinitions**> ::= VARIABLE <**Datatype**> <**VID**> { ; <**Datatype**> <**VID**> }*.
| VARIABLE <**Datatype**> <**VID**> <**INTO**> <**literal**> { ; <**Datatype**> <**INTO**> <**literal**> }*.

<**FINALDefinitions**> ::= FINAL <**Datatype**> <**INTO**> <**literal**>
{ ; <**Datatype**> <**INTO**> <**literal**> }.
<**Datatype**> ::= ((NUM | BOOLEAN))

A **<STARTDefinition>** defines the unnamed program module by providing (1) an optional data definitions section for local variables and constants; and (2) a set of 0 or more executable statements that make up the program module body.

< **STARTDefinition** > ::= START
{ <**Statement**> }*
FIN

A start module body consists of 0 or more statements that begin execution when the program is run. Flow-of-control begins with the first statement in the program module's body; flow-of-control terminates after the last statement is executed. There is no explicit instruction used to terminate program module execution; instead, flow-of-control terminates when the flow-of-control "runs into" the program module FIN reserved word.

An **<expression>** computes a scalar int or bool value.

<**ORExpression**> ::= <**ANDExpression**> { ((OR | NOR | XOR)) <**ANDExpression**> }*
<**ANDExpression**> ::= <**NOTExpression**> { ((AND | NAND)) <**NOTExpression**> }*
<**NOTExpression**> ::= [NOT] <**COMPAREExpression**>
<**COMPAREExpression**> ::= <**BASICExpression**>
{ ((< | <= | = | > | >= | !=)) <**BASICExpression**> }*
<**BASICExpression**> ::= <**ADVANCEDExpression**>
{ ((+ | -)) <**ADVANCEDExpression**> }*
<**ADVANCEDExpression**> ::= <**FACTORExpression**>
{ ((* | / | %)) <**FACTORExpression**> }*
<**FACTORExpression**> ::= [((ABSOLUTE | + | -))] <**POWERExpression**>
<**POWERExpression**> ::= <**Increments**> [((^)) <**Increments**>]
<**Increments**> ::= <**PRIMARYExpression**> | ((++ | --)) <Var>
<**PRIMARYExpression**> ::= (<**ORExpression**>) | <**literal**> | <**VID**>

Java++ binary arithmetic operators are addition (lexeme +); subtraction (-); multiplication (*); division (/); integer remainder (%); and exponentiation (^). The evaluation of each binary arithmetic operator always requires 2 int operands and always results in a scalar int.

The unary arithmetic operators are identity (+); negation (-); absolute value (ABSOLUTE, a unary operator, not a function); the side-effecting operator increment (++); and the side-effecting operator decrement (--). Note The value returned by the side-effecting operators is the value after the side-effect is computed. For example, when x is 3, the assignment y << ++x. changes x to 4 (the side-effect) and assigns the value 4 to y.

The signatures of the arithmetic operators shown below are the only combinations of operands and operator allowed. It is a static semantic error when operator operands do not match the allowed signatures; that is, mixed-mode arithmetic expressions are not allowed.

- +: int × int → int
- -: int × int → int

- *: int × int → int
- /: int × int → int
- %: int × int → int
- +: int → int
- -: int → int
- ABSOLUTE: int → int
- ++: int → int
- --: int → int

Bool values may be combined using the bool binary conjunctive operators (AND, NAND); the bool binary disjunctive operators (OR, NOR, XOR); and the bool unary negation operator (NOT). The evaluation of the AND, NAND, OR, NOR, and XOR binary bool operators always requires 2 bool operands and always results in a scalar bool value. The evaluation of the NOT unary bool operator always requires 1 bool operand and always results in a scalar bool value. The signatures of the bool operators are shown below. It is a static semantic error when the bool operator operands are not bool.

- AND: bool × bool → bool
- NAND: bool × bool → bool
- OR: bool × bool → bool
- NOR: bool × bool → bool
- XOR: bool × bool → bool
- NOT: bool → bool

Int can be compared to each other using the less than (<), the less than or equal (<=), the equal (=), the greater than (>), the greater than or equal (>=), and the not equal (!=) binary relational or comparison operators. The relational operators always compute in a scalar bool value. The signatures of the binary comparison operators are shown below. It is a static semantic error when the binary comparison operator does not have 2 same-type operands (Notes (1) bool values cannot be compared. (2) bool and int values may not be intermixed. (3) Java++ is designed to be type sensitive therefore, the data type of the expressions required in the IF-statements and REPEATWHILE-statements must be bool.)

- <: int × int → bool
- <=: int × int → bool
- >: int × int → bool
- >=: int × int → bool
- ==: int × int → bool
- !=: int × int → bool

Java++ is like most other programming languages in that it allows the use of paired brackets in an expression to change the order of operator evaluation from that based solely on the operators' predefined precedencies and/or associativity's.

*For example, the 2 assignment-statements shown below contain expressions with 2 different operator evaluation orders, - * and * -, respectively.*

x << [y-z]*54.
x << y-z*54.

It is a run-time error when an int overflow during the run-time evaluation of an expression. The Java++program terminates with an appropriate error message displayed on the user's terminal.

***Java*++ Operators**	**Precedence**	**Associativity**
++ or --	1 = highest	Non-associative
^	2	Non-associative
ABSOLUTE +(unary) -(unary)	3	Non-associative (unary operators)
* / %	4	Left-to-right
+(binary) -(binary)	5	Left-to-right
< <= == > >= !=	6	Non-associative
NOT	7	Non-associative
AND NAND	8	Left-to-right
OR NOR XOR	9 = lowest	Left-to-right

The **<EITHERStatement>** adds computed value of int or bool expressions, and string literals to the console output buffer using only the minimum number of characters required to format the value being output. An Either-statement can either be a <**COUTStatement**> or a <**COUTLNStatement**> which will be discussed in the next paragraph. (Notes: (1) The bool value true is output as "T" and false as "F"; (2) There is no end line reserved word within Java++ as the coutln-statement automatically causes the output buffer to be displayed as 1 line on the user's terminal (Also using \n within a string will do the same); (3) Java++ does not allow for detailed formatting of value added to the output buffer.)

<**EITHERStatement**> ::= (cout | coutln) ([<string> | <ORExpression>]
{ ; (< string > | <ORExpression>) }*).

Using the "cout" reserved word will transform the Either-statement into a **<COUTStatement>** which adds string literals or expressions to the console output buffer using only the minimum number of characters required to format the value being output and does not go to the next line. However, using the "coutln" reserved word turns into a <**COUTLNStatement**> that

automatically goes to the next line once the string or expression has been output. Multiple strings can be strung together by using the semi-colon (;) prefix.

Here are several examples of PRINT-statements that use the int variable definition of x:

COUTLN-Statement	**Output**
VARIABLE NUM x << +11711. coutln("x is "; x; ".").	x is 11711.
VARIABLE NUM x << -5678. coutln("x is "; x; ".").	x is -5678.
coutln("\'"; [2 <= 2];"\' and "; (2 != 2); ".").	'T' and F.

The **<STDINStatement>** prompts with the string when specified (otherwise prompts with the default prompt "?"), inputs a 1 carriage return-terminated record from the console keyboard, then stores the int or bool-equivalent of the characters contained in the line into the int or bool variable specified. The user's input must be coded according to the syntax rules for Java++ int and bool literals. It is a run-time error when the input characters cannot be converted to the appropriate data value.

<**STDINStatement**> ::= STDIN [<string>] ; <VID>.

For example, the execution of the STDIN-statement shown below

STDIN "x? "; x.

looks like this on the console screen:

x? □

where □ represents the text cursor whereas the execution of the STDIN-statement shown below

STDIN; x.

looks like this on the console screen:

? □

The **<AssignmentStatement>** has the semantics of a classic assignment-statement; namely, the expression is evaluated yielding a scalar int or bool value, then the value is assigned to the 1-or-more scalar variable(s) on the left of the INTO (<<) operator. No coercion of right-hand side data type to left-hand side data type is permitted. It is a static semantic error when the data type of 1-or-more of the variables does not match the data type of the expression.

<**AssignmentStatement**> ::= <**VID**> { ; <**VID**> }* <**INTO**> <**literal**> .

Here are some simple examples,

VARIABLE NUM x; NUM y; BOOLEAN flag1, BOOLEAN flag2.
x << 1.

y << 2.
flag1; flag2 << TRUE.

Or these assignments also work,

VARIABLE BOOLEAN flag1 << TRUE.
VARIABLE NUM x; NUM y << 1.

The **<IFStatement>** has the semantics of the classical if-statement; specifically,

1. The IF expression is evaluated. When the expression is true, the set of statements between the open curly brace ({) and the first OTHERWISEIF or the OTHERWISE or the closed curly brace (}) (if no OTHERWISEIF or OTHERWISE clause are specified) is executed. All the other sets of statements in the IF-statement are not executed.
2. When the IF expression is false, the ELIF expressions are evaluated from top-to-bottom until a true OTHERWISEIF expression is found and its set of statements is executed. The set of statements for bypassed false OTHERWISE expressions are not executed and all the remaining sets of statements in the IF-statement are not executed.
3. When the IF expression is false and every OTHERWISEIF expression is also false, the set of statements between the OTHERWISE and the closed curly brace (}) (if the OTHERWISE is specified) is executed and all other sets of statements are not executed.
4. It is a static semantic error when the IF expression is not bool data type.

<**IFStatement**> ::= IF [<**ORExpression**>]
(({))
{ <**Statement**> }*
{ OTHERWISEIF [<**ORExpression**>] EXECUTE
{ <**Statement**> }* }*
[OTHERWISE
{ <**Statement**> }*]
((}))

The **<REPEATWHILEStatement>** is a combo form of both of the classic unbounded loops; this is, it can be used as a pre-test loop and a post-test loop, but in it most general form it is an unbounded mid-test loop! It is a static semantic error when the WHILE expression is not bool data type. (Note: It is not a static semantic error when both sets of nested statements are empty, but it is probably a logic error.) Here are its semantics.

<**REPEATWHILEStatement**> ::= REPEAT
{ <**Statement**> }*
WHILE [<**ORExpression**>]
(({))
{ <**Statement**> }*
((}))

1. The statements between REPEAT and WHILE are executed unconditionally.
2. The expression is evaluated.

3. When the expression is false, go to step 6.
4. Execute the statements in the body of the loop between the open curly brace ({) and the closed curly brace (}).
5. Go to step 1.
6. Next-statement after the REPEATWHILE-Statement.