OddBall Language Reference Manual

John Baldwin

Shrimohan Damani

Peter DePasquale

Caroline Larboulette

Michael Sonsini

Prashanth Suthrave

Jidesh Veeramachaneni

OddBall Language Reference Manual
by John Baldwin, Shrimohan Damani, Peter DePasquale, Caroline Larboulette, Michael Sonsini, Prashanth Suthrave, and Jidesh Veeramachaneni

$Revision: 1.21 $ Edition
Published $Date: 1999/12/16 14:38:26 $

This manual is intended to server as a user's manual to the OddBall programming language. It is geared towards programmers who have had previous experience with an imperative programming language.

Table of Contents

List of Tables

List of Examples

Chapter 1. Purpose

The purpose of the OddBall programming language is to provide the programmer with a fundamental and sufficient set of constructs with which to solve problems. Particularly, problems with a solution of a mathematical nature are easy to model by the language. Further, some of the constructs found in OddBall attempt to improve upon corresponding constructs found in other languages such as C.

Chapter 2. Name Value System

Identifiers are used to associate names with values in the OddBall language. Two primitive data types (integers and characters) in OddBall. We also support arrays of integers and characters. Each identifier is bound to a data type at compile time, and the type relationship can not be modified at run time.

Identifiers are identified by no more than 75 alphanumeric characters, the first of which must be an `<alphabet>`.

2.1. Scalars

There are two scalar types in OddBall: integer and character. The integer type consists of a signed integer that is the size of the machine word. For ease of explanation, the machine word size is assumed to be 32 bits throughout this manual. The character type is a signed 8 bit integer. There is also a special integer constant named `UNDEFINED`. This constant is used to indicate an unknown, invalid, or undefined value. It is also used as the default return value for functions that do not explicitly return a value. This second property is used to define this value as follows:

Example 2-1. Formal Definition of UNDEFINED

```
DEFINE undefined_function () RUNS
STOP.

VAR
   UNDEFINED : integer.
STOP.

   ...
   undefined_function() -> UNDEFINED.
   ...
```

For now, the `UNDEFINED` constant will use the smallest possible integer value. This is so that the constant impinges minimally on the range of the integer type. For example, if the integer type is 32 bits, then `UNDEFINED` would be `0x80000000`. However, this is an implementation specific detail, and programmers should not rely on this detail.

Table 2-1. Properties of Scalar Types

Type	Range	Operators
integer	-2147483647 ... 2147483647	<, >, <=, >=, =, +, -, *, /, %, ->, ,

character	-128 ... 127	->, ,

2.2. Type Checking

OddBall is designed to be a flexible language that is easy to use. Thus, it uses a very simplistic view of types. OddBall is a weakly-typed language. It does not set strict boundaries between characters and integers. Instead, character constants and variables are automatically promoted to integers before being evaluated in an expression. If the receiving variable of an assignment operator is a character, then the value of the expression on the right side of the operator is automatically demoted to a character just before assignment.

2.3. Arrays

In addition to scalar types, OddBall also includes one dimensional arrays of both of its scalar types. Arrays are allocated statically with a constant, fixed size in a `VAR` statement. Arrays may be passed to functions, but they are passed by reference, not by value.

Array indices are integers in the range 1 ... *`array size`*. Individual elements of an array may be indexed by postfixing the array variable name with an integer expression contained in bracket ([]) characters. For example, if `myArray` is an array of 10 integers, then `myArray[4]` references the 4th integer in the array.

2.4. The `VAR` Statement

The `VAR` statement is used to declare variables. It consists of the keyword `VAR` followed by a whitespace character and then zero or more variable declaration statements. Each variable declaration statement consists of a comma-separated list of one or more variables followed by a colon and a scalar type. Scalars are declared by simply listing their name in a comma-separated list. Arrays are declared by listing their name followed by the keyword `OF` followed by an array size. The example below declares a scalar integer `foo`, a scalar character `bar`, and an integer array `baz` of 10 elements.

Example 2-2. Variable Declarations

```
VAR
   foo, baz OF 10 : integer.
   bar : character.
```

```
STOP.
```

Chapter 3. Control Structures:

OddBall supports two basic control structures. The first is the `COND-LOOPS` statement, which repeats the execution of a specific section of code, so long as a test condition is met. It is analogous to the conventional `while` loop statement in C. To construct a `COND-LOOPS` statement that displays a character to the screen a variable number of times write:

Example 3-1. A Simple Loop

```
COND (localInt > 0) LOOPS
   WRITE someChar.
   localInt - 1 -> localInt.
STOP.
```

This loop writes someChar to the screen localInt times. On each iteration of the loop the value of localInt is decremented by one until finally it reaches 0. Since the conditional expression is localInt > 0 the loop terminates at this point. For an explanation of the `WRITE` statement see the Input and Output section.

The second control is the `COND-RUNS-OTHERWISE` statement. This statement will execute the code following the `RUNS` portion of the statement as long as a test condition is met. If the optional `OTHERWISE` portion of the statement appears and the test condition is not met, the code following the `OTHERWISE` portion is executed. It is analogous to the conventional `IF-THEN-ELSE` statement C. To construct a `COND-RUNS-OTHERWISE` that determines the maximum of two integers and stores the result in a third integer write:

Example 3-2. Conditional Execution (Branching)

```
COND (firstInt > secondInt) RUNS
   firstInt -> maximumInt.
STOP OTHERWISE RUNS
   secondInt -> maximumInt.
STOP.
```

This conditional statement begins by determining if firstInt is greater than secondInt. If this is true the code immediately following the `RUNS` keyword executes and firstInt is stored in maximumInt. Execution continues until the `STOP OTHERWISE` keywords are encountered. At this point execution continues from the next statement following the `STOP OTHERWISE` keywords. If the condition is not true and thus firstInt is not greater than secondInt the code immediately following the `OTHERWISE` keyword executes and secondInt is stored in maximumInt. Execution continues until the `STOP` statement is encountered at which point execution proceeds starting from the next statement appearing after the `STOP` statement.

Chapter 4. Imperatives

Imperatives in OddBall include assignment statements, branches, loops, mathematical expressions, and function calls.

The OddBall language, like other imperative language, supports assignment statements, various kinds of operators, and other expressions.

4.1. Assignment Statements

The assignment operator is used to assign the value of an expression to a scalar variable. The assignment operator is represented by the two-character symbol `->`. In the assignment statement the target variable in which the value is to be stored is on the right hand side of the assignment operator and the expression whose value is to be calculated and assigned is written on the left side.

Example 4-1. A Simple Assignment

```
a + b -> c.
```

This statement calculates the sum of the values stored in the memory locations represented by `a` and `b` and stores the resulting value in the memory location represented by `c`.

Here the element on the right hand side of the assignment operator must be a scalar variable. The assignment operation cannot be mapped to numeric constants on the right hand side. The left hand side can consist of any expression.

4.2. Expressions

Expressions consist of operands, or terms, which are joined by operators.

4.2.1. Operands

Operands in an expression are either variables, constants, or function calls. The value of a variable is the value of the memory location represented by that variable. The value of a function call is the value returned by that function. Integer constants are represented in decimal. Character constants are represented in single quotes. The single quote character (') is escaped by itself. Unprintable characters are represented by a `#` character followed by a decimal value all of which is enclosed in quotes.

Example 4-2. Valid Operands

```
fooBar $$ A variable
sqrt(2) $$ A function call
4 $$ An integer constant
’a’ $$ A character constant
”” $$ The ’ character constant
’#10’ $$ A newline character
```

4.2.2. Operators

OddBall supports two kind of operators: arithmetic operators and logic operators.

4.2.2.1. Arithmetic Operators

Table 4-1 lists the arithmetic operators supported by OddBall along with their precedence. Each of the operators take scalar arguments and apply to both integers and characters.

Table 4-1. Arithmetic Operators

Operator	Symbol	Precedence	Associativity	Type
Grouping	()	+5	left-to-right	unary
Multiplication	*	5	left-to-right	binary
Division	/	5	left-to-right	binary
Modulus	%	5	left-to-right	binary
Negation	-	4	right-to-left	unary
Addition	+	3	left-to-right	binary
Subtraction	-	3	left-to-right	binary
Comparison Operators	<, >, =, <=, >=	2	left-to-right	binary
Assignment	->	1	left-to-right	binary
Concatenation	,	0	left-to-right	binary

All of the arithemetic operators perform their expected function.

The comparison operators return a value of 0 if the comparison fails and a value of 1 if the comparison succeeds.

4.2.2.2. Logical Operators

OddBall does not include built-in support for logical operators such as `and`, `or`, `xor`, and `not`. However, since the comparison operators return 0 or 1, logical operators can be easily simulated via arithmetic.

To emulate `and`, simply multiply the results of two comparisons together. If both comparisons are true, then the multiplications will return 1, otherwise, it will return 0.

To emulate inclusive `or`, add the results of two comparisons together. If either of the results, or if both of the results are true, then the addition will return a value greater than 0.

To emulate exclusive or, `xor`, add the results of two comparisons and then test the result to see if it is equal to 1. This effectively removes the inclusive case from the inclusive or defined above.

To emulate `not`, test the result of a comparison to see if it is equal to 0. If the comparison failed and returned 0, then the test will return 1. If the comparison succeeded and returned 1, then the test will return 0.

The following code fragment demonstrates these four operations.

Example 4-3. Emulation of Logical Operators

```
$ emulate the condition (a > b) AND (b > c)
cond ((a > b) * (b > c)) runs
   writeln "A is greater than B and B is greater than C".
stop.

$ emulate the condition (a > b) OR (b > c)
cond ((a > b) + (b > c)) runs
   writeln "A is greater than B or B is greater than C or both".
stop.

$ emulate the condition (a > b) XOR (b > c)
cond ((a > b) + (b > c) = 1) runs
   writeln "A is greater than B or B is greater than C, but not both".
stop.

$ emulates the condition NOT (a > b)
cond ((a > b) = 0) runs
   writeln "A is not greater than B".
stop.
```

4.2.2.3. Precedence

Precedence determines the order in which operators are evaluated. Operators with higher precedence are executed before those with lesser precedence. Thus, `a * b + c` is evaluated as `(a * b) + c`, and `a + b * c` is evaluated as `a + (b * c)`. The parentheses, or grouping operators, are special with regards to precedence. They are only pseudo-operators who do not have an actual precedence. Instead, they increase the precedence of all operators in the subexpression that they contain. Thus, in the expression `(a + b) * (c + d)`, the `+` operators have higher precedence then the `*` operator, so they are evaluated first.

4.2.2.4. Associativity

Associativity determines the order in which operators of the same precedence are processed. “Left-to-right” operators give higher precedence to operators that are farther to the left in an expression. For example, `a * b / c` is evaluated as `(a * b) / c` instead of `a * (b / c)`. Similarly, “right-to-left” operators give higher precedence to operators that are farther to the right. Thus, `--a` is evaluated as `-(-a)` instead of `(-)a` (which does not make much sense).

Chapter 5. Input and Output

Both input and output are supported by OddBall. The `READ` statement allows for the user to input the value of an integer, a single character or an array of characters (string), depending on the destination of the value.

Output is achieved with either of the `WRITE` or `WRITELN` statements which print strings, variables (or combinations of both) to the output medium (standard output). The `WRITE` statement suppresses a trailing new line character after the output is printed, the `WRITELN` command adds it automatically.

A proposed web based interface to OddBall is being researched. Because OddBall can take interactive input on the command line a command line emulator may be necessary. We believe that emulating the command line in a web interface is beyond the scope of this project. However, much of the functionality of OddBall can be demonstrated in a web interface with little additional development. Therefore, a decision has been made to complete the initial OddBall compiler and if time permits, to develop a web interface.

Chapter 6. Abstractions

OddBall contains abstractions for functions and programs. The `program` function is the main executable body and programmatic starting point. Its abstraction properties are the same as that of any other function

Subprograms, or functions, are defined by the `DEFINE` statement, which denotes the function's name, parameter list and function body. Every function's name is an identifier, following the same rules as variable identifiers (see above). The parameter list is defined to be zero or more variable identifiers and their types. Integer and character parameters are passed by value to subprograms. Arrays of integers and characters are passed by reference.

Functions can not change the value of scalar parameters, although they can change the elements of array parameters. Functions can change the value of global variables. Also, like Pascal and C, local variables take precedence over global variables in case of a variable naming conflict. As far as semantics are concerned, function parameter variables are treated exactly the same as local variables.

A function cannot be called until it has been defined with the `DEFINE` statement.

6.1. The `DEFINE` Statement

The `DEFINE` statement defines the data and actions associated with a function. The `DEFINE` keyword is followed by the function name (the first executable body has the fixed name *program*). Next comes the parameter list. The parameter list consists of a list of zero or more variables along with their types inside of parentheses. After this comes the keyword STOP followed by the function body, which is a sequence of declarative and/or executable statements. Every functions ends with the `STOP` keyword followed by a period.

6.2. Function Names

As defined in the BNF, function names are identifiers which follow the same restrictions as variable names.

6.3. Function Parameters

The parameter list is a list of variables passed to the function from the calling function. A function can have any number of arguments. Scalars are passed by value, and arrays are passed by reference. Thus, if a function `foo` is defined to receive two variables `a` and `b` which are of type integer and array, respectively, then `a` is passed by value and `b` is passed by reference. If the function changes the value of `a` and and the value of an element of `b`, only the change in `b` is reflected back to the calling function.

6.4. Function Body

A function body can consist of zero or more executive or declarative statements. Local variables may be defined in a `VAR` declarative statement.

6.5. Return Values

Every function returns a value. A function can set the value to be returned by assigning a value to an implicitly declated local integer variable whose name is identical to that of the function's. Thus, it is a sematic error to declare a local variable with the same name as the function it is declared in. If an explicit return value is not assigned, then the function will return the constant `UNDEFINED`.

Appendix A. Language Definition in EBNF

Here is the formal syntax for OddBall in extended BNF.

```
<program> ::= { <declaration> . }+
<declaration> ::= <functiondef>  | <variabledecls>
<functiondef> ::= DE-
FINE <whitespace> <identifier> ({<parameters>}0/1) <actions> STOP
<identifier> ::= <alphabet> {<idchar>}*
<parameters> ::= <varlist> {;<varlist>}*
<varlist> ::= <variabledecl> {,<variabledecl>}* :<type>
<variabledecl> ::= <identifier> {<whitespace> OF <constant>}0/1
<type> ::= INTEGER |  CHARACTER
<variabledecls> ::= VAR <whitespace> {<varline>}0/1 STOP
<varline> ::= {<varlist> .}+
<actions> ::= {<action> .}*
<action> ::= <expression> | <input> | <output> | <variabledecls> | <conditional> | <loo
<expression> ::= <condition> {-> <variable>}0/1
<condition> ::=  <term> {<relation> <term>}*
<relation> ::= = | < | > | <= | >=
<term> ::= <simplefactor> {<termop> <simplefactor>}*
<termop> ::= + | -
<simplefactor> ::= {-}* <factor>
<factor> ::= <subexpression> {<factorop> <subexpression>}*
<factorop> ::= * |  / | %
<subexpression> ::= <variable> | <constant> |  (<expression>) | <functioncall>
<variable> ::= <identifier> {[<expression>]}0/1
<conditional> ::= COND (<expression>) RUNS <actions> STOP {<whitespace> OTH-
ERWISE <whitespace> RUNS <actions> STOP }
<loop> ::= COND (<expression>) LOOPS <actions> STOP
<functioncall> ::= <identifier> (<parameterlist>)
<parameterlist> ::= <expression> { ,<expression>}*
<input> ::= READ <whitespace> <variable>
<output> ::= <sameline> |  <nextline>
<sameline> ::= WRITE <whitespace> <showlist>
<nextline> ::= WRITELN <whitespace> <showlist>
<showlist> ::= <showitem> {,<showitem>}*
<showitem> ::= "<text>" | <expression>
<text> ::= {<textconstant>}+
<alphabet> ::= [A-Za-z]
<digit> ::= [0-9]
<idchar> ::= <alphabet> | <digit> | _
<constant> ::= <integer> | <character> | UNDEFINED
<character> ::= '<characterconstant>' | #integer
```

```
<textconstant> ::= <printabletextchar> | ""
<characterconstant> ::= <printablechar> | ″
<printablechar> ::= all printable characters except ’
<printabletextchar> ::= all printable characters except for "
<integer> ::= {<digit>}+
<whitespace> ::= space character | tab character | newline
```

Appendix B. Sample Program

This program demonstrates most of the statements of OddBall.

```
$$ Mess around with some random output
DEFINE subfunc (anInt : integer; someChar: character) RUNS
   VAR localInt : integer STOP.

   anInt -> localInt.

   WRITELN "This is the output from the subfunction".
   WRITELN "The parameters were ", anInt, "and ", someChar.
   COND (localInt > 0) LOOPS
      WRITE someChar.
      localInt - 1 -> localInt.
   STOP.
STOP.

$$ The program function defines the start of execution for the main program.
DEFINE program () RUNS
   VAR getInteger : integer.  $$ a scalar integer variable
       getChar : character.  $$ a scalar character variable
   STOP.

   $$ Prompt for and read in an integer
   WRITELN "Enter an integer.".
   READ getInteger.

   $$ Prompt for and read in a character
   WRITELN "Enter a character.".
   READ getChar.

   subfunc (getInteger, getChar).
STOP.
```

Appendix C. Sample Program

This function takes the positive x co-ordinates of two points on a number line. The functions determines the minimum co-ordinate of the two. It simulates the movements of the lower point towards the other point using textual output. Finally it returns the number of steps traveled to the calling function.

```
define cover_distance ( a,b : integer ) runs
    var
        equal,min : integer.
        steps : integer.
    stop.

    0 -> equal.
    cond( a < b ) runs
        a -> min.
    stop
    otherwise runs
        cond ( a > b ) runs
            b -> min.
        stop
        otherwise runs
            1 -> equal.
        stop.
    stop.

    cond ( min = a ) runs
        cond ( a < b ) loops
            a + 1 -> a.
            steps + 1 -> steps.
            writeln   "point a reaching point b".
        stop.
    stop
    otherwise runs
        cond ( min = b ) runs
            cond ( b < a ) loops
                b + 1 -> b.
                steps + 1 -> steps.
                writeln   "point b reaching point a".
            stop.
        stop.
    stop.

    cond ( equal = 1 ) runs
        2 -> cover_distance.
```

```
    stop
    otherwise runs
        steps -> cover_distance.
    stop.
stop.

define program () runs
    var
        a,b,result : integer.
    stop.

    5 -> a.
    10 -> b.

    cover_distance( a, b ) -> result.

    cond ( result = -2 ) runs
        writeln   "The points had the same horizontal coordinates.".
    stop
    otherwise runs
        writeln   "The number of steps covered is ", result.
    stop.
stop.
```