OpenMath Specification
Draft
Last Updated 94–Jul–12

1. Introduction

This document is a specification for a proposed OpenMath standard allowing applications which perform, use, or otherwise handle mathematics to communicate with one another.

The basic architecture of OpenMath is that of a client and a server. An OpenMath client can establish connections with one or more servers. For example, a mathematical word processor might want to talk to a Computer Algebra System (CAS) and a mathematical typesetting engine.

Communication between the client and the server is by means of ASCII strings. These string are used to write in a LISP-like language of nested expressions. Such a language is both simple, and powerful enough to express most any mathematical expression (and any attributes one wishes to attach to it). At the same time, it is does not favour a particular CAS or other potential OpenMath client or server.

The communication protocol is non-blocking on those platforms where this is possible (multitasking platforms). A request is sent to a server, but the client is not obliged to wait for the server to finish before it carries on with its own work.

2. Syntax of OpenMath

This section describes the syntax of OpenMath. Information about the semantics is given in Section 3: *Semantics of OpenMath*.

2.1 The Grammar

The following EBNF grammar defines the syntax of OpenMath. Communication between OpenMath clients and servers is in terms of Expression Sequences. Symbols enclosed by "<" and ">" in the grammar indicate lexical tokens, as defined in the Section 2.2: *Lexical Definitions*.

```
Expression ::= <Integer>
      | <BasedInteger>
      | <RealNumber>
      | <FloatingPointRepresentation>
      | <String>
      | <Symbol>
      | <BackReference>
      | ExpressionSequence

ExpressionSequence ::= "(" [ <Label> ] Expression { [ <Label> ] Expression } ")"
```

Notice that an Expression Sequence cannot be empty.

2.2 Lexical Definitions

The following are the individual lexical tokens from which expressions are constructed. Character classes mentioned in this section are defined in the next section. Note that Whitespace Characters may not appear within tokens other than Strings. Note that arbitrary sequences of zero or more Whitespace Characters may appear between tokens. At least one Whitespace Character is generally required between Integer, Based Integer, Real Number, Floating Point Representation, and Symbol tokens to prevent ambiguity.

It is recommended that OpenMath expressions contain a Whitespace Character that is interpreted as a text line separator by the local computing environment at least once every 80 characters. This will ensure compatibility with various text-based interchange mechanisms, such as electronic mail.

For similar reasons, it is also recommended that OpenMath expressions do not contain characters with ordinal values beyond 126, and do not contain significant characters with ordinal values below 32.

Parentheses

The characters "`(`" and "`)`" (ordinal values 40 and 41), when appearing outside of a String, are used to enclose sub-expressions.

Integers

An integer is a maximal sequence of one or more digits ("`0`" through "`9`"), optionally preceded by a "`-`" character (ordinal value 45). It is interpreted as a base-10 integer value. The largest allowable integer magnitude is client and/or server dependent.

Examples: `0` `-23` `398432348985435995433834`

Based Integers

A based integer is the "`$`" (ordinal value 36) character, followed immediately by an integer in the set {2, 3, ..., 16} specifying the number base (*b*) being used, followed immediately by the "`,`" character (ordinal value 44), followed immediately by a maximal sequence of one or more digits ("`0`" through "`9`", "`A`" through "`F`", or "`a`" through "`f`") optionally preceded by a "`-`" character (ordinal value 45). It is interpreted as an integer value in the specified base. Digits not valid in the specified base are not permitted in the digit sequence. Note that these are signed values; **not** *b*'s complement. The largest allowable integer magnitude is client and/or server dependent.

Examples: `$10,17129` `$16,-42E9` `$2,100001011101001` `$5,-23134`

Real Numbers

A real number is a maximal sequence of one or more digits containing exactly one "`.`" character (ordinal value 46), and optionally preceded by a "`-`" character (ordinal value 45). This may optionally be followed by the "`E`" or "`e`" character and an integer representing a base-10 exponent. The set of allowable real values is client and/or server dependent.

Examples: `1.` `.1` `1.0` `0.1` `-1.1` `3.E2` `.314E1` `31.4e-1`

Floating Point Representation

A floating point representation is used when it is desired to communicate an IEEE compatible double precision (64-bit) floating point value. The representation used is the character "`%`" (ordinal value 37), followed immediately by 16 hexadecimal digits, corresponding to the hexadecimal representation of the 8 bytes representing the floating point number, starting with the bytes representing the sign and exponent (note that this is independent of the byte ordering of the internal representation on any platform). For example, the following is a table of some floating point numbers and their Floating Point Representation equivalent:

1.0	`%3FF0000000000000`
-1.0	`%bff0000000000000`
2.0	`%4000000000000000`
-2.0	`%C000000000000000`
1.0e10	`%4202a05f20000000`
1.0e-10	`%3DDB7CDFD9D7BDBB`

Strings

A String is a sequence of zero or more Characters enclosed in double quotation marks (`"""`, ordinal value 34). If the either of the Special Characters `"""` or `"\"` appear in the string, they must be preceded by a `"\"` character (ordinal value 92). A `"\"` character (unless preceded by another `"\"` character) can also introduce a control character, as follows:

`"\a"`	-bell character (BEL, ordinal value 7).
`"\b"`	-backspace character (BS, ordinal value 8).
`"\e"`	-escape character (ESC, ordinal value 27).
`"\f"`	-formfeed character (FF, ordinal value 12).
`"\n"`	-linefeed character (LF, ordinal value 10).
`"\r"`	-carriage return (CR, ordinal value 13).
`"\t"`	-horizontal tab character (HT, ordinal value 9).
`"\v"`	-vertical tab character (VT, ordinal value 11).

Note that any Character (recall that the byte with ordinal value 0 is not considered a Character) can appear directly in a String; it is not necessary to use one of the control sequences shown above.

Furthermore, a `"\"` followed by a Whitespace Character is ignored entirely. This allows one, for example, to insert line breaks in the OpenMath representation of a string without inserting line breaks in the string itself.

The maximum length of a String is client and/or server dependent.

Examples: `"He said, \"Don't!\"\n"` `"This is a test"` `"5# # $3.00"`

Symbols

A Symbol is a sequence of one or more non-Whitespace Characters, not beginning with `"""`, `"#"`, `"$"`, `"%"`, `"-"`, `"."`, `"0"` through `"9"`, `"("`, `")"`, or `"\"`, and not containing `")"`. As a special exception, the Character `"-"` by itself is allowed as a Symbol.

The maximum length of a Symbol is client and/or server dependent.

Examples: `x` `num12` `Hello_world` `_x` `+` `<=` `=\=` `a/b` `-`

Labels

A Label consists of the character `"#"` (ordinal value 35), followed by a non-negative Integer, followed by the Character `":"` (ordinal value 58). The integer value of the label should be small enough to be represented in a 32-bit machine word (eg. $0 \le$ value ≤ 2147483647). There is no requirement that labels be generated sequentially, nor that they consist of consecutive values. For example, an expression may contain exactly three labels, `#123`, `#2`, and `#95`, in that order.

Examples: `#0:` `#123:`

Back References

A Back Reference consists of the character "#", followed by a non-negative Integer, and represents the sub-expression following the corresponding Label (which must already have appeared in the expression). If a label is redefined in the input stream, all subsequent references refer to the new definitions, but all earlier references refer to the old definition.

Examples: `#0` `#123`

The following are the classes of characters from which tokens are constructed:

Characters

A Character is the interpretation of the contents of one non-zero byte.

Visible Characters

A Visible Character is any Character whose ordinal value lies between 33 and 255.

Whitespace Characters

A Whitespace Character is any Character that is not a Visible Character.

Special Characters

A Special Character is one that has special meaning to OpenMath and must be suitably escaped when it appears within a string. The Special Characters are: """ (ordinal value 34) and "\" (ordinal value 92). A Special Character can be forced to be accepted as an ordinary Visible Character within a String by preceding it with "\" (ordinal value 92).

Digits

The Digits are "0" (ordinal value 48) through "9" (ordinal value 57).

3. Semantics of OpenMath

An Expression Sequence consists of one or more Expressions. The first Expression within an Expression Sequence is interpreted as an operator, with the remaining Expressions interpreted as operands.

There are three levels of operators in OpenMath: Computation Level, Structure Level, and Communication Level.

There is a collection of Computation Level Operators, outlined in Section 3.1: *Computation Level Operators*, which are part of the standard. Additional collections of Computation Level Operators can be supported. Such collections are organized into *Packages* (refer to Section 7: *Packages and Future Extensions*).

Similarly, there is a collection Structure Level Operators, outlined in Section 3.2: *Structure Level Operators*, which are part of the standard. Packages may add additional collections of Structure Level Operators.

3.1 Computation Level Operators

The meanings of operators are system dependent, but the following symbols, representing most of the function and constant names defined in Abramowitz and Stegun, Chapters 1, 3, and 4, should be taken as a standard subset of all possible operators. Functions whose names consist of non-Roman characters, and which do not have a commonly accepted Roman spelling, are named using the ISO-8879 SGML entities, using "&" (ordinal value 38) and "." (ordinal value 46) as the delimiters:

Expression	**Mathematical Meaning**
Arithmetic Operators	
(+ *x y* ...)	$x + y$
(* *x y* ...)	$x \bullet y$
(- *x*)	$-x$
(- *x y*)	$x - y$
(/ *x*)	$1 \div x$
(/ *x y*)	$x \div y$
(mod *x y*)	$x \pmod{y}$
(pow *x y*)	x^y
(! *x*)	$x!$
(| x) [1]	$\lvert x \rvert$
(sign *x*)	$-1\ \forall x < 0,\ 1\ \forall x > 0,\ 0$ when $x = 0$
(&lfloor. *x*) [1]	$\lfloor x \rfloor$
(&lceil. *x*) [1]	$\lceil x \rceil$
(trunc *x*)	$\lfloor x \rfloor\ \forall x \geq 0,\ \lceil x \rceil\ \forall x < 0$
(round *x*)	$\lfloor x+0.5 \rfloor\ \forall x \geq 0,\ \lceil x-0.5 \rceil\ \forall x < 0$
(&radic. *x*)	$\sqrt{x}$
Relations	
(= *x y*)	$x = y$
(&ne. *x y*)	$x \neq y$
(< *x y*)	$x < y$
(&le. *x y*)	$x \leq y$
(> *x y*)	$x > y$
(&ge. *x y*)	$x \geq y$
Transcendental Functions	
(ln *x*)	$\ln x$
(log10 *x*)	$\log_{10} x$
(log *x y*)	$\log_x y$
(exp *x*)	e^x
(sin *x*)	$\sin x$
(cos *x*)	$\cos x$
(tan *x*)	$\tan x$
(csc *x*)	$\csc x$
(sec *x*)	$\sec x$
(cot *x*)	$\cot x$
(arcsin *x*)	$\arcsin x$
(arccos *x*)	$\arccos x$
(arctan *x*)	$\arctan x$
(arccsc *x*)	$\operatorname{arccsc} x$
(arcsec *x*)	$\operatorname{arcsec} x$
(arccot *x*)	$\operatorname{arccot} x$
(sinh *x*)	$\sinh x$
(cosh *x*)	$\cosh x$
(tanh *x*)	$\tanh x$

(csch *x*)	$\operatorname{csch} x$
(sech *x*)	$\operatorname{sech} x$
(coth *x*)	$\coth x$
(arcsinh *x*)	$\operatorname{arcsinh} x$
(arccosh *x*)	$\operatorname{arccosh} x$
(arctanh *x*)	$\operatorname{arctanh} x$
(arccsch *x*)	$\operatorname{arccsch} x$
(arcsech *x*)	$\operatorname{arcsech} x$
(arccoth *x*)	$\operatorname{arccoth} x$

Calculus Operators

(diff *x y*)	$\frac{d}{dy} x$
(&int. *x y*)	$\int x \, dy$
(&int. *x y a b*)	$\int_a^b x \, dy$
(&sum. *x y a b*)	$\sum_{y=a}^{b} x$
(&prod. *x y a b*)	$\prod_{y=a}^{b} x$
(lim *x y z*)	$\lim_{y \to z} x$
(lim *x y z* -)	$\lim_{y \to z^-} x$
(lim *x y z* +)	$\lim_{y \to z^+} x$

Boolean Functions

(&comp. *x*)	$\neg x$
(&and. *x y* ...)	$x \wedge y$
(&or. *x y* ...)	$x \vee y$

Set Operators

(&cup. *x y* ...)	$x \cup y$
(&cap. *x y* ...)	$x \cap y$
(&setmn. *x y*)	$x \setminus y$
(&isin. *x y*)	$x \in y$

Constants

&pi.	π
&gamma.	γ
&infin.	∞
true	$x \vee \neg x$
false	$x \wedge \neg x$

Notes:

1. Each of these can contain an optional second argument which is the correponding closing delimiter, ie. (| x |), (&lfloor. x &rfloor.), and (&lceil. x &rceil.)

Note that none of these predefined operators have any detailed semantics associated with them. For example, it is not necessary that "sin" means "compute the floating point sine of", but only that it means "sine of". A mathematical formatting application would know to format this as a call to the function "sin()", whereas a floating point server would know to evaluate the sine of the numeric operand numerically.

When adding new operators, those whose names are or contain non-Roman characters should adopt the conventions of the ISO standard to rename these using Roman characters. The character "`&`" (ordinal value 38) should be used to introduce such names, and the character "`.`" (ordinal value 46) should be used to terminate such names. For example, the Greek character α would be renamed as "`&alpha.`".

3.2 Structure Level Operators

The Structure Level operators are used to build structures larger than individual Integers, Real Numbers, Strings, or Symbols. The following Structure Level operators comprise the standard subset:

Expression	**Meaning**
(ASSIGN x y)	$x \leftarrow y$
(APPLY x y ...)	$x(y \ldots) \; \forall x$ - This is used if one of the structure or communication level operators is needed as a function name.
(UNEVAL x)	x, but not evaluated.
(NAME x y ...)	$xy\ldots$
(COMPLEX x y)	x + yi
(SET x y ...)	$\{ x, y, \ldots \}$
(ARRAY n $d_1 \ldots d_n$ $a_{1\ldots1}$ $a_{1\ldots2} \ldots a_{d_1 \ldots d_n}$)	An n-dimensional array, the ith dimension of which has indices ranging from 1 ... d_i, and with the optionally specified elements listed in order of most rapidly changing last dimension.
(INDEX x y ...)	$x[y, \ldots]$
(NATIVE "*string*")	Process *string* in recipient's native language.
(ATTRIB *expr* $attr_1 \ldots attr_n$)	Attributed expression. Refer to Section 4: *Attributes*.

3.3 Communication Level Operators

The Communication Level operators are used to control the link between an OpenMath client and server. They do not relate directly to data or operations handled by the client or server. The standard subset is:

Expression	**Meaning**
(START)	Start (or restart) session.
(STARTED)	Response to (START).
(STOP)	End session.
(STOPPED)	Response to (STOP).
(DO *x*)	Execute expression. If an expression is already executing, cancel it and execute the new expression instead. Server will return (INTR) for the interrupted expression.
(SUSP "*string*")	Ask server to suspend session and give it the specified name.
(SUSP)	Server acknowledges suspension.
(RESUME "*string*")	Tell server to resume specified session.
(RESUMED)	Server acknowledges resumption.
(SYNTAX "*string*" *n*)	Server reports the specified error in the input, at offset *n* from the beginning of the input.
(ERROR "*string*")	The server reports that an error has occurred during computation.
(INTR)	Indicate that the previous (DO *x*) was interrupted by a newer (DO *x*).
(RESULT *x*)	Final returned result.
(DIAG "*string*")[1]	Debugging output from server.
(INFO "*string*")[1]	Information output from server.
(PRINT *expr*)[1]	Side-effect output from server.
(IDENT)	Request identification.
(IDENT "*string*")	Identification; response to (IDENT).
(PACKS)	Query which packages are understood.
(PACKS "*string*" "*string*" ...)	Indicate which packages are understood;

response to (PACKS).

Notes:

1. Not all servers will produce these expressions. It is only necessary that an OpenMath client recognize these expressions and process them appropriately (appropriately may mean discarding them in some applications).

Communication Level operators can appear only in the top level expression.

4. Attributes

Any expression can be tagged with attributes. These attributes may or may not affect the meaning of the expression, depending on the client or server processing the attributes. An expression *expr* can have attributes attached to it thus:

(ATTRIB *expr attr1 attr2 ... attrN*)

The individual attributes are themselves expressions. A client or server that does not recognize an attribute should ignore it. It may attempt to preserve attributes when possible. Here are some sample attributes:

(COLOR exprR exprG exprB)
(SIZE expr)
(FONT expr)
(STYLE expr)

Whether or not a particular client or server will recognize a particular attribute depends on that client or server. The only requirement is that an attributed name be recognized as a name. There is no requirement that two identical expressions with two different sets of attributes be considered the same expression, or that they be considered different expressions.

5. Efficiency Considerations

Using text strings makes OpenMath slightly less efficient than it might otherwise be, but this is more than made up for by the ease of interfacing with other applications (especially when debugging). However, the textual interface has been designed with efficiency in mind.

The syntax of OpenMath is very simple, making it possible to write a very efficient parser for it. Only one character of look-ahead is required, and no token look-ahead is required.

Error recovery is also quite simple, since it is only necessary to count opening and closing parentheses until one is back at the top level.

Most whitespace not part of a String can be elided, thereby reducing communication and parsing overhead. The only exception is whitespace between Integers, Real Numbers, Symbols. For example, the expression,

```
(sin (+ x y (- (pow (NAME "a" x) 2))))
```

can be (but need not be) written as:

```
(sin(+ x y(-(pow(NAME "a"x)2))))
```

Most of the Structure Level and Communication Level symbols are short (6 characters or less), reducing the time required to transmit them, and making it easy for a lexical analyzer to recognize them quickly.

6. Communications Media

OpenMath can be communicated in a variety of ways.

Under UNIX, the most obvious way is by pipe or socket. An OpenMath kernel can be implemented which simply reads from the standard input and writes to the standard output. Such a kernel can then be communicated with through a pipe. Using sockets on the other hand allows a more general OpenMath server implementation.

On Macintosh, communication can be done using Apple Events.

Under Microsoft Windows, communication can be done either by DLL function call, by Dynamic Data Exchange (DDE), or using sockets (WINSOCK). Under DOS, communication would have to be by function call, since there is no multi-tasking.

On NeXT, there is Speaker/Listener (in addition to the normal UNIX communication media).

On Amiga, there is the ARexx port.

The issue of which communication medium to use is more-or-less independent of the design of OpenMath itself.

7. Packages and Future Extensions

The standard allows extension by the addition of *Packages*, which consist of a collection of Computation and/or Structure Level Operators, and possibly additional Attributes. A package has a name, represented as a String, and the presence of a package can be determined using the PACKS operator.

At the time of this writing, several extension packages are being considered. These packages are:

Package	Possible Name
Linear Algebra	LINALG
Polynomials	POLYNOM
Programming	PROGRAM
Tensors	TENSOR
Graphics	GRAPHICS

For portability reasons, package names should be restricted to 8 characters, and should be insensitive to case. A registry for package names should be established to avoid name duplication.

Names of new operators within packages should be chosen so as not to conflict with standard names or names in other packages. Clearly, this will also require some sort of registry.