Microsoft Power Query for Excel Formula Language Specification

February 2014

1. Introduction

1.1 Overview

Microsoft Power Query for Excel (Power Query, for short) provides a powerful “get data” experience for Excel that encompasses many features. A core capability of Power Query is to filter and combine, that is, to “mash-up” data from one or more of a rich collection of supported data sources. Any such data mashup is expressed using the Power Query Formula Language (informally known as "M"). Power Query embeds M documents in Excel workbooks to enable repeatable mashup of data.

This document provides the specification for M. After a brief introduction that aims at building some first intuition and familiarity with the language, the document covers the language precisely in several progressive steps:

1. The ***lexical structure*** defines the set of texts that are lexically valid.
2. Values, expressions, environments and variables, identifiers, and the evaluation model form the language’s ***basic concepts***.
3. The detailed specification of ***values***, both primitive and structured, defines the target domain of the language.
4. Values have ***types***, themselves a special kind of value, that both characterize the fundamental kinds of values and carry additional metadata that is specific to the shapes of structured values.
5. The set of ***operators*** in M defines what kinds of expressions can be formed.
6. ***Functions***, another kind of special values, provide the foundation for a rich standard library for M and allow for the addition of new abstractions.
7. ***Errors*** can occur when applying operators or functions during expression evaluation. While errors are not values, there are ways to ***handle errors*** that map errors back to values.
8. ***Let expressions*** allow for the introduction of auxiliary definitions used to build up complex expressions in smaller steps.
9. ***If expressions*** support conditional evaluation.
10. ***Sections*** provide a simple modularity mechanism. (Sections are not yet leveraged by Power Query.)
11. Finally, a ***consolidated grammar*** collects the grammar fragments from all other sections of this document into a single complete definition.

For computer language theorists: the formula language specified in this document is a mostly pure, higher-order, dynamically typed, partially lazy functional language.

1.2 Expressions and values

The central construct in M is the ***expression***. An expression can be evaluated (computed), yielding a single ***value***.

Although many values can be written literally as an expression, a value is not an expression. For example, the expression `1` evaluates to the value *1*; the expressions `1+1` evaluates to the value *2*. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

The following examples illustrate the different kinds of values available in M. As a convention, a value is written using the literal form in which they would appear in an expression that evaluates to just that value. (Note that the `//` indicates the start of a comment which continues to the end of the line.)

- A ***primitive*** value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

```
123                 // A number
true                // A logical
"abc"               // A text
null                // null value
```

- A ***list*** value is an ordered sequence of values. M supports infinite lists, but if written as a literal, lists have a fixed length. The curly brace characters `{` and `}` denote the beginning and end of a list.

```
{123, true, "A"}    // list containing a number, a logical, and
                    //     a text
{1, 2, 3}           // list of three numbers
```

- A ***record*** is a set of ***fields***. A field is a name/value pair where the name is a text value that is unique within the field's record. The literal syntax for record values allows the names to be written without quotes, a form also referred to as ***identifiers***. The following shows a record containing three fields named "`A`", "`B`" and "`C`", which have values `1`, `2`, and `3`.

```
[
    A = 1,
    B = 2,
    C = 3
]
```

- A ***table*** is a set of values organized into columns (which are identifierd by name), and rows. There is no literal syntax for creating a table, but there are several standard functions that can be used to create tables from lists or records.

  For example:

```
#table( {"A", "B"}, { {1, 2}, {3, 4} } )
```

  This creates a table of the following shape:

A	B
1	2
3	4

- A ***function*** is a value which, when invoked with arguments, produces a new value. Function are written by listing the function's ***parameters*** in parentheses, followed by the goes-to symbol =>, followed by the expression defining the function. That expression typically refers to the parameters (by name).

```
(x, y) => (x + y) / 2
```

1.3 Evaluation

The evaluation model of the M language is modeled after the evaluation model commonly found in spreadsheets, where the order of calculation can be determined based on dependencies between the formulas in the cells.

If you have written formulas in a spreadsheet such as Excel, you may recognize the formulas on the left will result in the values on the right when calculated:

	A
1	=A2 * 2
2	=A3 + 1
3	1

	A
1	4
2	2
3	1

In M, parts of an expression can reference other parts of the expression by name, and the evaluation process will automatically determine the order in which referenced expressions are calculated.

We can use a record to produce an expression which is equivalent to the above spreadsheet example. When initializing the value of a field, we can refer to other fields within the record by using the name of the field, as follows:

```
[
    A1 = A2 * 2,
    A2 = A3 + 1,
    A3 = 1
]
```

The above expression is equivalent to the following (in that both evaluate to equal values):

```
[
    A1 = 4,
    A2 = 2,
    A3 = 1
]
```

Records can be contained within, or ***nest***, within other records. We can use the ***lookup operator*** (`[]`) to access the fields of a record by name. For example, the following record has a field named `Sales` containing a record, and a field named `Total` that accesses the `FirstHalf` and `SecondHalf` fields of the `Sales` record:

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

The above expression is equivalent to the following when it is evaluated:

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

Records can also be contained within lists. We can use the ***positional index operator*** (`{}`) to access an item in a list by its numeric index. The values within a list are referred to using a zero-based index from the beginning of the list. For example, the indexes `0` and `1` are used to reference the first and second items in the list below:

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // 2500
            ]
        },
    TotalSales = Sales{0}[Total] + Sales{1}[Total] // 4600
]
```

List and record member expressions (as well as let expressions, introduced further below) are evaluated using ***lazy evaluation***, which means that they are evaluated only as needed. All other expressions are evaluated using ***eager evaluation***, which means that they are evaluated immediately, when encountered during the evaluation process. A good way to think about this is to remember that evaluating a list or record expression will return a list or

record value that itself remembers how its list items or record fields need to computed, when requested (by lookup or index operators).

1.4 Functions

In M, a ***function*** is a mapping from a set of input values to a single output value. A function is written by first naming the required set of input values (the parameters to the function) and then providing an expression that will compute the result of the function using those input values (the body of the function) following the goes-to (`=>`) symbol. For example:

```
(x) => x + 1                    // function that adds one to a value
(x, y) =>  x + y                // function that adds two values
```

A function is a value just like a number or a text value. The following example shows a function which is the value of an `Add` field which is then ***invoked***, or executed, from several other fields. When a function is invoked, a set of values are specified which are logically substituted for the required set of input values within the function body expression.

```
[
    Add = (x, y) => x + y,
    OnePlusOne = Add(1, 1),      // 2
    OnePlusTwo = Add(1, 2)       // 3
]
```

1.5 Library

M includes a common set of definitions available for use from an expression called the ***standard library***, or just library for short. These definitions consist of a set of named values. The names of values provided by a library are available for use within an expression without having been defined explicitly by the expression. For example:

```
Number.E                        // Euler's number e (2.7182…)
Text.PositionOf("Hello", "ll")  // 2
```

1.6 Operators

M includes a set of operators that can be used in expressions. ***Operators*** are applied to ***operands*** to form symbolic expressions. For example, in the expression `1 + 2` the numbers `1` and `2` are operands and the operator is the addition operator (`+`).

The meaning of an operator can vary depending on what kind of values its operands are. For example, the plus operator can be used with other kinds of values than numbers:

```
1 + 2                   // numeric addition: 3
#time(12,23,0) + #duration(0,0,2,0)
                        // time arithmetic: #time(12,25,0)
```

Another example of an operator with operand-depending meaning is the combination operator (`&`):

```
"A" & "BC"              // text concatenation: "ABC"
```

```
{1} & {2, 3}           // list concatenation: {1, 2, 3}
[ a = 1 ] & [ b = 2 ]  // record merge: [ a = 1, b = 2 ]
```

Note that not all combinations of values may be supported by an operator. For example:

```
1 + "2"      // error: adding number and text is not supported
```

Expressions that, when evaluated, encounter undefined operator conditions evaluate to errors. More on errors in M later.

1.7 Metadata

Metadata is information about a value that is associated with a value. Metadata is represented as a record value, called a ***metadata record***. The fields of a metadata record can be used to store the metadata for a value.

Every value has a metadata record. If the value of the metadata record has not been specified, then the metadata record is empty (has no fields).

Metadata records provide a way to associate additional information with any kind of value in an unobtrusive way. Associating a metadata record with a value does not change the value or its behavior.

A metadata record value `y` is associated with an existing value `x` using the syntax `x meta y`. For example, the following associates a metadata record with `Rating` and `Tags` fields with the text value `"Mozart"`:

```
"Mozart" meta [ Rating = 5, Tags = {"Classical"} ]
```

For values that already carry a non-empty metadata record, the result of applying meta is that of computing the record merge of the existing and the new metadata record. For example, the following two expressions are equivalent to each other and to the previous expression:

```
("Mozart" meta [ Rating = 5 ]) meta [ Tags = {"Classical"} ]
"Mozart" meta ([ Rating = 5 ] & [ Tags = {"Classical"} ])
```

A metadata record can be accessed for a given value using the `Value.Metadata` function. In the following example, the expression in the `ComposerRating` field accesses the metadata record of the value in the `Composer` field, and then accesses the `Rating` field of the metadata record.

```
[
    Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],
    ComposerRating = Value.Metadata(Composer)[Rating]   // 5
]
```

1.8 Let expression

Many of the examples shown so far have included all the literal values of the expression in the result of the expression. The `let` expression allows a set of values to be computed, assigned names, and then used in a subsequent expression that follows the `in`. For example, in our sales data example, we could do:

```
let
    Sales2007 =
        [
            Year = 2007,
            FirstHalf = 1000,
            SecondHalf = 1100,
            Total = FirstHalf + SecondHalf // 2100
        ],
    Sales2008 =
        [
            Year = 2008,
            FirstHalf = 1200,
            SecondHalf = 1300,
            Total = FirstHalf + SecondHalf // 2500
        ]
  in Sales2007[Total] + Sales2008[Total] // 4600
```

The result of the above expression is a number value (`4600`) which was computed from the values bound to the names `Sales2007` and `Sales2008`.

1.9 If expression

The `if` expression selects between two expressions based on a logical condition. For example:

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

The first expression (`2 + 2`) is selected if the logical expression (`2 > 1`) is true, and the second expression (`1 + 1`) is selected if it is false. The selected expression (in this case `2 + 2`) is evaluated and becomes the result of the `if` expression (`4`).

1.10 Errors

An ***error*** is an indication that the process of evaluating an expression could not produce a value.

Errors are raised by operators and functions encountering error conditions or by using the `error` expression. Errors are handled using the `try` expression. When an error is raised, a value is specified that can be used to indicate why the error occurred.

```
let Sales =
    [
        Revenue = 2000,
        Units = 1000,
        UnitPrice = if Units = 0 then error "No Units"
                    else Revenue / Units
    ],
    UnitPrice = try Number.ToText(Sales[UnitPrice])
in "Unit Price: " &
    (if UnitPrice[HasError] then UnitPrice[Error][Message]
    else UnitPrice[Value])
```

The above example accesses the `Sales[UnitPrice]` field and formats the value producing the result:

```
"Unit Price: 2"
```

If the Units field had been zero, then the `UnitPrice` field would have raised an error which would have been handled by the `try`. The resulting value would then have been:

```
"No Units"
```

A `try` expression converts proper values and errors into a record value that indicates whether the `try` expression handled and error, or not, and either the proper value or the error record it extracted when handling the error. For example, consider the following expression that raises an error and then handles it right away:

```
try error "negative unit count"
```

This expression evaluates to the following nested record value, explaining the `[HasError]`, `[Error]`, and `[Message]` field lookups in the unit-price example before.

```
[
    HasError = true,
    Error =
        [
            Reason = "Expression.Error",
            Message = "negative unit count",
            Detail = null
        ]
]
```

A common case is to replace errors with default values. The `try` expression can be used with an optional `otherwise` clause to achieve just that in a compact form:

```
try error "negative unit count" otherwise 42
// 42
```

2. Lexical Structure

2.1 Documents

An M ***document*** is an ordered sequence of Unicode characters. M allows different classes of Unicode characters in different parts of an M document. For information on Unicode character classes, see *The Unicode Standard, Version 3.0*, section 4.5.

A document either consists of exactly one ***expression*** or of groups of ***definitions*** organized into ***sections***. Sections are described in detail in Chapter 10. Conceptually speaking, the following steps are used to read an expression from a document:

1. The document is decoded according to its character encoding scheme into a sequence of Unicode characters.
2. Lexical analysis is performed, thereby translating the stream of Unicode characters into a stream of ***tokens***. The remaining subsections of this section cover lexical analysis.
3. Syntactic analysis is performed, thereby translating the stream of tokens into a form that can be evaluated. This process is covered in subsequent sections.

2.2 Grammar conventions

The lexical and syntactic grammars are presented using ***grammar productions***. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, *non-terminal* symbols are shown in italic type, and `terminal` symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

if-expression:
    `if` *if-condition* `then` *true-expression* `else` *false-expression*

defines an *if-expression* to consist of the token `if`, followed by an *if-condition*, followed by the token `then`, followed by a *true-expression*, followed by the token `else`, followed by a *false-expression*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

variable-list:
    variable
    variable-list **,** *variable*

defines a *variable-list* to either consist of a *variable* or consist of a *variable-list* followed by a *variable*. In other words, the definition is recursive and specifies that a variable list consists of one or more variables, separated by commas.

A subscripted suffix "*opt*" is used to indicate an optional symbol. The production:

field-specification:
    `optional`*opt* *identifier* `=` *field-type*

is shorthand for:

field-specification:
    identifier `=` *field-type*
    `optional` *identifier* `=` *field-type*

and defines a *field-specification* to optionally begin with the terminal symbol `optional` followed by an *identifier*, the terminal symbol `=`, and a *field-type*.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

decimal-digit: one of
    `0 1 2 3 4 5 6 7 8 9`

is shorthand for:

decimal-digit:
    `0`
    `1`
    `2`
    `3`
    `4`
    `5`
    `6`
    `7`
    `8`
    `9`

2.3 Lexical Analysis

The *lexical-unit* production defines the lexical grammar for an M document. Every valid M document conforms to this grammar.

lexical-unit:
    *lexical-elements*opt

lexical-elements:
    lexical-element
    lexical-element *lexical-elements*

lexical-element:
    whitespace
    token
    comment

At the lexical level, an M document consists of a stream of *whitespace*, *comment*, and *token* elements. Each of these productions is covered in the following sections. Only *token* elements are significant in the syntactic grammar.

2.4 Whitespace

Whitespace is used to separate comments and tokens within an M document. Whitespace includes the space character (which is part of Unicode class Zs), as well as horizontal and vertical tab, form feed, and newline character sequences. Newline character sequences include carriage return, line feed, carriage return followed by line feed, next line, and paragraph separator characters.

whitespace:
Any character with Unicode class Zs
Horizontal tab character (`U+0009`)
Vertical tab character (`U+000B`)
Form feed character (`U+000C`)
Carriage return character (`U+000D`) followed by line feed character (`U+000A`)
new-line-character

new-line-character:
Carriage return character (`U+000D`)
Line feed character (`U+000A`)
Next line character (`U+0085`)
Line separator character (`U+2028`)
Paragraph separator character (`U+2029`)

For compatibility with source code editing tools that add end-of-file markers, and to enable a document to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to an M document:

- If the last character of the document is a Control-Z character (`U+001A`), this character is deleted.
- A carriage-return character (`U+000D`) is added to the end of the document if that document is non-empty and if the last character of the document is not a carriage return (`U+000D`), a line feed (`U+000A`), a line separator (`U+2028`), or a paragraph separator (`U+2029`).

2.5 Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines.

comment:
single-line-comment
delimited-comment

single-line-comment:

`//` *single-line-comment-characters*$_{opt}$

single-line-comment-characters:

single-line-comment-character

single-line-comment-characters single-line-comment-character

single-line-comment-character:

Any Unicode character except a *new-line-character*

delimited-comment:

`/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`

delimited-comment-text:

delimited-comment-section

delimited-comment-text delimited-comment-section

delimited-comment-section:

`/`

asterisks$_{opt}$ *not-slash-or-asterisk*

asterisks:

`*`

`*` *asterisks*

not-slash-or-asterisk:

Any Unicode character except `*` or `/`

Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a single-line comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.

Comments are not processed within text literals.

The example

```
/* Hello, world
*/
"Hello, world"
```

includes a delimited comment.

The example

```
// Hello, world
//
"Hello, world" // This is an example of a text literal
```

shows several single-line comments.

2.6 Tokens

A ***token*** is an identifier, keyword, literal, operator, or punctuator. Whitespace and comments are used to separate tokens, but are not considered tokens.

token:
    identifier
    keyword
    literal
    operator-or-punctuator

2.6.1 Character Escape Sequences

M text values can contain arbitrary Unicode characters. Text literals, however, are limited to graphic characters and require the use of ***escape sequences*** for non-graphic characters. For example, to include a carriage-return, linefeed, or tab character in a text literal, the `#(cr)`, `#(lf)`, and `#(tab)` escape sequences can be used, respectively. To embed the escape-sequence start characters `#(` in a text literal, the `#` itself needs to be escaped:

```
#(#)(
```

Escape sequences can also contain short (four hex digits) or long (eight hex digits) Unicode code-point values. The following three escape sequences are therefore equivalent:

```
#(000D)      // short Unicode hexadecimal value
#(0000000D)  // long Unicode hexadecimal value
#(cr)        // compact escape shorthand for carriage return
```

Multiple escape codes can be included in a single escape sequence, separated by commas; the following two sequences are thus equivalent:

```
#(cr,lf)
#(cr)#(lf)
```

The following describes the standard mechanism of character escaping in an M document.

character-escape-sequence:
    `#(` *escape-sequence-list* `)`

escape-sequence-list:
    single-escape-sequence
    single-escape-sequence `,` *escape-sequence-list*

single-escape-sequence:
    long-unicode-escape-sequence
    short-unicode-escape-sequence
    control-character-escape-sequence
    escape-escape

long-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

short-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit

control-character-escape-sequence:
    control-character

control-character:

`cr`

`lf`

`tab`

escape-escape:

`#`

2.6.2 Literals

A ***literal*** is a source code representation of a value.

literal:

logical-literal
number-literal
text-literal
null-literal

2.6.2.1 Null literals

The null literal is used to write the `null` value. The `null` value represents an absent value.

null-literal:

`null`

2.6.2.2 Logical literals

A logical literal is used to write the values `true` and `false` and produces a logical value.

logical-literal:

`true`

`false`

2.6.2.3 Number literals

A number literal is used to write a numeric value and produces a number value.

number-literal:

decimal-number-literal
hexadecimal-number-literal

decimal-number-literal:

decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$
`.` *decimal-digits* *exponent-part*$_{opt}$
decimal-digits *exponent-part*
decimal-digits

decimal-digits:

decimal-digit
decimal-digit *decimal-digits*

decimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`

exponent-part:
`e` *sign*$_{opt}$ *decimal-digits*
`E` *sign*$_{opt}$ *decimal-digits*

sign: one of
`+ -`

hexadecimal-number-literal:
`0x` *hex-digits*
`0X` *hex-digits*

hex-digits:
hex-digit
hex-digit hex-digits

hex-digit: one of
`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

A number can be specified in hexadecimal format by preceding the *hex-digits* with the characters `0x`. For example:

```
0xff // 255
```

Note that if a decimal point is included in a number literal, then it must have at least one digit following it. For example, `1.3` is a number literal but `1.` and `1.e3` are not.

2.6.2.4 Text literals

A text literal is used to write a sequence of Unicode characters and produces a text value.

text-literal:
`"` *text-literal-characters*$_{opt}$ `"`

text-literal-characters:
text-literal-character
text-literal-character text-literal-characters

text-literal-character:
single-text-character
character-escape-sequence
double-quote-escape-sequence

single-text-character:
Any character except `"` (`U+0022`) or `#` (`U+0023`) followed by `(` (`U+0028`)

double-quote-escape-sequence:
`""` (`U+0022`, `U+0022`)

To include quotes in a text value, the quote mark is repeated, as follows:

```
"The ""quoted"" text" // The "quoted" text
```

The *character-escape-sequence* (§2.6.1) production can be used to write characters in text values without having to directly encode them as Unicode characters in the document. For example, a carriage return and line feed can be written in a text value as:

```
"Hello world#(cr,lf)"
```

2.6.3 Identifiers

An ***identifier*** is a name used to refer to a value. Identifiers can either be regular identifiers or quoted identifiers.

identifier:
    regular-identifier
    quoted-identifier

regular-identifier:
    available-identifier
    available-identifier dot-character regular-identifier

available-identifier:
    A *keyword-or-identifier* that is not a *keyword*

keyword-or-identifier:
    identifier-start-character identifier-part-characters$_{opt}$

identifier-start-character:
    letter-character
    underscore-character

identifier-part-characters:
    identifier-part-character
    identifier-part-character identifier-part-characters

identifier-part-character:
    letter-character
    decimal-digit-character
    underscore-character
    connecting-character
    combining-character
    formatting-character

dot-character:
    `.` (`U+002E`)

underscore-character:
    `_` (`U+005F`)

letter-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:
    A Unicode character of classes Mn or Mc

decimal-digit-character:
    A Unicode character of the class Nd

connecting-character:
    A Unicode character of the class Pc

formatting-character:

A Unicode character of the class Cf

A *quoted-identifier* can be used to allow any sequence of zero or more Unicode characters to be used as an identifier, including keywords, whitespace, comments, operators and punctuators.

quoted-identifier:

`#"` *text-literal-characters*$_{opt}$ `"`

Note that escape sequences and double-quotes to escape quotes can be used in a *quoted identifier*, just as in a *text-literal*.

The following example uses identifier quoting for names containing a space character:

```
[
  #"1998 Sales" = 1000,
  #"1999 Sales" = 1100,
  #"Total Sales" = #"1998 Sales" + #"1999 Sales"
]
```

The following example uses identifier quoting to include the + operator in an identifier:

```
[
    #"A + B" = A + B,
    A = 1,
    B = 2
]
```

2.6.3.1 Generalized Identifiers

There are two places in M where no ambiguities are introduced by identifiers that contain blanks or that are otherwise keywords or number literals. These places are the names of record fields in a record literal and in a field access operator (`[ ]`) There, M allows such identifiers without having to use quoted identifiers.

```
[
    Data = [ Base Line = 100, Rate = 1.8 ],
    Progression = Data[Base Line] * Data[Rate]
]
```

The identifiers used to name and access fields are referred to as ***generalized identifiers*** and defined as follows:

generalized-identifier:

generalized-identifier-part

generalized-identifier separated only by blanks (`U+0020`) *generalized-identifier-part*

generalized-identifier-part:

generalized-identifier-segment

decimal-digit-character *generalized-identifier-segment*

generalized-identifier-segment:
keyword-or-identifier
keyword-or-identifier dot-character keyword-or-identifier

2.6.4 Keywords

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when using the identifier-quoting mechanism (§2.6.3) or where a generalized identifier is allowed (§2.6.3.1).

keyword: one of

```
and as each else error false if in is let meta not otherwise or
section shared then true try type #binary #date #datetime
#datetimezone #duration #infinity #nan #sections #shared #table #time
```

2.6.5 Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

operator-or-punctuator: one of

```
, ; = < <= > >= <> + - * / & ( ) [ ] { } @ ! ? => .. ...
```

3. Basic Concepts

This section discusses basic concepts that appear throughout the subsequent sections.

3.1 Values

A single piece of data is called a ***value***. Broadly speaking, there are two general categories of values: ***primitive values***, which are atomic, and ***structured values***, which are constructed out of primitive values and other structured values. For example, the values

```
1
true
3.14159
"abc"
```

are primitive in that they are not made up of other values. On the other hand, the values

```
{1, 2, 3}
[ A = {1}, B = {2}, C = {3} ]
```

are constructed using primitive values and, in the case of the record, other structured values.

3.2 Expressions

An ***expression*** is a formula used to construct values. An expression can be formed using a variety of syntactic constructs. The following are some examples of expressions. Each line is a separate expression.

```
"Hello World"            // a text value
123                      // a number
1 + 2                    // sum of two numbers
{1, 2, 3}                // a list of three numbers
[ x = 1, y = 2 + 3 ]     // a record containing two fields:
                         //        x and y
(x, y) => x + y          // a function that computes a sum
if 2 > 1 then 2 else 1   // a conditional expression
let x = 1 + 1  in x * 2  // a let expression
error "A"                // error with message "A"
```

The simplest form of expression, as seen above, is a literal representing a value.

More complex expressions are built from other expressions, called ***sub-expressions***. For example:

```
1 + 2
```

The above expression is actually composed of three expressions. The `1` and `2` literals are sub-expressions of the parent expression `1 + 2`.

Executing the algorithm defined by the syntactic constructs used in an expression is called ***evaluating*** the expression. Each kind of expression has rules for how it is evaluated. For example, a literal expression like `1` will produce a constant value, while the expression `a + b` will take the resulting values produced by evaluating two other expressions (`a` and `b`) and add them together according to some set of rules.

3.3 Environments and variables

Expressions are evaluated within a given environment. An ***environment*** is a set of named values, called ***variables***. Each variable in an environment has a unique name within the environment called an ***identifier***.

A top-level (or ***root***) expression is evaluated within the ***global environment***. The global environment is provided by the expression evaluator instead of being determined from the contents of the expression being evaluated. The contents of the global environment includes the standard library definitions and can be affected by exports from sections from some set of documents. (For simplicity, the examples in this section will assume an empty global environment. That is, it is assumed that there is no standard library and that there are no other section-based definitions.)

The environment used to evaluate a sub-expression is determined by the parent expression. Most parent expression kinds will evaluate a sub-expression within the same environment they were evaluated within, but some will use a different environment. The global environment is the *parent environment* within which the global expression is evaluated.

For example, the *record-initializer-expression* evaluates the sub-expression for each field with a modified environment. The modified environment includes a variable for each of the fields of the record, except the one being initialized. Including the other fields of the record allows the fields to depend upon the values of the fields. For example:

```
[
    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
]
```

Similarly, the *let-expression* evaluates the sub-expression for each variable with an environment containing each of the variables of the let except the one being initialized. The *let-expression* evaluates the expression following the `in` with an environment containing all the variables:

```
let
    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
in
    x + y + z       // environment: x, y, z
```

(It turns out that both *record-initializer-expression* and *let-expression* actually define *two* environments, one of which does include the variable being initialized. This is useful for advanced recursive definitions and covered in §3.3.1.)

To form the environments for the sub-expressions, the new variables are "merged" with the variables in the parent environment. The following example shows the environments for nested records:

```
[
    a =
    [
        x = 1,      // environment: b, y, z
        y = 2,      // environment: b, x, z
        z = x + y   // environment: b, x, y
    ],
    b = 3           // environment: a
]
```

The following example shows the environments for a record nested within a let:

```
Let
    a =
    [
        x = 1,      // environment: b, y, z
        y = 2,      // environment: b, x, z
        z = x + y   // environment: b, x, y
    ],
    b = 3           // environment: a
in
    a[z] + b        // environment: a, b
```

Merging variables with an environment may introduce a conflict between variables (since each variable in an environment must have a unique name). The conflict is resolved as follows: if the name of a new variable being merged is the same as an existing variable in the parent environment, then the new variable will take precedence in the new environment. In the following example, the inner (more deeply nested) variable `x` will take precedence over the outer variable `x`.

```
[
    a =
    [
        x = 1,      // environment: b, x (outer), y, z
        y = 2,      // environment: b, x (inner), z
        z = x + y   // environment: b, x (inner), y
    ],
    b = 3,          // environment: a, x (outer)
    x = 4           // environment: a, b
]
```

3.3.1 Identifier references

An *identifier-reference* is used to refer to a variable within an environment.

identifier-expression:
    identifier-reference

identifier-reference:
    exclusive-identifier-reference
    inclusive-identifier-reference

The simplest form of identifier reference is an *exclusive-identifier-reference*:

exclusive-identifier-reference:
    identifier

It is an error for an *exclusive-identifier-reference* to refer to a variable that is not part of the environment of the expression that the identifier appears within, or to refer to an identifier that is currently being initialized.

An *inclusive-identifier-reference* can be used to gain access to the environment that includes the identifier being initialized. If it used in a context where there is no identifier being initialized, then it is equivalent to an *exclusive-identifier-reference*.

inclusive-identifier-reference:
    `@` *identifier*

This is useful when defining recursive functions since the name of the function would normally not be in scope.

```
[
    Factorial = (n) =>
        if n <= 1 then
            1
        else
            n * @Factorial(n - 1),  // @ is scoping operator

    x = Factorial(5)
]
```

As with a *record-initializer-expression*, an *inclusive-identifier-reference* can be used within a *let-expression* to access the environment that includes the identifier being initialized.

3.4 Order of evaluation

Consider the following expression which initializes a record:

```
[
    C = A + B,
    A = 1 + 1,
    B = 2 + 2
]
```

When evaluated, this expression produces the following record value:

```
[
    C = 6,
    A = 2,
    B = 4
]
```

The expression states that in order to perform the `A + B` calculation for field `C`, the values of both field `A` and field `B` must be known. This is an example of a ***dependency ordering*** of calculations that is provided by an expression. The M evaluator abides by the dependency ordering provided by expressions, but is free to perform the remaining calculations in any order it chooses. For example, the computation order could be:

```
A = 1 + 1
B = 2 + 2
C = A + B
```

Or it could be:

```
B = 2 + 2
A = 1 + 1
C = A + B
```

Or, since `A` and `B` do not depend on each other, they can be computed concurrently:

```
B = 2 + 2    concurrently with    A = 1 + 1
C = A + B
```

3.5 Side effects

Allowing an expression evaluator to automatically compute the order of calculations for cases where there are no explicit dependencies stated by the expression is a simple and powerful computation model.

It does, however, rely on being able to reorder computations. Since expressions can call functions, and those functions could observe state external to the expression by issuing external queries, it is possible to construct a scenario where the order of calculation does

matter, but is not captured in the partial order of the expression. For example, a function may read the contents of a file. If that function is called repeatedly, then external changes to that file can be observed and, therefore, reordering can cause observable differences in program behavior. Depending on such observed evaluation ordering for the correctness of an M expression causes a dependency on particular implementation choices that might vary from one evaluator to the next or may even vary on the same evaluator under varying circumstances.

3.6 Immutability

Once a value has been calculated, it is ***immutable***, meaning it can no longer be changed. This simplifies the model for evaluating an expression and makes it easier to reason about the result since it is not possible to change a value once it has been used to evaluate a subsequent part of the expression. For instance, a record field is only computed when needed. However, once computed, it remains fixed for the lifetime of the record. Even if the attempt to compute the field raised an error, that same error will be raised again on every attempt to access that record field.

An important exception to the immutable-once-calculated rule applies to list and table values. Both have ***streaming semantics***. That is, repeated enumeration of the items in a list or the rows in a table can produce varying results. Streaming semantics enables the construction of M expressions that transform data sets that would not fit in memory at once.

Also, note that function application is *not* the same as value construction. Library functions may expose external state (such as the current time or the results of a query against a database that evolves over time), rendering them ***non-deterministic***. While functions defined in M will not, as such, expose any such non-deterministic behavior, they can if they are defined to invoke other functions that are non-deterministic.

A final source of non-determinsm in M are ***errors***. Errors stop evaluations when they occur (up to the level where they are handled by a try expression). It is not normally observable whether `a + b` caused the evaluation of `a` before `b` or `b` before `a` (ignoring concurrency here for simplicity). However, if the subexpression that was evaluated first raises an error, then it can be determined which of the two expressions was evaluated first.

4. Values

A value is data produced by evaluating an expression. This section describes the kinds of values in the M language. Each kind of value is associated with a literal syntax, a set of values that are of that kind, a set of operators defined over that set of values, and an intrinsic type ascribed to newly constructed values.

Kind	Literal
Null	`null`
Logical	`true  false`
Number	`0  1  -1  1.5  2.3e-5`
Time	`#time(09,15,00)`
Date	`#date(2013,02,26)`
DateTime	`#datetime(2013,02,26, 09,15,00)`
DateTimeZone	`#datetimezone(2013,02,26, 09,15,00, 09,00)`
Duration	`#duration(0,1,30,0)`
Text	`"hello"`
Binary	`#binary("AQID")`
List	`{1, 2, 3}`
Record	`[ A = 1, B = 2 ]`
Table	`#table({"X","Y"},{{0,1},{1,0}})`
Function	`(x) => x + 1`
Type	`type { number }  type table [ A = any, B = text ]`

The following sections cover each value kind in detail. Types and type ascription are defined formally in Chapter 5. Function values are defined in Chapter 9. The following sections list the operators defined for each value kind and give examples. The full definition of operator semantics follows in Chapter 6.

4.1 Null

A ***null value*** is used to represent the absence of a value, or a value of indeterminate or unknown state. A null value is written using the literal `null`. The following operators are defined for null values:

Operator	Result
`x > y`	Greater than
`x >= y`	Greater than or equal
`x < y`	Less than
`x <= y`	Less than or equal
`x = y`	Equal
`x <> y`	Not equal

The native type of the `null` value is the intrinsic type `null`.

4.2 Logical

A ***logical value*** is used for Boolean operations has the value true or false. A logical value is written using the literals `true` and `false`. The following operators are defined for logical values:

Operator	Result
`x > y`	Greater than
`x >= y`	Greater than or equal
`x < y`	Less than
`x <= y`	Less than or equal
`x = y`	Equal
`x <> y`	Not equal
`x or y`	Conditional logical OR
`x and y`	Conditional logical AND
`not x`	Logical NOT

The native type of both logical values (`true` and `false`) is the intrinsic type `logical`.

4.3 Number

A ***number value*** is used for numeric and arithmetic operations. The following are examples of number literals:

```
3.14  // Fractional number
-1.5  // Fractional number
1.0e3 // Fractional number with exponent
123   // Whole number
1e3   // Whole number with exponent
0xff  // Whole number in hex (255)
```

A number is represented with at least the precision of a ***Double*** (but may retain more precision). The ***Double*** representation is congruent with the IEEE 64-bit double precision standard for binary floating point arithmetic defined in [IEEE 754-2008]. (The ***Double*** representation have an approximate dynamic range from 5.0×10^{-324} to 1.7×10^{308} with a precision of 15-16 digits.)

The following special values are also considered to be ***number*** values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (§6.9).
- Positive infinity (`#infinity`) and negative infinity (`-#infinity`). Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.
- The ***Not-a-Number*** value (`#nan`), often abbreviated NaN. NaN's are produced by invalid floating-point operations, such as dividing zero by zero.

Binary mathematical operations are performed using a ***Precision***. The precision determines the domain to which the operands are rounded and the domain in which the operation is performed. In the absence of an explicitly specified precision, such operations are performed using ***Double Precision***.

- If the result of a mathematical operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a mathematical operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a mathematical operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

The following operators are defined for ***number*** values:

Operator	Result
`x > y`	Greater than
`x >= y`	Greater than or equal
`x < y`	Less than
`x <= y`	Less than or equal
`x = y`	Equal
`x <> y`	Not equal
`x + y`	Sum
`x - y`	Difference
`x * y`	Product
`x / y`	Quotient
`+x`	Unary plus
`-x`	Negation

The native type of number values is the intrinsic type `number`.

4.4 Time

A ***time value*** stores an opaque representation of time of day. A time is encoded as the number of *ticks since midnight*, which counts the number of 100-nanosecond ticks that have elapsed on a 24-hour clock. The maximum number of *ticks since midnight* corresponds to 23:59:59.9999999 hours.

Time values may be constructed using the #time instrinsic.

```
#time(hour, minute, second)
```

The following must hold or an error with reason code `Expression.Error` is raised:

0 ≤ hour ≤ 24

0 ≤ minute ≤ 59

0 ≤ second ≤ 59

In addition, if hour = 24, then minute and second must be zero:

```
#time(24,0,0) = #time(0,0,0)
```

The following operators are defined for time values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a date**:**

Operator	Left Operand	Right Operand	Meaning
`x + y`	`time`	`duration`	Date offset by duration
`x + y`	`duration`	`time`	Date offset by duration
`x - y`	`time`	`duration`	Date offset by negated duration
`x - y`	`time`	`time`	Duration between dates
`x & y`	`date`	`time`	Merged datetime

The native type of time values is the intrinsic type `time`.

4.5 Date

A ***date value*** stores an opaque representation of a specific day. A date is encoded as a number of *days since epoch*, starting from January 1, 0001 Common Era on the Gregorian calendar. The maximum number of days since epoch is 3652058, corresponding to December 31, 9999.

Date values may be constructed using the `#date` intrinsic.

```
#date(year, month, day)
```

The following must hold or an error with reason code `Expression.Error` is raised:

1 ≤ year ≤ 9999

1 ≤ month ≤ 12

1 ≤ day ≤ 31

In addition, the day must be valid for the chosen month and year.

The following operators are defined for date values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a date**:**

Operator	Left Operand	Right Operand	Meaning
`x + y`	`date`	`duration`	Date offset by duration
`x + y`	`duration`	`date`	Date offset by duration
`x - y`	`date`	`duration`	Date offset by negated duration
`x - y`	`date`	`date`	Duration between dates
`x & y`	`date`	`time`	Merged datetime

The native type of date values is the intrinsic type `date`.

4.6 DateTime

A ***datetime value*** contains both a date and time**.**

DateTime values may be constructed using the `#datetime` intrinsic.

```
#datetime(year, month, day, hour, minute, second)
```

The following must hold or an error with reason code `Expression.Error` is raised:

1 ≤ year ≤ 9999

1 ≤ month ≤ 12

1 ≤ day ≤ 31

0 ≤ hour ≤ 23

0 ≤ minute ≤ 59

0 ≤ second ≤ 59

In addition, the day must be valid for the chosen month and year.

The following operators are defined for datetime values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a datetime**:**

Operator	Left Operand	Right Operand	Meaning
`x + y`	`datetime`	`duration`	Datetime offset by duration
`x + y`	`duration`	`datetime`	Datetime offset by duration
`x - y`	`datetime`	`duration`	Datetime offset by negated duration
`x - y`	`datetime`	`datetime`	Duration between datetimes

The native type of datetime values is the intrinsic type `datetime`.

4.7 DateTimeZone

A ***datetimezone value*** contains a datetime and a timezone. A ***timezone*** is encoded as a number of *minutes offset from UTC*, which counts the number of minutes the time portion of the ***datetime*** should be offset from Universal Coordinated Time (UTC). The minimum number of *minutes offset from UTC* is -840, representing a UTC offset of -14:00, or fourteen hours earlier than UTC. The maximum number of *minutes offset from UTC* is 840, corresponding to a UTC offset of 14:00.

DateTimeZone values may be constructed using the `#datetimezone` intrinsic.

```
#datetimezone(
    year, month, day,
    hour, minute, second,
    offset-hours, offset-minutes)
```

The following must hold or an error with reason code `Expression.Error` is raised:

1 ≤ year ≤ 9999

1 ≤ month ≤ 12

1 ≤ day ≤ 31

0 ≤ hour ≤ 23

0 ≤ minute ≤ 59

0 ≤ second ≤ 59

-14 ≤ offset-hours ≤ 14

-59 ≤ offset-minutes ≤ 59

In addition, the day must be valid for the chosen month and year and, if offset-hours = 14, then offset-minutes <= 0 and, if offset-hours = -14, then offset-minutes >= 0.

The following operators are defined for datetimezone values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a datetimezone**:**

Operator	Left Operand	Right Operand	Meaning
`x + y`	`datetimezone`	`duration`	Datetimezone offset by duration
`x + y`	`duration`	`datetimezone`	Datetimezone offset by duration
`x - y`	`datetimezone`	`duration`	Datetimezone offset by negated duration
`x - y`	`datetimezone`	`datetimezone`	Duration between datetimezones

The native type of datetimezone values is the intrinsic type `datetimezone`.

4.8 Duration

A ***duration value*** stores an opaque representation of the distance between two points on a timeline measured 100-nanosecond ticks. The magnitude of a ***duration*** can be either positive or negative, with positive values denoting progress forwards in time and negative values denoting progress backwards in time. The minimum value that can be stored in a ***duration*** is -9,223,372,036,854,775,808 ticks, or 10,675,199 days 2 hours 48 minutes 05.4775808 seconds backwards in time. The maximum value that can be stored in a ***duration*** is 9,223,372,036,854,775,807 ticks, or 10,675,199 days 2 hours 48 minutes 05.4775807 seconds forwards in time.

Duration values may be constructed using the `#duration` intrinsic function:

```
#duration(0, 0, 0, 5.5)          // 5.5 seconds
#duration(0, 0, 0, -5.5)         // -5.5 seconds
#duration(0, 0, 5, 30)           // 5.5 minutes
#duration(0, 0, 5, -30)          // 4.5 minutes
#duration(0, 24, 0, 0)           // 1 day
#duration(1, 0, 0, 0)            // 1 day
```

The following operators are defined on duration values:

Operator	**Result**
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

Additionally, the following operators allow one or both of their operands to be a duration value**:**

Operator	**Left Operand**	**Right Operand**	**Meaning**
`x + y`	`datetime`	`duration`	Datetime offset by duration
`x + y`	`duration`	`datetime`	Datetime offset by duration
`x + y`	`duration`	`duration`	Sum of durations
`x - y`	`datetime`	`duration`	Datetime offset by negated duration
`x - y`	`datetime`	`datetime`	Duration between datetimes
`x - y`	`duration`	`duration`	Difference of durations
`x * y`	`duration`	`number`	N times a duration
`x * y`	`number`	`duration`	N times a duration
`x / y`	`duration`	`number`	Fraction of a duration

The native type of duration values is the intrinsic type `duration`.

4.9 Text

A ***text value*** represents a sequence of Unicode characters. Text values have a literal form conformant to the following grammar:

text-literal:
" *text-literal-characters*$_{opt}$ "

text-literal-characters:
text-literal-character
text-literal-characters text-literal-character

text-literal-character:
single-text-character
character-escape-sequence
double-quote-escape-sequence

single-text-character:
Any character except " (U+0022) or # (U+0023) followed by ((U+0028)

double-quote-escape-sequence:
"" (U+0022, U+0022)

The following is an example of a ***text*** value:

```
"ABC" // the text value ABC
```

The following operators are defined on ***text*** values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal
`x & y`	Concatenation

The native type of text values is the intrinsic type `text`.

4.10 Binary

A ***binary value*** represents a sequence of bytes. There is no literal format. Several standard library functions are provided to construct binary values. For example, `#binary` can be used to construct a binary value from a list of bytes:

```
#binary( {0x00, 0x01, 0x02, 0x03} )
```

The following operators are defined on ***binary*** values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The native type of binary values is the intrinsic type `binary`.

4.11 List

A ***list value*** is a value which produces a sequence of values when enumerated. A value produced by a list can contain any kind of value, including a list. Lists can be constructed using the initialization syntax, as follows:

list-expression:
`{` *item-list*$_{opt}$ `}`
item-list:
item
item `,` *item-list*
item:
expression
expression `..` *expression*

The following is an example of a *list-expression* that defines a list with three text values: `"A"`, `"B"`, and `"C"`.

```
{"A", "B", "C"}
```

The value `"A"` is the first item in the list, and the value `"C"` is the last item in the list.

- The items of a list are not evaluated until they are accessed.
- While list values constructed using the list syntax will produce items in the order they appear in *item-list,* in general, lists returned from library functions may produce a different set or a different number of values each time they are enumerated.

To include a sequence of whole number in a list, the `a..b` form can be used:

```
{ 1, 5..9, 11 }              // { 1, 5, 6, 7, 8, 9, 11 }
```

The number of items in a list, known as the ***list count***, can be determined using the `List.Count` function.

```
List.Count({true, false})    // 2
List.Count({})               // 0
```

A list may effectively have an infinite number of items; `List.Count` for such lists is undefined and may either raise an error or not terminate.

If a list contains no items, it is called an ***empty list***. An empty list is written as:

```
{}    // empty list
```

The following operators are defined for lists:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x & y`	Concatenate

For example:

```
{1, 2} & {3, 4, 5}        // {1, 2, 3, 4, 5}
{1, 2} = {1, 2}           // true
{2, 1} <> {1, 2}          // true
```

The native type of list values is the intrinsic type `list`, which specifies an item type of `any`.

4.12 Record

A ***record value*** is an ordered sequence of fields. A ***field*** consists of a ***field name***, which is a text value that uniquely identifies the field within the record, and a ***field value***. The field value can be any kind of value, including record. Records can be constructed using initialization syntax, as follows:

record-expression:
    `[` *field-list*$_{opt}$ `]`

field-list:
    field
    field `,` *field-list*

field:
    field-name `=` *expression*

field-name:
    generalized-identifier

The following example constructs a record with a field named `x` with value `1`, and a field named `y` with value `2`.

```
[ x = 1, y = 2 ]
```

The following example constructs a record with a field named `a` with a nested record value. The nested record has a field named `b` with value `2`.

```
[ a = [ b = 2 ] ]
```

The following holds when evaluating a record expression:

- The expression assigned to each field name is used to determine the value of the associated field.
- If the expression assigned to a field name produces a value when evaluated, then that becomes the value of the field of the resulting record.
- If the expression assigned to a field name raises an error when evaluated, then the fact that an error was raised is recorded with the field along with the error value that was raised. Subsequent access to that field will cause an error to be re-raised with the recorded error value.
- The expression is evaluated in an environment like the parent environment only with variables merged in that correspond to the value of every field of the record, except the one being initialized.
- A value in a record is not evaluated until the corresponding field is accessed.
- A value in a record is evaluated at most once.
- The result of the expression is a record value with an empty metadata record.
- The order of the fields within the record is defined by the order that they appear in the *record-initializer-expression*.
- Every field name that is specified must be unique within the record, or it is an error. Names are compared using an ordinal comparison.

```
[ x = 1, x = 2 ] // error: field names must be unique
[ X = 1, x = 2 ] // OK
```

A record with no fields is called an ***empty record***, and is written as follows:

```
[] // empty record
```

Although the order of the fields of a record is not significant when accessing a field or comparing two records, it is significant in other contexts such as when the fields of a record are enumerated.

The same two records produce different results when the fields are obtained:

```
Record.FieldNames([ x = 1, y = 2 ]) // [ "x", "y" ]
Record.FieldNames([ y = 1, x = 2 ]) // [ "y", "x" ]
```

The number of fields in a record can be determined using the `Record.FieldCount` function. For example:

```
Record.FieldCount([ x = 1, y = 2 })  // 2
Record.FieldCount([])                // 0
```

In addition to using the record initialization syntax `[ ]`, records can be constructed from a list of records describing each field. For example:

```
Record.FromList({
   [ Name = "a", Value = 1 ],
   [ Name = "b", Value = 2 ]})
```

The above is equivalent to:

```
[ a = 1, b = 2 ]
```

The following operators are defined for record values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`x & y`	Merge

The following examples illustrate the above operators. Note that record merge uses the fields from the right operand to override fields from the left operand, should there be an overlap in field names.

```
[ a = 1, b = 2 ] & [ c = 3 ]     // [ a = 1, b = 2, c = 3 ]
[ a = 1, b = 2 ] & [ a = 3 ]     // [ a = 3, b = 2 ]
[ a = 1, b = 2 ] = [ b = 2, a = 1 ]           // true
[ a = 1, b = 2, c = 3 ] <> [ a = 1, b = 2 ]   // true
```

The native type of record values is the intrinsic type `record`, which specifies an open empty list of fields.

4.13 Table

A ***table value*** is an ordered sequence of rows. A ***row*** is an ordered sequence of value. The table's type determines the length of all rows in the table, the names of the table's columns, the types of the table's columns, and the structure of the table's keys (if any).

There is no literal syntax for tables. Several standard library functions are provided to construct binary values. For example, `#table` can be used to construct a table from a list of row lists and a list of header names:

```
#table({"x", "x^2"}, {{1,1}, {2,4}, {3,9}})
```

The above example constructs a table with two columns, both of which are of `type any`. `#table` can also be used to specify a full table type:

```
#table(
    type table [Digit = number, Name = text],
    {{1,"one"}, {2,"two"}, {3,"three"}}
    )
```

Here the new table value has a table type that specifies column names and column types.

The following operators are defined for table values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal
`X & y`	Concatenation

Table concatenation aligns like-named columns and fills in `null` for columns appearing in only one of the operand tables. The following example illustrates table concatenation:

```
  #table({"A","B"}, {{1,2}})
& #table({"B","C"}, {{3,4}})
```

A	B	C
1	2	null
null	3	4

The native type of table values is a custom table type (derived from the intrinsic type `table`) that lists the column names, specifies all column types to be `any`, and has no keys. (See §5.6 for details on table types.)

4.14 Function

A ***function value*** is a value that maps a set of arguments to a single value. The details of ***function*** values are described in Chapter 9.

4.15 Type

A ***type value*** is a value that classifies other values. The details of ***type*** values are described in Chapter 5.

5. Types

A ***type value*** is a value that ***classifies*** other values. A value that is classified by a type is said to ***conform*** to that type. The M type system consists of the following kinds of types:

- Primitive types, which classify primitive values (`binary`, `date`, `datetime`, `datetimezone`, `duration`, `list`, `logical`, `null`, `number`, `record`, `text`, `time`, `type`) and also include a number of abstract types (`function`, `table`, `any`, and `none`)
- Record types, which classify record values based on field names and value types
- List types, which classify lists using a single item base type
- Function types, which classify function values based on the types of their parameters and return values
- Table types, which classify table values based on column names, column types, and keys
- Nullable types, which classifies the value null in addition to all the values classified by a base type
- Type types, which classify values that are types

The set of ***primitive types*** includes the types of primitive values a number of ***abstract types***, types that do not uniquely classify any values: `function`, `table`, `any`, and `none`. All function values conform to the abstract type `function`, all table values to the abstract type `table`, all values to the abstract type `any`, and no values to the abstract type `none`. An expression of type `none` must raise an error or fail to terminate since no value could be produced that conforms to type `none`. Note that the primitive types `function` and `table` are abstract because no function or table is directly of those types, respectively. The primitive types `record` and `list` are non-abstract because the empty record and the empty list are directly of those types.

All types that are not members of the closed set of primitive types are collectively referred to as ***custom types***. Custom types can be written using a *type-expression*:

type-expression:
    primary-expression
    `type` *primary-type*

type:
    parenthesized-expression
    primary-type

primary-type:
    primitive-type
    record-type
    list-type
    function-type
    table-type
    nullable-type

primitive-type: one of
    `any binary date datetime datetimezone duration function list logical`
    `none null number record table text time type`

The *primitive-type* names are ***contextual keywords*** recognized only in a *type* context. The use of parentheses in a *type* context moves the grammar back to a regular expression context, requiring the use of the type keyword to move back into a type context. For example, to invoke a function in a *type* context, parentheses can be used:

```
type nullable ( Type.ForList({type number}) )
// type nullable {number}
```

Parentheses can also be used to access a variable whose name collides with a *primitive-type* name:

```
let  record = type [ A = any ]  in  type {(record)}
// type {[ A = any ]}
```

The following example defines a type that classifies a list of numbers:

```
type { number }
```

Similarly, the following example defines a custom type that classifies records with mandatory fields named `X` and `Y` whose values are numbers:

```
type [ X = number, Y = number ]
```

The ascribed type of a value is obtained using the standard library function `Value.Type`, as shown in the following examples:

```
Value.Type( 2 )                 // type number
Value.Type( {2} )               // type list
Value.Type( [ X = 1, Y = 2 ] )  // type record
```

The `is` operator is used to determine whether a value's type is compatibile with a given type, as shown in the following examples:

```
1 is number                     // true
1 is text                       // false
{2} is list                     // true
```

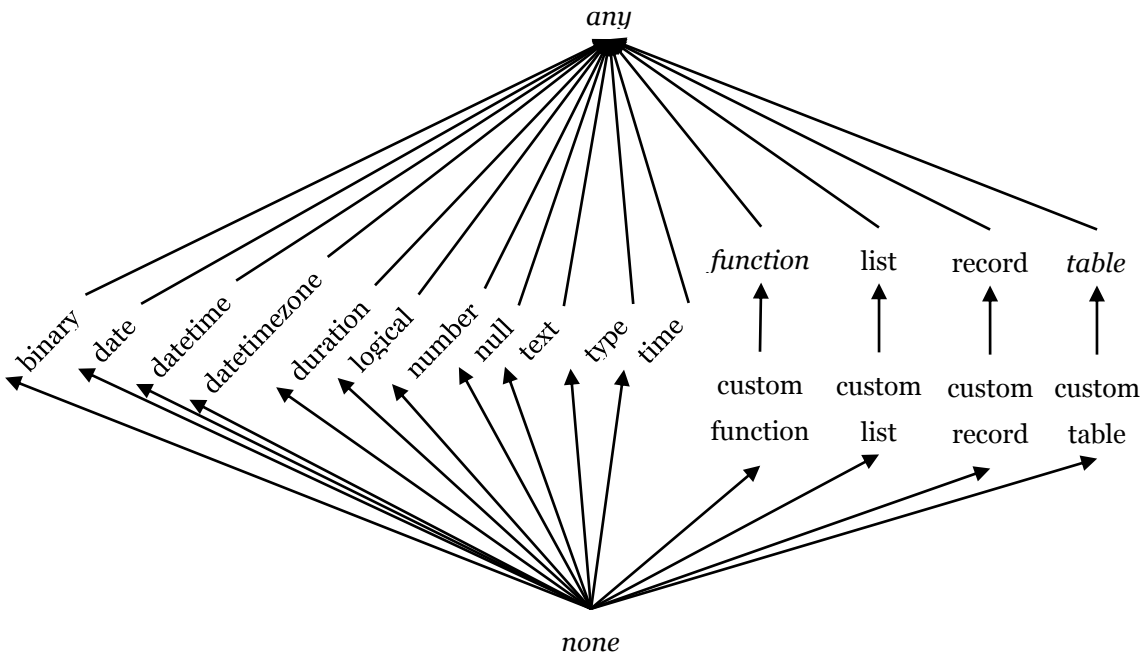
The `as` operator checks if the value is compatible with the given type, and raises an error if it is not. Otherwise, it returns the original value.

```
Value.Type( 1 as number )       // type number
```

```
{2} as text                         // error, type mismatch
```

Note that the `is` and `as` operators only accept primitive types as their right operand. M does not provide means to check values for conformance to custom types.

A type `X` is ***compatible*** with a type `Y` if and only if all values that conform to `X` also conform to `Y`. All types are compatible with type `any` and no types (but `none` itself) are compatible with type `none`. The following graph shows the compatibility relation. (Type compatibility is a reflexive and transitive. It forms a lattice with type `any` as the top and type `none` as the bottom value.) The names of abstract types are set in *italics*.



The following operators are defined for type values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal

The native type of type values is the intrinsic type `type`.

5.1 Primitive Types

Types in the M language form a disjoint hierarchy rooted at type `any`, which is the type that classifies all values. Any M value conforms to exactly one primitive subtype of `any`. The closed set of primitive types deriving from type `any` are as follows:

- `type null`, which classifies the null value

- `type logical`, which classifies the values true and false
- `type number`, which classifies number values
- `type time`, which classifies time values
- `type date`, which classifies date values
- `type datetime`, which classifies datetime values
- `type datetimezone`, which classifies datetimezone values
- `type duration`, which classifies duration values
- `type text`, which classifies text values
- `type binary`, which classifies binary values
- `type type`, which classifies type values.
- `type list`, which classifies list values
- `type record`, which classifies record values
- `type table`, which classifies table values
- `type function`, which classifies function values
- `type anynonnull`, which classifies all values excluding null

The intrinsic type `none` classifies no values.

5.2 Any Type

The type `any` is abstract, classifies all values in M, and all types in M are compatible with `any`. Variables of type any can be bound to all possible values. Since `any` is abstract, it cannot be ascribed to values – that is, no value is directly of type `any`.

5.3 List Types

Any value that is a list conforms to the intrinsic type `list`, which does not place any restrictions on the items within a list value.

list-type:
`{` *item-type* `}`

item-type:
type

The result of evaluating a *list-type* is a ***list type value*** whose base type is `list`.

The following examples illustrate the syntax for declaring homogeneous list types:

```
type { number }        // list of numbers
type { record }        // list of records
type {{ text }}        // list of lists of text values
```

A value conforms to a list type if the value is a list and each item in that list value conforms to the list type's item type.

The item type of a list type indicates a bound: all items of a conforming list conform to the item type.

5.4 Record Types

Any value that is a record conforms to the intrinsic type record, which does not place any restrictions on the field names or values within a record value. A ***record-type value*** is used to restrict the set of valid names as well as the types of values that are permitted to be associated with those names.

record-type:
    `[` *open-record-marker* `]`
    `[` *field-specification-list* `]`
    `[` *field-specification-list* `,` *open-record-marker* `]`

field-specification-list:
    field-specification
    field-specification `,` *field-specification-list*

field-specification:
    `optional`$_{opt}$ *identifier field-type-specification*$_{opt}$

field-type-specification:
    `=` *field-type*

field-type:
    type

open-record-marker:
    `...`

The result of evaluating a *record-type* is a type value whose base type is `record`.

The following examples illustrate the syntax for declaring record types:

```
type [ X = number, Y = number]
type [ Name = text, Age = number ]
type [ Title = text, optional Description = text ]
type [ Name = text, ... ]
```

Record types are ***closed*** by default, meaning that additional fields not present in the *field-specification-list* are not allowed to be present in conforming values. Including the *open-record-marker* in the record type declares the type to be ***open***, which permits fields not present in the field specification list. The following two expressions are equivalent:

```
type record     // primitive type classifying all records
type [ ... ]    // custom type classifying all records
```

A value conforms to a record type if the value is a record and each field specification in the record type is satisfied. A field specification is satisfied if any of the following are true:

- A field name matching the specification's identifier exists in the record and the associated value conforms to the specification's type

- The specification is marked as optional and no corresponding field name is found in the record

A conforming value may contain field names not listed in the field specification list if and only if the record type is open.

5.5 Function Types

Any function value conforms to the primitive type `function`, which does not place any restrictions on the types of the function's formal parameters or the function's return value. A custom ***function-type value*** is used to place type restrictions on the signatures of conformant function values.

function-type:
`function (` *parameter-specification-list*$_{opt}$ `)` *function-return-type*

parameter-specification-list:
required-parameter-specification-list
required-parameter-specification-list `,` *optional-parameter-specification-list*
optional-parameter-specification-list

required-parameter-specification-list:
required-parameter-specification
required-parameter-specification `,` *required-parameter-specification-list*

required-parameter-specification:
parameter-specification

optional-parameter-specification-list:
optional-parameter-specification
optional-parameter-specification `,` *optional-parameter-specification-list*

optional-parameter-specification:
`optional` *parameter-specification*

parameter-specification:
parameter-name parameter-type

function-return-type:
assertion

assertion:
`as` *nullable-primitve-type*

The result of evaluating a *function-type* is a type value whose base type is `function`.

The following examples illustrate the syntax for declaring function types:

```
type function (x as text) as number
type function (y as number, optional z as text) as any
```

A function value conforms to a function type if the return type of the function value is compatible with the function type's return type and each parameter specification of the function type is compatible to the positionally corresponding formal parameter of the function. A parameter specification is compatible with a formal parameter if the specified

parameter-type type is compatible with the type of the formal parameter and the parameter specification is optional if the formal parameter is optional.

Formal parameter names are ignored for the purposes of determining function type conformance.

5.6 Table types

A ***table-type value*** is used to define the structure of a table value.

table-type:
`table` *row-type*

row-type:
`[` *field-specification-list* `]`

The result of evaluating a *table-type* is a type value whose base type is `table`.

The ***row type*** of a table specifies the column names and column types of the table as a closed record type. So that all table values conform to the type `table`, its row type is type `record` (the empty open record type). Thus, type table is abstract since no table value can have type `table`'s row type (but all table values have a row type that is compatible with type `table`'s row type). The following example shows the construction of a table type:

```
type table [A = text, B = number, C = binary]
// a table type with three columns named A, B, and C
// of column types text, number, and binary, respectively
```

A table-type value also carries the definition of a table value's ***keys***. A key is a set of column names. At most one key can be designated as the table's ***primary key***. (Within M, table keys have no semantic meaning. However, it is common for external data sources, such as databases or OData feeds, to define keys over tables. Power Query uses key information to improve performance of advanced functionality, such as cross-source join operations.)

The standard library functions `Type.TableKeys`, `Type.AddTableKey`, and `Type.ReplaceTableKeys` can be used to obtain the keys of a table type, add a key to a table type, and replace all keys of a table type, respectively.

```
Type.AddTableKey(tableType, {"A", "B"}, false)
// add a non-primary key that combines values from columns A and B
Type.ReplaceTableKeys(tableType, {})
// returns type value with all keys removed
```

5.7 Nullable types

For any `type` *T*, a nullable variant can be derived by using *nullable-type*:

nullable-type:
`nullable` *type*

The result is an abstract type that allows values of type *T* or the value `null`.

```
42 is nullable number                  // true
null is nullable number                // true
```

Ascription of `type nullable` *T* reduces to ascription of `type null` or `type` *T*. (Recall that nullable types are abstract and no value can be directly of abstract type.)

```
Value.Type(42 as nullable number)       // type number
Value.Type(null as nullable number)     // type null
```

The standard library functions `Type.IsNullable` and `Type.NonNullable` can be used to test a type for nullability and to remove nullability from a type.

The following hold (for any `type` *T*):

- `type T` is compatible with `type nullable T`
- `Type.NonNullable(type T)` is compatible with `type T`

The following are pairwise equivalent (for any `type` *T*):

```
type nullable any
any

Type.NonNullable(type any)
type anynonnull

type nullable none
type null

Type.NonNullable(type null)
type none

type nullable nullable T
type nullable T

Type.NonNullable(Type.NonNullable(type T))
Type.NonNullable(type T)

Type.NonNullable(type nullable T)
type T

type nullable (Type.NonNullable(type T))
type T
```

5.8 Ascribed type of a value

A value's ***ascribed type*** is the type to which a value is *declared* to conform. When a value is ascribed a type, only a limited conformance check occurs. *M does not perform conformance checking beyond a nullable primitive type. M program authors that choose to ascribe values*

with type definitions more complex than a nullable primitive-type must ensure that such values conform to these types.

A value may be ascribed a type using the library function `Value.ReplaceType`. The function either returns a new value with the type ascribed or raises an error if the new type is incompatible with the value's native primitive type. In particular, the function raises an error when an attempt is made to ascribe an abstract type, such as `any`.

Library functions may choose to compute and ascribe complex types to results based on the ascribed types of the input values.

The ascribed type of a value may be obtained using the library function `Value.Type`. For example:

```
Value.Type( Value.ReplaceType( {1}, type {number} )
// type {number}
```

5.9 Type equivalence and compatibility

Type equivalence is not defined in M. Any two type values that are compared for equality may or may not return `true`. However, the relation between those two types (whether `true` or `false`) will always be the same.

Compatibility between a given type and a nullable primitive type can be determined using the library function `Type.Is`, which accepts an arbitrary type value as its first and a nullable primitive type value as its second argument:

```
Type.Is(type text, type nullable text)    // true
Type.Is(type nullable text, type text)    // false
Type.Is(type number, type text)           // false
Type.Is(type [a=any], type record)        // true
Type.Is(type [a=any], type list)          // false
```

There is no support in M for determining compatibility of a given type with a custom type.

The standard library does include a collection of functions to extract the defining characteristics from a custom type, so specific compatibility tests can be implemented as M expressions. Below are some examples; consult the M library specification for full details.

```
Type.ListItem( type {number} )
  // type number
Type.NonNullable( type nullable text )
  // type text
Type.RecordFields( type [A=text, B=time] )
  // [ A = [Type = type text, Optional = false],
  //   B = [Type = type time, Optional = false] ]
Type.TableRow( type table [X=number, Y=date] )
  // type [X = number, Y = date]
```

```
Type.FunctionParameters(
        type function (x as number, optional y as text) as number)
  // [ x = type number, y = type nullable text ]
Type.FunctionRequiredParameters(
        type function (x as number, optional y as text) as number)
  // 1
Type.FunctionReturn(
        type function (x as number, optional y as text) as number)
  // type number
```

6. Operators

This section defines the behavior of the various M operators.

6.1 Operator precedence

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the * operator has higher precedence than the binary + operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive-expression* consists of a sequence of *multiplicative-expression*'s separated by + or - operators, thus giving the + and - operators lower precedence than the * and / operators.

The *parenthesized-expression* production can be used to change the default precedence ordering.

parenthesized-expression:
`(` *expression* `)`

For example:

```
1 + 2 * 3       // 7
(1 + 2) * 3     // 9
```

The following table summarizes the M operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

Category	Expression	Description
Primary	*i* `@`*i*	Identifier expression
	`(`*x*`)`	Parenthesized expression
	x`[`*i*`]`	Lookup
	x`{`*y*`}`	Item access
	x`(…)`	Function invocation
	`{`*x*`, `*y*`, …}`	List initialization
	`[ `*i*` = `*x*`, … ]`	Record initialization
	`...`	Not implemented
Unary	`+`*x*	Identity
	`-`*x*	Negation
	`not `*x*	Logical negation
Metadata	*x* `meta` *y*	Associate metadata
Multiplicative	*x* `*` *y*	Multiplication
	x `/` *y*	Division
Additive	*x* `+` *y*	Addition
	x `-` *y*	Subtraction
Relational	*x* `<` *y*	Less than
	x `>` *y*	Greater than
	x `<=` *y*	Less than or equal
	x `>=` *y*	Greater than or equal
Equality	*x* `=` *y*	Equal
	x `<>` *y*	Not equal
Type assertion	*x* `as` *y*	Is compatible nullable-primitive type or error
Type conformance	*x* `is` *y*	Test if compatible nullable-primitive type
Logical AND	*x* `and` *y*	Short-circuiting conjunction
Logical OR	*x* `or` *y*	Short-circuiting disjunction

6.2 Operators and metadata

Every value has an associated record value that can carry additional information about the value. This record is referred to as the ***metadata record*** for a value. A metadata record can be associated with any kind of value, even `null`. The result of such an association is a new value with the given metadata.

A metadata record is just a regular record and can contain any fields and values that a regular record can, and itself has a metadata record. Associating a metadata record with a value is "non-intrusive". It does not change the value's behavior in evaluations except for those that explicitly inspect metadata records.

Every value has a default metadata record, even if one has not been specified. The default metadata record is empty. The following examples show accessing the metadata record of a text value using the `Value.Metadata` standard library function:

```
Value.Metadata( "Mozart" )    // []
```

Metadata records are generally *not preserved* when a value is used with an operator or function that constructs a new value. For example, if two text values are concatenated using the `&` operator, the metadata of the resulting text value is the empty record `[]`. The following expressions are equivalent:

```
"Amadeus " & ("Mozart" meta [ Rating = 5 ])
"Amadeus " & "Mozart"
```

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata (rather than merge metadata into possibly existing metadata).

The only operator that returns results that carry metadata is the meta operator (§6.5).

6.3 Structurally recursive operators

Values can be ***cyclic***. For example:

```
let l = {0, @l} in l
// {0, {0, {0, … }}}
[A={B}, B={A}]
// [A = {{ … }}, B = {{ … }}]
```

M handles cyclic values by keeping construction of records, lists, and tables lazy. An attempt to construct a cyclic value that does not benefit from interjected lazy structured values yields an error:

```
[A=B, B=A]
// [A = Error.Record("Expression.Error",
//         "A cyclic reference was encountered during evaluation),
//  B = Error.Record("Expression.Error",
//         "A cyclic reference was encountered during evaluation),
// ]
```

Some operators in M are defined by structural recursion. For instance, equality of records and lists is defined by the conjoined equality of corresponding record fields and item lists, respectively.

For non-cyclic values, applying structural recursion yields a ***finite expansion*** of the value: shared nested values will be traversed repeatedly, but the process of recursion always terminates.

A cyclic value has an ***infinite expansion*** when applying structural recursion. The semantics of M makes no special accomodations for such infinite expansions – an attempt to compare cyclic values for equality, for instance, will typically run out of resources and terminate exceptionally.

6.4 Selection and Projection Operators

The selection and projection operators allow data to be extracted from list and record values.

6.4.1 Item Access

A value may be selected from a list or table based on its zero-based position within that list or table using an *item-access-expression*.

item-access-expression:
    item-selection
    optional-item-selection

item-selection:
    primary-expression `{` *item-selector* `}`

optional-item-selection:
    primary-expression `{` *item-selector* `} ?`

item-selector:
    expression

The *item-access-expression* `x{y}` returns:

- For a list `x` and a number `y`, the item of list `x` at position `y`. The first item of a list is considered to have an ordinal index of zero. If the requested position does not exist in the list, an error is raised.
- For a table `x` and a number `y`, the row of table `x` at position `y`. The first row of a table is considered to have an ordinal index of zero. If the requested position does not exist in the table, an error is raised.
- For a table `x` and a record `y`, the row of table `x` that matches the field values of record `y` for fields with field names that match corresponding table-column names. If there is no unique matching row in the table, an error is raised.

For example:

```
{"a","b","c"}{0}                    // "a"
{1, [A=2], 3}{1}                    // [A=2]
{true, false}{2}                    // error
```

```
#table({"A","B"},{{0,1},{2,1}}){0}        // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}    // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]}    // error
#table({"A","B"},{{0,1},{2,1}}){[B=1]}    // error
```

The *item-access-expression* also supports the form `x{y}?`, which returns `null` when position (or match) `y` does not exist in list or table `x`. If there are multiple matches for `y`, an error is still raised.

For example:

```
{"a","b","c"}{0}?                         // "a"
{1, [A=2], 3}{1}?                         // [A=2]
{true, false}{2}?                         // null
#table({"A","B"},{{0,1},{2,1}}){0}        // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}    // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]}    // null
#table({"A","B"},{{0,1},{2,1}}){[B=1]}    // error
```

Item access does not force the evaluation of list or table items other than the one being accessed. For example:

```
{ error "a", 1, error "c"}{1}       // 1
{ error "a", error "b"}{1}          // error "b"
```

The following holds when the item access operator `x{y}` is evaluated:

- Errors raised during the evaluation of expressions `x` or `y` are propagated.
- The expression `x` produces a list or a table value.
- The expression `y` produces a number value or, if `x` produces a table value, a record value.
- If `y` produces a number value and the value of `y` is negative, an error with reason code `"Expression.Error"` is raised.
- If `y` produces a number value and the value of `y` is greater than or equal to the count of `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `x{y}?` is used, in which case the value `null` is returned.
- If `x` produces a table value and `y` produces a record value and there are no matches for `y` in `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `x{y}?` is used, in which case the value `null` is returned.
- If `x` produces a table value and `y` produces a record value and there are multiple matches for `y` in `x`, an error with reason code `"Expression.Error"` is raised.

No items in `x` other than that at position `y` is evaluated during the process of item selection. (For streaming lists or tables, the items or rows preceding that at position `y` are skipped over, which may cause their evaluation, depending on the source of the list or table.)

6.4.2 Field Access

The *field-access-expression* is used to ***select*** a value from a record or to ***project*** a record or table to one with fewer fields or columns, respectively.

field-access-expression:
    field-selection
    implicit-target-field-selection
    projection
    implicit-target-projection

field-selection:
    primary-expression field-selector

field-selector:
    required-field-selector
    optional-field-selector

required-field-selector:
    `[` *field-name* `]`

optional-field-selector:
    `[` *field-name* `]` `?`

field-name:
    generalized-identifier

implicit-target-field-selection:
    field-selector

projection:
    primary-expression required-projection
    primary-expression optional-projection

required-projection:
    `[` *required-selector-list* `]`

optional-projection:
    `[` *required-selector-list* `]` `?`

required-selector-list:
    required-field-selector
    required-selector-list `,` *required-field-selector*

implicit-target -projection:
    record-projection

The simplest form of field access is ***required field selection***. It uses the operator `x[y]` to look up a field in a record by field name. If the field `y` does not exist in `x`, an error is raised. The form `x[y]?` is used to perform ***optional field selection***, and returns `null` if the requested field does not exist in the record.

For example:

```
[A=1,B=2][B]                    // 2
[A=1,B=2][C]                    // error
```

```
[A=1,B=2][C]?                    // null
```

Collective access of multiple fields is supported by the operators for ***required record projection*** and ***optional record projection***. The operator `x[[y1],[y2],…]` projects the record to a new record with fewer fields (selected by `y1`, `y2`, …). If a selected field does not exist, an error is raised. The operator `x[[y1],[y2],…]` projects the record to a new record with the fields selected by `y1`, `y2`, …; if a field is missing, `null` is used instead.

For example:

```
[A=1,B=2][[B]]                   // [B=2]
[A=1,B=2][[C]]                   // error
[A=1,B=2][[B],[C]]?              // [B=2,C=null]
```

The forms `[y]` and `[y]?` are supported as a ***shorthand reference*** to the identifier `_` (underscore). The following two expressions are equivalent:

```
[A]
_[A]
```

The following example illustrates the shorthand form of field access:

```
let _ = [A=1,B=2] in [A] //1
```

The form `[[y1],[y2],…]` and `[[y1],[y2],…]?` are also supported as a shorthand and the following two expressions are likewise equivalent:

```
[[A],[B]]
_[[A],[B]]
```

The shorthand form is particularly useful in combination with the `each` shorthand, a way to introduce a function of a single parameter named `_` (for details, see §9.7). Together, the two shorthands simplify common higher-order functional expressions:

```
List.Select( {[a=1, b=1], [a=2, b=4]}, each [a] = [b])
// {[a=1, b=1]}
```

The above expression is equivalent to the following more cryptic looking longhand:

```
List.Select( {[a=1, b=1], [a=2, b=4]}, (_) => _[a] = _[b])
// {[a=1, b=1]}
```

Field access does not force the evaluation of fields other than the one(s) being accessed. For example:

```
[A=error "a", B=1, C=error "c"][B]     // 1
[A=error "a", B=error "b"][B]          // error "b"
```

The following holds when a field access operator `x[y]`, `x[y]?`, `x[[y]]`, or `x[[y]]?` is evaluated:

- Errors raised during the evaluation of expression `x` are propagated.

- Errors raised when evaluating field `y` are permanently associated with field `y`, then propagated. Any future access to field `y` will raise the identical error.
- The expression `x` produces a record or table value or an error is raised.
- If the identifier `y` names a field that does not exist in `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `…?` is used, in which case the value `null` is returned.

No fields of `x` other than that named by `y` is evaluated during the process of field access.

6.5 Metadata operator

The metadata record for a value is ammended using the ***meta operator*** (`x meta y`).

metadata-expression:
as-expression
as-expression `meta` *unary-expression*

The following example constructs a text value with a metadata record using the `meta` operator and then accesses the metadata record of the resulting value using `Value.Metadata`:

```
Value.Metadata( "Mozart" meta [ Rating = 5 ] )
// [Rating = 5 ]
Value.Metadata( "Mozart" meta [ Rating = 5 ] )[Rating]
// 5
```

The following holds when applying the metadata combining operator `x meta y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The `y` expression must be a record, or an error with reason code `"Expression.Error"` is raised.
- The resulting metadata record is `x`'s metadata record merged with `y`. (For the semantics of record merge, see §6.10.2.1.)
- The resulting value is the value from the `x` expression, without its metadata, with the newly computed metadata record attached.

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata (rather than merge metadata into possibly existing metadata). The following expressions are equivalent:

```
x meta y
Value.ReplaceMetadata(x, Value.Metadata(x) & y)
Value.RemoveMetadata(x) meta (Value.Metadata(x) & y)
```

6.6 Equality operators

The ***equality operator*** `=` is used to determine if two values are the equal. The ***inequality operator*** `<>` is used to determine if two values are not equal.

equality-expression:
relational-expression
relational-expression `=` *equality-expression*
relational-expression `<>` *equality-expression*

For example:

```
1 = 1                  // true
1 = 2                  // false
1 <> 1                 // false
1 <> 2                 // true
null = true            // false
null = null            // true
```

Metadata is not part of equality or inequality comparison. For example:

```
(1 meta [ a = 1 ]) = (1 meta [ a = 2 ])    // true
(1 meta [ a = 1 ]) = 1                     // true
```

The following holds when applying the equality operators `x = y` and `x <> y`:

- Errors raised when evaluating the x or y expressions are propagated.
- The `=` operator has a result of `true` if the values are equal, and `false` otherwise.
- The `<>` operator has a result of `false` if the values are equal, and `true` otherwise.
- Metadata records are not included in the comparison.
- If values produced by evaluating the `x` and `y` expressions are not the same kind of value, then the values are not equal.
- If the values produced by evaluating the x and y expression are the same kind of value, then there are specific rules for determining if they are equal, as defined below.
- The following is always true:

  ```
  (x = y) = not (x <> y)
  ```

The equality operators are defined for the following types:

- The `null` value is only equal to itself.

  ```
  null = null         // true
  null = true         // false
  null = false        // false
  ```

- The logical values `true` and `false` are only equal to themselves. For example:

  ```
  true = true         // true
  false = false       // true
  true = false        // false
  true = 1            // false
  ```

- Numbers are compared using the specified precision:

- If either number is `#nan`, then the numbers are not the same.
- When neither number is `#nan`, then the numbers are compared using a bit-wise comparison of the numeric value.
- `#nan` is the only value that is not equal to itself.

For example:

```
1 = 1,              // true
1.0 = 1             // true
2 = 1               // false
#nan = #nan         // false
#nan <> #nan        // true
```

- Two durations are equal if they represent the same number of 100-nanosecond ticks.
- Two times are equal if the magnitudes of their parts (hour, minute, second) are equal.
- Two dates are equal if the magnitudes of their parts (year, month, day) are equal.
- Two datetimes are equal if the magnitudes of their parts (year, month, day, hour, minute, second) are equal.
- Two datetimezones are equal if the corresponding UTC datetimes are equal. To arrive at the corresponding UTC datetime, the hours/minutes offset is subtracted from the datetime component of the datetimezone.
- Two text values are equal if using an ordinal, case-sensitive, culture-insensitive comparison they have the same length and equal characters at corresponding positions.
- Two list values are equal if all of the following are true:
  - Both lists contain the same number of items.
  - The values of each positionally corresponding item in the lists are equal. This means that not only do the lists need to contain equal items, the items need to be in the same order.

  For example:

  ```
  {1, 2} = {1, 2}     // true
  {2, 1} = {1, 2}     // false
  {1, 2, 3} = {1, 2}  // false
  ```

- Two records are equal if all of the following are true:
  - The number of fields is the same.
  - Each field name of one record is also present in the other record.
  - The value of each field of one record is equal to the like-named field in the other record.

  For example:

```
[ A = 1, B = 2 ] = [ A = 1, B = 2 ]        // true
[ B = 2, A = 1 ] = [ A = 1, B = 2 ]        // true
[ A = 1, B = 2, C = 3 ] = [ A = 1, B = 2 ] // false
[ A = 1 ] = [ A = 1, B = 2 ]               // false
```

- Two tables are equal if all of the following are true:
  - The number of columns is the same.
  - Each column name in one table is also present in the other table.
  - The number of rows is the same.
  - Each row has equal values in corresponding cells.

  For example:

```
#table({"A","B"},{{1,2}}) = #table({"A","B"},{{1,2}})
// true
#table({"A","B"},{{1,2}}) = #table({"X","Y"},{{1,2}})
// false
#table({"A","B"},{{1,2}}) = #table({"B","A"},{{2,1}})
// true
```

- A function value is equal to itself, but may or may not be equal to another function value. If two function values are considered equal, then they will behave identically when invoked.

  Two given function values will always have the same equality relationship.
- A type value is equal to itself, but may or may not be equal to another type value. If two type values are considered equal, then they will behave identically when queried for conformance.

  Two given type values will always have the same equality relationship.

6.7 Relational operators

The <, >, <=, and >= operators are called the ***relational operators***.

relational-expression:
additive-expression
additive-expression `<` *relational-expression*
additive-expression `>` *relational-expression*
additive-expression `<=` *relational-expression*
additive-expression `>=` *relational-expression*

These operators are used to determine the relative ordering relationship between two values, as shown in the following table:

Operation	Result
`x < y`	`true` if `x` is less than `y`, `false` otherwise
`x > y`	`true` if `x` is greater than `y`, `false` otherwise
`x <= y`	`true` if `x` is less than or equal to `y`, `false` otherwise
`x >= y`	`true` if `x` is greater than or equal to `y`, `false` otherwise

For example:

```
0 <= 1                // true
null < 1              // null
null <= null          // null
"ab" < "abc"          // true
#nan >= #nan          // false
#nan <= #nan          // false
```

The following holds when evaluating an expression containing the relational operators:

- Errors raised when evaluating the `x` or `y` operand expressions are propagated.
- The values produced by evaluating both the `x` and `y` expressions must be a number, date, datetime, datetimezone, duration, logical, null or time value. Otherwise, an error with reason code `"Expression.Error"` is raised.
- If either or both operands are `null`, the result is the `null` value.
- If both operands are logical, the value `true` is considered to be greater than `false`.
- If both operands are durations, then the values are compared according to the total number of 100-nanosecond ticks they represent.
- Two times are compared by comparing their hour parts and, if equal, their minute parts and, if equal, their second parts.
- Two dates are compared by comparing their year parts and, if equal, their month parts and, if equal, their day parts.
- Two datetimes are compared by comparing their year parts and, if equal, their month parts and, if equal, their day parts and, if equal, their hour parts and, if equal, their minute parts and, if equal, their second parts.
- Two datetimezones are compared by normalizing them to UTC by subtracting their hour/minute offset and then comparing their datetime components.
- Two numbers `x` and `y` are compared according to the rules of the IEEE 754 standard:
  - If either operand is `#nan`, the result is `false` for all relational operators.
  - When neither operand is `#nan`, the operators compare the values of the two floating-point operands with respect to the ordering

    −∞ < −max < ... < −min < −0.0 = +0.0 < +min < ... < +max < +∞

where min and max are the smallest and largest positive finite values that can be represented. The M names for −∞ and +∞ are `-#infinity` and `#infinity`.

Notable effects of this ordering are:

- Negative and positive zeros are considered equal.
- A `-#infinity` value is considered less than all other number values, but equal to another `-#infinity`.
- A `#infinity` value is considered greater than all other number values, but equal to another `#infinity`.

6.8 Conditional logical operators

The `and` and `or` operators are called the ***conditional logical operators***.

logical-or-expression:
logical-and-expression
logical-and-expression `or` *logical-or-expression*

logical-and-expression:
is-expression
is-expression `and` *logical-and-expression*

The `or` operator returns `true` when at least one of its operands is `true`. The right operand is evaluated if and only if the left operand is not `true`.

The `and` operator returns `false` when at least one of its operands is `false`. The right operand is evaluated if and only if the left operand is not `false`.

Truth tables for the `or` and `and` operators are shown below, with the result of evaluating the left operand expression on the vertical axis and the result of evaluating the right operand expression on the horizontal axis.

and	true	false	null	error
true	true	false	null	error
false	false	false	false	false
null	null	false	null	error
error	error	error	error	error

or	true	false	null	error

`true`	`true`	`true`	`true`	`true`
`false`	`true`	`false`	`null`	`error`
`null`	`true`	`null`	`null`	`error`
`error`	`error`	`error`	`error`	`error`

The following holds when evaluating an expression containing conditional logical operators:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The conditional logical operators are defined over the types `logical` and `null`. If the operand values are not of those types, an error with reason code `"Expression.Error"` is raised.
- The result is a logical value.
- In the expression `x or y`, the expression `y` will be evaluated if and only if `x` does not evaluate to `true`.
- In the expression `x and y`, the expression `y` will be evaluated if and only if `x` does not evaluate to `false`.

The last two properties give the conditional logical operators their "conditional" qualification; properties also referred to as "short-circuiting". These properties are useful to write compact ***guarded predicates***. For example, the following expressions are equivalent:

```
d <> 0 and n/d > 1
if d <> 0 then n/d > 1 else false
```

6.9 Arithmetic Operators

The +, -, * and / operators are the ***arithmetic operators***.

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `–` *multiplicative-expression*

multiplicative-expression:
metadata- expression
multiplicative-expression `*` *metadata-expression*
multiplicative-expression `/` *metadata-expression*

6.9.1 Precision

Numbers in M are stored using a variety of representations to retain as much information as possible about numbers coming from a variety of sources. Numbers are only converted from one representation to another as needed by operators applied to them. Two precisions are supported in M:

Precision	Semantics
`Precision.Decimal`	128-bit decimal representation with a range of ±1.0 × 10−28 to ±7.9 × 1028 and 28-29 significant digits.
`Precision.Double`	Scientific representation using mantissa and exponent; conforms to the 64-bit binary double-precision IEEE 754 arithmetic standard [IEEE 754-2008].

Arithmetic operations are performed by chosing a precision, converting both operands to that precision (if necessary), then performing the actual operation, and finally returning a number in the chosen precision.

The built-in arithmetic operators (`+`, `-`, `*`, `/`) use Double Precision. Standard library functions (`Value.Add`, `Value.Subtract`, `Value.Multiply`, `Value.Divide`) can be used to request these operations using a specific precision model.

- No numeric overflow is possible: `#infinity` or `-#infinity` represent values of magnitues too large to be represented.
- No numeric underflow is possible: `0` and `-0` represent values of magnitutes too small to be represented.
- The IEEE 754 special value `#nan` (NaN – Not a Number) is used to cover arithmetically invalid cases, such as a division of zero by zero.
- Conversion from Decimal to Double precision is performed by rounding decimal numbers to the nearest equivalent double value.
- Conversion from Double to Decimal precision is perfomed by rounding double numbers to the nearest equivalent decimal value and, if necessary, overflowing to `#infinity` or `-#infinity` values.

6.9.2 Addition operator

The interpretation of the addition operator (`x + y`) is dependent on the kind of value of the evaluated expressions x and y, as follows:

x	y	Result	Interpretation
`type number`	`type number`	`type number`	Numeric sum
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type duration`	`type duration`	Numeric sum of magnitudes
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	
`type` *datetime*	`type duration`	`type` *datetime*	Datetime offset by duration
`type duration`	`type` *datetime*	`type` *datetime*	
`type` *datetime*	`null`	`null`	
`null`	`type` *datetime*	`null`	

In the table, `type` *datetime* stands for any of `type date`, `type datetime`, `type datetimezone`, or `type time`. When adding a duration and a value of some type *datetime*, the resulting value is of that same type.

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

6.9.2.1 Numeric sum

The sum of two numbers is computed using the ***addition operator***, producing a number. For example:

```
1 + 1              // 2
#nan + #infinity   // #nan
```

The addition operator `+` over numbers uses Double Precision; the standard library function `Value.Add` can be used to specify Decimal Precision. The following holds when computing a sum of numbers:

- The sum in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008]. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x + y` is too large to be represented in the destination type, `z` is an infinity with the same sign as `x + y`.

+	y	+0	-0	+∞	-∞	NaN

x	z	x	x	+∞	−∞	NaN
+0	y	+0	+0	+∞	−∞	NaN
−0	y	+0	−0	+∞	−∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
−∞	−∞	−∞	−∞	NaN	−∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The sum in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

6.9.2.2 Sum of durations

The sum of two durations is the duration representing the sum of the number of 100-nanosecond ticks represented by the durations. For example:

```
#duration(2,1,0,15.1) + #duration(0,1,30,45.3)
// #duration(2, 2, 31, 0.4)
```

6.9.2.3 Datetime offset by duration

A *datetime* `x` and an duration `y` may be added using `x + y` to compute a new *datetime* whose distance from `x` on a linear timeline is exactly the magnitude of `y`. Here, *datetime* stands for any of `Date`, `DateTime`, `DateTimeZone`, or `Time` and a non-null result will be of the same type. The datetime offset by duration may be computed as follows:

- If the datetime's days since epoch value is specified, construct a new datetime with the following information elements:
  - Calculate a new days since epoch equivalent to dividing the magnitude of y by the number of 100-nanosecond ticks in a 24-hour period, truncating the decimal portion of the result, and adding this value to the x's days since epoch.
  - Calculate a new ticks since midnight equivalent to adding the magnitude of y to the x's ticks since midnight, modulo the number of 100-nanosecond ticks in a 24-hour period. If x does not specify a value for ticks since midnight, a value of 0 is assumed.
  - Copy x's value for minutes offset from UTC unchanged.
- If the datetime's days since epoch value is unspecified, construct a new datetime with the following information elements specified:
  - Calculate a new ticks since midnight equivalent to adding the magnitude of y to the x's ticks since midnight, modulo the number of 100-nanosecond ticks in a 24-hour period. If x does not specify a value for ticks since midnight, a value of 0 is assumed.
  - Copy x's values for days since epoch and minutes offset from UTC unchanged.

The following examples show calculating the absolute temporal sum when the datetime specifies the *days since epoch*:

```
#date(2010,05,20) + #duration(0,8,0,0)
    //#datetime( 2010, 5, 20, 8, 0, 0 )
    //2010-05-20T08:00:00


#date(2010,01,31) + #duration(30,08,0,0)
    //#datetime(2010, 3, 2, 8, 0, 0)
    //2010-03-02T08:00:00


#datetime(2010,05,20,12,00,00,-08) + #duration(0,04,30,00)
    //#datetime(2010, 5, 20, 16, 30, 0, -8, 0)
    //2010-05-20T16:30:00-08:00


#datetime(2010,10,10,0,0,0,0) + #duration(1,0,0,0)
   //#datetime(2010, 10, 11, 0, 0, 0, 0, 0)
   //2010-10-11T00:00:00+00:00
```

The following example shows calculating the datetime offset by duration for a given time:

```
#time(8,0,0) + #duration(30,5,0,0)
   //#time(13, 0, 0)
   //13:00:00
```

6.9.3 Subtraction operator

The interpretation of the subtraction operator (`x - y`) is dependent on the kind of the value of the evaluated expressions x and y, as follows:

x	Y	Result	Interpretation
`type number`	`type number`	`type number`	Numeric difference
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type duration`	`type duration`	Numeric difference of magnitudes
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	
`type` *datetime*	`type` *datetime*	`type duration`	Duration between datetimes
`type` *datetime*	`type duration`	`type` *datetime*	Datetime offset by negated duration
`type` *datetime*	`null`	`null`	
`null`	`type` *datetime*	`null`	

In the table, `type` *datetime* stands for any of `type date`, `type datetime`, `type datetimezone`, or `type time`. When subtracting a duration from a value of some type *datetime*, the resulting value is of that same type.

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

6.9.3.1 Numeric difference

The difference between two numbers is computed using the ***subtraction operator***, producing a number. For example:

```
1 - 1                  // 0
#nan - #infinity       // #nan
```

The subtraction operator `-` over numbers uses Double Precision; the standard library function `Value.Subtract` can be used to specify Decimal Precision. The following holds when computing a difference of numbers:

- The difference in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008]. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If `x - y` is too large to be represented in the destination type, `z` is an infinity with the same sign as `x - y`.

-	y	+0	-0	+∞	-∞	NaN
x	z	x	x	-∞	+∞	NaN

+0	-y	+0	+0	−∞	+∞	NaN
-0	-y	-0	+0	−∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
−∞	−∞	−∞	−∞	−∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The difference in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

6.9.3.2 Difference of durations

The difference of two durations is the duration representing the difference between the number of 100-nanosecond ticks represented by each duration. For example:

```
#duration(1,2,30,0) - #duration(0,0,0,30.45)
// #duration(1, 2, 29, 29.55)
```

6.9.3.3 Datetime offset by negated duration

A *datetime* `x` and a duration `y` may be subtracted using `x - y` to compute a new *datetime*. Here, *datetime* stands for any of `date`, `datetime`, `datetimezone`, or `time`. The resulting *datetime* has a distance from `x` on a linear timeline that is exactly the magnitude of `y`, in the direction opposite the sign of `y`. Subtracting positive durations yields results that are backwards in time relative to `x`, while subtracting negative values yields results that are forwards in time.

```
#date(2010,05,20) - #duration(00,08,00,00)
   //#datetime(2010, 5, 19, 16, 0, 0)
   //2010-05-19T16:00:00
#date(2010,01,31) - #duration( 30,08,00,00)
   //#datetime(2009, 12, 31, 16, 0, 0)
   //2009-12-31T16:00:00
```

6.9.3.4 Duration between two datetimes

Two *datetimes* `t` and `u` may be subtracted using `t - u` to compute the duration between them. Here, *datetime* stands for any of `date`, `datetime`, `datetimezone`, or `time`. The duration produced by subtracting `u` from `t` must yield `t` when added to `u`.

```
#date(2010,01,31) - #date(2010,01,15)
// #duration(16,00,00,00)
// 16.00:00:00
```

```
#date(2010,01,15)- #date(2010,01,31)
// #duration(-16,00,00,00)
// -16.00:00:00

#datetime(2010,05,20,16,06,00,-08,00) -
#datetime(2008,12,15,04,19,19,03,00)
// #duration(521,22,46,41)
// 521.22:46:41
```

Subtracting `t – u` when `u > t` results in a negative duration:

```
#time(01,30,00) - #time(08,00,00)
// #duration(0, -6, -30, 0)
```

The following holds when subtracting two *datetimes* using `t - u`:

- u + (t - u) = t

6.9.4 Multiplication operator

The interpretation of the multiplication operator (`x * y`) is dependent on the kind of value of the evaluated expressions x and y, as follows:

X	Y	Result	Interpretation
`type number`	`type number`	`type number`	Numeric product
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type number`	`type duration`	Multiple of duration
`type number`	`type duration`	`type duration`	Multiple of duration
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

6.9.4.1 Numeric product

The product of two numbers is computed using the ***multiplication operator***, producing a number. For example:

```
2 * 4                 // 8
6 * null              // null
#nan * #infinity      // #nan
```

The multiplication operator * over numbers uses Double Precision; the standard library function `Value.Multiply` can be used to specify Decimal Precision. The following holds when computing a product of numbers:

- The product in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008]. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, x and y are positive finite values. z is the result of `x * y`. If the result is too large for the destination type, z is infinity. If the result is too small for the destination type, z is zero.

*	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	-z	+0	-0	+∞	-∞	NaN
-x	-z	+z	-0	+0	-∞	+∞	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+∞	+∞	-∞	NaN	NaN	+∞	-∞	NaN
-∞	-∞	+∞	NaN	NaN	-∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The product in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

6.9.4.2 Multiples of durations

The product of a duration and a number is the duration representing the number of 100-nanosecond ticks represented by the duration operand times the number operand. For example:

```
#duration(2,1,0,15.1) * 2
// #duration(4, 2, 0, 30.2)
```

6.9.5 Division operator

The interpretation of the division operator `(x / y)` is dependent on the kind of value of the evaluated expressions x and y, as follows:

X	Y	Result	Interpretation
`type number`	`type number`	`type number`	Numeric quotient
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type number`	`type duration`	Fraction of duration
`type duration`	`type duration`	`type duration`	Numeric quotient of durations
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

6.9.5.1 Numeric quotient

The quotient of two numbers is computed using the ***division operator***, producing a number. For example:

```
8 / 2                 // 4
8 / 0                 // #infinity
0 / 0                 // #nan
0 / null              // null
#nan / #infinity      // #nan
```

The division operator `/` over numbers uses Double Precision; the standard library function `Value.Divide` can be used to specify Decimal Precision. The following holds when computing a quotient of numbers:

- The quotient in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008]. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

/	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	-z	+∞	-∞	+0	-0	NaN
-x	-z	+z	-∞	+∞	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN

+∞	+∞	−∞	+∞	−∞	NaN	NaN	NaN
−∞	−∞	+∞	−∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The sum in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

6.9.5.2 Quotient of durations

The quotient of two durations is the number representing the quotient of the number of 100-nanosecond ticks represented by the durations. For example:

```
#duration(2,0,0,0) / #duration(0,1,30,0)
// 32
```

6.9.5.3 Scaled durations

The quotient of a duration `x` and a number `y` is the duration representing the quotient of the number of 100-nanosecond ticks represented by the duration `x` and the number `y`. For example:

```
#duration(2,0,0,0) / 32
// #duration(0,1,30,0)
```

6.10 Structure Combination

The combination operator (`x & y`) is defined over the following kinds of values:

X	Y	Result	Interpretation
`type text`	`type text`	`type text`	Concatenation
`type text`	`null`	`null`	
`null`	`type text`	`null`	
`type date`	`type time`	`type datetime`	Merge
`type date`	`null`	`null`	
`null`	`type time`	`null`	
`type list`	`type list`	`type list`	Concatenation
`type record`	`type record`	`type record`	Merge
`type table`	`type table`	`type table`	Concatenation

6.10.1 Concatenation

Two text, two list, or two table values can be concatenated using `x & y`.

The following example illustrates concatenating text values:

```
"AB" & "CDE"              // "ABCDE"
```

The following example illustrates concatenating lists:

```
{1, 2} & {3}              // {1, 2, 3}
```

The following holds when concatenating two values using `x & y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- No error is propagated if an item of either `x` or `y` contains an error.
- The result of concatenating two text values is a text value that contains the value of x immediately followed by y. If either of the operands is null and the other is a text value, the result is null.
- The result of concatenating two lists is a list that contains all the items of `x` followed by all the items of `y`.
- The result of concatenating two tables is a table that has the union of the two operand table's columns. The column ordering of `x` is preserved, followed by the columns only appearing in `y`, preserving their relative ordering. For columns appearing only in one of the operands, `null` is used to fill in cell values for the other operand.

6.10.2 Merge

6.10.2.1 Record merge

Two records can be merged using `x & y`, producing a record that includes fields from both `x` and `y`.

The following examples illustrate merging records:

```
[ x = 1 ] & [ y = 2 ]               // [ x = 1, y = 2 ]
[ x = 1, y = 2 ] & [ x = 3, z = 4 ]  // [ x = 3, y = 2, z = 4 ]
```

The following holds when merging two records using `x + y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- If a field appears in both `x` and `y`, the value from `y` is used.
- The order of the fields in the resulting record is that of `x`, followed by fields in `y` that are not part of `x`, in the same order that they appear in `y`.
- Merging records does not cause evaluation of the values.
- No error is raised because a field contains an error.
- The result is a record.

6.10.2.2 Date-time merge

A date x can be merged with a time y using `x & y`, producing a datetime that combines the parts from both `x` and `y`.

The following example illustrates merging a date and a time:

```
#date(2013,02,26) & #time(09,17,00)
// #datetime(2013,02,26,09,17,00)
```

The following holds when merging two records using `x + y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The result is a datetime.

6.11 Unary operators

The `+`, `-`, and `not` operators are unary operators.

unary-expression:
type-expression
`+` *unary expression*
`-` *unary expression*
`not` *unary expression*

6.11.1 Unary plus operator

The unary plus operator `(+x)` is defined for the following kinds of values:

X	Result	Interpretation
`type number`	`type number`	Unary plus
`type duration`	`type duration`	Unary plus
`null`	`null`	

For other values, an error with reason code `"Expression.Error"` is raised.

The unary plus operator allows a `+` sign to be applied to a number, datetime, or null value. The result is that same value. For example:

```
+ - 1                  // -1
+ + 1                  // 1
+ #nan                 // #nan
+ #duration(0,1,30,0)  // #duration(0,1,30,0)
```

The following holds when evaluating the unary plus operator `+x`:

- Errors raised when evaluating `x` are propagated.
- If the result of evaluating `x` is not a number value, then an error with reason code `"Expression.Error"` is raised.

6.11.2 Unary minus operator

The unary minus operator `(-x)` is defined for the following kinds of values:

X	Result	Interpretation
`type number`	`type number`	Negation
`type duration`	`type duration`	Negation
`null`	`null`	

For other values, an error with reason code "`Expression.Error`" is raised.

The unary minus operator is used to change the sign of a number or duration. For example:

```
- (1 + 1)        // -2
- - 1            // 1
- - - 1          // -1
- #nan           // #nan
- #infinity      // -#infinity
- #duration(1,0,0,0)   // #duration(-1,0,0,0)
- #duration(0,1,30,0)  // #duration(0,-1,-30,0)
```

The following holds when evaluating the unary minus operator `-x`:

- Errors raised when evaluating `x` are propagated.
- If the expression is a number, then the result is the number value from expression `x` with its sign changed. If the value is NaN, then the result is also NaN.

6.11.3 Logical negation operator

The logical negation operator (`not`) is defined for the following kinds of values:

X	Result	Interpretation
`type logical`	`type logical`	Negation
`null`	`null`	

This operator computes the logical not operation on a given logical value. For example:

```
not true               // false
not false              // true
not (true and true)    // false
```

The following holds when evaluating the logical negation operator `not x`:

- Errors raised when evaluating `x` are propagated.
- The value produced from evaluating expression x must be a logical value, or an error with reason code "`Expression.Error`" must be raised. If the value is `true`, the result is `false`. If the operand is `false`, the result is `true`.

The result is a logical value.

6.12 Type operators

The operators `is` and `as` are known as the type operators.

6.12.1 Type compatibility operator

The type compatibility operator `x is y` is defined for the following types of values:

X	Y	Result
`type any`	*nullable-primitive-type*	`type logical`

The expression `x is y` returns `true` if the ascribed type of `x` is compatible with `y`, and returns `false` if the ascribed type of `x` is incompatible with `y`. `y` must be a *nullable-primitive-type*.

is-expression:
    as-expression
    is-expression `is` *nullable-primitive-type*

nullable-primitive-type:
    `nullable`$_{opt}$ *primitive-type*

Type compatibility, as supported by the `is` operator, is a subset of general type compatibility (§5) and is defined using the following rules:

- If `x` is null then it is compatible iff y is a nullable type or the type `any`.
- If `x` is non-null then if it is a compatible if the the primitive type of `x` is the same as `y`.

The following holds when evaluating the expression `x is y`:

- An error raised when evaluating expression `x` is propagated.

6.12.2 Type assertion operator

The type assertion operator `x as y` is defined for the following types of values:

X	Y	Result
`type any`	*nullable-primitive-type*	`type any`

The expression `x as y` asserts that the value `x` is compatible with `y` as per the `is` operator. If it is not compatible, an error is raised. `y` must be a *nullable-primitive-type*.

as-expression:
    equality -expression
    as-expression `as` *nullable-primitive-type*

The expression x as y is evaluated as follows:

- A type compatibility check `x is y` is performed and the assertion returns `x` unchanged if that test succeeds.
- If the compatibility check fails, an error with reason code `"Expression.Error"` is raised.

Examples:

```
1 as number               // 1
"A" as number             // error
null as nullable number   // null
```

The following holds when evaluating the expression `x as y`:

- An error raised when evaluating expression `x` is propagated.

7. Let

7.1 Let expression

A let expression can be used to capture a value from an intermediate calculation in a variable.

let-expression:
    `let` *variable-list* `in` *expression*

variable-list:
    variable
    variable `,` *variable-list*

variable:
    variable-name `=` *expression*

variable-name:
    identifier

The following example shows intermediate results being calculated and stored in variables `x`, `y` and `z` which are then used in a subsequent calculation `x + y + z`:

```
let
    x = 1 + 1,
    y = 2 + 2,
    z = y + 1
in
    x + y + z
```

The result of this expression is:

```
11    // (1 + 1) + (2 + 2) + (2 + 2 + 1)
```

The following holds when evaluating expressions within the *let-expression*:

- The expressions in the variable list define a new scope containing the identifiers from the *variable-list* production and must be present when evaluating the expressions within the *variable-list*productions. Expressions within the *variable-list* may refer to one-another.
- The expressions within the *variable-list* must be evaluated before the expression in the *let-expression* is evaluated.
- Unless the expressions in the *variable-list* are accessed, they must not be evaluated.
- Errors that are raised during the evaluation of the expressions in the *let-expression* are propagated.

A let expression can be seen as syntactic sugar over an implicit record expression. The following expression is equivalent to the example above:

```
[
    x = 1 + 1,
    y = 2 + 2,
    z = y + 1,
    result = x + y + z
][result]
```

8. Conditionals

The *if-expression* selects from two expressions based on the value of a logical input value and evaluates only the selected expression.

if-expression:
`if` *if-condition* `then` *true-expression* `else` *false-expression*

if-condition:
expression

true-expression:
expression

false-expression:
expression

The following are examples of *if-expressions*:

```
if 2 > 1 then 2 else 1              // 2
if 1 = 1 then "yes" else "no"       // "yes"
```

The following holds when evaluating an *if-expression*:

- If the value produced by evaluating the *if-condition* is not a logical value, then an error with reason code "`Expression.Error`" is raised.
- The *true-expression* is only evaluated if the *if-condition* evaluates to the value `true`.
- The *false-expression* is only evaluated if the *if-condition* evaluates to the value `false`.
- The result of the *if-expression* is the value of the *true-expression* if the *if-condition* is `true`, and the value of the *false-expression* is the *if-condition* is `false`.
- Errors raised during the evaluation of the *if-condition*, *true-expression*, or *false-expression* are propagated.

9. Functions

A ***function*** is a value that represents a mapping from a set of argument values to a single value. A function is invoked by provided a set of input values (the argument values), and produces a single output value (the return value).

9.1 Writing functions

Functions are written using a *function-expression*:

function-expression:
    `(` *parameter-list*$_{opt}$ `)` *function-return-type*$_{opt}$ `=>` *function-body*

function-body:
    expression

parameter-list:
    fixed-parameter-list
    fixed-parameter-list `,` *optional-parameter-list*
    optional-parameter-list

fixed-parameter-list:
    parameter
    parameter `,` *fixed-parameter-list*

parameter:
    parameter-name parameter-type$_{opt}$

parameter-name:
    identifier

parameter-type:
    assertion

function-return-type:
    assertion

assertion:
    `as` *nullable-primitve-type*

optional-parameter-list:
    optional-parameter
    optional-parameter `,` *optional-parameter-list*

optional-parameter:
    `optional` *parameter*

nullable-primitve-type
    `nullable`$_{opt}$ *primitive-type*

The following is an example of a function that requires exactly two values `x` and `y`, and produces the result of applying the `+` operator to those values. The `x` and `y` are ***parameters***

that are part of the *formal-parameter-list* of the function, and the `x + y` is the ***function body***:

```
(x, y) => x + y
```

The result of evaluating a *function-expression* is to produce a function value (not to evaluate the *function-body*). As a convention in this document, function values (as opposed to function expressions) are shown with the *formal-parameter-list* but with an ellipsis (`...`) instead of the *function-body*. For example, once the function expression above has been evaluated, it would be shown as the following function value:

```
(x, y) => ...
```

The following operators are defined for function values:

Operator	Result
`x = y`	Equal
`x <> y`	Not equal

The native type of function values is a custom function type (derived from the intrinsic type `function`) that lists the parameter names and specifies all parameter types and the return type to be `any`. (See §5.5 for details on function types.)

9.2 Invoking functions

The *function-body* of a function is executed by ***invoking*** the function value using an *invoke-expression*. Invoking a function value means the *function-body* of the function value is evaluated and a value is returned or an error is raised.

invoke-expression:
    primary-expression `(` *argument-list*$_{opt}$ `)`

argument-list:
    expression-list

Each time a function value is invoked, a set of values are specified as an *argument-list*, called the ***arguments*** to the function.

An *argument-list* is used to specify a fixed number of arguments directly as a list of expressions. The following example defines a record with a function value in a field, and then invokes the function from another field of the record:

```
[
    MyFunction = (x, y, z) => x + y + z,
    Result1 = MyFunction(1, 2, 3),            // 6
    Result2 = MyFunction("a", "b", "c")       // "abc"
]
```

The following holds when invoking a function:

- The environment used to evaluate the *function-body* of the function includes a variable that corresponds to each parameter, with the same name as the parameter. The value of each parameter corresponds to a value constructed from the *argument-list* of the *invoke-expression*, as defined in §9.3.
- All of the expressions corresponding to the function arguments are evaluated before the *function-body* is evaluated.
- Errors raised when evaluating the expressions in the *expression-list* or *function-expression* are propagated.
- The number of arguments constructed from the *argument-list* must be compatible with the formal parameters of the function, or an error is raised with reason code `"Expression.Error"`. The process for determining compatibility is defined in §9.3.

9.3 Parameters

There are two kinds of formal parameters that may be present in a *formal-parameter-list*:

- A ***required*** parameter indicates that an argument corresponding to the parameter must always be specified when a function is invoked. Required parameters must be specified first in the *formal-parameter-list*. The function in the following example defines required parameters `x` and `y`:

```
[
    MyFunction = (x, y) => x + y,
    Result1 = MyFunction(1, 1),         // 2
    Result2 = MyFunction(2, 2)          // 4
]
```

- An ***optional*** parameter indicates that an argument corresponding to the parameter may be specified when a function is invoked, but is not required to be specified. If an argument that corresponds to an optional parameter is not specified when the function is invoked, then the value `null` is used instead. Optional parameters must appear after any required parameters in a *formal-parameter-list*. The function in the following example defines a fixed parameter `x` and an optional parameter `y`:

```
[
    MyFunction = fn(x, optional y) =>
                    if (y = null) x else x + y,
    Result1 = MyFunction(1),            // 1
    Result2 = MyFunction(1, null),      // 1
    Result3 = MyFunction(2, 2),         // 4
]
```

The number of arguments that are specified when a function is invoked must be compatible with the formal parameter list. Compatibility of a set of arguments `A` for a function `F` is computed as follows:

- Let the value *N* represent the number of arguments `A` constructed from the *argument-list*. For example:

```
MyFunction()              // N = 0
MyFunction(1)             // N = 1
MyFunction(null)          // N = 1
MyFunction(null, 2)       // N = 2
MyFunction(1, 2, 3)       // N = 3
MyFunction(1, 2, null)    // N = 3
MyFunction(1, 2, {3, 4})  // N = 3
```

- Let the value *Required* represent the number of fixed parameters of `F` and *Optional* the number of optional parameters of `F`. For example:

```
()                        // Required = 0, Optional = 0
(x)                       // Required = 1, Optional = 0
(optional x)              // Required = 0, Optional = 1
(x, optional y)           // Required = 1, Optional = 1
```

- Arguments `A` are compatible with function `F` if the following are true:
  - (*N* >= *Fixed*) and (*N* <= (*Fixed* + *Optional*))
  - The argument types are compatible with `F`'s corresponding parameter types
- If the function has a delared return type, then the result value of the body of function F is compatible with F's return type if the following is true:
  - The value yielded by evaluating the function body with the supplied arguments for the function parameters has a type that is compatible with the return type.
- If the function body yields a value incompatible with the function's return type, an error with reason code `"Expression.Error"` is raised.

9.4 Recursive functions

In order to write a function value that is recursive, it is necessary to use the scoping operator (`@`) to reference the function within its scope. For example, the following record contains a field that defines the `Factorial` function, and another field that invokes it:

```
[
    Factorial = (x) =>
                if x = 0 then 1 else x * @Factorial(x - 1),
    Result = Factorial(3)    // 6
]
```

Similarly, mutually recursive functions can be written as long as each function that needs to be accessed has a name. In the following example, part of the `Factorial` function has been refactored into a second `Factorial2` function.

```
[
    Factorial = (x) => if x = 0 then 1 else Factorial2(x),
    Factorial2 = (x) => x * Factorial(x – 1),
    Result = Factorial(3)    // 6
]
```

9.5 Closures

A function can return another function as a value. This function can in turn depend on one or more parameters to the original function. In the following example, the function associated with the field `MyFunction` returns a function that returns the parameter specified to it:

```
[
    MyFunction = (x) => () => x,
    MyFunction1 = MyFunction(1),
    MyFunction2 = MyFunction(2),
    Result = MyFunction1() + MyFunction2()    // 3
]
```

Each time the function is invoked, a new function value will be returned that maintains the value of the parameter so that when it is invoked, the parameter value will be returned.

9.6 Functions and environments

In addition to parameters, the *function-body* of a *function-expression* can reference variables that are present in the environment when the function is initialized. For example, the function defined by the field `MyFunction` accesses the field `C` of the enclosing record `A`:

```
[
    A =
    [
        MyFunction = () => C,
        C = 1
    ],
    B = A[MyFunction]()      // 1
]
```

When `MyFunction` is invoked, it accesses the value of the variable `C`, even though it is being invoked from an environment (`B`) that does not contain a variable `C`.

9.7 Simplified declarations

The *each-expression* is a syntactic shortand for declaring untyped functions taking a single formal parameter named _ (underscore).

each-expression:
`each` *each-expression-body*

each-expression-body:
    function-body

Simplified declarations are commonly used to improve the readability of higher-order function invocation.

For example, the following pairs of declarations are semantically equivalent:

```
each _ + 1
(_) => _ + 1

each [A]
(_) => _[A]

Table.SelectRows( aTable, each [Weight] > 12 )
Table.SelectRows( aTable, (_) => _[Weight] > 12 )
```

10. Error Handling

The result of evaluating an M expression produces one of the following outcomes:

- A single value is produced.
- An ***error is raised***, indicating the process of evaluating the expression could not produce a value. An error contains a single record value that can be used to provide additional information about what caused the incomplete evaluation.

Errors can be raised from within an expression, and can be handled from within an expression.

10.1 Raising errors

The syntax for raising an error is as follows:

error-raising-expression:
`error` *expression*

Text values can be used as shorthand for error values. For example:

```
error "Hello, world" // error with message "Hello, world"
```

Full error values are records and can be constructed using the `Error.Record` function:

```
error Error.Record("FileNotFound", "File my.txt not found",
    "my.txt")
```

The above expression is equivalent to:

```
error
[
    Reason = "FileNotFound",
    Message = "File my.txt not found",
    Detail = "my.txt"
]
```

Raising an error will cause the current expression evaluation to stop, and the expression evaluation stack will unwind until one of the following occurs:

- A record field, section member, or let variable – collectively: an ***entry*** – is reached. The entry is marked as having an error, the error value is saved with that entry, and then propagated. Any subsequent access to that entry will cause an identical error to be raised. Other entries of the record, section, or let expression are not necessarily affected (unless they access an entry previously marked as having an error).
- The top-level expression is reached. In this case, the result of evaluating the top-level expression is an error instead of a value.
- A `try` expression is reached. In this case, the error is captured and returned as a value.

10.2 Handling errors

An *error-handling-expression* is used to handle an error:

error-handling-expression:
`try` *protected-expression otherwise-clause*$_{opt}$

protected-expression:
expression

otherwise-clause:
`otherwise` *default-expression*

default-expression:
expression

The following holds when evaluating an *error-handling-expression* without an *otherwise-clause*:

- If the evaluation of the protected-expression does not result in an error and produces a value x, the value produced by the error-handling-expression is a record of the following form:

```
[ HasErrors = false, Value = x ]
```

- If the evaluation of the protected-expression raises an error value e, the result of the error-handling-expression is a record of the following form:

```
[ HasErrors = true, Error = e ]
```

The following holds when evaluating an *error-handling-expression* with an *otherwise-clause*:

- The protected-expression must be evaluated before the otherwise-clause.
- The otherwise-clause must be evaluated if and only if the evaluation of the protected-expression raises an error.
- If the evaluation of the *protected-expression* raises an error, the value produced by the *error-handling-expression* is the result of evaluating the otherwise-clause.
- Errors raised during the evaluation of the otherwise-clause are propagated.

The following example illustrates an *error-handling-expression* in a case where no error is raised:

```
let
    x = try "A"
in
    if x[HasError] then x[Error] else x[Value]
// "A"
```

The following example shows raising an error and then handling it:

```
let
    x = try error "A"
in
    if x[HasError] then x[Error] else x[Value]
// [ Reason = "Expression.Error", Message = "A", Detail = null ]
```

An otherwise clause can be used to replace errors handled by a try expression with an alternate value:

```
try error "A" otherwise 1
// 1
```

If the otherwise clause also raises an error, then so does the entire try expression:

```
try error "A" otherwise error "B"
// error with message "B"
```

10.3 Errors in record and let initializers

The following example shows a record initializer with a field `A` that raises an error and is accessed by two other fields `B` and `C`. Field `B` does not handle the error that is raised by `A`, but `C` does. The final field `D` does not access `A` and so it is not affected by the error in `A`.

```
[
    A = error "A",
    B = A + 1,
    C = let x =
            try A in
                if not x[HasError] then x[Value]
                else x[Error],
    D = 1 + 1
]
```

The result of evaluating the above expression is:

```
[
    A = // error with message "A"
    B = // error with message "A"
    C = "A",
    D = 2
]
```

Error handling in M should be performed close to the cause of errors to deal with the effects of lazy field initialization and deferred closure evaluations. The following example shows an unsuccessful attempt at handling an error using a `try` expression:

```
let
    f = (x) => [ a = error "bad", b = x ],
    g = try f(42) otherwise 123
in
    g[a]     // error "bad"
```

In this example, the definition `g` was meant to handle the error raised when calling `f`. However, the error is raised by a field initializer that only runs when needed and thus after the record was returned from f and passed through the `try` expression.

10.4 Not implemented error

While an expression is being developed, an author may want to leave out the implementation for some parts of the expression, but may still want to be able to execute the expression. One way to handle this case is to raise an error for the unimplemented parts. For example:

```
(x, y) =>
    if x > y then
        x - y
    else
        error Error.Record("Expression.Error",
            "Not Implemented")
```

The ellipsis symbol (`...`) can be used as a shortcut for

```
error Error.Record("Expression.Error", "Not Implemented")
```

not-implemented-expression:

```
...
```

For example, the following is equivalent to the previous example:

```
(x, y) => if x > y then x - y else ...
```

11. Sections

A *section-document* is an M program that consists of multiple named expressions.

section-document:
    section

section:
    `section` *section-name*$_{opt}$ `;` *section-members*$_{opt}$

section-name:
    identifier

section-members:
    section-member
    sedction-members section-member

section-member:
    `shared`$_{opt}$ *section-member-name* `=` *expression* `;`

section-member-name:
    identifier

In M, a section is an organizational concept that allows related expressions to be named and grouped within a document. Each section has a *section-name*, which identifies the section and qualifies the names of the *section-members* declared within the section. A *section-member* consists of a *member-name* and an *expression*. Section member expressions may refer to other section members within the same section directly by member name.

The following example shows a section-document that contains one section:

```
section Section1;
A = 1;                          //1
B = 2;                          //2
C = A + B;                      //3
```

Section member expressions may refer to section members located in other sections by means of a *section-access-expression*, which qualifies a section member name with the name of the containing section.

section-access-expression:
    identifier `!` *identifier*

The following example shows a document containing two sections that are mutually referential:

```
section Section1;
A = “Hello”;                     //”Hello”
B = 1 + Section2!A;              //3

section Section2;
```

```
A = 2;                          //2
B = Section1!A & “ world!”;     //”Hello, world”
```

Section members may optionally be declared as `shared`, which omits the requirement to use a *section-access-expression* when referring to shared members outside of the containing section. Shared members in external sections may be referred to by their unqualified member name so long as no member of the same name is declared in the referring section and no other section has a like-named shared member.

The following example illustrates the behavior of shared members when used across sections within the same document:

```
section Section1;
shared A = 1;          // 1

section Section2;
B = A + 2;             // 3 (refers to shared A from Section1)

section Section3;
A = “Hello”;           // "Hello"
B = A + “ world”;      // "Hello world" (refers to local A)
C = Section1!A + 2;    // 3
```

Defining a shared member with the same name in different sections will produce a valid global environment, however accessing the shared member will raise an error when accessed.

```
section Section1;
shared A = 1;

section Section2;
shared A = “Hello”;

section Section3;
B = A;    //Error: shared member A has multiple definitions
```

The following holds when evaluating a section-document:

- Each *section-name* must be unique in the global environment.
- Within a section, each *section-member* must have a unique *section-member-name*.
- Shared section members with more than one definition raise an error when the shared member is accessed.
- The expression component of a *section-member* must not be evaluated before the section member is accessed.
- Errors raised while the expression component of a *section-member* is evaluated are associated with that section member before propagating outward and then re-raised each time the section member is accessed.

11.1 Document Linking

A set of M section documents can be linked into an opaque record value that has one field per shared member of the section documents. If shared members have ambiguous names, an error is raised.

The resulting record value fully closes over the global environment in which the link process was performed. Such records are, therefore, suitable components to compose M documents from other (linked) sets of M documents. There are no opportunities for naming conflicts.

The standard library functions Embedded.Value can be used to retrieve such "embedded" record values that correspond to reused M components.

11.2 Document Introspection

M provides programmatic access to the global environment by means of the `#sections` and `#shared` keywords.

11.2.1 #sections

The `#sections` intrinsic variable returns all sections within the global environment as a record. This record is keyed by section name and each value is a record representation of the corresponding section indexed by section member name.

The following example shows a document consisting of two sections and the record produced by evaluating the `#sections` intrinsic variable within the context of that document:

```
section Section1;
A = 1;
B = 2;

section Section2;
C = "Hello";
D = "world";

#sections
//[
//  Section1 = [ A = 1, B = 2],
//  Section2 = [ C = "Hello", D = "world" ]
//]
```

The following holds when evaluating `#sections`:

- The `#sections` intrinsic variable preserves the evaluation state of all section member expressions within the document.
- The `#sections` intrinsic variable does not force the evaluation of any unevaluated section members.

11.2.2 #shared

The `#shared` intrinsic variable returns a record containing the names and values of all shared section members currently in scope.

The following example shows a document with two shared members and the corresponding record produced by evaluating the `#shared` intrinsic variable within the context of that document:

```
section Section1;
shared A = 1;
B = 2;

Section Section2;
C = “Hello”;
shared D = “world”;

//[
//  A = 1,
//  D = “world”
//]
```

The following holds when evaluating `#shared`:

- The `#shared` intrinsic variable preserves the evaluation state of all shared member expressions within the document.
- The `#shared` intrinsic variable does not force the evaluation of any unevaluated section members.

12. Consolidated Grammar

12.1 Lexical grammar

lexical-unit:
    lexical-elements$_{opt}$

lexical-elements:
    lexical-element
    lexical-element lexical-elements

lexical-element:
    whitespace
    token
    comment

12.1.1 White space

whitespace:
    Any character with Unicode class Zs
    Horizontal tab character (`U+0009`)
    Vertical tab character (`U+000B`)
    Form feed character (`U+000C`)
    Carriage return character (`U+000D`) followed by line feed character (`U+000A`)
    new-line-character

new-line-character:
    Carriage return character (`U+000D`)
    Line feed character (`U+000A`)
    Next line character (`U+0085`)
    Line separator character (`U+2028`)
    Paragraph separator character (`U+2029`)

12.1.2 Comment

comment:
    single-line-comment
    delimited-comment

single-line-comment:
    `//` *single-line-comment-characters*$_{opt}$

single-line-comment-characters:
    single-line-comment-character
    single-line-comment-characters single-line-comment-character

single-line-comment-character:
    Any Unicode character except a *new-line-character*

delimited-comment:
`/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`

delimited-comment-text:
delimited-comment-section
delimited-comment-text *delimited-comment-section*

delimited-comment-section:
`/`
asterisks$_{opt}$ *not-slash-or-asterisk*

asterisks:
`*`
asterisks `*`

not-slash-or-asterisk:
Any Unicode character except `*` or `/`

12.1.3 Tokens

token:
identifier
keyword
literal
operator-or-punctuator

12.1.4 Character escape sequences

character-escape-sequence:
`#(` *escape-sequence-list* `)`

escape-sequence-list:
single-escape-sequence
escape-sequence-list `,` *single-escape-sequence*

single-escape-sequence:
long-unicode-escape-sequence
short-unicode-escape-sequence
control-character-escape-sequence
escape-escape

long-unicode-escape-sequence:
hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

short-unicode-escape-sequence:
hex-digit hex-digit hex-digit hex-digit

control-character-escape-sequence:
control-character

control-character:

`cr`

`lf`

`tab`

escape-escape:

`#`

12.1.5 Literals

literal:

number-literal

text-literal

null-literal

number-literal:

decimal-number-literal

hexadecimal-number-literal

decimal-digits:

decimal-digit

decimal-digit decimal-digits

decimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`

hexadecimal-number-literal:

`0x` *hex-digits*

`0X` *hex-digits*

hex-digits:

hex-digit

hex-digit hex-digits

hex-digit: one of

`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

decimal-number-literal:

decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$

`.` *decimal-digits* *exponent-part*$_{opt}$

decimal-digits exponent-part

decimal-digits

exponent-part:

`e` *sign*$_{opt}$ *decimal-digits*

`E` *sign*$_{opt}$ *decimal-digits*

sign: one of

`+ -`

text-literal:

`"` *text-literal-characters*$_{opt}$ `"`

text-literal-characters:
text-literal-character
text-literal-character *text-literal-characters*

text-literal-character:
single-text-character
character-escape-sequence
double-quote-escape-sequence

single-text-character:
Any character except `"` (`U+0022`) or `#` (`U+0023`) followed by `(` (`U+0028`)

double-quote-escape-sequence:
`""` (`U+0022`, `U+0022`)

null-literal:
`null`

12.1.6 Identifiers

identifier:
regular-identifier
quoted-identifier

regular-identifier:
available-identifier
available-identifier *dot-character* *regular-identifier*

available-identifier:
A *keyword-or-identifier* that is not a *keyword*

keyword-or-identifier:
letter-character
underscore-character
identifier-start-character *identifier-part-characters*

identifier-start-character:
letter-character
underscore-character

identifier-part-characters:
identifier-part-character
identifier-part-character *identifier-part-characters*

identifier-part-character:
letter-character
decimal-digit-character
underscore-character
connecting-character
combining-character
formatting-character

generalized-identifier:
generalized-identifier-part
generalized-identifier separated only by blanks (`U+0020`) *generalized-identifier-part*

generalized-identifier-part:
generalized-identifier-segment
decimal-digit-character *generalized-identifier-segment*

generalized-identifier-segment:
keyword-or-identifier
keyword-or-identifier *dot-character* *keyword-or-identifier*

dot-character:
`.` (`U+002E`)

underscore-character:
`_` (`U+005F`)

letter-character:
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:
A Unicode character of classes Mn or Mc

decimal-digit-character:
A Unicode character of the class Nd

connecting-character:
A Unicode character of the class Pc

formatting-character:
A Unicode character of the class Cf

quoted-identifier:
`#"` *text-literal-characters*$_{opt}$ `"`

12.1.7 Keywords and predefined identifiers

Predefined identifiers and keywords cannot be redefined. A quoted identifier can be used to handle identifiers that would otherwise collide with predefined identifiers or keywords.

keyword: one of

```
and as each else error false if in is let meta not otherwise or
section shared then true try type #binary #date #datetime
#datetimezone #duration #infinity #nan #sections #shared #table #time
```

12.1.8 Operators and punctuators

operator-or-punctuator: one of

```
, ; = < <= > >= <> + - * / & ( ) [ ] { } @ ? => .. ...
```

12.2 Syntactic grammar

12.2.1 Documents

document:
    section-document
    expression-document

12.2.2 Section Documents

section-document:
    section

section:
    `section` *section-name*$_{opt}$ `;` *section-members*$_{opt}$

section-name:
    identifier

section-members:
    section-member
    section-member section-members

section-member:
    `shared`$_{opt}$ *section-member-name* `=` *expression* `;`

section-member-name:
    identifier

12.2.3 Expression Documents

12.2.3.1 Expressions

expression-document:
    expression

expression:
    logical-expression
    each-expression
    function-expression
    let-expression
    if-expression
    error-raising-expression
    error-handling-expression

12.2.3.2 Logical expressions

logical-or-expression:
    logical-and-expression
    logical-and-expression `or` *logical-or-expression*

logical-and-expression:
is-expression
logical-and-expression `and` *is-expression*

12.2.3.3 Is expression

is-expression:
as-expression
is-expression `is` *nullable-primitive-type*

nullable-primitive-type:
`nullable`$_{opt}$ *primitive-type*

12.2.3.4 As expression

as-expression:
equality-expression
as-expression `as` *nullable-primitive-type*

12.2.3.5 Equality expression

equality-expression:
relational-expression
relational-expression `=` *equality-expression*
relational-expression `<>` *equality-expression*

12.2.3.6 Relational expression

relational-expression:
additive-expression
additive-expression `<` *relational-expression*
additive-expression `>` *relational-expression*
additive-expression `<=` *relational-expression*
additive-expression `>=` *relational-expression*

12.2.3.7 Arithmetic expressions

additive-expression:
multiplicative-expression
multiplicative-expression `+` *additive-expression*
multiplicative-expression `–` *additive-expression*
multiplicative-expression `&` *additive-expression*

multiplicative-expression:
metadata-expression
metadata-expression `*` *multiplicative-expression*
metadata-expression `/` *multiplicative-expression*

12.2.3.8 Metadata expression

metadata-expression:
    as-expression
    as-expression `meta` *unary-expression*

12.2.3.9 Unary expression

unary-expression:
    type-expression
    `+` *unary-expression*
    `-` *unary-expression*
    `not` *unary-expression*

12.2.3.10 Primary expression

primary-expression:
    literal-expression
    list-expression
    record-expression
    identifier-expression
    section-access-expression
    parenthesized-expression
    field-access-expression
    item-access-expression
    invoke-expression
    not-implemented-expression

12.2.3.11 Literal expression

literal-expression:
    literal

12.2.3.12 Identifier expression

identifier-expression:
    identifier-reference

identifier-reference:
    exclusive-identifier-reference
    inclusive-identifier-reference

exclusive-identifier-reference:
    identifier

inclusive-identifier-reference:
    `@` *identifier*

12.2.3.13 Section-access expression

section-access-expression:
    identifier ! *identifier*

12.2.3.14 Parenthesized expression

parenthesized-expression:
    (*expression*)

12.2.3.15 Not-implemented expression

not-implemented-expression:
    ...

12.2.3.16 Invoke expression

invoke-expression:
    primary-expression (*argument-list*$_{opt}$)

argument-list:
    expression
    expression , *argument-list*

12.2.3.17 List expression

list-expression:
    { *item-list*$_{opt}$ }

item-list:
    item
    item , *item-list*

item:
    expression
    expression .. *expression*

12.2.3.18 Record expression

record-expression:
    [*field-list*$_{opt}$]

field-list:
    field
    field , *field-list*

field:
    field-name = *expression*

field-name:
    generalized-identifier

12.2.3.19 Item access expression

item-access-expression:
    item-selection
    optional-item-selection

item-selection:
    primary-expression `{` *item-selector* `}`

optional-item-selection:
    primary-expression `{` *item-selector* `}` `?`

12.2.3.20 Field access expressions

field-access-expression:
    field-selection
    implicit-target-field-selection
    projection
    implicit-target-projection

field-selection:
    primary-expression field-selector

field-selector:
    required-field-selector
    optional-field-selector

required-field-selector:
    `[` *field-name* `]`

optional-field-selector:
    `[` *field-name* `]` `?`

field-name:
    generalized-identifier

implicit-target-field-selection:
    field-selector

projection:
    primary-expression required-projection
    primary-expression optional-projection

required-projection:
    `[` *required-selector-list* `]`

optional- projection:
    `[` *required-selector-list* `]` `?`

required-selector-list:
    required-field-selector
    required-field-selector `,` *required-selector-list*

implicit-target -projection:
    record-projection

12.2.3.21 Function expression

function-expression:
`(` *parameter-list*$_{opt}$ `)` *return-type*$_{opt}$ `=>` *function-body*

function-body:
expression

parameter-list:
fixed-parameter-list
fixed-parameter-list `,` *optional-parameter-list*
optional-parameter-list

fixed-parameter-list:
parameter
parameter `,` *fixed-parameter-list*

parameter:
parameter-name parameter-type$_{opt}$

parameter-name:
identifier

parameter-type:
assertion

return-type:
assertion

assertion:
`as` *type*

optional-parameter-list:
optional-parameter
optional-parameter `,` *optional-parameter-list*

optional-parameter:
`optional` *parameter*

12.2.3.22 Each expression

each-expression:
`each` *each-expression-body*

each-expression-body:
function-body

12.2.3.23 Let expression

let-expression:
`let` *variable-list* `in` *expression*

variable-list:
variable
variable `,` *variable-list*

variable:
variable-name `=` *expression*

variable-name:
identifier

12.2.3.24 If expression

if-expression:
`if` *if-condition* `then` *true-expression* `else` *false-expression*

if-condition:
expression

true-expression:
expression

false-expression:
expression

12.2.3.25 Type expression

type-expression:
primary-expression
`type` *primary-type*

type:
parenthesized-expression
primary-type

primary-type:
primitive-type
record-type
list-type
function-type
table-type
nullable-type

primitive-type: one of
`any anynonnull binary date datetime datetimezone duration function`
`list logical none null number record table text type`

record-type:
`[` *open-record-marker* `]`
`[` *field-specification-list* `]`
`[` *field-specification-list* `,` *open-record-marker* `]`

field-specification-list:
field-specification
field-specification `,` *field-specification-list*

field-specification:
`optional`$_{opt}$ *identifier field-type-specification*$_{opt}$

field-type-specification:
    `=` *field-type*

field-type:
    type

open-record-marker:
    `...`

list-type:
    `{` *item-type* `}`

item-type:
    type

function-type:
    `function (` *parameter-specification-list*$_{opt}$ `)` *return-type*

parameter-specification-list:
    required-parameter-specification-list
    required-parameter-specification-list `,` *optional-parameter-specification-list*
    optional-parameter-specification-list

required-parameter-specification-list:
    required-parameter-specification
    required-parameter-specification `,` *required-parameter-specification-list*

required-parameter-specification:
    parameter-specification

optional-parameter-specification-list:
    optional-parameter-specification
    optional-parameter-specification `,` *optional-parameter-specification-list*

optional-parameter-specification:
    `optional` *parameter-specification*

parameter-specification:
    parameter-name parameter-type

table-type:
    `table` *row-type*

row-type:
    `[` *field-specification-list* `]`

nullable-type:
    `nullable` *type*

12.2.3.26 Error raising expression

error-raising-expression:
    `error` *expression*

12.2.3.27 Error handling expression

error-handling-expression:

`try` *protected-expression otherwise-clause*$_{opt}$

protected-expression:

expression

otherwise-clause:

`otherwise` *default-expression*

default-expression:

expression