The Microsoft code name "M" Modeling Language Specification

Version 0.5

© 2009 Microsoft Corporation. All rights reserved.

This draft specification is provided “as is” and Microsoft disclaims all warranties whatsoever with respect to this specification including specifically for merchantability and fitness for a particular purpose. The posting of this specification does not grant you or your company any license or other rights, either express or implied, or other rights to any intellectual property owned or controlled by Microsoft, its affiliates or any other third party. You are granted no rights to use “Microsoft” or other Microsoft trademarks or trade names in any way or any form hereunder.

[The Microsoft code name "M" Modeling Language Specification 1](#_Toc244322226)

[1 Introduction 7](#_Toc244322227)

[1.1 Expressions 7](#_Toc244322228)

[1.1.1 Collections 8](#_Toc244322229)

[1.1.2 Lists 10](#_Toc244322230)

[1.1.3 Entities 10](#_Toc244322231)

[1.2 Language 11](#_Toc244322232)

[1.2.1 Basics 11](#_Toc244322233)

[1.2.2 Character Processing 13](#_Toc244322234)

[1.2.3 Output 16](#_Toc244322235)

[1.3 Types 18](#_Toc244322236)

[1.3.1 Collection types 21](#_Toc244322237)

[1.3.2 List Types 22](#_Toc244322238)

[1.3.3 Nullable types 22](#_Toc244322239)

[1.3.4 Entity types 23](#_Toc244322240)

[1.3.5 Declaring fields 24](#_Toc244322241)

[1.3.6 Constraints on entity types 25](#_Toc244322242)

[1.4 Queries 26](#_Toc244322243)

[1.4.1 Filtering 26](#_Toc244322244)

[1.4.2 Selection 27](#_Toc244322245)

[1.5 Modules 27](#_Toc244322246)

[2 Lexical Structure 28](#_Toc244322247)

[2.1 Pre-Processing 29](#_Toc244322248)

[2.1.1 Directives 30](#_Toc244322249)

[2.2 Lexical Analysis 30](#_Toc244322250)

[2.2.1 Whitespace 30](#_Toc244322251)

[2.2.2 Comments 30](#_Toc244322252)

[2.3 Identifiers 31](#_Toc244322253)

[2.4 Keywords 32](#_Toc244322254)

[2.5 Operators and punctuators 33](#_Toc244322255)

[2.6 Literals 33](#_Toc244322256)

[2.6.1 Date literals 33](#_Toc244322257)

[2.6.2 DateTime literals 34](#_Toc244322258)

[2.6.3 DateTimeOffset literals 34](#_Toc244322259)

[2.6.4 Decimal literals 34](#_Toc244322260)

[2.6.5 Guid literals 35](#_Toc244322261)

[2.6.6 Integer literals 35](#_Toc244322262)

[2.6.7 Logical literals 35](#_Toc244322263)

[2.6.8 Null literal 36](#_Toc244322264)

[2.6.9 Scientific literals 36](#_Toc244322265)

[2.6.10 Text literals 36](#_Toc244322266)

[2.6.11 Time literals 38](#_Toc244322267)

[3 Text Pattern Expressions 40](#_Toc244322268)

[3.1 Primary Expressions 40](#_Toc244322269)

[3.1.1 Character Class 40](#_Toc244322270)

[3.1.2 References 40](#_Toc244322271)

[3.1.3 Repetition operators 41](#_Toc244322272)

[3.1.4 Inline Rules 41](#_Toc244322273)

[3.1.5 Any 42](#_Toc244322274)

[3.2 Term Operators 42](#_Toc244322275)

[4 Productions 43](#_Toc244322276)

[4.1 Pattern Declaration 43](#_Toc244322277)

[4.2 Term Declaration 43](#_Toc244322278)

[4.2.1 Error 44](#_Toc244322279)

[4.3 Constructors 45](#_Toc244322280)

[4.3.1 Constructor operators 47](#_Toc244322281)

[4.3.2 Default Constructor 48](#_Toc244322282)

[4.4 Precedence 49](#_Toc244322283)

[4.4.1 Production Precedence 49](#_Toc244322284)

[4.4.2 Term Precedence 50](#_Toc244322285)

[5 Rules 52](#_Toc244322286)

[5.1 Token Rules 52](#_Toc244322287)

[5.1.1 Final Modifier 52](#_Toc244322288)

[5.2 Syntax Rules 53](#_Toc244322289)

[5.3 Interleave Rules 53](#_Toc244322290)

[5.4 Inline Rules 53](#_Toc244322291)

[5.5 Rule Parameters 54](#_Toc244322292)

[6 Languages 55](#_Toc244322293)

[6.1 Main Rule 55](#_Toc244322294)

[6.2 Cross-language rule references 56](#_Toc244322295)

[7 Types 57](#_Toc244322296)

[7.1 Type declaration 57](#_Toc244322297)

[7.2 Subtyping 59](#_Toc244322298)

[7.3 Operators 59](#_Toc244322299)

[7.4 Intrinsic Types 60](#_Toc244322300)

[7.4.1 Any 61](#_Toc244322301)

[7.4.2 General 61](#_Toc244322302)

[7.4.2.1 Members 61](#_Toc244322303)

[7.4.3 Number 61](#_Toc244322304)

[7.4.3.1 Operators 62](#_Toc244322305)

[7.4.3.2 AutoNumber 63](#_Toc244322306)

[7.4.4 Text 63](#_Toc244322307)

[7.4.4.1 Members 63](#_Toc244322308)

[7.4.4.2 Length declaration 64](#_Toc244322309)

[7.4.5 Logical 64](#_Toc244322310)

[7.4.6 Guid 65](#_Toc244322311)

[7.4.7 Date 65](#_Toc244322312)

[7.4.8 DateTime 65](#_Toc244322313)

[7.4.9 DateTimeOffset 65](#_Toc244322314)

[7.4.10 Time 65](#_Toc244322315)

[7.5 Entity 66](#_Toc244322316)

[7.5.1 Declaration 66](#_Toc244322317)

[7.5.2 Identity 66](#_Toc244322318)

[7.5.3 Operators 67](#_Toc244322319)

[7.5.4 Ascription 67](#_Toc244322320)

[7.6 Collections 68](#_Toc244322321)

[7.6.1 Declaration 68](#_Toc244322322)

[7.6.2 Operators 68](#_Toc244322323)

[7.6.3 Members 69](#_Toc244322324)

[7.6.4 Indexers 70](#_Toc244322325)

[7.6.4.1 Selectors 70](#_Toc244322326)

[7.6.4.2 Projectors 71](#_Toc244322327)

[7.7 Lists 72](#_Toc244322328)

[7.7.1 Declaration 72](#_Toc244322329)

[7.7.2 Operators 73](#_Toc244322330)

[7.8 Null 73](#_Toc244322331)

[8 Computed and Stored Values 75](#_Toc244322332)

[8.1 Computed Value Declaration 75](#_Toc244322333)

[8.1.1 Overloading 75](#_Toc244322334)

[8.2 Fields 76](#_Toc244322335)

[8.3 External Declarations 76](#_Toc244322336)

[9 Expressions 78](#_Toc244322337)

[9.1 Operators 78](#_Toc244322338)

[9.1.1 Operator precedence and associativity 78](#_Toc244322339)

[9.2 Member access 79](#_Toc244322340)

[9.2.1 Symbol lookup 80](#_Toc244322341)

[9.3 Initializers 80](#_Toc244322342)

[9.3.1 Collection Initializer 81](#_Toc244322343)

[9.3.2 List Initializer 81](#_Toc244322344)

[9.3.3 Record Initializer 81](#_Toc244322345)

[9.3.4 Kind Pattern 81](#_Toc244322346)

[9.3.5 Non Local Initialization 82](#_Toc244322347)

[9.4 Invocation Expression 83](#_Toc244322348)

[9.5 Primary expressions 83](#_Toc244322349)

[9.5.1 Simple names 84](#_Toc244322350)

[9.5.2 Parenthesized expressions 84](#_Toc244322351)

[9.5.3 Context variable 84](#_Toc244322352)

[9.5.4 About operator 84](#_Toc244322353)

[9.6 Unary operators 85](#_Toc244322354)

[9.7 Multiplicity 86](#_Toc244322355)

[9.8 Arithmetic operators 86](#_Toc244322356)

[9.9 Relational and type-testing operators 87](#_Toc244322357)

[9.10 Equality operators 87](#_Toc244322358)

[9.11 Logical operators 87](#_Toc244322359)

[9.12 Conditional operators 87](#_Toc244322360)

[9.12.1 Coalescing operator 87](#_Toc244322361)

[9.12.2 Conditional operator 88](#_Toc244322362)

[9.13 Query expressions 88](#_Toc244322363)

[9.13.1 From clause 89](#_Toc244322364)

[9.13.2 Where clause 89](#_Toc244322365)

[9.13.3 Select clause 90](#_Toc244322366)

[9.13.4 Join clause 90](#_Toc244322367)

[9.13.5 Let clause 91](#_Toc244322368)

[9.13.6 Group By clause 91](#_Toc244322369)

[9.13.7 Accumulate clause 92](#_Toc244322370)

[9.14 Compact query expressions 92](#_Toc244322371)

[9.14.1 Where operator 92](#_Toc244322372)

[9.14.2 Select operator 93](#_Toc244322373)

[9.15 Binary and Collection operators 94](#_Toc244322374)

[9.16 Expressions 94](#_Toc244322375)

[10 Module 95](#_Toc244322376)

[10.1 Compilation Unit 95](#_Toc244322377)

[10.2 Module Declaration 95](#_Toc244322378)

[10.3 Inter-Module Dependencies 96](#_Toc244322379)

[10.4 Compilation Episode 100](#_Toc244322380)

[10.5 Storage 100](#_Toc244322381)

[11 Attributes 102](#_Toc244322382)

[11.1 Case Insensitive 102](#_Toc244322383)

[11.2 Classification 102](#_Toc244322384)

[11.3 Format Scope 103](#_Toc244322385)

[11.4 Nest 103](#_Toc244322386)

[12 Catalog 104](#_Toc244322387)

[13 Standard Library 108](#_Toc244322388)

[13.1 Data types 108](#_Toc244322389)

[13.1.1 Binary 108](#_Toc244322390)

[14 Glossary 109](#_Toc244322391)

#  Introduction

The Microsoft code name "M" Modeling Language, hereinafter referred to as M, is a language for modeling domains using text. A domain is any collection of related concepts or objects. Modeling domain consists of selecting certain characteristics to include in the model and implicitly excluding others deemed irrelevant. Modeling using text has some advantages and disadvantages over modeling using other media such as diagrams or clay. A goal of the M language is to exploit these advantages and mitigate the disadvantages.

A key advantage of modeling in text is ease with which both computers and humans can store and process text. Text is often the most natural way to represent information for presentation and editing by people. However, the ability to extract that information for use by software has been an arcane art practiced only by the most advanced developers. The language feature of M enables information to be represented in a textual form that is tuned for both the problem domain and the target audience. The M language provides simple constructs for describing the shape of a textual language – that shape includes the input syntax as well as the structure and contents of the underlying information. To that end, M acts as both a schema language that can validate that textual input conforms to a given language as well as a transformation language that projects textual input into data structures that are amenable to further processing or storage.

M builds on 4 basic concepts:

1. Language – a collection of rules that recognize free text and produce a structured representation of relevant concepts in the text.
2. Data – a sparse textual representation of information amenable to automated storage, transformation and communication.
3. Constraint – a rule that recognizes specific structure and relationships within data.
4. Transformation – a mapping between source data and result data.

## Expressions

The easiest way to get started with M is to look at some values. M has intrinsic support for constructing values. The following is a legal value in M:

"Hello, world"

The quotation marks tell M that this is the text value Hello, world. M literals can also be numbers. The following literal:

1

is the numeric value one. Finally, there are two literals that represent logical values:

true

false

We’ve just seen examples of using literals to write down textual, numeric, and logical values. We can also use expressions to write down values that are computed.

An M expression applies an *operator* to zero or more *operands* to produce a *result*. An operator is either a built-in operator (e.g., +) or a user-defined function (which we’ll look at in Section **Error! Reference source not found.**). An *operand* is a value that is used by the operator to calculate the *result* of the expression, which is itself a value. Expressions nest, so the operands themselves can be expressions.

We'll write the result of evaluating an expression as *expression* ⇓ *result*. The down arrow is not part of the M language. It is used for documentation purposes only. Here are some examples:

1 ⇓ 1 // A literal evaluates to itself

1 + 1 ⇓ 2 // Math works as usual

1 + "a" ⇓ *error* // Some expressions results in error

M defines two equality operators: equals, ==, and not equals, !=, both of which result in either true or false based on the equivalence/nonequivalence of the two operands. Here are some expressions that use the equality operators:

1 == 1 ⇓ true

"Hello" == "hELLO" ⇓ false

true != false ⇓ true

M defines the standard four relational operators less-than <, greater-than >, less-than-or-equal <=, and greater-than-or-equal >=, which work over numeric and textual values. M also defines the standard three logical operators: and &&, or ||, and not ! that combine logical values.

The following expressions show these operators in action:

1 < 4 ⇓ true

1 == 1 ⇓ true

1 > 4 ⇓ false

1 + 1 == 3 ⇓ false

(1 + 1 == 3) || (2 + 2 < 10) ⇓ true

(1 + 1 == 2) && (2 + 2 < 10) ⇓ true

### Collections

An M *collection* is a value that groups together zero or more *elements* which themselves are values. We can write down a collection using { }:

{ 1, 2 }

{ 1 }

{ }

As with simple values, the equivalence operators == and != are defined over collections. In M, two collections are considered equivalent if and only if each element has a distinct equivalent element in the other collection. That allows us to write the following equivalence expressions:

{ 1, 2 } == { 1, 2 }

{ 1, 2 } != { 1 }

both of which are true.

The elements of a collection can consist of different kinds of values:

{ true, "Hello" }

and these values can be the result of arbitrary calculation:

{ 1 + 2, 99 – 3, 4 < 9 }

which is equivalent to the collection:

{ 3, 96, true }

The order of elements in a collection is not significant.

{ 1, 2 } == { 2, 1 } ⇓ true

Finally, collections can contain duplicate elements, which are significant.

{ 1, 2, 2 } == { 1, 2 } ⇓ false

M defines a set of built-in operators that are specific to collections. The in operator tests whether a given value is an element of the collection. The result of the in operator is a logical value that indicates whether the value is or is not an element of the collection.

1 in { 1, 2, 3 } ⇓ true

1 in { "Hello", 9 } ⇓ false

M defines a Count member on collections that calculates the number of elements in a collection.

{ 1, 2, 2, 3 }.Count ⇓ 4

The postfix # operator also returns the count of a collection

{ 1, 2, 2, 3 }# ⇓ 4

As noted earlier, M collections may contain duplicates. You can apply the Distinct member to get a version of the collection with any duplicates removed:

{ 1, 2, 3, 1 }.Distinct ⇓ { 1, 2, 3 }

The result of Distinct is not just a collection but is also a set, i.e. a collection of distinct elements.

M also defines set union "|" and set intersection "&" operators, which also yield sets:

({ 1, 2, 3, 1 } | { 1, 2, 4 }) == { 1, 2, 3, 4 }

({ 1, 2, 3, 1 } & { 1, 2, 4 }) == { 1, 2 }

Note that union and intersection always return collections that are sets, even when applied to collections that contain duplicates.

M defines the subset and superset using <= and >=. Again these operations convert collections to sets. The following expressions evaluate to true.

{ 1, 2 } <= { 1, 2, 3 } ⇓ true

{ "Hello", "World" } >= { "World" } ⇓ true

{ 1, 2, 1 } <= { 1, 2, 3 } ⇓ true

The where operator applies a logical expression (called the *predicate*) to each element in a collection (called the *domain*) and results in a new collection that consists of only the elements for which the predicate holds true. To allow the element to be used in the predicate, the where operator introduces the symbol value to stand in for the specific element being tested.

For example, consider this expression that uses a where operator:

{ 1, 2, 3, 4, 5, 6 } where value > 3 ⇓ { 4, 5, 6 }

In this example, the domain is the collection { 1, 2, 3, 4, 5, 6 } and the predicate is the expression value > 3. Note that the identifier value is available only within the scope of the predicate expression. The result of this expression is the collection { 4, 5, 6 }.

While the where operator allows elements to be accessed based on a calculation over the values of each element. There are situations where it would be much more convenient to simply assign names to each element and then access the element values by its assigned name. M defines a distinct kind of value called an *entity* for just this purpose.

### Lists

Basic collections are unordered. Ordered collections are represented by lists. The following values are lists:

[1, 2, 3]

["hello", 1, true]

[]

Elements in a list have a distinct position. Two lists with the same elements but in different positions are not equal:

[1,2,3] == [3,2,1] ⇓ false

Ordered collections are collections so members and operators defined on collections work on lists as well. Where possible the order of the collection is maintained:

[1, 2, 3].Count ⇓ 3

[1, 2, 3, 4, 5, 6, 7, 8, 9] where value > 5 ⇓ [6, 7, 8, 9]

[1, 2, 3] select value \* value ⇓ [1, 4, 9]

### Entities

An *entity* consists of zero or more name-value pairs called *fields*. Here’s a simple entity value:

{ X => 100, Y => 200 }

This entity has two fields: one named X with the value of 100, the other named Y with the value of 200.

Entity initializers can use arbitrary expressions as field values:

{ X => 50 + 50, Y => 300 - 100 }

And the names of members can be arbitrary Unicode text:

{ @[Horizontal Coordinate] => 100, @[Vertical Coordinate] => 200 }

If the member name matches the Identifer pattern, it can be written without the surrounding ' @[]'. An identifier must begin with an upper or lowercase letter or "\_" and be followed by a sequence of letters, digits, "\_", and "$".

Here are a few examples:

HelloWorld => 1 // matches the Identifier pattern

'Hello World' => 1 // doesn’t match identifier pattern – escape it

\_HelloWorld => 1 // matches the Identifier pattern

A => 1 // matches the Identifier pattern

'1' => 1 // doesn’t match identifier pattern – escape it

It is always legal to use '@[] ' to escape symbolic names; however, most of the examples in this document use names that don’t require escaping and therefore do not use escaping syntax for readability.

M imposes no limitations on the values of entity members. It is legal for the value of an entity member to refer to another entity:

{ TopLeft { X => 100, Y => 200 }, BottomRight { X => 400, Y => 100 } }

or a collection:

{ LotteryPicks { 1, 18, 25, 32, 55, 61 }, Odds => 0.00000001 }

or a collection of entities:

{

 Color => "Red",

 Path {

 { X => 100, Y => 100 },

 { X => 200, Y => 200 },

 { X => 300, Y => 100 },

 { X => 300, Y => 100 },

 }

}

This last example illustrates that entity values are legal for use as elements in collections.

Entity initializers are useful for constructing new entity values. M defines the dot, ".", operator over entities for accessing the value of a given member. For example, this expression:

{ X => 100, Y => 200 }.X

yields the value of the X member, which in this case is 100. The result of the dot operator is just a value that is subject to subsequent operations. For example, this expression:

{ Center { X => 100, Y => 200 }, Radius => 3 }.Center.Y

yields the value 200.

## Language

### Basics

A M language definition consists of one or more named rules, each of which describe some part of the language. The following fragment is a simple language definition:

language HelloLanguage {

 syntax Main = "Hello, World";

}

The language being specified is named HelloLanguage and it is described by one rule named Main. A language may contain more than one rule; the name Main is used to designate the initial rule that all input documents must match in order to be considered valid with respect to the language.

Rules use patterns to describe the set of input values that the rule applies to. The Main rule above has only one pattern, "Hello, World" that describes exactly one legal input value:

Hello, World

If that input is fed to the M processor for this language, the processor will report that the input is valid. Any other input will cause the processor to report the input as invalid.

Typically, a rule will use multiple patterns to describe alternative input formats that are logically related. For example, consider this language:

language PrimaryColors {

 syntax Main = "Red" | "Green" | "Blue";

}

The Main rule has three patterns – input must conform to one of these patterns in order for the rule to apply. That means that the following is valid:

Red

as well as this:

Green

and this:

Blue

No other input values are valid in this language.

Most patterns in the wild are more expressive than those we’ve seen so far – most patterns combine multiple terms. Every pattern consists of a sequence of one or more grammar terms, each of which describes a set of legal text values. Pattern matching has the effect of consuming the input as it sequentially matches the terms in the pattern. Each term in the pattern consumes zero or more initial characters of input – the remainder of the input is then matched against the next term in the pattern. If all of the terms in a pattern cannot be matched the consumption is “undone” and the original input will used as a candidate for matching against other patterns within the rule.

A pattern term can either specify a literal value (like in our first example) or the name of another rule. The following language definition matches the same input as the first example:

language HelloLanguage2 {

 syntax Main = Prefix ", " Suffix;

 syntax Prefix = "Hello";

 syntax Suffix = "World";

}

Like functions in a traditional programming language, rules can be declared to accept parameters. A parameterized rule declares one or more “holes” that must be specified to use the rule. The following is a parameterized rule:

syntax Greeting(salutation, separator) = salutation separator "World";

To use a parameterized rule, one simply provides actual rules as arguments to be substituted for the declared parameters:

syntax Main = Greeting(Prefix, ", ");

A given rule name may be declared multiple times provided each declaration has a different number of parameters. That is, the following is legal:

syntax Greeting(salutation, sep, subject) = salutation sep subject;

syntax Greeting(salutation, sep) = salutation sep "World";

syntax Greeting(sep) = "Hello" sep "World";

syntax Greeting = "Hello" ", " "World";

The selection of which rule is used is determined based on the number of arguments present in the usage of the rule.

A pattern may indicate that a given term may match repeatedly using the standard Kleene operators (e.g., ?, \*, and +). For example, consider this language:

language HelloLanguage3 {

 syntax Main = Prefix ", "? Suffix\*;

 syntax Prefix = "Hello";

 syntax Suffix = "World";

}

This language considers the following all to be valid:

Hello

Hello,

Hello, World

Hello, WorldWorld

HelloWorldWorldWorld

Terms can be grouped using parentheses to indicate that a group of terms must be repeated:

language HelloLanguage3 {

 syntax Main = Prefix (", " Suffix)+;

 syntax Prefix = "Hello";

 syntax Suffix = "World";

}

which considers the following to all be valid input:

Hello, World

Hello, World, World

Hello, World, World, World

The use of the + operator indicates that the group of terms must match at least once.

### Character Processing

In the previous examples of the HelloLanguage, the pattern term for the comma separator included a trailing space. That trailing space was significant, as it allowed the input text to include a space after the comma:

Hello, World

More importantly, the pattern indicates that the space is not only allowed, but is required. That is, the following input is not valid:

Hello,World

Moreover, exactly one space is required, making this input invalid as well:

Hello, World

To allow any number of spaces to appear either before or after the comma, we could have written the rule like this:

syntax Main = "Hello" " "\* "," " "\* "World";

While this is correct, in practice most languages have many places where secondary text such as whitespace or comments can be interleaved with constructs that are primary in the language. To simplify specifying such languages, a language may specify one or more named interleave patterns.

An interleave pattern specifies text streams that are not considered part of the primary flow of text. When processing input, the M processor implicitly injects interleave patterns between the terms in all syntax patterns. For example, consider this language:

language HelloLanguage {

 syntax Main = "Hello" "," "World";

 interleave Secondary = " "+;

}

This language now accepts any number of whitespace characters before or after the comma. That is,

Hello,World

Hello, World

Hello , World

are all valid with respect to this language.

Interleave patterns simplify defining languages that have secondary text like whitespace and comments. However, many languages have constructs in which such interleaving needs to be suppressed. To specify that a given rule is not subject to interleave processing, the rule is written as a token rule rather than a syntax rule.

Token rules identify the lowest level textual constructs in a language – by analogy token rules identify words and syntax rules identify sentences. Like syntax rules, token rules use patterns to identify sets of input values. Here’s a simple token rule:

token BinaryValueToken = ("0" | "1")+;

It identifies sequences of 0 and 1 characters much like this similar syntax rule:

syntax BinaryValueSyntax = ("0" | "1")+;

The main distinction between the two rules is that interleave patterns do not apply to token rules. That means that if the following interleave rule was in effect:

interleave IgnorableText = " "+;

then the following input value:

0 1011 1011

would be valid with respect to the BinaryValueSyntax rule but not with respect to the BinaryValueToken rule, as interleave patterns do not apply to token rules.

M provides a shorthand notation for expressing alternatives that consist of a range of Unicode characters. For example, the following rule:

token AtoF = "A" | "B" | "C" | "D" | "E" | "F";

can be rewritten using the range operator as follows:

token AtoF = "A".."F";

Ranges and alternation can compose to specify multiple non-contiguous ranges:

token AtoGnoD = "A".."C" | "E".."G";

which is equivalent to this longhand form:

token AtoGnoD = "A" | "B" | "C" | "E" | "F" | "G";

Note that the range operator only works with text literals that are exactly one character in length.

The patterns in token rules have a few additional features that are not valid in syntax rules. Specifically, token patterns can be negated to match anything not included in the set, by using the difference operator (-). The following example combines difference with any. Any matches any single character. The expression below matches any character that is not a vowel:

any - ("A"|"E"|"I"|"O"|"U")

Token rules are named and may be referred to by other rules:

token AorBorCorEorForG = (AorBorC | EorForG)+;

token AorBorC = "A".."C";

token EorForG = "E".."G";

Because token rules are processed before syntax rules, token rules cannot refer to syntax rules:

syntax X = "Hello";

token HelloGoodbye = X | "Goodbye"; // illegal

However, syntax rules may refer to token rules:

token X = "Hello";

syntax HelloGoodbye = X | "Goodbye"; // legal

The M processor treats all literals in syntax patterns as anonymous token rules. That means that the previous example is equivalent to the following:

token X = "Hello";

token temp = "Goodbye";

syntax HelloGoodbye = X | temp;

Operationally, the difference between token rules and syntax rules is when they are processed. Token rules are processed first against the raw character stream to produce a sequence of named tokens. The M processor then processes the language’s syntax rules against the token stream to determine whether the input is valid and optionally to produce structured data as output. The next section describes how that output is formed.

### Output

M processing transforms text into structured data. The shape and content of that data is determined by the syntax rules of the language being processed. Each syntax rule consists of a set of productions, each of which consists of a pattern and an optional projection. Patterns were discussed in the previous sections and describe a set of legal character sequences that are valid input. Projections describe how the information represented by that input should be produced.

Each production is like a function from text to structured data. The primary way to write projections is to use a simple construction syntax that produces graph-structured data suitable for programs and stores. For example, consider this rule:

syntax Rock =

 "Rock" => Item { Heavy { true }, Solid { true } } ;

This rule has one production that has a pattern that matches "Rock" and a projection that produces the following value (using a notation known as M graphs):

Item {

 Heavy { true },

 Solid { true }

}

Rules can contain more than one production in order to allow different input to produce very different output. Here’s an example of a rule that contains three productions with very different projections:

syntax Contents

 = "Rock" => Item { Heavy { true }, Solid { true } }

 | "Water" => Item { Consumable { true }, Solid { false } }

 | "Hamster" => Pet { Small { true }, Legs { 4 } } ;

When a rule with more than one production is processed, the input text is tested against all of the productions in the rule to determine whether the rule applies. If the input text matches the pattern from exactly one of the rule’s productions, then the corresponding projection is used to produce the result. In this example, when presented with the input text "Hamster", the rule would yield:

Pet {

 Small { true },

 Legs { 4 }

}

as a result.

To allow a syntax rule to match no matter what input it is presented with, a syntax rule may specify a production that uses the empty pattern, which will be selected if and only if none of the other productions in the rule match:

syntax Contents

 = "Rock" => Item { Heavy { true }, Solid { true } }

 | "Water" => Item { Consumable { true }, Solid { false } }

 | "Hamster" => Pet { Small { true }, Legs { 4 } }

 | empty => NoContent { } ;

When the production with the empty pattern is chosen, no input is consumed as part of the match.

To allow projections to use the input text that was used during pattern matching, pattern terms associate a variable name with individual pattern terms by prefixing the pattern with an identifier separated by a colon. These variable names are then made available to the projection. For example, consider this language:

language GradientLang {

 syntax Main

 = from:Color ", " to:Color => Gradient { Start { from }, End { to } } ;

 token Color

 = "Red" | "Green" | "Blue";

}

Given this input value:

Red, Blue

The M processor would produce this output:

Gradient {

 Start { "Red" },

 End { "Blue" }

}

Like all projection expressions we’ve looked at, literal values may appear in the output graph. The set of literal types supported by M and a couple examples follow:

* Text literals – "ABC", "\u[Smile]"
* Integer literals – 25, -34
* Real literals – 0.0, -5.0E15
* Logical literals – true, false
* Null literal – null

The projections we’ve seen so far all attach a label to each graph node in the output (e.g., Gradient, Start, etc.). The label is optional and can be omitted:

syntax Naked = t1:First t2:Second => { t1, t2 };

The label can be an arbitrary string – to allow labels to be escaped, one uses the id operator:

syntax Fancy = t1:First t2:Second => id("Label with Spaces!"){ t1, t2 };

The id operator works with either literal strings or with variables that are bound to input text:

syntax Fancy = name:Name t1:First t2:Second => id(name){ t1, t2 };

Using id with variables allows the labeling of the output data to be driven dynamically from input text rather than statically defined in the language. This example works when the variable name is bound to a literal value. If the variable was bound to a structured node that was returned by another rule, that node’s label can be accessed using the labelof operator:

syntax Fancier = p:Point => id(labelof(p)) { 1, 2, 3 };

The labelof operator returns a string that can be used both in the id operator as well as a node value.

The projection expressions shown so far have no notion of order. That is, this projection expression:

A { X { 100 }, Y { 200 } }

is semantically equivalent to this:

A { Y { 200 }, X { 100 } }

and implementations of M are not required to preserve the order specified by the projection. To indicate that order is significant and must be preserved, brackets are used rather than braces. This means that this projection expression:

A [ X { 100 }, Y { 200 } ]

is *not* semantically equivalent to this:

A [ Y { 200 }, X { 100 } ]

The use of brackets is common when the sequential nature of information is important and positional access is desired in downstream processing.

Sometimes it is useful to splice the nodes of a value together into a single collection. The valuesof operator will return the values of a node (labeled or unlabeled) as top-level values that are then combinable with other values as values of new node.

syntax ListOfA

 = a:A => [a]

 | list:ListOfA "," a:A => [ valuesof(list), a ];

Here, valuesof(list) returns the all the values of the list node, combinable with a to form a new list.

Productions that do not specify a projection get the default projection.

For example, consider this simple language that does not specify projections:

language GradientLanguage {

 syntax Main = Gradient | Color;

 syntax Gradient = from:Color " on " to:Color;

 token Color = "Red" | "Green" | "Blue";

}

When presented with the input "Blue on Green” the language processor returns the following output:

Main[ Gradient [ "Blue", " on ", "Green" ] ] ]

These default semantics allows grammars to be authored rapidly while still yielding understandable output. However, in practice explicit projection expressions provide language designers complete control over the shape and contents of the output.

## Types

Expressions give us a great way to write down how to *calculate* values based on other values. Often, we want to write down how to *categorize* values for the purposes of validation or allocation. In M, we categorize values using types.

An M type describes a collection of acceptable or *conformant* values. We use types to constrain which values may appear in a particular context (*e.g.*, an operand, a storage location).

With a few notable exceptions, M allows types to be used as collections. For example, we can use the in operator to test whether a value conforms to a given type. The following expressions are true:

1 in Number

"Hello, world" in Text

Note that the names of the built-in types are available directly in the M language. We can introduce new names for types using type declarations. For example, this type declaration introduces the type name as a synonym for the Text simple type:

type @[My Text] : Text;

With this type name now available, we can write the following:

"Hello, world" in @[My Text]

Note that the name of the type @[My Text] contains a space and is subject to the same escaping rules as the member names in entities.

While it is moderately useful to introduce your own names for an existing type, it’s far more useful to apply a predicate to the underlying type:

type SmallText : Text where value.Count < 7;

In this example, we’ve constrained the universe of possible Text values to those in which the value contains less than seven characters. That means that the following holds true:

"Terse" in SmallText

!("Verbose" in SmallText)

Type declarations compose:

type TinyText : SmallText where value.Count < 6;

The preceding is equivalent to the following:

type TinyText : Text where value.Count < 6;

It’s important to note that the name of the type exists so an M declaration or expression can refer to it. We can assign any number of names to the same type (e.g., Text where value.Count < 7) and a given value either conforms to all of them or to none of them. For example, consider this example:

type A : Number where value < 100;

type B : Number where value < 100;

Given these two type definitions, both of the following expressions will evaluate to true:

1 in A

1 in B

If we introduce the following third type:

type C : Number where value > 0;

we can also state this:

1 in C

In M types are sets of values and it is possible to define a new type by explicitly enumerating those values.

type PrimaryColors { "Red", "Blue", "Yellow" }

This is how an enumeration is defined in M. Any type in M is a collection of values. For example the types Logical and Integer8 defined below could be defined as the collections:

{ true, false }

{-128, -127, ..., -1, 0, 1, ..., 127}

A general principle of M is that a given value may conform to any number of types. This is a departure from the way many object-based systems work, in which a value is bound to a specific type at initialization-time and is a member of the finite set of supertypes that were specified when the type was defined.

One last type related operation bears discussion –the type ascription operator ":". The type ascription operator asserts that a given value conforms to a specific type.

In general, when we see values in expressions, M has some notion of the expected type of that value based on the declared result type for the operator or function being applied. For example, the result of the logical and operator "&&" is declared to be conformant with type Logical.

It is occasionally useful (or even required) to apply additional constraints to a given value – typically to use that value in another context that has differing requirements.

For example, consider the following simple type definition:

type SuperPositive : Number where value > 5;

And let’s now assume that there’s a function named CalcIt that is declared to accept a value of type SuperPositive as an operand. We’d like M to allow expressions like this:

CalcIt(20)

CalcIt(42 + 99)

and prohibit expressions like this:

CalcIt(-1)

CalcIt(4)

In fact, M does exactly what we want for these four examples. This is because these expressions express their operands in terms of simple built-in operators over constants. All of the information needed to determine the validity of the expressions is readily and cheaply available the moment the M source text for the expression is encountered.

However, if the expression draws upon dynamic sources of data or user-defined functions, we must use the type ascription operator to assert that a value will conform to a given type.

To understand how the type ascription operator works with values, let’s assume that there is a second function, GetVowelCount, that is declared to accept an operand of type Text and return a value of type Number that indicates the number of vowels in the operand.

Since we can’t know based on the declaration of GetVowelCount whether its results will be greater than five or not, the following expression is not a legal M expression:

CalcIt( GetVowelCount(someTextVariable) )

Because GetVowelCount’s declared result type Number includes values that do not conform to the declared operand type of CalcIt which is SuperPositive, M assumes that this expression was written in error and will refuse to even attempt to evaluate the expression.

When we rewrite this expression to the following legal expression using the type ascription operator:

CalcIt( GetVowelCount(someTextVariable) : SuperPositive )

we are telling M that we have enough understanding of the GetVowelCount function to know that we’ll always get a value that conforms to the type SuperPositive. In short, we’re telling M we know what we’re doing.

But what if we don’t? What if we misjudged how the GetVowelCount function works and a particular evaluation results in a negative number? Because the CalcIt function was declared to only accept values that conform to SuperPositive, the system will ensure that all values passed to it are greater than five. To ensure this constraint is never violated, the system may need to inject a dynamic constraint test that has a potential to fail when evaluated. This failure will not occur when the M source text is first processed (as was the case with CalcIt(-1)) – rather it will occur when the expression is actually evaluated.

Here’s the general principle at play.

M implementations will typically attempt to report any constraint violations before the first expression is evaluated. This is called *static* enforcement and implementations will manifest this much like a syntax error. However, as we’ve seen, some constraints can only be enforced against live data and therefore require *dynamic* enforcement.

In general, the M philosophy is to make it easy for the user to write down their intention and put the burden on the M implementation to “make it work.” However, to allow a particular M program to be used in diverse environments, a fully featured M implementation should be configurable to reject M program that rely on dynamic enforcement for correctness in order to reduce the performance and operational costs of dynamic constraint violations.

### Collection types

M defines a *type* *constructor* for specifying collection types. The collection type constructor restricts the *type* and *count* of elements a collection may contain. All collection types are restrictions over the intrinsic type Collection, which all collection values conform to:

{ } in Collection

{ 1, false } in Collection

! ("Hello" in Collection)

The last example is interesting, in that it illustrates that the collection types do not overlap with the simple types. There is no value that conforms to both a collection type and a simple type.

A collection type constructor specifies both the type of element and the acceptable element count. The element count is typically specified using one of the three operators:

{T\*} - zero or more Ts

{T+} - one or more Ts

{T#m..n} – between m and n Ts.

The collection type constructors can either use operators or be written longhand as a constraint over the intrinsic type Collection:

type SomeNumbers : {Number+};

type TwoToFourNumbers : {Number#2..4};

type ThreeNumbers : {Number#3};

type FourOrMoreNumbers : {Number#4..};

These types describe the same sets of values as these longhand definitions:

type SomeNumbers : { Number \*} where value.Count >= 1 ;

type TwoToFourNumbers : { Number \*} where value.Count >= 2

 && value.Count <= 4;

type ThreeNumbers : { Number \*} where value.Count == 3;

type FourOrMoreNumbers : { Number \*} where value.Count >= 4;

Independent of which form is used to declare the types, we can now assert the following hold:

!({ } in TwoToFourNumbers)

!({ "One", "Two", "Three" } in TwoToFourNumbers)

{ 1, 2, 3 } in TwoToFourNumbers

{ 1, 2, 3 } in ThreeNumbers

{ 1, 2, 3, 4, 5 } in FourOrMoreNumbers

The collection type constructors compose with the where operator, allowing the following type check to succeed:

{ 1, 2 } in {(Number where value < 3)\*} where value.Count % 2 == 0

Note that the where inside the parentheses applies to elements of the collection, and the where outside the parentheses operator applies to the collection itself.

### List Types

Lists types are specified in a similar manner to collection types.

type ListOfNumbers : [ Number \*];

### Nullable types

We have seen many useful values: 42, "Hello", {1,2,3}. The distinguished value null serves as a place holder for some other value that is not known. A type with null in the value space is called a nullable type. The value null can be added to the value space of a type with an explicit union of the type and a collection containing null or using the postfix operator ?. The following expressions are true:

! (null in Integer)

null in Integer?

null in (Integer | { null } )

The ?? operator converts between a null value and known value:

null ?? 1 == 1

Arithmetic operations on a null operand return null:

1 + null == null

null \* 3 == null

Logical operators, conditional, and constraints require non nullable operands.

### Entity types

Just as we can use the collection type constructors to specify what kinds of collections are valid in a given context, we can do the same for entities using entity types.

An entity type declares the expected members for a set of entity values. The members of an entity type are called *fields*. The value of a field is stored. All entity types are restrictions over the Entity type.

Here is the simplest entity type:

type MyEntity : Language.Entity;

The type MyEntity does not declare any fields. In M, entity types are *open* in that entity values that conform to the type may contain fields whose names are not declared in the type. That means that the following type test:

{ X => 100, Y => 200 } in MyEntity

will evaluate to true, as the MyEntity type says nothing about fields named X and Y.

Most entity types contain one or more field declarations. At a minimum, a field declaration states the name of the expected field:

type Point { X; Y; }

This type definition describes the set of entities that contain *at least* fields named X and Y irrespective of the values of those fields. That means that the following type tests will all evaluate to true:

{ X => 100, Y => 200 } in Point

{ X => 100, Y => 200, Z => 300 } in Point // more fields than expected OK

! ({ X => 100 } in Point) // not enough fields – not OK

{ X => true, Y => "Hello, world" } in Point

The last example demonstrates that the Point type does not constrain the values of the X and Y fields – any value is allowed. We can write a new type that constrains the values of X and Y to numeric values:

type NumericPoint {

 X : Number;

 Y : Number where value > 0;
}

Note that we’re using type ascription syntax to assert that the value of the X and Y fields must conform to the type Number. With this in place, the following expressions all evaluate to true:

{ X => 100, Y => 200 } in NumericPoint

{ X => 100, Y => 200, Z => 300 } in NumericPoint

! ({ X => true, Y => "Hello, world" } in NumericPoint)

! ({ X => 0, Y => 0 } in NumericPoint)

As we saw in the discussion of simple types, the name of the type exists only so that M declarations and expressions can refer to it. That is why both of the following type tests succeed:

{ X => 100, Y => 200 } in NumericPoint

{ X => 100, Y => 200 } in Point

even though the definitions of NumericPoint and Point are independent.

### Declaring fields

Fields are named units of storage that hold values. M allows you to initialize the value of a field as part of an entity initializer. However, M does not specify any mechanism for changing the value of a field once it is initialized. In M, we assume that any changes to field values happen outside the scope of M.

A field declaration can indicate that there is a default value for the field. Field declarations that have a default value do not require conformant entities to have a corresponding field specified (we sometimes call such field declarations *optional fields)*. For example, consider this type definition:

type Point3d {

 X : Number;

 Y : Number;

 Z => -1 : Number; // default value of negative one
}

Because the Z field has a default value, the following type test will succeed:

{ X => 100, Y => 200 } in Point3d

Moreover, if we apply a type ascription operator to the value:

({ X => 100, Y => 200 } : Point3d)

we can now access the Z field like this:

({ X => 100, Y => 200 } : Point3d).Z

This expression will yield the value -1.

If a field declaration does not have a corresponding default value, conformant entities must specify a value for that field. Default values are typically written down using the explicit syntax shown for the Z field of Point3d. If the type of a field is either nullable or a zero-to-many collection, then there is an implicit default value for the declaring field of null for optional and {} for the collection.

For example, consider this type:

type PointND {

 X : Number;

 Y : Number;

 Z : Number?; // Z is optional

 BeyondZ : {Number\*}; // BeyondZ is optional too
}

Again, the following type test will succeed:

{ X => 100, Y => 200 } in PointND

and ascribing the PointND to the value will allow us to get these defaults:

({ X => 100, Y => 200 } : PointND).Z == null

({ X => 100, Y => 200 } : PointND).BeyondZ == { }

The choice of using a nullable type vs. an explicit default value to model optional fields typically comes down to style.

### Constraints on entity types

Like all types, a constraint may be applied to an entity type using the where operator. Consider the following type definition:

type HighPoint {

 X : Number;

 Y : Number;
} where X < Y;

In this example, all values that conform to the type HighPoint are guaranteed to have an X value that is less than the Y value. That means that the following expressions:

{ X => 100, Y => 200 } in HighPoint

! ({ X => 300, Y => 200 } in HighPoint)

both evaluate to true.

Now consider the following type definitions:

type Point {

 X : Number;

 Y : Number;

}

type Visual {

 Opacity : Number;

}

type VisualPoint {

 DotSize : Number;

} where value in Point && value in Visual;

The third type, VisualPoint, names the set of entity values that have at least the numeric fields X, Y, Opacity, and DotSize.

Because it is a common desire to factor member declarations into smaller pieces that can be easily composed, M provides explicit syntax support for this. We can rewrite the VisualPoint type definition using that syntax:

type VisualPoint : Point, Visual {

 DotSize : Number;

}

To be clear, this is just shorthand for the long-hand definition above that used a constraint expression. Both of these definitions are equivalent to this even longer-hand definition:

type VisualPoint {

 X : Number;

 Y : Number;

 Opacity : Number;

 DotSize : Number;

}

Again, the names of the types are just ways to refer to types – the values themselves have no record of the type names used to describe them.

## Queries

Queries operate over unordered and ordered collections. M provides LINQ query comprehensions and adds several features to make authoring simple queries more concise. The keywords, where and select are available as binary infix operators. Also, indexers are automatically added to strongly typed collections. These features allow common queries to be authored more compactly as illustrated below.

### Filtering

Filtering extracts elements from an existing collection. Consider the following collection:

People {

 { First => "Mary", Last => "Smith", Age => 24 },

 { First => "John", Last => "Doe", Age => 32 },

 { First => "Dave", Last => "Smith", Age => 32 },

}

This query extracts people with Age == 32 from the People collection:

from p in People

where p.Age == 32

select p

An equivalent query can be written with either of the following expressions:

People where value.Age == 32

People.Age(32)

The where operator takes a collection on the left and a Logical expression on the right. The where operator introduces a keyword identifier value into the scope of the Logical expression that is bound to each member of the collection. The resulting collection contains the members for which the expression is true. The expression:

Collection where Expression

is exactly equivalent to:

from value in Collection

where Expression

select value

Collection types gain *indexer* members that correspond to the fields of their corresponding element type. That is, this:

Collection . Field ( Expression )

is equivalent to:

from value in Collection

where Field == Expression

select value

### Selection

Select is also available as an infix operator. Consider the following simple query:

    from p in People

    select p.First + p.Last

This computes the select expression over each member of the collection and returns the result. Using the infix select it can be written equivalently as:

People select value.First + value.Last

The select operator takes a collection on the left and an arbitrary expression on the right. As with where, select introduces the keyword identifier value that ranges over each element in the collection. The select operator maps the expression over each element in the collection and returns the result. The expression:

Collection select Expression

Is exactly equivalent to:

from value in Collection

select Expression

A trivial use of the select operator is to extract a single field:

People select value.First

Collections are augmented with accessors to fields which can be extracted directly. For example People.First yields a new collection containing all the first names and People.Last yields a collection with all the last names.

## Modules

All of the examples shown so far have been “loose M” that is taken out of context. To write a legal M program, all source text must appear in the context of a *module definition*. A module defines a top-level namespace for any type names that are defined. A module also defines a scope for defining extents that will store actual values, as well as computed values.

Here is a simple module definition:

module Geometry {

 // declare a type

 type Point {

 X : Integer; Y : Integer;

 }

 // declare some extents

 Points : {Point\*};

 Origin : Point;

 // declare a computed value

 TotalPointCount { Points.Count + 1; }

}

In this example, the module defines one type named Geometry.Point. This type describes what point values will look like, but doesn’t mention any locations where those values can be stored.

This example also includes two module-scoped extents (Points and Origin). Module-scoped field declarations are identical in syntax to those used in entity types. However, fields declared in an entity type simply name the *potential* for storage once an extent has been determined; in contrast fields declared at module-scope name *actual* storage that must be mapped by an implementation in order to load and interpret the module.

Modules may refer to declarations in other modules by using an *import directive* to name the module containing the referenced declarations. For a declaration to be referenced by other modules, the declaration must be explicitly exported using an *export directive.*

Consider this module:

module MyModule {

 import HerModule; // declares HerType

 export MyType1;

 export MyExtent1;

 type MyType1 : {Logical\*};

 type MyType2 : HerType;

 MyExtent1 : {Number\*};

 MyExtent2 : HerType;

}

Note that only MyType1 and MyExtent1 are visible to other modules. This makes the following definition of HerModule legal:

module HerModule {

 import MyModule; // declares MyType1 and MyExtent1

 export HerType;

 type HerType : Text where value.Count < 100;

 type Private : Number where !(value in MyExtent1);

 SomeStorage : MyType1;

}

As this example shows, modules may have circular dependencies.

# Lexical Structure

An M program consists of one or more source files, known formally as compilation units. A compilation unit file is an ordered sequence of Unicode characters. Compilation units typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding. The byte order mark is optional.

Any Unicode code point with either of the following properties:

* General\_Category=Surrogate
* Noncharacter\_Code\_Point=True

is disallowed from occurring literally in M program text, as is any code point with General\_Category=Control unless it matches WhitespaceChar. The notation “none of” implicitly excludes any of these disallowed characters.

Rather than define a separate meta-language, this specification uses syntax, token, and interleave rules to define the M language. The concepts are introduced in §1.2 and defined in §5.

## Pre-Processing

Pre-processing directives provide the ability to conditionally skip sections of source files, as a separate pre-processing step.

token PPDirective

 = PPDeclaration
 | PPConditional;

This needs token defs.

The following pre-processing directives are available:

* #define which is used to define conditional compilation symbols.
* #if, #else, and #endif, which are used to conditionally skip sections of source code.

A pre-processing directive always occupies a separate line of source code and always begins with a # character and a pre-processing directive name. White space may *not* occur before the # character and between the # character and the directive name.

A source line containing a #define, #if, #else, or #endif directive may end with a single-line comment. Delimited comments (the /\* \*/ style of comments) are not permitted on source lines containing pre-processing directives. All #define directives have to appear before the first #if directive.

Pre-processing directives are neither tokens nor part of the syntactic grammar of M. However, pre-processing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of an M program. For example, after pre-processing the source text:

#define A
type C
{
#if A
 F {}
#else
 G {}
#endif

#if B
 H {}
#else
 I {}
#endif
}

results in the exact same sequence of tokens as the source text:

type C
{
 F {}
 I {}
}

Thus, whereas lexically, the two programs are quite different, syntactically, they are identical.

### Directives

## Lexical Analysis

### Whitespace

Whitespace is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

interleave Whitespace

 = WhitespaceCharacter+;

token WhitespaceCharacter

 = "\u0009" // Horizontal Tab

 | "\u000B" // Vertical Tab

 | "\u000C" // Form Feed

 | "\u0020" // Space

 | NewLineCharacter;

token NewLineCharacter

 = "\u000A" // New Line

 | "\u000D" // Carriage Return

 | "\u0085" // Next Line

 | "\u2028" // Line Separator

 | "\u2029"; // Paragraph Separator

### 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.

interleave Comment

 = CommentDelimited

 | CommentLine;

token CommentDelimited

 = "/\*" CommentDelimitedContent\* "\*/";

token CommentDelimitedContent

 = !("\*")

 | "\*" !("/");

token CommentLine

 = "//" CommentLineContent\*;

token CommentLineContent

 = !NewLineCharacter;

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

Comments are not processed within Text literals.

The example

// This defines a

// Person entity
//
type Person {

 Name : Text;

 Age : Number;

}

shows three single-line comments.

The example

/\* This defines a

 Person entity
\*/
type Person {

 Name : Text;

 Age : Number;

}

includes one delimited comment.

## Identifiers

A regular identifier begins with a letter or underscore and then any sequence of letter, underscore, dollar sign, or digit. An escaped identifier is enclosed in '@[]'. It contains any sequence of Text literal characters.

syntax Identifier

 = id:IdentifierVerbatim

 | id:IdentifierName;

token IdentifierBegin = "\_" | Letter;

token IdentifierCharacter

 = IdentifierBegin

 | "$"

 | DecimalDigit;

token IdentifierCharacters

 = IdentifierCharacter+;

token IdentifierVerbatim

 = "@[" IdentifierVerbatimCharacters "]";

token IdentifierVerbatimCharacter

 = !( "]" )

 | IdentifierVerbatimEscape;

token IdentifierVerbatimCharacters

 = IdentifierVerbatimCharacter+;

token IdentifierVerbatimEscape

 = "\\\\" | "\\]";

## Keywords

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when escaped with ' '.

Keywords:

any
accumulate
by
empty
equals
error
export
false
final
from
group
id
identity
import
in
interleave
join
language
labelof
left
let
module
null
precedence
right
select
syntax
token
true
type
unique
value
valuesof
where

The following keywords are reserved for future use:

checkpoint identifier nest override new virtual partial

## 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.

[ ] ( ) . , : ; ? = < > <= >= == != + - \* / % & | ! && || ~ << >> { } # .. @ ' " ?? => ::

## Literals

A literal is a source code representation of a value.

syntax Literal

 = Binary

 | Date

 | DateTime

 | DateTimeOffset

 | Decimal

 | Guid

 | Integer

 | Logical

 | Null

 | Scientific

 | Text

 | Time;

### Date literals

Date literals are used to write a date independent of a specific time of day.

token Date

 = Sign? DateYear "-" DateMonth "-" DateDay;

token DateDay

 = "01" | "02" | "03" | "04" | "05" | "06" | "07" | "08" | "09" | "10"

 | "11" | "12" | "13" | "14" | "15"

 | "16" | "17" | "18" | "19" | "20" | "21" | "22" | "23" | "24" | "25"

 | "26" | "27" | "28" | "29" | "30"

 | "31";

token DateMonth

 = "01" | "02" | "03" | "04" | "05" | "06" | "07" | "08"

 | "09" | "10" | "11" | "12";

token DateYear

 = DecimalDigit DecimalDigit DecimalDigit DecimalDigit;

The type of a DateLiteral is Date.

* 0001-01-01 is the representation of January1st, 1 AD.
* There is no year 0, therefore ‘0000-01-01’ is not a valid Date Time
* -0001-01-01 is the representation of January1st, 1 BC.

Examples of date literal follow:

0001-01-01

2008-08-14

-1184-03-01

### DateTime literals

DateTime literals are used to write a time of day on a specific date independent of time zone.

token DateTime

 = DateLiteral "T" TimeLiteral;

The type of a DateTime literal is DateTime.

Example of date time literal follow:

2008-08-14T13:13:00

0001-01-01T00:00:00

2005-05-19T20:05:00

### DateTimeOffset literals

DateTimeOffset literals are used to write a time of day on a specific date within a specific time zone.

token DateTimeOffset = Date "T" Time TimeZone;

token TimeZone

 = Sign OffsetTimeHourMinute

 | "Z";

token OffsetTimeHour

 = "00" | "01" | "02" | "03" | "04" | "05" | "06" | "07" | "08" | "09"

 | "10" | "11" | "12" | "13" | "14";

token OffsetTimeHourMinute

 = OffsetTimeHour ":" TimeMinute;

The type of a DateTimeOffset literal is DateTimeOffset.

Example of date time literal follow:

2008-08-14T13:13:00+06:00

0001-01-01T00:00:00-03:00

2005-05-19T20:05:00Z

### Decimal literals

Decimal literals are used to write fixed-point or exact number values.

token Decimal = DecimalDigit+ "." DecimalDigit+;

Decimal literals default to the smallest standard library type that that can contain the value. Examples of decimal literal follow:

99.999

0.1

1.0

### Guid literals

token GuidLiteral

 = "#" "[" HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit

 HexDigit "-" HexDigit HexDigit HexDigit HexDigit "-"

 HexDigit HexDigit HexDigit HexDigit "-"

 HexDigit HexDigit HexDigit HexDigit "-"

 HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit

 HexDigit HexDigit HexDigit HexDigit HexDigit "]";

token HexDigit = "0".."9" | "a".."f" | "A".."F";

The type of a GuidLiteral is Guid.

Examples of Guid literal follows:

#[a0ee7e0f-c6ac-4c63-b57f-816a5259595a]

#[7fbc28ba-8205-45ca-983e-ece117f7a776]

#[a05e63ca-25de-43a6-bf70-0bc04d40a000]

### Integer literals

Integer literals are used to write integral values.

token Integer

 = Decimal

 | Hexadecimal;

token Decimal = DecimalDigit+;

token DecimalDigit = "0".."9";

token Hexadecimal

 = "0x" HexDigit\*

 | "0X" HexDigit\*;

Decimal literals default to the smallest precision type that can contain the value, starting with Integer32. Hexadecimal literals default to the smallest precision type that can contain the value starting with Unsigned32.

Examples of integer literal follow:

0

123

999999999999999999999999999999

0x00

0XFF

0x1234

### Logical literals

Logical literals are used to write logical values.

syntax Logical

 = "true"

 | "false";

The type of a Logical literal is Logical.

Examples of logical literal:

true

false

### Null literal

The null literal is equal to no other value.

syntax Null

 = "null";

The type of a null literal is Null.

An example of the null literal follows:

null

### Scientific literals

Scientific literals are used to write floating-point or inexact numbers.

token ScientificLiteral

 = Decimal "e" Sign? DecimalDigit+

 | Decimal "E" Sign? DecimalDigit+;

token Sign

 = "+"

 | "-";

Scientific literals default to the smallest precision type that can contain the value, starting with Double.

This is not clear.

Examples of scientific literal follow:

0.31416e+1

9.9999e-1

0.0E0

### Text literals

M supports two forms of Text literals: regular Text literals and verbatim Text literals.

A regular Text literal consists of zero or more characters enclosed in either double quotes, as in "hello", or in single quotes, as in 'hello', and may include both simple escape sequences (such as \t for the tab character), and hexadecimal and Unicode escape sequences.

A verbatim Text literal consists of an @ character followed by a double-quote character, zero or more characters, and a closing double-quote character (or likewise using only single-quote characters). A simple example is @"hello" and @'hello'. In a verbatim Text literal, the characters between the delimiters are interpreted exactly as they occur in the compilation unit, the only exception being a QuoteEscapeSequence. In particular, simple escape sequences, and hexadecimal and Unicode escape sequences are not processed in verbatim Text literals. A verbatim Text literal may span multiple lines.

token TextLiteral

 = RegularStringLiteral

 | VerbatimStringLiteral;

token RegularStringLiteral

 = '"' DoubleQuoteTextCharacter\* '"'

 | "'" SingleQuoteTextCharacter\* "'";

token VerbatimStringLiteral

 = '@' '"' DoubleQuoteTextVerbatimCharacter\* '"'

 | '@' "'" SingleQuoteTextVerbatimCharacter\* "'";

token SingleQuoteTextCharacter

 = SingleQuoteTextSimple

 | CharacterEscapeSimple

 | CharacterEscapeUnicode ;

token SingleQuoteTextSimple

 = !( '\u0027' // Single Quote

 | '\u005C' // Backslash

 | NewLineCharacter) ;

token SingleQuoteTextVerbatimCharacter

 = !('\u0027') // SingleQuote

 | SingleQuoteTextVerbatimCharacterEscape ;

token SingleQuoteTextVerbatimCharacterEscape = '\u0027' '\u0027';

token SingleQuoteTextVerbatimCharacters = SingleQuoteTextVerbatimCharacter+;

token DoubleQuoteTextCharacter

 = DoubleQuoteTextSimple

 | CharacterEscapeSimple

 | CharacterEscapeUnicode ;

token DoubleQuoteTextSimple

 = !( '\u0022' // DoubleQuote

 | '\u005C' // Backslash

 | NewLineCharacter) ;

token DoubleQuoteTextVerbatimCharacter

 = !('\u0022') // DoubleQuote

 | DoubleQuoteTextVerbatimCharacterEscape ;

token DoubleQuoteTextVerbatimCharacterEscape = '\u0022' '\u0022';

token DoubleQuoteTextVerbatimCharacters = DoubleQuoteTextVerbatimCharacter+;

token NewLineCharacter

 = '\u000A' // New Line

 | '\u000D' // Carriage Return

 | '\u0085' // Next Line

 | '\u2028' // Line Separator

 | '\u2029'; // Paragraph Separator

token CharacterEscapeSimple = '\u005C' CharacterEscapeSimpleCharacter;

token CharacterEscapeSimpleCharacter

 = "'" // Single Quote

 | '"' // Double Quote

 | '\u005C' // Backslash

 | '0' // Null

 | 'a' // Alert

 | 'b' // Backspace

 | 'f' // Form Feed

 | 'n' // New Line

 | 'r' // Carriage Return

 | 't' // Horizontal Tab

 | 'v'; // Vertical Tab

token CharacterEscapeUnicode

 = "\\u" HexDigit#4

 | "\\U" HexDigit#8;

The value of the hexadecimal number in the NumericEscape must be between 0 and 10FFFF, but must not be a surrogate (that is, must not be between D800 and DFFF).

The type of a Text literal is Text.

Examples of text literal follow:

"a"

"\u2323"

"Hello World"

@"""Hello World"""

"\u2323"

The following values are all equal:

"@"

"\u26"

"\u0026"

"\@"

### Time literals

Time literals are used to write a time of day independent of a specific date or time zone.

token TimeLiteral

 = TimeHourMinute ":" TimeSecond;

token TimeHour

 = "00" | "01" | "02" | "03" | "04" | "05" | "06" | "07" | "08" | "09"

 | "10" | "11"

 | "12" | "13" | "14" | "15" | "16" | "17" | "18" | "19" | "20" | "21"

 | "22" | "23";

token TimeHourMinute

 = TimeHour ":" TimeMinute;

token TimeMinute

 = "0" DecimalDigit

 | "1" DecimalDigit

 | "2" DecimalDigit

 | "3" DecimalDigit

 | "4" DecimalDigit

 | "5" DecimalDigit;

token TimeSecond

 = "0" DecimalDigit TimeSecondDecimalPart?

 | "1" DecimalDigit TimeSecondDecimalPart?

 | "2" DecimalDigit TimeSecondDecimalPart?

 | "3" DecimalDigit TimeSecondDecimalPart?

 | "4" DecimalDigit TimeSecondDecimalPart?

 | "5" DecimalDigit TimeSecondDecimalPart?;

token TimeSecondDecimalPart = "." DecimalDigits;

Examples of time literal follow:

11:30:00

01:01:01.111

13:13:00

# Text Pattern Expressions

Text pattern expressions perform operations on the sets of possible text values that one or more terms recognize.

## Primary Expressions

A primary expression can be:

* A text literal
* A reference to a syntax or token rule
* An expression indicating a repeated sequence of primary expressions of a specified length
* An expression indicating any of a continuous range of characters
* An inline sequence of pattern declarations

The following grammar reflects this structure.

syntax SyntaxPrimaries

 = TextLiteral

 | ReferencePrimary

 | RepetitionPrimary

 | InlineRulePrimary

 | CharacterClassPrimary

 | AnyPrimary;

### Character Class

A character class is a compact syntax for a range of continuous characters. This expression requires that the text literals be of length 1 and that the Unicode offset of the right operand be greater than that of the left.

syntax CharacterClassPrimary

 = TextLiteral ".." TextLiteral

The expression "0".."9" is equivalent to:

"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

### References

A grammar reference is the name of another rule possibly with arguments for parameterized rules. All rules defined within the same language can be accessed without qualification. The protocol to access rules defined in a different language within the same module are defined in §6.2. The protocol to access rules defined in a different module are defined in §10.3.

syntax GrammarReference
 = Identifier
 | GrammarReference "." Identifier
 | GrammarReference "." Identifier "(" TypeArguments ")"
 | Identifier "(" TypeArguments ")";

syntax TypeArguments
 = PrimaryExpression
 | TypeArguments "," PrimaryExpression;

Note that whitespace between a rule name and its arguments list is significant to discriminate between a reference to a parameterized rule and a reference without parameters and an inline rule. In a reference to a parameterized rule, no whitespace is permitted between the identifier and the arguments.

### Repetition operators

The repetition operators recognize a primary expression repeated a specified number of times. The number of repetitions can be stated as a (possibly open) integer range or using one of the Kleene operators, ?, +, \*.

syntax RepetitionPrimary

 = Primary Range
 | Primary CollectionRanges;

syntax Range
 = "?"
 | "\*"
 | "+";

syntax CollectionRanges
 = "#" IntegerLiteral
 | "#" IntegerLiteral ".." IntegerLiteral?;

The left operand of .. must be greater than zero and less than the right operand of .., if present.

|  |  |  |
| --- | --- | --- |
| "A"#5 | recognizes exactly 5 "A"s | "AAAAA" |
| "A"#2..4 | recognizes from 2 to 4 "A"s | "AA", "AAA", "AAAA" |
| "A"#3.. | recognizes 3 or more "A"s | "AAA", "AAAA", "AAAAA", . . . |

The Kleene operators can be defined in terms of the collection range operator:

"A"? is equivalent to "A"#0..1

"A"+ is equivalent to "A"1..

"A"\* is equivalent to "A"#0..

### Inline Rules

An inline rule is a means to group pattern declarations together as a term.

syntax InlineRulePrimary
 = "(" ProductionDeclarations ")";

An inline rule is typically used in conjunction with a range operator:

"A" ("," "A")\* recognizes 1 or more "A"s separated by commas.

Although syntactically legal, variable bindings within inline rules are not accessible within the constructor of the containing production. Inline rules are described further in §5.4.

### Any

The any term is a wildcard that matches any text value of length 1.

syntax AnyPrimary
 = "any";

"1", "z", and "\*" all match any.

## Term Operators

A primary term expression can be thought of as the set of possible text values that it recognizes. The term operators perform the standard set difference, intersection, and negation operations on these sets. (Pattern declarations perform the union operation with |.)

syntax TextPatternExpression
 = Difference;

syntax Difference
 = Intersect
 | Difference "-" Intersect;

syntax Intersect
 = Inverse
 | Intersect "&" Inverse;

syntax Inverse
 = Primary
 | "!" Primary;

Inverse requires every value in the set of possible text values to be of length 1.

("11" | "12") – ("12" | "13") // recognizes "11"

("11" | "12") & ("12" | "13") // recognizes "12"

!("11" | "12") // is an error

!("1" | "2") // recognizes any text value of length 1 other than "1" or "2"

# Productions

A production is a pattern and an optional constructor. Each production is a scope. The pattern may establish variable bindings which can be referenced in the constructor. A production can be qualified with a precedence that is used to resolve a tie if two productions match the same text (See §4.4.1).

syntax ProductionDeclaration
 = ProductionPrecedence? PatternDeclaration Constructor?;

syntax Constructor
 = "=>" TermConstructor;

syntax ProductionPrecedence
 = "precedence" IntegerLiteral ":";

## Pattern Declaration

A pattern declaration is a sequence of term declarations or the built-in pattern empty which matches "".

syntax PatternDeclaration
 = "empty"
 | TermDeclarations?;

syntax TermDeclarations
 = TermDeclaration
 | TermDeclarations TermDeclaration;

## Term Declaration

A term declaration consists of a pattern expression with an optional variable binding, precedence and attributes. The built-in term error is used for error recovery and discussed in §4.2.1.

syntax TermDeclaration
 = "error"
 | Attributes? TermPrecedence?VariableBinding?TextPatternExpression;

syntax VariableBinding
 = Name ":";

syntax TermPrecedence
 = "left" "(" IntegerLiteral ")"
 | "right" "(" IntegerLiteral ")";

A variable associates a name with the output from a term which can be used in the constructor. Precedence is discussed in §4.4.2.

The error term is used to facilitate error recovery. This is described in §4.2.1.

### Error

The error production enables error recover. Consider the following example:

module HelloWorld {

 language HelloWorld {

 syntax Main

 = HelloList;

 token Hello

 = "Hello";

 checkpoint syntax HelloList

 = Hello

 | HelloList "," Hello

 | HelloList "," error;

 }

}

The language recognizes the text "Hello,Hello,Hello" as expected and produces the following default output:

Main[

 HelloList[

 HelloList[

 HelloList[

 Hello

 ],

 ,,

 Hello

 ],

 ,,

 Hello

 ]

]

The text "Hello,hello,Hello" is not in the language because the second "h" is not capitalized (and case sensitivity is true). However, rather than stop at "h", the language processor matches "h" to the error token, then matches "e" to the error token, etc. until it reaches the comma. At this point the text conforms to the language and normal processing can continue. The language process reports the position of the errors and produces the following output:

Main[

 HelloList[

 HelloList[

 HelloList[

 Hello

 ],

 error["hello"],

 ],

 ,,

 Hello

 ]

]

Hello occurs twice instead of three times as above and the text the error token matched is returned as error["hello"].

## Constructors

A term constructor is the syntax for defining the output of a production. A node in a term constructor can be:

* An atom consisting of
	+ A literal
	+ A reference to another term
	+ An operation on a reference
* A ordered collection of successors with an optional label
* An unordered collection of successors with an optional label

The following grammar mirrors this structure.

syntax TermConstructor
 = TopLevelNode;

syntax Node
 = Atom
 | OrderedTerm
 | UnorderedTerm;

syntax TopLevelNode
 = TopLevelAtom
 | OrderedTerm
 | UnorderedTerm;

syntax Nodes
 = Node
 | Nodes "," Node;

syntax OrderedTerm
 = Label? "[" Nodes? "]";

syntax UnorderedTerm
 = Label? "{" Nodes? "}";

syntax Label
 = Identifier
 | "id" "(" Atom ")";

syntax Atom
 = TopLevelAtom
 | "valuesof" "(" VariableReference ")";

syntax TopLevelAtom
 = TextLiteral
 | DecimalLiteral
 | LogicalLiteral
 | IntegerLiteral
 | NullLiteral
 | VariableReference
 | "labelof" "(" VariableReference ")";

syntax VariableReference
 = Identifier;

Each production defines a scope. The variables referenced in a constructor must be defined within the same production's pattern. Variables defined in other productions in the same rule cannot be referenced. The same variable name can be used across alternatives in the same rule.

Consider three alternatives for encoding the output of the same production. First, the default constructor (See §4.3.2):

module Expression {

 language Expression {

 token Digits = ("0".."9")+;

 syntax Main = E;

 syntax E

 = Digits

 | E "+" E ;

 }

}

Processing the text "1+2" yields:

Main[E[E[1], +, E[2]]]

This output reflects the structure of the grammar and may not be the most useful form for further processing. The second alternative cleans the output up considerably:

module Expression {

 language Expression {

 token Digits = ("0".."9")+;

 syntax Main

 = e:E => e;

 syntax E

 = d:Digits => d

 | l:E "+" r:E => Add[l,r] ;

 }

}

Processing the text "1+2" with this language yields:

Add[1, 2]

This grammar uses three common patterns.

* Productions with a single term are passed through. This is done for the single production in Main and the first production in E.
* A label, Add, is used to designate the operator.
* Position is used to distinguish the left and right operand.

The third alternative uses a record like structure to give the operands names:

module Expression {

 language Expression {

 token Digits = ("0".."9")+;

 syntax Main

 = e:E => e;

 syntax E

 = d:Digits => d

 | l:E "+" r:E => Add{Left{l},Right{r}} ;

 }

}

Processing the text "1+2" with this language yields:

Add{Left{1}, Right{2}}

Although somewhat more verbose than the prior alternative, this output does not rely on ordering and forces consumers to explicitly name Left or Right operands. Although either option works, this has proven to be more flexible and less error prone.

### Constructor operators

Constructor operators allow a constructor to use a variable reference as a label, extract the successors of a variable reference or extract the label of a variable reference.

Consider generalizing the example above to support multiple operators. This could be done by adding a new production for each operator -, \*, /, ^. Alternatively a single rule can be established to match these operators and the output of that rule can be used as a label using id:

module Expression {

 language Expression {

 token Digits = ("0".."9")+;

 syntax Main

 = e:E => e;

 syntax Op

 = "+" => "Add"

 | "-" => "Subtract"

 | "\*" => "Multiply"

 | "/" => "Divide" ;

 syntax E

 = d:Digits => d

 | l:E o:Op r:E => id(o){Left[l],Right[r]} ;

 }

}

Processing the text "1+2" with this language yields the same result as above. Processing
"1 / 2" yields:

Divide{Left{1}, Right{2}}

This language illustrates the id operator. See §4.4.2 for a more realistic expression grammar.

The valuesof operator extract the successors of a variable reference. It is used to flatten nested output structures. Consider the language:

module Digits {

 language Digits {

 syntax Main = DigitList ;

 token Digit = "0".."9";

 syntax DigitList

 = Digit

 | DigitList "," Digit ;

 }

}

Processing the text "1, 2, 3" with this language yields:

Main[

 DigitList[

 DigitList[

 DigitList[

 1

 ],

 ,,

 2

 ],

 ,,

 3

 ]

]

The following grammar uses valuesof and the pass through pattern above to simplify the output:

module Digits {

 language Digits {

 syntax Main = dl:DigitList => dl ;

 token Digit = "0".."9";

 syntax DigitList

 = d:Digit => DigitList[d]

 | dl:DigitList "," d:Digit => DigitList[valuesof(dl),d] ;

 }

}

Processing the text "1, 2, 3" with this language yields:

DigitList[1, 2, 3]

This output represents the same information more concisely.

### Default Constructor

If a constructor is not defined for a production the language process defines a default constructor. For a given production, the default projection is formed as follows:

1. The label for the result is the name of the production’s rule.
2. The successors of the result are an ordered sequence constructed from each term in the pattern.
3. \* and ? create an unlabeled sequence with the elements.
4. ( ) results in an anonymous definition.
	1. If it contains constructors ( a:A => a), then the output is the output of the constructor.
	2. If there are no constructors, then the default rule applied on the anonymous definition and the output is enclosed in square brackets [ A’s result ]
5. Token rules do not permit a constructor to be specified and output text values.
6. Interleave rules do not permit a constructor to be specified and do not produce output.

Consider the following language:

module ThreeDigits {

 language ThreeDigits {

 token Digit = "0".."9";

 syntax Main

 = Digit Digit Digit ;

 }

}

Given the text "123" the default output of the language processor follows:

Main[

 1,

 2,

 3

]

## Precedence

The M language processor is tolerant of such ambiguity as it is recognizing subsequences of text. However, it is an error to produce more than one output for an entire text value. Precedence qualifiers on productions or terms determine which of the several outputs should be returned.

### Production Precedence

Consider the classic dangling else problem as represented in the following language:

module IfThenElse {

 language IfThenElse {

 syntax Main = S;

 syntax S

 = empty

 | "if" E "then" S

 | "if" E "then" S "else" S;

 syntax E = empty;

 interleave Whitespace = " ";

 }

}

Given the input "if then if then else", two different output are possible. Either the else binds to the first if-then:

if

then

 if

 then

else

Or it binds to the second if-then:

if

then

 if

 then

 else

The following language produces the output immediately above, binding the else to the second if-then.

module IfThenElse {

 language IfThenElse {

 syntax Main = S;

 syntax S

 = empty

 | precedence 2: "if" E "then" S

 | precedence 1: "if" E "then" S "else" S;

 syntax E = empty;

 interleave Whitespace = " ";

 }

}

Switching the precedence values produces the first output.

### Term Precedence

Consider a simple expression language which recognizes:

2 + 3 + 4

5 \* 6 \* 7

2 + 3 \* 4

2 ^ 3 ^ 4

The result of these expressions can depend on the order in which the operators are reduced. 2 + 3 + 4 yields 9 whether 2 + 3 is evaluated first or 3 + 4 is evaluated first. Likewise, 5 \* 6 \* 7 yields 210 regardless of the order of evaluation.

However, this is not the case for 2 + 3 \* 4. If 2 + 3 is evaluated first yielding 5, 5 \* 4 yields 20. While if 3 \* 4 is evaluated first yielding 12, 2 + 12 yields 14. This difference manifests itself in the output of the following grammar:

module Expression {

 language Expression {

 token Digits = ("0".."9")+;

 syntax Main = e:E => e;

 syntax E

 = d:Digits => d

 | "(" e:E ")" => e

 | l:E "^" r:E => Exp[l,r]

 | l:E "\*" r:E => Mult[l,r]

 | l:E "+" r:E => Add[l,r];

 interleave Whitespace = " ";

 }

}

"2 + 3 \* 4" can result in two outputs:

Mult[Add[2, 3], 4]

Add[2, Mult[3, 4]]

According to the rules we all learned in school, the result of this expression is 14 because multiplication is performed before addition. This is expressed in M by assigning "\*" a higher precedence than "+". In this case the result of an expression changed with the order of evaluation of different operators.

The order of evaluation of a single operator can matter as well. Consider 2 ^ 3 ^ 4. This could result in either 8 ^ 4 or 2 ^ 81. In term of output, there are two possibilities:

Exp[Exp[2, 3], 4]

Exp[2, Exp[3, 4]]

In this case the issue is not which of several different operators to evaluate first but which in a sequence of operators to evaluate first, the leftmost or the right most. The rule in this case is less well established but most languages choose to evaluate the rightmost "^" first yielding 2 ^ 81 in this example.

The following grammar implements these rules using term precedence qualifiers. Term precedence qualifiers may only be applied to literals or references to token rules.

module Expression {

 language Expression {

 token Digits = ("0".."9")+;

 syntax Main = E;

 syntax E

 = d:Digits => d

 | "(" e:E ")" => e

 | l:E right(3) "^" r:E => Exp[l,r]

 | l:E left(2) "\*" r:E => Mult[l,r]

 | l:E left(1) "+" r:E => Add[l,r];

 interleave Whitespace = " ";

 }

}

"^" is qualified with right(3). right indicates that the rightmost in a sequence should be grouped together first. 3 is the highest precedence, so "^" will be grouped most strongly.

# Rules

A rule is a named collection of alternative productions. There are three kinds of rules: syntax, token, and interleave. A text value conforms to a rule if it conforms to any one of the productions in the rule. If a text value conforms to more than one production in the rule, then the rule is ambiguous. The three different kinds of rules differ in how they treat ambiguity and how they handle their output.

syntax RuleDeclaration
 = Attributes? MemberModifier? Kind Name RuleParameters? RuleBody ";";

syntax Kind
 = "token"
 | "syntax"
 | "interleave";

syntax MemberModifier
 = "final";

syntax RuleBody
 = "=" ProductionDeclarations;

syntax ProductionDeclarations
 = ProductionDeclaration
 | ProductionDeclarations "|" ProductionDeclaration;

The rule Main below recognizes the two text values "Hello" and "Goodbye".

module HelloGoodby {

 language HelloGoodbye {

 syntax Main

 = "Hello"

 | "Goodbye";

 }

}

## Token Rules

Token rules recognize a restricted family of languages. However, token rules can be negated, intersected and subtracted which is not the case for syntax rules. Attempting to perform these operations on a syntax rule results in an error. The output from a token rule is the text matched by the token. No constructor may be defined.

### Final Modifier

Token rules do not permit precedence directives in the rule body. They have a built in protocol to deal with ambiguous productions. A language processor attempts to match all tokens in the language against a text value starting with the first character, then the first two, etc. If two or more productions within the same token or two different tokens can match the beginning of a text value, a token rule will choose the production with the longest match. If all matches are exactly the same length, the language processor will choose a token rule marked final if present. If no token rule is marked final, all the matches succeed and the language processor evaluates whether each alternative is recognized in a larger context. The language processor retains all of the matches and begins attempting to match a new token starting with the first character that has not already been matched.

## Syntax Rules

Syntax rules recognize all languages that M is capable of defining. The Main start rule must be a syntax rule. Syntax rules allow all precedence directives and may have constructors.

## Interleave Rules

An interleave rule recognizes the same family of languages as a token rule and also cannot have constructors. Further, interleave rules cannot have parameters and the name of an interleave rule cannot be references.

Text that matches an interleave rule is excluded from further processing.

The following example demonstrates whitespace handling with an interleave rule:

module HelloWorld {

 language HelloWorld {

 syntax Main =

 = Hello World;

 token Hello

 = "Hello";

 token World

 = "World";

 interleave Whitespace

 = " ";

 }

}

This language recognizes the text value "Hello World". It also recognizes "Hello World",
" Hello World", "Hello World ", and "HelloWorld". It does not recognize
"He llo World" because "He" does not match any token.

## Inline Rules

An inline rule is an anonymous rule embedded within the pattern of a production. The inline rule is processed as any other rule however it cannot be reused since it does not have a name. Variables defined within an inline rule are scoped to their productions as usual. A variable may be bound to the output of an inline rule as with any pattern.

In the following Example1 and Example2 recognize the same language and produce the same output. Example1 uses a named rule AppleOrOrange while Example2 states the same rule inline.

module Example {

 language Example1 {

 syntax Main

 = aos:AppleOrOrange\*

 => aos;

 syntax AppleOrOrange

 = "Apple" => Apple{}

 | "Orange" => Orange{};

 }

 language Example2 {

 syntax Main

 = aos:("Apple" => Apple{} | "Orange" => Orange{})\*

 => aos;

 }

}

## Rule Parameters

A rule may define parameters which can be used within the body of the rule.

syntax RuleParameters
 = "(" RuleParameterList ")";

syntax RuleParameterList
 = RuleParameter
 | RuleParameterList "," RuleParameter;

syntax RuleParameter
 = Identifier;

A single rule identifier may have multiple definitions with different numbers of parameters. The following example uses List(Content,Separator) to define List(Content) with a default separator of ",".

module HelloWorld {

 language HelloWorld {

 syntax Main

 = List(Hello);

 token Hello

 = "Hello";

 syntax List(Content, Separator)

 = Content

 | List(Content,Separator) Separator Content;

 syntax List(Content) = List(Content, ",");

 }

}

This language will recognize "Hello", "Hello,Hello", "Hello,Hello,Hello", etc.

# Languages

A language is a named collection of rules for imposing structure on text.

syntax LanguageDeclaration
 = Attributes?"language" Name LanguageBody;

syntax LanguageBody
 = "{" RuleDeclarations? "}";

syntax RuleDeclarations
 = RuleDeclaration
 | RuleDeclarations RuleDeclaration;

The language that follows recognizes the single text value "Hello World":

module HelloWorld {

 language HelloWorld {

 syntax Main

 = "Hello World";

 }

}

## Main Rule

A language may consist of any number of rules. The following language recognizes the single text value "Hello World":

module HelloWorld {

 language HelloWorld {

 syntax Main

 = Hello Whitespace World;

 token Hello

 = "Hello";

 token World

 = "World";

 token Whitespace

 = " ";

 }

}

The three rules Hello, World, and Whitespace recognize the three single text values "Hello", "World", and " " respectively. The rule Main combines these three rules in sequence. The difference between syntax and token rules is described in §5.1.

Main is the distinguished start rule for a language. A language recognizes a text value if and only if Main recognizes a value. Also, the output for Main is the output for the language.

## Cross-language rule references

Rules are members of a language. A language can use rules defined in another language using member access notation. The HelloWorld language recognizes the single text value "Hello World" using rules defined in the Words language:

module HelloWorld {

 language Words {

 token Hello

 = "Hello";

 token World

 = "World";

 }

 language HelloWorld {

 syntax Main

 = Words.Hello Whitespace Words.World;

 token Whitespace =

 = " ";

 }

}

All rules defined within the same module are accessible in this way. Rules defined in other modules must be exported and imported as defined in §10.3.

# Types

The types of the M language are divided into two main categories: intrinsic types and derived types. An intrinsic type is a type that cannot be defined using M language constructs but rather is defined entirely in the M Language Specification. An *intrinsic* type (e.g. Number, Entity, Collection) may name a super-type as part of its specification. Values are an instance of exactly one intrinsic type, and conform to the specification of that one intrinsic type and all of its super-types.

A *derived* type (e.g. Integer32, Person, Cars) is a type whose definition is constructed in M source text using the type constructors that are provided in the language. A derived type is defined as a constraint over another type, which creates an explicit subtyping relationship. Values conform to any number of derived types simply by virtue of satisfying the derived type’s constraint. There is no explicit affiliation between a value and a derived type – rather a given value that conforms to a derived type’s constraint may be interpreted as that type or any other derived type using type *ascription*.

## Type declaration

M offers a broad range of options in defining types. Any expression that returns a collection can be declared as a type. The type predicates for entities and collections are expressions and fit this form. A type declaration may explicitly enumerate its members or be composed of other types.

The syntax for a type declaration follows:

syntax TypeDeclaration
 = "type" Identifier ";"
 | "type" Identifier InitializationExpression ";"?
 | "type" Identifier EntityTypeExpression ";"?
 | "type" Identifier EntityTypeExpression "where"

 WhereExpressions ";"
 | "type" Identifier ":" Expression ";"
 | "type" Identifier ":" TypeReferences ";"
 | "type" Identifier ":" TypeReferences EntityTypeExpression ";"?
 | "type" Identifier ":" TypeReferences EntityTypeExpression "where"

 WhereExpressions ";";

The Identifier in a type declaration introduces a new symbol into the module level scope.

syntax TypeReference
 = QualifiedIdentifier;

syntax TypeReferences
 = TypeReference
 | TypeReferences "," TypeReference;

The QualifiedIdentifier in TypeReference must either refer to a type declaration available within the current scope (§9.2.1).

The declaration:

type SomeNewType;

declares a new type SomeNewType with no constraints. Any value satisfies this type.

The following example explicitly enumerates the values of type PrimaryColors and uses it in the EntityExpression which defines the type Car.

type PrimaryColors {"Red", "Blue", "Yellow"}

type Car {

 Make : Text;

 Model : Text;

 Color : PrimaryColors;

}

These common cases do not require a colon between the declaration name and the definition.

Type declarations can be built up from expressions that return collections. The type PrimaryColors above could be constructed from singleton sets.

type PrimaryColors2 : {"Red"} | {"Blue"} | {"Yellow"}

Since the expression {"Red"} | {"Blue"} | {"Yellow"} == {"Red", "Blue", "Yellow"} the two declarations are equivalent.

However an expression which does not return a collection is not a semantically valid type.

type NonSense : 1 + 1;

is a syntactically valid declaration, but not useful as a type since no value of X would ever satisfy, the following expression because 2 is not a collection.

X in 2

Entity types may be composed as well. Consider the following two distinct types:

type Vehicle {

 Owner : Text;

 Registration : Text;

}

type HasWheels {

 Wheels : Integer32;

}

The type Vehicle requires that instances have Owner and Registration fields. The type HasWheels requires instances have a Wheels field. These two types can be combined into a new type Car that requires Owner, Registration, and Wheels fields.

type Car : Vehicle & HasWheels;

In this usage, ampersand requires that Car meet all the requirements of both arguments.

This definition of Car can be further restricted since cars have 4 wheels. Such restrictions can be specified with a constraint (§9.14.1).

type Car2 : Vehicle & HasWheels where value.Wheels == 4;

It is common to extend types with additional fields and restrict values. M provides the following syntax to simplify this case.

type Car3 : Vehicle {

 Wheels : Integer32;

} where value.Wheels == 4;

## Subtyping

M is a structurally typed language rather than a nominally typed language like C++ or C#. A structural type is a specification for a set of values. Two types are equivalent if the exact same collection of values conforms to both regardless of the name of the types.

It is not required that a type be named to be used. A type expression is allowed wherever a type reference is required. Types in M are simply expressions that return collections.

If every value that conforms to type A also conforms to type B, we say that A is a *subtype* of B (and that B is a *super*-*type* of A). Subtyping is transitive, that is, if A is a subtype of B and B is a subtype of C, then A is a subtype of C (and C is a super-type of A). Subtyping is reflexive, that is, A is a (vacuous) subtype of A (and A is a super-type of A).

## Operators

Types are considered collections of all values that satisfy the type predicate. For that reason, any operation on a collection (§7.6.2) can be applied to a type and a type can be manipulated with expressions like any other collection value.

The relational operators ( <, >, <=, >=, ==, != ) compare the value spaces of two types and return a Logical value. For example, the operator <= on types computes the subtype relation.

(Car <= Vehicle) == true

(Car <= HasWheels) == true

(Car <= Colors) == false

The where constraint restricts the value space of a type to those elements satisfying the right operand's logical expression.

The following binary operations take Collection as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| &, | | Collection | Collection |

The union and intersection operators (|, &) operate on the type's value spaces. Intersection, &, can be thought of as specialization, restriction, or subtyping. Union, |, can be thought of as generalization or inducing a supertype.

The following postfix operators take types as a left operand.

|  |  |
| --- | --- |
| Operator | Return |
| ? | Type |
| {Type +} {Type \*} | Type |
| {Type #m..n} | Type |

? is a postfix operator that adds null to the value space of its operand. T? is equivalent to

T | { null }

The multiplicities lift a type to a collection of that type with the appropriate cardinality. For example:

{Date\*} // A collection of any number of dates

{Person+} // A collection of one or more people

{Wheel#2..4} // A collection of two to four wheels

## Intrinsic Types

The following table lists the intrinsic types that are defined as part of the M Language Specification:

|  |  |  |
| --- | --- | --- |
| **Type** | **Super Type** | **Description**  |
| Any |  | All values. |
| Binary | General | A sequence of binary octets. |
| Collection | Any | An unordered group of (potentially duplicate) values. |
| Date | General | A calendar date. |
| DateTime | General | A calendar date and time of day. |
| DateTimeOffset | General | A calendar date, time of day and time zone. |
| Decimal | Number | A fixed-point or exact number. |
| Entity | Any | A collection of labeled values. |
| General | Any | All simple values. |
| Guid | General | A globally unique identifier. |
| Integer | Decimal | A signed, integral value.  |
| List | Collection | An ordered sequence of values. |
| Logical | General | A logical flag. |
| Null | Any | Contains the single value null. |
| Number | General | Any numeric value. |
| Scientific | Number | A floating-point or exact number. |
| Text | General | A sequence of Unicode characters. |
| Time | General | A time of day and time zone. |
| Unsigned | Integer | An unsigned, integral value.  |

### Any

All values are members of this type.

The following binary operations take Any as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| in, !in | Collection | Logical |

The in operator returns true if some member of the right operand is equal (==) to the left operand. The !in operator returns true if the in operator would return false.

### General

All values that are not members of Entity or Collection (or null) are members of this type. It has no additional operators beyond those defined on Any.

#### Members

General types have one member ToText which returns a default text representation of the value.

ToText() : Text;

Examples:

2.ToText

"Hello".ToText

True.ToText

### Number

Number is an abstract type with four subtypes enumerated below. Each of these subtypes is further refined to a type with a precision. A type of a smaller precision may always be converted to the same type of a larger precision. Converting from a larger precision to a smaller precision tests for overflow at runtime.

The arithmetic operations (+, -, \*, /, %) defined below are specialized to return the most specific type of its operands (e.g. Integer8 + Integer8 returns Integer8, Decimal9 + Decimal38 returns Decimal38)

|  |  |
| --- | --- |
| Type | Precision |
| Integer | Integer8 |
| Integer16 |
| Integer32 |
| Integer64 |
| Unsigned | Unsigned8 |
| Unsigned16 |
| Unsigned32 |
| Unsigned64 |
| Decimal | Decimal9 |
| Decimal19 |
| Decimal28 |
| Decimal38 |
| Scientific | Single |
| Double |

#### Operators

The following unary operations take Number as a right operand.

|  |  |
| --- | --- |
| Operator | Return |
| +, - | Number |
| ! | Unsigned |

The - operator computes the numeric negation of its operand.

The ! operator computes the bitwise negation of its operand.

The following binary operations take Number as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| +, - | Number | Number |
| \*, /, % | Number | Number |
| >, <, <=, >=, ==, != | Number | Logical |
| &, |, ^ | Unsigned | Unsigned |

The following operations may cause underflow and overflow errors:

* The predefined unary - operator
* The predefined +, -, \*, and / binary operators
* Explicit numeric conversions from one Number type to another

The following operations may cause a divide-by-zero error:

* The predefined / and % binary operators

The bitwise and (&), bitwise exclusive or (^), and bitwise or (|) operators implicitly convert their operands to the same length. The smaller operand is padded with zeros on the left.

The precedence of the bitwise and, or, and exclusive or is lower than it is in many other languages.

#### AutoNumber

Unique numbers can be generated with the AutoNumber computed value. This is a special form for ensuring unique identities. Consider the following example:

type Person {

 Id : Integer32 => AutoNumber();

 Name : Text;

 Age : Integer32;

 Spouse : Person;

} where identity Id;

People : {Person\*};

Each instance of Person will receive an Id value that is unique for each extent that contains Person instances.

AutoNumber has a number of restrictions. The default value should not be overridden and AutoNumber may only be used on identity fields.

### Text

Text is a sequence of Unicode characters.

The following postfix operator takes Text as a left operand.

|  |  |
| --- | --- |
| Operator | Return |
| # | Unsigned |

The postfix # operator returns the count of characters in a Text string.

The following binary operations take Text as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| + | Text | Text |
| >, <, <=, >=, ==, != | Text | Logical |

The binary + operator concatenates two Text strings.

The relational operators perform a lexicographic comparison on the Text strings and return a Logical value.

#### Members

The following members are defined on Text.

Count() : Unsigned;

Like(pattern : Text) : Logical;

PatternIndex(pattern : Text) : Integer;

Count provides the number of characters in the text.

Like returns true if the input is matched by the pattern.

PatternIndex returns the starting position of the first match of the pattern in the text or -1 if the pattern is not found.

The pattern is of the following form:

syntax Pattern
 = PatternElement
 | Pattern PatternElement;

syntax PatternElement
 = NormalCharacter
 | "\_"
 | "%"
 | "[" NormalCharacter - NormalCharacter "]"
 | "[^" NormalCharacter - NormalCharacter "]"

This will be aligned with grammars.

Dash matches any single character. Percent matches zero or more characters. A character range matches any single character in the range. And an excluded character range matches any character not in the range.

#### Length declaration

Text declarations may take a single parameter to specify the maximum length of the text field. The expression

Text(N);

is equivalent to

Text where value.Count <= N;

The expression

Text(N,M)

is equivalent to

Text where value.Count >= N && value.Count <= M;

The expression

Text(N,null)

is equivalent to

Text where value.Count >= N;

This special form is specific to the Text type.

### Logical

The following unary operator takes Logical as an operand.

|  |  |
| --- | --- |
| Operator | Return |
| ! | Logical |

The following binary operations take Logical as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| &&, || | Logical | Logical |
| ==, != | Logical | Logical |

The following ternary operator takes Logical as a right operand.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Middle Operand | Left Operand | Return |
| ? : | Any | Any | Any |

### Guid

The following binary operations take Guid as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| ==, != | Guid | Logical |

Guids are created with the system defined NewGuid computed value.

NewGuid() : Guid;

### Date

The following binary operations take Date as a left operand:

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| + | Time | DateTime |
| >, <, <=, >=, ==, != | Date | Logical |

### DateTime

The following binary operations take DateTime as a left operand:

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| >, <, <=, >=, ==, != | DateTime | Logical |

### DateTimeOffset

The following binary operations take DateTimeOffset as a left operand:

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| >, <, <=, >=, ==, != | DateTimeOffset | Logical |

### Time

The following binary operations take Time as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| + | Date | DateTime |
| >, <, <=, >=, ==, != | Time | Logical |

## Entity

An EntityTypeExpression specifies the members for a set of entity values (commonly referred to as entities). Those members are fields.

Entity types are distinct from extents. The definition of an entity type does not imply allocation of storage. Storage is allocated when an extent of entity type is declared within a module.

The fields of an entity can be assigned default values and the values can be constrained with expressions. The names of all fields within an entity type declaration must be distinct.

### Declaration

The following syntax defines a collection of all possible instances that satisfy the structure and constraint.

syntax EntityTypeExpression
 = "{" EntityMemberDeclarations "}";

syntax EntityMemberDeclarations
 = EntityMemberDeclaration
 | EntityMemberDeclarations EntityMemberDeclaration;

syntax EntityMemberDeclaration
 = FieldDeclaration;

An entity type which through intersection or refinement results in two default values for the same named field is an error.

### Identity

The identity constraint controls the representation of identity. If it is specified the selected fields are used to represent the identity. If no identity constraint is specified, the entity cannot be referenced or compared.

The identity constraint may be specified either on entity declarations or on extent declarations. The identity constraint requires that the elements in the constraint are unique within each extent (not across extents) as with the unique constraint. An identity declaration on a derived type supersedes that of any types it derives from. As a result, there can be only one identity constraint on an entity or an extent.

Consider the following example:

type Container {

 Id : Integer32;

 Capacity: Integer32;

} where identity Id;

CoffeeCups : {Container\*} { {Id => 1, Capacity => 12} }

WaterBottles : {Container\*} { {Id => 1, Capacity => 12} }

EqualityTest() {

 from c in CoffeeCups

 from w in WaterBottles

 where c == w

 select "Never"

}

It is legal for the two extents to contain instances whose Id fields are equal. Having the same Id field does not equate the instances. The computed value EqualityTest will always return the empty collection because identity is relative to an extent.

An implementation of M may restrict the types of fields used to form identity.

### Operators

The following binary operations take Entity as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| ==, != | Entity | Logical |

The equality operations on entities compare identity (shallow equal). == returns true if both operands refer to an instance with the same identity in the same collection.

### Ascription

An entity defines a constraint over a set of values. An entity type can be ascribed to any value which satisfies its constraint. Ascribing an entity type allows the computed values defined in the entity to be applied to the value.

Consider the following two entities and two instances (the square root (SQRT) and absolute value (ABS) functions must be provided by a library, they are not intrinsic).

type PointOnPlane {

 X : Single;

 Y : Single;

 DistanceFromOrigin : Single { SQRT(X \* X + Y \* Y) }

}

type PointOnLine {

 X : Single;

 Y : Single;

 DistanceFromOrigin : Single { ABS(X) \* SQRT(2) }

} where X == Y;

Point1 => {X => 1, Y => 1};

Point2 => {X => 0, Y => 1};

Both entities define fields X and Y and a computed value DistanceFromOrigin although the implementation of the computed value differs. The first entity, PointOnPlane, allows any X,Y combination—the entire X,Y plane. The second entity, PointOnLine, has a constraint that restricts the values that can be members of the type.

Point1 is a member of both PointOnPlane and PointOnLine. Both declarations of DistanceFromOrigin are valid and yield the same result.

Point2 can be ascribed PointOnPlane, but not of PointOnLine since the constraint X == Y is not satisfied. This prevents the alternative declaration of DistanceFromOrigin from producing an incorrect result.

## Collections

Collections are unordered and may contain elements which are equal. M provides operators to construct strongly typed collections and in some cases defined below escalates members on elements to members on the collection.

### Declaration

New collection types are defined by a type constructor and a multiplicity ( +, \*, #m..n ).

syntax CollectionTypeExpression

 = "{" UnaryExpression "+" "}"
 | "{" UnaryExpression "\*" "}"
 | "{" UnaryExpression "#" IntegralRange "}";

syntax IntegralRange
 = IntegerLiteral
 | IntegerLiteral ".."
 | IntegerLiteral ".." IntegerLiteral;

|  |  |
| --- | --- |
| Type Expression | Multiplicity |
| {TypeReference \* } | Zero to Many |
| {TypeReference + } | One to Many |
| {TypeReference # N } | Exactly N |
| {TypeReference # Low .. High } | From the Low bound to High bound |
| {TypeReference # Low .. } | From the Low bound to any number |

The default value for a collection type is the empty collection, written {}. The one-to-many multiplicity constraint forbids an empty collection, so must have at least one member on initialization.

### Operators

The following postfix unary operator takes Collection as a left operand.

|  |  |
| --- | --- |
| Operator | Return |
| # | Unsigned |

The # operator returns the count of elements in the collection.

The following binary operations take Collection as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| >, <, <=, >=, ==, != | Collection | Logical |
| where | Logical | Collection |
| select | Any | Collection |
| &, | | Collection | Collection |

The relational operator >= returns true if the left operand has every element of the right operand with equal or greater multiplicity. The remainder of the operators are defined as follows:

A == B ~ A >= B && B >= A

A > B ~ A >= B && A != B

A != B ~ !(A == B)

A <= B ~ B >= A

A < B ~ B > A

The where operator returns a new collection containing only the elements from the left operand that satisfy the predicate on the right when evaluated on the iteration variable value. If the type of the left operand is {T\*} the type of the result will be {T\*}.

The select operator returns a new collection containing equal number of elements as the left operand that are the result of evaluating the expression on the right over the iteration variable value. If the type of the left operand is {T\*} and the result of evaluating the expression on the right is R, then the type of the result is {R\*}.

The intersect and union ( &, | ) convert their operands to sets and perform set intersection, and union respectively.

For the operators that return a collection, the inferred element type of the resulting collection is the most specific type which the elements of both operands may be converted to.

### Members

The following members are defined on all collections:

Choose() : Any;

Count() : Unsigned;

Distinct() : Collection;

Choose picks an arbitrary element from a collection. The return type is the element type. The result of calling Choose on an empty collection is undefined.

Count returns the total number of elements in a collection. The return type is a Number.

Distinct removes all duplicates in a collection. The return type is the same as the collection.

The following members are defined on collections of type {Logical\*}:

All() : Logical;

Exists() : Logical;

All returns false if false is an element of the collection and true otherwise. Exists returns true if true is an element of the collection and false otherwise.

The following members are defined on collections that are subtypes of {Number\*}:

Average() : Scientific;

Maximum() : Number;

Minimum() : Number;

Sum() : Number;

Maximum, Minimum, and Sum are specialized to return the element type of the collection.

Average computes the Sum of the collection and divides that by the Count.

Maximum returns the largest value in the collection.

Minimum returns the smallest value in the collection.

Sum returns the arithmetic summation of the values in the collection.

### Indexers

A collection may be accessed using language generated indexers of two kinds, selectors and projectors. A selector extracts members of a collection with a member that matches a value. A projector extracts all values of a field from a collection. Both of these operations can be accomplished with query expressions; however, this notation is more compact.

#### Selectors

The compiler will generate indexers for all fields of Person for {Person\*}.

Consider the following example:

type Person {

 Id : Integer64 => AutoNumber();

 Name : Text;

 HairColor : Text;

} where identity Id, unique Name;

People : {Person\*} {

 {Name => "Mary", HairColor => "Brown"},

 {Name => "John", HairColor => "Brown"},

 {Name => "Fritz", HairColor => "Blue"}

};

 Consider the following expressions:

People.Name("Mary")

evaluates to:

{{Name => "Mary", HairColor => "Brown" }}

People.Name("Bill")

evaluates to:

{}

People.HairColor("Brown")

evaluates to:

{

 {Name => "Mary", HairColor => "Brown"},

 {Name => "John", HairColor => "Brown"}

}

// Assuming the Fritz record was assigned the Id 123

People.Id(123)

evaluates to:

{{Name => "Fritz", HairColor => "Blue"}}

The expression:

Collection.MemberField(Expression)

is equivalent to:

from c in Collection

where c.MemberField == Expression

select c

The identity auto indexer is special in that it is also an indexer directly on the collection, so the following expression is legal:

People(123) == {{Name => "Fritz", HairColor => "Blue"}}

If the designer chose a different representation for identity, it would be the default indexer as shown in the following variant of the above example:

type Person {

 Name : Text;

 HairColor : Text;

} where identity Name;

People("Mary") == {{Name => "Mary", HairColor => "Brown" }}

Assuming the identity constraint for a collection is defined using the following pattern:

identity(IdField1, IdField2, ...)

the following expression

Collection (Expression1, Expression2, ...)

is equivalent to:

(from c in Collection

where c.IdentityField1 == Expression1, c.IdField2 == Expression2, ...

select c).Choose

which always returns at most 1 element in the collection.

#### Projectors

Projectors return the values 0f one field from each member of a collection.

Again, consider the following example:

type Person {

 Id : Integer64 => AutoNumber();

 Name : Text;

 HairColor : Text;

} where identity Id, unique Name;

People : {Person\*} {

 {Name => "Mary", HairColor => "Brown"},

 {Name => "John", HairColor => "Brown"},

 {Name => "Fritz", HairColor => "Blue"}

}

The following expressions all evaluate to true:

People.Name == {"Mary", "John", "Fritz"}

People.HairColor == {"Brown", "Brown", "Blue"}

People.HairColor.Distinct == {"Brown", "Blue"}

Note that the returned collection may have duplicates. To obtain a duplicate free collection, use Distinct.

The expression:

Collection.MemberField

is equivalent to:

from c in Collection
select c.MemberField

In the event that the identifier for the projector is equal to a member on collection, the projector is not added. Specifically, Choose, Count, and Distinct will not be added as projectors.

## Lists

Lists are ordered collections.

### Declaration

New collection types are defined by a type constructor and a multiplicity ( +, \*, #m..n ).

syntax CollectionTypeExpression

 = "[" UnaryExpression "+" "]"
 | "[" UnaryExpression "\*" "]"
 | "[" UnaryExpression "#" IntegralRange "]";

|  |  |
| --- | --- |
| Type Expression | Multiplicity |
| [TypeReference\*] | Zero-to-Many |
| [TypeReference+] | One-to-Many |
| [TypeReference#N] | Exactly N |
| [TypeReference#Low..High] | From the Low bound to High bound |
| [TypeReference#Low..] | From the Low bound to any number |

The default value for a collection type is the empty collection, written {}. The one-to-many multiplicity constraint forbids an empty collection, so must have at least one member on initialization.

### Operators

The following postfix unary operator takes List as a left operand.

|  |  |
| --- | --- |
| Operator | Return |
| # | Unsigned |

The # operator returns the count of elements in the list

The following binary operations take Collection as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| >, <, <=, >=, ==, != | List | Logical |
| where | Logical | List |
| select | Any | List |

The >= operator returns true if the right operand has an equal or greater number of elements than the left operand and the element at every position in the left operand is less than or equal to the element in the same position in the right operand. The remaining relational operators are defined using the axioms defined for collections.

This is a lexicographic ordering to make text work. It does not align with collection. e.g.
[4,5,6,7] >= [1,2,3]

The where and select operators on lists behave similarly to collections; however, the return type is a list rather than a collection and order is maintained.

For the operators that return a collection, the inferred element type of the resulting collection is the most specific type which the elements of both operands may be converted to.

## Null

Null is a type with a single value null. It is used in conjunction with other types to add null to the value space and make a nullable type. Nullable types can be specified with the postfix operator ? or with a union of the type and Null.

The type below has two nullable fields, SSN and Spouse.

type Person {

 Name : Text;

 SSN : Text?;

 Spouse : Person | Null;

}

Nullability is idempotent. T?? is the same as T? Collections cannot be made nullable therefore {T\*}? is not a legal type. Elements of collections can be nullable so {T? \*} is a legal type.

Except as noted, binary operations defined to take a left operand of T, right operand of S and return type of R are lifted to accept T?, S? and return R?. If either actual operand is null, the operation will return null. Logical operations && and || are not lifted.

The following binary operations take Null as a left operand.

|  |  |  |
| --- | --- | --- |
| Operator | Right Operand | Return |
| == != | Null | Logical |
| ?? | Any | Any |

The return type of ?? is specialized to the type for the left operand without the null value. The type of the right operand must be compatible with the type of the left operand.

The default value of type Null is null.

# Computed and Stored Values

M provides two primary means for values to come into existence: computed values and stored values (a.k.a. fields). Computed values occur within module declarations.

A computed value is derived from evaluating an expression. In contrast, a field stores a value and the contents may change over time[[1]](#footnote-1).

## Computed Value Declaration

A ComputedValueDeclaration binds a name to an expression that is used to compute the resultant value.

syntax ComputedValueDeclaration
 = Identifier FormalParameters ReturnTypeopt ExpressionBody
 | "extern" Identifer FormalParameters ReturnType ";";

syntax FormalParameters
 = "(" Parameters? ")";

syntax Parameters
 = Parameter
 | Parameters "," Parameter;

syntax Parameter
 = Identifier TypeAscription?;

syntax ReturnType
 = TypeAscription;

syntax ExpressionBody
 = "=>" Expression ";";

If the type of a parameter or the ReturnType is not explicitly specified, the parameter or return type will be implicitly inferred from the ExpressionBody. For example, the following two ComputedValueDeclarations have the same meaning:

Add(x : Integer32, y : Integer32) => x + y;

Add(x : Integer32, y : Integer32) : Integer32 => x + y ;

ComputedValueDeclaration introduces the formal parameters into scope. It is an error to have more than one formal parameter with the same Identifier.

### Overloading

A module may define multiple computed values with the same name. In this case, selection is determined based on the number of arguments. It is an error to define two computed values with the same name and the same number of arguments.

Within a module the names of computed values, extents and types must be disjoint.

## Fields

A field is a storage location. A field declaration specifies the name of the field and optionally a type ascription which constrains values in the field to be of the ascribed type. An initial value may be defined by either equating the field with an Expression or using an InitializationExpression (§9.3).

If an initializer is present without a TypeAscription, the type of the field is the type of the initializer. If both an initializer and TypeAscription are present, then the initializer must conform to the TypeAscription.

syntax FieldDeclaration
 = DottedIdentifer TypeAscription? "=>" Expression ";"
 | DottedIdentifier TypeAscription? InitializationExpression ";"?;
DottedIdentifers TypeAscription? ";"
 = "extern" Identifier TypeAscription ";";

syntax DottedIdentifiers
 | DottedIdentifier
 | DottedIdentifiers "," DottedIdentifier;

syntax DottedIdentifer
 = IdentifierPath
 | "." IdentifierPath;

syntax IdentifierPath
 | Identifier
 | IdentifierPath "." Identifier;

The only DottedIdentifer that is legal within an entity type declaration is a single Identifer.

A leading dot resolves the identifier within the top level module scope rather than the lexical scope.

## External Declarations

The extern feature allows an M author to declare the ‘shape’ or signature of some construct, while the implementation of a runtime not defined in M. Only module level fields (extents) and computed values may be marked extern.

The M language defines a keyword, extern which can be applied to computed value or extent declarations. When applied to a computed value declaration, that declaration must explicitly specify a return type and must not provide a body for the computed value. When applied to an extent declaration, that declaration must specify a type.

module Microsoft.Samples

{

 extern Add (x:Integer32, y:Integer32) : Integer32;

 extern HostName() : Text;

 extern People : {{

 Id : Integer32 => AutoNumber;

 Name : Text;

 Age : Integer16;

 }\*} where identity Id;

}

A declaration marked with extern can be referred to anywhere such a declaration could be referred to if it were not marked extern. That is, marking a declaration as extern does not affect where that declaration can be referred to in M. Unless an extern declaration is explicitly exported, it may only be used by the declaring module.

# Expressions

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

## Operators

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, \*, and /. Examples of operands include literals, fields, and expressions.

There are the following kinds of operators:

* Unary operators take one operand and use prefix notation such as –x.
* Binary operators take two operands and use infix notation such as x + y.
* Ternary operator. Only one ternary operator, ?:, exists; it takes three operands and uses infix notation (c? x: y).
* Query comprehensions

Precedence determines how operators are grouped with operands. Unless otherwise specified, the order of evaluation of operands is undefined.

A single syntactic operator may have different meanings depending on the type of its operands. That is, an operator may be overloaded. In this case, the meaning and the return type is determined by selecting the most specific super type for both operands for which a meaning and return type are specified in §2.

### Operator precedence and associativity

Precedence and associativity determine how operators and operands are grouped together. For example, the expression x + y \* z is evaluated as x + (y \* z) because the \* operator has higher precedence than +. The following table summarizes all operators in order of precedence from highest to lowest:

|  |  |
| --- | --- |
| Category | Operators |
| Primary | x.y  f(x) |
| Unary | +  -  !  # identity unique |
| Multiplicity (unary postfix) | ?  +  \*  # |
| Multiplicative | \*  /  % |
| Additive | +  - |
| Relational and type testing | <  >  <=  >=  in !in x:T |
| Equality | ==  != |
| Logical And (conjunction) | && |
| Logical Or (disjunction)  | || |
| Null Coalescing | ?? |
| Conditional | ?: |
| Query Comprehension | from join let where select group by accumulate |
| Where  | where |
| Select | select  |
| Bitwise And, Intersection | & |
| Bitwise Exclusive Or | ^ |
| Bitwise Or, Union | | |

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed. All binary operators are left associative, that is, operations are performed left to right. For example, x + y + z is evaluated as (x + y) + z.

Precedence and associativity can be controlled using parentheses. For example, x + y \* z first multiplies y by z and then adds the result to x, but (x + y) \* z first adds x and y and then multiplies the result by z.

## Member access

A MemberAccessExpression takes an expression which resolves to a scope as the left operand and a symbol as the right operand. Evaluating the expression returns the value bound to the symbol in the scope.

syntax MemberAccessExpression
 = PrimaryExpression "." MemberName;

syntax MemberName
 = Identifier;

A MemberAccessExpression consists of a PrimaryExpression, followed by a "." token, followed by a member selector. Consider the following MemberAccessExpression:

{Name => "Bill", Age => 23}.Age

The member access operator looks up the symbol Age in the scope defined by the instance:

{Name => "Bill", Age => 23}

### Symbol lookup

A symbol lookup is the process whereby the meaning of a name in a context is determined. A symbol lookup may occur as part of evaluating a SimpleName or a MemberAccess in an expression.

M is a lexically scoped language. Scopes introduce symbols and may nest and an inner scope may introduce a symbol which hides a symbol in an outer scope. Initially a symbol is resolved against the lexically innermost scope. If no matching symbol is found in the innermost scope, lookup proceeds in the containing scope. This process continues until the outermost scope is reached which is always a module.

The following are examples of scopes:

* An entity definition.
* A module.
* A field definition.
* The left hand side of a where expression.
* A query expression.

A member lookup of a name N in a type T is processed as follows: The set of all accessible members named N declared in T and the base types of T is constructed. If no members named N exist and are accessible, then the lookup produces no match.

Field declarations override lexical scoping to prevent the type of a declaration binding to the declaration itself. The ascribed type of a field declaration must not be the declaration itself; however, the declaration may be used in a constraint. Consider the following example:

type A;

type B {

 A : A;

}

The lexically enclosing scope for the type ascription of the field declaration A is the entity declaration B. With no exception, the type ascription A would bind to the field declaration in a circular reference which is an error. The exception allows lexical lookup to skip the field declaration in this case.

A declaration may be used within a constraint on the ascribed type as in the following example:

type Node {

 Label : Text;

 Parent : Node;

}

Nodes : {(Node where value.Parent in Nodes)\*};

The right operand of the in clause stipulates that the Parent field of a node must be within the collection being defined, Nodes.

## Initializers

This needs to be cleaned up to make entities, collections, and lists primitive.

Entity types and collections use a common initialization syntax.

An InitializationExpression constructs a new instance of a collection or entity.

syntax InitializationExpression
 = "{" Elements? "}"

 | "[" Elements? "]"

 | Kind? "{" Properties "}";

syntax Elements

 = Expression ","?

 | Elements "," Expression;

syntax Properties

 = Property ","?

 | Properties "," Property;

syntax Kind

 = Identifier;

### Collection Initializer

The following are three examples of unordered collections:

{ 1, 2, 3, 4 }

{ 1, 1, 1, 1, }

{ [], "Hello", 24 }

### List Initializer

The following are three examples of lists:

[ "Red", "Blue", "Green"]

[]

[{}, 3, [], ]

### Record Initializer

The following are three examples of records:

{ Name => "John", Age => 24 }

{ Title => "Specifying Systems", Author => "Leslie Lamport", }

{ Colors => {"Red", "Blue", "Green"} }

### Kind Pattern

The kind pattern allows records to have a discriminator. It is a syntactic transformation that transforms the following pattern *Identifier* { . . . } into { Kind => "*Identifier*", . . .}.

Consider the following examples:

Person { Name => "John", Age => 24 }

Cat { Name => "Fluffy", Age => 2 }

Dog { Name => "Rover", Age => 8 }

These are equivalent to:

{ Kind => "Person", Name => "John", Age => 24 }

{ Kind => "Cat", Name => "Fluffy", Age => 2 }

{ Kind => "Dog", Name => "Rover", Age => 8 }

### Non Local Initialization

This section needs to be updated to new structures design and we should look at whether the feature is needed any longer.

It is frequently useful to initialize a value in another structure. In the following example computers have zero to many boards. This relationship is implemented as a reference from Board to Computer.

type Computer {

 Id : Integer32 => AutoNumber();

 Processor : Text;

} where identity Id;

type Board {

 Id : Integer32 => AutoNumber();

 Kind : Text;

 Computer : Computer;

} where identity Id;

Boards : {(Board where value.Computer in Computers)\*};

Computers : {Computer\*};

Creating an instance of a computer requires initializing both the computer and the boards. This can be initialized "bottom up" as follows:

Computers {

 MyPC { Processor => "x86"}

}

Boards {

 Graphics { Computer => Computers.MyPC },

 Sound { Computer => Computers.MyPC },

 Network { Computer => Computers.MyPC },

}

Using non local initialization, this same structure can be initialized "top down" as follows:

Computers {

 MyPC { Processor => "x86",

 .Boards {

 Graphics { Computer => MyPC },

 Sound { Computer => MyPC },

 Network { Computer => MyPC },

 }

 }

}

The dot prefix to the Boards label (.Boards) does not introduce a new label into the current scope. Rather, it looks up the symbol at the extent scope and adds the content to that extent.

## Invocation Expression

The Identifier in an InvocationExpression resolves to a computed value declaration of the same name and arity. Evaluating an invocation expression causes each argument to be evaluated. The result of each argument is bound to the formal parameter in the corresponding position. The result of evaluating the body of the computed value declaration is the value of the invocation expression.

syntax InvocationExpression
 = Identifier "(" Arguments? ")";

syntax Arguments
 = Argument
 | Arguments "," Argument;

syntax Argument
 = Expression;

## Primary expressions

The following rules define the grammar for primary expressions.

syntax PrimaryExpression
 = Literal
 | SimpleName
 | ParenthesizedExpression
 | MemberAccessExpression
 | InvocationExpression
 | InitializationExpression
 | EntityTypeExpression
 | ContextVariable
 | AboutExpression;

Literal is defined in §2.5. EntityTypeExpression is defined in §7.5. The remaining non terminals are defined in this section, §8.3.

### Simple names

A SimpleName consists of a single identifier.

syntax SimpleName
 = Identifier;

In the expression:

Person.Age

Both Person and Age are SimpleNames.

### Parenthesized expressions

A ParenthesizedExpression consists of an Expression enclosed in parentheses.

syntax ParenthesizedExpression
 = "(" Expression ")";

A ParenthesizedExpression is evaluated by evaluating the Expression within the parentheses.

### Context variable

The following rules define the grammar for context variables.

syntax ContextVariable
 = "value";

Value is defined in §9.14.1.

### About operator

The about is used to obtain the entry in Language.Catalog (§12) for a symbol. About is a special form, not a computed value. The symbol must be visible in the current scope and is resolved using normal symbol resolution rules. On evaluation, rather than returning the value the symbol represents, about returns a reference to the metadata for the symbol itself.

syntax AboutOperator
 = "about" "(" DottedIdentifer ")";

Examples:

module Test {

 import Language.Catalog;

 type Person {

 Name : Text;

 Age : Integer;

 }

 People : {Person\*};

 Children() { People where Age < 18 }

 Documentation : {{

 About : Language.Catalog.Declaration,

 Description : Text

 }\*};

 Documentation {

 {

 About => about(Person).Declaration,

 Description => "Describes people"

 },

 {

 About => about(People).Declaration,

 Description => "Contains people"

 },

 {

 About => about(Children).Declaration,

 Description => "Extracts young people"

 },

 }

}

The following table defines the classes of symbols that are resolved with about and the resulting entry in Language.Catalog.

|  |  |
| --- | --- |
| **Symbol** | **Language.Catalog**  |
| Computed Value | ComputedValueGroup |
| Extent | Extents |
| Module | Modules |
| Type | Types |

## Unary operators

The following rules define the grammar for unary operators.

syntax UnaryExpression
 = PrimaryExpression
 | "+" PrimaryExpression
 | "-" PrimaryExpression
 | "!" PrimaryExpression
 | PrimaryExpression "#"
 | IdentityExpression
 | UniqueExpression;

syntax IdentityExpression
 = identity Identifier
 | identity "(" Identifiers ")";

syntax UniqueExpression
 = unique Identifier
 | unique "(" Identifiers ")";

The identity constraint is discussed in §7.5.2.

The type rules for unary operators are defined in Number §7.4.3.1, Logical §7.4.5, and Collection §7.6.2.

Examples of unary operators follow:

+1

-2

!true

!0x00

{1,2,3}#

identity Id

unique Name

## Multiplicity

The following rules define the grammar for multiplicity operators.

syntax MultiplicityExpression
 = UnaryExpression
 | UnaryExpression "?"
 | CollectionTypeExpression

 | ListTypeExpression;

The type rules for multiplicity operators are defined in type operators §7.3.

Examples of multiplicity expressions follow:

Integer32?

[Text#2..4]

{{Name : Text; Age : Integer32}\*}

## Arithmetic operators

The following rules define the grammar for arithmetic operators.

syntax AdditiveExpression
 = MultiplicativeExpression
 | AdditiveExpression "+" MultiplicativeExpression
 | AdditiveExpression "-" MultiplicativeExpression;

syntax MultiplicativeExpression
 = UnaryExpression
 | MultiplicativeExpression "\*" MultiplicityExpression
 | MultiplicativeExpression "/" MultiplicityExpression
 | MultiplicativeExpression "%" MultiplicityExpression ;

The type rules on arithmetic operators are defined in Number §7.4.3, Text §7.4.4, Date §7.4.7, Time §7.4.8.

Examples of arithmetic operators follow:

1 + 1

2 \* 3

"Hello " + "World"

## Relational and type-testing operators

The following rules define the grammar for relational and type testing operators.

syntax RelationalExpression
 = AdditiveExpression
 | RelationalExpression "<" ShiftExpression
 | RelationalExpression ">" ShiftExpression
 | RelationalExpression "<=" ShiftExpression
 | RelationalExpression ">=" ShiftExpression
 | RelationalExpression "in" ShiftExpression
 | RelationalExpression ":" ShiftExpression;

The type rules on relational and type-testing operators are throughout §2.

## Equality operators

The following rules define the grammar for equality operators.

syntax EqualityExpression
 = RelationalExpression
 | EqualityExpression "==" RelationalExpression
 | EqualityExpression "!=" RelationalExpression;

The type rules on equality operators are throughout §2.

## Logical operators

The following rules define the grammar for logical operators.

syntax LogicalAndExpression
 = EqualityExpression
 | LogicalAndExpression "&&" EqualityExpression;

syntax LogicalOrExpression:
 = LogicalAndExpression
 | LogicalOrExpression "||" LogicalAndExpression;

The type rules on logical operators are defined in Logical §7.4.5.

## Conditional operators

There are two conditional operators coalesce and conditional.

### Coalescing operator

The ?? operator is called the null coalescing operator.

syntax NullCoalescingExpression
 = LogicalOrExpression
 | LogicalOrExpression "??" NullCoalescingExpression;

A null coalescing expression of the form a ?? b requires a to be nullable. If a is not null, the result of a ?? b is a; otherwise, the result is b. The operation evaluates b only if a is null.

b must be of the same type as a without the value null.

### Conditional operator

The ?: operator is called the conditional operator. It is at times also called the ternary operator.

syntax ConditionalExpression:
 = NullCoalescingExpression
 | NullCoalescingExpression "?" Expression ":" Expression;

A conditional expression of the form b ? x : y first evaluates the condition b. Then, if b is true, x is evaluated and becomes the result of the operation. Otherwise, y is evaluated and becomes the result of the operation. A conditional expression never evaluates both x and y.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form a ? b : c ? d : e is evaluated as a ? b : (c ? d : e).

The first operand of the ?: operator must be an expression of a type that can be implicitly converted to Logical otherwise a compile-time error occurs. The middle and left operands must be of compatible types. The result of the conditional is the least specific type.

## Query expressions

Query expressions provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as Transact SQL and XQuery.

A query expression begins with a from clause and ends with either a select, group or accumulate clause. The initial from clause can be followed by zero or more from, let, or where clauses. Each from, let and join clause introduces a scope that adds one iteration identifier that is in scope for the remainder of the query expression.

syntax QueryExpression
 = ConditionalExpression
 | QueryFromClause QueryBody;

syntax QueryBody
 = QueryBodyClauses? QueryConstructor;

syntax QueryBodyClauses
 = QueryBodyClause
 | QueryBodyClauses QueryBodyClause;

syntax QueryBodyClause
 = QueryFromClause
 | QueryLetClause
 | QueryWhereClause
 | QueryJoinClause;

syntax QueryConstructor
 = QuerySelectClause
 | QueryGroupClause
 | QueryAccumulateClause;

syntax QueryFromClause
 = "from" Identifier "in" ConditionalExpression;

syntax QueryLetClause
 = "let" Identifier "=" ConditionalExpression;

syntax QueryJoinClause
 = "join" Identifier "in" Expression "on" Expression "equals"

 ConditionalExpression;

syntax QueryWhereClause
 = "where" ConditionalExpression;

syntax QuerySelectClause
 = "select" ConditionalExpression;

QueryGroupClause
 = "group" Expression "by" ConditionalExpression;

QueryAccumulateClause
 = QueryLetClause "accumulate" ConditionalExpression;

### From clause

The from clause introduces an identifier that ranges over a collection. The following query body is interpreted once for each element in the collection with the identifier bound to that element. Thus if there are two from clauses in sequence, the query body following the second will be interpreted for the first element of the first collection and every element of the second collection, then for the second element of the first collection and every element of the second collection, etc. This ordering will be respected for ordered collections (lists). For unordered collections the elements may be processed in any order.

The following are examples of the from clause:

from n in {1,2,3,4,5}

select n

from n1 in {1,2,3,4,5}

from n2 in {1,2,3,4,5}

select n1\*n2

from p in People

select { Name => p.Name, Age => p.Age }

### Where clause

A where clause contains a predicate that filters further interpretation within a query expression. The expression

where true

has no effect.

The expression

where false

blocks further interpretation. The query expression will always return the empty set.

The following are examples of the where clause:

from n in {1, 2, 3, 4, 5}

where n%2 == 0

select n

from p in People

where p.Age > 17

select p

from n1 in {1,2,3,4,5}

from n2 in {1,2,3,4,5}

where n1 != n2

select n1\*n2

### Select clause

The select clause is a query constructor and determines results of the query expression. Consider a select clause of the following form:

select Expression

Where expression has type T. The type of the query expression will be [T\*] if all collections in the from and join expressions are ordered and will be {T\*} otherwise.

The following are examples of the select clause:

from n in {1,2,3,4,5}

select n%2

from p in People

select p.Name

from p in People

select {Name => p.Name, Age => p.Age}

### Join clause

The join clause is a compact syntax for a common pattern. The expression

join x in Collection on Expression1 equals Expression2

is equivalent to

from x in Collection

where Expression1 == Expression2

The following are examples of the join clause:

from n1 in {1,2,3,4,5}

join n2 in {1,2,3,4,5} on n1 equals n2

select n1\*n2

from c in Customers

join o in Orders on c.Id equals o.CustomerId

select { Customer => c.Name, OrderedOn => o.Submitted }

from a in Actors

join r in Roles on a.Id equals r.ActorId

join m in Moves on m.Id equals r.MovieId

select { Actor => a.FirstName + " " + a.LastName, Movie => m.Title }

### Let clause

The let clause is a compact syntax for a common pattern. The expression

let x = Expression

is equivalent to

from x in {Expression}

The following are examples of the let clause:

from n in {1,2,3,4,5}

let pi = 3.1415

select { Radius => n, Area => n\*n\*pi }

from p in People

let FullName = p.First + " " + p.Last

select { Name => FullName, Age => p.Age }

from a in Actors

let FullName = a.FirstName + " " + a.LastName

join r in Roles on a.Id equals r.ActorId

join m in Moves on m.Id equals r.MovieId

select { Actor => FullName, Movie => m.Title }

### Group By clause

The group by clause is a compact syntax for a common pattern. The expression:

from c in Collection

group c by F(c)

is equivalent to

from key in (Collection select F(value)).Distinct

select { Key => key, Value => (Collection where key == F(value)) }

The following are examples of the group by clause:

from n in {1,2,3,4,5}

group n by n%2

from p in People

group p by p.Age

from c in Cars

group c by c.Color

### Accumulate clause

The accumulate keyword generalizes Sum, Min, Max et cetera. Its purpose is to repeatedly apply an expression to each element in a collection and accumulate the result.

The type of the expression in the let clause and the expression in the accumulate clause must be compatible. This type is the type of the expression.

Consider the following fragment:

from c in Collection

let a = Expression1

accumulate Expression2

Expression1 is evaluated and the result is bound to the identifier a. Expression2 is evaluated once for every element of Collection. Upon each evaluation the result is bound to the identifier a.

As an example, the following M code sums the elements in the collection Numbers:

from n in Numbers

let i = 0

accumulate i + n

The following computes minimum of a collection of {Integer32\*}:

from n in Integers

let i = MaxInteger32

accumulate i < n ? i : n

The following computes the maximum of a collection of {Integer32\*}:

from n in Integers

let i = MinInteger32

accumulate i > n ? i : n

The following returns false if a collection contains false and true otherwise:

from b in TruthValues

let r = true

accumulate b && r

The following returns true if a collection contains true and false otherwise:

from b in TruthValues

let r = false

accumulate b || r

## Compact query expressions

There are two compact forms for query expressions the binary infix where and select.

### Where operator

The infix where operation filters elements from a collection that match a predicate.

syntax WhereExpression
 = QueryExpression
 | QueryExpression "where" WhereExpressions;

syntax WhereExpressions
 = WhereExpression
 | WhereExpressions "," WhereExpression;

The WhereExpression introduces the identifier value into the scope of the right hand side to refer to an element of the collection on the left. The right hand side may also use any other identifiers that are in lexical scope.

The following example uses value to filter the Numbers collection:

OneToTen : Number where value > 0 && value <= 10;

When used over a collection type, value refers to the collection:

SmallCollection : {Number\*} where value.Count == 2;

SmallCollectionOneToTen : {(Number where value > 0 && value <=10)\*}

 where value.Count < 10;

Formalizing this convention:

*QueryExpression* where *Expression*

Is a compact syntax for the following expression:

from value in *QueryExpression*

where *Expression*

select value

### Select operator

The select operator applies an expression to every element in a collection and returns the results in a new collection.

syntax SelectExpression
 = WhereExpression
 | WhereExpression "select" Expression;

The SelectExpression introduces the identifier value into the scope of the right hand side to refer to an element of the collection on the left. The right hand side may also use any other identifiers that are in lexical scope.

Examples of the select operator follow:

{1, 2, 3} select value \* 2

People select { Name => value.Name, Age => value.Age }

{{}, {1}, {1,1}} select value#

The select operator can be rewritten to a full query expression:

*Collection* select *Expression*

Is equivalent to:

from value in *Collection*

select *Expression*

## Binary and Collection operators

The following rules define the grammar for binary and collection operators.

syntax InclusiveOrExpression
 = ExclusiveOrExpression
 | InclusiveOrExpression "|" ExclusiveOrExpression;

syntax ExclusiveOrExpression
 = AndExpression
 | ExclusiveOrExpression "^" AndExpression;

syntax AndExpression
 = SelectExpression
 | AndExpression "&" SelectExpression;

The type rules on binary and collection operators are defined in Number §7.4.3, and Collection §7.6.2.

## Expressions

An expression is a sequence of operands and operators. Applying the operator to the operand yields a value.

syntax Expression
 = InclusiveOrExpression;

# Module

A module is a scope which contains declarations of types (§2), extents (§8.2), and computed values (§8.1). Modules override lexical scoping to import symbols which have been exported from another module.

## Compilation Unit

Several modules may be contained within a CompilationUnit, typically a text file.

syntax CompilationUnit
 = ModuleDeclarationList;

syntax ModuleDeclarationList
 = ModuleDeclaration
 | ModuleDeclarationList ModuleDeclaration;

## Module Declaration

A ModuleDeclaration is a named container/scope for type declarations, field declarations, and computed value declarations.

syntax ModuleDeclaration
 = "module" QualifiedIdentifer ModuleBody ";"?;

syntax QualifiedIdentifier
 = Identifier
 | QualifiedIdentifier "." Identifier;

syntax ModuleBody
 = "{" ImportDirectives? ExportDirectives? ModuleMemberDeclarations? "}";

syntax ModuleMemberDeclarations
 = ModuleDeclaration

 | ModuleMemberDeclarations ModuleMemberDeclaration;

syntax ModuleMemberDeclaration
 = LanguageDeclaration
 | FieldDeclaration
 | ComputedValueDeclaration
 | TypeDeclaration;

Each ModuleDeclaration has a QualifiedIdentifier that uniquely qualifies the declarations contained by the module.

Each ModuleMemberDeclaration may be referenced either by its Identifier or by its fully qualified name by concatenating the QualifiedIdentifier of the ModuleDeclaration with the Identifier of the ModuleMemberDeclaration (separated by a period).

For example, given the following ModuleDeclaration:

module PeopleData {

 Names : {Text\*};

}

The fully qualified name of the field is PeopleData.Names, or using escaped identifiers, 'PeopleData'.'Names'. It is always legal to use a fully qualified name where the name of a declaration is expected.

Modules are not hierarchical or nested. That is, there is no implied relationship between modules whose QualifiedIdentifier share a common prefix.

For example, consider these two declarations:

module A {

 N : Number;

}

module A.B {

 NPlusOne : { N + 1 }

}

Module A.B is in error, as it does not contain a declaration for the identifier N. That is, the members of Module A are not implicitly imported into Module A.B.

## Inter-Module Dependencies

M uses ImportDirectives and ExportDirectives to explicitly control which declarations may be used across module boundaries.

syntax ExportDirectives
 = ExportDirective
 | ExportDirectives ExportDirective;

syntax ExportDirective
 = "export" Identifiers;

syntax ImportDirectives
 = ImportDirective
 | ImportDirectives ImportDirective;

ImportDirective
 = "import" ImportModules ";"
 | "import" QualifiedIdentifier "{" ImportMembers "}" ";";

syntax ImportMember
 = Identifier ImportAlias?

syntax ImportMembers
 = ImportMember
 | ImportMembers "," ImportMember;

syntax ImportModule
 = QualifiedIdentifier ImportAlias?;

syntax ImportModules
 = ImportModule
 | ImportModules "," ImportModule;

syntax ImportAlias
 = "as" Identifier;

A ModuleDeclaration contains zero or more ExportDirectives, each of which makes a ModuleMemberDeclaration available to declarations outside of the current module.

A ModuleDeclaration contains zero or more ImportDirectives, each of which names a ModuleDeclaration whose declarations may be referenced by the current module.

A ModuleMemberDeclaration may only reference declarations in the current module and declarations that have an explicit ImportDirective in the current module.

An ImportDirective is not transitive, that is, importing module A does not import the modules that A imports.

For example, consider this ModuleDeclaration:

module People.Types {

 export Person;

 SecretNumber : Number;

 type Person { FirstName : Text; Age : Number; }

 People : {Person\*};

}

The field People.Types.SecretNumber may only be referenced from within the module People.Types. The type People.Types.Person may be referenced in any module that has an ImportDirective for module People.Types, as shown in this example:

module People.Data {

 import People.Types;

 export Names;

 Names : {Text\*};

 Friends : {People.Types.Person\*};

}

The example above used the fully qualified name to refer to People.Types.Person. A qualified name beginning with an imported module name refers to the object in the imported module and is not a new definition. For example, People.Types.People below references the previously declared extent. It does not declare a new extent.

module People.Data {

 import People.Types;

 People.Types.People {

 { Name => "Mary", Age => 23 },

 { Name => "Joe", Age => 32 },

 }

}

An ImportDirective may also specify an ImportAlias that provides a replacement Identifier for the imported declaration:

module People.Data {

 import People.Types as pt;

 export Names;

 Names : {Text\*};

 Friends : {pt.Person\*};

}

An ImportAlias replaces the name of the imported declaration. That means that the following is an error:

module People.Data {

 import People.Types as pt;

 export Names;

 Names : {Text\*};

 Friends : {People.Types.Person\*};

}

It is legal for two or more ImportDirectives to import the same declaration, provided they specify distinct aliases. For a given module, at most one ImportDirective may use a given alias.

If an ImportDirective imports a module without specifying an alias, the declarations in the imported module may be referenced without the qualification of the module name. That means the following is also legal:

module People.Data {

 import People.Types;

 export Names;

 Names : {Text\*};

 Friends : {Person\*};

}

When two modules contain same-named declarations, there is a potential for ambiguity. The potential for ambiguity is not an error – ambiguity errors are detected lazily as part of resolving references.

Consider the following two modules:

module A {

 export X;

 X : Number;

}

module B {

 export X;

 X : Number;

}

It is legal to import both modules either with or without providing an alias:

module C {

 import A, B;

 Y { 1 + 2 }

}

This is legal because ambiguity is only an error for references, not declarations. That means that the following is a compile-time error:

module C {

 import A, B;

 Y { X + 2 } // error: unqualified identifier X is ambiguous

}

This example can be made legal either by fully qualifying the reference to X:

module C {

 import A, B;

 Y { A.X + 2 } // no error

}

or by adding an alias to one or both of the ImportDirectives:

module C {

 import A;

 import B as bb;

 Y { X + 2 } // no error, refers to A.X

 Z { bb.X + 2 } // no error, refers to B.X

}

Because module names may contain periods, there is a potential ambiguity when module names share a common prefix. Consider these two modules:

module A {

 export Z, B;

 type Z { C : Number; }

 B : Z;

}

module A.B {

 export C;

 C : Number;

}

If a module imports both of these modules, the QualifiedIdentifier A.B.C is inherently ambiguous, as it could either refer to the C field in module A.B or to the C field of the B field of module A. To disambiguate, one must use an alias to break the tie:

module F {

 import A;

 import A.B as ab;

 G { ab.C } // returns the C field of module A.B

 H { A.B.C } // returns the C field of the B field of module A

}

An ImportDirective may either import all exported declarations from a module or only one of them. The latter is enabled by specifying an ImportMember as part of the directive. For example, Module Plot2D imports only Point2D and PointPolar from the module Geometry:

module Geometry {

 export Point2D, Point2DPolar, Point3D;

 type Point2D { X : Number; Y : Number; }

 type Point2DPolar { R : Number; T : Number; }

 type Point3D : Point2D { Z : Number; }

}

module Plot2D {

 import Geometry {Point2D, Point2DPolar};

 Points : {Point2D\*};

 PointsPolar : {Point2DPolar\*};

}

An ImportDirective that contains an ImportMember only imports the named declarations from that module. This means that the following is a compilation error because module Plot3D references Point3D which is not imported from module Geometry:

module Plot3D {

 import Geometry {Point2D};

 Points : {Point3D\*};

}

An ImportDirective that contains an ImportTarget and an ImportAlias assign the replacement name to the imported type, field, or computed value declaration.

## Compilation Episode

Multiple compilation units may contribute declarations to a module of the same name.

The types and computed values of a module are sealed by a compilation episode. A subsequent compilation episode may not contribute additional types or computed values. Initial values for module level field declarations may be contributed in subsequent compilation episodes.

Each fragment must explicitly import the symbols used within that fragment and may only export symbols defined within that fragment.

## Storage

All dynamic storage in M is modeled as module-scoped FieldDeclarations called extents. The declaration of an extent may be spread across multiple sections of program text. Consider the following example:

// catalog.m

module Catalog {

    type Product {

        Name : Text;

        Price : Decimal9;

        Product(Name,Price);

    }

    Products : {Product\*};

}

// groceries.m

module Catalog {

    Products {

        Product("Soap", 1.29),

        Product("Tuna", 2.49)

    }

}

// hardware.m

module Catalog {

    Products {

        Product("Lightbulb", 0.99),

        Product("Screwdriver", 5.99)

    }

}

The resulting Products extent will contain:

{

    Product("Soap", 1.29),

    Product("Tuna", 2.49),

    Product("Lightbulb", 0.99),

    Product("Screwdriver", 5.99)

}

The mapping of module-scoped FieldDeclarations to physical storage is implementation-specific and outside the scope of this specification.

# Attributes

Attributes provide metadata which can be used to interpret the language feature they modify.

syntax AttributeSections
 = AttributeSection
 | AttributeSections AttributeSection;

syntax AttributeSection
 = "@{" Nodes "}";

## Case Insensitive

The CaseSensitive attribute controls whether tokens are matched with our without case sensitivity. The default value is true. The following language recognizes "Hello World", "HELLO World", and "hELLO WorLD".

module HelloWorld {

 @{CaseInsensitive[true]}

 language HelloWorld {

 syntax Main

 = Hello World;

 token Hello

 = "Hello";

 token World

 = "World";

 interleave Whitespace

 = " ";

 }

}

## Classification

The classification annotation drives colorization in language services processors. The following classifications are supported:

@{Classificaton["ClassificationKind"]}

syntax ClassificationKinds
 = Comment
 | Delimiter
 | Identifier
 | Keyword
 | Literal
 | Operator
 | String
 | Text
 | Unknown
 | Whitespace;

An example follows:

@{Classification["Operator"]} token Plus = "+";

@{Classificaiton["Keyword"]} token ModuleToken = "module";

@{Classificaiton["Delimiter"]} token Comma = ",";

## Format Scope

@{FormatScope["Open"]}

@{FormatScope["Close"]}

## Nest

@{Nest}

# Catalog

module Language.Catalog

{

 export CollectionTypes;

 export ComputedValues;

 export ConstantKinds;

 export Constants;

 export Declarations;

 export EntityTypes;

 export ExpressionKinds;

 export Expressions;

 export Extents;

 export Fields;

 export IdentityFields;

 export Imports;

 export IntrinsicTypes;

 export Languages;

 export MembershipConstraints;

 export Parameters;

 export Types;

 export UniqueConstraints;

 CollectionTypes : {{

 Base : Types;

 MinimumCount : Integer64 => 0;

 MaximumCount : Integer64? => null;

 ElementType : Types;

 }\*} where identity(Base);

 ComputedValueGroups : {{

 Declaration : Declarations;

 }\*} where identity(Declaration);

 ComputedValues : {{

 Id : Integer32 => AutoNumber;

 ComputedValueGroup : ComputedValueGroups;

 ReturnType : Types;

 Body : Text?;

 }\*} where identity(Id);

 Constants : {{

 Expression : Expressions;

 ConstantKind : ConstantKinds;

 Value : General;

 }\*} where identity(Expression);

 type ConstantKinds : Text where value in {

 "Null","Binary","DateTime","DateTimeOffset","Decimal","Guid","Integer",

 "Logical","Scientific","Text","Time","Unsigned" };

 // Root of the things M authors can declare (types, extents, computed values,languages)

 Declarations : {{

 Id : Integer32 => AutoNumber;

 Name : Text;

 DefinedIn : Modules;

 HasSyntheticName : Logical => false;

 Exported : Logical => false;

 Kind : Text where value in { "Type", "Extent", "ComputedValue", "Language" };

 }\*} where identity(Id);

 EntityTypes : {{

 Base : Types;

 Nullable : Logical => false;

 }\*} where identity(Base);

 Extents : {{

 Declaration : Declarations;

 ExtentType : Types;

 }\*} where identity(Declaration);

 Expressions : {{

 Id : Integer32 => AutoNumber;

 ExpressionType : Types;

 ExpressionKind : ExpressionKinds;

 }\*} where identity(Id);

 type ExpressionKinds : Text where value in {

 "Coalesce","Conditional","Constant","GraphExpressionSymbolo","Invocation",

 "MemberAccess","SelectExpressionSymbol","TypeAscription","TypeReference",

 "VariableReference","WhereExpressionSymbol","FromExpressionSymbol",

 "IntoExpressionSymbol","JoinExpressionSymbol","LetExpressionSymbol"

 };

 Fields : {{

 Id : Integer32 => AutoNumber;

 Name : Text;

 Position : Integer16; // Position of field in entity declaration

 FieldType : Types; // Type of this field

 AutoGenerated : (Text where value in { "Identity", "IdSequence", "NewGuid" })?;

 EntityType : EntityTypes; // EntityType that this field belongs to

 }\*} where identity(Id),unique(Position,EntityType);

 IdentityFields : {{

 Id : Integer32 => AutoNumber;

 EntityType : EntityTypes;

 Position : Integer16; // Position in the identity constraint declaration

 Field : Fields;

 }\*} where identity(Id),unique(EntityType,Position),unique(Field);

 Imports : {{

 Id : Integer32 => AutoNumber;

 ImportingModule : Modules;

 Kind : Text where value in { "Module", "Member" };

 }\*} where identity(Id);

 ModuleImports : {{

 Id : Imports;

 Module : Modules;

 } where identity(Id), value.Module.Id != value.Id.ImportingModule.Id \*};

 MemberImports : {{

 Id : Imports;

 Member : Declarations where value.HasSyntheticName == false;

 } where identity(Id), value.Member.DefinedIn.Id != value.Id.ImportingModule.Id \*};

 IntrinsicTypes : {{

 Base : Types;

 Nullable : Logical => false;

 // Min/Max are used to record length constraints on Text/Binary, range constraints on integer numerics, nonsensical for Logical, Guid

 Min : Integer64 => 0;

 Max : Integer64?;

 }\*} where identity(Base);

 Languages : {{

 Declaration : Declarations;

 }\*} where identity(Declaration);

 MembershipConstraints : {{

 Id : Integer32 => AutoNumber;

 Kind : Text where value in { "In", "Subset", "Superset" };

 TargetExtent : Extents;

 }\*} where identity(Id);

 Modules : {{

 Id : Integer32 => AutoNumber;

 Name : Text where value.Count <=400;

 }\*} where identity(Id),unique(Name);

 Parameters : {{

 Id : Integer32 => AutoNumber;

 ComputedValue : ComputedValues;

 Position : Integer16;

 Name : Text where value.Count <=400;

 ParameterType : Types;

 }\*} where identity(Id),unique(ComputedValue,Position),unique(ComputedValue,Name);

 Types : {{

 Declaration : Declarations;

 BuiltInType : Text; // Named of standard library type upon which this type is based.

 MembershipConstraint : MembershipConstraints?;

 ConstraintExpression : Text?;

 }\*} where identity(Declaration);

 UniqueConstraints : {{

 Id : Integer32 => AutoNumber;

 CollectionType : CollectionTypes;

 }\*} where identity(Id);

 UniqueFields : {{

 Id : Integer32 => AutoNumber;

 UniqueConstraint : UniqueConstraints;

 Position : Integer16; // Position in the unique constraint declaration

 Field : Fields;

 }\*} where identity(Id),unique(UniqueConstraint,Position);

}

# Standard Library

## Data types

### Binary

Binary is a list of Unsigned8.

type Binary : [Unsigned8\*];

# Glossary

|  |  |
| --- | --- |
| **abstract** | An abstract type describes properties common to many other types, but cannot itself be represented. |
| **alias** | An alias is a local name for a symbol defined in another module. An alias may be specified on an import. |
| **ambiguity** | An ambiguity results when the language processor cannot uniquely determine which of several constructors to run for a given segment of text because the segment is recognized by more than one production. The competing productions do not have to be in the same rule. Ambiguities are resolved by refactoring the grammar and applying precedence directives. |
| **apply** | A computed value is applied to its arguments to yield a new value. |
| **arity** | Arity is the number of arguments a computed value requires. |
| **ascribe** | Ascribing a type to a value requires that the value conform to the type and exposes the behaviors defined on that type. |
| **binding** | A pattern may be ascribed with a named variable scoped to the production. When that pattern matches a section of text, that section of text is bound to the variable for use the constructor.  |
| **collection** | A collection is a container for zero or more values. The values in a collection are called elements. The same value may occur in a collection multiple times but have no order. This is also known as a bag or multiset. |
| **comment** | A comment is a sequence of characters in a compilation unit that is excluded from semantic analysis. |
| **compilation unit** | A compilation unit is an ordered sequence of Unicode characters that conform to the lexical and grammatical requirements. |
| **concrete** | A concrete type can be represented. |
| **conflict** | A conflict results whenever the language processor faces a choice in what action to take on a section of text. There are two sorts of conflicts: shift-reduce and reduce-reduce. Not all conflicts result in ambiguities because some conflicts are resolved by applying precedence directives and some conflicts are local. A local conflict means that the language processor faces multiple choices on a section of text, but in a larger context the only one collection of choices recognizes the text. |
| **conform** | Text conforms to a language if the text matches the Main rule.  |
| **constraint** | A constraint is a logical test to determine if an element is a member of a collection. |
| **constructor** | A constructor defines the output for a production. |
| **context-free** | Context-free is a term from computer science to designate a class of languages. All languages in this class can be written with a single non-terminal on the left hand side of the rule and can be recognized with a push down automaton.  |
| **declaration** | A declaration binds a symbol to a type or computed value. |
| **default** | A default is a value to be used unless no other is specified. |
| **derived** | A derived type is constraint over some other type. |
| **dynamic** | A dynamic quality is any quality which is not static. |
| **element** | An element is a value in a collection. |
| **entity** | An entity is a set of labeled values. |
| **export** | An export makes a symbol visible outside the module in which it was declared in. It is not visible within another module unless it is imported. |
| **expression** | An expression is a sequence of Unicode characters that conforms to the Expression production in the grammar. In general it consists of an operator and one or more operands. Applying the operator to the operand yields a new value. |
| **extent** | An extent is a location which holds a value at the module scope. |
| **field** | A field is a location which holds a value at the entity scope. |
| **grammar** | A grammar is a set of rules which determine if a sequence of characters conform to a language. As used in the specification these rules are context-free. |
| **identifier** | An identifier is a symbolic name for use in a program. Type names, formal parameters, module names, and names of computed values are all examples of identifiers. |
| **identity** | Identity is a value that implies equality between entities. The identity value can be used as a surrogate for the entity itself as in a reference. |
| **import** | An import brings a symbol defined in another module into scope in the current module. An import may be aliased. |
| **initializer** | An initializer in an expression that returns a collection or entity value. |
| **instance** | An instance is a value that conforms to an entity type. |
| **interleave** | Designating a rule interleave separates the sections of text recognized by the rule from normal language processing. These are used to define insignificant whitespace and comment rules. |
| **intrinsic** | Intrinsic definitions are part of the language and cannot be added to or extended by a library. |
| **language** | A language is a collection of rules potentially with a designated Main rule. If the language has a Main rule, and a text value matches that rule, then the text value conforms to the language or is recognized by the language. In this case, the output of the rule is the output of the language. |
| **lexical** | A lexical rule determines if a sequence of characters conform to a language. As used in this specification lexical rules are regular expressions and ambiguity is resolved by taking the longest match. |
| **library** | A library is an informal name for a commonly used collection of declarations. |
| **literal** | A literal is a sequence of characters that represent a simple value. |
| **member** | A member is a field or computed value within an entity scope. |
| **module** | A module is a container of extents, type, language and computed value declarations. |
| **nominal type** | Values in a nominal type system names the types they are members of. Also, subtypes name their and supertypes. |
| **null** | The value null is a distinct value that is used as a place holder. |
| **operand** | An operand is an expression used as an argument to an operator. |
| **operator** | An operator is a character or sequence of characters that is used in an expression to denote a computed value. |
| **overloading** | Overloading is giving the same symbol multiple interpretations within the same scope. The only form of overloading in M is on the number of arguments to a computed value. |
| **parser** | A parser is a language processor. |
| **pattern** | A pattern occurs on the left side of a production and recognizes text and potentially binds the text to a variable. |
| **precedence** | Precedence is a statement of preference between two choices available to the language processor.  |
| **prefix** | A prefix is a section of a text value from the beginning of the value to an arbitrary point. Both the empty value and the whole text value are prefixes of a text value. |
| **pre-processing** | Pre-processing includes or excludes portions of a compilation unit as determined by preprocessing directives. |
| **production** | A production consists of a pattern and an optional constructor. |
| **program** | A program is a collection of compilation units which conform to the lexical, grammatical, and semantic rules of the language. |
| **projector** | A projector yields a new collection with one member from all the elements in a source collection. |
| **query** | A query is an expression which yields a new collection from one or more input collections. |
| **recognize** | A language recognizes a text value if it matches the Main rule. |
| **reduce-reduce** | A reduce-reduce conflict occurs if two rules match the same section of text. It may be resolved using precedence. |
| **reference** | A reference holds the identity value of another entity. |
| **regular** | Regular is a term from computer science for a class of languages. It is a subclass of the context free languages in which the rules are tail recursive and can be recognized by a finite-state automation. |
| **representation** | A representation is a storage format for a value. Typically a representation will restrict precision or stipulate an encoding. |
| **rule** | A rule is a collection of productions with an optional name. There are three forms of rules, syntax, token, and interleave. |
| **scope** | A scope is a container for symbol declarations. In M scopes are lexical which means they can be determined from the block structure of the language (unless otherwise specified). |
| **selector** | A selector yields a new collection containing all the elements of a source collection with a member equal to a value given as an argument. |
| **set** | A collection of distinct elements. |
| **shift-reduce** | A shift-reduce conflict occurs when the language processor can take different actions on two productions or within a single production. It may be resolved using precedence.  |
| **start rule** | The start rule is the first rule a language processor attempts to match when recognizing a text value. The start rule for M is named Main. |
| **static** | A static quality is any quality which can be determined by reviewing the source text alone. |
| **storage** | A location that holds a value (see field and extent). |
| **structural type** | A type whose membership is determined by satisfying a structural pattern and possibly other constraints. |
| **subtype** | A subtype, S, is a type that permits only values of some other type, T. |
| **supertype** | A super type, T, is a type that permits all of the values of some other type, S. |
| **symbol** | A symbol is an identifier in the source text. Symbols are defined within scopes. Modules control the visibility of symbols. |
| **syntax** | Syntax is used in two ways. In the general sense, syntax is the rules that define a language. In M, syntax is used to designate a rule which defines a context-free language. |
| **token** | Token is used to designate a rule which defines a regular language. It is also used to name the sections of a text value recognized by such a rule. |
| **type** | A type is a predicate over values that yields a collection. |
| **unique** | An element is unique within a collection if no other element is equal it. |
| **value** | Syntactically, a value is and expression constructed solely from literals and initializers. It contains no variable reverences, computed values or operators. Conceptually a value is any abstract notion that can be represented in this way. |
| **variable** | A variable is a symbolic name for value. |

1. M currently provides no language constructs for mutating the contents of a field. However, implementations are likely to provide out-of-band mechanism for update. [↑](#footnote-ref-1)