MGrammar Language Specification

February 11, 2009

1 Introduction

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 success of XML is evidence that there is significant demand for using text to represent information – this evidence is even more compelling considering the relatively poor readability of XML syntax and the decade-long challenge to make XML-based information easily accessible to programs and stores. The emergence of simpler technologies like JSON and the growing use of meta-programming facilities in Ruby to build textual domain specific languages (DSLs) such as Ruby on Rails or Rake speak to the desire for natural textual representations of information. However, even these technologies limit the expressiveness of the representation by relying on fixed formats to encode all information uniformly, resulting in text that has very few visual cues from the problem domain (much like XML).

The MGrammar Language (M_g) was created to enable information to be represented in a textual form that is tuned for both the problem domain and the target audience. The M_g 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_g 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. The data that results from M_g processing is compatible with M_g's sister language, The "Oslo" Modeling Language, "M", which provides a SQL-compatible schema and query language that can be used to further process the underlying information.

1.1 Language Basics

A M_g -based 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_g 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.

1.2 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_g 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_g 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_g 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_g 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.

1.3 Output

M_g 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 D 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_g 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_g and a couple examples follow:

- Text literals – `"ABC", 'ABC'`
- 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_g 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 productions:

```
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 [ "Red", " 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.

1.4 Modularity

All of the examples shown so far have been "loose M_g" that is taken out of context. To write a legal M_g document, all source text must appear in the context of a *module definition*. A module defines a top-level namespace for any languages that are defined.

Here is a simple module definition:

```
module Literals {
  // declare a language
  language Number {
    syntax Main = ('0'..'9')+;
  }
}
```

In this example, the module defines one language named `Literals.Number`.

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 MyLanguage1;

  language MyLanguage1 {
    syntax Main = HerLanguage.Options;
  }
  language MyLanguage2 {
    syntax Main = "x"+;
  }
}
```

Note that only `MyLanguage1` is visible to other modules. This makes the following definition of `HerModule` legal:

```
module HerModule {
  import MyModule; // declares MyLanguage1
  export HerLanguage;

  language HerLanguage {
    syntax Options = (('a'..'z')+ ('on'|'off'))*;
  }
  language Private { }
}
```

As this example shows, modules may have circular dependencies.

2 Lexical Structure

2.1 Programs

An M_g 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.

Conceptually speaking, a program is compiled using four steps:

1. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens. Lexical analysis evaluates and executes pre-processing directives.
2. Syntactic analysis, which translates the stream of tokens into an abstract syntax tree.
3. Semantic analysis, which resolves all symbols in the abstract syntax tree, type checks the structure and generates a semantic graph.
4. Code generation, which generates instructions from the semantic graph for some target runtime, producing an image.

Further tools may link images and load them into a runtime.

2.2 Grammars

This specification presents the syntax of the M_g programming language using two grammars. The lexical grammar defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The syntactic grammar defines how the tokens resulting from the lexical grammar are combined to form M_g programs.

2.2.1 Grammar notation

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

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

IdentifierVerbatim:
    `[` *IdentifierVerbatimCharacters* `]`

defines an *IdentifierVerbatim* to consist of the token "`[`", followed by *IdentifierVerbatimCharacters*, followed by the token "`]`".

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

DecimalDigits:
  DecimalDigit
  DecimalDigits DecimalDigit

defines *DecimalDigits* to either consist of a *DecimalDigit* or consist of *DecimalDigits* followed by a *DecimalDigit*. In other words, the definition is recursive and specifies that a decimal-digits list consists of one or more decimal digits.

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

DecimalLiteral:
  IntegerLiteral `.` *DecimalDigit* *DecimalDigits*$_{opt}$

is shorthand for:

DecimalLiteral:
  IntegerLiteral `.` *DecimalDigit*
  IntegerLiteral `.` *DecimalDigit* *DecimalDigits*

and defines a *DecimalLiteral* to consist of an *IntegerLiteral* followed by a ‘.’ a *DecimalDigit* and by optional *DecimalDigits*.

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

Sign: one of
  `+ -`

is shorthand for:

Sign:
  `+`
  `-`

Conversely, exclusions are designated with the phrase "none of". For example, the production

TextSimple: none of
  `"`
  `\`
  NewLineCharacter

permits all characters except ‘"’, ‘\’, and new line characters.

2.2.2 Lexical grammar

The lexical grammar of M_g is presented in 2.3. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form tokens, white space, and comments (§2.3.2).

Every source file in an M_g program must conform to the *Input* production of the lexical grammar.

2.2.3 Syntactic grammar

The syntactic grammar of M_g is presented in the chapters and appendices that follow this chapter. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form M_g programs.

Every source file in an M_g program must conform to the C*ompilationUnit* production of the syntactic grammar.

2.3 Lexical analysis

The *Input* production defines the lexical structure of an M_g source file. Each source file in an M_g program must conform to this lexical grammar production.

Input:
    InputSection$_{opt}$

InputSection:
    InputSectionPart
    InputSection InputSectionPart

InputSectionPart:
    InputElements$_{opt}$ NewLine

InputElements:
    InputElement
    InputElements InputElement

InputElement:
    Whitespace
    Comment
    Token

Four basic elements make up the lexical structure of an M_g source file: line terminators, white space, comments, and tokens. Of these basic elements, only tokens are significant in the syntactic grammar of an M_g program.

The lexical processing of an M_g source file consists of reducing the file into a sequence of tokens which becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, but otherwise these lexical elements have no impact on the syntactic structure of an M_g program.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence `//` is processed as the beginning of a single-line comment because that lexical element is longer than a single `/` token.

2.3.1 Line terminators

Line terminators divide the characters of an M_g source file into lines.

NewLine:
    NewLineCharacter
    `U+000D U+000A`

NewLineCharacter:
  `U+000A` *// Line Feed*
  `U+000D` *// Carriage Return*
  `U+0085` *// Next Line*
  `U+2028` *// Line Separator*
  `U+2029` *// Paragraph Separator*

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

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

2.3.2 Comments

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

Comment:
  CommentDelimited
  CommentLine

CommentDelimited:
  `/*` *CommentDelimitedContents$_{opt}$* `*/`

CommentDelimitedContent:
  `*` none of `/`

CommentDelimitedContents:
  CommentDelimitedContent
  CommentDelimitedContents CommentDelimitedContent

CommentLine:
  `//` *CommentLineContents$_{opt}$*

CommentLineContent: none of
  NewLineCharacter

CommentLineContents:
  CommentLineContent
  CommentLineContents CommentLineContent

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
// Logical literal
//
syntax LogicalLiteral
   = "true"
   | "false" ;
```

shows three single-line comments.

The example

```
/* This defines a
   Logical literal
*/
syntax LogicalLiteral
   = "true"
   | "false" ;
```

includes one delimited comment.

2.3.3 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.

Whitespace:
  WhitespaceCharacters

WhitespaceCharacter:
  `U+0009` *// Horizontal Tab*
  `U+000B` *// Vertical Tab*
  `U+000C` *// Form Feed*
  `U+0020` *// Space*
  NewLineCharacter

WhitespaceCharacters:
  WhitespaceCharacter
  WhitespaceCharacters WhitespaceCharacter

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

Token:
  Identifier
  Keyword
  Literal
  OperatorOrPunctuator

2.4.1 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 square brackets. It contains any sequence of `Text` literal characters.

Identifier:
 IdentifierBegin IdentifierCharacters$_{opt}$
 IdentifierVerbatim

IdentifierBegin:
 `_`
 Letter

IdentifierCharacter:
 IdentifierBegin
 `$`
 DecimalDigit

IdentifierCharacters:
 IdentifierCharacter
 IdentifierCharacters IdentifierCharacter

IdentifierVerbatim:
 `[` *IdentifierVerbatimCharacters* `]`

IdentifierVerbatimCharacter:
 none of `]`
 IdentifierVerbatimEscape

IdentifierVerbatimCharacters:
 IdentifierVerbatimCharacter
 IdentifierVerbatimCharacters IdentifierVerbatimCharacter

IdentifierVerbatimEscape:
 `\\`
 `\]`

Letter:
 `a..z`
 `A..Z`

DecimalDigit:
 `0..9`

DecimalDigits:
 DecimalDigit
 DecimalDigits DecimalDigit

2.4.2 Keywords

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when escaped with square brackets `[]`.

Keyword: oneof:

```
    any empty error export false final id import interleave language
    labelof left module null precedence right syntax token true valuesof
```

The following keywords are reserved for future use:

```
checkpoint identifier nest override new virtual partial
```

2.4.3 Literals

A literal is a source code representation of a value.

Literal:
  DecimalLiteral
  IntegerLiteral
  LogicalLiteral
  NullLiteral
  TextLiteral

Literals may be ascribed with a type to override the default type ascription.

2.4.3.1 Decimal literals

Decimal literals are used to write real-number values.

DecimalLiteral:
  DecimalDigits `.` *DecimalDigits*

Examples of decimal literal follow:

```
0.0
12.3
999999999999999.999999999999999
```

2.4.3.2 Integer literals

Integer literals are used to write integral values.

IntegerLiteral:
  -$_{opt}$ *DecimalDigits*

Examples of integer literal follow:

```
0
123
999999999999999999999999999999
-42
```

2.4.3.3 Logical literals

Logical literals are used to write logical values.

LogicalLiteral: one of
  `true false`

Examples of logical literal:

```
true
false
```

2.4.3.4 Text literals

M_g supports two forms of `Text` literals: regular text literals and verbatim text literals. In certain contexts, text literals must be of length one (single characters). However, M_g does not syntactically single characters.

A regular text literal consists of zero or more characters enclosed in single or double quotes, as in `"hello"` or `'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 a 'commercial at' character (`@`) followed by a single- or double-quote character (`'` or `"`), zero or more characters, and a closing quote character that matches the opening one. A simple example is `@"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 *SingleQuoteEscapeSequence* or a *DoubleQuoteEscapeSequence*, depending on the opening quote. 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.

A simple escape sequence represents a Unicode character encoding, as described in the table below.

Escape sequence	**Character name**	**Unicode encoding**
`\'`	Single quote	`0x0027`
`\"`	Double quote	`0x0022`
`\\`	Backslash	`0x005C`
`\0`	Null	`0x0000`
`\a`	Alert	`0x0007`
`\b`	Backspace	`0x0008`
`\f`	Form feed	`0x000C`
`\n`	New line	`0x000A`
`\r`	Carriage return	`0x000D`
`\t`	Horizontal tab	`0x0009`
`\v`	Vertical tab	`0x000B`

Since M_g uses a 16-bit encoding of Unicode code points in `Text` values, a Unicode character in the range `U+10000` to `U+10FFFF` is not considered a `Text` literal of length one (a single character), but is represented using a Unicode surrogate pair in a `Text` literal.

Unicode characters with code points above `0x10FFFF` are not supported.

Multiple translations are not performed. For instance, the text literal `\u005Cu005C` is equivalent to `\u005C` rather than `\`. The Unicode value `U+005C` is the character `\`.

A hexadecimal escape sequence represents a single Unicode character, with the value formed by the hexadecimal number following the prefix.

TextLiteral:
  `'` *SingleQuotedCharacters* `'`
  `"` *DoubleQuotedCharacters* `"`
  `@` `'` *SingleQuotedVerbatimCharacters*$_{opt}$ `'`
  `@` `"` *DoubleQuotedVerbatimCharacters*$_{opt}$ `"`

SingleQuotedCharacters:
  SingleQuotedCharacter
  SingleQuotedCharacters SingleQuotedCharacter

SingleQuotedCharacter:
  SingleQuotedCharacterSimple
  CharacterEscapeSimple
  CharacterEscapeUnicode

SingleQuotedCharacterSimple: none of
  `'`
  \
  NewLineCharacter

SingleQuotedVerbatimCharacters:
  SingleQuotedVerbatimCharacter
  SingleQuotedVerbatimCharacters SingleQuotedVerbatimCharacter

SingleQuotedVerbatimCharacter:
  none of `'`
  SingleQuotedVerbatimCharacterEscape

SingleQuotedVerbatimCharacterEscape:
  `" "`

DoubleQuotedCharacter:
  DoubleQuotedCharacterSimple
  CharacterEscape

DoubleQuotedCharacters:
  DoubleQuotedCharacter
  DoubleQuotedCharacters DoubleQuotedCharacter

DoubleQuotedCharacterSimple: none of
  `"`
  \
  NewLineCharacter

DoubleQuotedVerbatimCharacter:
  none of `"`
  DoubleQuotedVerbatimCharacterEscape

DoubleQuotedVerbatimCharacters:
  DoubleQuotedVerbatimCharacter
  DoubleQuotedVerbatimCharacters DoubleQuotedVerbatimCharacter

DoubleQuotedVerbatimCharacterEscape:
  `""`

CharacterEscape:
  CharacterEscapeSimple
  CharacterEscapeUnicode

CharacterEscapeSimple:
  `\` *CharacterEscapeSimpleCharacter*

CharacterEscapeSimpleCharacter: one of
  `' " \ 0 a b f n r t v`

CharacterEscapeUnicode:
  `\u` *HexDigit HexDigit HexDigit HexDigit*
  `\U` *HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit*

Examples of text literals follow:

```
'a'
'\u2323'
'\x2323'
'2323'
"Hello World"
@"""Hello,
World"""
"\u2323"
```

2.4.3.5 Null literal

The null literal is equal to no other value.

NullLiteral:
  `null`

An example of the null literal follows:

```
null
```

2.4.4 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.

OperatorOrPunctuator: one of
  `[ ] ( ) . , : ; ? = => + - * & | ^ { } # .. @ ' "`

2.5 Pre-processing directives

Pre-processing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code as a separate pre-processing step.

PPDirective:
PPDeclaration
PPConditional
PPDiagnostic
PPRegion

The following pre-processing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, 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 occur before the `#` character and between the `#` character and the directive name.

A source line containing a `#define`, `#undef`, `#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.

Pre-processing directives are neither tokens nor part of the syntactic grammar of M_g. 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_g program. For example, after pre-processing the source text:

```
#define A
#undef B
language C
{
#if A
    syntax F = "ABC";
#else
    syntax G = "HIJ";
#endif
#if B
    syntax H = "KLM";
#else
    syntax I = "DEF";
#endif
}
```

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

```
language C
{
    syntax F = "ABC";
    syntax I = "DEF";
}
```

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

2.5.1 Conditional compilation symbols

The conditional compilation functionality provided by the `#if`, `#else`, and `#endif` directives is controlled through pre-processing expressions and conditional compilation symbols.

ConditionalSymbol:
    Any *IdentifierOrKeyword* except `true` or `false`

A conditional compilation symbol has two possible states: defined or undefined. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a `#define` directive is processed, the conditional compilation symbol named in that directive becomes defined in that source file. The symbol remains defined until an `#undef` directive for that same symbol is processed, or until the end of the source file is reached. An implication of this is that `#define` and `#undef` directives in one source file have no effect on other source files in the same program.

When referenced in a pre-processing expression, a defined conditional compilation symbol has the Logical value `true`, and an undefined conditional compilation symbol has the Logical value `false`. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the value `false`.

Conditional compilation symbols can only be referenced in `#define` and `#undef` directives and in pre-processing expressions.

2.5.2 Pre-processing expressions

Pre-processing expressions can occur in `#if` directives. The operators `!`, `==`, `!=`, `&&` and `||` are permitted in pre-processing expressions, and parentheses may be used for grouping.

PPExpression:
    Whitespace$_{opt}$ *PPOrExpression* *Whitespace*$_{opt}$

OrExpression:
    PPAndExpression
    PPOrExpression *Whitespace*$_{opt}$ `||` *Whitespace*$_{opt}$ *PPAndExpression*

PPAndExpression:
    PPEqualityExpression
    PPAndExpression *Whitespace*$_{opt}$ `&&` *Whitespace*$_{opt}$ *PPEqualityExpression*

PPEqualityExpression:
    PPUnaryExpression
    PPEqualityExpression *Whitespace*$_{opt}$ `==` *Whitespace*$_{opt}$ *PPUnaryExpression*
    PPEqualityExpression *Whitespace*$_{opt}$ `!=` *Whitespace*$_{opt}$ *PPUnaryExpression*

PPUnaryExpression:
    PPPrimaryExpression
    `!` *Whitespace*$_{opt}$ *PPUnaryExpression*

PPPrimaryExpression:
  `true`
  `false`
  ConditionalSymbol
  `(` *Whitespace*$_{opt}$ *PPExpression Whitespace*$_{opt}$ `)`

When referenced in a pre-processing expression, a defined conditional compilation symbol has the Logical value `true`, and an undefined conditional compilation symbol has the Logical value `false`.

Evaluation of a pre-processing expression always yields a Logical value. The rules of evaluation for a pre-processing expression are the same as those for a constant expression, except that the only user-defined entities that can be referenced are conditional compilation symbols.

2.5.3 Declaration directives

The declaration directives are used to define or undefine conditional compilation symbols.

PPDeclaration:
  Whitespace$_{opt}$ `#` *Whitespace*$_{opt}$ `define` *Whitespace ConditionalSymbol PPNewLine*
  Whitespace$_{opt}$ `#` *Whitespace*$_{opt}$ `undef` *Whitespace ConditionalSymbol PPNewLine*

PPNewLine:
  Whitespace$_{opt}$ *SingleLineComment*$_{opt}$ *NewLine*

The processing of a `#define` directive causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of an `#undef` directive causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.

A `#define` may define a conditional compilation symbol that is already defined, without there being any intervening `#undef` for that symbol. The example below defines a conditional compilation symbol `A` and then defines it again.

```
#define A
#define A
```

A `#undef` may "undefine" a conditional compilation symbol that is not defined. The example below defines a conditional compilation symbol `A` and then undefines it twice; although the second `#undef` has no effect, it is still valid.

```
#define A
#undef A
#undef A
```

2.5.4 Conditional compilation directives

The conditional compilation directives are used to conditionally include or exclude portions of a source file.

PPConditional:
  PPIfSection PPElseSection$_{opt}$ *PPEndif*

PPIfSection:
  Whitespace$_{opt}$ `#` *Whitespace$_{opt}$* `if` *Whitespace PPExpression PPNewLine ConditionalSection$_{opt}$*

PPElseSection:
  Whitespace$_{opt}$ `#` *Whitespace$_{opt}$* `else` *PPNewLine ConditionalSection$_{opt}$*

PPEndif:
  Whitespace$_{opt}$ `#` *Whitespace$_{opt}$* `endif` *PPNewLine*

ConditionalSection:
  InputSection
  SkippedSection

SkippedSection:
  SkippedSectionPart
  SkippedSection SkippedSectionPart

SkippedSectionPart:
  SkippedCharacters$_{opt}$ NewLine
  PPDirective

SkippedCharacters:
  Whitespace$_{opt}$ NotNumberSign InputCharacters$_{opt}$

NotNumberSign:
  Any *InputCharacter* except #

As indicated by the syntax, conditional compilation directives must be written as sets consisting of, in order, an `#if` directive, zero or one `#else` directive, and an `#endif` directive. Between the directives are conditional sections of source code. Each section is controlled by the immediately preceding directive. A conditional section may itself contain nested conditional compilation directives provided these directives form complete sets.

A *PPConditional* selects at most one of the contained *ConditionalSection*s for normal lexical processing:

- The *PPExpression*s of the `#if` directives are evaluated in order until one yields `true`. If an expression yields `true`, the *ConditionalSection* of the corresponding directive is selected.
- If all *PPExpressions* yield `false`, and if an `#else` directive is present, the *ConditionalSection* of the `#else` directive is selected.
- Otherwise, no *ConditionalSection* is selected.

The selected *ConditionalSection*, if any, is processed as a normal *InputSection*: the source code contained in the section must adhere to the lexical grammar; tokens are generated from the source code in the section; and pre-processing directives in the section have the prescribed effects.

The remaining *ConditionalSection*s, if any, are processed as *SkippedSection*s: except for pre-processing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from the source code in the section; and pre-processing directives in the section must be lexically correct but are not otherwise processed. Within a *ConditionalSection* that is being processed as a *Skipped-Section*, any nested

*ConditionalSection*s (contained in nested `#if`...`#endif` and `#region`...`#endregion` constructs) are also processed as *SkippedSection*s.

Except for pre-processing directives, skipped source code is not subject to lexical analysis. For example, the following is valid despite the unterminated comment in the `#else` section:

```
#define Debug             // Debugging on
module HelloWorld {
    language HelloWorld {
        syntax Main =
#if Debug
            "Hello World"
        ;
#else
        /* Unterminated comment!
#endif
    }
}
```

Note, that pre-processing directives are required to be lexically correct even in skipped sections of source code.

Pre-processing directives are not processed when they appear inside multi-line input elements. For example, the program:

```
module HelloWorld {
    language HelloWorld {
        syntax Main = @'
#if Debug
            "Hello World"
        ;
#else
        /* Unterminated comment!
#endif'
    }
}
```

generates a language which recognizes the value:

```
#if Debug
            "Hello World"
        ;
#else
        /* Unterminated comment!
#endif
```

In peculiar cases, the set of pre-processing directives that is processed might depend on the evaluation of the *PPExpression*. The example:

```
#if X
    /*
#else
    /* */ syntax Q = empty;
#endif
```

always produces the same token stream (`syntax Q = empty;`), regardless of whether or not `X` is defined. If `X` is defined, the only processed directives are `#if` and `#endif`, due to the multi-line comment. If `X` is undefined, then three directives (`#if`, `#else`, `#endif`) are part of the directive set.

3 Text Pattern Expressions

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

3.1 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.

Primary:
ReferencePrimary
TextLiteral
RepetitionPrimary
CharacterClassPrimary
InlineRulePrimary
AnyPrimary

3.1.1 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.

CharacterClassPrimary:
TextLiteral `..` *TextLiteral*

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

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

3.1.2 References

A reference primary 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 §7.3.

ReferencePrimary:
GrammarReference

GrammarReference:
  Identifier
  GrammarReference `.` *Identifier*
  GrammarReference `.` *Identifier* `(` *TypeArguments* `)`
  Identifier `(` *TypeArguments* `)`

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.

3.1.3 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, `?`, `+`, `*`.

RepetitionPrimary:
  Primary Range
  Primary CollectionRanges

Range:
  `?`
  `*`
  `+`

CollectionRanges:
  `#` *IntegerLiteral*
  `#` *IntegerLiteral* `..` *IntegerLiteral*$_{opt}$

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..`

3.1.4 Inline Rules

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

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.

3.1.5 Any

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

Any:

```
    any
```

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

3.1.6 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"]`.

3.2 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 `|`.)

TextPatternExpression:
  Difference

Difference:
  Intersect
  Difference `-` *Intersect*

Intersect:
  Inverse
  Intersect `&` *Inverse*

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"`.

4 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).

ProductionDeclaration:
  ProductionPrecedence$_{opt}$ PatternDeclaration Constructor$_{opt}$

Constructor
  `=>` *TermConstructor*

ProductionPrecedence:
  `precedence` *IntegerLiteral* `:`

4.1 Pattern Declaration

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

PatternDeclaration:
  `empty`
  TermDeclarations$_{opt}$

TermDeclarations:
  TermDeclaration
  TermDeclarations TermDeclaration

4.2 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 §**Error! Reference source not found.**.

TermDeclaration:
  `error`
  Attributes$_{opt}$ TermPrecedence$_{opt}$ VariableBinding$_{opt}$ TextPatternExpression

VariableBinding:
  Name `:`

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 in conjunction with the checkpoint rule modifier to facilitate error recovery. This is described in §**Error! Reference source not found.**.

4.3 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.

TermConstructor:
  TopLevelNode

Node:
  Atom
  OrderedTerm
  UnorderedTerm

TopLevelNode:
  TopLevelAtom
  OrderedTerm
  UnorderedTerm

Nodes:
  Node
  Nodes `,` *Node*

OrderedTerm:
  Label$_{opt}$ `[` *Nodes*$_{opt}$ `]`

UnorderedTerm:
  Label$_{opt}$ `{` *Nodes*$_{opt}$ `}`

Label:
  Identifier
  `id` `(` *Atom* `)`

Atom:
  TopLevelAtom
  `valuesof` `(` *VariableReference* `)`

TopLevelAtom:
  TextLiteral
  DecimalLiteral
  LogicalLiteral
  IntegerLiteral
  NullLiteral
  VariableReference
  `labelof` `(` *VariableReference* `)`

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.

4.3.1 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.

4.3.2 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.
    a. If it contains constructors (a:A => a), then the output is the output of the constructor.
    b. 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
]
```

4.4 Precedence

The M_g 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.

4.4.1 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.

4.4.2 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_g 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.

5 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.

RuleDeclaration:
  Attributes$_{opt}$ *MemberModifiers*$_{opt}$ *Kind Name RuleParameters*$_{opt}$ *RuleBody*$_{opt}$ `;`

Kind:
  `token`
  `syntax`
  `interleave`

MemberModifiers:
  MemberModifier
  MemberModifiers MemberModifer

MemberModifier:
  `final`
  `identifier`

RuleBody:
  `=` *ProductionDeclarations*

ProductionDeclarations:
  ProductionDeclaration
  ProductionDeclarations `|` *ProductionDeclaration*

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

```
module HelloGoodby {
    language HelloGoodbye {
        syntax Main
          = "Hello"
          | "Goodbye";
    }
}
```

5.1 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.

5.1.1 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.

5.1.2 Identifier Modifier

The identifier modifier applies only to tokens. It is used to lower the precedence of language identifiers so they do not conflict with language keywords.

5.2 Syntax Rules

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

5.3 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.

5.4 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;
    }
}
```

5.5 Rule Parameters

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

RuleParameters:
    (*RuleParameterList*)

RuleParameterList:
    RuleParameter
    RuleParameterList , *RuleParameter*

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.

6 Languages

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

LanguageDeclaration:
    Attributes$_{opt}$ `language` *Name LanguageBody*

LanguageBody:
    `{` *RuleDeclarations*$_{opt}$ `}`

RuleDeclarations:
    RuleDeclaration
    RuleDeclarations RuleDeclaration

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

```
module HelloWorld {
    language HelloWorld {
        syntax Main
          = "Hello World";
    }
}
```

6.1 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.

6.2 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 §7.

7 Modules

An M_g module is a scope which contains declarations of languages (§6). Declarations exported by an imported module are made available in the importing module. Thus, modules override lexical scoping that otherwise governs M_g symbol resolution. Modules themselves do not nest.

7.1 Compilation Unit

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

CompilationUnit:
    ModuleDeclarations

ModuleDeclarations:
    ModuleDeclaration
    ModuleDeclarations ModuleDeclaration

7.2 Module Declaration

A *ModuleDeclaration* is a named container/scope for language declarations.

ModuleDeclaration:
    `module` *QualifiedIdentifer ModuleBody* `;`$_{opt}$

QualifiedIdentifier:
    Identifier
    QualifiedIdentifier `.` *Identifier*

ModuleBody:
    `{` *ImportDirectives ExportDirectives ModuleMemberDeclarations* `}`

ModuleMemberDeclarations:
    ModuleMemberDeclaration
    ModuleMemberDeclarations ModuleMemberDeclaration

ModuleMemberDeclaration:
    LanguageDeclaration

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 BaseDefinitions {
    export Logical;
    language Logical {
        syntax Literal = "true" | "false";
    }
}
```

The fully qualified name of the language is `BaseDefinitions.Logical`, or using escaped identifiers, `[BaseDefinitions].[Logical]`. 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 {
    language L {
        token I = ('0'..'9')+;
    }
}
module A.B {
    language M {
        token D = L.I '.' L.I;
    }
}
```

Module `A.B` is in error, as it does not contain a declaration for the identifier `L`. That is, the members of Module `A` are not implicitly imported into Module `A.B`.

7.3 Inter-Module Dependencies

M$_g$ uses *ImportDirective*s and *ExportDirective*s to explicitly control which declarations may be used across module boundaries.

ExportDirectives:
  ExportDirective
  ExportDirectives ExportDirective

ExportDirective:
  `export` *Identifiers*`;`

ImportDirectives:
  ImportDirective
  ImportDirectives ImportDirective

ImportDirective:
  `import` *ImportModules* `;`
  `import` *QualifiedIdentifier* `{` *ImportMembers* `}` `;`

ImportMember:
  Identifier *ImportAlias$_{opt}$*

ImportMembers:
  ImportMember
  ImportMembers `,` *ImportMember*

ImportModule:
  QualifiedIdentifier *ImportAlias$_{opt}$*

ImportModules:
  ImportModule
  ImportModules `,` *ImportModule*

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 Language.Core {
    export Base;

    language Internal {
        token Digit = '0'..'9';
        token Letter = 'A'..'Z' | 'a'..'z';
    }

    language Base {
        token Identifier = Letter (Letter | Digit)*;
    }
}
```

The definition `Language.Core.Internal` may only be referenced from within the module `Language.Core`. The definition `Language.Core.Base` may be referenced in any module that has an *ImportDirective* for module `Language.Core`, as shown in this example:

```
module Language.Extensions {
    import Language.Core;

    language Names {
        syntax QualifiedIdentifier
          = Language.Core.Base.Identifier '.' Language.Core.Base.Identifier;
    }
}
```

The example above uses the fully qualified name to refer to `Language.Core.Base`. An *ImportDirective* may also specify an *ImportAlias* that provides a replacement *Identifier* for the imported declaration:

```
module Language.Extensions {
    import Language.Core as lc;

    language Names {
        syntax QualifiedIdentifier
          = lc.Base.Identifier '.' lc.Base.Identifier;
    }
}
```

An *ImportAlias* replaces the name of the imported declaration. That means that the following is an error:

```
module Language.Extensions {
    import Language.Core as lc;

    language Names {
        syntax QualifiedIdentifier
          = Language.Core.Base.Identifier '.' Language.Core.Base.Identifier;
    }
}
```

It is legal for two or more *ImportDirectives* to import the same declaration, provided they specify distinct aliases. For a given compilation episode, 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 Language.Extensions {
    import Language.Core;

    language Names {
        syntax QualifiedIdentifier = Base.Identifier '.' Base.Identifier;
    }
}
```

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 L;
    language L {
        token X = '1';
    }
}
module B {
    export L;
    language L {
        token X = '2';
    }
}
```

It is legal to import both modules either with or without providing an alias:

```
module C {
    import A, B;
    language M {
        token Y = '3';
    }
}
```

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;
    language M {
        token Y = L.X | '3';
    }
}
```

This example can be made legal either by fully qualifying the reference to `L`:

```
module C {
    import A, B;
    language M {
        token Y = A.L.X | '3';   // no error
    }
}
```

or by adding an alias to one or both of the *ImportDirectives*:

```
module C {
    import A;
    import B as bb;
    language M {
        token Y = L.X | '3';   // no error, refers to A.L
        token Z = bb.L.X | '3';   // no error, refers to B.L
    }
}
```

An *ImportDirective* may either import all exported declarations from a module or only a selected subset of them. The latter is enabled by specifying *ImportMembers* as part of the directive. For example, Module `Plot2D` imports only `Point2D` and `PointPolar` from the Module `Geometry`:

```
module Geometry {
    import Algebra;
    export Geo2D, Geo3D;
    language Geo2D {
        syntax Point = '(' Numbers.Number ',' Numbers.Number ')';
        syntax PointPolar = '<' Numbers.Number ',' Numbers.Number '>';
    }
    language Geo3D {
        syntax Point =
             '(' Numbers.Number ',' Numbers.Number ',' Numbers.Number ')';
    }
}
module Plot2D {
    import Geometry {Geo2D};
    language Paths {
        syntax Path = '(' Geo2D.Point* ')';
        syntax PathPolar = '(' Geo2D.PointPolar* ')';
    }
}
```

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 Geo3D which is not imported from module Geometry:

```
module Plot3D {
    import Geometry {Geo2D};
    language Paths {
        syntax Path = '(' Geo3D.Point* ')';
    }
}
```

An *ImportDirective* that contains an *ImportAlias* on a selected imported member assigns the replacement name to the imported declaration, hiding the original export name.

```
module Plot3D {
    import Geometry {Geo3D as geo};
    language Paths {
        syntax Path = '(' geo.Point* ')';
    }
}
```

Aliasing an individual imported member is useful to resolve occasional conflicts between imports. Aliasing an entire imported module is useful to resolve a systemic conflict. For example, when importing two modules, where one is a different version of the other, it is likely to get many conflicts. Aliasing at member level would lead to a correspondingly long list of alias declarations.

8 Attributes

Attributes provide metadata which can be used to interpret the language feature they modify.

AttributeSections:
    AttributeSection
    AttributeSections AttributeSection

AttributeSection:
    `@{` *Nodes* `}`

8.1 Case Sensitive

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 {
    @{CaseSensitive[false]}
    language HelloWorld {
        syntax Main
          = Hello World;
        token Hello
          = "Hello";
        token World
          = "World";
        interleave Whitespace
          = " ";
    }
}
```