AsmL: The Abstract State Machine Language

October 7, 2002

Abstract

This document describes AsmL, a specification language based on abstract state machines.

Foundations of Software Engineering – Microsoft Research

Contents

1 Introduction 1

1.1 Executable specifications 1
1.2 Other Approaches 2
1.3 Applications 3
1.4 Features 3
1.5 Design goals 4
1.6 Audience 4
1.7 Notation 5
  1.7.1 Conventions for terminology 5
  1.7.2 Syntax 5
  1.7.3 Language version 5
1.8 Comments 6

2 Lexical Structure 7

2.1 AsmL source 7
2.2 Handling of control characters 7
2.3 Tokens 7
2.4 Comments 8
2.5 Identifiers 8
2.6 Literals 9
  2.6.1 Null 9
  2.6.2 Boolean literals 9
  2.6.3 Integer literals 9
  2.6.4 Literals for real numbers 10
  2.6.5 String literals 10
  2.6.6 Character literals 11
2.7 Keywords 11

3 Declarations 13

3.1 Block structure 13
3.2 Kinds of declarations 15
3.3 The Main() method 16
3.4 Names 16
3.5 Declaration Scope 16
  3.5.1 Unique declarations required per scope 16
  3.5.2 Shadowing of identifiers 17

  3.5.3 Order unimportant within a scope 17
  3.5.4 Closure of scope 17
3.6 Continuation of declarations 18

4 Values, Constructors and Patterns 19

4.1 Values 19
4.2 Constructors 19
4.3 Literal constructors 20
4.4 Datatype constructors 20
  4.4.1 Instance constructors 20
  4.4.2 Compound value constructors 21
  4.4.3 Enum constructors 21
4.5 Collection constructors 21
  4.5.1 Tuple construction 22
  4.5.2 Set construction 22
  4.5.3 Sequence construction 23
  4.5.4 Map construction 23
4.6 Patterns 24
  4.6.1 Universal patterns 25
  4.6.2 Literal patterns 25
  4.6.3 Identifier patterns 25
  4.6.4 The type pattern 26
  4.6.5 Tuple pattern 26
  4.6.6 Datatype pattern 27
  4.6.7 The maplet pattern 27
4.7 Binders 28
  4.7.1 Parallel binding semantics 30
  4.7.2 Order of bindings 30

5 Types 31

5.1 Type expressions 31
  5.1.1 Disjunctive types 32
  5.1.2 Option types 32
  5.1.3 Product types 32
  5.1.4 Named types 33
  5.1.5 Instantiated types 33
5.2 Operations on types 34
5.3 Built-in types 34
5.4 Subtypes 36

5.5 Type Declarations 36
5.5.1 User-declared subtypes 37
5.5.2 Interface declarations 37
5.5.3 Datatype declarations 38
5.5.4 Datatype variants 39
5.5.5 Enumerations 40
5.5.6 Constrained types 42
5.5.7 Constraints on type parameters 44

6 Members...46

6.1 Fields 46
6.1.1 Type constraints on values of field instances 46
6.1.2 Constants 47
6.1.3 Variables 47
6.1.4 Initialization of field instances 47
6.1.5 Kinds of fields 48
6.1.6 Indexing field names 49
6.1.7 Indexing parameters 49
6.2 Methods 50
6.2.1 Kinds of methods 51
6.2.2 Functions and procedures 52
6.2.3 Operators 52
6.2.4 Conversion methods 52
6.2.5 Method parameters 53
6.2.6 Static method selection 53
6.2.7 Dynamic method selection 56
6.2.8 Return values 56
6.2.9 Recursive methods 57
6.2.10 Type-parameterized, generic methods 57
6.2.11 Constructor methods 57
6.2.12 Disambiguation of method names 58
6.3 Constraints 59

7 Statements and Expressions...60

7.1 Statement blocks 60
7.2 Local fields 62
7.3 Assertion statements 63
7.4 Nondeterministic choice statements 65
7.5 Return statements 65
7.6 Try/catch statements 68
7.7 Conditional expressions 66
7.7.1 If-then-else expressions 66
7.7.2 Match expressions 66
7.7.3 Defaults for conditionals 67
7.8 Quantifying expressions 68
7.9 Selection expressions 70
7.10 Primary Expressions 71
7.10.1 Logical operations 72
7.10.2 Type query expressions 72
7.10.3 Type coercion expressions 72
7.11 Apply expressions 73
7.12 Atomic expression 74
7.13 Enumerated types 75
7.14 The do expression 77

8 State Operations...78

8.1 Update statements 78
8.1.1 Consistency of update statements 80
8.1.2 Locations 80
8.1.3 Partial and total updates 81
8.2 Parallel update blocks 81
8.3 Sequential blocks 82
8.3.1 Effect of recursion on sequential steps 83
8.3.2 Scope of constants and variables 83
8.3.3 Iterated steps 83
8.4 The skip statement 84
8.5 Processes 84
8.6 Agents 84
8.7 Exploration expressions 84

9 Namespaces..86

9.1 Unit of compilation (assembly) 86
9.2 Namespaces 86
9.3 Qualified names 87
9.4 Import directives 87
9.4.1 Units of compilation 89
9.5 Linkage 89

9.6 Literate programming environment 90

10 .NET Extensions 91

10.1 Modifiers 91

10.2 Attributes 92

10.3 Delegates 92

10.4 Properties 93

10.5 Events 93

10.6 Type integration 93

11 Library 95

11.1 Set operations 95

11.2 Sequence operations 95

11.3 Map operations 96

11.4 String operations 96

12 List of Examples 97

13 Grammar.. 99

13.1 Lexical level 99
13.1.1 Identifiers 99
13.1.2 Literals 99
13.1.3 Boolean literals 99
13.1.4 Integer literals 99
13.1.5 Literals for real numbers 99
13.1.6 String literals 100
13.1.7 Character literals 100
13.1.8 Keywords 100

13.2 Unit of compilation (assembly) 101

13.3 Values, constructors and patterns 101
13.3.1 Constructors 101
13.3.2 Patterns 101
13.3.3 Binders 102

13.4 Type expressions 102

13.5 Type declarations 102
13.5.1 Type Parameters 102
13.5.2 Type Relations 102
13.5.3 Interface 102
13.5.4 Datatype declaration 103
13.5.5 Enumerations 103
13.5.6 Constrained Types 103

13.6 Members 103
13.6.1 Fields 103
13.6.2 Methods 103
13.6.3 Constraints 104

13.7 Statements and expressions 104
13.7.1 Local fields 104
13.7.2 Assertion statements 105
13.7.3 Nondeterministic choice statements 105
13.7.4 Return statements 105
13.7.5 Try/catch statements 105
13.7.6 Conditional expressions 105
13.7.7 Quantifying expressions 105
13.7.8 Selection expressions 105
13.7.9 Primary Expressions 106
13.7.10 Apply expressions 106
13.7.11 Atomic expression 106

13.8 Runtime states 106
13.8.1 Update statements 106
13.8.2 Parallel update blocks 106
13.8.3 Sequential blocks 107
13.8.4 Exploration expressions 107

13.9 .NET Compatibility 107
13.9.1 Modifiers 107
13.9.2 Attributes 107
13.9.3 Delegates 107
13.9.4 Properties 108
13.9.5 Events 108

1 Introduction

1.1 Executable specifications

AsmL is a software specification language based on abstract state machines. It is used for creating human-readable, machine-executable models of a system's operation in a way that is *minimal* and *complete* with respect to any given user-defined level of abstraction. We call specifications written in AsmL *executable specifications*.

Like traditional specifications, executable specifications are descriptions of how software components work. Unlike traditional specifications, executable specifications have a single, unambiguous meaning. This meaning comes in the form of an abstract state machine (ASM), a mathematical model of the system's evolving, runtime state.

AsmL specifications may be run as a program, for instance, to simulate how a particular system will behave or to check the behavior of an implementation against its specification. However, unlike traditional programs, executable specifications are intended to be minimal. In other words, although they are faithful in describing, without omission, everything that is part of the chosen level of detail, they are equally faithful in leaving unspecified what is outside that level of detail.

Thus, unlike programs, executable specifications restrict themselves to the constraints and behavior that *all* correct implementations of the system will have in common. In other words, an executable specification must be as clear about the freedom given to correct implementations of the system it describes as it is about constraints.

For example, executable specifications do not constrain the order of operations unless it is significant, whereas current-day programs realize a sequential order of operation as an implementation decision.

This can be seen with an example:

Example 1 In-place sorting

```
var A = [3, 10, 5, 7, 1]
indices = {0, 1, 2, 3, 4}

Main()
  step until fixpoint
    choose i in indices, j in indices
           where i < j and A(i) > A(j)
      A(i) := A(j)
      A(j) := A(i)
  step
    WriteLine(A)        // prints [1, 3, 5, 7, 10]
```

This executable specification uses an abstract state machine for in-place sorting via a single-swap algorithm.

The machine performs sequential steps that swap the values of A whose elements are denoted by indices `i` and `j` such that `i` is less than `j` and the values `A(i)` and `A(j)` are out of order. It runs until no further updates are possible, that is, until the sequence is in order. As a final step, it prints the sorted sequence. The state of the machine at each step is entirely characterized by the value of the sequence `A` in that step.

The specification is minimal. The first point is that `choose` expression does not say how the two indices are selected, only that whatever indices are chosen must be distinct indices of out-of-order elements. Hence, many sorting algorithms, including quicksort and bubble sort, would be consistent with what we have specified.

Also, our example does not say how the swap operation happens. The values of the variables change as an atomic transaction. This leaves each implementation to decide how to perform the sequential swap, for instance, with an intervening copy to a temporary location.

1.2 Other Approaches

There are several other mathematical approaches besides abstract state machines that provide an operational model of software systems. An operational model is one that describes a system in terms of a mathematical machine. The most famous of these is the *Turing machine*, which can precisely represent any computable function as the evolving state of a machine that reads and writes binary digits to a serial memory. The difficulty, of course, is that the Turing machine's representation does not correspond to any commonsense view of the system that might aid human understanding.

ASMs, on the other hand, employ the *user's* view of the system as the vocabulary of the abstract machine that models the computation. As a consequence, with AsmL, one can describe the system's state in terms of variables and operations that make sense to the user. Thus, we say that an executable specification is a *faithful* model that *step-for-step* simulates a system at a given level of detail.

There are also a number of approaches that give an algebraic model of software systems, in contrast to an operational model. Algebraic models use algebraic equations that represent static constraints and definitions (that is, the rules relating the input and the output of a system).

AsmL embraces the formalism of algebraic specification but extends it (and this is crucial) with the dynamic properties of ASMs. Thus, AsmL can be used to build algebraic models of a system but is not limited to static definitions and correctness constraints. Instead, the symbolic vocabulary that characterizes an abstract state machine may include *dynamic state* variables whose values evolve during the run.

AsmL's focus is entirely on faithfully describing discrete systems in terms of evolving state. Thus, AsmL does not have an associated methodology for theorem proving or model checking, although executable specifications are well suited as input for many types of static analysis such as these. (An executable specification written in AsmL will typically have a static analysis search space that is several orders of magnitude smaller than an equivalent implementation written in a standard programming language.)

1.3 Applications

Executable specifications written in AsmL have some remarkable properties.

First, AsmL models can be run as simulations of the system they describe. This means the development team can, even before any code has been written, explore the proposed design and anticipate how different features will interact. However an AsmL model is more than a prototype or reference implementation, since it is a *complete* representation of a chosen level of design detail. In other words, a properly constructed AsmL model will say what each correct implementation *must* do, what it *may* do and what it *must not* do.

Second, AsmL models can be run in parallel with the implementation of the systems they describe to check that the specifications and the implementations agree. Not only does this verify the implementation, but it also ensures that the specification is up-to-date.

Finally, AsmL provides the rigor needed for algorithmic test case generation and, in many cases, for model checking and verification.

1.4 Features

AsmL is intended to be the standard ASM-based specification language for the growing worldwide ASM community, including software professionals working on large, real-world projects.

AsmL includes a state-of-the-art type system with extensive support for type parameterization and type inference. Using clear semantics, it provides a unified view of classes used for object-oriented programming, in addition to structured data types. It supports mathematical set operations—such as comprehension and quantification—that are useful for writing high-level specifications.

Along with taking advantage of the most sophisticated advances in language design, it was important that the language be practical, accessible, and easily integrated with the tools currently used by the development community. To this end, AsmL implementations can target real-world system environments, such as Microsoft's COM and .NET platforms. Its syntax was designed to read as much like pseudo-code as possible, making it understandable to members of the development team other than programmers. Developers, analysts, testers, managers, and documentation writers should be able to read an executable specification with only a modest amount of training.

As a specification language, we wanted AsmL to incorporate features that would make modeling actual systems as straightforward as possible. The language includes fundamental support for nondeterministic behavior.

AsmL is also capable of describing the evolving state of asynchronous, concurrent systems. It has been successfully applied to both protocols and component design.

1.5 Design goals

AsmL is designed to achieve the following goals:

- AsmL should be a practical specification language that scales to the needs of the largest commercial software projects, including operating systems and distributed software components.
- AsmL should be faithful to the spirit and clear semantics of abstract state machines.
- Executable specifications written in AsmL should look like pseudocode and be readable by anyone familiar with at least one other computer language.
- AsmL should be small, self-consistent and easy to explain.
- AsmL should not require an overly complex implementation.

The design was an engineering challenge. Focusing on these goals may have ruled out some language features that were more powerful, elegant, flexible and comfortable to mathematicians, language specialists and the existing ASM community in favor of syntax and features that met the needs of users from the world of commercial software development. (For example, array indices begin with zero in AsmL following the conventions of commercial programming languages, rather than with one as is the standard mathematical practice.)

We leave it to the reader to decide how successfully these design goals have been met.

1.6 Audience

We intend this reference manual to be useful to experienced software professionals and to language implementers. (Notes to language implementers are called out separately from the body text.) We have attempted to keep the descriptions precise while providing a generous number of examples.

Nonetheless, this manual is not a tutorial of abstract state machines nor is it a guide for applying executable specifications to software projects. Neither is it a primer on modern programming language design. For these purposes the reader should look elsewhere, including the *AsmL Tutorial* We also caution the reader against overlooking the importance of training and a certain amount of apprenticeship when first attempting to use AsmL on a commercial project.

1.7 Notation

1.7.1 Conventions for terminology

We use a special text color for terminology that is defined in the document. Additionally, terms are italicized they are defined. For example, we define *terminology* as a phrase with special meaning. Terminology may appear anywhere in the document.

Terminology is given special text color only once per paragraph. Subsequent occurrences of identical terminology within a paragraph are not given special formatting.

In the index found at the end of this document, the page number of each definition of new terminology is given in bold font.

1.7.2 Syntax

We use a Backus-Naur formalism to give the syntax of AsmL.

Terminal symbols are given in any of four forms: 1) in fixed-width `bold`, 2) by strings (for example, "`=`"), 3) by characters in single quotes or 4) as Unicode characters in hexadecimal form (for example, `\u00A0`).

Non-terminals are set in roman *italics* and are defined using the symbol "`::=`".

Alternatives are separated by a vertical bar, '|'. Ranges of characters are given by two adjacent periods, for example, `'a'..'z'` indicates any of the twenty-six lowercase Latin characters.

Parentheses "`(`" … "`)`" are used for grouping.

Curly braces in the form "`{`" … "`}`" are used to indicate zero or more repetitions.

Square braces in the form "`[`" … "`]`" indicate that the enclosed expression is optional.

Underlining indicates one or more occurrences of a production using identical indentation on a new line as separation. This convention is explained more fully in section 3.1 below.

1.7.3 Language version

This manual documents AsmL2.

1.8 Comments

AsmL is available for download at http://research.microsoft.com/foundations/asml.

Comments about the AsmL language or implementation should be sent to asml@microsoft.com.

Comments related to this manual can also be sent to the editor of the reference manual, using either v-colinc@microsoft.com or colinc@modeled-computation.com.

2 Lexical Structure

This section describes the lexical structure of AsmL text.

2.1 AsmL source

AsmL source is a sequence of characters (its *text*) encoded using the Unicode character set.

2.2 Handling of control characters

Except for the form feed, line feed and carriage return characters, AsmL rejects all *control characters* in the range `\u0000` through `\u001F` that may appear in the text of a program by issuing an error message. In particular, AsmL source may not contain the horizontal tab character (`\u0009`).

Carriage-return characters (`\u000D`) and *form-feed characters* (`\u000C`) are interpreted as *new-line characters* (`\u000A`). However, any carriage-return character that immediately precedes a new-line character is ignored (this affects only the line numbering of diagnostic error messages).

After adjusting for control characters, AsmL interprets the text of a program as a sequence of source lines. Each *source line* is a sequence of characters that ends with a new-line character. AsmL will implicitly terminate the text of a source with a new-line character if one is not already present.

2.3 Tokens

The text of an AsmL program is scanned as a sequence of tokens, possibly separated by white space and comments. Tokens are the terminal symbols of the AsmL grammar.

A *token* is a case-sensitive sequence of characters. There are three kinds of tokens: identifiers, literals and keywords. (These are described in the sections that follow.) Identifiers, literals and keywords have their own grammatical context and are not interchangeable. For example, a keyword may not be used in a context that expects a literal or identifier.

White space is required to separate tokens that begin or end with letter or digit characters; otherwise, white space is optional. For example, *graphemes* (that is, tokens like ">=" that do not contain letters) do not require white space separation.

White space is a sequence of one or more white space characters. A *white space character* is either the *space* (`\u0020`) or the new-line character (`LF`, or `\u000A`).

AsmL's lexical analysis uses the "longest prefix" rule. At each point, the longest possible character string satisfying the token production is read. So, although "`class`" is a keyword, "`classes`" is not. Similarly, the string ">=" would be

interpreted as the token for greater-than-or-equals instead of two tokens ">" and "=."

2.4 Comments

Comments are sequences of characters that are ignored by the parser when scanning AsmL text into a sequence of tokens. There are two forms used for comments.

A *line comment* begins with two forward slash characters ("//") and continues to the end of the source line

A *nested comment* begins with the character sequence "`/*`" and ends with the character sequence "`*/`". Nested comments may span multiple source lines.

The character sequences "`/*`" and "`//`" have no special significance within comments. The sequence "`*/`" has no significance within a line comment.

2.5 Identifiers

```
id          ::=  initIdChar { idChar } { ''' }
initIdChar  ::=  letter | ideographic | '@' | '_'
idChar      ::= letter | combining | ideographic
             |   digit | extender | underscore
letter      ::= // per Unicode section 4.5, letter,
                 excluding  combining characters
combining   ::= \u20DD | \u20DE | \u20DF | \u20E0
digit       ::= // per Unicode section 4.6, digit char
ideographic ::= \u2FF0..\u2FFF
extender    ::=  \u00B7 | \u02D0 | \u02D1 | \u0387 | \u0640
             |   \u0E46 | \u0EC6 | \u3005 | \u3031..\u3035
             |   \u309B..\u309D | \u309E | \u30FC..\u30FE
             |   \uFF70 | \uFF9E | \uFF9F
underscore  ::=  \u005F | \uFF3F
```

Identifier tokens are user-defined symbolic names.

The form used for AsmL identifiers is consistent with the conventions used for Microsoft Common Language Specification [CLS] with two exceptions. The first is that, unlike the CLS, AsmL permits the underscore character ('_', or `\u005F`) and the "Commercial At" character (`'@'`, or `\u0040`) to be used as initial characters of an identifier. The second is that it is permissible for an AsmL identifier to be suffixed by one or more apostrophe characters (`\u0027`).

The *letter* production is also equivalent to the Microsoft .NET Frameworks library function `System.Char.IsLetter()`, if the characters `\u20DD`, `\u20DE`, `\u20DF` and `\u20E0` are excluded.

The *digit* production is also equivalent to the Microsoft .NET Framework library function `System.Char.IsDigit()`.

Note to users

We recommend that users adopt as a coding convention that identifiers within the scope of an enclosing statement block, such as the names of local variables, be placed in "camel" case. Camel case means that lowercase letters are used, except that secondary words in a compound name are capitalized. Examples are "`begin`" and "`beginScope`." Camel case should also be used as the names of fields defined within datatypes. The identifiers of global fields, types and methods should be capitalized.

2.6 Literals

```
literal      ::= null | boolean | integer | real | string | char
```

Literals are tokens that denote values of certain built-in types. See section 4 below for more information about values and section 5.3 for more information about AsmL's built-in types.

2.6.1 Null

The literal `null` denotes a value that is distinct from all other values. The value `null` typically designates a default value.

The value `null` is of type `Null`.

2.6.2 Boolean literals

```
boolean      ::= true | false
```

The *Boolean literals* `true` and `false` are the values of the `Boolean` type.

2.6.3 Integer literals

```
integer      ::= (decimal | hexadecimal) [ integerSuffix ]
decimal      ::= digits
hexadecimal  ::= '0' ('x' | 'X') hexDigit { hexDigit }
integerSuffix ::= 'l' | 'L' | 's' | 'S' | 'b' | 'B'
digits       ::= digit { digit }
hexDigit     ::= digit | 'a' .. 'f' | 'A' .. 'F'
```

Integer literals may be given in either decimal notation or hexadecimal notation.

Decimal notation is a sequence of one or more digits.

Hexadecimal notation is a sequence of one or more hexadecimal digits prefixed by the characters `'0x'` or `'0X'`. A *hexadecimal digit* is a (decimal) digit or one of the characters 'a' through 'f' or 'A' through 'F' (corresponding to numbers whose decimal representations are 10 through 15 respectively).

The distinction between decimal and hexadecimal is only a matter of notation. In other words, the literals `31` and `0x1F` are two ways to denote the same value.

The type of an integer literal is `Integer`, unless the optional suffix `b`, `s` or `l` (or, in capital letters, `B`, `S`, `L`) is specified, in which case the literal is of type `Byte`, `Short` or `Long`, respectively.

Integer literals with differing suffixes denote distinct values. In other words, the domains of the various built-in types of integers are disjoint.

2.6.4 Literals for real numbers

```
real         ::= digits '.' digits [ exponent ] [ realSuffix ]
exponent     ::= ('e' | 'E') [ '+' | '-' ] digits
realSuffix   ::= 'f' | 'F'
```

A *literal for a real number* includes one or more digits to the left and to the right of a decimal point, followed an optional exponent. If provided, the exponent consists of the letter `'E'` or `'e'`, an optional sign (`'+'` or `'-'`) and a sequence of digits. The exponent indicates a power of ten by which the numeric value should be multiplied.

The type given by a real-number literal is `Double`, unless the literal has the suffix `F` or `f`, in which case the value is of type `Float`.

Numeric literals, whether real numbers or integers, that fall outside the domain of their type generate an error.

Literals suffixed by `f` are distinct from those not so suffixed. In other words, the domains of the types `Double` and `Float` are disjoint.

2.6.5 String literals

```
string       ::= quote { strChar } quote
strChar      ::= readable | whiteChar | sQuote | '\' esc
readable     ::= (see text below)
quote        ::= '"'
esc          ::= 'b' | 'f' | 'n' | 't' | 'r'
               | ('u' hexDigit hexDigit hexDigit hexDigit)
```

A *string literal* contains between its delimiting double quotes zero or more readable characters, single quote characters (`\u0027`), white space characters and escaped characters.

In AsmL *readable characters* include all letter characters, digits, the space character (`\u0020`) as well as all of the characters used in AsmL for keywords. The character `'\'` (`\u005C`) is not a readable character. White space characters other than the space character are not readable characters. The single quote and double quote characters are not readable characters.

An *escaped character* consists of a backslash character “`\`” (`\u005c`) followed by an *escape code*.

Escape codes may denote the control characters "backspace" (`\b`), "form feed" (`\f`), "new line" (`\n`) and "horizontal tab" (`\t`).

Escape codes may also be in numeric form to denote a character by its Unicode encoding. The hexadecimal escape code begins with a “u” and is followed by four hexadecimal digits, for example “`\u0022`”.

The sequences of characters “ `/*`”, “`*/`” and “ `//`” have no special significance within a string literal.

The value denoted by a string literal is of type `String`.

2.6.6 Character literals

```
char      ::=  sQuote ( readable | quote | '\' esc ) sQuote
sQuote    ::=  "'"
```

Character literals denote values of the built-in type `Char`. Between its delimiting single quotes, a character literal contains a readable character, a double quote character (`\u0022`) or an escaped character.

2.7 Keywords

AsmL recognizes the following tokens as keywords.

`->`	`{`	`error`	`interface`	`out`	`sum`
`..`	`|`	`event`	`internal`	`override`	`the`
`:=`	`}`	`exists`	`intersect`	`primitive`	`then`
`<=`	`abstract`	`explore`	`is`	`private`	`throw`
`<>`	`add`	`extends`	`let`	`procedure`	`to`
`>=`	`and`	`fixpoint`	`lt`	`process`	`try`
`(`	`any`	`for`	`lte`	`property`	`type`
`)`	`as`	`forall`	`match`	`protected`	`union`
`*`	`case`	`foreach`	`max`	`public`	`unique`
`+`	`catch`	`from`	`me`	`ref`	`until`
`,`	`choose`	`function`	`merge`	`remove`	`value`
`-`	`class`	`get`	`min`	`require`	`var`
`.`	`const`	`gt`	`mod`	`resulting`	`virtual`
`/`	`constraint`	`gte`	`mybase`	`return`	`where`
`:`	`delegate`	`holds`	`namespace`	`sealed`	`while`
`<`	`do`	`if`	`ne`	`search`	
`=`	`else`	`ifnone`	`new`	`set`	
`>`	`elseif`	`implements`	`not`	`shared`	
`?`	`ensure`	`implies`	`notin`	`skip`	
`[`	`enum`	`import`	`of`	`step`	
`]`	`enumerated`	`in`	`operator`	`structure`	
`+=`	`*=`	`initially`	`or`	`subset`	
`_`	`eq`	`inout`	`otherwise`	`subseteq`	

Alternatives `eq`, `ne`, `lt`, `gt`, `le` and `ge` may be substituted for "=", "<>", "<", "<=", ">" and ">=", respectively. (This makes it easier for AsmL source code to be integrated into XML documents in some situations.)

The keywords `the`, `min`, `max` and `sum` used to introduce a select expression (see section 7.9 below) may also be used as identifier tokens.

3 Declarations

An AsmL program consists of declarations that establish the program's *vocabulary*, a fixed set of symbols with defined operational meaning. This section describes how to interpret the token sequence described in the previous section as an AsmL program.

Each *declaration* establishes the meaning of an identifier (called a *declared name*) within its scope. The definition of a declared name is *static*. In other words, the meaning of a program's vocabulary does not change during the run of the program.

Note to users

Declarations in AsmL have rigorous mathematical semantics. This means that there is only one interpretation of a program written in AsmL and that this interpretation can be directly and completely expressed in mathematical terms.

For example, declaring a name as a `Set` *in AsmL means that the name denotes an abstract entity with the same properties as a finite set in mathematical set theory. Even "state-changing" operations such as updating the value of a variable can be precisely understood in terms of operations on an abstract mathematical machine.*

It is not necessary to understand AsmL's mathematical foundation in order to use or implement the language. In fact one of AsmL's primary design motivations is to make clear mathematical semantics practical in the world of commercial software development without requiring software professionals to become mathematicians.

As a consequence, this document does not give the full semantics of AsmL, although we do add "notes to users" throughout the text to clarify semantic issues that could be confusing.

Declarations may be nested, and the order of declarations in a program does not matter.

Note that AsmL also provides namespaces to govern the visibility of declared names. Namespaces are not required, and so we will defer them until section 8 below.

3.1 Block structure

AsmL declarations sometimes use layout (that is, indentation and new lines) to indicate block structure. In other words, AsmL interprets a new line and indentation as *delimiting* certain lists of entities.

In the grammar that follows, an underlined term represents a list of that term, and the parser will recognize indented layout as a delimiting token between items in the list. For instance, "*stm*" would be an indented list of "*stm*" terms.

The first item in the list must be indented (possibly on a new line) with respect to the first token of the production in which the list occurs. For this purpose, the definition of a named term is the containing production.

All items that follow the first must start on a new line with the same offset as the first list item (called *block offset* of the list). A character's *offset* is the number of characters in the line that precede it within its source line. Comments are significant when calculating a character's offset on a source line.

Lines consisting entirely of white space and comments are ignored for the purposes of indented layout.

The end of the list is not delimited. The list terminates when the enclosing production continues.

Compatibility Note

Previous version of AsmL allowed semicolons as an alternative way to separate items in a list. The use of semicolons as separators has been removed from AsmL.

Example 2 Indentation as block structure

```
/*
 *  enum    ::= "enum" id [ "extends" typeExp ] [ element ]
 *  element ::= id ["=" exp]
 */

enum Color1
  Red
  Green
  Blue

enum Color2 { Orange Yellow Violet }
```

Note the first token of the production is "`enum`", so every `element` has to be indented with respect to the column where "`enum`" appears. Each element must be identically indented. Indentation is not required for the second enum because curly braces have been used to indicate the extent of the list.

Example 3 Indentation as block structure

```
/*
 * ifExpr    ::= if exp [then] stm
 *               { elseif exp [then] stm }
 *               [ else stm ]
 */

Main()
  var x as Integer = 1
```

```
var y as Integer = 2
let flag = if x < y then x else y
let flag2 = if x > y then
              x
            else
              y
step
  if x > y then x := x + 1
                y := x + 2
       else
    x := 33
step
  WriteLine(x)
```

Example 3 shows how indentation can be used for blocks of expressions. Note that the indentation of the last list is relative to `if` (and not `else`), since `if` begins the production in which the *stm* was given in the syntax.

For namespaces the offside rule also treats each entity as a list entity. For namespaces the first token of a compilation unit determines the block offset.

3.2 Kinds of declarations

declaration ::= *import* | *type* | *member*

Declarations are import declarations, type declarations or member declarations.

Type declarations (see section 5.3) provide the named structures familiar to object-oriented programmers, such as interfaces and classes. Type declarations define new named types or if type parameters are given new type families.

Member declarations (see section 6 below) provide fields and methods. Member declarations may be nested inside of a type declaration or appear *globally*, outside of any type declaration.

Example 4 Declarations

```
const max_Integer = 10   // global member decl

class Cell               // type declaration
  var cont as Integer    // member decl nested inside type decl

Main()                   // global member decl
  WriteLine(max_Integer)
```

3.3 The Main() method

The operational meaning of a program is given by its `Main()` method. In other words, `Main()` is the top-level entry point, like `main()` in the "C" programming language.

3.4 Names

name ::= { *id* "." } *id*

Names used in the program consist of one or more identifiers (see 2.5 above) separated by a dot ("`.`"). They may be either simple or qualified.

Simple names do not contain a dot ("`.`"). *Qualified names* are those that include a dot ("`.`").

For example, `Pressure_2` and `Control.Common.Pressure_2` are well-formed names. The form `.Pressure_2` is not a name, since the dot ("`.`") must be preceded by an identifier.

Note that qualified names are defined in AsmL at the token level, not the lexical level. This means that white space and comments may appear in between the tokens that constitute a qualified name.

We use the terms *name* and *identifier* interchangeably throughout the rest of this reference. The grammar makes it clear when a qualified name may be used instead of a simple name.

3.5 Declaration Scope

The *scope* of a declared name is the region of the program text within which the declared name has meaning.

Unless otherwise noted, the scope of a declared name `N` is the *enclosing scope*, that is, the region given by the declaration that contains `N`'s declaration in nested form. If `N`'s declaration is not nested within another declaration, it has *global scope* (that is, it is defined within the namespace `Main` as we will see later in section 9.2). A name with global scope is called a *global name*.

3.5.1 Unique declarations required per scope

All declared names must be distinct within their scope. For example, an error occurs if a type declaration and a field declaration introduce the same name in the same scope. It is also not allowed to give a field the same name as a method.

There are exceptions to this rule: overloaded method names and continued declarations.

Overloaded methods are distinguished by their argument types as well as their names. It is therefore possible that two distinct methods will have the same name. See section 6.2.6 below.

Continued declarations allow a single declaration to be split into sections. For example, a class declaration may introduce methods in lexically separate blocks. See section 3.6 below.

Implementation Note

The AsmL currently does not check prevent a field and a method from having identical names. This will be corrected in a subsequent release.

3.5.2 Shadowing of identifiers

Names introduced either by declarations nested within a type declaration (assuming the `shared` keyword is absent) or by statements (see section 7.2 below) are called *locally declared names*.

Locally declared names hide global names. For example, names introduced inside of methods for local variables may be the same as global variables. In this case, any references to the name are interpreted using the local definition.

Note that the shadowed names are still available by means of qualified names. See section 9.3 below for the use of qualified names.

Local names are not allowed to shadow other local names, regardless of nesting level of their respective scopes.

Shadowing the names of types is not allowed.

3.5.3 Order unimportant within a scope

The order of declarations in a scope is of no significance. However, there are two exceptions.

First, the order that field declarations occur in class or structure declaration determines the order of the parameters of the default construction expression for that datatype.

Second, the order of elements in an enumeration determines the default numeric values associated with those elements. See section 5.5.5 below.

3.5.4 Closure of scope

Every scope in a program must be *closed*. In other words, every simple name referenced within a scope must be a declared name visible in that scope.

3.6 Continuation of declarations

AsmL allows a type declaration, namespace or method to be divided into distinct lexical blocks.

In general, a declaration is simply the union of separate lexical blocks. In all cases, the interpretation is "union of constraint." That is, the information provided by all declarations of a given name within the same scope must not contradict.

Implementation note

When a method declaration is continued, only one occurrence of the method may have a body. This is a restriction may be relaxed in future versions of AsmL.

Example 5 Continuation of declarations

```
class Cell
  const id as String

  SetValue(i as Integer)

  GetValue() as Integer

Main()
  step
    let c = new Cell( "ID1", 42)
  step
    WriteLine(c.GetValue())

class Cell                    // continuation of class
  var storage as Integer

  SetValue(i as Integer)      // continuation of method
    Storage := i

  GetValue() as Integer       // continuation of method
    return storage
```

4 Values, Constructors and Patterns

4.1 Values

Values are the immutable, abstract entities that exist during the run of a program.

Evaluating an expression (i.e., a formula) at runtime produces a value. For example, if we evaluate the expression `1 + 3` we get the value `4`.

Values comprise the domain of each type. (See section 5 below for information about types.)

The fundamental operations that apply to all values are equality (the "`=`" operator) and set membership (the "`in`" operator). We may always query whether two expressions represent the same value and whether a given value is an element of a given set.

Note to users

Values are "elements" in the mathematical sense. That is, they are the abstract entities used as members of mathematical sets.

The notion of a value's "identity" is fundamental. Thus, values are immutable, primitive entities that do not change as the system runs.

Of course, a variable (a named location that contains a value) may be associated with various values as the system's state evolves during the run of the program. When we speak of changing "the value of a variable" it is only the association of variable to value that changes.

4.2 Constructors

```
constructor ::=    literal
              |    datatypeConstructor
              |    collectionConstructor
```

Constructors denote values.

A constructor can be in one of several forms, called *construction expressions*. There are three kinds of construction expressions: literals, datatype constructors and collection constructors.

It is possible for a single value to have more than one form of construction expression. For example, the literals `0x10` and `16` denote the same value. (The first is just a hexadecimal representation.)

It is also possible that a construction expression will produce distinct values when invoked in different contexts. For example, each invocation of the operator `new` (to create instances of a class) will result in a distinct, new value.

4.3 Literal constructors

A *literal constructor* denotes a value of a built-in type such as `Boolean`, `String` and `Integer`. The syntax for each kind of literal is given above in section 2.6.

Example 6 Literal constructors

```
“This is a string”       // string literal
2.0                      // literal for real number
0x02                     // Integer literal in hexadecimal
```

4.4 Datatype constructors

```
datatypeConstructor ::= [ new ] typeName [ "(" [ exps ] ")" ]
```

Datatype constructors denote values of class, structure and enum types. The syntax for type names is given in section 5.1 below. The syntax for expressions ("*exps*") is in section 7 below.

4.4.1 Instance constructors

The form `new` *typeName* (*arg1*, *arg2*, …) is called an *instance constructor*. The type name given in an instance constructor must be that of a class. The arguments provide values for the instance-level fields.

Each invocation of an instance constructor always denotes a new, distinct value, called an instance of the class. Note that two instance constructors in the same form with identical arguments denote two *different* values.

The parentheses after the type name may optionally be omitted if the class does not include fields that need to be initialized. The keyword `new` is required when instantiating values of class types.

If the type name given in a datatype constructor is that of an instantiated type (see section 5.1.5 below), then the name of the corresponding type family may be sometimes be substituted for the type name. This may happen when the arguments given to the constructor fully constrain the type instantiation. See section 5.1.5 below for an example.

Example 7 Constructing instances

```
class Person
  name as String

Main()
  if new Person("Bob") <> new Person("Bob") then
    WriteLine("Instance constructors always yield values " +
              "that are distinct from all other values.")
```

4.4.2 Compound value constructors

The form *typeName* (*arg1*, *arg2*, …) denotes a *compound value*, that is, a value of a structure type. The type name given in a compound value constructor must be that of a structure. Note that the keyword `new` must not be used when constructing values of a structure type.

Note that two compound value constructors in the same form with identical arguments denote the *same* value (assuming *free construction*, the absence of nondeterminism in the constructor's initialization).

The parentheses after the type name may optionally be omitted if the structure does not include fields that need to be initialized.

Example 8 Constructing compound values

```
structure Point
  x as Integer
  y as Integer

Main()
  if Point(1, 2) = Point(1, 2) then
    WriteLine("Compound value constructors denote " +
              "the same values if their arguments " +
              "are identical.")
```

4.4.3 Enum constructors

The datatype constructor provides the syntax for enum values. This is just *elementName*.

Example 9 Constructing enumerated values

```
enum Color
  Red
  Green

Main()
  let x = Green            // Green is a constructor
  match x
    Green: WriteLine("x is Green")
```

4.5 Collection constructors

```
collectionConstructor ::= tupleExp | setExp | seqExp | mapExp
tupleExp    ::= "(" exp "," exps ")"
setExp      ::= "{" [ comprehension | exps | range ] "}"
seqExp      ::= "[" [ comprehension | exps | range ] "]"
mapExp ::= "{" ( mapComprehension | mapExps | "->") "}"
```

```
range          ::= exp ".." exp
comprehension  ::= exp "|" binders
mapComprehension ::=    maplet "|" binders
mapExps        ::= maplet { "," maplet }
maplet         ::= exp "->" exp
```

Collection constructors yield values of AsmL's built-in types for sequences, sets, maps and tuples.

4.5.1 Tuple construction

A construction expression in the form (*arg1*, *arg2*, …) denotes a *tuple*, or an element of a product type (see section 4).

Note that the form (*arg*) denotes the value given by *arg*. The form `()` is not the constructor of any value. An error will occur if `()` appears in a context that requires a value.

Example 10 Constructing tuples

```
(1, 2, "abc")       // value of type (Integer, Integer, String)
```

4.5.2 Set construction

Construction expressions for the built-in type family `Set` have three forms: set range, set comprehension and set display.

A *set range* is in the form {*arg1*..*arg2*}, where *arg1* and *arg2* are expressions. The set range denotes the set of all values greater than or equal to *arg1* and less than or equal to *arg2*. Both arguments must be of the same type. The argument types for a set range may be `Integer`, `Long`, `Short`, `Byte` and `Char`.

Set comprehension denotes sets in terms of iteration expressions. Its form is {*exp* | *binder1*, *binder2*, …}. The values given by evaluating *exp* in each binding context constitute the value of the set denoted by the comprehension expression. Binders are described below in 4.7.

Set display is an enumeration of values in the form {*arg1*, *arg2*, …}, denoting the set that contains each of the given values. Duplicate values are ignored. The order that values are given in a set display does not matter.

Example 11 Constructing sets

```
x = {2..5}                    // same as {3, 2, 5, 4}
y = {i | i in x where i < 4}  // same as {2, 3}
z = {3, 2}                    // same as y
```

4.5.3 Sequence construction

Construction expressions for the built-in type `Seq` have three forms: sequence range, sequence comprehension and sequence display.

A *sequence range* is in the form `[`*arg1*`..`*arg2*`]`, where *arg1* and *arg2* are expressions. The set range denotes the ordered sequence of all values greater than or equal to *arg1* and less than or equal to *arg2*. Both arguments must be of the same type. The argument types for a set range may be `Integer`, `Long` `Short`, `Byte` and `Char`.

Sequence comprehension denotes sequence in terms of iteration expressions. Its form is `[`*exp* `|` *binder1* , *binder2* , ... `]`. The values given by evaluating *exp* for each binding in left-to-right order produce the sequence of values denoted by the comprehension. Binders are described below in 4.7

Sequence display is an enumeration of values in the form `[`*arg1*, *arg2*, ...`]`, denoting the sequence whose ith element equals the ith argument in the constructor. The order of elements is significant, and duplicate values are respected.

Example 12 Constructing sequences

```
x = [2..5]                      // same as [2, 3, 4, 5]
y = [i | i in x where i < 4]    // same as [2, 3]
z = [2, 3]                      // same as y
w = [2, 2, 3]                   // not the same as z
```

4.5.4 Map construction

Map display is an enumeration of individual element-to-element associations in the form $\{key_1$ `->` val_1, key_2 `->` val_2, ...$\}$. A map display denotes a map value `M` such that `M(`key_i`)` yields val_i for each key_i and val_i given. If any two values key_i and key_j are the same, then val_i and val_j must denote identical values, or an error occurs.

Map comprehension denotes a map in terms of iterated expressions. Its form is {*expr1* `->` *expr2* `|` *binder1* , *binder2* , ...}. This form denotes a `Map` value constructed by evaluating *expr1* and *expr2* for each iterated binding and collecting the key/value pairs into a table. Binders are described below in 4.7.

The form `{->}` denotes the empty map.

Example 13 Constructing maps

```
x = {2..5}
y = {i -> i + 1 | i in x where i < 4}
z = {2 -> 3, 3 -> 4}                    // same as y
WriteLine(z(2))                         // prints 3
```

4.6 Patterns

```
pat         ::= "_"
              |   literal
              |   id [ as typeExp ]
              |   tuplePat
              |   datatypePat
              |   mapletPat

tuplePat    ::= "(" pats ")"
datatypePat ::= typeName [ "(" [ pats ] ")" ]
mapletPat   ::= pat "->" pat
 pats       ::= pat { "," pat }
```

Patterns are destructuring forms. With patterns, the user can decompose a value into its constituent parts using syntax that mirrors the value's constructor (see section 4.3).

Patterns are used for *matching*, the process of testing whether the constructor of a given value has the same form as a given pattern. Matching occurs when the pattern form is consistent with the constructor of the value being matched.

Pattern syntax is also used for *binding*, the process of associating an identifier with a value. (The "let" statement is an example of binding.) Note that matching must also occur if any binding is to take place.

Patterns occur in four contexts in AsmL:

- As cases in a `match` statement (see section 7.6.2 below).
- In a `let` statement to indicate the names that will be bound to values (see section 7.2 below).
- In binder clause to give the names that will take on multiple, iterated values (see 4.7 below).
- Within another pattern, to form a nested pattern.

Example 14 Symmetry of construction and pattern matching

```
structure Point
  x as Int eger
  y as Integer

Main()
  let p = Point(3, 2)            // constructor
  let Point(a, b) = p            // pattern
  WriteLine(a)                   // prints 3
```

Note in the example how the *constructor* `Point(3, 2)` has the same form as the *pattern* `Point(a, b)`. The constructor yields a value, while the pattern is matched against an existing value to bind `a = 3` and `b = 2`.

4.6.1 Universal patterns

The *universal pattern* is an underscore token ("_"). The universal pattern can be matched against any `value` but does not result in a new binding of a name to a value.

Note that the underscore token has special meaning and may be used in AsmL only for the universal pattern.

4.6.2 Literal patterns

A *literal pattern* has the same form a literal (such as a string literal or a numeric literal). A match occurs if the value being tested equals the literal given. No binding results.

Example 15 Pattern matching without binding

```
CheckRemainder(i as Integer, r as Integer)
  match i mod r
    0: WriteLine(“Divides evenly!”)
    1: WriteLine(“Has one left over”)
    _: WriteLine(“Has more than one left over”)

Main()
  CheckRemainder(3, 2)       // prints "Has one left over"
```

In Example 15 the value of expression `i mod r` matches the pattern `1` (since in this example `i mod r` means `3 mod 2`, or the value `1`.)

4.6.3 Identifier patterns

An *identifier pattern* matches any value, and a binding is established between the name and the matched value. Its syntax is just that of an identifier.

Example 16 Single-name patterns

```
Main()
  let x = (1, “first”)

  choose y in {1, 2}
    WriteLine({z | z in {0..y}})  // prints {0, 1} or {0, 1, 2}
```

In Example 16 `x`, `y` and `z` are identifier patterns.

4.6.4 The type pattern

A *type pattern* has the form *id* as *type*. It is similar to the identifier pattern, but the type pattern only succeeds if the value being matched is a subtype of *type*. If a match occurs, the value is bound to the name *id* with declared type *type*.

Example 17 Type patterns

```
structure Point
  x as Integer
  y as Integer

structure ColorPoint extends Point
  color as String

PrintPointColor(p as Point)
  match p
    cp as ColorPoint:
      WriteLine(cp.color)
    _:
      WriteLine("No color present")

Main()
  a = ColorPoint(1, 2, "red")
  PrintPointColor(a)         // prints "red"
```

The form `cp as ColorPoint` in Example 17 is a type pattern This example shows a type-safe way of "downcasting," or determining at runtime whether a value is in the domain a type other than its declared type.

4.6.5 Tuple pattern

The form (*pattern* , *pattern* ...) is called the *tuple pattern*. The pattern matches if its form is the same as the construction expression of the given value and each of its patterns match pairwise with those of the value. It is possible that pattern matching is recursive.

Example 18 Tuple pattern

```
Main()
  let a = (1, (2, "abc"))
  let (b, (_, c)) = a        // b is 1, c is "abc"
  WriteLine(c)               // prints "abc"
```

4.6.6 Datatype pattern

A *datatype pattern* has the form *typeName* (*pattern1* , *pattern2*, …). The pattern matches if the name and patterns match the default construction expression of the given value. This is similar in form to the default construction expression of that datatype, either class, structure or enum.

If the constructor of a structure or class does not have any parameters, then the pattern corresponding to that constructor may omit the parentheses. The patterns for enums do not include parentheses.

Note that, unlike the class constructor, the datatype pattern does not use the keyword `new`. (This is an exception to the rule stated above that patterns have the same syntax as constructors.)

Example 19 Destructing patterns for structures

```
structure List of T
  case Nil
  case Cons
    head as T
    tail as List of T

Main()
  let x = Cons of Integer(2, Nil of Integer)
  let Cons of Integer(a, _) = x           // same as a = 2
  let y = Cons(10, x)

  match y
    Cons of Integer(10, Cons of Integer(2, _)):
       WriteLine("Matched y with nested pattern")
```

Note to users

Pattern matching should not be used for datatypes that inherit fields from a supertype. (The behavior in this case is undefined and may change in future versions of AsmL.)

4.6.7 The maplet pattern

A *maplet pattern* has the form *pattern1* `->` *pattern2*. The symbol "->" is read as "maps to".

The context in which a maplet pattern may appear is more restricted than other kinds of patterns. A maplet pattern may only appear within a binder form (see 4.7 below), before the keyword `in`. A maplet pattern may not be used within a match case statement, within a let binding or nested within another pattern. The only use of a maplet pattern is to produce bindings for key/value associations given in a map.

The maplet pattern in the form *pat1* `->` *pat2* `in` *exp* produces bindings for every case where *pat1* matches a key value of the map given by *exp* and *pat2* matches the lookup value associated with that key.

Example 20 Maplet patterns

```
const myMap = {"one" -> 1, "two" -> 2, "three" -> 3}

IsOdd(x as Integer) as Boolean
  return (1 = x mod 2)

Main()
  step
    let oddNumbers = {i | i -> j in myMap where IsOdd(j)}
    WriteLine(oddNumbers)           // prints {"one", "three"}

  step
    let two = the i | i -> 2 in myMap
    WriteLine(two)                  // prints "two"
```

In Example 20 the forms `i -> j` and `i -> 2` are maplet patterns. `OddNumbers` is the set of all `i` such that the key/value pair `i`-mapsto-`j` is found in the table `myMap` and `j` is an odd number. `Two` is the (unique) i such that `i`-mapsto-`2` is found in the table `myMap`.

Note to users

Maplet patterns are more restricted than other patterns. This arises from the fact that there is no value corresponding to key/value associations that constitute a map.

4.7 Binders

```
binders     ::= binder { "," binder }
binder      ::= pat ( in | "=" ) exp [ where exp ]
```

AsmL uses a form called a *binder* for associating names with values. Binders are used for

- comprehension (see sections 4.5.2, 4.5.3 and 4.5.4 above),
- quantification (see section 7.7 below),
- nondeterministic choice expressions (see 7.4 below),
- parallel update, and
- sequential iteration.

Binders give the `identifiers` to be bound by means of a pattern (see 4.6 above), the token “`in`” or “=”, and an expression that provides the `values` that will be associated with the given identifiers. Each binder clause in a series is delimited by a comma (`","`). Each binder may optionally include a `where` clause that further restricts the bindings produced.

Depending on context, binders support simple binding, iterated binding and nondeterministic choice. Simple binding and nondeterministic choice result in a single association of names to values; iterated binding produces multiple associations of names and values.

Simple binding occurs when the equal sign ("=") is used in a binder.

Iterated binding occurs when the "`in`" keyword is used in a binder, except that within a choose-expression the "`in`" keyword is interpreted as nondeterministic choice.

With iterated binding, a binder produces one name/value association for each possible match of the pattern to the left of "`in`" with each value in the set, sequence or `map` that appears to the right of "`in`".

If there is more than one binder, the iteration occurs in a *nested binding*. This means that the bindings proceed in an outer-to-inner fashion, with the left-most binder acting as the outer-most loop. In a nested binding, it is possible to use identifiers introduced in a binder within expressions that occur in any other binders that appear to the right.

There is special handling of an identifier pattern within a binder that operates on the built-in map type. In this case, the value bound will be taken from the key values of the map. In other words, the form `x in m`, where `m` is a map will be interpreted as `x in Indices(m)`. (The built-in library function `Indices()` returns the key values of a map as a set.)

Nondeterministic choice has the same form as iterated binding, but only one binding is created. That is, of the possible iterated bindings, one is selected in a nondeterministic manner.

Binders may include a `where` clause to constrain the binding. In this case, the bindings are filtered to only those where the expression given in the `where` clause has the value `true`. The expression may refer to names introduced in the pattern that precedes it.

Example 21 Simple, iterated and nondeterministic bindings

```
Main()
  let a = 1
  let b = 2
  step foreach i in {a, b, 3}   // iterated binding
    WriteLine(i)

  step                          // nondeterministic choice
    choose x in {a, b, 3}
      WriteLine(x)
```

```
  step                          // nested binding
    let suits = {"Hearts", "Spades", "Clubs", "Diamonds"}
    let numbers = {"Ace", "2", "3", "4", "5", "6", "7", "8",
                   "9", "10", "Jack", "Queen", "King"}
    let deck = {(n, s) | n in numbers, s in suits}
    WriteLine(Deck)       // prints 52 pairs in arbitrary order
```

4.7.1 Parallel binding semantics

Iterated bindings may occur with sequential or parallel semantics, depending on the context where they appear. This is a feature of AsmL that differs from other programming languages. For example, the expression `forall i in {1, 2, 3} holds i < 4` creates three bindings for the identifier i. However, these bindings are simultaneous, not sequential (that is, they occur in parallel). You cannot assume that the bindings occur in sequence, one after another.

4.7.2 Order of bindings

Iterated bindings that operate over sequences occur in the same order as the sequence. Iterated bindings over maps and sets are unordered.

5 Types

A *type* characterizes a collection of values called the type's *domain*.

Types are not values. Instead, types constrain which values may appear in a given context. For example, an error will occur if the user attempts to update a variable with a value that is outside of domain of the type declared for that variable. Similarly, an error will occur if arguments provided when a method is invoked violate the type constraints given for the method.

It is possible that a given value may be an element of more than one type.

5.1 Type expressions

```
typeExp     ::=  optionType { or optionType }
optionType  ::=  atomicType [ "?" ]
atomicType  ::=  typeName | "(" typeExp { "," typeExp } ")"
typeName    ::=  name [ typeArgs ]
typeArgs    ::=  of  optionType  [ to optionType ]
               |  of "<" typeExp { "," typeExp } ">"
```

Types are denoted by *type expressions*.

Example 22 Type expressions

```
var v1 as Integer                    // type given by name
var v2 as (Integer)                  // same as Integer

var v3 as Integer?                   // option type

var v4 as Set of <Integer>           // instantiated, 1 arg
var v5 as Set of Integer             // (alternate form)

var v6 as Map of <Integer, String>   // instantiated, 2 args
var v7 as Map of Integer to String   // (alternate form)

var v8 as (Integer, String)          // product type

var v9 as (Integer?, Set of String)? // nested type expression

var v10 as Integer or String         // disjunctive type

var v11 as (Integer or String)?      // nested type expression
```

Example 22 shows the declaration of eleven variables. Each variable is declared as being of a type given by the type expression that follows the as keyword. The keyword `var` is short for "variable."

The following subsections describe the various kinds of type expressions.

5.1.1 Disjunctive types

A *disjunctive type* in the form *t* `or` *s* includes all of the values of type *t* plus the values of type *s*.

Example 23 Disjunctive type

```
MyFun(x as Integer or String) as String
  if x is Integer then
    return "Found integer."
  else
    return "Found string."

Main()
  step
    WriteLine(MyFun(2))       // prints "Found integer."
  step
    WriteLine(MyFun("abc"))   // prints "Found string."
  step
    WriteLine(MyFun(3.0))     // causes type error
```

Example 23 shows a function that accepts either an `Integer` or a `String` as its argument. A type error occurs when the program passes a value of type double (3.0) to the function.

5.1.2 Option types

An *option type* in the form *t*? includes all of the values of type *t* plus the special value `null`. An option type is just shorthand syntax for the frequently used disjunctive type *t* `or Null`.

For example, a variable declared using the option type `Boolean?` could contain either of the Boolean values `true` and `false` or the value `null`.

Note that unlike other many other languages, class types in AsmL do not include the null value in their domains. Contexts that permit a null value must indicate this explicitly by using an appropriate option type or disjunctive type.

5.1.3 Product types

A *product type* is an ordering of two or more types in the form (*t*₁, *t*₂, …).

For example, the type `(Integer, String)` has as values all pairs whose first element is an `Integer` and whose second element is a `String`. Thus, the pair `(1, "abc")` is a value of type `(Integer, String)`. (The values of product types are called *tuples* and are denoted inside parentheses with comma-delimited expressions.)

A *parenthesized type form* `(` *t* `)` is equivalent to *t*. The parenthesized type form is not a product type.

5.1.4 Named types

A type may be given by name *Named types* may either be built-ins such as `Integer` and `String` (see section 5.3 below), or they may be user-declared (see section 5.5 below).

5.1.5 Instantiated types

A type name followed by type arguments denotes an *instantiated type*. Type arguments are recognizable by the keyword `of`.

AsmL provides for type families. Types that come from type families are called instantiated types. For example, `Set of Integer`, `Set of String` and `Set of Char` are instantiated types that come from the built-in type family, `Set`, that defines generic operations for unordered collections of distinct elements.

Note that type families are not themselves types. In other words, `Set` is not a type but `Set of Integer` is.

Type arguments are given by the keyword `of` followed by a sequence of comma-delimited type expressions within angle brackets ("<" and ">"). For example, `of <Integer>` and `of <String, Integer, Integer>` are type arguments.

If a type argument includes only one type, then the angle brackets may be omitted, as in `Set of Integer`.

If there are two type arguments, the syntax "`of` *t1* `to` *t2*" may be used to mean "`of` <*t1*, *t2*>".

Example 22 above includes instantiated types with type arguments.

Example 24 Type families

```
structure Bucket of T
  maxBucketSize as Integer = 10
  contents as Set of T
  IsBucketSizeOK() as Boolean
    return Size(contents) <= maxBucketSize

Main()
  var b as Bucket of Integer
  step
     b := Bucket({1, 2, 3})
  step
     if (b.IsBucketSizeOK()) then
       WriteLine("Bucket b is not too big.")
```

Example 24 shows the declaration of a type family `Bucket`. The declaration of local variable `b` in the `Main()` method refers to a specific instantiated type `Bucket of Integer` taken from the type family `Bucket`. In other words, `Bucket` is a generic family of types from which any number of instantiated types may be drawn (based on the specific choice of type `T` in each instantiated type).

Note to users

Type families are often used to describe collections.

5.2 Operations on types

Types support three operations: membership testing, enumeration of values and conversion.

With membership testing, it is possible to determine whether any particular value is in the domain of a given type by means of the operator "`is`" This is further described in section 7.10.2 below.

For some types (called *enumerated types*) it possible to query for all values of a type's domain. The syntax is "`enum of T`"; the value produced is a set of values of type `T`. See section 7.13 below for more.

Type conversion occurs using the operator "as". The form *exp* `as` *typeExp* applies an appropriate conversion operation to the value given by *exp*. AsmL uses the CLS convention for defining conversion operations. See section 7.10.3 below.

Example 25 Type operations

```
class Color
  Red
  Green
  Blue

Main()
  step
    WriteLine( enum of Color)   // prints {Red, Green, Blue}
  step
    if Blue is Color
      let x = Blue as Short + 1s
      WriteLine(x)
```

Example 25 illustrates the three type operations.

5.3 Built-in types

AsmL includes the following *built-in types*.

Type	Description
`Null`	The `null` value
`Boolean`	The values `true` and `false`
`Byte`	8-bit signed integers

`Short`	16-bit signed integer s
`Integer`	32-bit signed integer type
`Long`	64-bit signed integer type
`Float`	Single-precision 32-bit floating-point format type as specified in IEEE 754.
`Double`	Double-precision 64-bit floating-point format type as specified in IEEE 754
`Char`	Unicode character
`String`	Unicode character string; e.g., `"abc"`

AsmL includes the following built-in type families for collections of values.

Type Family	Description
`Set of T`	Unordered, finite collections of distinct elements of type `T`
`Seq of T`	Ordered, finite sequences of elements of type `T`
`Map of T to S`	Tables that map distinct keys of type `T` to values of type `S`

Values of the built-in types are given by literals (see 2.6 above) and expressions. The AsmL library provides additional operations for built-in types. See section 11 below for a list of library operations.

Note that type `String` is distinct from the instantiated type `Seq of Char` even though they support almost the same set of operations.

All of the AsmL-provided types are structures (see 5.5 below). This means that semantic equality (or structural equivalence) forms the basis of equality testing for built-in types.

Note to users

Although semantic or structural equality is common in mathematics, it is less common in the tradition of commercial programming languages.

For example, with structural equality two sequences are considered to be the same value if they contain the same number elements and each element is equal.

One consequence of this view of object equality is that there is no notion of "pointers," "references" or "shared memory" for values of any of the built-in types. This means, for example, that two variables, each containing the same sequence of Integers, may be independently updated.

5.4 Subtypes

A type may be a *subtype* of several other types. The hierarchy of types given by the type-subtype relation is a directed, acyclic graph.

If type `T` is a subtype of type `S`, then each value in the domain of `T` is also in the domain of `S`. In other words, all of the constraints associated with type `S` apply to contexts that require a subtype of `S`. A subtype relationship may be declared using the "extends" or "implements" keywords (see section 5.5.1 below).

A type `T` that is a subtype of S is said to be a *direct subtype* of type `S` if `T` is not a subtype of any other subtype of `S`.

Type `S` is said to be a *supertype* of type `T` if `T` is a subtype of `S`. In like manner, type `S` is a *direct supertype* of type `T` if `T` is a direct subtype of `S`.

Subtype relationships extend through instantiations of type families of structure types but not through instantiations of type families of class types. For example, if `T` is a subtype of type `S` then the instantiated type `Set of T` is a subtype of type `Set of S`, since `Set of T` is a structure. In contrast, for the type family defined by "`class Foo of X ...`", `Foo of T` would not be a subtype of `Foo of S` when `T` is a subtype of `S`.

5.5 Type Declarations

```
type          ::= [ attributes ] { typeModifier }
                  ( class | structure | interface |
                    enum | delegate | constrainedType )
```

Type declarations introduce new named types, or if type parameters are given, new type families. User-declared types (or type families) may be classes, structures, interfaces, enumerations, delegates or constrained types.

In the discussion that follows we use the term "type" to mean a named type. This includes, if type parameters are present in the type declaration, any instantiated type generated from a type family. See section 5.1.5 above for more information about instantiated types.

A type's members—for example, its fields and methods—consist of *local members* (whose declarations are nested within the type's declaration) as well as all members declared in the type's supertypes. A local method may specialize (that is, override or replace) a method given in a supertype. Fields may not be specialized by subtypes.

Attributes and type modifiers are provided for compatibility with Microsoft's Common Language Specification (CLS). They are described below in section 10.

Delegates are provided for compatibility with CLS. They are also described below.

5.5.1 User-declared subtypes

Type declarations may augment the type hierarchy (that is, establish new subtype relations) by means of `extends` and `implements` clauses.

The types identified by the `extends` and `implements` clauses indicate the direct supertypes of the type being declared. For a type family `T`, the direct supertypes of an instantiated type `T of <T1, T2, …>` are given by substituting its type arguments into each type family that appears in an `extends` or `implements` clause of `T`'s declaration.

Subtypes introduced by `extends` must match the kind of declaration; for instance, it is an error for a class to extend a structure or interface. Classes extend classes; structures extend structures; and interfaces extend other interfaces.

Classes and structures may extend only one other class or structure; interfaces may extend any number of other interfaces. However, even if an interface appears multiple times in the transitive closure of another interface's direct supertypes, the interface contributes its members to the derived interface only once. In other words, the same type in several paths of the graph of direct supertypes denotes the same instance of this supertype.

Classes and structures are said to *implement* the interfaces given by their `implements` clause. (Interfaces may not implement anything.) Unless preceded by the keyword "`abstract`," a class or structure that includes an `implements` clause must provide a method (with method body) for each method of interface that is a supertype of the class or structure.

All interfaces implicitly extend the built-in interface `Object`. All classes and structures implicitly implement `Object`. (AsmL provides the implementation.)

5.5.2 Interface declarations

```
interface    ::= interface id [ typeParams ] [ typeRelations ]
                    [ declaration ]

typeParams ::= of id [ to id ]
             |  of "<" typeParam {"," typeParam } ">"

typeParam  ::= id [ typeRelations ]

typeRelations    ::= extends typeExps [ implements typeExps ]
                  |   implements typeExps

typeExps   ::= typeExp { and typeExps }
```

Interface declarations define new abstract types. (An *abstract type* has no corresponding constructor —the values of an abstract type are only of those of its subtypes.)

Interfaces may not contain field declarations, and a method declared within an interface may not provide a method body. Thus, interfaces provide a vocabulary (or *type signature*) without implementation. Methods are described in section 6 below.

Implementation Note

The current AsmL compiler does not issue an error message if a body is provided for a method declared in an interface. (The method body is ignored.)

This will be corrected in a future release.

See section 5.5.1 above for how new the `extends` clause may establish new subtype relations.

Example 26 Interface declaration

```
interface IStream
  Read() as Char
```

As mentioned in section 5.1.5 above, if type parameters are given, then a type declaration (including declarations for interfaces, classes and structures) introduces a type family. Example 24 above gives an example of a user-declared type family.

See section 5.5.7 below for information on type relation constraints that may appear in type parameters.

5.5.3 Datatype declarations

```
class        ::= [ enumerated ] class id [ typeParams ]
                 [ typeRelations ]
                 [ variantOrDecl ]
structure    ::= structure id [ typeParams ]
                 [ typeRelations ]
                 [ variantOrDecl ]

variantOrDecl     ::=        declaration | variant
```

A *datatype declaration* introduces a new type of structure or class (or, if type parameters are present, a new type family). Unlike interface declarations, datatype declarations may include data fields.

Structures and classes are operationally distinct. The difference between them is described in section 4.3 above.

See section 5.5.1 above for how new subtype relations may be established by datatype declarations.

See section 5.5.4 below for datatype variants.

See Section 5.5.7 below for information on type relation constraints that may appear in type parameters.

See Section 6 below for a description of members.

AsmL does not provide the ordering operations <, >, >= and <= for structures.

5.5.4 Datatype variants

variant ::= `case` *id* [*declaration*]

Class and structure declarations may include *variants*, or subtypes declared with special in-line syntax.

A variant of datatype `T` expands into a new type declaration that extends `T`. The name the new type is given after the `case` keyword, followed by member declarations of the new type.

Note to users

Cases should be used when the intent is to emphasize that a datatype occurs in several variant forms (and that the variants have no independent use).

In contrast, declaring each variant as a lexically independent datatype emphasizes the independence of each subtype in an object-oriented style.

Example 27 Datatype variants

```
structure List of T
  case Nil
  case Cons
    head as T
    tail as List of T

first of T  (l as List of T) as T
  match l
    Nil()     : throw NoSuchElementException("first")
    Cons(h, _): return h

Main()
  x = Cons("a", Nil of String())
  WriteLine(first(x))              // prints "a"
```

Example 27 shows a typical use of datatype variants. Instantiated types based on the `List` type family have two variants: either the value `Nil()` that represents the empty list or a pair consisting of a head element and a tail list. Note that operations defined for datatypes with cases often use pattern matching (via the `match` operator, see section 4.6 above) to process individual cases based on the variant's form.

Example 27 can be translated as the following:

Example 28 Structure case as subtypes

```
structure List of T

structure Nil of T extends List of T

structure Cons of T extends List of T
  head as T
  tail as List of T

first of T  (l as List of T) as T
  match l
    Nil()     : throw NoSuchElementException("first")
    Cons(h, _): return h

Main()
  x = Cons("a", Nil of String())
  WriteLine(first(x))              // prints "a"
```

5.5.5 Enumerations

enum ::= `enum` *id* [`extends` *typeExp*] [*element*]
element ::= *id* [`"="` *exp*]

Enumeration declarations or introduce new types (called *enums*) whose domains are given statically within the declaration

Enums may be mapped to the integer types, `Byte`, `Short`, `Integer` and `Long` if an `extends` clause is provided in the declaration. In this case, each element of the enumeration will be a value of the given type, and the enum will be a subtype of the given type. If no `extends` clause is present, "`extends Integer`" is taken as the default.

By default, the first element of an enum is the value `1b`, `1s`, `1`, or `1l`, depending on the enum 's supertype. User-provided numeric values may be associated with an element of an enum if an equals sign ("=") follows the element. By default, elements without user-provided numeric values increase incrementally by one. If continued definitions are used, then order is arbitrary between blocks.

Enumerations support the <, >, <=, >= operators.

Example 29 Enumeration with user-provided values

```
enum MyEnum extends Integer
  E_E1 = 10
  E_E2                                    // has value 11
  E_E3 = 20

Main()
  let x = E_E1
  match x
    E_E1: WriteLine("case 1")            // prints "case 1"
    E_E2: WriteLine("case 2")            // doesn't print
    E_E3: WriteLine("case 12")           // doesn't print
```

Range comprehension is defined for enumerations.

Example 30 Enum Ranges

```
enum Color
  Red
  Orange
  Yellow
  Green
  Blue
  Indigo
  Violet

x = {Orange..Blue}
// same as {Orange, Yellow, Green, Blue}

IsWarmColor(c as Color) as Boolean
   return (x < Green)
```

Enums are a subtype of `Integer` or, if so declared, a subtype of any other number type. You can say:

```
enum LongBits extends System.Int64
  mask1 = 0x101010101010110
```

It is possible to use enums as bit fields. Enum values are subtypes of Integer, so you can use `BitAnd`, `BitOr`, etc. with them. Note that the result is an `Integer` and must be explicitly converted back into an enum value:

```
enum StatusCode
  ActiveNoError   = 0
  InactiveNoError = 1
  ActiveError     = 2
  InactiveError   = 3
```

```
IsError(x as StatusCode) as Boolean
  return (BitOr(x, 2) = 1)

IsActive(x as StatusCode) as Boolean
  return (BitOr(x, 1) = 0)
```

5.5.6 Constrained types

constrainedType `::= type` *id* `[` *typeParams* `] [ "="` *valueExp* `]`
valueExp `::=` *typeExp* `[ where` *exp* `]`

A declaration of a *constrained type* introduces a new named type (or type family if type parameters are given) that is defined in terms of another type. The name "constrained type" comes from the fact the new type may be defined in way that excludes some values (via the "where" clause) of the type on which it is based.

If a value expression is provided, it must be a Boolean valued. The keyword `value` is used as a parameter that will be given an appropriate binding when the constraint is checked.

Constrained types are abstract. (This means that they define no constructors of their own. The constructor of the underlying type is used instead.)

The declaration of a constrained type establishes a new subtype relation. The constrained type is a direct subtype of the type given after the "=" sign. In the example below, type `SmallInt` is a subtype of `Integer`.

Example 31 Constrained type

```
type SmallInt = Integer where value in {1, 2, 3}

type IntOrString = Integer or String

MyFun1(x as SmallInt) as IntOrString
  match x
    1: return 1
    2: return 2
    3: return "Neither 1 nor 2"

MyFun2(x as SmallInt, y as SmallInt) as SmallInt
  return ((x + y) mod 3) + 1

Main()
  step
    WriteLine(MyFun1(1))          // prints 1
  step
    WriteLine(MyFun1(3))          // prints "Neither 1 nor 2"
  step
    WriteLine(MyFun1(4))          // causes type error
```

Example 31 declares two constrained types, `SmallInt` and `IntOrString`. The example shows how a constrained type can serve as a "type alias," or abbreviated way to write a complicated type expression. It also shows how a constrained type can be used to factor data-oriented preconditions into to the type declaration. It was not necessary to give preconditions to functions `MyFun1` and `MyFun2` because the relevant constraint had already been factored into the type `SmallInt`.

The idea behind constrained types is that it is often convenient to factor common preconditions into the type system, rather than by repeating identical constraint expressions in many places. Here is an example:

```
class Event
  var IsCurrent as Boolean = false

GetTimeUntilStart(e as Event) as Time
  require e.IsCurrent

GetTimeUntilFinish(e as Event) as Time
  require e.IsCurrent

NotifyOrganizer(e as Event)
  require e.IsCurrent
```

Each of the methods contains a common precondition that constrains the applicability of the method to "current" events. In AsmL 2, we can factor this constraint into the type system:

```
type CurrentEvent = Event where value.IsCurrent

GetTimeUntilStart(e as CurrentEvent) as Time

GetTimeUntilFinish(e as CurrentEvent) as Time

NotifyOrganizer(e as CurrentEvent)
```

The idea is that the "`IsCurrent`" constraint would apply as if it were a precondition.

Implementation Note

This feature is only partially implemented in the current distribution. In the present release of AsmL 2, the constraints that follow the "where" clause are permitted syntactically but not checked at runtime. This will be changed in an updated distribution.

Nonetheless, it is recommended that constrained types be used as documentation of the modeler's intent.

It is recommended as a matter of style to factor common preconditions (i.e., preconditions that appear identically in many methods) into a constrained type declaration.

5.5.7 Constraints on type parameters

The type parameters given in the declaration of a type family `F` may include optional *type relation constraints* that limit which types may be used when creating instantiated types based on `F`.

To perform this check, each type in the type arguments of an instantiated type is compared with the type relation constraints given in the declaration of the applicable type family. An error occurs if any of the type arguments is not a subtype of every type given in the type relation constraint for that type argument.

The syntax of type relation constraints is given above in section 5.5.2.

Example 32 Constraints on type parameters

```
interface ILabel
 Label() as String

structure LabeledList of <T implements ILabel>
  MySeq as Seq of T
  Labels() as Seq of String
    return [i.Label() | i in MySeq]

class Foo implements ILabel
  name as String
  Label() as String
    return name

Main()
  let f1 = new Foo("Label 1")
  let f2 = new Foo("Label 2")
  var myList as LabeledList of Foo
  step
    myList := LabeledList([f1, f2])
  step
    WriteLine(myList.Labels())   // prints ["Label 1", "Label
2"]
```

Example 32 shows an example of a type family `LabeledList` whose instantiations are required to be based upon types that implement the `ILabel` interface. Angle brackets ("<" and ">" are used to delimit the type parameters and prevent the type constraints from being misinterpreted as `LabeledList`'s `implements` clause.

The `Foo` class implements `ILabel`; therefore, it is permitted may be used to create an instantiated type based on `LabeledList`.

6 Members

```
member    ::= [ attributes ] { memberModifier }
              ( constant | variable | method |
                constraint | property | event )

memberModifier ::= shared | virtual | override
                 |  extendedMemberModifier
```

Member declarations define the static vocabulary (such as field names and method signatures) that gives the operational behavior of the program.

Member declarations consist of fields (either constant or variable), methods, constraints, properties and events.

Members may be prefixed by attributes and method modifiers. These are described in section 10 below.

Properties and events are provided for compatibility with the Common Language Specification (CLS) and are described below in sections 0 and 0.

6.1 Fields

```
constant    ::= { fieldModifier } [ const ] id
                  ( as typeExp [ "=" exp ] | "=" exp )

variable    ::= { fieldModifier }
              var id ( as typeExp ["=" exp ] | "=" exp )
```

Field declarations introduce names that will be associated with values at runtime.

Each distinct occurrence of a relationship between a field name and a value during the program's run is called a *field instance*. A field declaration may result in more than one such name/value association.

For example, if the field defines an instance-level variable in a class, there will be one name/value association of the given name for each instance of the class.

Section 6.1.5 below describes the various contexts that produce field instances.

6.1.1 Type constraints on values of field instances

All fields have an associated type that constrains which values may be referred to using the field name. An error occurs if an attempt is made to associate a field name to a value that is not in the domain of the type declared for the field. The field's type constraint applies to all field instances when they are initialized and, if the field is a variable, when they are updated to a new value.

The type constraint may be explicitly declared by means of the as clause (in the form `as` *type*) or given implicitly by the type associated with the field initialization expression. A field that does not include a type constraint must specify an initial value.

6.1.2 Constants

A field instance whose field declaration does not contain the keyword `var` is called a *constant*. Constants may be optionally prefixed by the keyword `const` (for constant).

The value of a constant is its initial value. A compiler error occurs if an update statement attempts to change the value of a constant.

Indexing fields (i.e., those declared within a structure) may not use the keyword `const`.

6.1.3 Variables

If the field declaration includes the keyword `var`, then each of its field instances is a *variable*.

Variables are implicitly parameterized by a step of an abstract state machine. In other words, asking for the value of a variable only makes sense with respect to a particular step of a given abstract state machine. See section 8.3 below for information how abstract state machines are created.

Update statements (see 8.1 below) are the only mechanism for changing the value of a variable. Updates to variables occur atomically during the step transition of the abstract state machine that provides the context for the update operation.

Indexing fields (i.e., those declared within a structure) may not use the keyword `var`.

6.1.4 Initialization of field instances

A field declaration may optionally include a *field initialization expression* after an equal sign ("`=`") to specify the initial value of each field instance that arises as a result of the declaration.

If there is no field initialization expression, then field instances are initialized in the following way:

- If the field instance is created by invoking a default construction expression for an enclosing type, then the initial value will be given as an argument to the construction expression. The order of parameters in the default constructor is the same order as the field names appear in the type declaration.
- If the field instance is created by invoking a user-provided construction expression for an enclosing type, then the initial value will be given by a binding expression in the body of the constructor declaration. This is described below in section 6.2.11.
- In all other cases, the variable will remain uninitialized and any attempt to read its value will fail with an error message.

Field initialization expressions are evaluated during the initialization of the runtime context for each field instance. The initialization expression of a global field occurs before the program runs. The initialization of instance-level field occurs when a new instance of the class is created.

The initialization of instance fields is atomic. In other words, the initialization occurs in a single step. There is no order of initialization.

6.1.5 Kinds of fields

A field is said to be a global field, a local field, an instance-level field or an indexing field depending on the form and lexical context of its declaration.

A field declared outside of a type declaration, or within a type declaration using the `shared` keyword, is called a *global field*. Global fields produce just one field instance for the entire run of a program.

A field declared in a class without the `shared` keyword is called an *instance-level field*. These fields have one instantiation per instance of the class in which the field declaration occurs.

A field declared in a structure without the `shared` keyword is called an *indexing field*. Indexing fields (that is, fields declared in structures) are never instantiated as field instances. Instead, indexing field names are *indexers*, or labels that identify the constituent parts of a compound value.

Some fields arise dynamically from the evaluation of expressions. These are called local fields and appear in certain expression contexts as described in section 7.2 below.

Note to users

Informally, one can think of each field instance as a distinct area of the system's memory. The memory associated with a field instance is never shared with any other field instance. For example, updating a variable has an effect only upon the field instance being updated (there is no "aliasing" in AsmL).

Nonetheless, the memory associated with a field instance may be structured into sub-elements and may even store a variable number of elements, including complex, nested data structures such as trees and graphs.

One can view indexing fields as a way to access the components of a structured memory in the same manner as bit-fields in languages like C.

Another way to see the difference between values of structure types and values of class types is as the difference between call-by-value and call-by-reference. A method call that takes a structure value as an argument can never modify that structure, since call-by-value semantics will be used. In contrast, a method that takes an instance of a class as an argument could modify one of the variables defined by that instance. Instances of classes use call-by-reference semantics.

(A structure is a series of values grouped together, while an instance of a class is a unique object identifier.)

6.1.6 Indexing field names

Example 33 Field containing a compound value

```
structure Point2
  x as Integer
  y as Integer

var myPoint as Point2 = Point2(0, 0)

Main()
  step
    myPoint.x := 2
    WriteLine(myPoint.x)    // prints 0
  step
    WriteLine(myPoint.x)    // prints 2
```

Example 33 shows a single field instance (in this case, a global variable named `myPoint`) that contains a compound value whose structure is given by `Point2`. The value of the global `myPoint` field instance is indexed by named x- and y-coordinates.

Note that the keyword "`var`" indicates that the field `myPoint` may be updated. The indexers `x` and `y` do not need to be annotated with `var` (and in fact may not be) because they never correspond to independent field instances. Instead, to determine whether `x` can be updated, one needs always to find the field instance that contains the value of type `Point2`.

6.1.7 Indexing parameters

When compound values contain a variable number of components, they use *indexing parameters* instead of indexing field names as labels.

For example, sequences use integer subscripts as indexers, while maps use arbitrary values as subscripts. It is possible in the case of maps for indexing parameters to be given as a tuple.

The indices of a sequence begin at zero.

Example 34 Indexing parameters vs. indexing field names

```
myList   = ["a", "b", "c"]
myStruct = Point3(1, 2, 3)

structure Point3
  x as Integer
  y as Integer
  z as Integer

Main() =
   step WriteLine(myList(1))         // prints "b"
   step WriteLine(myStruct.y)        // prints 2
```

Note that the compound values `myList` and `myStruct` are both composed of three constituent values. In the case of the sequence, these components are labeled by integer subscripts. The value named `myStruct` uses indexing field names to label its internal components.

Indexing parameters are tuples of expressions that evaluate to values of arbitrary types. Indexing parameters can be thought of as a generalization of array subscripts.

AsmL provides four built-in datatypes that support indexing parameters: `Set`, `Map`, `Sequence` and `String`.

The syntax for applying an indexing parameter is *m* (*arg1*, *arg2*, ...). For example, if `m` is a `Map of (String, Integer) to Integer`, then `m("abc", 1)` would be used as the lookup operation.

6.2 Methods

```
method     ::= [ methodKind ] methodId [ typeParams ]
               signature [ stm ]
methodKind ::= function | procedure
methodId   ::= name | operator ( binaryOp | unaryOp )
signature  ::= params [ result ]
result     ::= as typeExp
params     ::= "(" [ param { "," param } ] ")"
param      ::= [ attributes ] [ paramModifier ]
               [ id as ] typeExp
```

A *method declaration* associates a name with a parameterized expression.

During the run of the program, a method may be invoked by supplying values for each of the method's formal parameters. Method invocation always occurs within the context of an abstract state machine.

6.2.1 Kinds of methods

Depending on its form and context, a method declaration (also called a *method*) is one of four kinds: a global method, an instance-level method, a value-level method or a constructor method.

Methods declared outside of a type declaration, methods declared as `shared` within a type declaration, converters and operators are called *global methods*.

Methods declared without the `shared` keyword within a class declaration are called *instance-level methods*.

Methods declared without the `shared` keyword within a structure declaration are called *value-level methods*.

Constructor methods have the same name as the enclosing datatype. Constructor methods are described in section 6.2.11 below.

Example 35 Kinds of methods

```
m1()
  WriteLine(“M1”)                 // global method

class C1
  shared m2()                   // global method
    WriteLine(“M2”)
  m3()                          // instance-level method
    WriteLine(“M3”)

structure S1
  shared m4()                   // global method
    WriteLine(“M4”)
  m5()                          // value-level method
    WriteLine(“M5”)

Main()
  c = new C1()
  s = S1()

  step m1()                     // invoke each method
  step m2()
  step c.m3()
  step m4()
  step m5(s)
```

6.2.2 Functions and procedures

The keywords `function` and `procedure` may optionally be used to annotate whether a method may make updates to state. Methods annotated with the keyword `function` may make no updates to state. Methods prefixed by `procedure` may change state.

Implementation note

The current AsmL compiler treats the annotation of function or procedure as a comment. A future version of the tool will perform conformance checking for this attribute.

6.2.3 Operators

AsmL supports a set of *operators* for built-in types. In addition to the predefined implementations, user-defined implementations can be introduced using operator declarations. Operator declarations are top-level declarations; they may not be nested inside of a type declaration.

Dynamic method dispatch (see section 6.2.7 below) applies to the first parameter of an operator. Static method dispatch (see section 6.2.6 below) applies to all parameters of an operator.

Example 36 Example: Operator declaration

```
structure Rational
  num as Integer
  denom as Integer

operator + (x as Rational, y as Rational) as Rational
```

6.2.4 Conversion methods

AsmL is very restrictive with implicit conversions. AsmL does not provide implicit conversions between types, except to allow a subtype to be used in contexts where the supertype is expected.

Every other conversion is defined by global user defined ”ToTARGETTYPE” methods, that take a value of the source type, and return a value of the target type.

Example 37 Example: Conversion methods

```
structure Dollar
  value as Integer

ToInteger(x as Dollar) as Integer      // global method
ToDollar(x as Integer) as Dollar       // global method
```

Such a conversion method is permitted from source type S to target type T only if the following is true:

- S and T are different types
- S is not subclass of T, nor is T a subclass of S.
- Neither S nor T is an interface.
- Neither S nor T is a generic type.

Implementation Note

These conditions are not yet checked. A future version of AsmL will implement them.

6.2.5 Method parameters

When invoked during the program's run, the method's formal parameters are bound to actual arguments. In other words, method calls create a new set of bindings, specific to a new runtime context for that invocation, of values to the formal parameters.

A method's *formal parameters* are the names given in the method's parameter list. The value bound to each formal parameter must be a subtype of the type associated with the corresponding formal parameter name. The number of formal parameters in a method declaration is fixed.

Method declarations that do not use the keyword `shared` and that appear within a type declaration are implicitly parameterized by a formal parameter, `me`, that is bound in the runtime context to an entity of the type given by the enclosing type declaration.

Note that the forms `x.f()` and `f(x)` are equivalent in AsmL. Methods may be invoked using either form. See section 7.11 below. When we speak of a method's parameters, we include the implicit initial parameter "`me`."

6.2.6 Static method selection

Static method selection provides additional flexibility in naming methods by allowing argument types to disambiguate method names. This is sometimes referred to as *overloading* method names and is determined by the program text itself (i.e., the declared types) and not by any runtime type information.

The built-in operator "+" is an example: `1 + 3` is distinct from `"abc" + "def"`. The first means arithmetic plus for integers; the second means string concatenation. We can tell statically which version of "+" is intended based on the argument types provided, integers and strings in this example.

The following describes AsmL's rules for overloading method names.

A method is said to be *applicable* if the type of the each argument (as statically deduced from the program's text) is the same type as or a subtype of the type given in the method's parameter list. Methods prefixed by the `override` keyword are excluded from the set of applicable methods used to determine static method selection. (Such methods are dynamically dispatched, as described in section 6.2.7 below.)

For each method invocation, only one of the applicable methods (called the *selected method*) will be invoked. The selected method is the applicable method with the most specific declared parameters.

The parameters of method declaration M1 are said to be *more specific* than those of method declaration M2 if the type of each parameter declared in M1 is a subtype of (or the same type as) the corresponding parameter declared in M2, provided that at least one type given in M1's parameters is a strict subtype of (that is, not the same type as) its corresponding parameter in M2.

An error occurs if the most specific method of any set of applicable methods cannot be determined.

An error occurs if, for a given invocation, no applicable methods have been declared.

Disjunctive types (see section 5.1.1 above) may participate in static method selection. The type `T or S` will be considered to be supertype of both type `T` and type `S` for the purposes of static method selection, as described above. The types `S or T` and `T or S` will be considered to be the same type. The type `T or T` is the same as type `T`.

Constrained types (see section 5.5.6 above) may be used for the purposes of static method selection. Resolution of overloading will occur as if the underlying type referenced by the constrained type had been given in the parameter list of the method declaration.

If a method invocation appears within a type declaration and could be interpreted either as a call to a global method and as a call to a method declared within the type (or its supertypes), then the global method is ignored. In other words, instance-level and value-level methods are interpreted as `me`.*id* (*arg1*, *arg2*, …) if the context allows. This interpretation excludes any global methods in the form *id* (*arg1*, *arg2*, …) from the set of applicable methods.

Example 38 Static method selection

```
MyPrint(a as Integer, b as Integer)
  WriteLine("Two integers")

MyPrint(a as Integer)
  WriteLine("Integer")
```

```
MyPrint(a as Null)
  WriteLine("Null")

MyPrint(a as Integer?)
  WriteLine("Integer or Null")

type Token = Integer or String

MyPrint(a as Token)
  WriteLine("Integer or String")

type SmallToken = String where Length(value) < 10

MyPrint(a as SmallToken)
  WriteLine("String")

Main()
  let a as String = "abc"
  let b as String or Integer = 1
  let c as Integer? = 1
  let d = 1                  // implicit type "Integer"
  let e = null               // implicit type "Null"
  let f as Integer? = null
  let g = "A long string"    // implicit type "String"

  MyPrint(a)                // prints "String"
  MyPrint(b)                // prints "Integer or String"
  MyPrint(c)                // prints "Integer or Null"
  MyPrint(d)                // prints "Integer"
  MyPrint(d, d)             // prints "Two integers"
  MyPrint(e)                // prints "Null"
  MyPrint(f)                // prints "Integer or Null"
  MyPrint(g)                // runtime error
```

Example 38 illustrates the fact that static method selection is determined by the declared types of the arguments provided and not their actual values. Note that for the value g, the selected method was `MyPrint(a as SmallToken)`, since the `SmallToken` and `String` are equivalent for static method resolution and since `String` is a subtype of "`String or Integer`." This invocation results in a runtime error because the value of g does not satisfy the constraint given by `SmallToken` (that the string length be less than `10`). The presence of a type constraint does not affect overloading.

Implementation Note

The current AsmL implementation does not support the full overloading functionality described in this section. Currently, the overloading "`Integer?`" is not supported. Also, the overloading of disjunctive types is not fully implemented.

6.2.7 Dynamic method selection

In addition to the static method selection described in the previous section, AsmL supports dynamic method selection, also called dynamic method dispatch. *Dynamic method selection* allows the choice of method to be deferred until an actual parameter is provided at runtime

AsmL follows the conventions of the Microsoft's Common Language Specification (CLS) in its handling of dynamic method selection. Only the first argument to a method (typically , the implicit argument named "`me`") affects dynamic method selection. Selection is based on the most specific datatype for this parameter.

If a method is to be eligible for dynamic method dispatch, it must be declared using the keyword "`virtual`." Any method that specializes a virtual method must be declared using the keyword "`override`."

Example 39 Dynamic method selection

```
class Food
  id as String
  virtual PrintName() as String
    return "<Food " + id + ">"

class Fruit extends Food

class Apple extends Fruit
  override PrintName() as String
    return "<Apple " + id + ">"

PrintNames(s as Seq of Food)
  step foreach f in s
    WriteLine(f.PrintName())

Main()
  PrintNames([new Apple("1"), new Fruit("2"), new Food("3")])
```

Example 39 shows a typical use of dynamic method dispatch. Running this example with cause the following to be printed:

```
<Apple 1>
<Food 2>
<Food 3>
```

The `PrintName()` method is dynamically selected.

6.2.8 Return values

A method declaration may specify the type of the value it returns.

Return values are optional.

The return value of a method is the return value of the statement block that forms its body. See the return statement (section 7.5) for more information.

6.2.9 Recursive methods

Method invocations can be recursive.

6.2.10 Type-parameterized, generic methods

A *generic method* has the same form as any other method declaration, except that one or more of the parameter and result types depend on locally defined type parameters. All of these type parameters are defined in the local type parameterization.

Like a parameterized type family (see section 5.1.5 above), a generic method represents a family of related methods. In order to instantiate a generic method with actual types, the actual types must either be specified either at the point of the application or they must be clear from the context of the method's invocation.

6.2.11 Constructor methods

AsmL2 allows for user-written constructors, as an alternative to the implicit, default constructor. Here is an example:

Example 40 User-provided constructor

```
class Foo
  var a as Integer
  const b as String

  Foo( b as String)
    a = b.Length
```

The constructor is given by a method whose name is the same as the name of the class or structure that contains it.

Fields of the structure or class are initialized by the bindings of the constructor. In other words, the local bindings of the constructor (including named arguments passed to the constructor) provide the initial values of fields of the same name.

The keyword "`me`" may not appear within the constructor method of a structure type. For classes, the keyword "`me`" may be used within the constructor method, but accessing (either reading or updating) any fields by means of the identifier "`me`" will cause an error.

A continuation constructor may be called from a base class using the syntax `mybase(`*arg1, arg2, …*`)`.

Example 41 User-provided constructor with inheritance

```
class Foo
   a as Integer
   b as String

class Bar extends Foo
   c as Boolean
   Bar(a', b', c')
      mybase(a', b')
      c = c'
```

If provided, the "`mybase`" constructor continuation must be the first statement in the constructor that contains it.

6.2.12 Disambiguation of method names

If a datatype implements two interfaces, it is possible for an ambiguity in method names to arise within an enclosing type declaration. This occurs when the two interfaces each declare a method of the same name and same argument types.

To handle this case, it is possible in AsmL to use a qualified name as the method identifier in a method declaration. The qualified name includes the type name of the interface that provided the method signature.

When invoking the method, either a type conversion operation must be used or the method must be called using the qualified name.

Example 42 Disambiguation of method names

```
interface IStream
  Read() as Integer

interface IReader
  Read() as Integer

class Foo implements IStream and IReader
  IStream.Read() as Integer
     return 1
  IReader.Read() as Integer
     return 2

Main()
  let f = new Foo()
  let s = f as IStream
  let r = f as IReader
  let val = (s.Read(), r.Read())
  WriteLine(val)              // prints (1, 2)
```

Implementation Note

The functionality described in this section is not yet implemented in the AsmL compiler. It is currently not possible to implement interfaces with identical methods.

6.3 Constraints

```
constraint   ::= constraint [ label ] exp
label        ::= ( id | literal ) ":"
```

A constraint is a Boolean-valued condition used to check the integrity of data-oriented restrictions. A constraint declared within a datatype must always be `true`, or an error will occur.

Example 43 Constraint declaration

```
structure Rational
  numerator as Integer
  denom as Integer

  constraint NonZeroDivsor : denom <> 0

Main()
  let r1 = Rational(1, 2)        // OK
  let r2 = Rational(2, 0)        // error occurs
```

7 Statements and Expressions

```
stm          ::= local
               |  assert
               |  choice
               |  return
               |  operationalStm
               |  exp

exp          ::= branchExp
               |  exceptExp
               |  quantifierExp
               |  selectExp
               |  binaryExp
               |  enum of typeExp
               |  type of typeExp
               |  do stm
               |  exploration

exps         ::= exp { "," exp }
```

Statements and expressions serve three purposes: 1) to express values in terms of other values, 2) to query the current state of variables and 3) to propose new values of variables that will take effect in the next step of an abstract state machine.

The syntax of AsmL allows an expression to be used whenever a statement is expected. In this sense AsmL expressions act as "statements" or "update operations" in addition to their traditional role of denoting values. (The converse is not true: statements may not be used in contexts that expect an expression.)

In this section (section 7) we describe statements and expressions that make no changes to state. Later, in section 8, we describe state-changing operations.

7.1 Statement blocks

When invoked at runtime, *statement blocks* (that is, a list of *stm* productions) create field instances for local fields, check runtime constraints, evaluate expressions and optionally yield a return value.

A number of contexts in the grammar expect statement blocks to provide the meaning of operations. Statement blocks occur inside:

- a method declaration (see section 6.2 above), as the *method body*, or operation performed when the method is invoked during the run of the program;
- a parallel update statement (see section 8.1.3 below), to effect the individual updates of each parallel binding;
- a step of a sequential block (see section 8.3 below), to configure an

abstract state machine's variables in the following step;

- a nondeterministic choice expression (see 7.4 below), to operate on the value chosen;
- each branch of an if-then-else expression (see section 7.6.1 below), to indicate the operation performed if the conditions given in the conditional guard expression are satisfied;
- each case of match expression (see section 7.6.2 below), as the operation performed when the case's pattern matches a given value;
- a try block (see section 7.7 below), as the protected operation;
- each case of exception handler (see section 7.7 below), as the operation performed when an exception matches the selection criteria of the handler.

A statement block begins with zero or more declarations of local fields. After the local declarations may appear zero or more assertions. After the assertions may appear zero or more expressions.

An optional `return` clause terminates the statement block. The expression given after the `return` keyword becomes the value of the statement block. If no `return` clause is used, the block does not return a value.

In general, AsmL statements execute in parallel. Updates to state do not take effect immediately. As a consequence, AsmL imposes only partial order on the evaluation of the expressions given in a statement block:

- The expressions that give the initial values of the local fields are evaluated prior to any precondition assertions. Currently, local field instances are initialized in the order of their appearance in the block. (In a future version of AsmL, local fields will be initialized using a partial order given by resolving any field-to-field value dependencies.)
- Preconditions will be evaluated prior to statement-level expressions in the block.
- Statement-level expressions will be evaluated prior to providing the return value to the calling context. If there is no return value, then the evaluation of statement-level expressions in the invocation of the block is not considered to be synchronous (that is the caller need not wait for completion). The order of evaluation of each expression in block is not constrained. This includes the evaluation of the expression that provides the return value.
- Postcondition assertions will be evaluated after the block's statement. An AsmL implementation must delay the evaluation of postconditions until all updates of the current step have been performed. In this sense, postconditions can be seen as constraints on the application of updates. There is no guarantee that the postcondition will be evaluated prior to the delivery of the statement block's return value in its invocation context.

7.2 Local fields

```
local        ::= letBinding
               | { localVariableModifier } localVar
letBinding   ::= [ let ] pat "=" exp
localVar     ::= ( var | initially ) id
                           ( as typeExp ["=" exp ] | "=" exp )
```

Statements that are similar to field declarations (see section 6.1 above) in form and meaning may occur within statement blocks as a means of introducing *local fields*, either constants or variables.

Local fields have one field instance (see section 6.1.5 above) for each invocation of their enclosing statement block. Local fields are both locally scoped and ephemeral; that is, they are visible in their scope during the lifetime of the runtime context associated with the particular invocation of the statement block.

The scope of local fields is the region of their statement block that follows their declaration. (There is one exception to this; see the "step" statement in section 8.3.3 below.)

Local constants are introduced as the result of pattern-based bindings in the form `let` *pattern* `"="` *exp* (see 4.6 above). A pattern-based binding may establish more than one name/value association.

Note to users

The `let` keyword is optional when introducing local constants; however, it s use is recommended as a matter of style to avoid confusion with the Boolean expression for equality testing, `x = y`.

As a way to introduce local variables the keyword "`var`" is interchangeable with the keyword "`initially`." The latter emphasizes the role of a local variable within the algorithm given in a method body.

Statements that introduce local variables have the form as variables given by field declarations.

Example 44 Local fields

```
class Identifier

Main()
  var x = new Identifier()
  let (a, b) = (“abc”, “def”)
  let c as String = a
  let y = x
```

Note to users

The expression that provides the initial value of a local field is evaluated only once in any invocation of a statement block.

This means that any local fields initialized by nondeterministic expressions (including expressions that return a different value every time they are invoked, such as the class construction operator "new"), can be relied upon to contain just one value for the duration of the invocation context.

7.3 Assertion statements

assert ::= *constraint* | *require* | *ensure*
require ::= `require` [*label*] *exp*
ensure ::= `ensure` [*label*] *exp*

An *assertion* constrains the behavior of the running program for the purposes of error checking. An AsmL implementation may optionally halt the program's `run` if an assertion's constraint has not been met, but assertions do not otherwise affect the meaning the program's run. In particular, a precondition or postcondition may not cause an update statement to be evaluated. If it does, an error will occur.

There are three forms of assertions: preconditions, postconditions and data-oriented constraints. These are introduced by the keywords `require`, `ensure` and `constraint`, respectively.

The expression given by a *precondition* is a predicate that must evaluate to `true` if the constraint is to be satisfied. The predicate is evaluated in a context that includes the statement block's local field instances.

The expression given by a *postcondition* is a predicate that must evaluate to `true` if the constraint is to be satisfied. The predicate is evaluated in a context that includes statement block's local field instances and, if the statement block includes a return statement, a binding of the identifier `result` to the statement block's return value.

Constraints introduced within statement blocks are have the same syntax as constraints declared as members. See section 6.3 above for the syntax. Like preconditions, constraints check that a Boolean condition is true. However, constraints offer the additional feature of checking that the condition is true even when updates to variables occur.

Example 45 Runtime assertion checking

```
Incr(x as Integer) as Integer
  require x >= 0
  ensure result = x + 1
  return ((((x + 1) * 2) - 2) / 2) + 1

Main()
  step WriteLine(Incr(1))
  step WriteLine(Incr(99))
```

The special form `resulting` *selectExpr* may be used within a postcondition to constrain the update set of the current step. The *resulting expression* returns the value that the variable designated by the selectExpr (see section 8.1.2) will have in the following sequential step of the current abstract state machine. This value is only known after the update set has been completly determined (that is, just prior to beginning of the subsequent sequential step).

Thus, the checking of a postcondition constraint that includes a "resulting expression" is not synchronized with the statement block in which it occurs. Instead, the constraint will be checked later, after all (parallel) updates have been calculated for the current step.

Example 46 Use of a resulting expression

```
var Counter = 1

Increment()
  require Counter >= 0
  ensure resulting Counter = Counter + 1

  Counter := ((((Counter + 1) * 2) - 2) / 2) + 1

Main()
  step Increment()
  step WriteLine(Counter)
  step Increment()
  step WriteLine(Counter)
```

Compatibility Note

The behavior of the resulting expression may differ from this description in the current AsmL 2 implementation.

The current AsmL2 implementation does not take into consideration all of the updates. The resulting expression queries the statement block's contribution to the current update set of the expressions with respect to a given location (see 7.5.1). In other words, it yields the value of a location after any updates created within the block will have been applied.

Since the order of expression evaluation is not given, the values returned by "resulting expressions" in AsmL 2 cannot be predicted in every case without introducing substeps at a lower level of abstraction than given in the model. (The value will be predictable in cases where a "total" update has occurred.)

7.4 Nondeterministic choice statements

```
choice      ::= choose [ unique ] binders stm
                [ ifnone stm ]
```

Choose-expressions using the keyword `choose` bind names to values using nondeterministic choice.

A statement-level choose-expression begins with the keyword `choose` and includes a statement block. In this form, all of the bindings established by the *binders* clause will be available for reference within the statement block.

A statement-level choose-expression may optionally provide an `ifnone` clause. If the choose-expression provides no bindings (for instance, when choosing from the empty set), the `ifnone` statement block will be evaluated. In that case, the value of the choose-expression is the return value of the `ifnone` statement block. Otherwise, the return value is that of the statement block following the *binders* clause.

If no `ifnone` clause is provided for a statement-level choose-expression then it defaults to "`ifnone skip`".

Example 47 Statement-level nondetermism

```
Main()
  S = {"a", "b", "c"}
  choose i in S
    WriteLine(i + " was chosen.")
```

The keyword `unique` may be added as a constraint to indicate that the selection is deterministic. An error will occur if the `unique` keyword has been used and there is more than one possible value to be selected.

7.5 Return statements

```
return      ::= return exp
```

A return statement is used as the last statement of a block to indicate the return value of that block.

AsmL does not issue an error if the return value of a statement block (or method) is ignored in the calling context.

Note to users

Unlike many other languages, AsmL uses "return" to indicate the value returned from a statement block, not from a method. The `return` *statement has no effect on control flow.*

7.6 Conditional expressions

```
branchExp   ::= ifExpr | matchExpr
ifExpr      ::= if exp [ then ] stm
                { elseif exp [ then ] stm }
                [ else stm ]
matchExp    ::= match exp case [ otherwise stm ]
case        ::= pat [ where exp ] ":" stm
```

All *conditional expressions* in AsmL that return values are in the form `if` *exp* `then` *expr1* `else` *expr2*.

The expression that follows "`if`" must be of type `Boolean` and is called the conditional *guard*. The value of a conditional expression is the value of *expr1* if the guard evaluates to `true`; otherwise it is *expr2*. Only one of *expr1* and *expr2* will be evaluated. If no `else` clause is provided, then the default is "`else skip`".

Conditional expressions may be in the form of an if-then-else expression, a match expression or a logical operation.

Note to users

The intent of guard expressions is to control which of the branches of the conditional expression will be taken.

It is generally a poor modeling approach to allow guards to update variables. Future versions of AsmL may generate a runtime error if the evaluation of a guard results attempts to alter state by updating variables.

7.6.1 If-then-else expressions

If-then-else expressions with `elseif` clauses are normalized as follows:

```
if g1 then e1 elseif g2 then e2 else e3
```

is interpreted as

```
if g1 then e1 else (if g2 then e2 else e3)
```

A value-level if-then-else-expression must always provide an `else` expression. (A value-level expression returns a value. This is in contrast to a statement-level if that return a value.)

Note that `elseif` and `else if` are distinct in terms of the layout rules for block structure given in section 3.1 above.

The keyword `then` is optional.

7.6.2 Match expressions

The simplest *match expression* is the single-case form `match` *exp pattern* : *stm*

Expressions in this form attempt to pattern-match `pattern` (in the manner described in section 4.6 above) with the value given by evaluating *exp* in the current context. If the match succeeds, then the bindings given by the pattern are established in a new scope and the statement block given immediately after the matched pattern is evaluated. An error occurs if the *exp* does not match *pattern*, unless an `otherwise` alternative is given.

Match-expressions with more than one case can be interpreted by nesting.

```
match v
    pattern1:  stm
    pattern2:  stm
```

Match can be interpreted as the following:

```
if pattern1 matches v then
   pattern1 = v
   stm
else (if pattern2 matches v then
      pattern2 = v
      stm
    else
      throw NoMatchException)
```

See section 4.6 above for examples of matching.

7.6.3 Defaults for conditionals

AsmL consistently uses "`skip`" as the default for statement-level conditionals and "`error`" as the default for conditionals that return a value.

For example "`ifnone skip`" is the default for statement-level choose and "`ifnone error`" is the default for expression-level choose. (Expression-level "choose" occurs when the statement block includes a `return` statement.)

For example,

```
let x = choose s in {}
             add s to ChosenValues
             return s
```

would produce a runtime error, since there is no value to return. In contrast,

```
choose s in {}
   DoSomething()
```

would just skip (i.e., do nothing without causing an error).

All other conditional forms make the same distinction between statement-level and expression-level defaults:

`if ... then`	"`else`" is optional; assume "`else skip`" in statement contexts, "`else error`" in expression contexts where a return value is expected.
`match ...`	If no matching case found, assume "`otherwise skip`" in statement contexts, but assume "`otherwise error`" for expression-level match.

It is also possible to write an expression-level if that does not have an else clause: `let x = if a > 0 then 4`. In this example, a runtime error occurs if `a` is not greater than `4`.

7.7 Try/catch expressions

```
exceptExp  ::=  try stm catch case
             |  throw exp
             |  error exp
```

AsmL supports exception handling with try/catch expressions.

The value of a try/catch expression is value of the statement block given in the try clause unless an exception occurs.

An exception can be generated explicitly using an expression of the form `throw` *exp*, where *exp* evaluates to a reference of an object that is derived from `System.Exception` class, or it may arise from a runtime event such as a divide-by-zero error.

If an exception occurs during the evaluation of the try block, then exception handling is invoked as follows.

First, all updates that were collected inside the try block are discarded.

The creation of new instances of classes during the evaluation of the try block is not reversed when an exception is thrown. This allows, for example, a newly created instance to be used as the exception (that is, as the value of a `throw` expression). The value of each field of the new instances will be the initial value given by the instance's constructor.

Next, the exception (raised by a `throw` expression or generated by the runtime environment) is matched against the cases given in the `catch` clause. The form of the exception cases is identical to the cases of a `match` expression. Pattern matching (identical to that of `match`) is used to determine which error case applies.

If an error case is matched, then the value of the statement block of that case is the value of the try/catch expression. (Updates introduced by the matched exception handler become part of the current step.)

If no exception handling case matches the value thrown, then the exception is thrown in the runtime context that contains the try block. The process proceeds recursively, and the program halts if no handler can be matched in the outermost context.

If more than one exception is thrown within the current step of the current abstract state machine, only one (chosen nondeterministically) will be matched against cases given in the `catch` clause.

The expression `error` *exp* can be used to express an unrecoverable error. The expression can be any expression, for example a string. (You do not need to define an exception data type to signal an error.) "Error" may be used in any statement context. Like "`return`," the keyword "`error`" is not followed by parentheses.

Errors may not be processed by any exception handler. The program will halt when an error occurs.

```
function F(x as Integer) as Result
  Y := y + 1
  if P(x, y) then
    return ok
  else
    error “F’s condition violated”
```

7.8 Quantifying expressions

quantifierExp `::= forall` *binders* `holds` *exp*
`| exists [ unique ]` *binders*

Quantifying expressions return true or false depending on whether a condition (given by *exp*) has been met *universally* over some collection of bindings or *existentially* by at least one example.

The *universal quantifier* consists the keyword `forall` followed by one or more binders for which a given condition must hold (given by the `holds` clause) if the quantifier is `true`. If the binders produce no bindings (for instance, if they iterate over an empty set), then the expression given by the holds clause is not evaluated, but the value of the `forall` expression is `true`.

The bindings produced by the `forall` expression may be referenced within the expression given by the `holds` clause.

The *existential quantifier* consists the keyword `exists` followed by one or more binders. If all of the names given in the binders may be bound to values, then the existential quantifier is `true`. If the binders produce no bindings (for instance, if they iterate over an empty set), then the value of the `exists` expression is `false`.

Example 48 Quantifying expressions

```
S = {1, 2, 3, 4, 5, 6}

odd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  v1 = forall i in S holds odd(i)                      // false
  v2 = exists i in S where i > 4                       // true
  v3 = forall i in S where i > 4 holds odd(i)          // false
  v4 = forall i in S where i > 100 holds odd(i)        // true
  v5 = forall i in S holds exists j in S where i < j   // false
  v6 = exists i in S where exists j in S where i < j   // true
  v7 = exists i in S, j in S where i < j               // true
  v8 = exists i in S, j in S where i + 1 = j           // true
  v9 = forall i in S, j in S holds i mod j < 6         // true
  WriteLine([v1, v2, v3, v4, v5, v6, v7, v8, v9])
```

7.9 Selection expressions

selectExp `::=` *selector comprehension* `[ifnone` *exp*`]`
selector `::= any | the | min | max | sum`

A *selection expression* is used to query for values from a domain given by a comprehension clause. (Recall from 4.5 above that comprehensions are in the form *exp* | *binder1*, *binder2*, ...)

The value of a selection expression depends upon the selector keyword.

- For "`any`," the value of *exp* for any one of the bindings produced by the *binders* clause. The selection is nondeterministic.
- The keyword "`the`" adds a constraint: there must be exactly one possible binding, or an error occurs.
- The keywords "`min`" and "`max`" are used to select the smallest and largest values possible. (The operations ">" and "<" must be defined for the data type in question.)
- The keyword "sum" causes the value returned to be the arithmetic sum of all values given by *exp*. (The operator "+" must be defined for the data type in question.)

An error occurs if the binders produce no bindings, unless the optional `ifnone` clause is provided. In this case, the value given after `ifnone` provides the default value of the selection expression.

Example 49 Value-level nondeterministic choice

```
Main()
  let S = {1, 2, 3, 4, 5}
  step
    let y = any i | i in S where i < 4
    WriteLine(y)                        // prints 1, 2, or 3
  step
    let z = the i | i in S where i < 2
    WriteLine(z)                        // prints 1
  step
    let w = sum i + 1 | i in S
    WriteLine(w)                        // prints 20
```

Example 49 includes two local fields whose values come from choose expressions. The first can be read as "let `y` equal any `i` such that `i` is an element of `S` where `i` is less than `4`." The second reads as "let `z` equal the (unique) `i` such that `i` is an element of `S` and `i < 2`"

The selectors `min`, `max` and `sum` are deterministic and return the minimum element, maximum element and the sum of all elements described by the comprehension.

Example 50 Selection expressions

```
const S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
const T = {-1, 2, 3, 5, 7}

IsOdd(x as Integer) as Boolean
  return (x mod 2 = 1)

Main()
  let v1 = (any x | x in T where IsOdd(x) and x > 0)
  let v2 = (the val | val in T where val notin S)
  let v3 = (max x + y | x in S, y in T)
  let v4 = (min x | x in S + T)

  // v1 is one of {3, 5, 7}
  // v2 is -1
  // v3 is 17
  // v4 is -1
```

Although parsing without parentheses works, it is considered to be good style to put parentheses around everyselection expression.

7.10 Primary Expressions

```
binaryExp   ::= primaryExp { binaryOp primaryExp }
primaryExp  ::= unaryOp applyExp
              | applyExp [ ( is | as ) typeExp ]
              | resulting exp

unaryOp     ::= not | "-"
binaryOp    ::= implies | and [ then ] | or [ else ]
              | "*" | "/" | mod | "+" | "-"
              | union | intersect | merge
              | subset | subseteq | in | notin
              | "=" | "<>" | "<" | ">" | "<=" | ">="
              | eq  | ne  | lt  | gt  | lte  | gte
```

Primary expressions consist of logical operations, arithmetic operations, and the invocation of methods.

The meaning of the logical operators is given above in 7.7.

The arithmetic and relational expressions are defined in the AsmL library. They appear in this reference only by virtue of their special syntactic form.

To be written: Give a precedence table.

7.10.1 Logical operations

The logical operations `and` and `or` are commutative in AsmL. There is no implied order of evaluation of the operands.

Alternate forms are provided for the case of "sequential and" and "sequential or" where the order of evaluation is significant. The meaning of the logical operators `and then`, `or else` and `implies` are given by the following table.

`E1 and then e2`	`if e1 then e2 else false`
`E1 or else e2`	`if e1 then true else e2`
`E1 implies e2`	`(not e1) or e2`

7.10.2 Type query expressions

Type queries are `Boolean` expressions that return `true` if a value is of a given type. Type queries are in the form *applyExpr* `is` *type*. See section 4 above for more about types.

7.10.3 Type coercion expressions

AsmL allows the user to convert types using expressions in the form *addExpr* `as` *type*. The type coercion operator invokes the converter method that applies to the type being converted.

Conversions among built-in types are provided in the runtime library. See the accompanying document: “AsmL Standard Library Reference”.

Example 51 Type conversions of built-in types

```
Main()
  //conversions using convertors
  step WriteLine(1b as Short)        // prints 1
  step WriteLine(1 as Double)        // prints 1.0

  // conversions using functions
  step WriteLine(ToInteger(1.9))     // prints 1
  step WriteLine(ToChar("a"))        // prints 'a'
  step WriteLine(ToSet([1,2,1]))     // prints {1, 2}
```

7.11 Apply expressions

```
applyExp   ::= atomicExp { argList }
             |  mybase arglist { argList }
argList    ::= "(" [ exps ] ")" | "." id [ typeArgs ] }
```

Apply expressions are used for global method application, instance-level method application, map application, field access and constructor invocation. This form also appears in the update statement given in 8.1 below and in the resulting expression given in 7.3.

Global method application is in the form *id* (*arg1*, *arg2*, …). Note that method names do not denote values in AsmL. Thus, a "method" is never the value of an expression.

Instance-level method application is in the form *atomicExp* . *id* (*arg1*, *arg2*, …) where the value of *atomicExp* is an instance of a class or a compound value of a structure. Section 6.2.6 describes how a method is selected for application based on the types of its arguments.

Map application is in the form *exp* (*arg1*, *arg2*, …) where the value of *exp* is a map (that is, a value of type `Map`). The value of the expression is an element of the map's range. If a tuple matching (*arg1*, *arg2*, …) is not in the map's domain, an exception is thrown. Otherwise, the result is the matching range value.

Field access is in the form *exp*.*id* where *exp* is a value of a datatype that includes *id* as a field. Note that *id* is equivalent to `me`.*id* within a type declaration for fields defined within the type (or any of its supertypes).

AsmL allows additional flexibility in how methods are applied to arguments. Two syntactic forms may be used: either `x.f(a, b)` or `f(x, a, b)`. These forms have equivalent meaning.

```
class C
  f(x as A)

Main()
  c.f(x)
  f(c, x)         // means the same as c.f(x)
```

The form `mybase`(*arg1*, *arg2*, ...) is used within a method to invoke the corresponding method of a direct supertype. (The method must have been specialized using the `override` keyword.)

Example 52 Invocation syntax

```
f()
  WriteLine("Global method f() was invoked.")

class Foo
  i = "Field i was accessed."
  g()
    WriteLine("Instance-level method g() was invoked.")

h = {1 -> "Map h was applied with (1) as argument",
     2 -> "Map h was applied with (2) as argument"}

Main()
  c = new Foo()
  step f()                // global method invoked
  step c.g()              // instance method invoked
  step WriteLine(h(1))    // map application
  step WriteLine(c.i)     // field access
```

7.12 Atomic expression

```
atomicExp  ::= constructor | me | value
             |  "(" exp ")"
             |  id [ typeArgs ]
```

Atomic expressions denote a value in the form of a constructor, a named value expression or the keyword `me`.

Constructors of values are given in 4.3.

A *named value expression* consists of an identifier. It denotes the value of a field instance (either a constant or a variable) whose name is the same as the given identifier. For variables, the value returned is always with respect to the current step of the an abstract state machine)

The name may be local, instance-based or global The interpretation of the name follows AsmL's priority of name visibility: local first, instance-level second and global third.

The keyword `me` may be used as an expression within a class declaration's field initialization expressions and instance-level methods to denote the current instance of the class in the invocation context. It may also be used in the where-clause of a constrained type if the underlying base type is a reference type.

The keyword `me` may be used as an expression within a structure declaration's value-level methods to denote compound value in the invocation context. The keyword `me` may not be used in a structure's declaration 1) as part of any field initialization expression or 2) on the left hand side of an update statement.

The keyword `value` may be used as an expression within the setter of a property, the adder or remover of an event, or within the where-clause of a constrained type if the underlying base type is a value type.

7.13 Enumerated types

In AsmL, "`enum of x`" where `x` is a type expression may be used to mean the set of all values of a given type. The keyword `enum` is short for "enumeration," so "`enum of T`" means an "enumeration of all values of type `T`."

Example 53 Enumerated types

```
Main()
  step foreach val in enum of Boolean
    WriteLine(val)        // prints true, false or false, true
```

If "`enum of T`" is used in a context where a set of values is expected, the type must be *computationally enumerable*. (Otherwise you may not query for its values.) The following built-in types are computationally enumerable: `Boolean`, `Char` and `Null`. All other built-in types are not enumerable.

Whether type T is enumerable or not, the expression `x is T` is available to test whether `x` is a value of type T.

There is no way to test whether a type is enumerable.

A disjunctive type `T or S` (see section 5.1.1 above) is computationally enumerable if both type `T` and type `S` are enumerable types. An option type `T?` (see section 5.1.2 above) is computationally enumerable if type `T` is an enumerable type.

Example 54 Enumerable disjunctive types

```
Main()
    step foreach val in enum of Boolean or Null
    WriteLine(val)        // prints true, false and null
```

A user-defined structure or product type is computationally enumerable if all of its fields are enumerable.

Example 55 Enumerable structure type

```
structure Flag
   f1 as Boolean
   f2 as Boolean

Main()
  step foreach f in enum of Flag
    WriteLine(f)    // Flag(true, true), Flag(true, false),
                   // Flag(talse, frue) and Flag(false, false)
```

User-defined classes may optionally be declared as enumerable. The keyword "`enumerated`" may precede a class declaration to indicate all instances of the class should be tracked. (Note that instances of enumerated classes may be reclaimed by the garbage collector over time.)

Example 56 Enumerated class

```
enumerated class A

Main()
  step
    let a1 = new A()
    let a2 = new A()
  step foreach x in enum of A
    WriteLine(x)             // prints two values
```

Note that a step is required in this example. If there were no "step" separating the invocation of "new" and the "forall" statement, then there would be no values for the iteration, since the update to "A" takes effect as of the next step.

User-defined enums are enumerable.

Example 57 Enumeration of enum values

```
enum Color
  Red
  Green
  Blue

Main()
  WriteLine(Size(enum of Color))        // prints 3
```

A constrained type defined by "*type* `where` *expr*" (see section 5.5.6 above) is computationally enumerable if the type given is enumerable. In addition, a constrained type is enumerable (regardless of whether the type given before the `where` keyword is enumerable) if the expression following the `where` keyword is in the form "`value in` *expr2*".

Example 58 Enumeration of constrained type

```
type MyType = Integer where value in {1, 2, 3}

Main()
  WriteLine(Size(enum of MyType))       // prints 3
```

Compatibility note

The functionality described in this section is not fully implemented in the current version of AsmL (but will be in a future release). The current implementation differs from the description given above in that 1) all structures and conjunctive types are not enumerable and 2) all option types and disjunctive types are not enumerable. Also, the current implementation accepts only a type name instead of the more general type expression in an "enum of" expression.

7.14 The do expression

The form `do` *statement-list* allows a statement block to be placed in a context that would otherwise expect an expression. The value of the `do` expression is given by the `return` statement in the block.

Note to users

The `do` expression is not normally needed in modeling. It is provided for orthogonality. For example, "do" might be used by a code-generation tool or compiler for inlining.

---

8 State Operations

```
operationalStm  ::=    update
                 |  parallelUpdate
                 |  sequence
                 |  skip
```

This section describes the part of AsmL that deals with runtime state.

AsmL uses the semantics of abstract state machines as the framework for the dynamic aspects of the program. The practical effect of this is that AsmL has runtime contexts called *states* with fixed associations of variable names to values. The change from one state to another occurs as an atomic transaction called a step. Within a step, any number of changes to variables may be proposed (by means of the update statement described below), but the changes have only effect for subsequent states. Within a given state, variables always have the same, fixed values.

New runtime contexts may be established in four ways:

- A sequential block (in the form `step` … `step` …), also called a *machine*, denotes a series of runtime states. It is possible that one or more of the steps may be iterated. The accumulated changes from all the steps will be proposed as updates in the current runtime context. This is described below in section 8.3.
- A process is a distinct runtime context associated with the invocation of a method. It differs from a machine in that accumulated changes from its run are not integrated as updates into the runtime context that spawned it.
- An agent is a separate area of memory (that is, a distinct collection of field instances) with associated operations that occur on demand as transactions.
- An exploration expression creates a tree of runtime states by exploring all nondeterministic execution paths for a given expression. The result of exploration is a collection of values taken from each possible state. Like processes, the accumulated changes to state from subprocesses are discarded. This is described below in section 8.7.

8.1 Update statements

```
update     ::= applyExp ( ":=" | "*=" | "+=" ) exp
            |  add exp to applyExp
            |  remove exp [ from applyExp ]
```

Update statements determine a new value for the variable given by *applyExpr* in the following step of the abstract state machine associated with the current invocation context. There are three kinds of update statements: The update operator “:=” replaces the old of a variable with a new one in the next step. The `add … to` operation adds an element to a set. The `remove … from` operation removes an element from a set or map.

The form `x := exp` is equivalent to `x := x + exp`.

The form `x *= exp` is equivalent to `x := x * exp`.

Note that an update statement has no effect on the value associated with the given variable in the current step. Instead, the variable will be associated with the proposed value as of the subsequent sequential step of the current abstract state machine.

Example 59 Update statement

```
var i = 3

Main()
  step while i > 0
    i := i – 1              // updates i for next step
    WriteLine(i)            // prints 3, prints 2, prints 1
```

Example 59 contains an update statement, `i := i – 1`, that causes the decremented value of variable `i` to become the value of `i` in the next step of the abstract state machine introduced by the `step` expression. Note that the `WriteLine` expression will write the current value of field `i` in each step, not the proposed value, even though the update statement occurs in the source before the `WriteLine` statement.

Update statements do not return a value.

Example 60 Update statements

```
f1 = 100                        // global constant
var f2 = “abc”                  // global variable
var f3 = {1, 2, 3}              // global var w/ compound value

class Foo
  var f4 = “abc”                // instance variable
  shared var f5 = “efg”         // global variable

Main()
  c = new Foo()                 // local constant
  var f6 = “abc”                // local variable
  step
    f1 := 200                   // error! “f1” is constant
    f2 := “def”                 // OK, update global variable
    remove 2 from f3            // OK, update indexer
    f4 := “efg”                 // error! “f4” is out of scope
    c.f4 := “efg”               // OK, update instance variable
    f5 := “hij”                 // error! “f5” is not visible
                                // in the current scope
    c.f5 := “hij”               // OK, update global variable
                                // that is in the scope of c
    c := new Foo()              // error! “c” is constant
    f6 := “def”                 // OK, update local variable
```

8.1.1 Consistency of update statements

All update statements invoked with respect to a step of an abstract state machine must be consistent, or the error `InconsistentUpdate` will be thrown.

Consistent in this context means that no contradiction could arise as a result of the update. For example, if `S` is a set-valued variable, then any update that adds elements to `S` would be considered to be consistent, since each addition could be considered to be independent of any other addition. In contrast, if `x` is an Integer-valued variable, then updating `x` to the value 3 and the value 4 in the same step would be produce a contradiction.

8.1.2 Locations

The left-hand side of an update statement identifies the variable (a specific field instance) whose value will become the proposed value (given by the right-hand side of the update statement) in the subsequent step.

The syntactic form used on the left-hand side of an update statement is called a *location*. It consists of a variable followed by optional indexers, as described in sections 6.1.6 and 6.1.7.

Identifying which of a location's terms constitute the variable being updated and which are indexers is not evident from the syntax. However, the distinction between variables and indexing fields can be determined from the field declaration's form and whether the field declaration was nested in a class or structure declaration.

Note to users

Most users of AsmL may safely ignore the distinction between variables and indexers, since it only becomes important in determining whether an update-related inconsistency has occurred in the relatively infrequent case of nested structures. An example of a nested structure is a `Map` *that contains other* `Maps` *as elements in its range.*

Implementers and others who are interested in these details should read on. Other readers should skip to section 8.1.3 below.

The following algorithm can be used to analyze a location and identify the variable (i.e., field instance) and any indexers that may follow it (after allowing for the possible presence of a namespace qualifier as described in 9.3).

Initialize an empty sequence that will contain the indexers. As described in "Fields" above, each indexer will either be an indexing field name or a tuple of indexing parameters.

Do the appropriate case from among the following, iterating until a variable has been found:

- If the location only has one term, interpret this name as a local variable, instance-level variable, global variable within the current scope and stop.
- If the location is in the form `N.M` where `M` is an identifier, then evaluate `N` as an expression in the current scope. If the result is a compound value (that is, of type `structure`), then push `M` onto the front of the indexer list. Then, take `N` as the location and iterate. However, if the result of evaluating `N` is an instance of a class, then interpret `M` as the name of an instance-level field associated with `N`'s value and stop.
- If the location is in the form `N(…)` where `(…)` is a tuple expression, then evaluate `N` as an expression in the current scope. If the result is a compound value (in particular, of type `Map`, `Set` or `Seq`), then evaluate the tuple expression in the current scope and push it onto the front of the indexer list. Then, take `N` as the location and iterate. However, if the result of evaluating `N` in the current scope is not a compound value (for example, an instance of a class), an error occurs.

The result of this process will be a variable and a sequence of indexers.

8.1.3 Partial and total updates

Another way to understand the behavior of updates is in terms of partial and total updates.

When an update statement directly sets the value of a variable (without the use of indexers), then a total update has occurred. When an update statement uses indexers, then a partial update has occurred.

As mentioned above in section 8.1.1, all updates (including partial updates) must be consistent.

8.2 Parallel update blocks

parallelUpdate ::= `forall` *binders* <u>*stm*</u>

Multiple updates may be added to the current step using a forall statement in the form `forall` *binder1*, *binder2*, … *stm*.

The statement list *stm* is evaluated for each binding generated by the binders (see section 4.7). The updates that result from each evaluation of the statement block are added to the update set of the current step.

No value is returned from a forall statement.

Example 61 Parallel update

```
var MySet as Set of Integer = {}

const MyIntegers = {1, 2, 3, 4, 5}

Main()
  step
    forall i in MyIntegers
      require Size(MySet) = 0
      add (i + 1) to MySet            // add each i + 1 to set
  step
    WriteLine(Size(MySet))            // prints 5
```

Example 61 illustrates the parallel nature of a forall statement. The assertion `require Size(MySet) = 0` checks that there are no elements in the set-valued variable `MySet` in each iteration. The constraint is satisfied because all of the parallel updates are deferred until the subsequent sequential step.

Thus, although iteration is present, the run consists of just two state transitions. In the initial state, the value of `MySet` is the empty set, `{}`. In the second state, `MySet` is `{2, 3, 4, 5, 6}`.

8.3 Sequential blocks

```
sequence   ::=  step
step       ::=  step  [ label  ] [ iterator ] stm
iterator   ::=  foreach  binders
             |   for id "=" exp to exp
             |   while exp
             |   until ( exp | fixpoint )
```

Sequential blocks cause a new abstract state machine to run.

The run of the machine is given as a sequence of discrete *steps*.

Each step is performed in the lexical order it appears. If an *iteration clause* is given, the step repeats until a stopping condition is met.

When the sequential block has completed all of its steps, its cumulative update set is added to the update set of the current step (that is, the context in which the sequential block was invoked). In other words, it is as if all of the updates to variables produced by the sequential block are collapsed into a single block of proposed (possibly partial) updates in the enclosing scope.

The *cumulative update set* is an aggregation of all update sets of the sequential machine, with updates in later steps overriding the updates of previous steps for any locations that are updated more than once during the run of the sequential block. Partial updates are treated consistently.

Note to users

Readers who are interested in the precise semantics of partial updates should refer to the Microsoft Research website.

8.3.1 Effect of recursion on sequential steps

If a sequential block is invoked recursively (that is, as part of recursive method invocation), then a new abstract state machine is created for each level of recursion.

8.3.2 Scope of constants and variables

Any local field declarations found in steps of the sequential block are visible in all of succeeding steps. Each succeeding step clause establishes a new scope nested within the scope of the (lexically) previous step.

8.3.3 Iterated steps

Steps of a sequential block may be iterated if they are introduced by `foreach`, `while` or `until`.

The iterated steps proceed sequentially until their stopping condition has been met.

In the case of `until fixpoint`, the stopping condition is met if no non-trivial updates have been made in the step. Updates that occur to variables declared in abstract state machines that are nested inside the fixed-point loop are not considered. An update is considered non-trivial if the new value is different from the old value.

Each iterative step forms a distinct step of the abstract state machine introduced by evaluating the sequential block.

Example 62 Sequential and parallel steps

```
reachable of T (root as T, arcs as Set of (T, T)) as Set of T
 var reachable = {root}
 step until fixpoint            // sequential step
   forall (l, r) in arcs        // parallel update
     if l in reachable and r notin reachable then
       add r to reachable
 step
   return reachable
```

```
Main()
  arcs = {(1, 2), (2, 3), (4, 5), (3, 1), (10, 9)}
  WriteLine(reachable(3, arcs))        // prints {1, 2, 3}
```

Example 62 gives an algorithm that calculates the reachable nodes of a directed, possibly cyclic, graph. The local variable `reachable` is a set of nodes that have been seen so far. The algorithm includes sequential aspects (iterating after each update of the nodes on the frontier) and concurrent aspects (visiting newly visible nodes).

The `Main()` method does not include steps. From its point of view, the program is entirely functional. It sees only the cumulative effect of the sequential steps that occurred in the subprogram that calculated the reachable nodes.

8.4 The skip statement

The skip statement (with syntax `skip`) is a null statement that performs no update and returns no value.

Example 63 Skip statement

```
Main()
  var a = 0
  step
    if 2 > 1 then
      a := 2
    else
      skip
  step
    WriteLine(a)                 // prints 2
```

8.5 Processes

[TBD]

8.6 Agents

[TBD]

8.7 Exploration expressions

```
exploration ::= explore exp
             |  search exp
```

The explore statement takes an expression which must return a value. The expression is evaluated as often as different choices (or combinations of choices) are possible during the execution of that expression. (In particular, if the expression is deterministic, then the expression is evaluation exactly once.) The result of the explore statement is a sequence containing one result value for each possible combination of choices.

The search expression takes an expression, may or may not return a value. Like explore, the search expression tries different possible choices. But unlike explore, not all possibilities are explored. Search stops the search as soon as the expressions succeeded once.

Example 64 Select expressions

```
Choose() as (Integer,Integer)
  x = any i | i in {1..3}
  y = any i | i in {2..x} // note that for x=1, no possible
                          // solution exists; thus x=1 will be
                          // eliminated from the search by
                          // “explore” and “search”.
  return (x,y)

Main()
  WriteLine(explore Choose())
  // prints a sequence containing the following pairs
  // (2,2), (3,2), (3,3)
  // in any order.

  WriteLine(search Choose())
  // this prints exactly one pair.
```

9 Namespaces

AsmL provides a module system that allows names (see section 3.3 above) to be reused without conflict in different parts of the program. Each of these name-distinct modules is given by a *namespace declaration*.

Note that the only effect of namespace declarations is the visibility of names (that is, whether simple names or qualified names must be used.)

9.1 Unit of compilation (assembly)

An AsmL program is given syntactically as an assembly.

assembly ::= [*namespaceOrDecl*]
namespaceOrDecl ::= *namespace* | *declaration*

A *program* consists either of declarations, or of one or more namespaces which in turn contain declarations.

9.2 Namespaces

namespace ::= [*attributes*] `namespace` *name*
declaration ::= *import* | *type* | *member*

A *namespace declaration* introduces a new scope (see section 3.5 above) for the names introduced by the declarations nested with it.

A namespace declaration consists of an optional namespace clause followed by directives and declarations. The *namespace clause* introduces a new scope (distinguished by a namespace identifier). Directives affect how identifiers used within a given namespace will be recognized. Declarations are described in section 3 above.

The *namespace identifier* may be a qualified name or a simple name.

If the program does not include a namespace clause, then its declarations are interpreted as having been preceded by “`namespace Application`”, and an error will occur if a namespace clause appears anywhere in the program. In other words, if the default namespace is used, then no user-provided namespace declarations are allowed.

The order of namespace declarations in the program does not matter. Namespaces are processed together without the need for forward declaration of elements referenced in the source before their definition.

Example 65 Namespaces

```
namespace Main

import MyProg

Main()
```

```
 DoTopLevel()

namespace MyProg

DoTopLevel()
  WriteLine("Hello, world!")
```

9.3 Qualified names

The qualified form of a name (see section 3.3 above) is visible within the scope of any namespace. The full name of the namespace is used as the identifier's prefix.

(Names declared within a namespace may be used in unqualified form within that namespace.)

Example 66 Use of qualified names

```
namespace MyProg.MySubprogram

DoTopLevel()
  WriteLine("Hello, world!")

namespace Main

Main()
  MyProg.MySubprogram.DoTopLevel()
```

9.4 Import directives

import ::= `import` *name* `[ "=" ` *name* `]`

An *import directive* introduces names declared outside of a namespace declaration for use as simple names.

The *external identifier* provided by an import directive may be a namespace identifier of a namespace declaration of the current program, or it may identify external module such as a library, whose definition is given by the external implementation environment.

Example 67 Import Directives

```
namespace Application
import System             // import directives
import System.IO
import SysIO = System.IO  // renaming
```

All of the names declared in the imported namespace become available as simple names within the namespace containing the directive. These are known as *imported names*.

The global namespace `Application` has no special behavior with respect to the visibility of names; it too must be imported if its names are to be used as simple names within the scope of another namespace.

The import directive is not transitive; that is, names made visible inside a namespace `N` by virtue of the import directive may not be used as simple names within a namespace that imports `N`.

The qualified forms of imported names are available within the namespace that contains the import directive.

It is possible for a namespace to include a nested declaration of the same simple name as one of the imported names. However, each time such a name is used, the meaning must be clarified by explicit qualification; neither can be used as a simple name. In like manner, if two imported names are the same, then their qualified forms must always be used.

Example 68 Explicit qualification required

```
namespace N1.S1
Foo()
  WriteLine("N1.S1.Foo")

namespace N2
Foo()
  WriteLine("N2.Foo")

namespace Application
import N1.S1
import N2

Foo()
  WriteLine("Main.Foo")

Main()
  step Application.Foo()        // qualified even in local
scope
  step N1.S1.Foo()
  step N2.Foo()
```

Example 68 illustrates the fact that qualified names must be used whenever imported names produce the possibility of ambiguity.

9.4.1 Units of compilation

The external identifiers used by the import directive may refer to namespaces that are not declared within the program but are provided by separate *units of compilation*, such as the built-in library.

The division of a program into separate *units of compilation* does not affect its meaning. Namespaces imported from separate units of compilation behave as if their declarations were provided as part of the program (except that external units of compilation may provide their own namespaces even if the program uses the default namespace).

Implementation note

A namespace declaration may not be split across multiple units of compilation. The program may not introduce new declarations into namespaces imported from the external environment. An error occurs if the program contains a namespace clause with the same name as an externally provided namespace.

The namespace `AsmL` contains AsmL's standard library of operations. `AsmL` is implicitly imported into every namespace.

9.5 Linkage

AsmL does not specify how it interacts with entities provided by the external environment.

The representation of values by the language implementation is abstract.

The mechanism by which AsmL invokes external, foreign routines that are introduced by import directives is not part of AsmL. In particular, if external routines must be invoked in a particular order, the steps of an abstract state machine must explicitly give this order. (By design, the order of evaluation of expressions within a step is not specified.)

Except for the convention of the name `Main` to denote the program's entry point, the way in which AsmL programs may be invoked by the outside environment is not part of AsmL.

For example, AsmL does not have the concept of a thread of execution, since all computation within a step of an abstract state machine proceeds in parallel. An AsmL implementation is free to interact with the external operating environment in any way that preserves the semantics of the language, for example, by using as many processes and threads as it desires. Different implementations might make very different choices in this area. Thus, the language definition does not specify the mechanism for synchronizing AsmL objects with those provided by an external runtime environment.

Nonetheless, Microsoft's implementation of AsmL for Windows provides extensive integration with .NET. (This integration provides for synchronization between AsmL objects and the external environment, but this is not part of AsmL.) As a result, AsmL models may be invoked from test harnesses written in any .NET-compliant language such as Visual Basic.NET and Visual C#. The .NET integration is described in separate documentation.

It may come as a surprise that an implementation of a language focused on rigorous semantic modeling devotes so much energy on integration with an external operating environment. Our experience so far is that sophisticated integration with external operating environments is an essential part of making rigorous approaches relevant to software specification and testing in the commercial environment. As an example of this point, the .NET integration provided by Microsoft's AsmL compiler has been used by test harnesses that check whether an implementation (written in a standard commercial programming language) agrees at runtime with its (mathematically precise) executable specification written in AsmL.

9.6 Literate programming environment

Another tenet of AsmL approach is smooth integration into existing software development processes. In practice, this important human consideration means that AsmL source will occur most frequently as "pseudo-code" inside of existing text-oriented documentation.

In Microsoft, virtually all specification documents used for internal development projects are encoded as binary files in Microsoft Word (".doc") format. Microsoft's AsmL toolset is capable of processing AsmL source directly from Word files (using a special AsmL "style" in the word processor).

The result of this processing step is a text file structured as XML markup that conforms to the "`AsmL.dtd`" schema. This schema allows AsmL source to be interleaved with marked-up text and links to graphics that document the design.

The benefit of XML mark-up is that it has a variety of processing options in the documentation work flow, for example, as the basis of code review templates, test plans, reference material for customer support personnel and even as part of the product's external documentation. Putting AsmL-based specifications at the center of it documentation process maximizes the benefit a development team will receive from its investment in precise, testable specifications.

10 .NET Extensions

This section lists features of AsmL that are specific to the .NET framework.

Note to users

These features should not be used for modeling, but only as a means of achieving interoperability. In some cases they provide a way of bypassing AsmL update semantics. This may be desired when integrating AsmL models into the external environment (for example, connecting a model to a graphical user interface), but it makes the models less analyzable for the purposes of testing and establishing program semantics.

10.1 Modifiers

```
typeModifier        ::= extensibility | access

access              ::= public | private | protected | internal
extensibility       ::= abstract | sealed

extendedMemberModifier ::= extensibility | access | primitive

paramModifier       ::= primitive ref | primitive out
                      |  out | inout

localVariableModifier ::= primitive
```

Modifiers may be added to type declarations, members, parameters of methods and local variables.

The modifiers `virtual` and `override` are used to provide methods that may be specialized by subtypes. The keyword `virtual` indicates that a default implementation is provided; however, a subtype may override this default. The keyword `override` precedes a method that replaces the default given in s supertype. (The corresponding supertype method must be `virtual` or `abstract`.)

Override must be used whenever a method replacement occurs. If neither "`virtual`" nor "`abstract`" is specified in a method's declaration in the base type, then this method may not be specialized in a derived type.

The extensibility modifiers (`abstract` and `sealed`) may be added to a type or member declaration to indicate whether additional definitions may (or in the case of `abstract`, must) be provided by subtypes. A `sealed` method (or any method of a sealed datatype) may not be extended.

The modifier `primitive` may be applied to methods, method parameters and local variables. If provided, it indicates that AsmL update semantics do not apply. Instead, updates to primitive variables take effect with each update statement.

AsmL's parameter modifiers allow for call-by-reference and output parameters.

The modifiers for access (`public`, `private`, `protected` and `internal`) have the same meaning as other CLS-compliant languages. The modifier may limit the accessibility of a type's members. If unspecified, the type's visibility is `internal`.

A member is *accessible* if it may be referred to by using a simple name, a qualified name or the dot (".") operator. Thus, if a member is not accessible in a given context, then there is no way to refer to it.

The members of types declared as `public` are accessible in every scope.

The members of types declared as `private` are accessible only within the lexical scope of the type's declaration.

The members of types declared as `protected` are accessible only within the scope that contains the type declaration.

The members of types declared as `internal` are accessible only within the current compilation unit.

Private members are only visible in the current scope. They are present but not visible in subtypes.

10.2 Attributes

```
attributes     ::= { attribute }
attribute      ::= "[" [ target ] attributeConstructor
                      { "," attributeConstructor } "]"
target         ::= id ":"
attributeConstructor ::=  id | id "(" attributeExps ")"
attributeExps::= [ exps ] [ namedAttrArgs ]
namedAttrArgs    ::= [ namedAttributeArg { "," namedAttrArg } ]
namedAttrArg     ::=       id "=" exp
```

Attributes in AsmL are implemented using the conventions of the Common Language Specification (CLS). Refer to CLS documentation for their use.

10.3 Delegates

```
delegate    ::= delegate id [ typeParams ] signature
```

Delegates in AsmL are implemented using the conventions of the Common Language Specification (CLS). Refer to CLS documentation for more information.

Example 69 Delegate

```
delegate IntFunc(i as Integer) as Integer
```

```
square(i as Integer) as Integer
  return i * i

structure Incrementer
  by as Integer
  Action(i as Integer) as Integer
    return i + by

Main()
  a = new IntFunc(square)
  b = new IntFunc(Incrementer(21).Action)
  WriteLine(a(4))                // prints 16
  WriteLine(b(21))               // prints 42
```

10.4 Properties

property ::= `property` (*name* | `me`) [*params*] `as` *typeExp*
(*setter* [*getter*] | *getter* [*setter*])
setter ::= `set` [*stm*]
getter ::= `get` [*stm*]

Properties in AsmL are implemented using the conventions of the Common Language Specification (CLS). Refer to CLS documentation for their use.

10.5 Events

event ::= `event` *name* `as` *typeExp*
(*adder* [*remover*] | *remover* [*adder*])
adder ::= `add` [*stm*]
remover ::= `remove` [*stm*]

Events in AsmL are implemented using the conventions of the Common Language Specification (CLS). Refer to CLS documentation for their use.

10.6 Type integration

The AsmL built-in types `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Char` and `String` are extensions of built-in CLS types. This means that any .NET Framework method with the appropriate parameter type may be invoked on an AsmL value of these types.

AsmL classes and structures are implemented as CLS classes. If an AsmL structure is prefixed by the keyword "`primitive`" then it is implemented as a CLS structure. (Note that AsmL structures are more general than CLS structures. In particular, an AsmL structure may be recursive.)

CLS classes may be made available in AsmL by means of the "`import`" declaration. Since the CLS type system does not distinguish null objects (as does AsmL's), all parameters of class type `T` will be mapped to AsmL type `T?` when imported.

10.7 Reflection

AsmL allows access to the "type" object provided by the CLR reflection interface. The syntax is "`type of T`" where T is any type expression. Operations on this value are defined by the .NET Framework.

11 Library

11.1 Set operations

The AsmL library provides the following operations on the built-in type family `Set`:

```
BigUnion(Set of Set of T) as Set of T
BigIntersect(Set of Set of T) as Set of T
ChooseSubset(Set of T) as Set of T
ChooseNonemptySubset(Set of T) as Set of T
Size(Set of T) as Integer
operator `+`  (Set of T, Set of T) as Set of T  // union
operator union (Set of T, Set of T) as Set of T // union
operator `*`  (Set of T, Set of T) as Set of T  // intersection
operator intersect (Set of T, Set of T) as Set of T
operator `-`  (Set of T, Set of T) as Set of T  // difference
operator `<`  (Set of T, Set of T) as Boolean  // proper subset
operator subset
operator `<=` (Set of T, Set of T) as Boolean  // subset or equal
operator subseteq
operator `>`  (Set of T, Set of T) as Boolean  // proper superset
operator `>=` (Set of T, Set of T) as Boolean  // superset or eql
operator in (T, Set of T) as Boolean           // membership tst
```

AsmL provides "`union`", "`intersect`" and "`subset`" for set union, intersection and subset (or equal) as well as the equivalent "+", "*", "<" and "<=" operations. The operator "-" is set difference.

11.2 Sequence operations

The AsmL library provides the following operations on the built-in type family `Seq`:

```
Head(Seq of T) as T                     // the first element
Tail(Seq of T) as Seq of T              // all but first
Last(Seq of T) as T                     // the last element
Front(Seq of T) as Seq of T             // all but last

Indices(Seq of T) as Set of Integer     // {0..Size(s)-1}
IndexRange(Seq of T) as Seq of Integer  // [0..Size(s)-1]
Values(Seq of T) as Set of T            // {i | i in s}
Reverse(Seq of T) as Seq of T           // in backward order

Length(Seq of T) as Integer             // number of entries
Size(Seq of T) as Integer               // synonym of Length()

Drop(Seq of T, Integer) as Seq of T   // all but first n elements
Take(Seq of T, Integer) as Seq of T     // first n elements
Subseq(Seq of T, Integer, Integer) as Seq of T

IndexOf(Seq of T, Seq of T) as Integer  // start of 1st subseq
LastIndexOf(Seq of T, Seq of T) as Integer// start of last subseq
```

```
Zip(Seq of A, Seq of B) as Seq of (A, B) // pairwise combination
Unzip(Seq of (A, B)) as (Seq of A, Seq of B) // pairwise split

operator in (T, Seq of T) as Boolean    // find element in seq
operator + (Seq of T, Seq of T) as Seq of T   // concatenate
```

The "-" operator for sequences is not provided.

11.3 Map operations

The AsmL library provides the following operations on the built-in type family `Seq`:

```
Indices(Map of T to S) as Set of T            // domain
Values(Map of T to S) as Set of S             // range
Size(Map of T to S) as Integer

operator union(Map of T to S, Map of T to S) as Map of T to S
operator + (Map of T to S, Map of T to S) as Map of T to S
operator in(T, Map of T to S) as Boolean      // checks domain
```

11.4 String operations

The AsmL2 library provides a `String` datatype that is compatible with the .NET Framework `System.String`. However, in addition, future versions of the compiler will support all of the sequence operations, as if `String` were a subtype of the AsmL type `Seq of Char`.

12 List of Examples

Example 1 In-place sorting 1
Example 2 Indentation as block structure 14
Example 3 Indentation as block structure 14
Example 4 Declarations 15
Example 5 Continuation of declarations 18
Example 6 Literal constructors 20
Example 7 Constructing instances 20
Example 8 Constructing compound values 21
Example 9 Constructing enumerated values 21
Example 10 Constructing tuples 22
Example 11 Constructing sets 22
Example 12 Constructing sequences 23
Example 13 Constructing maps 23
Example 14 Symmetry of construction and pattern matching 24
Example 15 Pattern matching without binding 25
Example 16 Single-name patterns 25
Example 17 Type patterns 26
Example 18 Tuple pattern 26
Example 19 Destructing patterns for structures 27
Example 20 Maplet patterns 28
Example 21 Simple, iterated and nondeterministic bindings 29
Example 22 Type expressions 31
Example 23 Disjunctive type 32
Example 24 Type families 33
Example 25 Type operations 34
Example 26 Interface declaration 38
Example 27 Datatype variants 39
Example 28 Structure case as subtypes 40
Example 29 Enumeration with user-provided values 41
Example 30 Enum Ranges 41
Example 31 Constrained type 42
Example 32 Constraints on type parameters 44
Example 33 Field containing a compound value 49
Example 34 Indexing parameters vs. indexing field names 50
Example 35 Kinds of methods 51
Example 36 Example: Operator declaration 52
Example 37 Example: Conversion methods 52
Example 38 Dynamic method selection 56
Example 39 User-provided constructor 57
Example 40 User-provided constructor with inheritance 58
Example 41 Disambiguation of method names 58
Example 42 Constraint declaration 59
Example 43 Local fields 62
Example 44 Runtime assertion checking 63
Example 45 Use of a resulting expression 64

Example 46 Statement-level nondetermism 65
Example 47 Quantifying expressions 70
Example 48 Value-level nondeterministic choice 71
Example 49 Selection expressions 71
Example 50 Type conversions of built-in types 73
Example 51 Invocation syntax 74
Example 52 Enumerated types 75
Example 53 `Enumerable structure type` 76
Example 54 Enumerated class 76
Example 55 Update statement 79
Example 56 Update statements 79
Example 57 Parallel update 82
Example 58 Sequential and parallel steps 83
Example 59 Skip statement 84
Example 60 Select expressions 85
Example 61 Namespaces 86
Example 62 Use of qualified names 87
Example 63 Import Directives 87
Example 64 Explicit qualification required 88
Example 65 Delegate 92

13 Grammar

This section provides a summary of the AsmL grammar.

13.1 Lexical level

13.1.1 Identifiers

```
id           ::=  initIdChar { idChar } { '′' }
initIdChar   ::=  letter | ideographic | '@' | '_'
idChar       ::= letter | combining | ideographic
               |  digit | extender | underscore
letter       ::= // per Unicode section 4.5, letter,
                  excluding combining characters
combining    ::= \u20DD | \u20DE | \u20DF | \u20E0
digit        ::= // per Unicode section 4.6, digit char
ideographic  ::= \u2FF0..\u2FFF
extender     ::=  \u00B7 | \u02D0 | \u02D1 | \u0387 | \u0640
               |  \u0E46 | \u0EC6 | \u3005 | \u3031..\u3035
               |  \u309B..\u309D | \u309E | \u30FC..\u30FE
               |  \uFF70 | \uFF9E | \uFF9F
underscore   ::=  \u005F | \uFF3F
```

13.1.2 Literals

```
literal      ::=  null | boolean | integer | real | string | char
```

13.1.3 Boolean literals

```
boolean      ::=  true | false
```

13.1.4 Integer literals

```
integer       ::=  (decimal | hexadecimal) [ integerSuffix ]
decimal       ::=  digits
hexadecimal   ::=  '0' ('x' | 'X') hexDigit { hexDigit }
integerSuffix ::=  'l' | 'L' | 's' | 'S' | 'b' | 'B'
digits        ::=  digit { digit }
hexDigit      ::=  digit | 'a' .. 'f' | 'A' .. 'F'
```

13.1.5 Literals for real numbers

```
real         ::=  digits '.' digits [ exponent ] [ realSuffix ]
exponent     ::=  ('e' | 'E') [ '+' | '-' ] digits
realSuffix   ::=  'f' | 'F'
```

13.1.6 String literals

```
string       ::=  quote { strChar } quote
strChar      ::=  readable | whiteChar | sQuote | '\' esc
readable     ::=  (see text)
quote        ::=  '"'
esc          ::=  'b' | 'f' | 'n' | 't' | 'r'
               |  ('u' hexDigit hexDigit hexDigit hexDigit)
```

13.1.7 Character literals

```
char         ::=  sQuote (readable | quote | '\' esc) sQuote
sQuote       ::=  "'"
```

13.1.8 Keywords

`->`	`_`	`eq`	`initially`	`operator`	`step`
`..`	`{`	`error`	`inout`	`or`	`structure`
`:=`	`|`	`event`	`interface`	`otherwise`	`subset`
`<=`	`}`	`exists`	`internal`	`out`	`subseteq`
`<>`	`abstract`	`explore`	`intersect`	`override`	`sum`
`>=`	`add`	`extends`	`is`	`primitive`	`the`
`(`	`and`	`fixpoint`	`let`	`private`	`then`
`)`	`any`	`for`	`lt`	`procedure`	`throw`
`*`	`as`	`forall`	`lte`	`process`	`to`
`+`	`case`	`foreach`	`match`	`property`	`try`
`,`	`catch`	`from`	`max`	`protected`	`type`
`-`	`choose`	`function`	`me`	`public`	`union`
`.`	`class`	`get`	`merge`	`ref`	`unique`
`/`	`const`	`gt`	`min`	`remove`	`until`
`:`	`constraint`	`gte`	`mod`	`require`	`value`
`;`	`delegate`	`holds`	`mybase`	`resulting`	`values`
`<`	`do`	`if`	`namespace`	`return`	`var`
`=`	`else`	`ifnone`	`ne`	`sealed`	`virtual`
`>`	`elseif`	`implements`	`new`	`search`	`where`
`?`	`ensure`	`implies`	`not`	`set`	`while`
`[`	`enum`	`import`	`notin`	`shared`	
`]`	`enumerated`	`in`	`of`	`skip`	

+= *=			

13.2 Unit of compilation (assembly)

```
assembly     ::= [ namespaceOrDecl ]
namespaceOrDecl  ::= namespace  | declaration
namespace  ::=  [ attributes ] namespace name
name        ::= id  { "." id }
declaration  ::= import  | type  | member
import       ::= import name [ "=" name ]
```

13.3 Values, constructors and patterns

13.3.1 Constructors

```
constructor ::=      literal
              |      datatypeConstructor
              |      collectionConstructor

datatypeConstructor ::= [ new ] typeName [ "(" [ exps ] ")" ]

collectionConstructor ::= tupleExp  | setExp  | seqExp  | mapExp
tupleExp     ::= "("  exp "," exps  ")"
setExp       ::= "{"  [ comprehension  | exps  | range ] "}"
seqExp       ::= "["  [ comprehension  | exps  | range ] "]"
mapExp ::= "{"  ( mapComprehension  | mapExps  | "->") "}"
range        ::= exp ".." exp
comprehension  ::= exp "|" binders
mapComprehension ::=     maplet "|" binders
mapExps      ::= maplet {"," maplet }
maplet       ::= exp "->" exp
```

13.3.2 Patterns

```
pat          ::= "_"
               |  literal
               |  id [ as typeExp ]
               |  tuplePat
               |  datatypePat
               |  mapletPat

tuplePat     ::= "(" pats ")"
datatypePat ::= typeName   [ "(" [ pats ] ")" ]
```

```
mapletPat   ::= pat "->" pat
pats        ::= pat { "," pat }
```

13.3.3 Binders

```
binders      ::=  binder {"," binder}
binder       ::=  pat ( in | "=" ) exp [ where exp ]
```

13.4 Type expressions

```
typeExp      ::= optionType  { or optionType  }
optionType   ::= atomicType  [ "?" ]
atomicType   ::= typeName | "(" typeExp { "," typeExp } ")"
typeName     ::= name [ typeArgs ]
typeArgs     ::= of   optionType  [ to optionType ]
               |  of "<" typeExp { "," typeExp } ">"
```

13.5 Type declarations

```
type         ::= [ attributes ] { typeModifier }
                 ( class  | structure  | interface |
                   enum  | delegate | constrainedType  )
```

13.5.1 Type Parameters

```
typeParams ::= of id  [ to id ]
             |  of "<" typeParam  {"," typeParam } ">"

typeParam   ::= id  [ typeRelations ]
```

13.5.2 Type Relations

```
typeRelations     ::= extends typeExps [ implements typeExps ]
                  |   implements typeExps

typeExps    ::= typeExp { and typeExps }
```

13.5.3 Interface

```
interface    ::= interface id [ typeParams ] [ typeRelations ]
                   [ declaration ]
```

13.5.4 Datatype declaration

```
class        ::= [ enumerated ] class id [ typeParams ]
                 [ typeRelations ]
                 [ variantOrDecl ]
structure    ::= structure id [ typeParams ]
                 [ typeRelations ]
                 [ variantOrDecl ]

variantOrDecl    ::=       declaration | variant

variant      ::= case id [ declaration ]
```

13.5.5 Enumerations

```
enum         ::= enum id [ extends typeExp ] [ element ]
element      ::= id [ "=" exp ]
```

13.5.6 Constrained Types

```
constrainedType ::= type id [ typeParams ] [ "=" valueExp ]
valueExp     ::= typeExp [ where exp ]
```

13.6 Members

```
member       ::= [ attributes ] { memberModifier }
                 ( constant | variable | method |
                   constraint | property | event )

memberModifier ::= shared | virtual | override
                  |  extendedMemberModifier
```

13.6.1 Fields

```
constant     ::= [ const ] id
                    ( as typeExp [ "=" exp ] | "=" exp )

variable     ::= var id ( as typeExp ["=" exp ] | "=" exp )
```

13.6.2 Methods

```
method       ::= [ methodKind ] methodId [ typeParams ]
                 signature [ stm ]
methodKind   ::= function | procedure
methodId     ::= name | operator ( binaryOp | unaryOp )
```

```
signature    ::= params [ result ]
result       ::= as typeExp
params       ::= "(" [ param { "," param } ] ")"
param        ::= [ attributes ] [ paramModifier ]
                 [ id as ] typeExp
```

13.6.3 Constraints

```
constraint   ::= constraint [ label ] exp
label        ::= ( id | literal ) ":"
```

13.7 Statements and expressions

```
stm          ::= local
               |  assert
               |  choice
               |  return
               |  operationalStm
               |  exp

exp          ::= branchExp
               |  exceptExp
               |  quantifierExp
               |  selectExp
               |  binaryExp
               |  enum of type
               |  type of type
               |  do stm
               |  exploration

exps         ::= exp { "," exp }
```

13.7.1 Local fields

```
local        ::= letBinder
               |  { localVariableModifier } localVar
letBinder    ::= [ let ] pat "=" exp
localVar     ::= ( var | initially ) id
                          ( as typeExp ["=" exp ] | "=" exp )
```

13.7.2 Assertion statements

```
assert       ::=  constraint | require | ensure
require      ::=  require [ label ] exp
ensure       ::=  ensure [ label ] exp
```

13.7.3 Nondeterministic choice statements

```
choice       ::=  choose [ unique ] binders stm
                  [ ifnone stm ]
```

13.7.4 Return statements

```
return       ::=  return exp
```

13.7.5 Conditional expressions

```
branchExp    ::=  ifExpr | matchExpr
ifExpr       ::=  if exp [ then ] stm
                  { elseif exp [ then ] stm }
                  [ else stm ]
matchExp     ::=  match exp case [ otherwise stm ]
case         ::=  pat [ where exp ] ":" stm
```

13.7.6 Try/catch expressions

```
exceptExp    ::=  try stm catch case
               |  throw exp
               |  error exp
```

13.7.7 Quantifying expressions

```
quantifierExp    ::= forall binders holds exp
                  |  exists [ unique ] binders
```

13.7.8 Selection expressions

```
selectExp    ::=  selector comprehension [ifnone exp]
selector     ::=  any |the | min | max | sum
```

13.7.9 Primary Expressions

```
binaryExp    ::=  primaryExp { binaryOp primaryExp }
primaryExp   ::=  unaryOp applyExp
               |  applyExp [ ( is | as ) typeExp ]
               |  resulting exp

unaryOp      ::=  not | "-"
binaryOp     ::=  implies | and [ then ] | or [ else ]
               |  "*" | "/" | mod | "+" | "-"
               |  union | intersect | merge
               |  subset | subseteq | in | notin
               |  "=" | "<>" | "<" | ">" | "<=" | ">="
               |  eq  |  ne  | lt  | gt  | lte  | gte
```

13.7.10 Apply expressions

```
applyExp     ::=  atomicExp { argList }
               |  mybase arglist { argList }
argList      ::=  "(" [ exps ] ")" | "." id [ typeArgs ] }
```

13.7.11 Atomic expression

```
atomicExp    ::=  constructor | me | value
               |  "(" exp ")"
               |  id [ typeArgs ]
```

13.8 Runtime states

```
operationalStm   ::=     update
               |  parallelUpdate
               |  sequence
               |  skip
```

13.8.1 Update statements

```
update       ::=  applyExp ( ":=" | "*=" | "+=" ) exp
               |  add exp to applyExp
               |  remove exp [ from applyExp ]
```

13.8.2 Parallel update blocks

```
parallelUpdate ::= forall binders stm
```

13.8.3 Sequential blocks

```
sequence   ::= step
step       ::= step  [ label ] [ iterator ] stm
iterator   ::= foreach binders
             |  for id "=" exp to exp
             |  while exp
             |  until ( exp | fixpoint )
```

13.8.4 Exploration expressions

```
exploration ::= explore exp
              |  search exp
```

13.9 .NET Compatibility

13.9.1 Modifiers

```
typeModifier      ::= extensibility | access

access            ::= public | private | protected | internal
extensibility     ::= abstract | sealed

extendedMemberModifier ::= extensibility | access | primitive

paramModifier     ::= primitive ref | primitive out
                    |  out | inout

localVariableModifier ::= primitive
```

13.9.2 Attributes

```
attributes    ::= { attribute }
attribute     ::= "[" [ target ] attributeConstructor
                     { "," attributeConstructor } "]"
target        ::= id ":"
attributeConstructor ::=   id  | id "(" attributeExps ")"
attributeExps::= [ exps ] [ namedAttrArgs ]
namedAttrArgs   ::= [ namedAttributeArg  { "," namedAttrArg } ]
namedAttrArg    ::=       id "=" exp
```

13.9.3 Delegates

```
delegate    ::= delegate  id  [ typeParams ] signature
```

13.9.4 Properties

```
property    ::= property  ( name | me  ) [ params ] as  typeExp
                ( setter [ getter ] | getter [ setter ] )
setter      ::= set  [ stm  ]
getter      ::= get  [ stm  ]
```

13.9.5 Events

```
event       ::= event name as  typeExp
                (  adder  [  remover  ]  |  remover  [  adder  ]  )
adder       ::= add  [ stm  ]
remover     ::= remove  [  stm  ]
```