1 Front Matter

1.1 Title

1.1.1 The Micron Intermediate Language Specification

1.2 Version

1.2.1 2024-04-16, work in progress

1.3 Author

1.3.1 me@rochus-keller.ch

1.4 Additional Credits

1.4.1 This specification was derived from the ECMA-335 standard (3rd edition 2005).

1.5 License

1.6 Table of Contents

2 Language Specification

2.1 Overview

2.1.1 The Micron Intermediate Languae (MIL) uses a subset of the ECMA-335 Common Intermediate Language (CIL) without managed types, and with manual memory management, and extended by explicit control-flow syntax. Because of the latter no explicit branch instructions are required, and different types of analyses, as well as translations to high-level languages like C or lower-level representations like CIL or LLVM IR are straight-forward.

2.1.2 In contrast to CIL, MIL targets the feature set of languages like Micron or C. The OO features of CIL are not supported, but could be implemented on top of MIL if required. MIL is designed for direct compatibility with the C ABI of all supported architectures.

2.2 Representations

2.2.1 MIL exists in two equivalent representations: a human-readable textual representation, and an efficient binary representation. Both map to a common structure.

2.2.2 Textual Representation

2.2.2.1 The MIL textual representation is a string of characters encoded using the UTF-8 variable-width encoding as defined in ISO/IEC 10646. Identifiers, numbers, operators, and delimiters are represented using the ASCII character set; strings and comments can be either represented in the ASCII or Latin-1 (as defined in ISO/IEC 8859-1) character set.

2.2.2.2 Syntax

2.2.2.2.1 An extended Backus-Naur Formalism (EBNF) is used to describe the textual syntax of MIL:

2.2.2.2.1.1 Alternatives are separated by **|**.

2.2.2.2.1.2 Brackets **[** and **]** denote optionality of the enclosed expression.

2.2.2.2.1.3 Braces **{** and **}** denote its repetition (possibly 0 times).

2.2.2.2.1.4 Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning.

2.2.2.2.1.5 Symbols of the language vocabulary (terminal symbols) are denoted by strings formatted in bold face.

2.2.2.3 Identifiers

2.2.2.3.1 Identifiers are sequences of letters, digits and dollar signs. The first character cannot be a digit.

2.2.2.3.2 Syntax:

2.2.2.3.2.1
```
ident  = ( letter | '_' | '$' ) { letter | digit | '_' | '$' }

letter = 'A' ... 'Z' | 'a' ... 'z'

digit  = '0' ... '9'
```

2.2.2.4 Numbers

2.2.2.4.1 Number literals are optionaly signed integer or real constants.

2.2.2.4.2 If the literal is specified with the suffix 'H', the representation is hexadecimal, if it is specified with suffix 'O', the representation is ocal, or if it is specified with suffic 'B', the representation is binary, otherwise the representation is decimal.

2.2.2.4.3 A real number always contains a decimal point and at least one digit before the point. Optionally it may also contain a decimal scale factor. The letter 'E' means *times ten to the power of*.

2.2.2.4.4 Syntax:

2.2.2.4.4.1
```
unsigned  = (digit {digit} ['O'|'B' | digit {hexDigit} 'H')
integer = ['+' | '-'] unsigned

real     = ['+' | '-'] digit {digit} '.' {digit} [Exponent]

Exponent = 'E' ['+' | '-'] digit {digit}

hexDigit = digit | 'A' ... 'F'

digit    = '0' ... '9'
```

2.2.2.4.5 Even though only the upper-case version is shown here, also the lower-case versions of the suffices and scale factor letters are supported.

2.2.2.5 Characters

2.2.2.5.1 Character constants are denoted by the ordinal number of the character in hexadecimal notation followed by the letter X (or x).

2.2.2.5.2 Syntax:

2.2.2.5.2.1
```
character = digit {hexDigit} ('X' | 'x')
```

2.2.2.5.3 A character is encoded as a 8-bit code value using the ISO/IEC 8859-1 Latin-1 encoding scheme.

2.2.2.6 Strings

2.2.2.6.1 Strings are sequences of printable characters enclosed in single (') or double (") quote marks. The opening quote must be the same as the closing quote and must not occur within the string. A string must not extend over the end of a line. The number of characters in a string is called its length. A string of length 1 can be used wherever a character constant is allowed and vice versa.

2.2.2.6.2 Syntax:

2.2.2.6.2.1
```
string = ''' {character} ''' | '"' {character} '"'
```

2.2.2.6.3 Hex Strings

2.2.2.6.3.1 Hex strings are sequences of bytes encoded in hexadecimal format and enclosed in dollar signs. The number of hex digits in the string must be even, two hex digits per byte. The number of bytes in a hex string is called its length. Line breaks and other white space between the dollar signs is ignored. Both upper and lower-case hex digits are supported.

2.2.2.6.3.2 Syntax:

2.2.2.6.3.2.1
```
hexstring = '#' {hexDigit} '#'
```

2.2.2.6.3.3 Examples:

2.2.2.6.3.3.1
```
const arrow = #0F0F 0060 0070 0038 001C 000E 0007 8003 C101 E300
               7700 3F00 1F00 3F00 7F00 FF00#
```

2.2.2.7 Reserved Words

2.2.2.7.1 MIL has no reserved words; a lot of identifiers have dedicated meanings in a specific context of the source code, which usually doesn't interfere with field, parameter and variable naming.

2.2.2.8 Comments

2.2.2.8.1 Comments are arbitrary character sequences opened by the bracket `(*` and closed by `*)`. Comments may be nested. They do not affect the meaning of a program. Oberon+ also supports line comments; text starting with `//` up to a line break is considered a comment.

2.2.3 Binary Representation

2.2.3.1 TODO

2.3 Declarations and scope rules

2.3.1 Every identifier occurring in a program must be introduced by a declaration, unless it is a predeclared identifier. Declarations also specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, or a procedure. The identifier is then used to refer to the associated object.

2.3.2 The *scope* of an object *x* extends textually from the point of its declaration to the end of the block (module, procedure, struct, or union) to which the declaration belongs and hence to which the object is *local*.

2.3.2.1 No identifier may denote more than one object within a given scope;

2.3.2.2 A type*T* of the form POINTER TO *T1* can be declared at a point where *T1* is still unknown. The declaration of *T1* must follow in the same block to which *T* is local;

2.3.3 An identifier declared in a module block may be followed by an export mark ("*") in its declaration to indicate that it is exported. An identifier *x* exported by a module *M* may be used in other modules, if they import *M*. The identifier is then denoted as *M.x* in these modules and is called a *qualified identifier*.

2.3.4 Syntax of textual representation:

2.3.4.1
```
qualident = [ident '.'] ident
identdef = ident ['*']
```

2.4 Types

2.4.1 A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration associates an identifier with a type. In the case of structured types (arrays, structs and unions) it also defines the structure of variables of this type. A structured type cannot contain itself.

2.4.2 Syntax of textual representation:

2.4.2.1
```
TypeDeclaration = identdef '=' type

type            = NamedType | ArrayType | StructUnionType | PointerType | ProcedureType
```

2.4.3 Basic types

2.4.3.1 The basic types are denoted by predeclared identifiers. Either all capital or all lower case identifiers are supported (only lower case versions shown).

2.4.3.2

Name	Description
bool	True (1) / false (0) value
char	Latin-1 8-bit char

float32	IEC 60559:1989 32-bit float
float64	IEC 60559:1989 64-bit float
int8	Signed 8-bit integer
int16	Signed 16-bit integer
int32	Signed 32-bit integer
int64	Signed 64-bit integer
intptr	Signed integer address, native size
uint8	Unsigned 8-bit integer
uint16	Unsigned 16-bit integer
uint32	Unsigned 32-bit integer
uint64	Unsigned 64-bit integer

2.4.4 Alias types

2.4.4.1 A type can be an alias of another type. The other type is referenced with a qualident. If the other type is in another module, it must be exported. A type is considered the same type as its alias.

2.4.4.2 Syntax of textual representation:

2.4.4.2.1
```
NamedType = qualident
```

2.4.5 Structured types

2.4.5.1 Array types

2.4.5.1.1 An array is a structure consisting of a number of elements which are all of the same type, called the element type. The number of elements of an array is called its length. The length is a positive integer (uint32). The elements of the array are designated by indices, which are integers between 0 and the length minus 1.

2.4.5.1.2 Syntax of textual representation:

2.4.5.1.2.1
```
ArrayType = ARRAY [ length ] OF NamedType | '[' [ length ] ']' NamedType
length = unsigned
```

2.4.5.1.3 Arrays declared without length are called *open arrays*. They are restricted to pointer base types, and element types of open array types.

2.4.5.2 Struct and union types

2.4.5.2.1 MIL struct and union types correspond to the ones of the C programming language, including alignment and layout rules.

2.4.5.2.2 A struct type is a structure consisting of a fixed number of elements, called fields, with possibly different types. The struct type declaration specifies the name and type of each field. The scope of the field identifiers extends from the point of their declaration to the end of the struct type, but they are also visible within designators referring to elements of struct variables. If a struct type is exported, field identifiers that are to be visible outside the declaring module must be marked. They are called public fields; unmarked elements are called private fields.

2.4.5.2.3 A union type is declared like a struct type, but in contrast to the latter all fields reside at the same memory position. A union can be pictured as a chunk of memory that is used to store variables of different data types. Once a new value is assigned to a field, the existing data is overwritten with the new data.

2.4.5.2.4 Syntax of textual representation:

2.4.5.2.4.1
```
StructUnionType = (STRUCT | UNION) { FieldList [';'] } END
FieldList = IdentList ':' NamedType
IdentList = identdef { [','] identdef }
```

2.4.6 Pointer types

2.4.6.1 Variables of a pointer type P assume as values pointers to variables of some type T. T is called the pointer base type of P and can be of any type.

2.4.6.2 Syntax of textual representation:

2.4.6.2.1
```
PointerType = ( POINTER TO | '^' ) NamedType
```

2.4.6.3 Any pointer variable or parameter may assume the value NIL, which points to no variable at all.

2.4.7 Procedure types

2.4.7.1 Variables of a procedure type T have a pointer to a procedure (or NIL) as value. If a procedure P is assigned to a variable of type T, the formal parameter lists and result types of P and T must *match.*

2.4.7.2 Syntax of textual representation:

2.4.7.2.1
```
ProcedureType = (PROCEDURE|PROC) [FormalParameters]
```

2.5 Expressions

2.5.1 Expressions are calculated using a conceptual evaluation stack. The actual implementation is not supposed to be stack based. The conceptual evaluation stack is made up of conceptual slots that can hold any data type.

2.5.2 An expression consists of a sequence of instructions which push or pop values of a specified type to and from the evaluation stack.

2.5.3 The evaluation stack is empty upon procedure entry. Its contents are entirely local to a procedure and are preserved across *call* instructions. Arguments to other procedures and their return values are also placed on the evaluation stack. The evaluation stack is not addressable. At all times it is possible to deduce which one of a reduced set of types is stored in any stack location at a specific point in the instruction stream.

2.5.4 While MIL, in general, supports the full set of types described in 2.4 Types, MIL only requires that values be one of:

2.5.4.1 int64 , an 8-byte signed integer

2.5.4.2 int32 , a 4-byte signed integer

2.5.4.3 intptr, a signed integer of either 4 or 8 bytes, whichever is more convenient for the target architecture

2.5.4.4 F , a floating point value (float32 , float64 , or other representation supported by the underlying hardware)

2.5.4.5 a user-defined value type

2.5.5 Syntax of textual representation:

2.5.5.1
```
Expression = { ExpInstr }
ExpInstr = 'add' | 'and' | ('call' | 'calli') qualident | 'castptr' qualident
| 'ceq' | 'cgt' | 'cgt_un' | 'clt' | 'clt_un'
| 'conv_i1' | 'conv_i2' | 'conv_i4' | 'conv_i8' | 'conv_r4' | 'conv_r8'
| 'conv_u1' | 'conv_u2' | 'conv_u4' | 'conv_u8' | 'conv_ip'
| 'div' | 'div_un' | 'dup' | 'initobj' qualident | ('ldarg'|'ldarg_s') (unsigned|ident)
| 'ldarg_0' | 'ldarg_1' | 'ldarg_2' | 'ldarg_3'
| ('ldarga'|'ldarga_s') (unsigned|ident)
| ('ldc_i4'|'ldc_i8'|'ldc_i4_s') integer | ('ldc_r4'|'ldc_r8') real
| 'ldc_i4_0' | 'ldc_i4_1' | 'ldc_i4_2' | 'ldc_i4_3' | 'ldc_i4_4' | 'ldc_i4_5'
| 'ldc_i4_6' | 'ldc_i4_7' | 'ldc_i4_8' | 'ldc_i4_m1' | 'ldc_obj' constructor
| ('ldelem'|'ldelema') qualident | 'ldelem_i1' | 'ldelem_i2'
| 'ldelem_i4' | 'ldelem_i8' | 'ldelem_u1' | 'ldelem_u2'
| 'ldelem_u4' | 'ldelem_u8' | 'ldelem_r4' | 'ldelem_r8' | 'ldelem_ip'
| ('ldfld'|'ldflda') qualident '.' ident | 'ldftn' qualident
| 'ldind_i1' | 'ldind_i2' | 'ldind_i4' | 'ldind_i8' | 'ldind_u1' | 'ldind_u2'
| 'ldind_u4' | 'ldind_r4' | 'ldind_u8' | 'ldind_r8' | 'ldind_ip'
| ('ldloc'|'ldloc_s'|'ldloca'|'ldloca_s')(unsigned|ident)
| 'ldloc_0' | 'ldloc_1' | 'ldloc_2' | 'ldloc_3' | 'ldnull'
| 'ldobj' qualident | 'ldproc' qualident | 'ldstr' string
| ('ldvar'|'ldvara') qualident | 'mul' | 'neg'
| ('newarr'|'newvla'|'newobj') qualident
| 'not' | 'or' | 'rem' | 'rem_un' | 'shl' | 'shr' | 'shr_un' | 'sub' | 'xor'
```

2.5.5.2
```
ConstExpression = qualident | integer | real | string | hexstring
```

2.5.6 TODO: keep designators to ease alias analysis.

2.5.7 Constructors

2.5.7.1 With constructors, struct, union, array and pointer literals can be declared.

2.5.7.2 Syntax of textual representation:

2.5.7.2.1
```
constructor = NamedType component_list
component_list = '{' [ component {[','] component} ] '}'
component = [ident '='] ( ConstExpression | component_list )
```

2.5.7.3 A constructor consists of an explicit type and a list of either named or anonymous components. Named and anonymous components cannot be mixed in the list.

2.5.7.4 If NamedType is a struct type, then there is either an anonymous component for each field of the struct in the order of declaration, or there is a named component for each field in arbitrary order. For union types, only named components can be used, and only one option of the union can be initialized in the constructor.

2.5.7.5 If NamedType is an array type, then there is an anonymous componend for each element of the array. The array type may be an open array in which case the number of elements is determined by the number of components.

2.5.7.6 If NamedType is a pointer type, then there is exactly one anonymous component which is an unsigned integer type constant representing the address.

2.5.7.7 For each field or element which is of struct, union, array or pointer type, an embedded constructor is required. Since the exact type of the field or element is known, the NamedType prefix is not required.

2.5.8 add

2.5.8.1

Binary	Text Format	Description
58	add	Add two values, returning a new value.

2.5.8.2 `…, value1, value2 -> …, result`

2.5.8.3 The add instruction adds *value2* to *value1* and pushes the result on the stack. Overflow is not detected for integral operations; floating-point overflow returns +inf or -inf.

2.5.8.4 The acceptable operand types and their corresponding result data type are encapsulated as follows:

2.5.8.4.1

A's Type	B's Type				
int32	**int64**	**intptr**	**F**		
int32	int32	x	intptr	x	
int64	x	int64	x	x	
intptr	intptr	x	intptr	x	
F	x	x	x	F	

2.5.9 and – bitwise AND

2.5.9.1

Binary	Text Format	Description
5F	and	Bitwise and of two integral values, returns an integral value.

2.5.9.2 `…, value1, value2 -> …, result`

2.5.9.3 The and instruction computes the bitwise and of *value1* and *value2* and pushes the result on the stack. The acceptable operand types and their corresponding result data type are encapsulated as follows:

2.5.9.3.1

	int32	**int64**	**intptr**	**F**
int32	int32	x	intptr	x
int64	x	int64	x	x
intptr	intptr	x	intptr	x
F	x	x	x	x

2.5.10 call – call a function

2.5.10.1

Binary	Text Format	Description
28 <*T*>	call *function*	Call function described by *function.*

2.5.10.2 `…, arg1, arg2 … argN -> …, retVal`

2.5.10.3 `…, arg1, arg2 … argN -> …`

2.5.10.4 The call instruction calls the function indicated by the descriptor. The latter indicates the function to call, and the number, type, and order of the arguments that have been placed on the stack to be passed to that function, or possibly returned from the function. The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, and so on

2.5.10.5 TODO: tail calls

2.5.11 calli – indirect function call

2.5.11.1

Binary	Text Format	Description
29 <*T*>	calli *function*	Call function indicated on the stack with arguments described by *function*.

2.5.11.2 `…,ftn, arg1, arg2 … argN -> …, retVal`

2.5.11.3 `…,ftn, arg1, arg2 … argN -> …`

2.5.11.4 The calli instruction calls *ftn* (a pointer to a function entry point) with the arguments arg1 … argN. The types of these arguments, and a possible return type, are described by the signature. The *ftn* argument is assumed to be a pointer to native code (of the target machine) that can be legitimately called with the arguments described by the signature. Such a pointer can be created using the ldftn instruction, or could have been passed in from external code. The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, and so on.

2.5.11.5 **NOTE** In MIL, the function pointer (ftn) is pushed on the stack first, before the arguments, in contrast to ECMA-335 CIL, where the function pointer is pushed after the arguments.

2.5.11.6 TODO: tail call

2.5.12 castptr – cast a pointer to a type

2.5.12.1

Binary	Text Format	Description
74 <*T*>	castptr *type*	Cast pointer *ptr* to *type*

2.5.12.1.1 `…, ptr -> …, ptr2`

2.5.12.1.2 The castptr instruction reinterprets the pointer to type1 on the stack as a pointer to type2.

2.5.12.1.3 If *ptr* is null, castptr succeeds and returns null.

2.5.13 ceq – compare equal

2.5.13.1

Binary	Text Format	Description
FE 01	ceq	Push 1 (of type int32) if *value1* equals *value2*, else push 0.

2.5.13.2 `…, value1, value2 -> …, result`

2.5.13.3 The ceq instruction compares *value*1 and *value*2. If *value*1 is equal to *value*2, then 1 (of type int32) is pushed on the stack. Otherwise, 0 (of type int32) is pushed on the stack. For floating-point numbers, ceq will return 0 if the numbers are unordered (either or both are NaN). The infinite values are equal to themselves.

2.5.13.4 The acceptable operand types are encapsulated as follows:

2.5.13.4.1

	int32	int64	intptr	F
int32	ok	x	ok	x
int64	x	ok	x	x
intptr	ok	x	ok	x
F	x	x	x	ok

2.5.14 cgt – compare greater than

2.5.14.1

Binary	Text Format	Description
FE 02	cgt	Push 1 (of type int32) if *value1* > *value2*, else push 0.

2.5.14.2 The cgt instruction compares *value1* and *value2*. If *value1* is strictly greater than *value2*, then 1 (of type int32) is pushed on the stack. Otherwise, 0 (of type int32) is pushed on the stack. For floating-point numbers, cgt returns 0 if the numbers are unordered (that is, if one or both of the arguments are NaN). As with IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g., +infinity > 5.0 > -infinity).

2.5.14.3 The acceptable operand types are encapsulated in table 2.5.13.4.1.

2.5.15 cgt_un – compare greater than, unsigned or unordered

2.5.15.1

Binary		

	Text Format	Description
FE 03	cgt_un	Push 1 (of type int32) if *value1* > *value2*, unsigned or unordered, else push 0.

2.5.15.2 `…, value1, value2 -> …, result`

2.5.15.3 The cgt.un instruction compares *value1* and *value2*. A value of 1 (of type int32) is pushed on the stack if

2.5.15.3.1 for floating-point numbers, either value1 is strictly greater than value2, or value1 is not ordered with respect to value2.

2.5.15.3.2 for integer values, value1 is strictly greater than value2 when considered as unsigned numbers. Otherwise, 0 (of type int32) is pushed on the stack.

2.5.15.4 As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g., +infinity > 5.0 > -infinity).

2.5.15.5 The acceptable operand types are encapsulated in table 2.5.13.4.1.

2.5.16 clt – compare less than

2.5.16.1

Binary	Text Format	Description
FE 04	clt	Push 1 (of type int32) if *value1* < *value2*, else push 0.

2.5.16.2 `…, value1, value2 -> …, result`

2.5.16.3 The clt instruction compares *value1* and *value2*. If *value1* is strictly less than *value2*, then 1 (of type int32) is pushed on the stack. Otherwise, 0 (of type int32) is pushed on the stack. For floating-point numbers, clt will return 0 if the numbers are unordered (that is, one or both of the arguments are NaN). As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g., +infinity > 5.0 > -infinity).

2.5.16.4 The acceptable operand types are encapsulated in table 2.5.13.4.1.

2.5.17 clt_un – compare less than, unsigned or unordered

2.5.17.1

Binary	Text Format	Description
FE 05	clt_un	Push 1 (of type int32) if *value1* < *value2*, unsigned or unordered, else push 0.

2.5.17.2 `…, value1, value2 -> …, result`

2.5.17.3 The clt.un instruction compares *value1* and *value2*. A value of 1 (of type int32) is pushed on the stack if

2.5.17.3.1 for floating-point numbers, either value1 is strictly less than value2, or value1 is not ordered with respect to value2.

2.5.17.3.2 for integer values, value1 is strictly less than value2 when considered as unsigned numbers. Otherwise, 0 (of type int32) is pushed on the stack.

2.5.17.4 As per IEC 60559:1989, infinite values are ordered with respect to normal numbers (e.g., +infinity > 5.0 > -infinity).

2.5.17.5 The acceptable operand types are encapsulated in table 2.5.13.4.1.

2.5.18 conv – data conversion

2.5.18.1

Binary	Text Format	Description
67	conv_i1	Convert to int8, pushing int32 on stack.
68	conv_i2	Convert to int16, pushing int32 on stack.
69	conv_i4	Convert to int32, pushing int32 on stack.
6A	conv_i8	Convert to int64, pushing int64 on stack.
6B	conv_r4	Convert to float32, pushing F on stack.
6C	conv_r8	Convert to float64, pushing F on stack.
D2	conv_u1	Convert to unsigned int8, pushing int32 on stack.
D1	conv_u2	Convert to unsigned int16, pushing int32 on stack.
6D	conv_u4	Convert to unsigned int32, pushing int32 on stack.
6E	conv_u8	Convert to unsigned int64, pushing int64 on stack.

D3	conv_ip	Convert to intptr, pushing intptr on stack.

2.5.18.2 `…, value -> …, result`

2.5.18.3 Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. Note that integer values of less than 4 bytes are extended to int32 (not intptr) when they are loaded onto the evaluation stack, and floating-point values are converted to the F type.

2.5.18.4 Conversion from floating-point numbers to integral values truncate the number toward zero. When converting from a float64 to a float32, precision might be lost. If *value* is too large to fit in a float32, the IEC 60559:1989 positive infinity (if *value* is positive) or IEC 60559:1989 negative infinity (if *value* is negative) is returned. If overflow occurs when converting one integer type to another, the high-order bits are silently truncated. If the result is smaller than an int32, then the value is sign-extended to fill the slot.

2.5.18.5 If overflow occurs converting a floating-point type to an integer, or if the floating-point value being converted to an integer is a NaN, the value returned is unspecified.

2.5.18.6 The acceptable operand types and their corresponding result data type is encapsulated as follows:

2.5.18.6.1

Convert-To	**Input (from evaluation stack)**			
int32	**int64**	**intptr**	**F**	
int8 **uint8** **int16** **uint16**	Truncate[1]	Truncate[1]	Truncate[1]	Truncate to zero[2]
int32 **uint32**	Nop	Truncate[1]	Truncate[1]	Truncate to zero[2]
int64	Sign extend	Nop	Sign extend	Truncate to zero[2]
uint64	Zero extend	Nop	Zero extend	Truncate to zero[2]
intptr	Sign extend	Truncate[1]	Nop	Truncate to zero[2]
All Float Types	To Float	To Float	To Float	Change precision[3]

2.5.18.6.2 1) “Truncate” means that the number is truncated to the desired size (i.e., the most significant bytes of the input value are simply ignored). If the result is narrower than the minimum stack width of 4 bytes, then this result is zero extended (if the target type is unsigned) or sign-extended (if the target type is signed). Thus, converting the value 0x1234 ABCD from the evaluation stack to an 8-bit datum yields the result 0xCD; if the target type were int8, this is sign-extended to give 0xFFFF FFCD; if, instead, the target type were unsigned int8, this is zero-extended to give 0x0000 00CD.

2.5.18.6.3 2) “Truncate to zero” means that the floating-point number will be converted to an integer by truncation toward zero. Thus 1.1 is converted to 1, and –1.1 is converted to –1.

2.5.18.6.4 3) Converts from the current precision available on the evaluation stack to the precision specified by the instruction. If the stack has more precision than the output size the conversion is performed using the IEC 60559:1989 “round-to-nearest” mode to compute the low order bit of the result.

2.5.19 div – divide values

2.5.19.1

Binary	**Text Format**	**Description**
5B	div	Divide two values to return a quotient or floating-point result.

2.5.19.2 `…, value1, value2 -> …, result`

2.5.19.3 *result* = *value1* div *value2* satisfies the following conditions:

2.5.19.3.1 |*result*| = |*value1*| / |*value2*|, and

2.5.19.3.2 *sign(result) = +, if sign(value1) = sign(value2),* or
–, if sign(value1) ~= sign(value2)

2.5.19.4 The div instruction computes *result* and pushes it on the stack.

2.5.19.5 Integer division truncates towards zero.

2.5.19.6 Floating-point division is per IEC 60559:1989. In particular, division of a finite number by 0 produces the correctly signed infinite value and

2.5.19.6.1 0 / 0 = NaN

2.5.19.6.2 infinity / infinity = **NaN**

2.5.19.6.3 X / infinity = 0

2.5.19.7 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.8.4.1.

2.5.20 div_un – divide integer values, unsigned

2.5.20.1

Binary	Text Format	Description
5C	div_un	Divide two values, unsigned, returning a quotient.

2.5.20.2 `…, value1, value2 -> …, result`

2.5.20.3 The div_un instruction computes value1 divided by value2, both taken as unsigned integers, and pushes the result on the stack.

2.5.20.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.8.4.1.

2.5.21 dup – duplicate the top value of the stack

2.5.21.1

Binary	Text Format	Description
25	dup	Duplicate the value on the top of the stack.

2.5.21.2 `…, value -> …, value, value`

2.5.21.3 The dup instruction duplicates the top element of the stack.

2.5.22 initobj – initialize the value at an address

2.5.22.1

Binary	Text Format	Description
FE 15 <*T*>	initobj *type*	Initialize the value at address *dest*.

2.5.22.2 `…, dest -> …,`

2.5.22.3 The initobj instruction initializes a record or fixed length array with a default value. *type* is a reference to the declaration. *dest* is a pointer to the record or array. The initobj instruction initializes each field/element of *dest* to null or a zero of the appropriate built-in type.

2.5.22.4 TODO: do we really need this?

2.5.23 ldarg – load argument onto the stack

2.5.23.1

Binary	Text Format	Description
FE 09 <*uint16*>	ldarg *num*	Load argument numbered *num* onto the stack.
0E <*uint8*>	ldarg_s *num*	Load argument numbered *num* onto the stack, short form.
02	ldarg_0	Load argument 0 onto the stack.
03	ldarg_1	Load argument 1 onto the stack.
04	ldarg_2	Load argument 2 onto the stack.
05	ldarg_3	Load argument 3 onto the stack.

2.5.23.2 `… -> …, value`

2.5.23.3 The ldarg *num* instruction pushes onto the evaluation stack, the *num*'th incoming argument, where arguments are numbered 0 onwards. The ldarg instruction can be used to load a value type or a built-in value onto the stack by copying it from an incoming argument. The type of the value is the same as the type of the argument, as specified by the current method's signature.

2.5.23.4 The ldarg.0, ldarg.1, ldarg.2, and ldarg.3 instructions are efficient encodings for loading any one of the first 4 arguments. The ldarg.s instruction is an efficient encoding for loading argument numbers 4–255.

2.5.23.5 For procedures that take a variable-length argument list, the ldarg instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

2.5.23.6 Arguments that hold an integer value smaller than 4 bytes long are expanded to type int32 when they are loaded onto the stack. Floating-point values are expanded to their native size (type F).

2.5.24 ldarga – load an argument address

2.5.24.1

Binary	Text Format	Description
FE 0A <*uint16*>	ldarga *argNum*	Fetch the address of argument *argNum*.
0F <*uint8*>	ldarga_s *argNum*	Fetch the address of argument *argNum*, short form.

2.5.24.2 `…, -> …, address of argument number argNum`

2.5.24.3 The ldarga instruction fetches the address (of type intptr) of the *argNum*'th argument, where arguments are numbered 0 onwards. The address will always be aligned to a natural boundary on the target machine. The short form (ldarga.s) should be used for argument numbers 0–255.

2.5.24.4 For procedures that take a variable-length argument list, the ldarga instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

2.5.25 ldc – load numeric constant

2.5.25.1

Binary	Text Format	Description
20 <*int32*>	ldc_i4 *num*	Push *num* of type int32 onto the stack as int32.
21 <*int64*>	ldc_i8 *num*	Push *num* of type int64 onto the stack as int64.
22 <*float32*>	ldc_r4 *num*	Push *num* of type float32 onto the stack as F.
23 <*float64*>	ldc_r8 *num*	Push *num* of type float64 onto the stack as F.
16	ldc_i4_0	Push 0 onto the stack as int32.
17	ldc_i4_1	Push 1 onto the stack as int32.
18	ldc_i4_2	Push 2 onto the stack as int32.
19	ldc_i4_3	Push 3 onto the stack as int32.
1A	ldc_i4_4	Push 4 onto the stack as int32.
1B	ldc_i4_5	Push 5 onto the stack as int32.
1C	ldc_i4_6	Push 6 onto the stack as int32.
1D	ldc_i4_7	Push 7 onto the stack as int32.
1E	ldc_i4_8	Push 8 onto the stack as int32.
15	ldc_i4_m1	Push -1 onto the stack as int32.
1F <*int8*>	ldc_i4_s *num*	Push *num* onto the stack as int32, short form.

2.5.25.2 `… -> …, num`

2.5.25.3 The ldc *num* instruction pushes number *num* or some constant onto the stack. There are special short encodings for the integers –128 through 127 (with especially short encodings for –1 through 8). All short encodings push 4-byte integers on the stack. Longer encodings are used for 8-byte integers and 4- and 8-byte floating-point numbers, as well as 4-byte values that do not fit in the short forms.

2.5.25.4 There are three ways to push an 8-byte integer constant onto the stack

2.5.25.4.1 For constants that shall be expressed in more than 32 bits, use the ldc.i8 instruction.

2.5.25.4.2 For constants that require 9–32 bits, use the ldc.i4 instruction followed by a conv.i8.

2.5.25.4.3 For constants that can be expressed in 8 or fewer bits, use a short form instruction followed by a conv.i8.

2.5.25.5 There is no way to express a floating-point constant that has a larger range or greater precision than a 64-bit IEC 60559:1989 number, since these representations are not portable across architectures.

2.5.26 ldc_obj – load a literal object to the stack

2.5.26.1

Binary	Text Format	Description
EE 12 <*T*>	ldc_obj *constructor*	Load the constructed literal object to the stack.

2.5.26.2 `… -> …, val`

2.5.26.3 The ldc_obj instruction pushes the object literal constructed by *constructor* onto the stack.

2.5.27 ldelem – load element from array

2.5.27.1

Binary	Text Format	Description
A3 <*T*>	ldelem *type*	Load the element at *index* onto the top of the stack.

2.5.27.2

90	ldelem_i1	Load the element with type int8 at *index* onto the top of the stack as an int32.
92	ldelem_i2	Load the element with type int16 at *index* onto the top of the stack as an int32.
94	ldelem_i4	Load the element with type int32 at *index* onto the top of the stack as an int32.
96	ldelem_i8	Load the element with type int64 at *index* onto the top of the stack as an int64.
91	ldelem_u1	Load the element with type unsigned int8 at *index* onto the top of the stack as an int32.
93	ldelem_u2	Load the element with type unsigned int16 at *index* onto the top of the stack as an int32.
95	ldelem_u4	Load the element with type unsigned int32 at *index* onto the top of the stack as an int32.
96	ldelem_u8	Load the element with type unsigned int64 at *index* onto the top of the stack as an int64 (alias for ldelem.i8).
98	ldelem_r4	Load the element with type float32 at *index* onto the top of the stack as an F
99	ldelem_r8	Load the element with type float64 at *index* onto the top of the stack as an F.
97	ldelem_ip	Load the element with type intptr at *index* onto the top of the stack as a intptr.

2.5.27.3 `…, pointer to array, index -> …, value`

2.5.27.4 The ldelem instruction loads the value of the element with index *index* (of type intptr or int32) in the zero-based one-dimensional array (pointed to by the intptr on the stack), and places it on the top of the stack. The type of the return value is indicated by the type in the instruction.

2.5.28 ldelema – load address of an element of an array

2.5.28.1

Binary	Text Format	Description
8F <*T*>	ldelema *type*	Load the address of element at *index* of type onto the top of the stack.

2.5.28.2 `…, pointer to array, index -> …, address`

2.5.28.3 The ldelema instruction loads the address of the element with index *index* (of type int32 or intptr) in the zero-based one-dimensional array *array* (pointed to by the intptr on the stack, of element type *type)* and places it on the top of the stack.

2.5.29 ldfld – load field of a struct or union

2.5.29.1

Binary	Text Format	Description
7B <*T*>	ldfld *field*	Push the value of *field* of struct or union *ptr,* onto the stack.

2.5.29.2 `…, ptr -> …, value`

2.5.29.3 The ldfld instruction pushes onto the stack the value of a field of *ptr*. *ptr* shall be an of type intptr. *field* is a reference to an element of a structured type. The return type is that associated with *field*. ldfld pops the pointer off the stack and pushes the value for the field in its place.

2.5.30 ldflda – load field address

2.5.30.1

Binary	Text Format	Description
7C <*T*>	ldflda *field*	Push the address of *field* of struct or union *ptr* on the stack.

2.5.30.2 `…, ptr -> …, address`

2.5.30.3 The ldflda instruction pushes the address of a field of *ptr*. *ptr* is of type intptr. The value returned by ldflda is an intptr. *field* is a reference to an element of a structured type

2.5.31 ldind – load value indirect onto the stack

2.5.31.1

Binary	Text Format	Description
46	ldind_i1	Indirect load value of type int8 as int32 on the stack.
48	ldind_i2	Indirect load value of type int16 as int32 on the stack.
4A	ldind_i4	Indirect load value of type int32 as int32 on the stack.
4C	ldind_i8	Indirect load value of type int64 as int64 on the stack.
47	ldind_u1	Indirect load value of type unsigned int8 as int32 on the stack.
49	ldind_u2	Indirect load value of type unsigned int16 as int32 on the stack.
4B	ldind_u4	Indirect load value of type unsigned int32 as int32 on the stack.
4E	ldind_r4	Indirect load value of type float32 as F on the stack.
4C	ldind_u8	Indirect load value of type unsigned int64 as int64 on the stack (alias for ldind.i8).
4F	ldind_r8	Indirect load value of type float64 as F on the stack.
4D	ldind_ip	Indirect load value of type intptr on the stack

2.5.31.2 `…, addr -> …, value`

2.5.31.3 The ldind instruction indirectly loads a value from address *addr* (an intptr) onto the stack. The source value is indicated by the instruction suffix.

2.5.31.4 Note that integer values of less than 4 bytes are extended to int32 when they are loaded onto the evaluation stack. Floating-point values are converted to F type when loaded onto the evaluation stack.

2.5.31.5 The address specified by *addr* shall be to a location with the natural alignment of <*type*>. The results of all CIL instructions that return addresses (e.g., ldloca and ldarga) are safely aligned. For data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms.

2.5.32 ldloc – load local variable onto the stack

2.5.32.1

Binary	Text Format	Description
FE 0C<*uint16*>	ldloc *indx*	Load local variable of index *indx* onto stack.
11 <*uint8*>	ldloc_s *indx*	Load local variable of index *indx* onto stack, short form.
06	ldloc_0	Load local variable 0 onto stack.
07	ldloc_1	Load local variable 1 onto stack.
08	ldloc_2	Load local variable 2 onto stack.

09	ldloc_3	Load local variable 3 onto stack.

2.5.32.2 `… -> …, value`

2.5.32.3 The ldloc *indx* instruction pushes the contents of the local variable number *indx* onto the evaluation stack, where local variables are numbered 0 onwards. The ldloc.0, ldloc.1, ldloc.2, and ldloc.3 instructions provide an efficient encoding for accessing the first 4 local variables. The ldloc.s instruction provides an efficient encoding for accessing local variables 4–255.

2.5.32.4 The type of the value is the same as the type of the local variable, which is specified in the method header.

2.5.32.5 Local variables that are smaller than 4 bytes are expanded to type int32 when they are loaded onto the stack. Floating-point values are expanded to their native size (type F).

2.5.33 ldloca – load local variable address

2.5.33.1

Binary	Text Format	Description
FE 0D <*unsigned int16*>	ldloca *indx*	Load address of local variable with index *indx.*
12 <*unsigned int8*>	ldloca_s *indx*	Load address of local variable with index *indx, short form.*

2.5.33.2 `… -> …, address`

2.5.33.3 The ldloca instruction pushes the address of the local variable number *indx* onto the stack, where local variables are numbered 0 onwards. The value pushed on the stack is already aligned correctly for use with instructions like ldind and stind. The result is a pointer (type intptr). The ldloca.s instruction provides an efficient encoding for use with the local variables 0–255. (Local variables that are the subject of ldloca shall be aligned as described in the ldind instruction, since the address obtained by ldloca can be used as an argument to ldind.)

2.5.34 ldnull – load a null pointer

2.5.34.1

Binary	Text Format	Description
14	ldnull	Push a null reference on the stack.

2.5.34.2 `… -> …, null value`

2.5.34.3 The ldnull pushes a null pointer (type intptr) on the stack.

2.5.34.4 TODO: type of the null pointer

2.5.35 ldobj – copy a value from an address to the stack

2.5.35.1

Binary	Text Format	Description
71 <*T*>	ldobj *type*	Copy the value stored at address *src* to the stack.

2.5.35.2 `…, src -> …, val`

2.5.35.3 The ldobj instruction dereferences src and copies its value to the stack. *type* is a reference to the struct or union. *src* is an intptr.

2.5.36 ldproc – load procedure pointer

2.5.36.1

Binary	Text Format	Description
FE 06 <*T*>	ldproc *proc*	Push a pointer to a procedure referenced by *proc,* on the stack.

2.5.36.2 `… -> …, ftn`

2.5.36.3 The ldproc instruction pushes a pointer (type intptr) to the native code implementing the procedure described by *proc* (a descriptor) onto the stack. The value pushed can be called using the calli instruction.

2.5.36.4 A method pointer can be passed to external code (e.g., as a callback routine).

2.5.37 ldstr – load a literal string

2.5.37.1

Binary	Text Format	Description
72 <*T*>	ldstr *string*	Push the address of the literal *string*.

2.5.37.2 `…, -> …, address`

2.5.37.3 The ldstr instruction pushes the address (intptr) representing the zero-terminated string literal as a pointer to an open array of char.

2.5.37.4 The result of two ldstr instructions referring to the same string literal return precisely the same address.

2.5.37.5 TODO: the equivalent for binary strings, but this time with byte count

2.5.38 ldvar – load module variable

2.5.38.1

Binary	Text Format	Description
7E <*T*>	ldvar *var*	Push the value of *var* on the stack.

2.5.38.2 `…, -> …, value`

2.5.38.3 The ldvar instruction pushes the value of a module variable on the stack. *var* is a reference to the variable. The return type is that associated with *var*.

2.5.39 ldvara – load module variable address

2.5.39.1

Binary	Text Format	Description
7F <*T*>	ldvara *var*	Push the address of the module variable, *var,* on the stack.

2.5.39.2 `…, -> …, address`

2.5.39.3 The ldvara instruction pushes the address (a intptr) referring to a module variable.

2.5.40 mul – multiply values

2.5.40.1

Binary	Text Format	Description
5A	mul	Multiply values.

2.5.40.2 `…, value1, value2 -> …, result`

2.5.40.3 The mul instruction multiplies *value1* by *value2* and pushes the result on the stack. Integral operations silently truncate the upper bits on overflow. For floating-point types, 0 × infinity = NaN.

2.5.40.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.8.4.1.

2.5.41 neg – negate

2.5.41.1

Binary	Text Format	Description
65	neg	Negate *value*.

2.5.41.2 `…, value -> …, result`

2.5.41.3 The neg instruction negates *value* and pushes the result on top of the stack. The return type is the same as the operand type. Negation of integral values is standard twos-complement negation. In particular, negating the most negative number (which does not have a positive counterpart) yields the most negative number. Negating a floating-point number cannot overflow; negating NaN returns NaN.

2.5.41.4 The acceptable operand types and their corresponding result data types are encapsulated as follows:

2.5.41.4.1

Operand Type	int32	int64	intptr	F
Result Type	int32	int64	intptr	F

2.5.42 newarr – create a zero-based, one-dimensional array

2.5.42.1

Binary	Text Format	Description
8D <*T*>	newarr *etype*	Create a new array with elements of type *etype*.

2.5.42.2 `…, numElems -> …, pointer to array`

2.5.42.3 The newarr instruction pushes a pointer to a new zero-based, one-dimensional array whose elements are of type *etype*. *numElems* (of type intptr or int32) specifies the number of elements in the array. Valid array indexes are 0 ≤ index < *numElems*. The elements of an array can be any type. Elements of the array are initialized to the default value of the appropriate type.

2.5.43 newvla – create a variable-length array on stack

2.5.43.1

Binary		

	Text Format	Description
EE 0F <*T*>	newvla *etype*	Create a new VLA with elements of type *etype*.

2.5.43.2 `…, numElems -> …, pointer to array`

2.5.43.3 The newvla instruction pushes a pointer to a new zero-based, one-dimensional array whose elements are of type *etype*. *numElems* (of type intptr or int32) specifies the number of elements in the array. In contrast to newarr, newvla tries to allocated the array on the stack, and the memory is automatically disposed when the function finishes. Valid array indexes are 0 ≤ index < *numElems*. The elements of an array can be any type. Elements of the array are initialized to the default value of the appropriate type.

2.5.43.4 TODO: how can we

2.5.44 newobj – create a new object

2.5.44.1

Binary	Text Format	Description
73 <*T*>	newobj *type*	Allocate an instance of type (struct or union).

2.5.44.2 `…, -> …, pointer to instance`

2.5.44.3 The newobj instruction creates a new instance of *type* and initializes all the fields in the new instance to 0 (of the proper type) or null as appropriate.

2.5.45 not – bitwise complement

2.5.45.1

Binary	Text Format	Description
66	not	Bitwise complement.

2.5.45.2 `…, value -> …, result`

2.5.45.3 The *not* instruction computes the bitwise complement of the integer value on top of the stack and leaves the result on top of the stack. The return type is the same as the operand type.

2.5.45.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.9.3.1.

2.5.46 or – bitwise OR

2.5.46.1

Binary	Text Format	Description
60	or	Bitwise or of two integer values, returns an integer.

2.5.46.2 `…, value1, value2 -> …, result`

2.5.46.3 The or instruction computes the bitwise or of the top two values on the stack and leaves the result on the stack.

2.5.46.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.9.3.1.

2.5.47 rem – compute remainder

2.5.47.1

Binary	Text Format	Description
5D	rem	Remainder when dividing one value by another.

2.5.47.2 `…, value1, value2 -> …, result`

2.5.47.3 The rem instruction divides *value1* by *value2* and pushes the remainder *result* on the stack.

2.5.47.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.8.4.1.

2.5.47.5 **For integer operands**

2.5.47.5.1 result = value1 rem value2 satisfies the following conditions:

2.5.47.5.1.1 result = value1 – value2×(value1 div value2), and

2.5.47.5.1.2 0 ≤ |result| < |value2|, and

2.5.47.5.1.3 sign(result) = sign(value1),

2.5.47.5.2 where div is the division instruction, which truncates towards zero.

2.5.47.6 **For floating-point operands**

2.5.47.6.1 rem is defined similarly as for integer operands, except that, if *value2* is zero or *value1* is infinity, *result* is NaN. If *value2* is infinity, *result* is *value1*. This definition is different from the one for floating-point remainder in the IEC 60559:1989 Standard. That Standard specifies that *value1* div *value2* is the nearest integer instead of truncating towards zero.

2.5.48 rem_un – compute int eger remainder, unsi gned

2.5.48.1

Binary	Text Format	Description
5E	rem_un	Remainder when dividing one unsigned value by another.

2.5.48.2 `…, value1, value2 -> …, result`

2.5.48.3 The rem_un instruction divides value1 by value2 and pushes the remainder result on the stack. (rem_un treats its arguments as unsigned integers, while rem treats them as signed integers.)

2.5.48.4 result = value1 rem_un value2 satisfies the following conditions:

2.5.48.4.1 result = value1 – value2×(value1 div_un value2), and

2.5.48.4.2 0 ≤ result < value2,

2.5.48.5 where div_un is the unsigned division instruction. rem_un is unspecified for floating-point numbers.

2.5.48.6 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.8.4.1.

2.5.49 shl – shift integer left

2.5.49.1

Binary	Text Format	Description
62	shl	Shift an integer left (shifting in zeros), return an integer.

2.5.49.2 `…, value, shiftAmount -> …, result`

2.5.49.3 The shl instruction shifts *value* (int32, int64 or intptr) left by the number of bits specified by *shiftAmount*. *shiftAmount* is of type int32 or intptr. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*.

2.5.49.4 See the following table for details of which operand types are allowed, and their corresponding result type.

2.5.49.4.1

		Shift-By				
int32	**int64**	**intptr**	**F**			
To Be Shifted	**int32**	int32	x	int32	x	
	int64	int64	x	int64	x	
	intptr	intptr	x	intptr	x	
	F	x	x	x	x	

2.5.50 shr – shift integer right

2.5.50.1

Binary	Text Format	Description
63	shr	Shift an integer right (shift in sign), return an integer.

2.5.50.2 `…, value, shiftAmount -> …, result`

2.5.50.3 The shr instruction shifts *value* (int32, int64 or intptr) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type int32 or intptr. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. shr replicates the high order bit on each shift, preserving the sign of the original value in *result*.

2.5.50.4 See table 2.5.49.4.1 for details of which operand types are allowed, and their corresponding result type.

2.5.51 shr_un – shift integer right, unsigned

2.5.51.1

Binary	Text Format	Description
64	shr_un	Shift an integer right (shift in zero), return an integer.

2.5.51.2 `…, value, shiftAmount -> …, result`

2.5.51.3 The shr.un instruction shifts *value* (int32, int 64 or intptr) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type int32 or intptr. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. shr.un inserts a zero bit on each shift.

2.5.51.4 See table 2.5.49.4.1 for details of which operand types are allowed, and their corresponding result type.

2.5.52 sub – subtract numeric values

2.5.52.1

Binary	Text Format	Description
59	sub	Subtract value2 from value1, returning a new value.

2.5.52.2 `…, value1, value2 -> …, result`

2.5.52.3 The sub instruction subtracts *value2* from *value1* (value1 - value2, i.e. right side was pushed last) and pushes the result on the stack. Overflow is not detected for the integral operations; for floating-point operands, sub returns +inf on positive overflow, -inf on negative overflow, and zero on floating-point underflow.

2.5.52.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.8.4.1.

2.5.53 xor – bitwise XOR

2.5.53.1

Binary	Text Format	Description
61	xor	Bitwise xor of integer values, returns an integer.

2.5.53.2 `..., value1, value2 -> ..., result`

2.5.53.3 The xor instruction computes the bitwise xor of *value1* and *value2*and leaves the result on the stack.

2.5.53.4 The acceptable operand types and their corresponding result data type are encapsulated in 2.5.9.3.1.

2.6 Constants

2.6.1 TODO

2.6.2 Only single literals, no expressions; or maybe calculated at runtime and write-once variable

2.6.3 Maybe an initializer expression independent of the module begin.

2.7 Variables

2.7.1 Variable declarations introduce module variables by defining an identifier and a data type for them.

2.7.2 Syntax:

2.7.2.1
```
VariableDeclaration = IdentList ':' NamedType
```

2.8 Procedures

2.8.1 A procedure declaration consists of a procedure heading and a procedure body. The heading specifies the procedure identifier and the formal parameters. The body contains local variable declarations and statements. The procedure identifier must be repeated at the end of the procedure declaration.

2.8.2 There are two kinds of procedures: proper procedures and function procedures. The latter are activated as a constituent of an expression and yield a result that is an operand of the expression. Proper procedures are activated by a procedure call. A procedure is a function procedure if its formal parameters specify a result type. Each control path of a function procedure must return a value.

2.8.3 All variables declared within a procedure body are local to the procedure. The variable names are optional. If present, they must be unique within the local variable list. Local variables are numbered from left to right, starting from zero. Local variables can be referenced by their name or by their number.

2.8.4 The call of a procedure within its declaration implies recursive activation.

2.8.5 A procedure can be declared EXTERN to indicate that it is implemented in an external library with C calling conventions. EXTERN can be followed by the name the procedure has in that external library, otherwise the same name as in the identdef is assumed.

2.8.6 A procedure can be an alias of another procedure. The other procedure is referenced with a qualident. If the other procedure is in another module, it must be exported.

2.8.7 Syntax of textual representation:

2.8.7.1
```
ProcedureDeclaration = ProcedureHeading [';'] ( ProcedureBody | EXTERN [ident] )
                       | ( PROCEDURE | PROC ) identdef '=' qualident

ProcedureHeading     = ( PROCEDURE | PROC ) identdef [ FormalParameters ]

ProcedureBody        = [VAR { LocalDeclaration [';'] }]
                       BEGIN StatementSequence END ident

LocalDeclaration     = [IdentList ':'] NamedType
```

2.8.8 Formal parameters

2.8.8.1 Formal parameters are identifiers declared in the formal parameter list of a procedure. They correspond to actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called.

2.8.8.2 The scope of a formal parameter extends from its declaration to the end of the procedure block in which it is declared. A function procedure without parameters must have an empty parameter list. The result type of a procedure cannot be an open array.

2.8.8.3 The formal parameter names are optional. If present, they must be unique within the formal parameter list. Formal parameters are numbered from left to right, starting from zero. Formal parameters can be referenced by their name or by their number.

2.8.8.4 Syntax of textual representation:

2.8.8.4.1
```
FormalParameters = '(' [ FPSection { ';' FPSection } [ ';' '..'] ] ')'
                   [ ':' ReturnType ]

ReturnType       = NamedType

FPSection        = [ident { [','] ident } ':'] NamedType
```

2.8.8.5 Procedures can have a variable number of parameters, which is indicated by `'..'` (ellipsis).

2.8.8.6 TODO: va_list and access instruction

2.9 Statements

2.9.1 TODO: pcall and raise

2.9.2 Syntax of textual representation:

2.9.2.1
```
StatementSequence = { Statement }
Statement = ('call' | 'calli') qualident | 'disp' | RepeatUntil
| 'exit' | 'goto' ident | IfThenElse
| 'label' ident | 'line' unsigned | Loop | 'pop' | 'ret'
| ('starg'|'starg_s') (unsigned|ident)
| 'stelem' qualident | 'stelem_i1' | 'stelem_i2' | 'stelem_i4' | 'stelem_i8'
| 'stelem_r4' | 'stelem_r8' | 'stelem_ip'
| 'stfld' qualident '.' ident
| 'stind_i1' | 'stind_i2' | 'stind_i4' | 'stind_i8' | 'stind_r4' | 'stind_r8' | 'stind_ip'
| ('stloc' |'stloc_s') (unsigned|ident) | 'stloc_0' | 'stloc_1' | 'stloc_2' | 'stloc_3'
| 'stobj' qualident | 'stvar' qualident '.' ident | Switch | WhileDo
```

2.9.3 call – call a function

2.9.3.1 See 2.5.10.

2.9.3.2 Call pop if the function returns a value.

2.9.4 calli – indirect function call

2.9.4.1 See 2.5.11.

2.9.4.2 Call pop if the function returns a value.

2.9.5 disp

2.9.5.1

Binary	Text Format	Description
EE 0C	disp	Free memory previously allocated by Newarr or Newobj.

2.9.5.2 `…,addr-> …,`

2.9.5.3 TODO: type to dispose

2.9.6 exit

2.9.6.1

Binary	Text Format	Description
EE 09	exit	Breaks the enclosing Loop statement.

2.9.6.2 `…,-> …,`

2.9.6.3 TODO

2.9.7 goto

2.9.7.1

Binary	Text Format	Description
EE 0B	goto *name*	Jump to a label declared in the same scope.

2.9.7.2 …,-> …,

2.9.7.3 TODO

2.9.8 if then else

2.9.8.1

Binary	Text Format	Description
EE 01	if	Begin of an IF statement and the condition expression.
EE 02	then	End of the If condition expression and begin of the If statement sequence
EE 03	else	End of the If statement sequence and begin of the Else statement sequence.
EE 04	end	End of the If or Else statement sequence.

2.9.8.2 …,-> …,

2.9.8.3 Syntax of textual representation:

2.9.8.3.1
```
IfThenElse = 'IF' Expression 'THEN' StatementSequence
[ 'ELSE' StatementSequence ] 'END'
```

2.9.8.4 TODO

2.9.9 label

2.9.9.1

Binary	Text Format	Description
EE 0A	label *name*	Specifies a label which can be referenced in Goto statements.

2.9.9.2 …,-> …,

2.9.9.3 TODO

2.9.10 line

2.9.10.1

Binary	Text Format	Description
EE 05 <*uint32*>	line *num*	The line number *num* in the source file of the module.

2.9.10.2 …,-> …,

2.9.10.3 TODO

2.9.11 loop

2.9.11.1

Binary	Text Format	Description
EE 08	loop	Begin of Loop statement sequence.
EE 04	end	End of the Loop statement sequence.

2.9.11.2 …,-> …,

2.9.11.3 Syntax of textual representation:

2.9.11.3.1
```
Loop = 'LOOP' StatementSequence 'END'
```

2.9.11.4 TODO

2.9.12 pop – remove the top element of the stack

2.9.12.1

Binary	Text Format	Description
26	pop	Pop *value* from the stack.

2.9.12.2 …, value -> …

2.9.12.3 The pop instruction removes the top element from the stack.

2.9.13 repeat until

2.9.13.1

Binary		

	Text Format	Description
EE 10	repeat	Begin of Repeat statement and begin of the statement sequence.
EE 11	until	End of the statement sequence and begin of the Until condition expression.
EE 04	end	End of the Until condition expression.

2.9.13.2 `…,-> …,`

2.9.13.3 Syntax of textual representation:

2.9.13.3.1
```
RepeatUntil = 'REPEAT' StatementSequence 'UNTIL' Expression 'END'
```

2.9.13.4 TODO

2.9.14 ret – return from method

2.9.14.1

Binary	Text Format	Description
2A	ret	Return from method, possibly with a value.

2.9.14.2 `…, value -> …`

2.9.14.3 Return from the current method. The return type, if any, of the current method determines the type of value to be fetched from the top of the stack and copied onto the stack of the method that called the current method. The evaluation stack for the current method shall be empty except for the value to be returned.

2.9.15 starg – store a value in an argument slot

2.9.15.1

Binary	Text Format	Description
FE 0B <*uint16*>	starg *num*	Store *value* to the argument numbered *num.*
10 <*uint8*>	starg_s *num*	Store *value* to the argument numbered *num*, short form.

2.9.15.2 `…, value -> …`

2.9.15.3 The starg *num* instruction pops a value from the stack and places it in argument slot *num*. The type of the value shall match the type of the argument, as specified in the current method's signature. The starg.s instruction provides an efficient encoding for use with the first 256 arguments.

2.9.15.4 For procedures that take a variable argument list, the starg instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

2.9.15.5 Storing into arguments that hold an integer value smaller than 4 bytes long truncates the value as it moves from the stack to the argument. Floating-point values are rounded from their native size (type F) to the size associated with the argument.

2.9.16 stelem – store element to array

2.9.16.1

Binary	Text Format	Description
A4 <*T*>	stelem *etype*	Replace array element at *index* with the *value* on the stack

2.9.16.2

9C	stelem_i1	Replace *array* element at *index* with the int8 *value* on the stack.
9D	stelem_i2	Replace *array* element at *index* with the int16 *value* on the stack.
9E	stelem_i4	Replace *array* element at *index* with the int32 *value* on the stack.
9F	stelem_i8	Replace *array* element at *index* with the int64 *value* on the stack.
A0	stelem_r4	Replace *array* element at *index* with the float32 *value* on the stack.
A1	stelem_r8	Replace *array* element at *index* with the float64 *value* on the stack.
9B	stelem_ip	Replace *array* element at *index* with the intptr *value* on the stack.

2.9.16.3 `…, pointer to array, index, value, -> …`

2.9.16.4 The stelem instruction replaces the value of the element with zero-based index *index* (of type intptr or int32) in the one-dimensional array *array* (pointed to by a pointer), with *value*. The value has the type specified by *etype* or the instruction suffix.

2.9.17 stfld – store into a field of an object

2.9.17.1

Binary	Text Format	Description
7D <*T*>	stfld *field*	Replace the *value* of *field* of the object *obj* with *value.*

2.9.17.2 `…, pointer to obj, value -> …,`

2.9.17.3 The stfld instruction replaces the value of a field of an *obj* via a pointer (type intptr) with *value*. *field* is a field member reference. stfld pops the value and the pointer off the stack and updates the object.

2.9.18 stind – store value indirect from stack

2.9.18.1

Binary	Text Format	Description
52	stind_i1	Store value of type int8 into memory at address
53	stind_i2	Store value of type int16 into memory at address
54	stind_i4	Store value of type int32 into memory at address
55	stind_i8	Store value of type int64 into memory at address
56	stind_r4	Store value of type float32 into memory at address
57	stind_r8	Store value of type float64 into memory at address
DF	stind_ip	Store value of type intptr into memory at address

2.9.18.2 `…, addr, val -> …`

2.9.18.3 The stind instruction stores value *val* at address *addr* (an intptr type). The address specified by *addr* shall be aligned to the natural size of *val*. The results of all instructions that return addresses (e.g., ldloca and ldarga) are safely aligned. For data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms.

2.9.19 stloc – pop value from stack to local variable

2.9.19.1

Binary	Text Format	Description
FE 0E <*uint16*>	stloc *indx*	Pop a value from stack into local variable *indx.*
13 <*uint8*>	stloc_s *indx*	Pop a value from stack into local variable *indx*, short form.
0A	stloc_0	Pop a value from stack into local variable 0.
0B	stloc_1	Pop a value from stack into local variable 1.
0C	stloc_2	Pop a value from stack into local variable 2.
0D	stloc_3	Pop a value from stack into local variable 3.

2.9.19.2 `…, value -> …`

2.9.19.3 The stloc *indx* instruction pops the top value off the evaluation stack and moves it into local variable number *indx*, where local variables are numbered 0 onwards. The type of *value* shall match the type of the local variable as specified in the current function's locals signature. The stloc.0, stloc.1, stloc.2, and stloc.3 instructions provide an efficient encoding for the first 4 local variables; the stloc.s instruction provides an efficient encoding for local variables 4–255.

2.9.19.4 Storing into locals that hold an integer value smaller than 4 bytes long truncates the value as it moves from the stack to the local variable. Floating-point values are rounded from their native size (type F) to the size associated with the argument.

2.9.20 stobj – store a value at an address

2.9.20.1

Binary	Text Format	Description

81 <*T*>	stobj *type*	Store a value of type *type* at an address.

2.9.20.2 …, dest, src -> …,

2.9.20.3 The stobj instruction copies the value *src* of type *type* to the address *dest*.

2.9.21 stvar – store a module variable

2.9.21.1

Format	Assembly Format	Description
80 <*T*>	stvar *var*	Replace the value of *var* with *val*.

2.9.21.2 …, val -> …,

2.9.21.3 The stvar instruction replaces the value of a module variable with a value from the stack.

2.9.22 switch

2.9.22.1

Binary	Text Format	Description
EE 0D	switch	Begin of a Switch statement and switch expression.
EE 0E	case	Begin of a Case integer list and end of switch expression.
EE 02	then	End of the Case integer list and begin of a Then statement sequence.
EE 03	else	End of a Then statement sequence an begin of the Else statement sequence.
EE 04	end	End of a Then or the Else statement sequence.

2.9.22.2 …,-> …,

2.9.22.3 Syntax of textual representation:

2.9.22.3.1
```
Switch = 'SWITCH' Expression { 'CASE' integer { [','] integer }
'THEN' StatementSequence }
[ 'ELSE' StatementSequence ] 'END'
```

2.9.22.4 TODO

2.9.23 while do

2.9.23.1

Binary	Text Format	Description
EE 06	while	Begin of While Do statement and begin of the condition expression.
EE 07	do	End of the While condition expression and begin of the Do statement sequence
EE 04	end	End of the Do statement sequence.

2.9.23.2 …,-> …,

2.9.23.3 Syntax of textual representation:

2.9.23.3.1
```
WhileDo = 'WHILE' Expression 'DO' StatementSequence 'END'
```

2.9.23.4 TODO

2.10 Modules

2.10.1 A module is a collection of declarations of constants, types, variables, and procedures, together with a sequence of statements for the purpose of assigning initial values to the variables. A module is a compilation unit.

2.10.2 Syntax of textual representation:

2.10.2.1
```
module     = MODULE ident [ MetaParams ] [';']
               { ImportList | DeclarationSequence }
           [ BEGIN StatementSequence ] END ident ['.']

ImportList = IMPORT import { [','] import } [';']

import     = [ ident ':=' ] ImportPath ident [ MetaActuals ]

ImportPath = { ident '.' }
```

```
DeclarationSequence  = { TYPE { TypeDeclaration [';'] }
                     | VAR { VariableDeclaration [';'] }
                     | ProcedureDeclaration [';'] }
```

2.10.3 The import list specifies the names of the imported modules. If a module A is imported by a module M and A exports an identifier x, then x is referred to as A.x within M.

2.10.4 If A is imported as B := A, the object x must be referenced as B.x. This allows short alias names in qualified identifiers.

2.10.5 The import can refer to a module by means of a module name optionally prefixed with an import path. There is no requirement that the import path actually exists in the file system, or that the source files corresponding to an import path are in the same file system directory. It is up to the compiler or linker how source files are mapped to import paths. An imported module with no import path is first looked up in the import path of the importing module.

2.10.6 A module must not import itself (neither directly nor indirectly).

2.10.7 Identifiers that are to be exported (i.e. that are to be visible in client modules) must be marked by an export mark in their declaration.

2.10.8 The statement sequence following the symbol BEGIN is executed when the module is loaded, which is done after the imported modules have been loaded. It follows that cyclic import of modules is illegal.

2.10.9 Generic Modules

2.10.9.1 Modules can be made generic by adding formal meta parameters. Meta parameters represent types or constants; the latter include procedure names. Meta parameters default to types, but can be explicitly prefixed with the TYPE reserved word; the CONST prefix designates a constant meta parameter. A CONST meta parameter can be constrained with a named type, in which case the actual meta parameter must correspond to this type; the correspondence is established when the generic module is instantiated; the type of the actual meta parameter must be compatible with the constraint type.

2.10.9.2 Generic modules can be instantiated with different sets of meta actuals which enables the design of reusable algorithms and data structures. The instantiation of a generic module occurs when importing it. A generic module can be instantiated more than once in the same module with different actual meta parameters.

2.10.9.3 Syntax of textual representation:

2.10.9.3.1

```
MetaParams       = '(' MetaSection { [';'] MetaSection } ')'

MetaSection      = CONST ident { [','] ident } ':' NamedType
                   | [ TYPE ] ident { [','] ident }

TypeConstraint   = NamedType | ProcedureType

MetaActuals      = '(' ConstExpression { [','] ConstExpression } ')'
```

2.10.9.4 Meta parameters can be used within the generic module like normal types or constants. If no type constraint is present, the types and constants can be used wherever no information about the actual type is required; otherwise the type constraint determines the permitted operations.

2.10.9.5 **NOTE** Generics in MIL are useful to transport generic implementations in symbol files for later instantiations (instead of the source code).

2.11 TODO: type checking rules, structral compatibility, packing

3 Complete syntax of the textual representation

3.1 ↳

```
ident  = ( letter | '_' | '$' ) { letter | digit | '_' | '$' }

letter = 'A' ... 'Z' | 'a' ... 'z'

digit  = '0' ... '9'
```

3.2 ↳

```
unsigned  = (digit {digit} ['O'|'B' | digit {hexDigit} 'H')
integer = ['+' | '-'] unsigned

real     = ['+' | '-'] digit {digit} '.' {digit} [Exponent]

Exponent = 'E' ['+' | '-'] digit {digit}

hexDigit = digit | 'A' ... 'F'

digit    = '0' ... '9'
```

3.3 ↳

```
character = digit {hexDigit} ('X' | 'x')
```

3.4 ↳

```
string = ''' {character} ''' | '"' {character} '"'
```

3.5 ↳

```
hexstring = '#' {hexDigit} '#'
```

3.6 ↳

```
qualident = [ident '.'] ident
identdef = ident ['*']
```

3.7 ↳

```
TypeDeclaration = identdef '=' type

type            = NamedType | ArrayType | StructUnionType | PointerType | ProcedureType
```

3.8 ↳

```
NamedType = qualident
```

3.9 ↳

```
ArrayType = ARRAY [ length ] OF NamedType | '[' [ length ] ']' NamedType
length = unsigned
```

3.10 ↳

```
StructUnionType = (STRUCT | UNION) { FieldList [';'] } END
FieldList = IdentList ':' NamedType
IdentList = identdef { [','] identdef }
```

3.11 ↳

```
PointerType = ( POINTER TO | '^' ) NamedType
```

3.12 ↳

```
ProcedureType = (PROCEDURE|PROC) [FormalParameters]
```

3.13 ↳

```
Expression = { ExpInstr }
ExpInstr = 'add' | 'and' | ('call' | 'calli') qualident | 'castptr' qualident
| 'ceq' | 'cgt' | 'cgt_un' | 'clt' | 'clt_un'
| 'conv_i1' | 'conv_i2' | 'conv_i4' | 'conv_i8' | 'conv_r4' | 'conv_r8'
| 'conv_u1' | 'conv_u2' | 'conv_u4' | 'conv_u8' | 'conv_ip'
| 'div' | 'div_un' | 'dup' | 'initobj' qualident | ('ldarg'|'ldarg_s') (unsigned|ident)
| 'ldarg_0' | 'ldarg_1' | 'ldarg_2' | 'ldarg_3'
| ('ldarga'|'ldarga_s') (unsigned|ident)
| ('ldc_i4'|'ldc_i8'|'ldc_i4_s') integer | ('ldc_r4'|'ldc_r8') real
| 'ldc_i4_0' | 'ldc_i4_1' | 'ldc_i4_2' | 'ldc_i4_3' | 'ldc_i4_4' | 'ldc_i4_5'
| 'ldc_i4_6' | 'ldc_i4_7' | 'ldc_i4_8' | 'ldc_i4_m1' | 'ldc_obj' constructor
| ('ldelem'|'ldelema') qualident | 'ldelem_i1' | 'ldelem_i2'
| 'ldelem_i4' | 'ldelem_i8' | 'ldelem_u1' | 'ldelem_u2'
| 'ldelem_u4' | 'ldelem_u8' | 'ldelem_r4' | 'ldelem_r8' | 'ldelem_ip'
| ('ldfld'|'ldflda') qualident '.' ident | 'ldftn' qualident
| 'ldind_i1' | 'ldind_i2' | 'ldind_i4' | 'ldind_i8' | 'ldind_u1' | 'ldind_u2'
| 'ldind_u4' | 'ldind_r4' | 'ldind_u8' | 'ldind_r8' | 'ldind_ip'
| ('ldloc'|'ldloc_s'|'ldloca'|'ldloca_s')(unsigned|ident)
| 'ldloc_0' | 'ldloc_1' | 'ldloc_2' | 'ldloc_3' | 'ldnull'
| 'ldobj' qualident | 'ldproc' qualident | 'ldstr' string
| ('ldvar'|'ldvara') qualident | 'mul' | 'neg'
| ('newarr'|'newvla'|'newobj') qualident
| 'not' | 'or' | 'rem' | 'rem_un' | 'shl' | 'shr' | 'shr_un' | 'sub' | 'xor'
```

3.14 ↳

```
ConstExpression = qualident | integer | real | string | hexstring
```

3.15 ↳

```
constructor = NamedType component_list
component_list = '{' [ component {[','] component} ] '}'
component = [ident '='] ( ConstExpression | component_list )
```

3.16 ↳

```
VariableDeclaration = IdentList ':' NamedType
```

3.17 ↳

```
ProcedureDeclaration = ProcedureHeading [';'] ( ProcedureBody | EXTERN [ident] )
                       | ( PROCEDURE | PROC ) identdef '=' qualident

ProcedureHeading     = ( PROCEDURE | PROC ) identdef [ FormalParameters ]

ProcedureBody        = [VAR { LocalDeclaration [';'] }]
                       BEGIN StatementSequence END ident

LocalDeclaration     = [IdentList ':'] NamedType
```

3.18 ↳

```
FormalParameters = '(' [ FPSection { ';' FPSection } [ ';' '..'] ] ')'
                   [ ':' ReturnType ]

ReturnType       = NamedType

FPSection        = [ident { [','] ident } ':'] NamedType
```

3.19 ↳

```
StatementSequence = { Statement }
Statement = ('call' | 'calli') qualident | 'disp' | RepeatUntil
| 'exit' | 'goto' ident | IfThenElse
| 'label' ident | 'line' unsigned | Loop | 'pop' | 'ret'
| ('starg'|'starg_s') (unsigned|ident)
| 'stelem' qualident | 'stelem_i1' | 'stelem_i2' | 'stelem_i4' | 'stelem_i8'
| 'stelem_r4' | 'stelem_r8' | 'stelem_ip'
| 'stfld' qualident '.' ident
| 'stind_i1' | 'stind_i2' | 'stind_i4' | 'stind_i8' | 'stind_r4' | 'stind_r8' | 'stind_ip'
| ('stloc' |'stloc_s') (unsigned|ident) | 'stloc_0' | 'stloc_1' | 'stloc_2' | 'stloc_3'
| 'stobj' qualident | 'stvar' qualident '.' ident | Switch | WhileDo
```

3.20 ↳

```
module     = MODULE ident [ MetaParams ] [';']
               { ImportList | DeclarationSequence }
           [ BEGIN StatementSequence ] END ident ['.']

ImportList = IMPORT import { [','] import } [';']

import     = [ ident ':=' ] ImportPath ident [ MetaActuals ]

ImportPath = { ident '.' }

DeclarationSequence  = { TYPE { TypeDeclaration [';'] }

                     | VAR { VariableDeclaration [';'] }

                     | ProcedureDeclaration [';'] }
```

3.21 ↳

```
MetaParams       = '(' MetaSection { [';'] MetaSection } ')'

MetaSection      = CONST ident { [','] ident } ':' NamedType
                   | [ TYPE ] ident { [','] ident }

TypeConstraint   = NamedType | ProcedureType

MetaActuals      = '(' ConstExpression { [','] ConstExpression } ')'
```

3.22 ↳

```
IfThenElse = 'IF' Expression 'THEN' StatementSequence
[ 'ELSE' StatementSequence ] 'END'
```

3.23 ↳

```
Loop = 'LOOP' StatementSequence 'END'
```

3.24 ↳

```
RepeatUntil = 'REPEAT' StatementSequence 'UNTIL' Expression 'END'
```

3.25 ↳

```
Switch = 'SWITCH' Expression { 'CASE' integer { [','] integer }
'THEN' StatementSequence }
[ 'ELSE' StatementSequence ] 'END'
```

3.26 ↳

```
WhileDo = 'WHILE' Expression 'DO' StatementSequence 'END'
```

4 References

4.1 TODO