Mitten Specification

Oliver Katz

April 26, 2013

Contents

1 Introduction

1.1 How to Read this Specification

A leading character or a character at the beginning of a token is assumed to be at the left-hand side for right-to-left alphabets. A following character or a character at the end of a token is assumed to be on the right-hand side.

An ellipsis means that ambiguous content should take its place.

All ”punctuation” tokens (brackets, whitespace, end-of-line tokens) are reprogrammable so that Mitten may be easily typed by anyone on any device. Although the specification must show the token-ambiguous definition of all parsed constructs, the default syntax will be shown also for clarity.

All character bytes are written in hexadecimal according to the ASCII (8-bit) encoding.

The coding style used in the examples is not required, nor is it recommended. It is just my coding style.

For best results, tie your hands behind your back and close your eyes.

2 Paradigm

Everything is an object. Even things you don't want to be objects are objects. Mitten is objectified and rightly so, because everything is the object of its desires. Objects are made up of data, methods, and green cheese. The data within objects are objects themselves, so where does it end? It is OK. The snake is not eating its tail; it is safe for another day. There are atomic classes. They are emulated within the compiler. Atomic classes can take the types of integral values ranging from 8-bit to 32-bit, floating-point types either 32-bit or 64-bit, memory address types either 32-bit or 64-bit (depending on architecture), a class "inheriting" the address class for function pointers, a class class (classes themselves are objects), and a highly important green cheese class.

Memory addresses can point to data of many different classes, so in order to solve this problem, the ambiguous class is used. It can be anything; its type is lazily evaluated until it is used. An ambiguous object can take on any type, but must keep the first type assigned to it for the rest of its lifetime. It is a cruel, cruel fate. Take pity upon on the ambiguous type. It does not have a class itself, as it is only a compiler construct. You should not try to objectify the ambiguous type.

3 Lexing

Lexical analysis for Mitten searches the file for specific tokens, looking for diamonds in the rough. Everything in-between are symbols deliminated by whitespace (I do not mean to play the race card, see 3.4). Whitespace does not separate tokens within quoted strings (see 3.5.7) or characters see 3.5.6).

3.1 Input Files

3.1.1 Extension

The recomended extension to use for Mitten source files is `*.n`. Why not `*.m`? Because it is taken by those evil people over at Objective-C (see **??**). The idea being that you could have a hypothetical file called `mitte.n`. [1]

3.1.2 Encoding

While 8-bit ASCII and UTF-8 are the primary encodings, UTF-16 and UTF-32 can both be converted to the subset of UTF-8 (UTF-16 or UTF-32 specific characters are illegal).

3.2 Comments

If there are two forward slashes anywhere except in a string (see 3.5.7), the rest of the line is ignored (until a newline or a cartridge return, see 3.5.7).

[1]The people over at Objective-C are not completely evil. Completely, no. Partially, yes.

```
... // ...
```

Block comments are started with a forward slash followed by an asterisk and ended by the reverse. The contents are ignored, as well as the forward slashes and asterisks wrapping the contents.

```
/* ... */
```

These are not reprogrammable, as most tokens are. These are constant across all files and compiler configurations. Otherwise, there would be no standard way to see where comments are. It would be easy to write confusing code where default comment symbols wiped hard drives and the apocalypse would come. Comments wiping hard drives is well known to be one of the four horsemen. It would also be difficult to extract documentation from comments with third-party programs.

If the first instance of a `*/` in a file is before any instance of `/*`, it is illegal and the sheriff will shoot you dead in the heart. If the last instance of `/*` is not completed with an instance of `*/`, it is illegal. If either boundary of a block comment is within a quoted string (see 3.5.7) or character (see 3.5.6) it is ignored (there is no comment). Feel free to comment amongst yourselves.

3.3 Token Configuration Files

Token configuration files are passed to the compiler to tell it what tokens to search for. The syntax of the token configuration file is very simple. There are lines separated by either newline characters (0A), cartridge returns (0D), or semicolons (';'). The lines begin with a symbol describing what token type should be programmed with the line, followed with a left curly bracket (''). A sequence of string tokens separated by commas (','), followed by a right curly bracket ('') finishes the line. Whitespace around these is ignored, although it deliminates tokens much like the default syntax. The contents of the strings are all the possible values for the token type. Symbols and values are dynamic, and thus cannot be reprogrammed.

The purpose of token configuration files is to make typing easier on keyboards/devices/whatevers where typing the default tokens is difficult to the point where reprogramming these tokens is the only way that programming is possible. It is not to make Mitten into your own custom programming language. The structure of the language and most of the syntax does not change, only the tokens. If you find that certain symbols are difficult to type, then use variable names to alias them.

It is recommended that, when releasing code to audiences outside of your keyboard/device/whatever, that you use the default tokens, instead of packaging the source code with a token configuration file for compilation, even though this is possible and you are, of course, more than free to do whatever you wish.

3.3.1 Token Symbol List

Token Type Symbol	Default Values	Description
`Whitespace`	`" ", "\t", "\n", "\r"`	Ignored deliminators
`BoundaryBeginExpression`	`"("`	Begins an expression
`BoundaryEndExpression`	`")"`	Ends an expression
`BoundaryBeginComplex`	`"["`	Begins a complex value
`BoundaryEndComplex`	`"]"`	Ends a complex value
`BoundaryBeginScope`	`"{"`	Begins a new scope
`BoundaryEndScope`	`"}"`	Ends an existing scope
`ArgumentSeparator`	`","`	Separates arguments
`OperatorAssign`	`"="`	Assignment operator
`OperatorAccess`	`"."`	Element access operator
`OperatorAdd`	`"+"`	Addition operator
`OperatorSubtract`	`"-"`	Subtraction operator
`OperatorMultiply`	`"*"`	Multiplication operator
`OperatorDivide`	`"/"`	Division operator
`OperatorModulate`	`"%"`	Modulation operator
`OperatorAndBitwise`	`"&"`	Bitwise and operator
`OperatorAndLogical`	`"&&"`	Logical and operator
`OperatorOrBitwise`	`"|"`	Bitwise or operator
`OperatorOrLogical`	`"||"`	Logical or operator
`OperatorXOrBitwise`	^	Bitwise exclusive or operator
`OperatorNegateBitwise`	`"~"`	Bitwise negation operator
`OperatorNegateLogical`	`"!"`	Logical negation operator
`OperatorBitShiftLeft`	`"<<"`	Bit-shift left operator
`OperatorBitShiftRight`	`">>"`	Bit-shift right operator
`OperatorLessThan`	`"<"`	Less-than comparison
`OperatorLessThanOrEqualTo`	`"<="`	Less-than-or-equal-to comparison
`OperatorGreaterThan`	`">"`	Greater-than comparison
`OperatorGreaterThanOrEqualTo`	`">="`	Greater-than-or-equal-to comparison
`OperatorEqualTo`	`"=="`	Equal-to comparison
`OperatorNotEqualTo`	`"!="`	Not-equal-to comparison
`OperatorAssign`	`"="`	Assignment operator
`OperatorAddAssign`	`"+="`	Add and assign operator
`OperatorIncrement`	`"++"`	Increment-by-one operator
`OperatorSubtractAssign`	`"-="`	Subtract and assign operator
`OperatorDecrement`	`"--"`	Decrement-by-one operator
`OperatorMultiplyAssign`	`"*="`	Multiply and assign operator
`OperatorDivideAssign`	`"/="`	Divide and assign operator
`OperatorModulateAssign`	`"%="`	Modulate and assign operator
`OperatorAndBitwiseAssign`	`"&="`	Bitwise and and assign operator
`OperatorOrBitwiseAssign`	`"|="`	Bitwise or and assign operator
`OperatorXOrBitwiseAssign`	`"^="`	Bitwise exclusive-or and assign operator
`OperatorNegateBitwiseAssign`	`"~="`	Bitwise negate and assign operator
`OperatorBitShiftLeftAssign`	`"<<="`	Bit-shift left and assign operator
`OperatorBitShiftRightAssign`	`">>="`	Bit-shift right and assign operator
`EndOfLine`	`";"`	End-of-line token

3.4 Whitespace

Whitespace by default (see 3.3.1) may be a space (20), a horizontal tab (09), a newline (0A), or a cartridge return (0D).

3.5 Tokens

Many tokens are single characters or strings of characters. Some tokens are more complicated. Some tokens may span multiple lines.

The tokens fall into types (see 3.3.1). Additional types are `Value`, `Type`, and `Symbol`.

Value tokens must be either string tokens (see 3.5.7), character tokens (see 3.5.6), be numeric values (only containing characters within `"-.0123456789abcdefABCDEF"`), or be boolean values (see 3.5.2).

Type tokens can be found in the data type tables (see 3.5.1).

Symbols, aside from being symbols within the programming language, happen to only be tokens which do not fit into any of the other categories.

3.5.1 Data Types

The following `Type` tokens are all type keywords:

`any`	A compile-time construct for an ambiguous type
`bool`	8-bit boolean type
`byte`	8-bit integer type
`int`	32-bit integer type
`float`	32-bit floating type
`ref`	32/64-bit reference type
`long`	Double the size of a type (except ref)
`unsigned`	Make the type unsigned (except ref)
`const`	Make the type constant (pass-by-value only)
`void`	A null type
`class`	Used for class definition
`abstract`	Used for class definition
`private`	Used for class member definition
`protected`	Used for class member definition
`public`	Used for class member definition
`variadic`	Used for creating variadic methods

The any type can take the place of anything. Once a type is assigned to it in a specific situation, it must keep that type for the rest of its life. It cannot be accessed before it has a type assigned to it without casting (it will result in the null value for that class from `__new__()`).

The size of the reference type depends on whether the host architecture is 32 or 64 bit. The use of the ref type is not recommended; it is intended for bit compatibility and OS-specific functionality only.

There are also the pedantic types:

`b8`	8-bit boolean
`i8`	8-bit integer
`ui8`	8-bit unsigned integer
`i16`	16-bit integer
`ui16`	16-bit unsigned integer
`i32`	32-bit integer
`ui32`	32-bit unsigned integer
`f32`	32-bit float
`f64`	64-bit float

Classes act as types.

3.5.2 Binary Numbers

Binary numbers can be either true or false. A token `true`, `True`, or `TRUE` either denotes falsehood or I am being incredibly sarcastic. A token `false`, `False`, or `FALSE` denotes actual falsehood. They are value tokens.

3.5.3 Integral Numbers

Integers are at least one character in length. They can only contain numeric characters, although they can be prefixed with a minus sign (`-`) to denote negativity (this takes precedence over the binary operation for subtraction, see 4.3):

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

You can have leading zeros. They are value tokens.

3.5.4 Floating-point Numbers

Floating-point numbers are identical to integral numbers (see 3.5.3), except that they contain the floating point:

```
.
```

If the floating point is leading the token, all characters following are decimals. It may not lead the minus sign if one is present in the token. If the floating point is at the end of the token, the content of the token is assumed to be whole-number, but the token is still a floating-point token. They are value tokens.

3.5.5 Hexadecimal Numbers

Hexadecimal numbers have no floating-point, nor are they signed (see 3.5.3 and 3.5.4). They contain integral characters (see 3.5.3) as well as the alphabetical characters ranging from a to f:

```
a b c d e f
```

Hexadecimal numbers are case-insensitive and letters of different case are allowed within the same token. They are value tokens.

3.5.6 Characters

The following is a character token:

```
'...'
```

It can contain any single character with the exception of escape codes. Escape codes begin with a back-slash and may take up multiple characters. It can contain any symbol that is legal in a string (see 3.5.7) including single-quotes. These are allowed because there is no such thing as an empty character token: `''`. They are value tokens.

A double forward slash or the beginning or ending of any comment (`//`, `/*`, or `*/`, see 3.2) will be ignored within the boundaries of the character token, however since both are over one character in length and do not fall under the exception for escape codes, they are illegal.

3.5.7 Strings

The following is a double-quoted string:

```
"..."
```

It can contain alphanumeric characters, backslashes for use in escape codes, punctuation characters, and whitespace characters (spaces, tabs, and newlines) . Double-quotation marks (`"`) are not allowed as they would be mistaken for the end of the string token.

Punctuation characters:

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

The length of a string may range from 0 to infinity, although infinite strings are not possible due to computers' historical difficulties with infinite things.

Escape codes are as follows:

`\0`	Null byte (00)
`\a`	Bell character (07)
`\t`	Horizontal tab (09)
`\n`	Newline character (0A)
`\v`	Vertical tab (0B)
`\f`	Form feed (0C)
`\r`	Cartridge return character (0D)
`\\`	Backslash character (5C)
`\x...`	Hex byte (...)

The hex byte may be followed by a two digit hex code to insert that hexadecimal value directly into the string. Please do not overdose on cartridge returns; take a new line with your life.

The end result will be suffixed with a null byte. They are value tokens.

These tokens are brought into the program as `string` classes (see 6.6)

3.5.8 Hex Strings

Hex strings are used to perform magical hexes on programs that we do not like. They are, obviously, incredibly useful on a level most muggles do not understand.

They are identical to strings with the following differences.

They are prefixed with the alphabetical character '`x`':

```
x"..."
```

They store sequences of hexadecimal numbers, each being two characters long. This is because two-character hexadecimal numbers denote 8-bit values. These numbers are separated by any whitespace (see 3.4).

The end result will not be suffixed with a null byte, unlike strings (see 3.5.7). They are value tokens.

These tokens are brought into the program as `ref`s to data blocks.

3.5.9 Special Tokens

There are some tokens which are replaced with other tokens after parsing as follows (they must be replaced after parsing, due to the tokens dependency on knowing the current and parent functions):

`__FILE__`	A string containing the file path (as given to the compiler)
`__HEADER__`	True if the current file is a header file, otherwise false
`__FUNCTION_NAME__`	A string containing the name of the current function (empty if no current function)
`__FUNCTION_RETURN__`	A reference to the class used for the return of the current function (null if no current function)
`__FUNCTION_ARGUMENT_SIZE__`	An integer containing the number of arguments of the current function (`-1` if no current function)
`__FUNCTION_ARGUMENTS__`	A string containing the arguments (as written in the source code)
`__PARENT_FUNCTION_NAME__`	Like `__FUNCTION_NAME__`, but for the function calling the current function.
`__PARENT_FUNCTION_RETURN__`	Like `__FUNCTION_RETURN__`, but for the function calling the current function.
`__PARENT_FUNCTION_ARGUMENT_SIZE__`	Like `__FUNCTION_ARGUMENT_SIZE__`, but for the function calling the current function.
`__PARENT_FUNCTION_ARGUMENTS__`	Like `__FUNCTION_ARGUMENTS__`, but for the function calling the current function.
`__DATE__`	A string containing the date at compile time (formatted as `"MM/DD/YY"`)
`__DATE_YEAR__`	An integer containing the year at compile time (2000 C.E. giving a value of 2000)
`__DATE_MONTH__`	An integer ranging from 1 to 12 representing the date at compile time
`__DATE_DAY__`	An integer representing the day of the month at compile time
`__DATE_WEEKDAY__`	An integer representing the day of the week at compile time (0 being Sunday, ranging up to Saturday at 6)
`__TIME__`	A string containing the time at compile time (formatted as `"HH:MM:SS TIMEZONE"` in 24-hour)
`__TIME_HOUR__`	An integer containing the 24-hour time at compile time
`__TIME_MINUTE__`	An integer containing the minute at compile time
`__TIME_SECOND__`	An integer containing the second at compile time
`__TIME_ZONE__`	A string containing the timezone at compile time (i.e. `"UTC"` or `"EST"`)
`__OS__`	An integer representing the operating system (see the `__OS_...__` tokens for possible values of `__OS__`)
`__OS_UNIX__`	An integer valued at 0
`__OS_LINUX__`	An integer valued at 1
`__OS_OSX__`	An integer valued at 2
`__OS_WINDOWS__`	An integer valued at 3
`__OS_OTHER__`	An integer valued at 4
`__ARCH_BITS__`	An integer representing the number of bits in the architecture (can be 32 or 64)
`__ARCH__`	An integer representing the architecture (see the `__ARCH_...__` tokens for possible values of `__ARCH__`)
`__ARCH_X86__`	An integer valued at 0
`__ARCH_X86_64__`	An integer valued at 1
`__ARCH_ARM__`	An integer valued at 2
`__ARCH_AMD32__`	An integer valued at 3
`__ARCH_AMD64__`	An integer valued at 4

`__REGISTERS__`	An integer representing the number of registers in the architecture
`__REGISTER_LOW_BOUND__`	The ID of the lowest register for general use
`__REGISTER_HIGH_BOUND__`	The ID of the highest register for general use
`__REGISTER_RETURN__`	The ID of the register used for returning
`__REGISTER_SYSCALL__`	The ID of the register used for the argument of the syscall interrupt
`__REGISTER_EAX__`	The ID of the EAX register
`__REGISTER_ECX__`	The ID of the ECX register
`__REGISTER_EDX__`	The ID of the EDX register
`__REGISTER_EBX__`	The ID of the EBX register
`__REGISTER_ESP__`	The ID of the ESP register
`__REGISTER_EBP__`	The ID of the EBP register
`__REGISTER_ESI__`	The ID of the ESI register
`__REGISTER_EDI__`	The ID of the EDI register
`__AP_INIT__`	The initial offset of the argument pointer

Architecture specific special token values by default:

```
__REGISTERS__ = 8
__REGISTER_LOW_BOUND = __REGISTER_EAX__
__REGISTER_HIGH_BOUND = __REGISTER_EBX__
__REGISTER_RETURN__ = __REGISTER_EAX__
__REGISTER_SYSCALL = __REGISTER_EBX__
__REGISTER_EAX__ = 0
__REGISTER_ECX__ = 1
__REGISTER_EDX__ = 2
__REGISTER_EBX__ = 3
__REGISTER_ESP__ = 4
__REGISTER_EBP__ = 5
__REGISTER_ESI__ = 6
__REGISTER_EDI__ = 7
__AP_INIT__ = 8
```

The `__OS__`, `__OS_...__`, `__ARCH__`, `__ARCH_...__`, `__REGISTER...__`, `__FPU_...__`, and `__AP_...__` tokens should be allowed to be specified by the compiler so that more operating systems and architectures may be supported. They are value tokens.

4 Parsing

4.1 Inline Assembly

The body of a special expression (see 4.4) with the symbol "`asm`" contains a mini-language for generating assembly code to be directly inserted into the compilation. The mini-language contains the following operands (assembly commands):

```
nop const mov mov_disp ld st add sub mul and or xor pusharg poparg
leave ret int cmp breq brne brlt brle brgt brge jmp call fld fst
fadd fsub fmul fdiv fcmp fchs fst_rot fnop ftan fatan fsin fcos
fsincos fsqrt
```

The instructions are separated by `EndOfLine` tokens (see 3.3.1). The instructions can take up to three arguments separated by `Whitespace`. They can be either registers, symbols, or integer tokens.

The registers are as follows:

```
eax ecx edx ebx esp ebp esi edi
ax cx dx bx
al cl dl bl
```

The instructions follow the following formats:

```
nop
const REGISTER SYMBOL|INTEGER
mov8 REGISTER REGISTER
mov16 REGISTER REGISTER
mov32 REGISTER REGISTER
mov64 REGISTER REGISTER
mov32_disp REGISTER REGISTER SYMBOL|INTEGER
mov64_disp REGISTER REGISTER SYMBOL|INTEGER
ld8 REGISTER SYMBOL
ld16 REGISTER SYMBOL
ld32 REGISTER SYMBOL
ld64 REGISTER SYMBOL
st8 REGISTER SYMBOL
st16 REGISTER SYMBOL
st32 REGISTER SYMBOL
st64 REGISTER SYMBOL
add8 REGISTER REGISTER
add16 REGISTER REGISTER
add32 REGISTER REGISTER
add64 REGISTER REGISTER
sub8 REGISTER REGISTER
sub16 REGISTER REGISTER
sub32 REGISTER REGISTER
sub64 REGISTER REGISTER
mul8 REGISTER REGISTER
mul16 REGISTER REGISTER
mul32 REGISTER REGISTER
mul64 REGISTER REGISTER
and8 REGISTER REGISTER
and16 REGISTER REGISTER
and32 REGISTER REGISTER
and64 REGISTER REGISTER
or8 REGISTER REGISTER
or16 REGISTER REGISTER
or32 REGISTER REGISTER
or64 REGISTER REGISTER
xor8 REGISTER REGISTER
xor16 REGISTER REGISTER
xor32 REGISTER REGISTER
xor64 REGISTER REGISTER
pusharg8 REGISTER
pusharg16 REGISTER
pusharg32 REGISTER
```

```
pusharg64 REGISTER
poparg8 REGISTER
poparg16 REGISTER
poparg32 REGISTER
poparg64 REGISTER
leave
ret
int SYMBOL|INTEGER
cmp8 REGISTER SYMBOL|INTEGER
cmp16 REGISTER SYMBOL|INTEGER
cmp32 REGISTER SYMBOL|INTEGER
cmp64 REGISTER SYMBOL|INTEGER
breq SYMBOL
brne SYMBOL
brlt SYMBOL
brle SYMBOL
brgt SYMBOL
brge SYMBOL
jmp SYMBOL
call SYMBOL
fld SYMBOL
fst SYMBOL
fadd REGISTER REGISTER
fsub REGISTER REGISTER
fmul REGISTER REGISTER
fdiv REGISTER REGISTER
fcmp REGISTER REGISTER
fchs REGISTER
fst_rot
fnop
ftan
fatan
fsin
fcos
fsincos
fsqrt
```

There is one more instruction: `label SYMBOL`. It creates a new symbol containing the integer value of the code pointer within the assembly code. Labels are generated in place before the rest of the code and thus can be used to create relocatable code.

For example (assuming a 32-bit integer variable `x` has already been declared),

```
asm
{
    const eax 1; const ecx 2;

    add8 eax ecx;
    st32 eax x;
}
```

will add 1 to 2 and store the result in the variable `x`.

4.2 Type Expressions

```
... type BoundaryBeginExpression ... BoundaryEndExpression
```

by default,

```
... type ( ... )
```

They begin with a set of global type qualifiers. These can be

4.3 Complex Values

Complex values begin with a `BoundaryBeginComplex` (see 3.3.1). They contain a sequence of any constructs separated with `ArgumentSeparator`. They end with a `BoundaryEndComplex`.

For example,

```
[1, 2]
```

or,

```
[1+2, [1, 2]]
```

4.4 Operations

All operations are to be evaluated before their parent is evaluated.

An element access operation consists of a `Symbol` (see 3.3.1) followed by an `ElementAccess`, the operator for element access in the previous symbol.

```
SYMBOL .
```

For example (assuming an object `o` has already been declared with member `x`),

```
o.x
```

A unary right operation (either '`!`' or '`~`') is followed by an any construct. For example,

```
~o.x
```

You will notice the use of precedence (one operation must be parsed "first", before the other - see 4.3.2).

A unary left operation (either '`++`' or '`--`') is an any construct followed by an operation token. For example,

```
o.x++
```

A binary operation starts with an any construct followed by an operator token. It then ends with another any construct. For example,

```
o.x = 5
```

or,

```
o.x + 10
```

4.4.1 Expression Boundaries

Expressions may be bound by `BoundaryBeginExpression` and `BoundaryEndExpression` to directly specify a higher precedence (see 4.3.2). For example in,

```
1 + 2 * 3
```

the precedence is automatically generated based on the precedence levels of the operations. However, in the following example,

```
(1 + 2) * 3
```

The first operation is given higher precedence by direct specification using parenthesis.

4.4.2 Precedence

Some operations have precedence over other operations, which means that they have the illusion of being parsed before lower precedence operations. When parsing two binary operations where precedence is not specified by parenthesis, precedence goes by the following order (lowest to highest, ranging from `=` at 0 to `>>` at 12 - 32 binary operations):

```
= 0
+= 1
-= 1
*= 2
/= 3
%= 4
&= 4
|= 4
^= 4
~= 4
<<= 4
>>= 4
&& 5
|| 5
< 6
> 6
<= 6
>= 6
== 6
!= 6
+ 7
- 7
* 8
/ 9
% 10
& 10
| 10
^ 10
<< 11
```

```
>> 11
. 12
```

Unary operations all have higher precedence than binary operations, but do not have any precedence amongst themselves.

When parsing two operations following each other with no parenthesis to denote precedence directly (such as `1 + 2 + 3`), if the first operation has a higher precedence than the other (such as `1 * 2 + 3`) then the first operation is parsed and used as a member of the second operation. However, if the second operation has a higher precedence than the first (such as `1 + 2 * 3`) then the first operation starts to be parsed, but is only completed when the second operation is parsed and added as a member.

For example, `1 + 2 * 3` is the same as `1 + (2 * 3)`, while `(1 + 2) * 3` is the same as `1 * 3 + 2 * 3`.

4.5 Expressions

Expressions are the most complex parsing constructs, but are the most used within Mitten. They are of the structure:

```
Expression
    Type vector
    Symbol
    Complex value argument
    Argument vector
    Body
    Where body
```

The type vector contains a sequence (ranging from 0 to 6 in size - the most type tokens possible for a declaration is 6 and some declarations do not have types) of only `Type` tokens. The symbol is a single `Symbol`. The complex value argument consists of a single `ComplexValue` (optional). The argument vector (optional) is a sequence of any constructs separated by `ArgumentSeparator` tokens and bound with expression boundaries `BoundaryBeginExpression` and `BoundaryEndExpression`. The body (optional) is a sequence of expressions separated by `EndOfLine` tokens (including one at the end of the sequence) and bound by scope boundaries `BoundaryBeginScope` and `BoundaryEndScope`. The where body (optional) is identical to the body except that it is prefixed by the symbol "`where`".

Expressions with no body (or where body) must be followed with an `EndOfLine` token to denote their ending. Extra `EndOfLine` tokens after the body (or where body) will not cause problems, but are not necessary.

If there is both a where expression and a body, the where expression is generally executed prior to the body, but in the same manner.

Either a `Type` token or a `Symbol` must begin an expression. An expression must contain at least a complex value argument, an argument vector, or a body. It may contain all three. The ordering of the components is required. All of the following formats are legal expressions:

```
Symbol ComplexValue
Symbol ArgumentVector
Symbol Body
Symbol ComplexValue ArgumentVector
Symbol ArgumentVector Body
Symbol ComplexValue ArgumentVector Body
TypeVector Symbol ComplexValue
TypeVector Symbol ArgumentVector
TypeVector Symbol Body
TypeVector Symbol ComplexValue ArgumentVector
TypeVector Symbol ArgumentVector Body
TypeVector Symbol ComplexValue ArgumentVector Body
```

4.6 Any

An any construct can be either a single value or type token, a complex value construct, an operation, or an expression.

5 Special Expressions and Classes

5.1 Casts

```
cast BoundaryBeginComplex ... BoundaryEndComplex BoundaryBeginExpression ... BoundaryEndEx
```

by default,

```
cast [ ... ] ( ... ) ;
```

The complex value argument is a single type token denoting the type to cast to. The single argument is the value to be casted.

For example,

```
cast [bool] (1) == true
cast [int] ("5") == 5
cast [string] (5) == "5"
```

5.1.1 Casting between Internal Data Types

Integers may be casted around freely, although data may be lost in truncation from higher-precision types to lower-precision ones. Floats may be casted to integers with truncation. Integers may be casted to floats as whole numbers. Floats may be casted around freely, but again with the data loss in truncations between precisions. Unsigned data types and signed data types are casted directly: the same bits are used. Class definition tokens cannot be used within casts. A cast to an any type has no effect on the value. A value's type ID can also be used for casting (see 6), for example,

```
int x = 5;
x = cast [typeid(x)] ("5");
```

5.2 Returns

```
return BoundaryBeginExpression ... BoundaryEndExpression EndOfLine
```

by default,

```
return ( ... ) ;
```

Return statements denote the value of an expression. Expression values by default have the any type unless otherwise specified, so the return type is often auto-detected by the return type of the contents of the single return argument.

For example,

```
int x = if (5 > 4)
{
    return (1);
}
else
{
    return (0);
}
```

`x` should equal `1`.

5.3 If-Statements

```
if BoundaryBeginExpression ... BoundaryEndExpression BoundaryBeginScope ... BoundaryEndSco
```

by default,

```
if ( ... ) { ... }
```

If-statements are conditional on the boolean value of the single argument (automatically casted to boolean). If the value is true, the body will be executed, otherwise, execution will continue past the end of the body. Returns are legal within the body; upon falsehood a null value will be returned.

For example,

```
if (5 > 4 && 1 < 2)
{
    // this code will run because the condition is met
}

// this code will also run because it is outside the if-statement and
// does not rely on the condition.
```

5.4 Else-If-Statements

```
elif BoundaryBeginExpression ... BoundaryEndExpression BoundaryBeginScope ... BoundaryEndS
```

by default,

```
elif ( ... ) { ... }
```

For example,

```
if (5 < 4)
{
    // this shouldn't run
}
elif (1 < 2)
{
    // this should run
}
```

The last expression must be an if-statement or another else-if-statement. It is identical to an if-statement, except that it will only execute upon the falsehood of all the previous if and else-if statements. Returns are legal within the body; upon falsehood a null value will be returned.

5.5 Else-Statements

```
else BoundaryBeginScope ... BoundaryEndScope
```

by default,

```
else { ... }
```

The last expression must be an if-statement or an else-if-statement. It will only execute upon the falsehood of all the previous if and else-if statements. Returns are legal within the body; upon falsehood a null value is returned.

For example,

```
if (5 < 4)
{
    // this shouldn't run
}
elif (1 > 2)
{
    // neither should this
}
else
{
    // but this should
}
```

5.6 While Loops

```
while BoundaryBeginExpression ... BoundaryEndExpression BoundaryBeginScope ... BoundaryEnd
```

by default,

```
while ( ... ) { ... }
```

As long as the condition within the single argument (see 5.3) is true, execute the body in repetition. Returns are legal within the body; if the body is never executed a null value is returned. If the value of the expression is assigned to a vector, all of the return values generated in the repetition will be appended to the vector. If the expression is assigned to a non-vector type, the last returned value will be used.

For example,

```
while (true)
{
    diamonds();
}

// diamonds are forever
```

5.7 For Loops

```
for BoundaryBeginExpression ... ArgumentSeparator ... ArgumentSeparator ... BoundaryEndExp
```

by default,

```
for ( ... , ... , ... ) { ... }
```

The first argument of the for loop is an expression to be run at the beginning of the loop. The second argument is a conditional expression (see 5.3) on whether the loop repeats. The last argument is executed at the end of every cycle. Returns are legal within the body and are identical to while loops in nature (see 5.6).

For example,

```
for (int i = 0, i < 10, i++)
{
    get_an_apple();
}

// you should have be able to keep the doctor away for 10 days
```

5.8 For-Each Loops

```
for BoundaryBeginExpression ... ArgumentSeparator ... BoundaryEndExpression BoundaryBeginS
```

by default,

```
for ( ... , ... ) { ... }
```

The first argument of the for-each loop is a variable declaration (see 5.10 and **??**), although it may be empty. The second argument is a vector value. The body of the loop will be executed for each value of the type of the variable within the vector value. Returns are handled the same way as with for-loops (and while loops, see 5.6).

For-each loops only work on strings (using a character variable), vectors (using an element-type variable), or maps (using a key-type variable).

For example (see **??**),

```
for (byte c, "hello, world")
{
}
```

```
for (int i, [1, 2, 3])
{
}

for (int i, [[1, "a"], [2, "b"], [3, "c"]])
{
}
```

5.9 Continue and Break

```
continue BoundaryBeginExpression BoundaryEndExpression
break BoundaryBeginExpression BoundaryEndExpression
```

by default,

```
continue ( )
break ( )
```

If you run continue in a loop, it skips the rest of the code in the loop body. It can only be run in a loop. A call to break is equivalent to a null return in a loop. It, also, can only be run in a loop.

For example,

```
while (true) // this loop does nothing and is not infinite
{
    break();
}

int i = 0;
for (, i < 10, i++)
{
    continue();
    this_displays_something();
}
// nothing is displayed and i == 10
```

5.10 Simple Variable Declarations

```
... ...
```

The first part of the declaration is a sequence of type tokens. The second part is a symbol denoting the variable's name. If no value expression is provided by an `OperatorAssign`, a null value is assigned. You cannot declare a variable twice (with the same name) in the same scope.

For example,

```
int x; // will be 0
int y = 5;
int z = [1, 2];
```

5.11 Method Declarations

```
... ... BoundaryBeginExpression ... BoundaryEndExpression EndOfLine
... ... BoundaryBeginExpression ... BoundaryEndExpression BoundaryBeginScope ... BoundaryE
... OperatorAccess ... BoundaryBeginExpression ... BoundaryEndExpression EndOfLine
... OperatorAccess ... BoundaryBeginExpression ... BoundaryEndExpression BoundaryBeginScop
```

```
... ... ( ... ) ;
... ... ( ... ) { ... }
... . ... ( ... ) ;
... . ... ( ... ) { ... }
```

You will notice the two main forms: with no body and with body. When entering the body of a method, the `__FUNCTION...__` token's values must be assigned for use in the method. It is possible to declare a method within a method declaration, although the declared functions are only accessible within the parent function's body and cannot be made public in a class. Returns are legal; if no return is specified, the method is assumed to return a null value. The return type must match the type specified within the declaration.

The arguments are identical to variable declarations (see 5.10). The declarations must not have value expressions.

A function without a body is only a prototype and is assumed to be externally linked unless a re-declaration with a body is specified. The re-declaration must not differ from the prototype in any way (including argument names).

Functions can be overloaded by argument number and type as well as return type.

You will notice that the last two forms contain a period, the operator for class member access. Instead of a symbol denoting the member's name an operation expression (see 4.3) can be used. The method will be declared within the scope of the class. Only one period may be used; you cannot declare methods within a member of a specified class. You can think of these two forms as a `Type Symbol` expression in an `OperatorAccess` operation with a typeless method declaration.

You can use the variadic type token to create variadic methods (see 5.13 and 6.10). If the variadic type token is used, then no argument may be specified (empty argument list).

For example,

```
int max(int a, int b);

int max(int a, int b)
{
    return (if (a > b)
    {
        return (a);
    }
    else
    {
        return (b);
    });
}
```

```
variadic int first()
{
    return variadic_args()[0];
}
```

5.12 Method Calls

```
... BoundaryBeginExpression ... BoundaryEndExpression
... BoundaryBeginComplex ... BoundaryEndComplex BoundaryBeginExpression ... BoundaryEndExp
... OperatorAccess ... BoundaryBeginExpression ... BoundaryEndExpression
... OperatorAccess ... BoundaryBeginComplex ... BoundaryEndComplex BoundaryBeginExpression
```

by default,

```
... ( ... )
... [ ... ] ( ... )
... . ... ( ... )
... . ... [ ... ] ( ... )
```

The first part of the call is the symbol denoting the method's name. It may be preceded by a list of period-separated object names followed by another period. This denotes the object of which the method called is a member. The second part is a list of arguments to be passed to the function. They are expressions. If they are assignment operator expressions (see 4.3) and the variable being assigned is declared within the method, its declaration within the method is asserted for this call only to have the value specified in the argument. Otherwise, if the variable being assigned is declared outside the method but within the scope of the method call, the variable outside the method is assigned and the value is also passed as an argument to the method.

If a complex value argument is passed, it must contain the type of the return of the method. If the method is overloaded, a method is chosen based on its return type. If no overloaded method exists with that return type, it is assumed to be a cast of the first overloaded declaration of the method with the specified arguments.

The value of the expression is identical to the return of the declaration.

Classes within method calls are all pass-by-reference, unless the argument uses the const type token, in which case it is pass-by-value. The same goes for return types.

The arguments must match the arguments within the method declaration. Variadic methods must have no arguments.

For example (assuming there is an object `o` with a method `m` and a standalone method `f`),

```
f(5)
f [int] (5)
o.m(5)
o.m [int] (5)
```

5.13 Variadic

```
... variadic_args BoundaryBeginExpression BoundaryEndExpression
... variadic_args ( )
```

This expression, which acts much like a method call, can only be used within variadic method declaration bodies. It returns a vector containing all of the arguments passed to the method. For example,

```
variadic int first()
{
    return va()[0];
}
```

5.14 Class Declarations

```
class ... BoundaryBeginExpression ... BoundaryEndExpression BoundaryBeginScope ... Boundar
class ... BoundaryBeginScope ... BoundaryEndScope
```

by default,

```
class ... ( ... ) { ... }
class ... { ... }
```

Class declarations take only a single argument: the class to inherit from. If no class is given to inherit from, it inherits from the zygote class. The body contains a series of variable declarations and method declarations (with or without bodies, see 5.10, **??**, and 5.11). The use of the type tokens private and public come into use here.

If a member (a variable or method within the class) is private, it is only accessible by other members within the class within the scope of the class. It will not be accessible by members of the classes inheriting the current class.

If a member is protected, it is identical to the private members except that it will be accessible by the members of the classes inheriting the current class.

If a member is public, it may be accessible from any scope.

If a member is declared only within a source file and not in a header file (methods only), it is assumed to be protected unless otherwise specified.

Classes may not be declared inside class declarations. You cannot declare a class twice.

With every class declared, a new method is also declared with the same name as the class. It takes no arguments by default (it can be overridden using the `__new__` method within the class declaration, see 6) and returns a newly created object.

Every class contains a special method called `any __complex__ [ ... ]`. If this method is declared, calling an object with only a complex argument (no member specified) will call the `__complex__` method.

For example,

```
class pair_of_numbers
{
    private int a = 0;
    private int b = 0;

    void __new__();
    void __new__(int x, int y);
    int max();
}

void pair_of_numbers.__new__()
{
}

void pair_of_numbers.__new__(int x, int y)
{
    a = x;
    b = y;
}

int pair_of_numbers.max()
{
    return (if (a > b)
    {
        return (a);
    }
    else
    {
        return (b);
    });
}
```

You could use this class as follows:

```
pair_of_numbers p = pair_of_numbers(1, 2);
// p.max() == 2
```

5.15 Address and Deference

```
... addr BoundaryBeginExpression ... BoundaryEndExpression
```

by default,

```
... addr ( ... )
```

`addr` gets the address in memory of any element that you pass to it. For example,

```
int x = 5;
ref ptr = addr(x);

ref another_ptr = addr(if(true){return(5);}));
```

Then there is `deref`:

```
... deref BoundaryBeginComplex ... BoundaryEndComplex BoundaryBeginExpression ... Boundary
```

by default,

```
... deref [ ... ] ( ... )
```

which dereferences the reference or memory location that you pass to it in the type specified in the complex value argument. For example (building off the `addr` examples),

```
int y = deref [int] (ptr);
// y == 5

int z = deref [int] (another_ptr);
// z == 5

int w = deref [int] (addr(5));
// w == 5
```

If you attempt to dereference a null reference or memory location, a `null_exception` will be thrown (see 6.5).

5.16 Try/Catch/Throw Statements

```
try BoundaryBeginScope ... BoundaryEndScope
catch BoundaryBeginExpression ... ... BoundaryEndExpression BoundaryBeginScope ... Boundar
throw BoundaryBeginExpression ... ... BoundaryEndExpression
```

by default,

```
try { ... }
catch ( ... ... ) { ... }
throw ( ... ... )
```

The contents of the try body are executed normally. If a throw statement is executed within the try body, the catch statement is executed with a single argument. Each try statement may be suffixed with multiple catch statements, each with a unique type for its only argument. The argument is an exception passed to the body. If a throw statement's type does not match any of the catch statements or no try/catch statements are surrounding the throw, the exception handler method is called. It can be overloaded with the following declaration:

```
void __handle_exception__(exception e);
```

All exception types inherit the exception class.

Exceptions (with the default handler) kill the current thread.

For example,

```
try
{
    ref pointer = null;
    deref(pointer);
}
catch(null_exception e)
{
    // this will only catch null_exceptions
}
catch(exception e)
{
    // this will catch all other exceptions
}
```

5.17 Imports

```
import BoundaryBeginExpression ... BoundaryEndExpression EndOfLine
import_multi BoundaryBeginExpression ... BoundaryEndExpression EndOfLine
```

by default,

```
import ( ... ) ;
import_multi ( ... ) ;
```

Imports take a single argument: a string of the path to the file you are importing. The result of the import is that a public object will be created containing the contents of the module. It will have members according to the modules that it, in turn, imports.

Any given file can only be imported once. Any subsequent imports will be ignored. `import_multi` does not provide this functionality. A file imported with `import_multi` can be imported an infinite number of times.

If the path starts with a period, then it is relative. Otherwise the import path is searched for the file. Paths must not start with slashes for safety.

5.18 Compatibility

```
compatibility BoundaryBeginComplex "c" BoundaryEndComplex BoundaryBeginExpression ... Boun
compatibility BoundaryBeginComplex "c" BoundaryEndComplex BoundaryBeginExpression ... Argu
```

by default,

```
compatibility [ "c" ] ( ... ) ;
compatibility [ "c" ] ( ... , ... ) ;
```

You will notice the only language that Mitten is compatible with is C. If you chose to use a combination of Objective-C alongside Mitten, then you are not connected with the good-idea energy field.

The one-argument compatibility expression must contain a single string literal containing a pure-C prototype. Types are converted between C and Mitten as follows:

C	Mitten
`void`	void
`char`	byte
`short`	long byte
`int`	int
`long`	long int
`float`	float
`double`	long float
`void *`	ref
`char *`	string (with a class wrapper)
`short *`	long byte vector
`int *`	int vector
`long *`	long int vector
`float *`	float vector
`double *`	double vector

Structs, unions, and enums are not supported. Use `void *` instead and convert manually with pointer arithmetic and casting.

The two-argument compatibility expression must contain two literals: a string path to a pure-C header file and a vector of string literals containing the list of functions to import as externally-linked prototypes. The prototypes are searched for in the header file and imported in the same manner as with the one-argument compatibility expression. The prototypes must be on one line for this to work and must be suffixed by semicolons. For example, this would work:

```
int max(int a, int b);
```

while these would not,

```
int
max(int a, int b);

int max (
    int a,
    int b
);

int max(int a, int b)
{
    return (a > b ? a : b);
}
```

5.19 Inline Debugging

```
track BoundaryBeginExpression ... BoundaryEndExpression
debug BoundaryBeginExpression ... BoundaryEndExpression
```

by default,

```
track ( ... )
debug ( ... )
```

If you call `track` with a symbol as an argument, it tracks the usage of that argument. If it is a method, it prints debug messages stating the arguments that it is called with, the overloaded version used, and what the return value is. If it is a variable, it prints debug messages on every value change. Whether these messages are visible or not should be able to be set at compile time.

6 Data Model

If any class has the following methods, they can override the default operators:

```
... __add__(... other); // +
... __sub__(... other); // -
... __mul__(... other); // *
... __div__(... other); // /
... __mod__(... other); // %
... __and__(... other); // &
... __logical_and__(... other); // &&
... __or__(... other); // |
... __logical_or__(... other); // ||
... __xor__(... other); // ^
... __neg__(... other); // ~
... __logical_neg__(... other); // !
... __left__(... other); // <<
... __right__(... other); // >>
... __lt__(... other); // <
... __le__(... other); // <=
... __gt__(... other); // >
... __ge__(... other); // >=
... __eq__(... other); // ==
... __ne__(... other); // !=
... __assign__(... other); // =
... __assign_add__(... other); // +=
... __assign_sub__(... other); // -=
... __assign_mul__(... other); // *=
... __assign_div__(... other); // /=
... __assign_mod__(... other); // %=
... __assign_and__(... other); // &=
... __assign_or__(... other); // |=
... __assign_xor__(... other); // ^=
... __assign_neg__(... other); // ~=
... __assign_left__(... other); // <<=
... __assign_right__(... other); // >>=
```

If an `__assign_...__` method is not defined, the `__...__` and `__assign__` methods are called in sequence.

If a `... __cast__ [ ... ] ()` method is defined, casting will try it first. If it returns null, casting will automatically attempt to cast.

You can override creation and destruction with:

```
... __new__( ... );
void __del__();
```

Polymorphism is completely allowed for all classes.

6.1 Serialization

Two methods which cannot be overloaded are `ref __serialize__()` and `void __unserialize__(ref block)`. They save and load objects to raw data reference. Each of the members is recursively serialized and appended to the data block. References to these members are relocated using a symbol table at the beginning of the serialization. Symbols within the table consist of an unsigned integer type containing the offset of the memory block of the symbol from the beginning of the serialization followed by another unsigned integer type containing the unique ID of the member and the unique ID of the parent object. The serialization is of the following format:

```
unsigned int - number of symbols
unsigned int - size of serialization in total (in bytes)
symbol table - sequence of:
    unsigned int - offset of symbol data block
    unsigned int - member ID
    unsigned int - parent object ID
data segment - sequence of data blocks
```

The `__unserialize__` method overwrites the current object.

6.2 Object Lifetimes

Objects begin life when they are created and end life when they exit the scope. They are deallocated when they exit the scope, unless they are returned from within the scope to the outside scope. If they are not used or placed into a new variable after being returned, they are deallocated. Variables which are definitely null (no matter what path execution takes) must not be accessed. Variables which are accessed which are null should throw a `null_exception`.

The `__del__` method, by default, simply calls all deletion methods of the members of the class. The `__new__` method, by default, calls all initialization methods of the members with no arguments (if such methods exist). Otherwise, it sets the members to null.

6.3 Atomic Classes

All atomic classes still inherit the zygote class.

The only atomic class which overloads the logical operators is the bool class. It overloads the logical operators, the assignment operators, and the equal to and not equal to operators.

Atomic classes are the classes that are emulated by the compiler for integer, float, and reference types. Integer types, although varying in size and signedness, all have classes with the same methods. All operators are overloaded (see 6). A new value can be created from a reference by taking on the first integer value stored in the memory of the reference. They declare a set of extra method for integral operations:

```
int __pow__(int e);
```

Floats are identical, except that do not overload bit operations and they also declare a set of other methods for floating-point operations:

```
float __pow__(float e);
float __tan__();
float __atan__();
float __sin__();
float __asin__();
float __cos__();
float __acos__();
float __sqrt__();
```

References only overload the addition, subtraction, assignment, and comparation operators. They also have a `any __deref__()` method which dereferences the reference. They overload the complex value argument method to provide indexes:

```
any __complex__ [int idx];
any __complex__ [int tid, int idx];
```

The second complex value argument method uses the index to describe a reference to a block of objects with type ID tid, as opposed to bytes.

6.4 Zygote

The zygote class (type ID 14) has the following methods by default:

```
int __typeid__();
int __sizeof__();
string __typestr__();
string __filepath__();
int __inherits__();
```

`__typeid__` returns a type ID number unique to the class (zygote's should be 14). `__sizeof__` returns the size of an object in bytes. `__typestr__` returns a string containing the name of the class denoted by the declaration. `__filepath__` returns a string containing the value of the `__FILE__` token that the class was declared in. `__inherits__` returns the typeid of the class inherited from.

All classes that do not specify which class they inherit from inherit from zygote by default.

6.5 Exception

The exception class (type ID 15) has the following methods:

```
string __message__();
int __line__();
string __file__();
string __function_name__();
string __function_return__();
int __function_argument_size__();
string __function_arguments__();
string __timestamp__();
void __attach__(string title, any value);
map __attachments__();
exception __parent__();
```

The `__message__` method returns a message for use in displaying the exception. The `__line__` method returns the line number the exception occurred on. The `__function_...__` methods return the values of the corresponding special tokens (see 3.5.9). `__timestamp__` return a string timestamp of the format ”`HH:MM:SS MM:DD:YY`” (24-hour time, see 3.5.9). `__attach__` attaches a piece of data to the exception. `__attachments__` returns a map of attached data (the titles as the keys and the values as the values). `__parent__` returns the parent exception in the stack trace (null if none exists).

There are a number of internal exceptions:

```
null_exception allocation_exception outside_set_exception
cast_exception
```

`allocation_exception`s are used when there is not enough memory to allocate an object. `outside_set_exception`s are used when an index or key that is not valid was used. `cast_exception`s are used when dynamic casts result in null values.

6.6 String

The string class (type ID 16) has the following methods:

```
string __new__();
string __new__(byte c);
string __new__(byte c, unsigned int repeat);
string __new__(ref r);
void __del__();
ref __complex__(unsigned int idx);
ref data();
unsigned int size();
void append(byte c);
void append(string other);
string substr(unsigned int start);
string substr(unsigned int start, unsigned int end);
unsigned int find(byte pattern);
unsigned int find(string pattern);
unsigned int rfind(byte pattern);
unsigned int rfind(string pattern);
bool endswith(string pattern);
bool startswith(string pattern);
void remove(unsigned int idx);
void remove(unsigned int start, unsigned int end);
void insert(unsigned int idx, string other);
void replace(unsigned int start, unsigned int end, string with);
```

`__new__` allows you to create a string with a single byte, a single byte repeated a number of times, or a reference to a memory location containing a string. You can access character references with `__complex__`. `data` returns a reference to the raw C-compatible data of the string. The other functions are relatively self-explanatory. If a find method cannot find the pattern, the size of the string is returned.

6.7 Vector

The vector class (type ID 17) has the following methods:

```
vector __new__();
vector __new__(any elem);
void __del__();
ref __complex__(unsigned int idx);
ref data();
unsigned int size();
vector subvector(unsigned int start);
vector subvector(unsigned int start, unsigned int end();
void append(any elem);
void append(vector other);
void insert(unsigned int idx, vector other);
void remove(unsigned int idx);
void remove(unsigned int start, unsigned int end);
```

6.8 Map

The map class (type ID 18) has the following methods:

```
map __new__();
void __del__();
ref __complex__(any key);
unsigned int size();
void insert(map other);
void remove(any key);
vector values();
```

6.9 Method Reference

The methodref class (type ID 19) has the following methods:

```
methodref __new__();
methodref __new__(ref method);
void pusharg(any arg);
any call [int tid] ();
```

You create it from making references from methods. Then you use `pusharg` to add arguments to a call. Once you run `call`, all of the arguments are popped (the stack is cleared for the next call) and the result of the method is returned. You have to supply the type ID of the return of the function to the call for it to return correctly.

6.10 Internal Methods

The following all call internally defined methods within objects.

```
any new(int tid);
variadic any new();
void del(any obj);
int typeid(any obj);
int sizeof(any obj);
string typestr(any obj);
string filepath(any obj);
int inherits(any obj);
any pow(any obj, any exp);
any tan(any obj);
any atan(any obj);
any sin(any obj);
any asin(any obj);
any cos(any obj);
any acos(any obj);
any sqrt(any obj);
void attach(exception exc, string title, any value);
ref serialize(any obj);
any unserialize(ref serial);
```

`new` always takes a type ID argument and calls the `__new__` method of the corresponding class. `del` calls the `__del__` method of the object. `typeid` returns the type ID of the object. `sizeof` returns the size in memory of the object. `typestr` returns a string containing the name of the class of the object (as declared). `filepath` returns the file path, as passed to the compiler, in which the object's class was declared. `inherits` returns the type ID of the parent class of the class of the object. `pow` calls `__pow__`. `tan` calls `__tan__`. `atan` calls `__atan__`. `sin` calls `__sin__`. `asin` calls `__asin__`. `cos` calls `__cos__`. `acos` calls `__acos__`. `sqrt` calls `__sqrt__`. `attach` calls `__attach__`. `serialize` calls `__serialize__`. `unserialize` calls `__unserialize__`.