The Mank Programming Language Specification

Introduction

```
proc main {
  println!("Hello World!");
}
```

Mank is a modern, strongly typed, and garbage-collected mid-level language. Mank aims to provide more safety and portability than C, while also providing high-level features, such as lambdas & closures, type inference, ~~generics~~, and more.

Notation

Extended Backus–Naur form (ENBF) is used to show various rules and productions.

Lexical elements

Letters and digits

```
letter = ('A' .. 'Z') | ('a' .. 'z') ;
character = (* any valid character *) ;

decimal_digits = digit, { digit } ;
digit =  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
```

Comments

```
# This is a comment.
```

Single line comments start with # and stop at the end of the line.

Comments cannot start within a string literal. All comments are treated as whitespace, and they don't currently have any semantic meaning (e.g. documentation generation).

Integer literals

Integer literals are sequences of digits representing an integer constant.

```
integer_literal = decimal_digits ;
```

There are currently no special prefixes to allow for different bases.

```
42
1337
099     # = 99
000000  # = 0
100
```

Floating-point literals

Floating-point literals are decimal representations of floating-point constants.

Floating-point literals consist of an integer part, decimal point, and a fractional part. The fractional part may be omitted (though keeping the decimal point).

```
floating_point_literal = decimal_digits, '.', [ decimal_digits ] ;
```

```
42.       # = 42.0
10.0
3.14
1.00      # = 1.0
00032.    # = 32.0
0010.000  # = 10.0
093.0     # = 93.0
```

Identifiers

Identifiers name things within a program, such as variables and functions. Identifiers must start with a letter or underscore, and then can be followed by any number of letters, underscores, or numbers.

```
identifiers = (letter | '_'), { letter | digit | '_' } ;
```

```
foo_bar
FooBar
_foo
_0123
bar1
fooBar
```

Keywords

```
fun     proc    domain  until   pod     bind    continue  const
of      spawn   if      return  ref     loop    as        in
else    for     while   true    false   break   test      enum
```

These are reserved and cannot be used as identifiers.

Operators and punctuation

Operators and various types of assignment, along with punctuation:

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

Character literals

Character literals represent ASCII character constants.

Character literals consist of a single (optionally escaped with \) character surrounded by single quotes.

```
character_literal = "'", [ "\" ], character, "'" ;
```

```
'a'
'\n'  # a newline
'\e'  # escape char
'\''  # a single quote (') character
'"'
```

String literals

String literals represent a string constant, which is a sequence of characters.

String literals can contain escaped characters (in the same way as in character literals).

```
string_literal = '"', { ['\'], character }, '"'
```

```
"Hello World"
"I'm on a typewriter!\r\n"  # string with legacy line ending
"Name set to \"Ben\""       # string containing escaped double quotes
```

Types

```
Type = ReferenceType | BaseType ;
BaseType = TypeName | LambdaType | TupleType
         | FixedSizeArrayType | ListType | DelimitedType ;
DelimitedType = "|", Type, "|" ;

TypeList = Type, { ",", Type }, [","] ;
```

`TypeName` covers any type referred to by name, such as primitive types and pod types. Other types are constructed from simpler types, e.g. a tuple type could be `(i32, bool)`.

Type delimitations (`DelimitedType`) just group types, they can sometimes make things easier to read, and are necessary in some cases.

Boolean types

Booleans represented by the `bool` type and can take the value `true` or `false` (which are built-in keyword constants).

Numeric types

Numeric types are floating-point or integer types (which also includes `char`).

```
char  # set of all signed 8-bit integers (that use character literals)
i32   # set of all signed 32-bit integers
i64   # set of all signed 64-bit integers
```

```
f32   # set of all 32-bit floating-point numbers
f64   # set of all 64-bit floating-point numbers
```

More numeric types are planned but are currently unimplemented.

Reference types

A reference is a type that holds a pointer to a value of another type (the referenced type). References cannot be null.

In some cases (such as in lambda expressions, or bind statements), the referenced type can be omitted and inferred based on context.

```
ReferenceType = "ref", [ReferencedType] ;
ReferencedType = BaseType ;
```

```
ref i32
ref bool[10]
```

Fixed-size array types

Fixed-size arrays hold a fixed number (the length) of elements of another type (the element type). The length cannot be a negative number, the length of a fixed-sized array can be obtained from the `.length` attribute.

Fixed-size arrays can have multiple dimensions, given by a comma-separated list of dimensions. In outermost dimension is given first, preceding nested dimensions.

```
FixedSizeArrayType = ElementType, "[" Dimensions "]" ;
ElementType = BaseType ;
Dimensions = Dimension, { "," Dimension } ;
Dimension = integer_literal ;
```

```
f32[10]
i32[2,3]         # multidimensional array (two arrays of three)
(i32,bool)[999]  # 999 tuples of i32 and bool
```

String types

Strings represented by the `str` type are sequences of bytes (chars), with a positive length. Strings are immutable (they cannot be changed once created).

The length of any string can be obtained from the `.length` attribute. Strings are indexable from `0` to `length - 1`, indexing a string gives the character at that position in the string, as a `char`.

Pod types

A pod (from plain old data) is a collection of named elements, called fields, each with its own associated type.

Pod types are declared at the top level of a module and then referred to by name in the rest of the program.

Enum types

An enum (from enumeration) is a collection of named members which may be used as constants within a program (e.g. each member could be a state).

Optionally each member can be associated with tuple or pod-like data to form a tagged union. The data can then be extracted when matched against in a switch expression.

Enums are declared at the top level of a module and then referred to by name in the rest of the program.

Tuple types

A tuple is an unnamed collection of (also unnamed) elements of various types. The empty tuple is used as the void type (`void` in C or C++).

```
TupleType = "(", [TypeList], ")" ;
```

```
()                          # empty tuple/void
(i32, bool)                 # tuple of i32 and bool
(MyPod, (i32, f64), bool)   # tuple of a pod type, a nested tuple, and a bool
(char)                      # one element tuple
```

Lambda types

Lambda types are the types of lambda expressions, and functions/procedures. A specific lambda type represents all function-like objects with the same parameters, and return type.

```
LambdaType = "\", [ParameterTypes], "->", ReturnType ;
ParameterTypes = TypeList ;
ReturnType = Type ;
```

```
\ -> char         # lambda with no parameters, returning a char
\char -> ()       # lambda that takes a char and returns void
\i32, i32 -> i32  # lambda that takes two integers returning an integer
# lambda taking a list of integers and a lambda, returning a list of integers
\i32[], \i32 -> i32 -> i32[]
# written with explict delimiters
\i32[], |\i32 -> i32| -> i32[]
```

List types

Lists (also called vectors) hold a dynamic amount of elements of another type (the element type). The number of elements in a list can be queried with the `.length` attribute.

```
ListType = ElementType, "[", "]", ;
ElementType = BaseType ;
```

```
str[]           # list of strings
i32[][]         # list of lists of integers
(i32, bool)[]   # list of tuples of i32 and bool
```

Type and value properties

Type matches

Types either match or are different.

All named types (primitives like strings, bools, ints, along with pods) are always different to any other type (other than their own type).

- At the top level, any type can be matched with a reference of its type.
    - This excludes references nested in composite types.
        - e.g. `(i32, ref bool) != (i32, bool)`.
- Fixed-size arrays match if they're the same length and their element types match.
- Lists match if their element types match.
- Tuples match if all elements of the tuples match, in the same order.
- Lambdas match if their parameters and return types match respectively.

Note for references there are additional restrictions on the types of values they can be assigned to alongside matching types.

Type storage

All types with exception of strings and lists are stored on the stack, and are copied by value when assigned to a variable or passed to a function (for reference types the value is just the pointer/memory address).

Strings are either stored within the program's static data or on the heap (depending on if they're compile-time constants or not), and lists are always heap-allocated. When these types are copied only their internal pointers are copied. This poses no issues for strings (as they're immutable), but it does mean lists need to be explicitly copied in some cases (e.g. before passing a list to a function that does some in-place processing while also maintaining the original list).

Assignability

A value x is assignable to a type T if:

- x's type is identical to T.
- T is the target type in an assignment statement and is a reference to x's type, in which case the referenced value is set to x.
- x's type is a reference to T, x will be (automatically) dereferenced to a value of type T.

- T is a reference to x's type at a binding point, and x is an lvalue.

Binding points

Binding points are the points in programs where new references can be introduced (or copied). Note that once a reference is bound, there's no way to change the reference to point to another value.

References can only be assigned to lvalues, which are values that have a memory address (so can appear on the *left* of an assignment). All other values are rvalues. For example, a named variable has a memory address, but a literal expression does not.

The current binding points are:

- Return statements/implicit returns
- Variable declarations
- Destructuring assignments binds
- Pod literal fields
- Call arguments

Declarations and constructs

```
TypeAnnotation = ":", Type ;
OptionalTypeAnnotation = ":", [Type] ;
```

All the declarations in this section appear at the top level of a file (not within functions).

When resolving names the order of declarations **does not** matter (e.g. a function can refer to a constant declared after it).

For each category of declaration (function, pod, constant, etc) all declarations within a module must have a unique name, otherwise, you get a `redeclaration error`. Shadowing is allowed between declarations of different types (as a warning), but does not serve much purpose due to the out-of-order name resolution meaning only the last declaration will be used (at any point).

Constant declarations

Constant declarations bind a name to a compile-time constant. These are global and can be used between functions.

Constants can refer to other constants (that are possibly declared later) as long as it's not a recursive definition.

```
ConstDecl = "const", OptionalTypeAnnotation, "=", ConstantExpression ;
ConstantExpression = Expression ; (* constness checked in semantics *)
```

```
# constant with explict type
const WITH: i32 = 10000;
# constant with inferred type
const HEIGHT := 4000;
```

```
# constant using constants defined later
const PROGRAM_VERSION := "version-" + PROGRAM_MAJOR + "." + PROGRAM_MINOR;
const PROGRAM_MAJOR := "123";
const PROGRAM_MINOR := "21";

# invalid (recursive)
const INVALID := INVALID + 1;
```

Function/procedure declarations

Function declarations bind a name to a function. Mank supports several types of functions: functions (`fun`) which return a value, procedures (`proc`) that always return void, and `test` functions.

"`test`" functions are a special case, and are used by the built-in test runner. If not building with the `--tests` flag they are not compiled.

```
FunctionDecl = FunctionHeader, Block ;
FunctionHeader = ("fun", identifier, Return, [ParameterList])
               | ("proc", identifier, [ParameterList])
               | ("test", identifier) ;
Return = TypeAnnotation ;
ParameterList = "(", {identifier, TypeAnnotation}, ")" ;
```

Top-level functions are not strictly first-class functions, though they will be coerced to lambdas where needed (e.g. if you try to pass a top-level function as an argument to a function accepting a lambda). This is done by automatically wrapping the function within a lambda expression with a matching signature (same return and parameter types).

It is not possible to nest top-level functions to create closures/non-local access, but this is possible with lambdas.

A function is free to mutate any local variables, or parameters passed to it (though unless these are references or heap-allocated data like lists this is not observable outside the function).

If a function body ends with an expression (without a semicolon after it), then that expression is returned from the function. Values can also be returned early, or explicitly with return statements.

```
fun max: i32 (a: i32, b: i32) {
  if a > b { a } else { b}
}

# the parentheses for the arguments can be omitted there are none
fun get_version: str {
  "Mank v0.1"   # implict return: final expression (no semicolon)
}

fun compareInt: i32 (a: i32, b: i32) {
```

```
  return a - b; # explict return statement
}

proc main (args: str) {
  println!("Hello {name}", args[1]);
}

# invalid! procedures don't return a value
proc add: i32 (a: i32, b: i32) {
  a + b
}

# tests never can have returns or parameters
test that_addtion_works {
  assert!(1 + 1 == 2);
}
```

Pod declarations

Pod declarations bind a name to a pod type.

Each field name within the pod must be unique. Recursive pod declarations are invalid (as they lead to a pod of infinite size).

```
PodDecl = "pod", PodName, "{", [Fields], "}" ;
PodName = identifier ;
Fields = FieldDecl, { ",", FieldDecl }, [","] ;
FieldDecl = identifier, ":", Type ;
```

```
# An empty pod
pod Empty {}

# Pod called "MyPod" with 3 fields (foo, bar, baz)
pod MyPod {
  foo: i32,
  bar: bool[5],
  baz: ref f64
}

# Invalid recursive definition
pod Recursive {
  foo: Recursive;
}

# This is allowed due to the indirection from the reference
pod Recursive {
```

```
  foo: ref Recursive;
}
```

Enum declarations

Enum declarations bind a name to an enum type.

Each member within the enum must have a unique name. As with pods, recursive enums are invalid (e.g. using the enum being declared in the pod, or tuple data of a member). If a member has pod data then that member's declaration must follow the same rules as a normal pod declaration.

```
EnumDecl = "enum", EnumName, "{", [Members], "}" ;
EnumName = identifier ;
Members = MemberDecl, { ",", MemberDecl }, [","] ;
MemberDecl = identifier, [MemberData] ;
MemberData = TupleData | PodData;
TupleData = TupleType ; (* same as the tuple type *)
PodData = Fields ; (* same as the pod declaration fields *)
```

```
# a simple C-like enum of colours
enum Colour {
  RED,
  GREEN,
  BLUE
}

# a tagged union of input events
enum Event {
  Keypress(char),                # member with tuple like data,
  MouseClick { x: i32, y: i32 },  # with pod like data,
  Quit                           # and without any associated data
}
```

Structural bindings

Structural bindings are a way of binding names to the elements/fields (nested or otherwise) of an initializer. They are used within the structural binding statement and switch expressions.

```
StructuralBinding = TupleBindings | PodBindings ;
TupleBindings = "(", [TupleBindingList], ")" ;
TupleBindingList = TupleBind, {",", TupleBind}, [","] ;
TupleBind = Bind | StructuralBinding ;
PodBindings = "{", [PodBindingList], "}" ;
PodBindingList = PodBind, {",", PodBind}, [","] ;
PodBind = ".", identifier, ([ "/", (Bind |  StructuralBinding) ] |
```

```
TypeAnnotation);
Bind = identifier, [TypeAnnotation] ;
```

Given these definitions and an instance of `Example`

```
pod Other {
  apple: f64
}

pod Example {
  foo: i32,
  bar: f64,
  nested: Other,
  tuple: (f64, bool, str)
}
```

the following are all valid bindings:

```
# binds foo and bar to the fields with the same name
{.foo, .bar}
# the same with explict types
{.foo:i32, .bar:f64}
# binds the fields foo and bar, but renames them to my_foo and my_bar
{.foo/my_foo, .bar/my_bar:f64}
# binds foo (like before), but also binds the nested field apple
{.foo, .nested/{.apple}}
# binds bar (like before), but also the nested tuple elements as a, b, and c
{.bar, .tuple/(a, b, c)}
```

(in this context, to "bind" a field/element means to create a local variable for it)

Given this tuple (where `example` is an instance of `Example`)

```
(example, (1, true), 5.0)
```

the following are all valid bindings:

```
# binds the elements of the tuple to my_example, tuple, and num
(my_example, tuple, num)
# binds the first element to a the nested tuple elements to b and c, and
# then the final element to d
(a, (b, c), d)
# uses a pod binding on the first element, then binds the remaining elements
({.foo/a}, b, c)
# binds the elements to a, b, and c with type annotations on a and c
(a: Example, b, c: f64)
```

Note that unlike pod bindings (where fields can be skipped and reordered), tuple bindings must have the same shape as the initializing tuple.

Bindings can also take references to fields/elements of initializers if they're lvalues, by annotating the binding with a reference type:

```
# bind my_exampe as a reference (reference type inferred)
# also binds c to a reference (full reference type given)
(my_exampe: ref, b, c: ref f64)
```

Expressions

Expressions are loosely defined as anything that can yield a value. In Mank, this can be simple constants, variables, or mathematical formulas, as in most languages, but can also incorporate control low such as in if and switch expressions.

Unless otherwise stated, all expressions are assumed to be rvalues.

Note that in expressions that require certain types, references to those types are also valid (references are transparent).

Expression groups

Expressions are split into several categories based on where they can occur and their precedence.

```
Expression = UnaryOperation | BinaryOperation ;
PostfixExpression = PrimaryExpression, {Call, IndexAccess, FieldAccess} ;

PrimaryExpression =
    Path
  | Identifier
  | MarcoIdentifier
  | SpecializedIdentifier
  | PrimitiveLiteral
  | ParenthesisedExpression
  | Block
  | PodLiteral
  | TupleLiteral
  | ArrayLiteral
  | LambdaExpression
  | IfExpression
  | SwitchExpression ;
ExpressionWithoutStructs = (*
  same as Expression but cannot include struct literals at the top level *) ;
ExpressionWithoutCallsOrStructs = (*
  same as Expression but cannot include calls or struct literals
  at the top level *) ;
```

`PrimitiveLiteral` expressions are any literal for a primitive type (including strings) given in the lexical elements section.

Paths and identifiers

Paths and identifier expressions are used to refer to entities within a scope.

```
Identifier = identifier ;
MarcoIdentifier = Identifier, "!" ;
SpecializedIdentifier = Identifier, Specializations ;
Specializations = "@", "(", TypeList, ")" ;

(* >= 1 sections e.g. a::b -- a alone normally is an identifier*)
Path = Identifier, {"::", Identifier}, [Specializations] ;
```

- Identifiers can refer to variables, global constants, and functions.
- Macro identifiers are used to call builtin macros.
- Specialized identifiers are used to specialize generic functions or types (in cases where the specializations cannot be inferred).
- Paths are used to access entities within a named scope.
    - Currently, paths are only used to access enum members.

```
foo_bar         # identifier
println!        # marco identifier
new_vec@(i32)   # specialized identifier
Foo::Bar        # path
```

Calls

Given a expression `f` with a callable type F (lambda or function),

```
f(arg1, arg2, … argn)
```

calls `f` with arguments `arg1, arg2, … argn`. The arguments must be assignable to the parameter types of F. Arguments are evaluated in order (left to right) before the function is called. The type of the expression is the return type of F, and its value is passed back to the caller when the function returns.

If the return type of F is a reference type, then the expression can be used as an lvalue.

```
fun index_cats: ref Cat (cats: Cat[], index: i32) {
  cats[index]
}

proc add_angus (my_cats: Cat[], index: i32) {
  # call as lvalue (index_cats returns a reference to Cat)
  index_cats(my_cats, index) = Cat { .name = "Angus", .age = 5 };
```

```
  # normal rvalue call (here returning void -- empty tuple)
  print("Added Angus");
}
```

Blocks

A block is a possibly empty sequence of statements, optionally ending with an expression.

```
Block = "{", [StatementList], [Expression], "}" ;
StatementList = { Statement [";"] } ;
```

If a block ends with an expression, the block has the type and value of that expression, otherwise, the block is void (with the value the empty tuple).

```
{
  x := 10;
  x -= 1;
  x + 1     # ending expression (block has the value 10)
}
```

If expressions

"If" expressions conditionally select a block to execute based on a boolean expression. If the expression is `true` the "then" block (first block) is executed, otherwise, if given, the "else" block is executed.

```
IfExpression = "if", IfCondition, Block, ["else", (IfExpression | Block)] ;
IfCondition = ExpressionWithoutStructs ;
```

If both branches are supplied, they both must evaluate to the same type (which can be void). Then the "if" expression will take the value of the selected block.

```
fun bool_to_str: str (b: bool) {
  # evaluates to the string "true" or "false" depending on b
  if b { "true" } else { "false" }
}
```

If no "else" branch is given then the "then" block must evaluate to void.

Unary operations

```
UnaryOperation = {unary_op}, PostfixExpression ;
```

Operator (unary_op)	Meaning	Definition (given x)	Applicable types
+	plus	`0 + x`	integers, floats
-	minus	`0 - x`	integers, floats

Operator (unary_op)	Meaning	Definition (given x)	Applicable types
~	bitwise not	each bit of x flipped	integers
¬ or !	logical not	`if x { false } else { true }`	booleans
`ref`	take reference	if x is an lvalue take a reference to x	any type

x is any applicable type for the operator.

Note that unary operators can be repeated and mixed:

```
x := -1;    # single unary minus
y := -+-x;  # mutiple unary operations -(+(-(x)))
```

Binary operations

```
BinaryOperation = (Expression, binary_op, Expression)
                | (Expression, "as", Type) ;
```

The second form of binary expression is an as cast, and is described separately.

Operator (binary_op)	Meaning	Definition (given x, y)	Applicable types
+	add	`value_of(x)` + `value_of(y)`	integers, floats, strings
-	minus	`value_of(x)` − `value_of(y)`	integers, floats
*	times	`value_of(x)` × `value_of(y)`	integers, floats
/	divide	`value_of(x)` ÷ `value_of(y)`	integers, floats
%	modulo	integer remainder of `x / y`	integers
<<	left shift	logical left shift of `value_of(x)` by `value_of(y)`	integers
>>	right shift	arithmetic right shift of `value_of(x)` by `value_of(y)`	integers
<	less than	`true` if `value_of(x)` < `value_of(y)` otherwise `false`	integers, floats
>	greater than	`true` if `value_of(x)` > `value_of(y)` otherwise `false`	integers, floats
<=	less than or equal	¬`(x > y)`	integers, floats
>=	greater than or equal	¬`(x < y)`	integers, floats

Operator (`binary_op`)	Meaning	Definition (given x, y)	Applicable types
`==`	equal to	`(x >= y) && (x <= y)`	integers, floats
`!=`	not equal to	`¬(x == y)`	integers, floats
`&`	bitwise and	`0` at all bit positions where x and y differ (keep bit otherwise)	integers
`|!`	bitwise xor	`1` at all bit positions where x and y differ (zero otherwise)	integers
`|`	bitwise or	`1` at all bit positions where at least of one x and y contains a `1`	integers
`&&`	logical and	`if x { y } else { false }`	booleans
`||`	logical or	`if x { true } else { y }`	booleans

`value_of` gives the result of evaluating an expression.

Note:

- for strings: `x + y` is the string concatenation of x and y.
- for integers: `x / y` is the integer division (truncated towards zero).

Operator precedence

Precedence	Operator
100	*, /, %
90	+, -
80	<<, >>
70	<, <=, >, >=
60	==, !=
50	&
40	|!
30	|
20	&&
10	||

Note that postfix expressions always have higher precedence than unary operations, which have always have higher precedence than binary operations. However, unlike binary operations among unary and postfix expressions, all operations/expressions have equal precedence.

Index accesses

Given a with an indexable type (fixed-size array, list, or string)

```
a[idx]
```

accesses the element of a at the index `idx`.

`idx` must be an integer type and if `idx` is out of range at runtime, the program is aborted with an index error.

If a is a fixed-size array or list type:

- the type of `a[idx]` is the element type of the array
- if a is an lvalue, `a[idx]` is an lvalue.

Additionally, if a is a fixed-size array:

- constant indexes must be in range

If a is a string type:

- the type of `a[idx]` is always `char`

Field accesses

Given an expression x

```
x.f
```

accesses the field of the value x given by `f`, where `f` is an identifier.

If x is a fixed-size array, list type, or string:

- `x.length` gives the length as an integer

If x is an enum type:

- `x.tag` gives the unique tag of the enum member x contains as an integer

If x is a pod type:

- `x.<field>` accesses a field as defined the pod's declaration, the access type is the type of the declared field.
- if x is an lvalue, `x.f` is an lvalue.

Accessing non-existent fields is an error.

Tuple literals

A tuple literal constructs a value of a tuple type.

```
TupleLiteral = ("(", [TupleElements], ")") ;
TupleElements = Expression, ",", [Expression, {"," Expression}, [","]] ;
```

The types of the expressions in the tuple literal are used to derive the tuple type:

```
(1, true, 3.0)   # three element tuple -- (i32, bool, f64)
(1, true, 3.0,)  # tuple with trailing comma -- (i32, bool, f64)
((1,3), false)   # nested tuple literal -- ((i32, i32), bool)
(42,)            # single element tuple -- (i32)
(42)             # parenthesised expression (not a tuple)
```

If all members of a tuple are lvalues then the tuple becomes a special lvalue. If used on the left-hand side of an assignment of a tuple with matching types, assigns each lvalue to the corresponding value of the tuple.

```
a := 1;
b := 2;
c := 3;
# a = 1, b = 2, c = 3

(a, b, c) = (100, 200, 300);
# a = 100, b = 200, c = 300
```

Pod literals

A pod literal constructs a value for a pod type or pod enum member.

```
PodLiteral = Path, "{", [FieldInitializerList], "}" ;
FieldInitializerList = FieldInitializer, { ",", FieldInitializer } ;
FieldInitializer = ".", Identifier, "=", Expression ;
```

In a pod literal the order of fields does not matter, however, a pod literal is invalid if it does not provide all the fields from the pod declaration or repeats fields. Additionally, each field initializer's type must match the field type.

Given the following declarations:

```
pod Dog {
  age: i32,
  name: str,
  favorite_toy: DogToy
}

enum DogToy {
  Stick,
  Teddy{ name: str }
}
```

these are valid instantiations:

```
harrold := DogToy::Teddy { .name = "Harrold" };
cassie := Dog { .age = 10, .favorite_toy = harrold, .name = "Cassie" };
woody := Dog { .name "Woody", .age = 6, .favorite_toy = DogToy::Stick };
```

Array literals

An array literal construct a value of a fixed-size array type.

```
ArrayLiteral = "[", [ExpressionList], "]" ;
ExpressionList = Expression { ",", Expression } ;

ArrayRepeatLiteral = "[", "=", Expression, ";", Expression, "]" ;
```

There are two types of array literals:

- Normal list literals
  - For these, there must be at least one element in the array literal, and all elements must have matching types
- Repeat literals
  - These say how many times to repeat (given by the second expression -- which must be constant) an initializer (the first expression)

The type of the array literal is a fixed-size array of the element type, with the size being the number of elements in the literal:

```
a := [];                    # invalid empty literal (unknown type)
a := [1,2,3,4,5];           # i32[5]
a := [[1.1,2.1],[1.2,2.2]]; # f64[2,2]
a := [1,2,true,"hello"];    # invalid literal (mixed element types)
a := [=0; 10];              # i32[10] (repeat literal)
```

Lambda expressions

A lambda expression creates a lambda function with and has a lambda type.

```
LambdaExpression = "\", [LambdaParameterList], "->", [Return], Block ;
LambdaParameterList = {identifier, [TypeAnnotation]} ;
Return = TypeAnnotation ;
```

Type annotations for lambda parameters and return types are optional and can be (in most cases), inferred based on the context and usage of the lambda. If the types cannot be inferred the lambda is invalid.

The type of the lambda is a lambda type, with the corresponding parameter and return types to the lambda expression.

```
add := \x, y -> { x + y };  # untyped lambda that adds to values
result := add(1,2);         # inferred to be \i32, i32 -> i32 based on usage
```

```
sub := \x: f64, y: f64 -> f64 { x - y }; # explicitly type annotated lambda
```

Closures/captures

A lambda can form a closure if it captures its variables/values from its surrounding (lexical) environment. Values are captured by value (contrary to by reference), though existing references in the outer scope can be captured (though this can lead to dangling references if the lambda is returned from a function).

The environment of a lambda with captures is heap-allocated and lives as long as the lambda.

Some simple closures:

```
language_name := "Mank";
display_name := \ -> {
  println!("The {name} Programming Language", language_name);
}
display_name();
```

The above demonstrates a simple capture, the variable `language_name` is declared in the scope outside of the lambda (note that there are no declarations within the lambda). However, the lambda captures the variable from its outer scope, making it able to print it in its body.

Running this snippet will result in: `The Mank Programming Language` being printed.

```
make_adder := \x -> { \y -> { x + y } };
add_10 := make_adder(10);
fifteen := add_10(5); # = 15
```

This is another example that demonstrates closures and returning lambdas from functions. When `make_adder` is called it returns it's nested lambda (`\y -> { x + y }`), which captures the value of `x`. Then when `add_10` is called, the captured `x` is added to the parameter `y` to result in 15.

```
fun make_counter: \ -> i32 (start: i32) {
  count := start - 1;
  \ -> i32 {
    count += 1;
    count
  }
}
```

```
counter := make_counter(4);
counter() # = 4
counter() # = 5
counter() # = 6
```

Lambdas can also mutate their captured variables, as shown with the simple counter above.

As casts

An "as cast" converts values between types. They're the only way to convert between types as there are no implicit conversions.

They're a special form of binary operation.

Allowed (base) casts:

Source type	Target type	Notes
T	T	if the source and target types are the same the cast is a nop
`i32`	`f32`	
`i32`	`f64`	
`f32`	`i32`	fractional part discarded (round down)
`f64`	`i32`	fractional part discarded (round down)
`i32`	`char`	if the integer does not fit within the char the top bits are truncated
`char`	`i32`	the integer is the characters' ASCII value
`bool`	`i32`	1 if the bool is true 0 otherwise
`char`	`str`	creates a new string consisting of a single char

Additional casts are allowable due to the transitive (directed) closure of these base casts. For example, the cast `true as f64` is allowed as you can cast a `bool` to an `i32` then to an `f64`. A cast like `1.2 as bool` would be invalid (since there are no casts from any type to `bool`).

Switch expressions

"Switch" expressions select a block to execute based on which case a value matches.

```
SwitchExpression = "switch", SwitchedExpression, "{", [SwitchCaseList] "}" ;
SwitchCaseList = SwitchCase, { "," SwitchCase }, [","] ;
SwitchCase = MatchExpression, [StructuralBindings], "=>", Block ;
MatchExpression = ExpressionWithoutCallsOrStructs ;
SwitchedExpression = ExpressionWithoutStructs ;
```

"Switch" expressions can be used on integer types along with enums. When used with an integer type the match expression for each case must be a constant expression.

```
# simple switch against an integer:
switch number => {
  1 => { println("one!"); },
  2 => { println("two!"); },
  1 + 2 => { println("three!"); }
}
```

```
# AST for simple s-expressions (+ 1 2 3 4 (* 5 6 4))
enum Expr {
  Number(f64),
  Add { operands: Expr[] },
  Mult { operands: Expr[] }
}
```

When used with enums each case can (optionally) include a structural binding to extract pod or tuple data from the enum member.

```
fun eval_expr: f64 (expr: Expr) {
  switch expr {
    Expr::Number(num) => { num },
    Expr::Add { .operands } => {
      sum := 0.;
      for i in 0 .. operands.length {
        sum += eval_expr(operands[i]);
      }
      sum
    },
    Expr::Mult { .operands } => {
      product := 1.;
      for i in 0 .. operands.length {
        product *= eval_expr(operands[i]);
      }
      product
    }
  }
}
```

Above shows a simple example of a (tagged union) enum being used to represent some S-expressions, which can then be evaluated recursively with a `switch`.

Statements

```
Statement =
    ExpressionStatement
  | VariableDeclaration
  | ReturnStatement
  | AssignmentStatement
  | ForLoop
  | WhileLoop
  | LoopStatement
  | LoopControl
  | StructuralBindingStatement ;
```

Expression statements

These allow expressions to appear in a statement context.

```
(* if the expression ends with a } the semicolon can be omitted *)
ExpressionStatement = Expression, ? ";" ?;
```

For expressions with no observable side-effects this has little purpose, but it's necessary for certain expressions such as ifs and calls.

```
println("Hello World");
1 + 3 + 1; # expression statement without any side-effects (a warning)
```

Variable declarations

Variable declarations introduce a new variable to a scope, binding them to a name, and giving them an initial value.

```
VariableDeclaration = identifier, OptionalTypeAnnotation, "=", Expression, ";";
```

Declarations can be type annotated, though in most cases the type is easily inferred from the initializer.

```
meaning_of_life := 42;
pi: f64 = 4.0;          # explicitly type annotated
```

Return statements

A "return" statement exits a function, optionally providing a return value. The type of the return value must match the function declaration, the return value can only bit omitted for functions that return `void`.

Once a "return" statement is reached, any statements after it won't be executed.

```
ReturnStatement = "return", [Expression], ";" ;
```

```
fun get_cool_number: i32 {
  return 1337;
  println("Yay!");  # unreachable
}
```

Assignments

Assignments change the value of an existing lvalue or lvalues. The type of the assigned expression must be assignable to the lvalue expression's type.

```
AssignmentStatement = LValueExpression, "=", Expression, ";" ;
LValueExpression = Expression ;
```

There is a special case where mutiple lvalues can be assigned if the left-hand side is an lvalue tuple.

```
pi = 3.14;
foo[3] = 100;
dog.name = "James";
(a, b, c) = (1, 2, 3);  # lvalue tuple assignment
```

For loops

"For" loops repeatedly execute a block for every value in a given range (incrementing by 1 in each iteration). The range must have a numeric type.

```
ForLoop = "for", identifier, [TypeAnnotation], "in", ForRange, Block ;
ForRange = ExpressionWithoutStructs ".." ExpressionWithoutStructs ;
```

The ranges are exclusive, so `0 .. 100` is 0, 1, 2 .., to 99, sometimes written as `[0,100)`.

```
# Prints "Hello!" 10 times
for i in 0 .. 10 {
  println("Hello!");
}
```

The loop body must evaluate to `void`.

While loops

"While" loops repeatedly execute a block while a boolean condition holds. The loop body (block) must evaluate to `void`.

```
WhileLoop = "while", WhileCondition, Block ;
WhileCondition = ExpressionWithoutStructs ;
```

```
# 2 to the power 5 with while loops
x := 5;
y := 1;
while x > 0 {
  y *= 2;
  x -= 1;
}
```

Loop statements

Loop statements execute a block forever (unless they encounter a `break` or `return`).

```
LoopStatement = "loop", Block ;
```

```
loop {
  println("It never ends!");
}
```

These are just a convenience and could equivalently be written as:

```
while true {
  println("It never ends!");
}
```

Loop control

Loop control statements allow for the early exit of a loop, or interaction of a loop. It is not valid to use these statements outside of a loop.

```
LoopControl = ("break" | "continue"), ";" ;
```

Inside a loop:

- If a `break` is encountered the control flow will jump to the first statement after the loop
- If a `continue` is encountered the loop will skip to the next iteration
    - If the loop is a "for" loop the loop counter will also be incremented

Both of these only apply to one loop at a time so inside a nested loop, a `break` or `continue` will only effect the inner most loop.

Using these outside of a loop is invalid.

Structural binding statement

A structural binding statement allows elements or fields to be extracted from pod or tuple values, using a structural binding.

```
StructuralBindingStatement = "bind", StructuralBinding, "=", Expression, ";" ;
```

One handy use case for this is to declare multiple variables at a time, using a tuple literal:

```
bind (a, b, c) = (1, true, 3.0);
```

Builtin functions and macros

I/O functions

```
fun putchar: i32 (c: char)
```

Writes a single character `c` to `stdout` and returns the character written if successful and `-1` if not.

```
fun stderr_putchar: i32 (c: char)
```

Same as `putchar` but writes to `stderr`.

```
fun getchar: i32
```

Returns a single character read from `stdin` or `-1` if the read fails.

```
proc print(s: str)
```

Prints the string `s` to `stdout`.

```
proc println(s: str)
```

Prints the string `s` to `stdout` with a newline.

```
proc eprint(s: str)
```

Prints the string `s` to `stderr`.

```
proc eprintln(s: str)
```

Prints the string `s` to `stderr` with a newline.

```
fun input: str
```

Reads a line from `stdin` and returns it as a string (without the newline).

```
fun prompt: str (msg: str)
```

Same as `input` but, if the message (`msg`) is non-empty print it first with a trailing space.

```
fun input_int: i32
```

Attempts to return an integer read from `stdin` (using `input`), if what is read is not an integer the user is asked to try again.

```
fun prompt_int: i32 (msg: str)
```

Same as `input_int` but adds a prompt in the same way as the `prompt` function.

Maths functions

All trigonometric assume their inputs are in radians.

```
fun sqrt: f64 (f: f64)
```

Calculates the square root of `f`.

```
fun pow: f64 (x: f64, y: f64)
```

Calculates `x` raised to the power `y`.

```
fun sin: f64 (f: f64)
```

Calculates the sine of `f`.

```
fun cos: f64 (f: f64)
```

Calculates the cosine of `f`.

```
fun tan: f64 (f: f64)
```

Calculates the tangent of `f`.

```
fun asin: f64 (f: f64)
```

Calculates the inverse sine of `f`.

```
fun atan2: f64 (x: f64, y: f64)
```

Calculates the arc tangent of the argument.

Utility functions

```
fun parse_int: (i32,bool) (s: str)
```

Attempt to parse the given string as an integer. The string is assumed to to be in base 10.

If the parse succeeds:

- the tuple `(n, true)` is returned (where n is the parsed integer)
- otherwise, `(-1, false)` is returned, the second element `false` indicates the parse failed.

```
fun int_to_string: str (i: i32)
```

Returns the (base 10) string representation of the integer `i`.

```
fun str_compare: i32 (a: str, b: str)
```

Lexicographically compares two strings.

The result is negative if $a < b$, zero if $a = b$, and positive if $a > b$.

```
fun str_equal: bool (a: str, b: str)
```

Compares two strings and returns `true` if they are the same and `false` if not.

Vector functions

Vector functions (are currently) special pseudo generic functions. They use the syntax of generics (that are as yet unimplemented) but are internally translated to type erased builtin functions.

For all functions other than `new_vec` the generic types can be inferred.

```
fun (T) new_vec: T[]
```

`new_vec(T)()` constructs a new empty vector (list type) of type T.

```
proc (T) push_back(v: ref T[], e: T)
```

Appends an element e to the back (end) of a vector v.

```
proc (T) pop_back(v: ref T[])
```

Removes the last element from the vector v.

```
proc (T) fill_vec(v: ref T[], e: T, count: i32)
```

Populates the vector v with `count` copies of the element e. If the vector is non-empty it is emptied first.

Exceptional functions

```
proc abort
```

Abnormally terminates the program by raising a SIGABRT signal.

```
proc fail(msg: str)
```

Abort (using abort) the program and provide a message to print to stderr.

Macros

Given a function or lambda f

```
pbind!(f,p1,p2,…,pn)
```

constructs a version of f with the last n parameters of bound (to fixed to a value).

Given a function or lambda f

```
curry!(f)
```

constructs a curried version of f.

The print! macro (and it's friends println!, eprint!, and eprintln!) are used to format and print a string.

The first argument must be a string literal which is the template, then additional parameters replace holes in the template given by {}.

All parameters of a print macro must be strings and the number of holes in the template must match the number of additional parameters.

```
eprintln!("No file named {} in {}", filename, folder);
print!("Hello {name}!", name);  # text inside a hole is ignored
```

The assert! is used to make sure a constraint or invariant has not been broken.

It takes a boolean expression and optionally a string. If the expression evaluates to true nothing happens, however, if it evaluates to false, the assertion fails. If an assertion fails, the filename and line number along with the failing expression and, if provided, the string (message) are printed to stderr.

```
assert!(1 + 1 == 2);
assert!(v.length > 0, "can't pop back on empty vector");
```

The vec! macro is used to construct a populated vector from an array literal.

```
vec!([1,2,3,4])
vec!(["the", "cat", "sat", "on", "the", "mat"])
```

...and they all lived happily ever after!

The End.