MoLa - an interpreted (via VM) programming language featuring modules, garbage collection, and other stuff

Modules

Modules are MoLa programs (files) which can be imported into other programs. Any .mola file can be imported using its location.

When a module gets imported, it is assigned a name under which it will be accessible to the importer program. When we import module A into program B, the following happens:

1. A gets executed normally, without access to B's objects.
2. Objects exported from A are combined into a special module objects. Each exported object gets assigned an identifier under which it can be accessed in that module object.
3. The module structure gets assigned the identifier provided in the `import` statement. From now on, exported objects from A are accessible to B.

A module structure is always stored as a global variable in the importer's environment.

Each program (file) is a module. Such module can be imported using its path in the filesystem. There are 3 types of modules:

1. Modules included in the `MOLA_MODULES_PATH` directory. A module at location `MOLA_MODULES_PATH/a/b/c.mola` is imported usign `import mola.a.b.c as c` statement.
2. Modules that are considered the part of the application. Such modules are usually part of the complete project, and are placed somewhere close to the importer program. Such a module with relative path of `~/a/b/c.mola` (where directory ~ contains the importer program) can be imported using `import a.b.c as c` statement. To go up a directory, the '^' token is used: `import ^^tools`
3. Modules that are somewhere else in the system are imported using the absolute path to the program. A module at location `a/b/c.mola` can be imported using `import "a/b/c.mola" as c` statement.

When a module code is executed, it has access to the things that it would have access if it was executed normally as the main program. This means that globals, functions, and types are directly accessible only within their module. Objects that are not exported become unaccessible to the importer; however, they stay accessible to the imported module.

In an application, a module can only be imported once. This means that next `import` statements will return a new reference to the already created module structure.

Each module has it's own environment. The module structure has a pointer to that environment.

A module structure holds the absolute offset of the first instruction of its code.

Objects

Object are references to values: Object -> Value.

- Objects always have a pointer to their type.
- Value of object X is noted as val(X)
- type object is an object that references a type value, which can be used to create new isntances of that type.
- An object O -> V can be copied in 2 ways:
    1. O` -> V (by reference)
    2. O` -> copy(v) (by value)
- When we copy x using rules defined by its type, we say that it is a copy by auto. we can write it as x -> copyauto(y)
- All builtin types are copied by reference.
- Builtin types are:
    1. immutable, if it is bool, int, float, null, error, or a type value
    2. mutable, if it is string, array, or a custom type
- When we have objects A -> X and B -> Y, then we can make A reference B's value (we say A references B) if we make A -> Y.
- When a reference/object x is removed, we simply make x point to null, and let GC deal with x.
- When a reference/object x is destroyed, we destroy val(x) and let GC deal with x.

Assignment

- Assignment x = y is executed in steps:
    1. if x is an identifier and a variable with such name doesn't exist - create x as an object with value null
    2. create a copy by auto of y: z
    3. make x a reference of z: x -> val(z)
    4. remove reference z
    5. **result: x -> copyauto(y)**
- Assignment by value can be forced using: x copies y
    1. if x is an identifier and a variable with such name doesn't exist - create x as an object with value null
    2. create a copy of y by value
    3. make x a reference of z: x -> val(z)
    4. remove reference z
    5. **result: x -> copy(val(y))**
- Assignment by reference can be forced using: x refs y
    1. if x is an identifier and a variable with such name doesn't exist - create x as an object with value null
    2. create a copy of y by reference
    3. make x a reference of z: x -> val(z)
    4. remove reference z
    5. **result: x -> val(y)**

Functions

- Functions are special objects that can only be created in the global scope (top level) of the module.
- Functions can see itself, global variables, types, and other functions created in that module.
- Functions can accept a fixed amount of arguments.

- Functions always return a value.
- Function arguments are passed to the function in 4 ways:
    1. by value when parameter is defined as: copy x such a parameter becomes a copy by value of the passed argument: arg -> copy(val(x))
    2. by reference when parameter is defined as: ref x such a parameter becomes a copy by reference of the passed argument: arg -> val(x)
    3. by original when parameter is defined as: orig x such a parameter becomes the passed argument, and therefore can: arg = x
    4. by auto when parameter is defined as: x such a parameter becomes a copy by auto of the passed argument: arg -> copyauto(x)
- Function may return a value in 3 ways:
    1. by value using: return copy x a copy by value gets returned: ret -> copy(val(x))
    2. by reference using: return ref x a copy by reference gets returned: ret -> val(x)
    3. by auto using: return x a copy by auto gets returned: ret -> copyauto(x)

Custom types

- Custom types are containers with fields and methods.
- Fields are fixed and new fields cannot be added to the created type object nor the type instance.
- Methods are functions (and have all their features and nuances) that can only be called with the instance object: x.method()
- Methods have access to the instance via "this" keyword.
- Methods have to be defined at the same level where the type is defined.
- Methods are stored in the type, therefore an instance has to access its type and then the method in order to run it.

Runtime errors

Errors in MoLa happen when something goes wrong: division by zero, negative index access in array, and others. An error is an object which contains 4 public fields:

- `code`. An integer uniquely correspoding to the error type.
- `reason`. A string describing the reason why the error has happened.
- `module`. A string containing the path to the module (file) where the error has happened.
- `line`. A line number in the module which caused the error.

However, when an error is uncaught, it prints all locations which are relevant to the error.

The error stack contains pairs (checkpoints): (module path, line number) which are currently executing. When an error is printed, the entire error stack gets printed (beginning with youngest checkpoint, ending with the oldest one) as well. Therefore, the programmer can see what code caused the error, as well as the entire executiong path to that code. The error stack receives new checkpoints when any MoLa instruction is executed. Checkpoints are removed when the respective instructions finish execution.

Basic errors are registered in `std/errors` module and available as integer codes: ZeroDivisionError, NameError, etc. New errors can be registered using the `registerError(defaultReason)` function from the same module. This function returnes an integer code of the newly registered error. Note that the defaultReason must be provided when registering the error; it will be included in the error when the `signal`

statement doesn't provide the `because` part.
An error can be unregistered using the `unregisterError(code)` function

To catch an error, the `try-catch` statement is needed:

```
try {...}
catch CODE1 as NAME {...}   // optional
catch CODE2 as NAME {...}   // optional
...
finally catch AS NAME {...}       // optional
```

When an error happens, the youngest try-catch statement (that covers the code that caused the error) tries to handle it:

1. If there is a `catch` clause with the same error code, then the error object is created and becomes available under name NAME, and the code of that clause is executed. If an error happens in that block, it gets raised outside of the current try-catch statement.
2. Otherwise, if there is a `finally catch` clause, then the error object is created and becomes available under name NAME, and the code of that clause is executed. If an error happens in that block, it gets raised outside of the current try-catch statement.
3. Otherwise, the error is raised (transfered) to the next youngest try-catch statement that covers the code that caused the error.

If the error ends up outside of a try-catch statement, it gets printed to the stderr, and the program ends with error.

Scope

Scope holds variables which can be accessed inside that that scope.

- Scopes have a pointer to the parent scope.
- Scope may or may not be allowed to access to the parent scope.
- Local variables are created in the current scope.
- When an identifier is searched in a scope, the following happens:
    1. If a function with that name exists in the module, it gets returned.
    2. If not 1, and a type with that name exists in the module, it gets returned.
    3. If not 2, and a global variable with that name exists in the module, it gets returned.
    4. If not 3, and a variable with that name exists in the module, it gets returned.
    5. If not 4, and the scope is allowed to access its parent scope, the search happens in the parent scope, starting with step 4.
    6. If not 5, and the scope is not allowed to access its parent scope, a NameError is signalled.
- A new is created in the following situations:
    1. Function call. Parent access is forbidden.
    2. while/for loop body. Parent access is allowed.
    3. if/else body. Parent access is allowed.
    4. Normal block statement (not in 1., 2., 3.). Parent access is allowed.
- Scope is destroyed when a respective code structure ends.

Garbage collection

The GC in MoLa does reference counting, assuming that the number of references to an object fits into a 32-bit unsigned integer. To handle cycles, simple mark-and-sweep cycle is run when there is too much memory allocated.

A checkphase is an event when the GC updates its information about memory usage. A checkphase happens when the Allocator allocates `CHECKPHASE_THRESHOLD` objects. `CHECKPHASE_RATIO` is the ratio between the number of bytes allocated (and not deallocated) since the last checkphase and the number of bytes allocated (and not deallocated) between the second last and last checkphases.

When an object's reference counter reaches 0, it becomes garbage, and therefore has to be destroyed. The GC pushes this object onto the garbage stack.

Each reference update causes destruction of `DESTRUCTION_COUNT * CHECKPHASE_RATIO` objects from the garbage stack.

During the checkphase, the GC decides that it needs to run a collection cycle if:

- new `CHECKPHASE_RATIO` is greater than `GC_CYCLE_THRESHOLD`.
- or the collection cycle hasn't been run for `GC_CYCLE_CHECKSPHASES_LIMIT` checkphases.

If a decision to run a collection cycle is taken, the GC does the following:

1. destroys all objects on the garbage stack.
2. runs a simple mark & sweep cycle to break reference cycles.

To implement this, the GC maintains a set of objects that have been created. When an object is destroyed, it is removed from this set. When a GC collection cycle is run:

- The roots are globals and locals (objects reachable from any active scope, the objects stack, or any other similar source). Functions and types are always alive do not reference any objects other then themselves, and therefore they are not processed during garbage collection. The roots are taken from all loaded modules.

Environment

An environment is a data structure assigned to each module. It holds all the variables created in that module, and implements lookup function.

An environment also contains all exported objects of the module, and lets other modules use those objects.

Global variables

Global variables are stored in the module's environment.

VM

To execute a program, it first has to be converted to a list of simple instructions, which will be then executed by the VM.

An instruction is an structure that contains details of how some action has to be executed. A MoLaVM instruction is made of:

- Instruction code. Denoted as `inscode(ins)`.
- The name of the file which caused the instruction to be generated. Denoted as `insfile(ins)`.
- The line number in the file which caused the instruction to be generated. Denoted as `insline(ins)`.
- Arguments of the instruction, packed into it during its creation. Denoted as `insarg_i(int)`, where i is the index of the argument.

Instructions of a module are stored in a single list. Instructions from new modules are appended to the end of the list. A module structure stores the absolute offset of the first instruction of that module in the list.

An instruction may access the arguments stack. By convention:

- "Runtime" arguments of the instruction have to be pushed on the stack.
- **Any instruction that requires arguments takes them and removes them from the stack before executing actual operation. This way memory leaks are prevented when returning / catching an error.**
- The execution result of the instruction has to be pushed on the stack, unless it is absent.

Structures:

1. Objects stack (arguments stack). Contains objects that are indended to be used as arugments in the instructions.
2. A stack of registered error handlers. Each handler (a catch block) is a tuple of:
    - error code.
    - absolute offset (of the catch block) to where the program has to jump.
    - the number of elements on the stack before the try block was executed. the stack is rollbacked to this state.
    - the current scope pointer.

The following is the list of MoLaVM instructions with their descriptions.

`POP`

- The stack must contain an object. Pops an object from the stack. Signals an InternalError if the stack is empty.

`SWAP`

- The stack must contain 2 objects.

Swaps two top elements on the stack. Signals an InternalError if there are less than 2 objects on the stack.

`CREATE_ENV`

Creates a new environment. The environment gets assigned an id, which is later used to switch to that environment. This id is assigned during runtime. The root program get's assigned the id of 0.

`SWITCH_ENV_INS`

Switches the current environment to the environment of the module where this instruction is located.

`SWITCH_ENV_OBJ`

- The stack must contain an object which is a function. Switches the current environment to the environment of the function located on the stack.

`IMPORT_MODULE module_path identifier`

- `module_path` is the path to the module we want to import.
- `identifier` is the name under which we want to import the module.

Imports a module.

1. Checks whether `module_path` is correct. Otherwise signals ImportError.
2. Checks whether there is no global variable with the name `identifier` in the current environment. If there is, signals NameCollisionError.
3. Checks whether the module at `module_path` has already been imported (directly or indirectly). If such, `identifier` becomes a reference to that module structure, and the instruction terminates.
4. Generates instructions and appends them to the end of the instructions list. All generated instructions receive a new env_id.
5. Executes the instructions of the module in the new environment.
6. Creates a module structure containing the environment under the name of `identifier`, as well as the absolute offset to the instructions in the imported module.

`EXPORT_OBJECT identifier`

- `identifier` is the name under which we want to export the object on the stack.
- the stack must contain an object.

Exports the objects on the stack under the name `identifier` by adding it to the current environment. Throws an InternalError if the stack is empty. If there's already an object exported with the given name, throws NameCollisionError.

`CREATE_GLOBAL identifier`

- `identifier` is the name of the global variable we want to create.

Creates a global variable in the current environment, initially with value `null`. Throws NameCollisionError if there's already an accessible object with the same name in the environment.

`CREATE_FUNCTION identifier mode_1 arg_1 ... mode_n arg_n`

- `identifier` is the name of the function.
- `mode_i` is the mode of the i-th argument (copy, ref, pass, or auto).
- `arg_i` is the name of the i-th argument.

Creates a new function object under the given name. The object will have a list of modifiers and argument names, which will be used when calling the function. The function objecmts contains 2 values:

1. (Absolute) position of the second next instruction (after this one) in its module.
2. A pointer to the module structure (which contains the absolute module offset).

Signals a NameCollisionError if an accessible object with the given name already exists. Signals a DuplicateNameError if there are duplicate argument names.

`CREATE_TYPE identifier f m name_1 ... name_n`

- `identifier` is the name of the type.
- `f` is the number of fields.
- `m` is the number of methods.
- `name_i` is the name of the i'th member of the type. These are listed in the order of: fields, methods.

Creates a new type under the given name. The type object contains the information about what fields and methods it has.

Signals a NameCollisionError if an accessible object with the given name already exists, or a DuplicateNameError, if there are duplicate field/method names.

`CREATE_METHOD identifier mode_1 arg_1 ... mode_n arg_n`

- `identifier` is the name of the function.
- `mode_i` is the mode of the i-th argument (copy, ref, or pass).
- `arg_i` is the name of the i-th argument.
- the stack must contain an object.

Creates a new method of the object on the stack.

Note: 'this' special argument has to be passed as a normal argument; it is not automatically added to the arguments list by this instruction.

The method object will have a list of modifiers and argument names, which will be used when calling the function. The object will also have a pointer to the next instruction (after the current one), which is where the method body will be generated. The type will receive a pointer to this method.

Signals an InternalError if the stack is empty. Signals a ValueError the object on the stack is not a type object. Signals a NameCollisionError if the method has already been defined. Signals an NameError if the type object didn't declare a method with such name. Signals an DuplicateNameError if there are duplicate argument names.

Note: this is used to create constructor / destructor as well. In those cases, `identifier` becomes `constructor` / `destructor` respectively.

`CREATE_SCOPE access_mode`

- `access_mode` is `WITH_PARENT_ACCESS`, if the scope should have access to its parent scope, and `WITHOUT_PARENT_ACCESS` otherwise.

Creates a new scope. The access to the parent scope is determined by the `access_mode` argument. If there is no parent scope, `access_mode` is disregarded.

`DESTROY_SCOPE`

Destroys the current scope. Signals an InternalError if there is no scope to destroy.

`JUMP_IF_FALSE offset`

- `offset` is the relative offset of the instruction where the VM has to jump if the value on the stack is `false`.
- the stack must contain an object.

If the object on the stack is `false`, jumps to the instruction at the position `offset` relative to the current instruction.

Signals a ValueError if the object on the stack is not boolean, InternalError if the stack is empty.

`JUMP offset`

- `offset` is the relative offset of the instruction where the VM has to jump.

Jumps to the instruction at the position `offset` relative to the current instruction.

`RETURN`

- the stack must contain an object.

Jumps to the instruction at the absolute offset given in the integer object on the stack.

Signals a ValueError if the object on the stack is not integer, InternalError if the stack is empty.

`REGISTER_CATCH`

- the stack must contain an object.

Registeres a new catch statement with the error code provided on the stack. If the value on the stack is -1, an universal handler is registered.

1. Checks whether the object on the stack is an integer, and takes its value.
2. Inserts a new tuple of (integer error code, absolute offset of the second next instruction after the current one, number of elements on the stack, current scope pointer) on the error handlers stack.

Signals InternalError if there is no object on the stack, ValueError if the object on the stack is not an integer.

`DESTROY_CATCH n`

Removes n tuples error handlers from the errors stack. Signals an InternalError if the errors stack contains less than n tuples

`SIGNAL_ERROR`

- the stack must contain 2 objects: bottom->top: ... s x

Signals an error with code x and message s.

1. Takes the object from the stack, extracting the error code.
2. Goes through the error handlers stack. If it doesnt locate a handler with the given error code, the program terminates.
3. Otherwise it destroys the current scope until it finds the right one. Then it jumps to the instruction at the absolute offset.
4. An error object is pushed on the stack.

Signals an InternalError if the stack is empty, or ValueError if the object x on the stack is not an integer, or s is not a string.

`CREATE_VAR identifier`

- the stack must contain an object.

Creates a new variable with name `identifier`, pointing to the object on the stack.

Signals a NameCollisionError if an accessible object with the given name already exists, InternalError if the stack is empty.

`COPY_BY_VALUE`

- the stack must contain an object.

Copies by value the object from the stack, and pushes that created copy on the stack.

Signals an InternalError if there is no object on the stack.

`COPY_BY_REFERENCE`

- the stack must contain an object.

Copies by reference the object from the stack, and pushes that created copy on the stack.

Signals an InternalError if there is no object on the stack.

`COPY_BY_AUTO`

- the stack must contain an object.

Copies by auto the object from the stack, and pushes that created copy on the stack.

Signals an InternalError if there is no object on the stack.

`ASSIGNMENT`

- the stack must contain 2 objects: bottom->top : ... object2 object1

Replaces the pointer Object1->Value1 to Object1->Value2. Object2 gets removed from the stack, Object1 stays.

Signals an InternalError if there are less than 2 objects on the stack.

LOGICAL_OR

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x or y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

LOGICAL_AND

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x and y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

BITWISE_OR

- the stack must contain 2 objects: bottom->top: ... x y

Creates an integer object with the value of (x | y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

BITWISE_XOR

- the stack must contain 2 objects: bottom->top: ... x y

Creates an integer object with the value of (x ^ y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

BITWISE_AND

- the stack must contain 2 objects: bottom->top: ... x y

Creates an integer object with the value of (x & y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

EQUAL

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x == y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack.

NOT_EQUAL

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x != y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack.

LESS_THAN

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x < y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack.

LESS_EQUAL

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x <= y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack.

GREATER_THAN

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x > y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack.

GREATER_EQUAL

- the stack must contain 2 objects: bottom->top: ... x y

Creates a boolean object with the value of (x >= y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack.

ADDITION

- the stack must contain 2 objects: bottom->top: ... x y

Creates an object with the value of (x + y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

SUBTRACTION

- the stack must contain 2 objects: bottom->top: ... x y

Creates an object with the value of (x - y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

LSHIFT

- the stack must contain 2 objects: bottom->top: ... x y

Creates an integer object with the value of (x << y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

RSHIFT

- the stack must contain 2 objects: bottom->top: ... x y

Creates an integer object with the value of (x >> y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

MULTIPLICATION

- the stack must contain 2 objects: bottom->top: ... x y

Creates an object with the value of (x * y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

DIVISION

- the stack must contain 2 objects: bottom->top: ... x y

Creates an object with the value of (x / y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

REMAINDER

- the stack must contain 2 objects: bottom->top: ... x y

Creates an object with the value of (x % y). The object is pushed on the stack.

Signals an InternalError if there are less than 2 objects on the stack. Signals a ValueError if the objects have invalid values.

NEGATION

- the stack must contain 1 object: bottom->top: ... x

Creates an object with the value of -x. The object is pushed on the stack.

Signals an InternalError if there are less than 1 object on the stack. Signals a ValueError if the objects have invalid values.

`INVERTION`

- the stack must contain 1 object: bottom->top: ... x

Creates an object with the value of ~x. The object is pushed on the stack.

Signals an InternalError if there are less than 1 object on the stack. Signals a ValueError if the objects have invalid values.

`LOGICAL_NOT`

- the stack must contain 1 object: bottom->top: ... x

Creates an object with the value of not x. The object is pushed on the stack.

Signals an InternalError if there are less than 1 object on the stack. Signals a ValueError if the objects have invalid values.

`CALL n`

- the stack must contain n + 1 object: bottom->top: ... f arg1 arg2 ... argn

Calls function f.

1. Checks whether the amount of arguments matches the amount expected by f. Raises an WrongNumberOfArgumentsError if it doesn't match.
2. Adds new local variables to the scope:
    - for i-th argument, a copy is created based on the i-th mode specified in f.
    - that copy is added to the scope under the i-th name stored in f.
3. The address of the next instruction is pushed on the stack as the return address.
4. Jumps to the address stored in the function object.

`ACCESS`

- the stack must contain 2 objects: bottom->top: ... x y

Returns x[y].

Signals an InternalError if there are less than 2 objects on the stack, ValueError if y is not integer, OutOfBoundsError if y is out of bounds.

`LOAD_BOOL value`

Creates and pushes a boolean object with value = `value` on the stack.

`LOAD_CHAR value`

Creates and pushes a character object with value = `value` on the stack.

`LOAD_INT value`

Creates and pushes an integer object with value = `value` on the stack.

`LOAD_FLOAT value`

Creates and pushes a float object with value = `value` on the stack.

`LOAD_STRING value`

Creates and pushes a string object with value = `value` on the stack.

`LOAD_NULL`

Creates and pushes a null object on the stack.

`LOAD identifier`

- `identifier` is the name of the object we want to locate.

`LOAD_FIELD identifier`

- The stack must contain an object

Finds the field with name `identifier` of the object on the stack, and pushes it on the stack. Signals InternalError if the stack is empty, NameError if the object on the stack doesn't have a field with the given name.

`LOAD_METHOD identifier`

- The stack must contain an object

The object on the stack is put in the special "caller" register. Finds the method with name `identifier` of the object on the stack, and pushes it on the stack. Signals InternalError if the stack is empty, NameError if the object on the stack doesn't have a method with the given name.

`NEW n`

- The stack must contain n+1 objects: bottom->top: t a_1 ... a_n

Creates an instance of the given type t, passing arguments a_1 ... a_n into the constructor.

Raises an InternalError if there are less than n+1 objects on the stack, ValueError if t is not a type object, and other errors that could happen when calling the constructor (which is a function).

MoLa code -> MoLaVM instructions

Some notations:

- `gen(x)` means taking x and converting it into a list of instructions. The instructions are returned.

Instructions have a number preceding them. This is used as a marker for the position of certain instructions. It is neither relative nor absolute; its only purpose is to mark the instructions in order.

If we want to specify that a jump instruction has to jump at a specific place, we can use the number of the line, preceding it with REL or ABS: JUMP REL@3 or JUMP ABS@3

Each program is started with a `CREATE_ENV` instruction.

`'import' module_path 'as' indetifier`

```
1   IMPORT_MODULE module_path identifier
```

`'export' expr_1 'as' name_1, ..., expr_n 'as' name_n`

```
#   gen(expr_1)
1   EXPORT_OBJECT name_1
#   gen(expr_2)
2   EXPORT_OBJECT name_2
    ...
#   gen(expr_n)
n   EXPORT_OBJECT name_n
```

`'global' identifier`

```
1   CREATE_GLOBAL identifier
```

`'function' identifier '(' mode_1 arg_1, ..., mode_n arg_n ')' block_stmt`

```
1   CREATE_FUNCTION identifier mode_1 arg_1 ... mode_n arg_n
2   JUMP REL4
#   gen(block_stmt)
3   RETURN
4
```

`'type' identifier '{' f_1 ... f_n m_1 ... m_k'}'`

```
1   CREATE_TYPE identifier n k f_1 ... f_n m_1 ... m_k
```

'method' name '(' mode_1 arg_1, ..., mode_n ')' 'of' type_name block_stmt

```
1   LOAD type_name
2   CREATE_METHOD name 'ref' 'this' mode_1 arg_1 ... mode_n arg_n
3   JUMP REL@5
#   gen(block_stmt)
4   RETURN
5
```

'constructor' '(' mode_1 arg_1, ..., mode_n ')' 'of' type_name block_stmt

```
1   LOAD type_name
2   CREATE_METHOD 'constructor' 'ref' 'this' mode_1 arg_1 ... mode_n arg_n
3   JUMP REL@5
#   gen(block_stmt)
4   RETURN
5
```

'destructor' 'of' type_name block_stmt

```
1   LOAD type_name
2   CREATE_METHOD 'destructor' 'ref' 'this'
3   JUMP REL@5
#   gen(block_stmt)
4   RETURN
5
```

'{' block_stmt '}'

```
1   CREATE_SCOPE WITH_PARENT_ACCESS
#   gen(block_stmt)
3   DESTROY_SCOPE
```

while expr stmt

```
1   gen(expr)
2   JUMP_IF_FALSE REL@4
#   gen(stmt)
3   JUMP REL@1
4
```

`'for' expr1? ';' expr2? ';' (expr3 | ';') stmt`

if expr2 is present:

```
1   CREATE_SCOPE WITH_PARENT_ACCESS
#   gen(expr1?)
2   gen(expr2)
3   JUMP_IF_FALSE REL@4
#   gen(stmt)
#   gen(expr3?)
#   JUMP REL@2
4   DESTROY_SCOPE
```

if expr2 is not present:

```
1   CREATE_SCOPE WITH_PARENT_ACCESS
#   gen(expr1?)
2   gen(stmt)
#   gen(expr3?)
#   JUMP REL@2
3   DESTROY_SCOPE
```

`'if' expr stmt1 ('else' stmt2)?`

If else part is present:

```
#   gen(expr)
1   JUMP_IF_FALSE REL@3
#   gen(stmt1)
2   JUMP REL@4
3   gen(stmt2)
4
```

If else part is absent:

```
#   gen(expr)
1   JUMP_IF_FALSE REL@2
#   gen(stmt1)
2
```

`continue`

```
1   DESTROY_SCOPE
#    ...
k   DESTROY_SCOPE
#   JUMP REL@X
```

Continue is meaningful only in 2 cases: (innermost loops that contain continue)

```
while ...           // X is after the condition expression
    ...
    {... {...
    continue;
    ...} ...}
    ...
```

```
for ...             // X is after the step expression
    ...
    {... {...
    continue;
    ...} ...}
    ...
```

In these cases, continue must destroy all the scopes where it is contained. `k` is the number of such open scopes.

Otherwise, the instruction is ignored.

`break`

```
1   DESTROY_SCOPE
#    ...
k   DESTROY_SCOPE
#   JUMP REL@X
```

Break is meaningful only in 2 cases: (innermost loops that contain continue)

```
while ...           // X is after the condition expression
    ...
    {... {...
    break;
    ...} ...}
    ...
// X is here
```

```
for ...
    ...
    {... {...
    break;
    ...} ...}
    ...
// X is here
```

In these cases, continue must destroy all the scopes where it is contained. `k` is the number of such open scopes.

Otherwise, the instruction is ignored.

`'return' ('copy' | 'ref' | 'pass' | ) expr`

The stack must contain the return address.

The return statement only makes sense in one case:

```
function f(...) {
    ...
    {... {...
    return ...
    ...} ...}
    ...
}
```

K is the number of open scopes that contain the return statement. On all other situations return is ignored.

If mode is `copy`:

```
#   DESTROY_SCOPE
    ...
K   DESTROY_SCOPE
#   gen(expr)
1   COPY_BY_VALUE
2   SWAP
3   RETURN
```

If mode is `ref`:

```
#   DESTROY_SCOPE
    ...
K   DESTROY_SCOPE
#   gen(expr)
1   COPY_BY_REFERENCE
```

```
2   SWAP
3   RETURN
```

If mode is `pass`:

```
#   DESTROY_SCOPE
    ...
K   DESTROY_SCOPE
#   gen(expr)
1   SWAP
2   RETURN
```

If mode is absent:

```
#   DESTROY_SCOPE
    ...
K   DESTROY_SCOPE
#   gen(expr)
#   COPY_BY_AUTO
1   SWAP
2   RETURN
```

`expr1 ('copies' | 'refs') expr2`

If mode is `copies`:

```
#   gen(expr2)
1   COPY_BY_VALUE
#   gen(expr1)
2   ASSIGNMENT
3   POP
```

If mode is `refs`:

```
#   gen(expr2)
1   COPY_BY_REFERENCE
#   gen(expr1)
2   ASSIGNMENT
3   POP
```

`'try' block_stmt ('catch' expr 'as' identifier block_stmt)* ('catch' '*' 'as' identifier block_stmt)?`

Error handlers are generated in reversed order. For a specific error handler (`catch expr as identifier block_stmt`), the following is generated:

```
#   gen(expr)
0   REGISTER_CATCH
1   JUMP REL@6
2   DESTROY_CATCH 1
3   SWITCH_ENV
4   CREATE_VAR identifier
#   gen(block_stmt)
#   DESTROY_CATCH *total handlers - pos of this one*
5   JUMP REL@*first instruction after the try-catch statement*
6
```

For a universal error handler (`catch * as identifier block_stmt`), the following is generated:

```
#   LOAD_INT -1
0   REGISTER_CATCH
1   JUMP REL@6
2   DESTROY_CATCH 1
3   SWITCH_ENV
4   CREATE_VAR identifier
#   gen(block_stmt)
5   JUMP REL@*first instruction after the try-catch statement
6
```

After the catch statements, the try block is generated:

```
#   gen(try_block)
#   DESTROY_CATCH *number of catch parts*
```

For each catch clause, a DESTROY_CATCH instruction is generated.

For example, code:

```
try try_stmt
catch expr1 as name1 catch_stmt1
catch expr2 as name2 catch_stmt2
catch * as name3 catch_stmt3
```

will be compiled into:

```
#   LOAD_INT -1
#   REGISTER_CATCH
```

```
  #   JUMP REL@1
  #   DESTROY_CATCH 1
  #   SWITCH_ENV
  #   CREATE_VAR name3
  #   gen(catch_stmt3)
  #   JUMP REL@4

  1   gen(expr2)
  #   REGISTER_CATCH
  #   JUMP REL@2
  #   DESTROY_CATCH 1
  #   SWITCH_ENV
  #   CREATE_VAR name2
  #   gen(catch_stmt2)
  #   DESTROY_CATCH 1
  #   JUMP REL@4

  2   gen(expr1)
  #   REGISTER_CATCH
  #   JUMP REL@3
  #   DESTROY_CATCH 1
  #   SWITCH_ENV
  #   CREATE_VAR name1
  #   gen(catch_stmt1)
  #   DESTROY_CATCH 2
  #   JUMP REL@4

  3   gen(try_stmt)
  #   DESTROY_CATCH 3
  4
```

'signal' expr1 'because' expr2

```
  #   gen(expr2)
  #   gen(expr1)
  #   SIGNAL_ERROR
```

'var' identifier '=' expr

```
  #   gen(expr)
  #   COPY_BY_AUTO
  #   CREATE_VAR identifier
```

A = B

```

```

```
#   gen(B)
#   COPY_BY_AUTO
#   gen(A)
#   ASSIGNMENT
```

`'when' A 'then' B 'else' C`

```
#   gen(A)
#   JUMP_IF_FALSE REL@1
#   gen(B)
#   JUMP REL@2
1   gen(C)
2
```

`'new' A '(' B_1, ..., B_n ')'` (clarify)

```
#   gen(A)
#   gen(B_1)
    ...
#   gen(B_n)
#   NEW n
```

All other operators get converted in the following way: `A op B`

```
#   gen(A)
#   gen(B)
#   INS
```

where INS is the instruction that executed operation op

`A '(' B_1, ..., B_n ')'

```
#   gen(A)
#   gen(B_1)
    ...
#   gen(B_n)
#   SWITCH_ENV *env of the function A*
#   CREATE_SCOPE WITHOUT_PARENT_ACCESS
#   CALL n
#   DESTROY_SCOPE
#   SWITCH_ENV
```

`A '.' name`

```
#   gen(A)
#   LOAD_FIELD name
```

`A ':' name`

```
#   gen(A)
#   LOAD_METHOD name
```

Builtin functions

Builtin functions (like print, cppload) are function objects that point to a C++ function. Such a function, when executed, runs the C++ function, and returns the result.

Builtin functions are stored in a separate module, which is implicitly imported into any other program.

C++ extensions

MoLa programs can load C++ functions, binding them to MoLa functions. To load such a function, `cppload` builtin function is used.

When a C++ function is loaded(dynamically), a new function object is created that points to that function (just like with the builtin functions). Loaded C++ functions cannot be unloaded.