LLL Compiler Documentation Documentation

Release 0.1

Ben Edgington

Sep 16, 2017

Contents:

Note: This documentation is a work in progress, so please exercise appropriate caution. It a personal effort and has no formal connection with the Ethereum Foundation.

Note: Everything in these docs pertains to the Solidity/LLL implementation, specifically the *develop* branch.

CHAPTER 1

LLL Introduction

Note: This documentation is a work in progress, so please exercise appropriate caution. It a personal effort and has no formal connection with the Ethereum Foundation.

Note: Everything in these docs pertains to the Solidity/LLL implementation, specifically the *develop* branch.

Background

LLL is one of the original Ethereum smart contract programming languages and provides a different perspective and programming discipline when compared to the ubiquitous Solidity language.

According to the Ethereum Homestead Documentation,

> Lisp Like Language (LLL) is a low level language similar to Assembly. It is meant to be very simple and minimalistic; essentially just a tiny wrapper over coding in EVM directly.

In particular, LLL doesn't hide from you the highly resource-constrained nature of the EVM and enables efficient use of those limited resources. LLL facilitates the creation of very clean EVM code whilst removing the worst of the pain of coding for the EVM directly: namely stack management and jump management.

These pages aim to provide a reference resource for LLL contract development.

While these pages are based on the Solidity/LLL implementation, there is also a Viper/LLL implementation. Some of the info here may carry across; some may not.

Resources

The list of LLL-related resources currently available is quite short.

Authoritative Resources

The sole authoritative resource on LLL is the compiler source code itself.

(While this documentation aims to be accurate, there will certainly be errors and omissions.)

Tutorials

Daniel Ellison has put together a number of tutorial articles and screencasts on getting started with LLL.

- A seven part series of articles entitled The Resurrection of LLL.
- An ongoing series of articles on the Consensys media pages with links to screencasts for some of them.

Original Resources

The original LLL documentation is still available on GitHub and remains the starting point for this documentation set. However, that documentation was last updated in 2014, and significant things have changed since then.

Example Code

- The deployed Ethereum Name Service Registry was written in LLL.
- There is also a sample ENS Resolver in LLL.
- An ERC20 token implementation in LLL.
- The compiler built-in macros and end-to-end test suite are also useful references.
- Once again, Daniel Ellison has some code examples and demonstrations of useful techniques.

Warning some of the following examples may use features that have changed or been removed. Some examples may no longer compile.

- The original LLL Examples for PoC 5.
- A GavCoin contract.

CHAPTER 2

Language Reference

Note: This documentation is a work in progress, so please exercise appropriate caution. It a personal effort and has no formal connection with the Ethereum Foundation.

Note: Everything in these docs pertains to the Solidity/LLL implementation, specifically the *develop* branch.

LLL Syntax

Expressions

Influenced by Lisp, everything in LLL should be thought of as expressions to be evaluated, rather than instructions to be executed. To quote Wikipedia on Lisp, "The interchangeability of code and data gives Lisp its instantly recognizable syntax."

An LLL expression is any of the following.

- An integer, optionally prefixed with "0x" for hex base: `42` or `0x2a`. Negative numbers cannot be directly input. (You can do `(sub 0 N)` or use the ~ *bitwise not operator*.)
- A string (see *rules for strings* below).
- An atom that has been defined previously via an argument-less `def` expression. E.g. with an existing definition `(def 'm (memsize))`, then `m` can be used as an atom—an expression without parentheses—which will expand to `(msize)`.
- An evaluated expression which takes the form of a parenthesised list of an operator followed by zero or more expressions which are the operands, `(OP EXPR1 EXPR2 ...)`, e.g. `(add 1 2)`. The operator `OP` is either a built-in *EVM opcode*, or a *parser expression*, or a macro defined in terms of these.

The above items are the basic syntax, and the last of these explains why parentheses are everywhere in LLL. There are some variations to the syntax introduced by the compact form described below.

Expressions evaluate either to "void", or to a single value that is left on top of the EVM stack, and can be worked on by further expressions:

```
(mstore 0x00 42) ;; Evaluates to "void"; leaves nothing on the stack
(mload 0x00)     ;; Evaluates to 42.
```

Compact notation

A compact form is available for certain expressions, particularly memory accesses. Used wisely it can enhance the readability of the code. Using the compact forms does not affect the code generated; the parser simply makes the substitutions below. Note that in all cases any whitespace is optional.

`{ ... }` `(seq ...)` - evaluates a sequence of expressions in order.

`@ EXPR` `(mload EXPR)` - loads the value of memory location EXPR.

`@@ EXPR` `(sload EXPR)` - loads the value of storage location EXPR.

`[ EXPR1 ] EXPR2` `(mstore EXPR1 EXPR2)` - writes the value of EXPR2 into memory location EXPR1. A colon, `:`, may optionally be added after the colsing bracket if desired for readability.

`[[ EXPR1 ]] EXPR2` `(sstore EXPR1 EXPR2)` - writes the value of EXPR2 into storage location EXPR1. A colon, `:`, may optionally be added after the closing bracket if desired for readability.

`$ EXPR` `(calldataload EXPR)` - reads a word of call data starting from byte EXPR.

Strings

A string in LLL is represented internally as a single 32 byte word, with the string left-aligned to the high-order byte in the (big-endian) word. Any bytes beyond the 32nd are ignored by the compiler. Note that a character may be represented by more than one byte since text is UTF-8 encoded by the compiler. Permissible characters in strings vary according to which of the representations is used.

There are two ways to represent strings in LLL:

1. As a fully quoted string: `"Hello, world!"`. Note that there is no escaping mechanism to allow a double quote character, `"`, to be included within the string. With that sole exception, all characters are OK including whitespace and newlines.
2. With a single quote at the beginning: `'forty-two`. The string may not include any whitespace characters, or the characters used in compact notation, parentheses or comments (`{`, `}`, `[`, `]`, `@`, `$`, `(`, `)`, `:`, `;`). However, both single and double quotes *are* allowed. `'forty-two` is identical to `"forty-two"` as far as the parser is concerned.

The following are all valid strings, also shown with their internal hexadecimal representations:

```
"Hello, world!"
0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000000

"$£¥€ - {}[]@():;"
0x24c2a3c2a5e282ac202d207b7d5b5d4028293a3b0000000000000000000000000000

'forty-two
0x666f7274792d74776f0000000000000000000000000000000000000000000000000000

'"forty-two"
0x22666f7274792d74776f2200000000000000000000000000000000000000000000000000
```

```
'
0xe38193e38293e381abe381a1e381afe4b896e7958c0000000000000000000000
```

Comments

A comment begins with a semicolon, `;`, and finishes at the end of the line. In common with Lisp, varying numbers of semicolons may be used to begin a comment depending on context, but this is purely stylistic.

The only exception is when the `;` character is within a double-quoted string, in which case it is treated as part of the string and does not begin a comment.

Common conventions

LLL code is generally written with everything in lower case, except in *asm* expressions where the EVM opcodes tend to be uppercased. Internally for parsing, all expressions are converted to uppercase except for literals/strings.

Literals, such as definition names created by `def` expressions, are case sensitive, so `(def 'x 42)` and `(def 'X 42)` define two distinct macros.

EVM Opcodes

Almost all valid EVM opcodes are automatically valid LLL operations, so expressions like the following will work as standard:

```
(add 1 2)
(signextend 1 0xff00)
(return 0x00 0x20)
(calldatasize)
(call allgas to value 0 0 0 0)
```

In terms of the EVM, the operands can be considered as items on the stack (the left-most being at the top of the stack) upon which the EVM operation works.

The opcodes implmented by the EVM are listed in libevmasm/Instruction.cpp and described in detail in the Ethereum Yellow Paper.

In general, LLL is intended to protect the programmer from having to deal with tedious stack manipulation, and directly operating on the stack is discouraged.

Nonetheless, `pop` creates a valid expression— `(pop EXPR)` —and it can be used to throw away the result of its argument expression. This could be used with `raw` to fine-tune which of the sub-expressions provides the final value of the `raw` expression, for example.

`pushN` is not available as an LLL expression. Literals like `42` are automatically pushed to the stack, so this instruction is not necessary. Other stack manipulation operations such as `swapN` and `dupN` are available but best avoided.

`jump`, `jumpi` and `jumpdest` are also best avoided as there is no easy way to compute the address of a `JUMPDEST`. Better to make use of the LLL control structures provided. However, a "throw" as it was originally implemented in Solidity (a jump to a non-valid address) can be performed with `(jump 0x00)`, as long as you can be sure that 0x00 is not a valid `JUMPDEST`. Nonetheless, the currently recommended way to do this is with `(panic)`; Solidity has now deprecated `throw`.

Parser expressions

In addition to the EVM opcodes, the the LLL parser provides a number of other operators for convenience.

Arithmetic Operators

Multi-ary

The following arithmetic operators can take one or more arguments.

+ `(+ 1 2 3 4 5)` evaluates to 15.

- `(- 1 2 3 4 5)` evaluates to -13.

***** `(* 1 2 3 4 5)` evaluates to 120.

/ `(/ 60 2 3)` evaluates to 10.

% - modulus operation. `(% 67 10 3)` evaluates to 1, i.e. (67%10)%3.

& - bitwise *and*. `(& 15 6 4)` evaluates to 4.

| - bitwise *or*. `(| 4 5 6)` evaluates to 7.

^ - bitwise *xor*. `(^ 1 2 3)` evaluates to 0.

When only one argument is provided then the expression evaluates to the value of that argument. I.e. `(/ 5)` evaluates to 5.

Binary

Binary comparison operators are available with the usual meanings: `<` , `<=` , `>` , `>=` , `=`, `!=`. If the comparison is true then they evaluate to 1: `(< 4 5)` -> 1. If the comparison is false they evaluate to 0: `(> 4 5)` -> 0.

Note that `<` , `<=` , `>` , `>=` all perform unsigned comparisons. So, `(> 1 (- 0 1))` evaluates to false, for example, which may be unexpected.

In addition, there are four signed comparison operators: `S<` , `S<=` , `S>` , `S>=`. Thus, `(S> 1 (- 0 1))` evaluates as true.

Unary

`~` is a bitwise not, corresponding to the EVM's `NOT` operation - it inverts all the bits in the operand (treated as a 32 byte word).

With care, this provides a compact way to specify negative numbers In the EVM's twos-complement arithmetic. `(~ 4)` is equivalent to -5, so `(+ 5 (~ 4))` evaluates to zero.

Macro definition - `def`

Overview

LLL macros provide a powerful way to make writing LLL code efficient.

There are two forms of macro definition. In the following, `NAME` is a quoted macro name as per the rules below, and `name` is the unquoted version, i.e. `'foo` and `foo` respectively.

- `(def NAME EXPR)` defines a macro without arguments, such as a constant. Wherever the atom `name` appears, it will be substituted with `EXPR`.

  E.g. `{(def 'foo 42) foo)}` evaluates to 42.

- `(def NAME (ARG1 ARG2 ...) EXPR)` defines a macro with zero or more arguments. When the expression `(name ARG1 ARG2 ...)` appears it will be substituted with the arguments passed to EXPR.

  E.g after defining `(def 'sum (l r) (+ l r))`, the expression `(sum 2 3)` will evaluate to 5. to 5. And after defining `(def 'panic () 0xfe)` then the expression `(panic)` will insert an invalid opcode, causing the EVM to throw.

  Macros with the same name but differing numbers of arguments are treated as different macros and do not conflict with each other.

Macros can be defined in terms of other macros and expansion will occur recursively until only native expressions remain.

Macro names

Although the `def` expression allows a wide latitude in assigning macro names, some restrictions apply if the macro name is to be usable. Essentially, the same rules apply as for single quoted strings, except that,

- there is no upper bound on length,
- a double quote mark may not be used in the name (single quote is OK), and
- the name may not begin with a numeral.

All of the following correctly evaluate to 100, but are perhaps ill-advised:

```
{(def '£ 100) £}
{(def 'a' 100) a'}
{(def 'a (sub 0 100)) (def '-a (sub 0 a)) -a}
{(def 'thismacronameislongerthan32characters 100)␣
↪thismacronameislongerthan32characters}
```

It is possible for macros to shadow built-in operators, EVM operators and previously defined macros. For example, the following works as a definition of a unary negation operator:

```
(def '- (n) (- 0 n))
```

After this, `(- 42)` evaluates to an int256 -42 rather than +42 as it normally would.

This feature ought, perhaps, to be used sparingly, if at all.

Macro scope

From Gav Wood's original documentation, the following applies to macro scoping. If anyone can make sense of this (or the source code) so as to explain it more simply (i.e. so I can understand it), I would be most grateful.

> Environmental definitions at the time of the macro's definition are recorded alongside the macro. When a definition must be resolved, definitions made within the present macro are treated preferentially, then arguments to the macro, then definitions & arguments made in the scope of which the macro was called (treated in the same order), then finally definitions & arguments stemming from the scope in which the macro was created (again, treated in the same order).

Whatever else this means, it does mean that macros cannot be defined recursively, so the following does not compile. (Actually, the compiler just chases its tail trying to recursively expand the macro until it eventually coredumps.)

```
;; This will not compile
(seq
  (def 'fac (n) (when (> n 1) (* n (fac (- n 1)))))
  (fac 5))
```

This is probably just as well, as the resulting code could be unexpected. It is important to remember that *macros are not functions*. Macros get fully expanded in place at each invocation. If you have 10 invocations in different places, the same code will be duplicated ten times.

Macro arguments

Evaluation of macro arguments is done *after* they have been substituted. This can be very significant if the arguments are complex expressions. It can lead to surprise explosions in gas usage, and potentially to unexpected side-effects from evaluating the same expressions multiple times.

Consider the following:

```
(seq
  (def 'round (a b) (* (/ a b) b))
  (round 35 (exp 2 5)))
```

This looks innocent enough. However, since the parameter `b` appears twice in the macro body, the `exp` expression will be evaluated twice. If the parameter expressions are more complex, this can quickly become expensive.

One way to deal with this is for the macro to store its arguments in memory temporarily if they appear more than once in the body:

```
(seq
  (def 'round (a b) (seq [0]:b (* (/ a @b) @b)))
  (round 35 (exp 2 5)))
```

As for side-effects, the following evaluates to 6 rather than 3 (which is what you would expect were expressions evaluated before substitution):

```
(seq
  (def 'inc (m) {[m]:(+ @m 1) @m})
  (def 'thrice (a) (+ a a a))
  (return (thrice (inc 0))))
```

Macro example

Here's a simple four-argument macro for raising ERC20 "Transfer" and "Approval" events:

```
(def 'event3 (id addr1 addr2 value)
  (seq
    (mstore 0x00 value)
    (log3 0x00 0x20 id addr1 addr2)))
```

We can use plain macros to store the ABI event ID constants for convenience:

```
;; Event IDs
(def 'transfer-event-id
  (sha3 0x00 (lit 0x00 "Transfer(address,address,uint256)")))
```

```
(def 'approval-event-id
  (sha3 0x00 (lit 0x00 "Approval(address,address,uint256)")))
```

Now it's easy to raise an event:

```
(event3 transfer-event-id (caller) to value)))
```

Including files - `include`

`(include "filename.lll")` inserts the contents of *filename.lll* at this point in the code being parsed. Note that, as ever, subject to the *rules for strings* (except that the length is unlimited), the filename can be given as a single quoted string: `(include 'filename.lll)`.

`include` can appear anywhere an expression would be valid. For example, this is fine and returns whatever the code in *foo.lll* evaluated to: `(return (include "foo.lll"))`. Note that the contents of the included file must evaluate to a single expression even if it is being included within a `seq` expression.

`include` may be used to insert external libraries of common macro definitions shared between projects.

Filepaths may be absolute or relative to the current directory. A filename on its own is looked for in the current directory.

Control structures

`seq`

`(seq EXPR1 EXPR2 ...)` evaluates all following expressions in order. It evaluates to the result of the final expression given.

`raw`

`(raw EXPR1 EXPR2 ...)` evaluates all following expressions in order. It evaluates to the result of the first non-void expression (i.e. the first expression that leaves anything on the stack - this can be manipulated with `pop`), or void if there is none.

For example, `(raw (pop 1) 2 (pop 3))` evaluates to 2.

We can use `raw` to avoid assigning a temporary variable when implementing Euclid's GCD algorithm:

```
;; Evaluates to GCD(a,b)
(seq
  (set 'a 1071)
  (set 'b 462)
  (while @b
    [a]:(raw @b [b]:(mod @a @b)))
  @a)
```

Normally the `while` body would need explicit temporary storage: `{[0x00]:@b [b]:(mod @a @b) [a]:@0x00})`. `raw`'s properties allow us to avoid this, as above. It saves 36 gas in this example! (Much more with bigger problems.)

if

This is an "if-then-else" construction.

In `(if PRED Y N)`: when the predicate `PRED` evaluates to non-zero, `Y` is evaluated; when `PRED` evaluates to zero, `N` is evaluated.

The following calculates the absolute value of signed 256 bit input:

```
(if (S< (calldataload 0x04) 0)
  (- 0 (calldataload 0x04))
  (calldataload 0x04))
```

when, unless

`(when PRED BODY)` evaluates `BODY`, discarding any result, if and only if `PRED` evaluates to a non-zero value.

For example, a "not-payable" guard:

```
(when (callvalue) revert)
```

`(unless PRED BODY)` evaluates `BODY`, discarding any result, if and only if `PRED` evaluates to zero.

A guard for checking that exactly one argument has been passed in the call data:

```
(unless
  (= 0x24 (calldatasize))
  revert)
```

while, until

`(while PRED BODY)` evaluates `PRED` and if the result is non-zero evaluates `BODY`, discarding the result. This is repeated while `PRED` remains non-zero.

Let's say you are putting data into contract storage at consecutive locations starting at zero. The following will count how many items you have. (For fewer than a hundred or so items it's likely cheaper to re-count them than to store a count separately.)

```
(seq
  [0x00]:0
  (while (sload @0x00) [0x00]:(+ 1 @0x00))
  @0x00)
```

`(until PRED BODY)` is the same as `while` except that it evaluates `BODY` when `PRED` is zero until and continues until it becomes non-zero.

Evaluates to the number of leading zero bytes in the call data (up to 32 max):

```
(seq
  [0x20]:(calldataload 0x04)
  (until
    (or (= @0x00 32) (byte @0x00 @0x20))
    [0x00]:(+ 1 @0x00))
  @0x00)
```

for

`(for INIT PRED POST BODY)` evaluates `INIT` once (ignoring any result), then evaluates `BODY` and `POST` (discarding the result of both) as long as `PRED` is true.

The following code computes factorials: 10! = 3628800 = 0x375f00 in this case.

```
(seq
  (for
    (seq (set 'i 1) (set 'j 1))       ; INIT
    (<= (get 'i) 10)                  ; PRED
    (mstore i (+ (get 'i) 1))         ; POST
    (mstore j (* (get 'j) (get 'i)))) ; BODY
  (get 'j))
```

This is one of the rare occasions where I think the compact notation is actually an improvement. The following compiles to the same bytecode.

```
(seq
  (for
    { (set 'i 1) (set 'j 1) } ; INIT
    (<= @i 10)                ; PRED
    [i]:(+ @i 1)              ; POST
    [j]:(* @j @i))            ; BODY
  @j)
```

Logical Operators `&&`, `||`, `!`

Logical "and", "or" and "not".

Both `&&` and `||` can take any non-zero number of arguments. They evaluate the arguments from left to right and perform short circuit evaluation so that evaluation of arguments stops as soon as the outcome is known. I.e. `(&& EXPR1 EXPR2 ...)` will stop evaluating after encountering an expression that evaluates to zero; `(|| EXPR1 EXPR2 ...)` will stop evaluating after encountering an expression that evaluates to non-zero.

The final value of the expression is the value of the last sub-expression if it is evaluated, otherwise 1 for `||` and 0 for `&&`. Thus, `(|| 123 456)` evaluates to 1 (due to the short-circuit), and `(&& 123 456)` evaluates to 456.

`!` is a unary logical not operator, thus it takes one argument. `(! EXPR)` evaluates to zero when `EXPR` evaluates to non-zero, and to one when `EXPR` evaluates to zero. It is equivalent to `(iszero EXPR)`.

Literals - `lit`

When literals must be included that can be placed into memory, there is the `lit` operation.

- `(lit POS STRING)` Places the string `STRING` into memory at `POS` and evaluates to its length. The usual *rules for strings* apply, except that there is no limit on the length.
- `(lit POS BIGINT)` Places `BIGINT` into memory at POS and evaluates to the number of bytes it takes. Unlike for the previous case, `BIGINT` may be arbitrarily large, and thus if specified as hex, can facilitate storing arbitrary binary data.

So, `(lit 0x40 "Hello, world!")` copies the string to memory starting at byte 0x40 and returns 13, the length of the string.

[Note that the former `(lit POS INT1 INT2 ...)` functionality was changed in PR #1329. I'm not sure of the background to this as it does look potentially useful.]

Variables

LLL has an analogue of variables. It is relatively cheap to write to and read from memory, so it can be efficient to store intermediate quantities temporarily in memory. Since LLL doesn't provide direct access to the EVM stack this is a practical alternative.

Variables provide a convenient way to automatically assign names to memory locations. This automatic approach may or may not be desirable, depending on how much control you wish to have over memory allocation. In any case, it's important to know how and where the variable storage is assigned so that it does not conflict with memory you may assign by other means.

For each variable created using a `set` or `with` expression, 32 bytes of memory are assigned, starting from memory location 0x80 = 128. So, for example, in `{(set 'x 1) (set 'y 2) (set 'z 3)}`, `x` is at 0x80, `y` is at 0xa0 and `z` is at 0xc0. Note that when a variable is `unset`, or goes out of the `with` scope, the memory space is *not reclaimed or reassigned*. Thus, the following will use sixty-four bytes of memory: `{(set 'foo 1) (unset 'foo) (set 'foo 2)}`.

Assigning - `set`

A variable is created with `(set NAME EXPR)`, where NAME is any valid string but with no restriction on length. So all of the following are valid, although not all may be wise choices...

```
(set 'x 42)
(set 'foo 42)
(set '41 42)
(set 'abcdefghijklmnopqrstuvwxyz0123456789 42)
(set '' 42)
(set " " 42)
(set "a b c" 42)
```

Accessing - `get`

The value of a variable can be accessed using the `(get NAME)` expression, where NAME is the same string used in the `set` expression:

```
(get 'foo)
```

Referencing - `ref`

The memory address where a variable is stored can be found using the `(ref NAME)` expression.

Alternatively, using the variable name unquoted evaluates to its address in memory: `foo` and `(ref 'foo)` are equivalent. This can be useful when using *compact notation*: `@foo` evaluates to the value of the variable and is equivalent to `(get 'foo)`. Note that using variable names unquoted like this restricts the space of variable names that may be assigned (no leading numerals, spaces, etc.).

Unassigning - `unset`

[Coming in PR #2520]

Variable names can unassigned with `(unset NAME)`. After this the name may no longer be referenced. If the same name is reassigned with `set` or `with` then a new memory location is assigned.

Temporary - `with`

[Coming in PR #2520]

A temporary variable may be assigned using the `(with NAME EXPR1 EXPR2)` expression. `EXPR2` is then evaluated with variable `NAME` set to `EXPR1`. `with` expressions may be nested for multiple local variables.

When the `with` expression ends, the variable name is unset, but the memory is not reclaimed or re-used.

The following evaluates to 5:

```
(with 'x 2
  (with 'y 3
    (+ @x @y)))
```

Memory allocation - `alloc`

[Exact behaviour still TBD - see PR #2545]

`(alloc SIZE)` provides `SIZE` contiguous bytes of memory starting from the current top of memory. It returns the start of the memory space allocated. This is memory that has not been previously written to (or read from), and is all initialised to zero.

Since memory is allocated in multiples of 32 bytes, the actual amount allocated is rounded up to the next 32 byte boundary:

```
(alloc 0)  ;; Does nothing, returns (msize) unchanged
(alloc 1)  ;; Allocates 32 bytes, returns the original (msize)
(alloc 32) ;; Allocates 32 bytes, returns the original (msize)
```

It isn't necessary at all to use `alloc` to reserve memory; the LLL programmer has complete control over how memory is laid out and used. However, `alloc` could be useful for macros that need to find some unused space in which to write return data, for example.

Note that the gas cost of memory is proportional to the number of bytes used up to 724 bytes, and increases superlinearly above that.

Assembler - `asm`

Low-level assembler may be included in line with one caveat; it must have transparent stack usage. This basically means that `JUMP` or `JUMPI` are best avoided; if used then ignoring their jump effects (and thus assuming the jump doesn't happen and the PC just gets incremented) must have a valid final result in terms of items deposited on the stack. Usage is:

```
(asm ATOM1 ATOM2 ...)
```

Where the `ATOM`s may be either valid, non-`PUSH` VM instructions or literals (in which case they will result in an appropriate `PUSH` instruction). The EVM assembler language is defined in the Yellow Paper.

For example, `(asm 69 42 ADD)` evaluates to the value 111. Note any assembler fragment that results in fewer than zero items being deposited on the stack or greater than 1 will almost certainly not interoperate with the rest of the language and thus cause compile errors.

Code - `lll`

For handling cases where code needs to be compiled and passed around, there is the `lll` expression:

```
(lll EXPR POS MAXSIZE)
(lll EXPR POS)
```

This places the EVM-code as compiled from `EXPR` into memory at position `POS` if and only if said EVM-code is at most `MAXSIZE` bytes, or if `MAXSIZE` is not provided. It evaluates to the number of bytes of memory written, i.e. either 0 or the number of bytes of EVM-code; if provided, this is always at most `MAXSIZE`.

Contract creation code will typically look something like:

```
{
  ;; Initialisation code goes here
  ;; This just records who the original creator is
  [[0]] (caller)

  ;; Return the contract code
  (return 0 (lll {
    ;; Contract code goes here
    ;; This just self-destructs if called by the original creator
    (when (= (caller) @@0) (selfdestruct (caller)))
  } 0))
}
```

There is a built-in macro, `returnlll` described below, that simplifies this pattern.

Code size - `bytecodesize`

`(bytecodesize)` evaluates to the total size of the compiled EVM bytecode in bytes.

This is useful when creating a constructor for a contract: arguments passed at contract creation are appended to the contract bytecode and can be accessed through a combination of the `codecopy` EVM instruction and `bytecodesize`.

The following will evermore return the initial argument that was appended to the bytecode used for contract creation:

```
(seq

  ;; constructor: store the passed-in data word which is appended to the bytecode
  (codecopy 0x00 (bytecodesize) 32)
  (sstore 0x00 @0x00)

  ;; contract body
  (returnlll
    (return (sload 0x00))))
```

Built-in Macros

A number of LLL macros are pre-defined by the compiler for convenience. They can be seen in the source file liblll/CompilerState.cpp. There is test code for most of the macros in test/liblll/EndToEndTests.cpp which may be a useful reference.

Utility

`(def 'panic () (asm INVALID))` Inserts an invalid instruction. It is conventional to use this to "throw" on an internal error.

`(def 'allgas (- (gas) 21))` A helper used by some of the message-call macros below.

Message-calls

`(def 'send (to value) (call allgas to value 0 0 0 0))` Transfer `value` Wei to the address `to`. No call data or return data. Evaluates to 1 on success of the transfer and 0 on failure.

`(def 'send (gaslimit to value) (call gaslimit to value 0 0 0 0))` As above, but provides the opportunity to specify the gas limit explicitly. This would be 21000 for a simple value transfer to an account.

`(def 'msg (to data) { [0]:data (msg allgas to 0 0 32) })` Message-call into an account with no transfer value and a single 32 byte word of `data`. Evaluates to a 32 byte word returned from the call, which also overwrites memory location 0x00.

`(def 'msg (to value data) { [0]:data (msg allgas to value 0 32) })` As above, but also transfers `value` Wei.

`(def 'msg (gaslimit to value data) { [0]:data (msg gaslimit to value 0 32) })` As above, but allows `gaslimit` to be set (the above calls use the `allgas` macro as the default.)

`(def 'msg (gaslimit to value data datasize) { (call gaslimit to value data datasize 0 32) (` As above, but can handle arbitrary amounts of input data. `data` is now the starting memory location, and `datasize` its length in bytes.

`(def 'msg (gaslimit to value data datasize outsize) { [0]:0 [0]:(msize) (call gaslimit to v` This version with six arguments allows all call parameters to be set, except that it will automatically allocate memory space for the arbitrary length returned data (`outsize` bytes of it) at the current top of memory. Evaluates to the memory location of the start of the return data.

Contract creation

`(def 'create (value code) { [0]:0 [0]:(msize) (create value @0 (lll code @0)) })` `create` with two arguments uses the built-in EVM CREATE opcode (which has three arguments) to create a new account with the associated `code` (as delivered by `returnlll`)). The value `value` is transferred to the new account. Returns 0 on failure, the new account's address on success.

`(def 'create (code) { [0]:0 [0]:(msize) (create 0 @0 (lll code @0)) })` As above, but without a value transfer. This could be defined more succinctly as `(def 'create (code) (create 0 code))`.

Note that in the above macros, memory location 0x00 is first written to in order to "reserve" it. This avoids an edge case where `msize` is initially zero and data gets overwritten by the `lll` operation.

Keccak256/SHA3 functions

`(def 'sha3 (loc len) (keccak256 loc len))` The EVM opcode for `SHA3` was changed to `KECCAK256` to reduce confusion. this macro ensures that legacy code continues to compile. It calculates the Keccak256 hash of the data in memory starting from `loc` and with length `len`. The expression evaluates to the result.

(def 'sha3 (val) { [0]:val (sha3 0 32) }) With one argument `sha3` evaluates to the Keccak256 hash of 32 byte input `val`. Note that memory location 0x00 is overwritten with the input parameter.

(def 'sha3pair (a b) { [0]:a [32]:b (sha3 0 64) }) Concatenates the two 32 byte arguments and returns the Keccak256 hash over the resulting 64 bytes. Overwrites memory locations 0x00-0x3f with the input parameters. The expression evaluates to the result.

(def 'sha3trip (a b c) { [0]:a [32]:b [64]:c (sha3 0 96) }) Concatenates the three 32 byte arguments and returns the Keccak256 hash over the resulting 96 bytes. Overwrites memory locations 0x00-0x5f with the input parameters. The expression evaluates to the result.

Returns

(def 'return (val) { [0]:val (return 0 32) }) Halt execution and return the 32 byte/256 bit argument, `val`, to the caller.

(def 'returnlll (code) (return 0 (lll code 0))) This is a convenience macro for handling the byte code of the body of the contract. Typically an LLL contract will have a structure on these lines: `{ CONSTRUCTOR-EXPRESSIONS (returnlll {BODY-EXPRESSIONS})}`.

Storage handling

(def 'makeperm (name pos) { (def name (sload pos)) (def name (v) (sstore pos v)) }) Helper macro for `perm`.

(def 'permcount 0) Helper macro for `perm`.

(def 'perm (name) { (makeperm name permcount) (def 'permcount (+ permcount 1))}) This allows named references to storage locations. `(perm 'foo)` creates two macros: `(foo EXPR)` that will store the value of `EXPR` in permanent storage; and `foo` which will evaluate to the value stored. Storage locations are assigned consecutively, starting numbering from zero. (The starting point could be changed by redefining `permcount` at the top of your code if desired.)

Built-in contracts

(def 'ecrecover (hash v r s) { [0] hash [32] v [64] r [96] s (msg allgas 1 0 0 128) }) Uses the built-in contract at address 0x01 to verify Ethereum signatures. If the signature is good then it will return the correct signing address. See the test cases for an example. Overwrites memory locations 0x00 - 0x7f and evaluates to the resulting address, or zero on failure.

(def 'sha256 (data datasize) (msg allgas 2 0 data datasize)) Uses the built-in contract at address 0x02 to calculate the SHA256 hash of arbitrary quantities of data stored beginning from memory location `data`. `datasize` is in bytes. Places the resulting hash at memory location 0x00.

(def 'ripemd160 (data datasize) (msg allgas 3 0 data datasize)) Uses the built-in contract at address 0x03 to calculate the RIPEMD-160 hash of arbitrary quantities of data stored beginning from memory location `data`. `datasize` is in bytes. Places the resulting hash at memory location 0x00.

(def 'sha256 (val) { [0]:val (sha256 0 32) }) Uses the built-in contract at address 0x02 to calculate the SHA256 hash of a 32 byte word of data. Places the resulting hash at memory location 0x00.

(def 'ripemd160 (val) { [0]:val (ripemd160 0 32) }) Uses the built-in contract at address 0x03 to calculate the RIPEMD-160 hash of a 32 byte word of data. Places the resulting hash at memory location 0x00.

Ether sub-units

(def 'wei 1) The smallest subunit. One Ether is `(* 1000000000000000000 wei)`.

(def 'szabo 1000000000000) The number of Wei in a Szabo. One Ether is `(* 1000000 szabo)`.

(def 'finney 1000000000000000) The number of Wei in a Finney. One Ether is `(* 1000 finney)`.

(def 'ether 1000000000000000000) The number of Wei in an Ether.

Shift instructions

These should be replaced by native instructions once supported by EVM

(def 'shl (val shift) (mul val (exp 2 shift))) Shift `val` left by `shift` bits, filling with zero bits. This is a relatively expensive operation. When the EVM finally has support for native `SHL` (EIP #145) then this macro should be removed.

(def 'shr (val shift) (div val (exp 2 shift))) Shift `val` right by `shift` bits, filling with zero bits. This is a relatively expensive operation. When the EVM finally has support for native `SHR` (EIP #145) then this macro should be removed. `(shr (calldataload 0x00) 224)` is a convenient way to extract the ABI function reference from the call data.

CHAPTER 3

LLL Compiler

Note: This documentation is a work in progress, so please exercise appropriate caution. It a personal effort and has no formal connection with the Ethereum Foundation.

Note: Everything in these docs pertains to the Solidity/LLL implementation, specifically the *develop* branch.

Installing the compiler

[Does this need to be covered? Link to DE's screencast? Move to resources section?]

[Reference the Solidity "building from source" section]

Invoking the compiler

[And what to do with the output...]

Compiler Options

The `lllc` compiler options are reasonably self-explanatory and are as follows.

When multiple options are used the following rules apply.

- `-h` and `-V` take precendence over all others, and the *first* one of these listed is executed.
- `-a`, `-b`, `-t`, `-x`, `-d` are mutually exclusive output formats. The *last* of these listed defines the output format created.

- `-o` can be used with any of the output formats.

-h,--help

Displays the following compiler options.

```
-b,--binary  Parse, compile and assemble; output byte code in binary.
-x,--hex  Parse, compile and assemble; output byte code in hex.
-a,--assembly  Only parse and compile; show assembly.
-t,--parse-tree  Only parse; show parse tree.
-o,--optimise  Turn on/off the optimiser; off by default.
-h,--help  Show this help message and exit.
-V,--version  Show the version and exit.
```

-x,--hex

Parse, compile and assemble; output byte code in hex.

This is the default.

```
> echo '(add 2 3)' | lllc
6003600201

> echo '(add 2 3)' | lllc --hex
6003600201
```

-a,--assembly

Only parse and compile; show assembly.

Outputs the intermediate assembly language which is shared with Solidity and compiled into the final bytecode. If the `-o` flag is used as well then the assembly language is displayed after optimisation.

```
> lllc -a erc20.lll
  jumpi(tag_42, iszero(callvalue))
   0x0
  dup1
  revert
tag_42:
  sstore(caller, 0x186a0)
  ...
```

-t,--parse-tree

Only parse; show parse tree.

The "parse tree" is the clean version of the source code which is fed to the LLL parser: all comments and linebreaks are removed, whitespace is normalised, numbers are all converted to decimal and quoted strings standardised.

```
> echo "(def 'foo (mload 0x0a)) ; define foo" | lllc -t
( def "foo" ( mload 10 ) )
```

-d,--disassemble

The `-d` option is not documented in the `--help` output. It decompiles hexadecimal EVM code into readable opcodes.

```
> echo 602a600055 | lllc -d
PUSH1 0x2A PUSH1 0x0 SSTORE
```

-b,--binary

Parse, compile and assemble; output byte code in binary.

I haven't found a use for this yet.

-o,--optimise

Turn on/off the optimiser; off by default.

The optimiser passes the assembly output through Solidity's optimiser. The main useful thing the optimiser can do is the replacement of constant expressions, but it doesn't always manage to spot all opportunities for this.

```
> echo '(add 1 (mul 2 (add 3 4)))' | lllc
600460030160020260010 1

> echo '(add 1 (mul 2 (add 3 4)))' | lllc -o
600f
```

-V,--version

Show the version and exit. Note that the short form is a capital `V`.

```
> lllc -V
LLLC, the Lovely Little Language Compiler
Version: 0.4.12-develop.2017.6.27+commit.b83f77e0.Linux.g++
```

CHAPTER 4

LLL and the ABI

Note: This documentation is a work in progress, so please exercise appropriate caution. It a personal effort and has no formal connection with the Ethereum Foundation or anyone else.

In order for smart contracts to interoperate with the rest of the Ethereum ecosystem, an Application Binary Interface (ABI) has been defined.

The ABI defines the way that data passed to and from contracts is represented in binary form: the sizes, structures and layouts that can be used.

LLL programmers are under no obligation at all to implement the ABI, but if you want any third-party to be able to read from or write to your contract then it is pretty much necessary to do so. And many tools within the Ethereum ecosystem expect to find the ABI implemented.

What follows is an introduction to interacting with the ABI in LLL. It doesn't aim to be at all comprehensive and the ABI Specification remains the authoritative source of information.

ABI Data format overview

Data within the ABI is passed exclusively via blocks of 32-byte words.

There are two types of data defined in the ABI and handled differently:

1. *Elementary types* are those that are represented entirely within a 32-byte word, including `bool`, `uint8`, `uint256`, `address`, `bytes32`.
2. *Dynamic types* are those that have variable size and may require multiple words to specify, such as `string`, `bytes`, or arrays such as `uint256[]`, `bool[]`. These are represented by a 32-byte pointer to where the actual data is stored along with its size.

In addition, only when calling a function within a contract, the four-byte "signature" of that function is prepended to the call data.

A complete set of call data to a contract has a structure like this:

```
// Function selector
[4 bytes: function selector]

// Argument list
[32 bytes: ARG1]
[32 bytes: ARG2]
...
[32 bytes: ARGn]

// Dynamic data
[32 bytes: length of ARGi data, Ni]
[Ni or 32*Ni bytes: actual data]
[32 bytes: length of ARGj data, Nj]
[Nj or 32*Nj bytes: actual data]
...
```

In the above, each `ARGm` in the argument list is either the actual data (for an elementary type), or a pointer to the start of the data (for a dynamic type).

Encoding Example

Putting this all together, calling a function with signature `foo(uint256,string,address)` looks like this:

```
0x00 [Function selector - 4 bytes 0xf2f69ca5]
0x04 [ARG1, elementary type, uint256 - 32 bytes, right-aligned]
0x24 [ARG2, dynamic type, pointer to start of the string data, 0x60]
0x44 [ARG3, elementary type, address - 32 bytes, right aligned]
0x64 [Length of ARG2 string - 32 bytes, right aligned]
0x84 [Start of string contents of ARG2. Continues for as many full words as the␣
↪string needs]
```

So, `foo(42, "Hello, world!", 0x0123456789012345678901234567890123456789)` would look like this, with 164 bytes of data:

```
;; Truncated Keccak-256 hash of foo(uint256,string,address)
0x00 f2f69ca5
;; uint256 42 in a 32-byte right-aligned word
0x04 000000000000000000000000000000000000000000000000000000000000002a
;; pointer to the start of the string data, 0x60 in this case
0x24 0000000000000000000000000000000000000000000000000000000000000060
;; address in a 32-byte right-aligned word
0x44 0000000000000000000000000123456789012345678901234567890123456789
;; the length of the string: 13 or 0x0d
0x64 000000000000000000000000000000000000000000000000000000000000000d
;; the string text, *left* aligned in a 32-byte word
0x84 48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Note that for the dynamic data in ARG2—the string—the pointer to the start of the string data ignores the function selector. I.e. the string data is considered to start at 0x60, not at 0x64.

Finally, then, the call data to send with the transaction is:

```
f2f69ca5000000000000000000000000000000000000000000000000000000000000002a0000000000000000000000000000000
```

All those zeros look wasteful, but the gas cost for a zero byte of input data is only 4, whereas it is 68 for a non-zero input byte.

Data Types

The following is a quick overview. Much more detailed descriptions and examples are provided in the ABI Specification.

Elementary Types

Each of the elementary types listed in the ABI specification is represented in the call data as a 32 byte word. Any smaller types, such as booleans or addresses, must be padded on the left (high order bytes) or right (low order bytes) with zero bytes: `bytes` types are left-aligned; others are right aligned.

bool(true): `0x0000000000000000000000000000000000000000000000000000000000000001`

uint8(42): `0x000000000000000000000000000000000000000000000000000000000000002a`

uint32(42): `0x000000000000000000000000000000000000000000000000000000000000002a`

int256(-1): `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`

int8(-1): `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`

address(0x314159265dD8dbb310642f98f50C066173C1259b) `0x000000000000000000000000314159265dD8dbb310642`

bytes32('0x1234'): `0x1234000000000000000000000000000000000000000000000000000000000000`

Dynamic Types

Data that is variable in length and could exceed the bounds of a 32-byte word is treated as a dynamic type.

Within the argument list part of the data, a dynamic type is represented by a 32 byte pointer to where the actual data is stored, which will be after the end of the argment list. The pointer is the offset in bytes from the beginning of the argument list to the word where the data's length is stored.

For most dynamic types, the length is stored in a 32 byte word as (effectively) a uint256. Immediately after the length comes the data.

The data occupies as much space as required by the length, rounded up to a multiple of 32 bytes/whole words. For `string` and `bytes` types, the data occupies one byte per unit of length specified. Simple one-dimensional arrays occupy one 32 byte word per element.

The ABI Specification has a good example.

Passing data

There are essentially four situations where we are passing data around in this format, if we include Event non-indexed data. In each case the data are in the same format as above, but are passed by different mechanisms.

Passing data to the Constructor

Constructor data at contract deployment is simply appended to the contract code as a block of 32-byte words with no function selector.

Accessing this constructor data is described in Daniel Ellison's 2nd article on the structure of an LLL contract.

Essentially, the first word of the ABI data can be copied to memory at position 0x00 using:

```
(codecopy 0x00 (bytecodesize) 32)
```

and you can continue parsing and processing the data from there.

Passing data to a function

When calling a function in a contract, all the necessary information is contained in the "call data" that forms part of the transaction. You can check the length of the call data with `(calldatasize)` - this evaluates to the number of bytes of call data available. Reading beyond the end of the call data is not an error, it just results in zero bytes being read.

Function call data at run time is prepended with the four-byte function selector as described below, but otherwise follows the same format of 32-byte blocks described above.

A convenient way to access the function selector is as follows.

```
(seq
  (mstore 0x00 0)
  (calldatacopy 0x1c 0x00 4))
```

This first zeroes all the bytes in memory location `0x00` and then copies the first four bytes of the call data to the last four bytes of the word at memory `0x00`. This can then easily be used in comparisons to find the right function:

```
(when (eq @0 0xf2f69ca5)
  (execute-function-foo))
```

See the [TODO] design patterns page for guidelines on implementing functions.

Returning data from a function

Returning data from a function follows exactly the same format of composing the data (whether elementary or dynamic) into 32-byte blocks, but omits any function selector.

Once the data has been marshalled into contiguous memory, it is returned as follows:

```
(return start length)
```

`start` is the start location of the data in memory to be returned in bytes, `length` is the length in bytes of data to be returned. To be ABI compliant, length must be a multiple of 32.

Events

Another way to expose internal data to the outside world is via what the ABI (and Solidity) calls "Events". These are just executions of the EVM `LOGn` opcodes.

EVM log entries and Events as specified by the ABI relate to each as follows.

EVM Log Entry

An EVM log entry comprises,

- An arbitrary length `data` blob.

- `n` topics, `topic[0]` to `topic[n-1]`, each of which is a 32-byte word with the corresponding data type specified in the event signature.

In addition, the EVM provides the address of the contract emitting the event.

In terms of LLL, the following generates a three-topic log entry with 32 bytes of `data` (read from memory starting at 0x00), `topic[0]` is the `event-id` as described below, and `topic[1]` and `topic[2]` are each an address:

```
(log3 0x00 32 event-id addr1 addr2)
```

ABI Event

The ABI specifies that the EVM Log entry maps to ABI Events as follows.

- `topic[0]` is the Event signature. This is like a function signature, but is the full 32-byte Keccak-256 hash over the event name and arguments.

  For example, an ERC20 "Transfer" Event has the signature, `keccak-256("Transfer(address,address,uint256)")`, which is `0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523`
- Further topics correspond to the first `n-1` arguments in the Event signature (the "indexed args").
- The `data` blob corresponds to the final argument of the Event signature or the "non-indexed" arguments (simplifying here; see the ABI Specification for the details).

For example, to produce an ERC20 `Transfer(address,address,uint256)` Event, we can use the following macro in LLL.

```
(def 'event3 (id addr1 addr2 value)
  (seq
    (mstore 0x00 value)
    (log3 0x00 0x20 id addr1 addr2)))
```

`id` is the 32-byte Event signature for `Transfer` as described above. This is recorded as `topic[0]` in the event log. `addr1` and `addr2` are two Ethereum addresses and are `topic[1]` and `topic[2]` respectively in the event log. The amount of the transfer, `value` is a `uint256` and is first written to memory and then recorded as the `data` element of the Event.

Techniques

This section aims to provide some practical suggestions around working with LLL and the ABI.

Functions

When calling a function in a contract in accordance with the ABI then the first four bytes of the call data are a truncated Keccak256 hash over the function signature. The left-most, highest-order four bytes of the hash are used. We will call this the function selector.

For example:

```
name()
→ 0x06fdde0383f15d582d1a74511486c9ddf862a882fb7904b3d9fe9b8b8e58a796
→ 0x06fdde03

transferFrom(address,address,uint256)
```

```
→ 0x23b872dd7302113369cda2901243429419bec145408fa8b352b3dd92b66c680b
→ 0x23b872dd
```

The function signature is the case-sensitive function name followed by a parenthesised list of its argument types in order. Allowable types are listed in the ABI Specification. Note that no argument names or spaces are included in the function signature string that is hashed.

Once again, this has nothing to do with LLL *per se*, only with how external entitities will interact with your contract written in LLL. The contract itself only sees the four byte function selector hash at the front of a block of data containing the function arguments (the "call data").

You can generate the function selector by pasting the function signature into a Keccak256 hash generator and taking the first four bytes only. Alternatively, from a web3.js 1.0.0 enabled console, you can do as follows:

```
> web3.utils.sha3("name()")
'0x06fdde0383f15d582d1a74511486c9ddf862a882fb7904b3d9fe9b8b8e58a796'

> web3.utils.sha3("transferFrom(address,address,uint256)")
'0x23b872dd7302113369cda2901243429419bec145408fa8b352b3dd92b66c680b'
```

See also Remix in the next section.

Generating the JSON ABI

To share your contract's interface with others, a JSON format for the contract's ABI is defined.

One way to generate the ABI for your contract relatively painlessly is to feed the function definitions into the Solidity compiler with the `--abi` flag. On the Linux command line, as follows:

```
echo 'interface Foo{function totalSupply() constant returns (uint256); function␣
↪transfer(address,uint256) returns (bool); event Transfer(address,address,uint256);}
↪' | solc --abi

Contract JSON ABI
[{"constant":true,"inputs":[],"name":"totalSupply","outputs":[{"name":"","type":
↪"uint256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"
↪","type":"address"},{"name":"","type":"uint256"}],"name":"transfer","outputs":[{
↪"name":"","type":"bool"}],"payable":false,"type":"function"},{"anonymous":false,
↪"inputs":[{"indexed":false,"name":"","type":"address"},{"indexed":false,"name":"",
↪"type":"address"},{"indexed":false,"name":"","type":"uint256"}],"name":"Transfer",
↪"type":"event"}]
```

The constructor ABI should also be included if relevant. Of course it's easier if you read from and write to files in practice.

You can also use the online Remix IDE to do this. Click on "Contract details (bytecode, interface etc.)" to see the Interface ABI generated. Remix will also tell you the function selector hashes, so you can do it all in one place.

Note that "constant" functions are those that don't change the blockchain state: i.e. they don't transfer value, change anything in storage or emit any events. These functions can be evaluated at zero gas cost on a local node without broadcasting a transaction to the blockchain.

Using web3.js to call the ABI

Once you have the JSON ABI descriptor for your contract then you can interact with it using standard tools such as web3.js (documentation), which is easier than messing around with the call data directly.

Note: The following examples all use web3.js version 1.0.0-beta.

You can see the raw data that would get sent to your contract without even deploying it. Web3.js will calculate it for you from the provided ABI and input arguments.

```
> var Web3 = require('web3');
undefined
> var web3 = new Web3();
undefined
> var myContract = new web3.eth.Contract([{inputs: [{type:'string'}], name: 'foo_
↪string', outputs: [], type: 'function'}]);
undefined
> myContract.methods.foo_string("abc").encodeABI()

↪'0x1099ee880000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000
↪'
```

This is what will be sent to the function as the transaction's call data when it is invoked on the blockchain.

- The first four bytes of `keccak-256("foo_string(string)")` are `0x1099ee88` which we see at the beginning as the function selector.
- Then we see the `uint256` quantity `0x20`, which is the start of the string in the call data (ignoring the function selector).
- Next is the `uint256` value `3`, the length of the string.
- Finally the three left-justified ASCII values `0x61`, `0x62`, `0x63`, which are just the string "abc".

Worked example

The following LLL code simply returns its input data (excluding the function selector) as an ABI compliant `bytes32[]` array.

```
(returnlll
  (seq

    ;; The size in bytes of the calldata minus the function descriptor
    (def 'datalen (- (calldatasize) 4))

    ;; Point to the start of the dynamic data to return (a bytes32[] array)
    [0x00]:0x20

    ;; First word of the dynamic data is the length (in words for bytes32)
    [0x20]:(/ datalen 32)

    ;; Copy the call data to memory as the bytes32[] contents.
    (calldatacopy 0x40 0x04 datalen)

    ;; Return the whole structure we have built
    (return 0x00 (msize))))
```

Note: The following breaks somewhere between web3.js 1.0.0-beta.11 and 1.0.0-beta.15 - aaagh!

You can use the version 1.0.0 web3.js interface to interact with this contract as follows. When setting the `from` address below, use an address that you have access to in your test environment. I'm using `testrpc -d` which generates a set of accounts automatically.

```
var Web3 = require('web3');
var web3 = new Web3('http://localhost:8545');

// You can define different "inputs" types here to play with the ABI.
// The LLL code doesn't care what they are.
var myContract = new web3.eth.Contract([{inputs: [{type:'uint256'},{type:'string'},
↪{type:'address'}], name: 'foo', outputs: [{type:'bytes32[]'}], type: 'function'}]);

// Put your from address in the below.
myContract.options.from='0x90f8bf6a479f320ead074411a4b0e7944ea8c9c1';

// This is the compiled bytecode of the LLL contract.
myContract.options.data =
↪'0x601c80600c6000396000f3006020600052602060043603046020526004360360046040375960 00f3
↪';

myContract.deploy().send().then(function(x){x.methods.foo(42,"LLL rocks!",
↪"0x1234567890123456789012345678901234567890").call().then(console.log)});
```

The output should look like this:

```
[ '0x000000000000000000000000000000000000000000000000000000000000002a',
  '0x0000000000000000000000000000000000000000000000000000000000000060',
  '0x0000000000000000000000001234567890123456789012345678901234567890',
  '0x000000000000000000000000000000000000000000000000000000000000000a',
  '0x4c4c4c20726f636b7321000000000000000000000000000000000000000000' ]
```

This is just the input call data passed back to us as a `bytes32[]` array, as we intended.

CHAPTER 5

Common Design Patterns

Note: This documentation is a work in progress, so please exercise appropriate caution. It a personal effort and has no formal connection with the Ethereum Foundation.

[Some overlap here with the ABI section, but no harm.]

Memory Layout

Constructor/Code

Functions

Input arguments

Function Guards

[Compare Solidity modifiers]

Storage

Arrays

[Enumerable permanent storage]

Mappings

[Non-enumerable permanent storage]