Let’s start by diving right into an abstruse example of the low-level lisp-like language (LLL).

```
{

[[69]] (caller )

(return 0 (lll

    (when (= (caller) @@69)

        (for {} (< @i (calldatasize)) [i](+ @i 64)

            [[ (calldataload @i) ]] (calldataload (+ @i 32))

        )

    )

0))

}
```

As the name suggests, LLL is a low level language. It operates more closely to the EVM bytecode than higher level languages like Solidity or Vyper.

LLL has direct access to storage and memory of the EVM so you can determine exactly where your data sits. When you use an EVM opcode in LLL, it translates directly to the bytecode representation of that opcode. In fact, all EVM opcodes are available to LLL.

It produces smaller binaries that Solidity. As an example, reimplementing uPort's registry contract in LLL. The Solidity implementation comes in at 591 bytes, compiled with the optimize flag. Admittedly, this is pretty small. But the LLL implementation compiles to 171 bytes. So in this case the LLL version is about 70% smaller than the Solidity version. That matters a lot when a contract will be accessed many times. It also makes deployment less costly. This is an extreme case, of course. Generally LLL binaries are about 30 to 40% smaller than the equivalent in Solidity.

LLL is considered lisp like because of its syntax. There are many parenthesis compared to other smart contract languages. LLL uses symbolic expression. Symbolic expressions are a notation for nested list data.

Lisp developers generally use indentation to indicate code blocks.

LLL also uses prefix notation, where operators are placed to the left of their operands. Infix notation is the more familiar notation (2 + 3), which looks like (+ 2 3) in prefix notation. The expression (+ @i 64) adds i and 64.

"Symbolic expressions are a means of representing semistructured data in human readable text form mostly composed of symbols and lists and extensively used in the Lisp programming language." -yourdictionary.com

An example of LLL

```
;; check that the caller is the node owner


(def `only-node-owner (node)
        (when (!= caller) (get-owner-node))

    (panic))
```

LLL uses semicolons for comments.

def starts the definition of an LLL macro. Macros can be used to clarify your code. They aren't even close to the power of Lisp macros. They're more like the C macro system. They do a simple substitution.

Here we're defining a macro called only-node-owner. This will be used as a modifier to ensure that only the node owner can execute a block of code. So when only-node-owner is used in the source, the macro will be looked up and its definition will be compiled into the bytecode.

Macros can take parameters. In this case the node's ID.

The rest of the code is what will be substituted at compile time when the macro is invoked. Here you can see the use of symbolic expressions and prefix notation. Everything is composed of lists defined within matching parentheses. Prefix notation can be seen with the use of !=. That operator comes first, and its operands follow. One of the != operands uses another macro I defined that's not shown here, called get-owner. So you can see that it's easy to build up complex code but keep things clear with the use of macros.

There is actually an LLL contract deployed on the mainnet. The registry contract of the Ethereum Name service (ENS) was written in LLL mainly because of the gas savings due to its size and the brevity of its code. It's the contract in the ENS system that's called more than any other.

Check out the github (https://github.com/ethereum/ens) for details about the LLL contracts as well as the Solidity counterparts.

Useful links:

Ethereum name service

****

This document is the adaptation of the presentation slides into a more easily readable format for reading.

Vyper

Vyper is an **experimental**, contract-oriented, pythonic programming language that targets the Ethereum Virtual Machine (EVM).

Vyper has a few principles and goals that aim to make it a language that is ideal for programming smart contracts.

1. Security is a primary focus for any smart contract language and Vyper maintains that "it should be possible and natural to build secure smart contracts in Vyper."

2. Vyper code should be not just human readable, but make it *difficult to write misleading code*. This directive is aimed at making contract **audits as successful as possible**.

3. Striving for **language and compiler simplicity** support the other goals by keeping confusing complexity under control.

To achieve these goals, Vyper implements the following features:

1. Bounds and overflow checking

2. Support for signed integers and decimal fixed point numbers

3. Decidability - Reliably compute upper bounds for gas consumption of any function call

3. Strong typing

4. Small and understandable compiler code

5. Limited support for pure functions

There is a notable lack of the following features that are present in Solidity:

1. Modifiers - Modifiers make it easier to write misleading code. It encourages writing code where execution jumps around the source file, making audits more difficult. Vyper encourages inline checks in each function to improve clarity.

2. Class Inheritance - Inheritance also makes it easier to write misleading code. Contracts' logic is spread across multiple files, decreasing readability and requires additional understanding about how inheritance works in case of conflicts.

3. Inline Assembly - Inline assembly makes it possible to manipulate variables without referencing it directly by name. This makes development and auditing more difficult and can obfuscate bugs.

4. Function overloading - Contracts with overloaded functions can be confusing. It may not be clear which function is being called in specific situations.

5. Operator overloading

6. Recursive calling - It is not possible to set an upper bound on gas limits in contracts with recursive calling

7. Infinite-length loops - It is not possible to set an upper bound on gas limits in contracts with infinite length loops

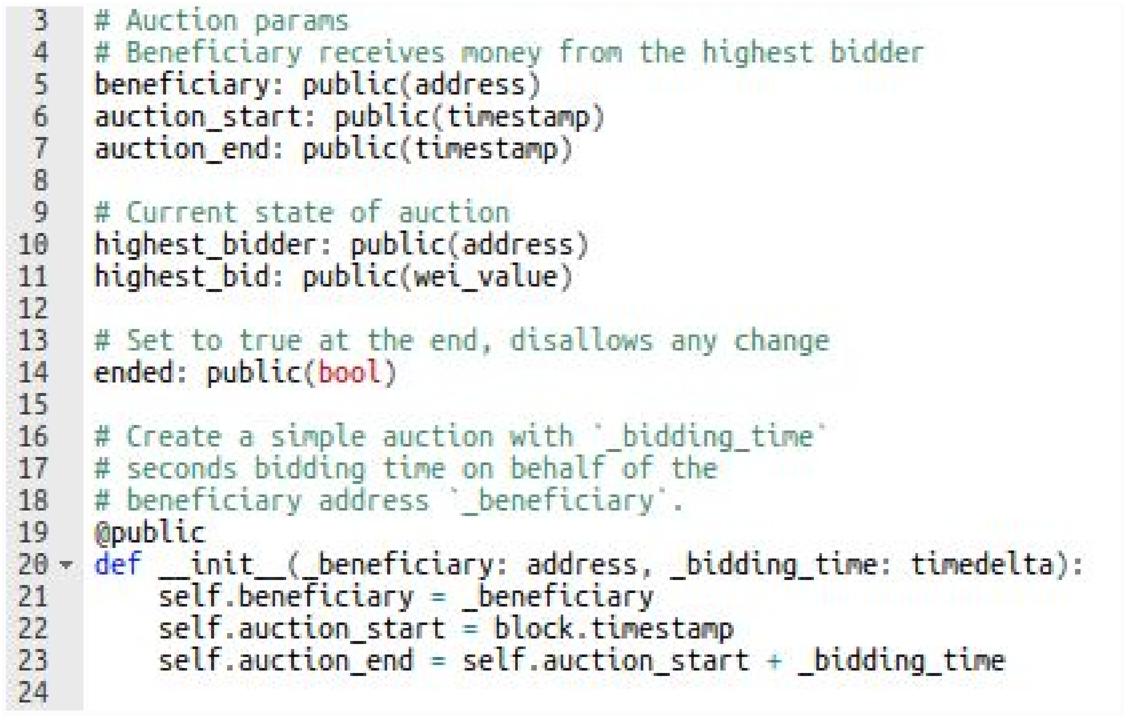
8. Binary fixed point - In binary fixed point, approximations are required (e.g. in Python 0.3 + 0.3 + 0.3 + 0.1 != 1).

As you can see in the list of features that Vyper is lacking, writing clear, understandable code is of primary importance.

Vyper is not trying to be a replacement for Solidity. It is meant to be a more security focused smart contract programming language and will likely not be able to do everything that Solidity can.

You can view the latest documentation here.

Vyper is a subset of Python syntax, making the syntax familiar to many developers.



```
# Auction params
# Beneficiary receives money from the highest bidder
beneficiary: public(address)
auction_start: public(timestamp)
auction_end: public(timestamp)

# Current state of auction
highest_bidder: public(address)
highest_bid: public(wei_value)

# Set to true at the end, disallows any change
ended: public(bool)

# Create a simple auction with `_bidding_time`
# seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
@public
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auction_start = block.timestamp
    self.auction_end = self.auction_start + _bidding_time

```

In Vyper, whitespace matters, so pay attention to spaces and tabs. Comment code using ‘**#**’.

Vyper compiles to LLL.You can explore examples interactively here: https://vyper.online.

Useful links:

Vyper Documentation

Vyper browser IDE

How I learned to stop worrying and love Viper — Part 1

How I learned to stop worrying and love Vyper — Part 2

How I learned to stop worrying and love Vyper — Part 3

An Early Look At Vyper - John Mardlin

****

This document is the adaptation of the presentation slides into a more easily readable format for reading.

Ethereum Improvement Proposals

What are Ethereum Improvement Proposals (or EIPs)?

Ethereum is an open source project with no one company or organization that controls the direction of the project. There is an open governance model where everyone is free to propose and discuss changes to the system. The EIP system is the forum to do this. You can view the EIP website here.

Some notable features of the EIP system are the submissions are open to everyone with a github account, which is very easy to get. The incorporation of a proposal into the platform is dictated by the quality of the idea rather than the reputation of the proposer.

There are several different stages that an EIP can be in. Draft EIPs are works in progress, are open for consideration and discussed on github. Accepted EIPs can be expected to be included in the next hard fork. Final EIPs are proposals that have already been adopted and deferred EIPs are not being considered for immediate adoption, but may be considered again in the future.

The structure of an EIP is important for clear and effective communication among so many project participants. An EIP includes the following sections:

- Preamble (metadata)
- Simple summary
- Abstract - A short description of the issue
- Motivation - Why is the existing protocol inadequate
- Specification - Describes the syntax and semantics of the new feature
- Rationale - Why did you make these design decisions?
- Backwards compatibility - Explain any backward incompatibilities and how they will be addressed
- Test Cases - These are mandatory for EIPs proposing consensus changes
- Implementation - What does an implementation of the EIP look like? This section must be completed before the EIP is given the status “Final”.

In practice, this system creates a set of community standards for changing the project.

EIP-20, a Standard Token Interface

Let’s look at one of the most popular EIPs, EIP-20 which specifies a standardized token interface. The implementation specification enables easy interoperability between Ethereum tokens and allows applications with complex functionality to reliably handle Ethereum tokens.

The development of tokens on Ethereum is under active development. The following EIPs are in the draft phase. Explore these EIPs to follow the development of token standards.

EIP 721 a non-fungible token standard

EIP 918 a minable token standard

EIP 777 a new advanced token standard

EIP 1046 ERC20 metadata extension

EIP 1080 Recoverable Token

Useful links:

Ethereum EIPs on Github

Walkthrough of an Ethereum Improvement Proposal

EIP-20 on Github

ERC20 on Ethereum Wiki

EIP-721 on Github