DRAFT: Clarity setup and contract structure

Part 1

Phillip G. Bradford*

November 28, 2024

Abstract

Smart contracts are distributed programs. These programs are executed by concensus on blockchains. *Clarity* is a Lisp-like language for writing smart contracts. Clarity smart contracts are on the Stacks chain which is itself on the Bitcoin blockchain.

To get started, these notes give basics of Clarity. Clairty contracts are set up for testing locally using *clarinet*.

1 Clarity setup

Proof-of-work blockchains are discussed in detail in [2]. There are a number of good sources of information on Clarity. Several are here,

https://clarity-lang.org/
https://docs.stacks.co/docs/clarity/
https://github.com/clarity-lang/reference
Clarity of Mind book [4]
https://metaschool.so/articles/guide-clarity-smart-contract-programming-language/
Videos [5, 6]

These notes offer a basic start in the Clarity smart-contract language.
Using a browser, go to the *hiro* website

*phillip.bradford@uconn.edu, phillip.g.bradford@gmail.com, University of Connecticut, School of Computing, Stamford, CT USA

```
https://platform.hiro.so
```

Hiro also built clarinet:

https://docs.hiro.so/clarinet/getting-started

Working from `https://platform.hiro.so` gives a cloud-based experience.

1. Open `https://platform.hiro.so` in a web-browser. It may be easiest to login using a google email account.
2. Connect `https://platform.hiro.so` to an editor or IDE. For example, it is easy to connect to *Microsoft VS Code*.

If you are using Microsoft VS Code, then install the Clarity plugin. See Figure 1.

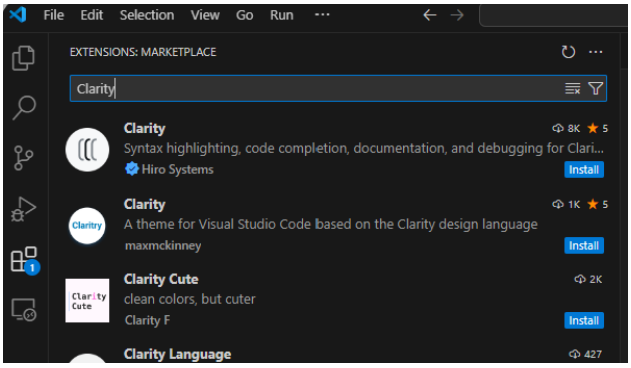

Figure 1: Microsoft Visual Studio Code Clarity Plugin

1.1 Command-line Clarinet

Clarity also runs directly from a *native clarinet command-line* on your personal machine. To directly install Clarity on Windows,

```
Windows> winget install clarinet
```

On a Mac, use *homebrew* to install *clarinet*.

```
MacOS> brew install clarinet
```

Listing 1: Clarinet console on Windows

```
Windows> clarinet console
[details]
>>
```

The "⟩⟩" is the clarinet prompt.

Clarinet can directly create a contract in a project.

Listing 2: Clarity project

```
windows> clarinet new proj01
windows> cd proj01
windows> clarinet contract new c1
```

Listing 3 was created by `Clarinet`. This listing outlines sections for key parts of a contract. These sections are delineated by comments ';;'.

Listing 3: Stubs for a Clarity contract

```
;; traits
;;

;; token definitions
;;

;; constants
;;

;; data vars
;;

;; data maps
;;

;; public functions
;;

;; read only functions
;;

;; private functions
;;
```

Each of these contract parts has value. The first section is the `public functions`.

2 Clarity contract structure and valuation

Clarity is a Lisp-like language so Clarity uses prefix notation for expressions. In prefix notation, each function or operator is placed first in a parenthesized expression. This is the prefix. The subsequent elements in the parenthesized expression are arguments to the funciton or operator. This operator or function is applied to the evaluated operands.

Clarity is strongly typed. Since Clarity is a functional lisp-like language, all type definitions are either from special-forms or as formal parameters of functions.
Example special forms include `define-public` and `define-data-var`.

Function parameter types are declared after the variable is named.

Take the next parenthesized expression. It has function or operator f that takes $k \geq 0$ arguments.

```
(f a1 ... ak)
```

All elements of the list `(f a1 ... ak)` are fully evaluated. An element is fully evaluated if it cannot be evaluated any further.

Each argument is fully evaluated before f can be applied to all arguments.

Listing 4 gives some examples.

Listing 4: Clarity + operator

```
Windows> clarinet console
>> (+ 1 2)
3
>> (+ 1 2 3)
6
>> (+ 1 2 3 4)
10
```

If a function or operator f takes no arguments, then it is invoked as,

```
(f)
```

A few more examples,

Listing 5: Clarity prefix operations for the - operator

```
>> (- 1 2)
-1
>> (- 1 2 3)
-4
>> (- 1 2 3 4)
-8
```

The next example shows cases where each argument must be evaluated before the initial operator is applied.

Listing 6: Clarity prefix operations

```
>> (+ 1 (* 2 3))
7
>> (+ 1 2 (* 3 4))
15
>> (+ (* 2 (+ 1 2 3)) (* 2 3))
18
>> (+ 1 2 (- 3 4) 5)
7
```

3 Hello world

Consider a *VS Code* terminal or a native *clarinet* command-line. Listing 2 shows the creation of a project and a contract.

In the terminal go to the *proj01/contracts* directory.

```
Windows> cd proj01\contracts\
```

Edit the c1.clar file and in the *public function* section, add

Listing 7: Clarity 'Hello world'

```
(define-public (hello)
               (ok "Hello world"))
```

Public Clarity functions must return one of these,

```
(ok ...)
(err ...)
```

This is why the `(hello)` function in Listing 7 returns `(ok "Hello")`.

Back to Clarity,

```
Windows\..\proj01\contracts> cd ..
Windows\..\proj01>
```

Contracts are run by the Clarity function `contract-call?`. This can be done in Clarity console,

```
Windows> pwd
..\proj01\
Windows> clarinet console
>> (contract-call? .c1 hello)
(ok "Hello")
```

Listing 8 shows a Clarity function parameter type. These types are declated immediately *after* the formal parameter variable. The type `(string-ascii 10)` immediately follows the formal parameter `name`.

Listing 8: Clarity 'Hello name' public function

```
(define-public (hello-name (name (string-ascii 10) ))
        (ok (concat "hello " name)))
```

If a string larger than 10 ASCII characters is passed ot the function `hello-name`, then Clarity gives an error.

Clarity's `concat` function takes two arguments.

Place this code in contract c1.clar and load it in the console,

```
Windows> clarinet console
>> (contract-call? .c1 hello-name "my friend")
```

```
(ok "Hello my friend")
```

4 Notes

Lisp is the functional language with the most staying power. There is extensive work on Lisp and related functional languages. Lisp languages incude *Common Lisp*, *Closure*, *Scheme*, and many others.

Scheme lisp played an important role in education. Scheme is simpler than many other Lisps. Two classics on Scheme lisp are [1, 3].

5 Exercises

1. Since Clarity's `concat` function takes only two arguments, show how to use several invocations of `concat` to concatenate three, four and five strings.

2. Add a second formal parameter to the public-function `hello-name`. With its two formal parameters modify the function so it does the following given the two actual parameters shown,

```
>> (hello-name Max Erwin)
(ok "Hello Max my name is Erwin")
```

3. Define a public function that takes two int parameters and returns their sum.

References

[1] Harold Abelson, Gerald Jay Sussman, with Julie Sussman: *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996.

[2] Phillip G. Bradford: *Chains that bind us*, Self-published December 2023. Available on Amazon.

[3] Daniel P. Friedman and Matthias Felleisen: *The Little Schemer*, Fourth edition, MIT Press, 1996.

[4] Marvin Janssen with contributions of lnow, Mike Cohen and Albert Catama: *Clarity of Mind*, https://book.clarity-lang.org/ 2023-04-11.

[5] Kenny Rogers: *Building an NFT with Stacks and Clarity*, https://blog.developerdao.com/building-an-nft-with-stacks-and-clarity, 2022-09-01.

[6] Kenny Rogers, Joe Bender: Stacks developer workshop: *Web3 for Bitcoin: The What, Why, and How of Building on Stacks*. Web3 for Bitcoin. Wed, Jun 29, 2022.