CLIP - A Crytographic Language with Irritating Parentheses

Author:
Duan Wei
wd2114@columbia.edu
Yi-Hsiu Chen
yc2796@columbia.edu

Instructor:
Prof. Stephen A. Edwards

July 24, 2013

Contents

1 Introduction

This manual describes CLIP, which is a programming language designed specifically for cryptographers. The purpose of CLIP is to provide users with an efficient and simple programming language to manipulate cryptographic operations, create innovative cryptographic algorithms and implement existing cryptographic protocols. As inheriting the mathematical nature of cryptography, CLIP is designed as a functional language. Its syntax is inspired by the classical functional language Lisp, but with more special types and operations to implement cryptographic algorithms. Essentially, CLIP allows cryptographers to build programs with more convenience than many other popular programming languages.

2 Lexical Conventions

CLIP has five classes of tokens: identifiers, keywords, constants, functions and other separators. Blanks, horizontal and vertical tabs, newlines and comments are ignored except as they separate tokens. Some whitespace is required to separate otherwise adjacent identifiers, keywords and constants.

2.1 Notation Conventions

Through the document, keywords are in `magenta` monospace, and built-in functions are in `cyan` monospace. Regular expression-like constructs are used to present the grammar.

2.2 Comments

A single line comment should begin with ∼∼, and ends till the end of that line is encountered. A multi-line comment should start with ∼∼∼ and end with ∼∼∼. Comments can nest, but they do not occur within a string or character literals.

2.3 Identifiers

An identifier is a sequence of alphabetic letters, digits and hyphens, with the first one being an alphabetic letter.

2.4 Keywords

The following identifiers are reserved as keywords and build-in functions. They should not be used as identifiers.

- `int`, `bit#`, `string`, `defun`, `defvar`
- `let`, `set`, `if`, `mod`, `pow`, `inverse`, `and`, `or`, `not`
- `less` , `greater`, `leq`, `geq`, `eq`, `neq`
- `parity`, `group`, `merge`, `make-vector`, `map`, `reduce`, `transpose`
- `zero`, `lambda`, `rand`, `int-of-bit`, `print`

3 Type

3.1 Basic Data Types

In CLIP, there are three non-function basic data types

- `int`: An integer with unlimited precision.
- `bit#`n: A bit sequence consist of n bits.

- `string`: A series of characters.

Note that, specially, the type `bit#1` is also used as boolean value, which is the return type of logic functions. `'0` means false and `'1` means true.

3.2 Derived Data Types

In addition, there is a derived type which is denoted as:

type`[`*index-1*`][`*index-2*`]...[`*index-n*`]`

This suggests a vector, with type being the type of elements consist of vector, and `[`*index-1*`][`*index-2*`]...[`*index-n*`]` being a series of positive whole numbers enclosed in square brackets, indicates the dimensionality of the vector. The element in the vector should be in the same type. The number of square brackets suggests the dimension of vector. For examples, `bit#2[5][5]` is a vector with two dimensions, while each dimension is five, and the elements in the vector are all 2-bit sequences.

The following construct:

vector-name`[`*index-1*`][`*index-2*`]...[`*index-n*`]`

is used to access a particular element in a vector. While *index-1* indicates that the element is at position *index-1* in the first dimension, *index-2* indicates that the element is at position *index-2* in the second dimension, etc. All index start with 0.

4 Syntax

4.1 Variable Declaration

In general, when we define a non-function data type, we use the keyword devar as following

`defvar` *identifier*`:`*type value*

The scope of the identifier defined by defvar begins directly after declaration and exists throughout the whole program. Here are some examples of binding global variables to different types of values

- `defvar n:int = 17`
- `defvar b:bit#3 = '001`
- `defvar s:string = "ThisIsAString"`
- `defvar v:bit#8[2][3] = {{'000 '010 '010} {'100 '000 '101}}`

4.2 Function Declaration

```
defun function-name:return-type argument-1:type-1 ... =
  expression;
```

The scope of the function defined by `defun` begins directly after declaration and exists throughout the whole program. Here is an example of calculating Fibonacci number

```
defun Fib:int n:int =
  (if (< n 1)
    1
    (+ (Fib (- n 1)) (Fib (- n 2))));
```

`(- n 1)` means calling function – with `n` and `1` as parameters, so it will return `n-1`. More details about function evaluation are discussed in next sub-section

4.3 Function Evaluation

Functions are evaluated with the following construct:

`(`*function-name expression-1 expression-2 ... expression-n*`)`

where *expression-1 expression-2 ... expression-n* are arguments passed into the function. The arguments must match the number and type of the arguments in the declared function being called. Functions are evaluated when they are called.

Example:

```
(merge a)
```

5 Expressions and Functions

There are three kinds of primary expressions as belows.

- *identifier*: Its type is specified by its declaration.
- *constant*: Its type is int, bits or string.
- *identifier*[*expression*]: An identifier followed by an expression in square brackets is a primary expression that yields the value at the int index of a vector.

There are two ways to create a compound expression.

- function call: The format is

  `(`*expression-1 expression-2 ... expression-n*`)`

  The first expression should be an identifier of a function call or a function call expression which returns a function.

- special expression: The format is

  ```
  <expr-or-type ... >
  ```

  This kind of compound expression is only used in some build-in function (like `let`, `lambda` and `make-vector`), several arguments will be grouped as single expression as an argument.

  Below, we will introduce the build-in functions of CLIP

5.1 Binding Functions

5.1.1 `let`

Values are bound to names through the construct

```
(let <identifier-1:type-1 value-1> ... <identifier-n:type-n value-2>
  expression)
```

which evaluates expression and binds the *value-i* to the *identifier-i* specified in the angle brackets. The scope of the value bound to the identifier through let begins directly after declaration and diminishes when the effect of let ends. Example:

```
$ (let <n 1>
    (+ n 1))
-> 2
```

5.1.2 `set`

The `set` function changes the value of variable that already binded and returns the value.

```
(set identifier value)
```

Example:

```
$ defvar a:int 17;
$ a
-> 17
$ (set a 23);
-> 23
$ a
-> 23
```

The first expression returns 17, the next two expression returns 23.

5.2 If-Else

The conditional expression evaluates to the second expression if the first is true, otherwise it evaluates to the third expression. If-else statement could be nested.

```
(if bit#1
  expression-1
  expression-2)
```

5.3 Arithmetic Functions

5.3.1 +

Addition function can have many arguments. It will return the sum of all arguments. The arguments can be an integer or the bit sequence. But the bit sequence need to have same length. Otherwise, it causes error. If all inputs are integers, the return type is also integer. If there is a bit sequence, then the return type is the bit vector with the same length. And the overflow part will be cut off.

```
(+ int-or-bits-1 int-or-bits-2 ... int-or-bits-n)
```

Example:

```
$ (+ 9 "abc");
-> error: the input type of function + is invalid
$ (+ 'b0011 19 'b0001)
-> 'b0111
```

5.3.2 *

Multiplication function can have many arguments. Every input should be an integer. The type return value is also an integer.

```
(* int-1 int-2 ... int-n)
```

5.3.3 -, /, mod, power, inverse

The subtraction, divide, module, power and inverse functions can only take two integer as arguments and returns another integer.

```
(- int-1 int-2)
(/ int-1 int-2)
(mod int-1 int-2)
(pow int-1 int-2)
(inverse int-1 int-2)
```

Example:

```
$ (- 23 17)
-> 6
$ (/ 9 4);
-> 2
$ (mod 23 17)
-> 6
$ (pow 3 5)
-> 243
$ (inverse 17 23); ~~ return a value x such that 17 * x = 1 (mod 23)
-> 19
```

5.4 Logic functions

5.4.1 and, or

The logic function and, or takes a number of arguments of the type bit#1 and yields a value of true, if the specified relation is true, or a value of false otherwise.

```
(and bit#1-1 bit#1-2 ... bit#1-n)
(or bit#1-1 bit#1-2 ... bit#1-n)
```

5.4.2 not

The logic function not takes only one argument of the type bit#1 and returns the opposite value.

```
(not bit#1)
```

5.5 Comparison Functions

5.5.1 less, greater, leq, geq

The comparison function less, greater, leq and geq take only two integers as arguments and compares expressions from left to right. Their return value is either true ('1), if the specified relation is true, or false('0) otherwise. leq means less than or equal with, while geq means greater than or equal with. The type of both arguments should be integers.

```
(less int-1 int-2)
(greater int-1 int-2)
(leq int-1 int-2)
(geq int-1 int-2)
```

5.5.2 eq, neq

The comparison function eq and neq takes only two arguments and compares expressions from left to right. The return value is true, if the specified relation is true, or false otherwise. eqmeans two arguments compared are equal while neq means not equal. Both arguments can be any types.

```
(eq expression-1 expression-2)
(neq expression-1 expression-2)
```

5.6 Bit Functions

5.6.1 &

The bitwise function & takes arguments of the same length bit sequences and yields a bit sequence which is the bitwise sum of all the inputs.

```
(& bit-sequence-1 bit-sequence-2 ... bit-sequence-n)
```

5.6.2 |

The bitwise function | takes arguments of the same length bits sequences and yields a bit sequence value which is the bitwise or of all the inputs.

```
(| bit-sequence-1 bit-sequence-2 ... bit-sequence-n)
```

5.6.3 ^

The bitwise function ^ takes arguments of the same length bits sequences and yields a bit sequence value which is the bitwise xor of all the inputs.

```
(^ bit-sequence-1 bit-sequence-2 ... bit-sequence-n)
```

5.6.4 parity

The input of parity function is a bit sequence. It will return one bit to indicate the parity of input.

```
(parity bit-sequence)
```

Example:

```
$ (parity '110101101)
-> '0
```

5.7 Shift and Rotation

5.7.1 <<, >>

The shift operators << (left shift) and >> (right shift) are binary operators. The first expression is the bit type value to be shifted, and the second expression is an integer which indicates the number of bits to be shifted. Zeros will be added into those empty bits result from shifting.

```
(<< bit-sequence-1 bit-sequence-2)
(>> bit-sequence-1 bit-sequence-2)
```

Example:

```
$ (<< '1111 2)
-> '1100
```

5.7.2 <<<, >>>

The rotation operators <<< (left rotate) and >>> (right rotate) are binary operators. The first expression is the bit type value to be rotated, and the second expression is an integer which indicates the number of bits to be rotated.

```
(<<< bit-sequence-1 bit-sequence-2)
(>>> bit-sequence-1 bit-sequence-2)
```

Example:

```
$ (<<< '11110000 3)
-> '10000111
```

5.8 Vector Function

5.8.1 group

The group function operates on a bit sequence, it cuts the bit sequence into pieces whose lengths are indicated by another integer parameter, then returns those pieces as a bits vector.

```
(group bit-sequence-1 bit-sequence-2)
```

Example:

```
$ (group '001011110010 4)
-> {'0010 '1111 '0010}
```

5.8.2 merge

The merge operator is the "opposite" of group operator. It operates on a bits vector and merge the elements sequentially into a single large bit sequence.

```
(merge bits-vector)
```

Example:

```
$ (merge {'0010 '1111 '0010})
-> '001011110010
```

5.8.3 make-vector

The make-vector function creates an arbitrary dimension vector and returns it. The first expression after make-vector indicates the return type. The second expression generally would be a function indicates the specific elements in the vector. `@i` suggests the i-th index of the vector.

```
(make-vector <type> expression)
```

Example:

```
$ (make-vector <int[3][3]> (* (+ 1 @1) @2))
-> {{0 1 2} {0 2 4} {0 3 6}}
```

5.8.4 map

A map function applies a given function to each element of a vector.

```
(map function vector)
```

Example:

```
$ defun plus2:int x:int = (+ x 2);
$ (map plus2 {1 3 7})
-> {3 5 9}
```

5.8.5 reduce

The reduce function combines all the elements of a vector using a binary operation.

```
(reduce funtion vector)
```

Example:

```
$ (reduce + 2 {1 3 7})
-> 13
```

5.8.6 transpose

The transpose function operates on a matrix, it returns the matrix whose rows as columns and columns as rows of the original matrix.

```
(transpose 2-dimension-vector)
```

Example:

```
$ (transpose {{'00 '01 '10} {'11 '11 '11}})
-> {{'00 '11} {'01 '11} {'10 '11}}
```

5.9 Miscellaneous Functions

5.9.1 lambda

The lambda function creates a function without binding to any identifier. The first parameter is the arguments of the output function, which are enclosed by angle bracket. The second parameter is an expression which indicate how the function work.

```
(lambda <var-1, var-2, ... var-n> expression)
```

Example:

```
$ ((lambda <x> (+ x 2)) 3)
-> 5
```

5.9.2 zero

The zero function takes an integer as the parameter and returns a bit sequence with all zeros and its size is the parameter.

```
(zero int)
```

Example:

```
$ (zero 5)
-> '00000
```

5.9.3 rand

The rand generate a bit sequence randomly whose length is indicated by the integer parameter.

```
(rand int)
```

Example:

```
$ (rand 9)
-> '010011011
```

5.9.4 int-of-bits

The int-of-bits function transforms a bit sequence into an integer. The conversion is in little-endian sense.

```
(int-of-bits bit-sequence)
```

Example:

```
$ (int-of-bits '1011)
-> 13
```

5.9.5 print

The print function prints a sequence of expressions and returns a single bit '1 after succeeded in printing. The expressions can be any type.

```
(print expression-1 ... expression-n)
```

6 Grammar

Below is a recapitulation of the grammar that was given above. The precedence of expressions is the same order as they are presented below. Expressions in the same group (indicates by ;) are given the same precedence.

```
expr =
    constant; identifier
    (expr) ;<expr-or-type>; (function expr)

type-declaration =
    identifier:type
value-binding =
    defvar type_declaration = expr
function-binding =
    defun type-declaration argument-list= expr
argument-list =
    identifier:type
type =
    basic-type
    type[ ]
```