CRYPS Language Reference Manual

Hsiu-Yu Huang
hh2360@columbia.edu

Minita Shah
mjs2225@columbia.edu

Saket Goel
sg2679@columbia.edu

Sarfraz Nawaz

sn2355@columbia.edu

1 Introduction

CRYPS is a language designed to help regular users as well as programmers perform cryptographic operations . The language has support for generation, storage and processing of numbers in singular and collection forms, which is essential to implementation of common cryptography mechanisms and research on newer ones.

CRYPS is intended to be used by diverse backgrounds of people. Internet users can use it to encipher messages before exchanging confidential information over mail or instant messengers. Also, it could be used for E-commerce and banking transactions requiring web authentication. Enterprises with intranet systems can encrypt employee credentials with CRYPS. Most importantly, it can be used as a tool to develop new cryptographic algorithms.

2 Lexical Conventions

2.1 Identifiers

An identifier can consist of one or more letters, digits and underscore. The first character should be a letter followed by any sequence of letters, digits and underscore character. First ten characters are significant. Uppercase and lowercase letters are different.

2.2 Comments

Single line comments start with a # character and are terminated at the end of a line.

Multiple line comments start with ‘(#’ and are terminated with ‘#)’.

2.3 Whitespace

The only way to represent whitespace is by binding it with double quotes (“ “) on either side, all other forms of whitespace are not considered.

2.4 Reserved Keywords

if

else

while

for

switch

case

break

default

return

int

void

AND

OR

inc

1.5 Data Types

Fundamental data types are **int** and **void.** Apart from the fundamental data types, there is a class of derived types constructed from fundamental data types in the following ways:

(1) Arrays of objects of **int** - The syntax convention for an array is an identifier followed by '[' and ']'.

(2) Matrix of objects of int

(3) Functions which may or may not return objects of a given type.

1.6 Constants

CRYPS has only integer constants.

1.7 Separators

The ',' symbol is used to represent a comma separated list. The symbol ';' is used to indicate the end of a statement.

1.8 Scope Rules

CRYPS is a block-structured language, and the scope of names declared in a block is within the body of the block. That is the language using static scoping.

3 Expressions and Operators

Symbol	Operations	Associativity
+	Addition	Left associative
-	Subtraction	Left associative
*	Multiplication	Left associative
/	Division	Left associative
^	Exponential	Left associative
%	Modulus	Left associative
=	Equal to	Left associative
!=	Not equal to	Left associative
<	Less than	Left associative
<=	Less than or Equal to	Left associative
>	Greater than	Left associative
>=	Greater than or Equal to	Left associative
<<	Left Shift	Left associative
>>	Right Shift	Left associative
@	EXOR	Left associative
,	Sequence	Left associative
AND	Logical And	Left associative
OR	Logical Or	Left associative
<-	Assignment	Right associative

3 Control Structures

Conditional Statement

if (expression) statement

if (expression) statement else statement

While statement

While (expression) statement

For statement

for identifier <- expression to expression statements **inc** val statements

Switch statement

switch expression {**case** constant statements **break**; **default** statements}

4 Declaration

Declarations are used within function definitions to specify the interpretation for each identifier; Declarations have the form:

declaration:

decl-specifiers declarator-list;

decl-specifiers:

type-specifier

type-specifier:

int

void

declarator-list:

declarator

declarator, declarator-list

Declarators have the syntax:

declarator:

identifier

declarator ( )

5 Namespace

We have only one namespace and have no support for features like class definitions or *typedef*s that require additional namespaces.

6 Grammar

prg -> statements

statements -> '{'(stmt';')+ '}'

stmt -> if-stmt | while-stmt | for-stmt | switch-stmt | function-defn

|var-decl | expn | return-stmt

expn -> expn binop expn | unary-op expn |'('expn')'

| function-call | array-val | val

binop -> '+'|'-'|''|'/'|'^'|'%'|'<'|'<='|'>'|'>='|'<<'|'>>'|'@'*

|'AND'|'OR'|'!='|'='

digit -> ['0' - '9']

assignment-op -> '<-'

unary-op -> '!'|'~'|'-'

type -> int | void

array -> '[' E | expn ']'

array-val -> id '[' expn ']'

matrix -> '[' E | expn ']' '[' E | expn ']'

matrix-val -> id '[' expn ']' '[' expn ']'

if-stmt -> if expn statements

|if expn statements (elseif expn statements) else statements*

while-stmt -> while expn statements

for-stmt -> for id assignment-op val 'to' val ('inc' val)? statements

switch-stmt -> switch expn '{'(case digit+ statements break';')+

default statements'}'

function-defn -> type id '('E | parameters')' statements

parameters -> type parameter ',' parameters | type parameter
parameter -> id | array | matrix

assignment-expn -> id assignment-op expn | id array assignment-op digit-set
| id matrix assignment-op digit-mat-set
digit-list -> digit ',' digit-list
digit-set -> '{' digit-list '}'
digit-mat-set -> '{' digit-mat-list '}'
digit-mat-list -> digit-list | digit-list ';' digit-mat-list
function-call -> id '(' E | argument-list ')'
argument-list -> argument | argument ',' argument-list
argument -> val|expn
val -> id | digit+
var-decl -> type var-decl-list
var-decl-list -> id | id',' var-decl-list
| assignment-expn | assignment-expn',' var-decl-list
| array | array ',' var-decl-list
| matrix | matrix ',' var-decl-list
return-stmt -> return | return expn | return val