# Language Reference Manual: Prime

---

Alexander Liebeskind (al3853), *Project Manager*
Nikhil Mehta (nm3077), *Language Guru*
Pedro B T Santos (pb2751), *Tester*
Thomas Tran (tkt2120), *Systems Architect*

Programming Languages and Translators, Spring 2021

# Contents

# 1 Introduction

PRIME is a language based off C, specifically created for the implementation of cryptography algorithms. As a result, PRIME contains a number of features intended to facilitate the implementation of encryption and decryption schemes. The types and operators included in PRIME are primarily oriented around modular arithmetic and large number processing, given the relevance of these two topics in modern cryptography. Additionally, PRIME includes many basic features for general ease of usage.

Due to syntactical and functional similarities between PRIME and C, the C Reference Manual (https://www.bell-labs.com/usr/dmr/www/cman.pdf) is a helpful resource in understanding PRIME and its underlying mechanics.

The following manual covers the types, operators, and functionalities of PRIME, as well as quick usage cases to help illustrate programming in PRIME. The language reference manual is also subject to update pending further changes to PRIME and added language components.

### 1.0.1 CFG

The CFG for PRIME is dispersed throughout the document to help show system architecture. The CFG is found at the end of a section (when applied to multiple subsection) or subsection (when specific to a single subsection), depending on the specialization of elements.

It is ultimately heavily similar to the CFG in K&R's C Reference Manual (2nd Edition). Please note that the terminals in PRIME are: identifiers, strings, constants/literals.

When we call a different non-terminal (especially from the -expressions) it is akin to denoting the precedence (low recurses to high).

The order in which the recursion occurs will denote the associativity eg: equality expr showing up on the left of == denotes left-associativity.

A subscript of opt stands for optional. You will note that in the return statement a return value is necessary. This is because PRIME has no void type. Unless specified otherwise (eg: by "one of"), the format of the CFG is nonterminal: followed by different productions on different lines.

# 2 Types

Types in PRIME are similar to that of C and C++ with a few others that are particularly useful for cryptography operations. Note that in the below CFG, the term constant is used to refer to fundamental literal types, which are then used to create all of the basic types in PRIME.

CFG:
type-specifier: one of
  int char string pt lint poly ring

params:
  parameter-declaration
  params , parameter-declaration

parameter-declaration:
  type-specifier identifier

program:
  declarations EOF

## 2.1 Initialization

In the absence of variable assignment, types are handled as literals (here 'constants', in the CFG). Initializing variables in PRIME is similar across all types, and may or may not use the assignment operator (described in detail below).

Examples:
```
type name;
type name = value;
```

CFG: declare-init:
  type-specifier declarator;

declarator:
  identifier
  identifier = assign-expr
  declarator(params$_{opt}$)
  declarator[assign-expr]

constant:
  integer-list
  char-lit
  string-lit

## 2.2 int

Just as in C, an `int` can take signed 32-bit integer values ranging from $-2,147,483,648$ to $2,147,483,647$. `int`s can be declared with or without an original value. If it is declared without an original value, it is automatically set to 0.

Examples:
```
int foo = 4;
int bar = foo;
int foobar;
```

## 2.3 char

The `char` can take 8-bit to represent ASCII characters. We denote `char` assignment by using apostrophes for literal characters. `char`s can be declared with or without an original value. If it is declared without an original value, it is undefined.

Examples:
```
char foo = 'a';
char bar;
```

## 2.4 string

A `string` is a datatype meant to hold a sequence of ASCII characters. We denote `string` assignment by using quotations for literal sequences of characters. `string`s can be declared with or without an original value. If it is declared without an original value it is set as the empty string.

Examples:
```
string foo = "hello";
string bar;
```

## 2.5 pt

A `pt` is a datatype meant to represent a point in n-space. `pts` work similarly to one-dimensional arrays in Java. When points are assigned a value, they must be defined with regards to a specific immutable dimension of `int` $n$ surrounded by brackets. Additionally, they may also specify coordinates of `int` and/or `lint` types.

When a `pt` is initialized with a set of coordinates, a value must be given for every dimension. These values must be enclosed in parenthesis and separated by commas. A `pt` declared without a dimension and corresponding coordinates is undefined. `pts` that are initialized with a dimension but without specific coordinates are initialized to be the origin $(0, 0, \ldots, 0)$. When any of these rules are not obeyed, the behavior is undefined.

Each dimensional coordinate within a `pt` is mutable and can be changed by using brackets to specify the dimension of interest. `pt` accessing is zero-indexed. Accessing an invalid index will cause undefined behavior.

Examples:
```
pt origin[3];
pt here[2] = [132, 1];
pt nowhere;
pt here[0] = 1;
```

CFG:
integer-list:
  integer
  integer-list, integer

## 2.6 lint

A `lint` or large integer is a very large integer. It is approximately 200 digits in length to conform with current state-of-the-art cryptographic security requirements. Large integer types are usually used to hold large primes for later use in computationally expensive products or exponents in practice.

Similarly to integers, `lints` can be declared without an initial value or with one. Since integers will by definition be smaller than `lints`, they can also be assigned to each other. However, lints will only be compared to other lints for computational efficiency.

Examples:
```
lint bigNumber;
lint anotherBigNumber = 2147483659;
int smallNumber = 51;
lint bigNumber = smallNumber;

if( bigNumber > anotherBigNumber ) { /* OK */
    return 0;
} else if( bigNumber == smallNumber ) { /* ERROR */
    ...
}
return bigNumber^smallNumber;
```

## 2.7 poly

The `poly` defines a univariate polynomial function with respect to a positive `lint` modulus. The polynomial function and modulus is defined within `[]` using a comma separated series of `pt` points in two dimension space, where the first dimension denotes the coefficient, and the second dimension denotes the exponent. Each point within a `poly` requires a unique value for the second dimension, as one

coefficient can only be defined for a unique exponent. The last element within the brackets denotes a modulus with type `lint`. The modulus of the polynomial should always be greater than or equal to zero.

Examples:
```
lint l = 4;
poly quadratic = [(2, 4), (1,2), (3,0), l];
/* Defines 2x^4 + x^2 + 3 with modulus 4 */
```

### 2.8 ring

The `ring` serves as the cornerstone of our cryptography-oriented language. A `ring` is modeled after the mathematical concept of a ring from modern algebra. Our `ring`s are built to support complex expressions of modular arithmetic. Typically, a mathematical ring is defined as a set of elements, not necessarily finite, under which two operators, often called addition and multiplication, are closed. The set of integers from 0 to $n$ mod $n$ form a ring with modular addition and multiplication. Stemming from addition and multiplication, other operations like subtraction and exponentiation can easily be defined.

A `ring` is defined with respect to an input type and modulus. Both of these specifiers are immutable once the `ring` is initialized. A `ring` declared without an input type and a modulus will be null.

Once a `ring` is defined, it will support the computation of modular arithmetic expressions with respect to its immutable modulus. `int` and `lint` moduli should always be greater than or equal to zero. Computations within the ring are denoted by writing the name of the ring followed by `.comp()`. Where an expression is written within the parentheses.

`ring`s are built to automatically support a set of standard operations, discussed below, when the input type matches the modulus type. The exception to this is with `pt`s where rings will automatically support operations between `pt`s with an `int` or `lint` modulus. For less trivial combinations, for example a ring that takes in `pt`s and is defined with a `poly` modulus, the user must redefine the operators. The user may overload an operator for any `ring`.

Operator overloading works similarly to that in C++. The name of the ring is written followed by a colon :, an operator, and two obligatory variables encased within parentheses to represent the right hand side and the left hand side of the expression. The new operator expression should be written within curly braces and must return a value.

Examples:
```
ring nullring;
ring mod7 = (pt[2], 7);
pt newpt[2] = ring.comp((3,5) + (2,6)); /* newpt = (5,4) */

mod7:+(lhs, rhs){return lhs - rhs;} /* addition changed to subtraction */
newpt = ring.comp((3,5) + (2,6)); /* newpt = (1, 6) */
```

## 3 Operators

### 3.1 Unary Operators

Unary operators include ! and -. Unary operators take precedence above all other mathematical operators.

CFG:
unary-operator: one of
- -  !

### 3.1.1 Not: !

$$! \; expression$$

The logical negation operator is used to obtain the logical opposite of a value. It works on `int`s and `lint`s. Since there are no booleans in PRIME, true and false may be represented by 0 and non-zero integers respectfully. If the expression evaluates to a non-zero integer, ! *expression* will return a 0. If the expression evaluates to zero, ! *expression* will return 1.

### 3.1.2 Negative: -

$$- \; expression$$

The negative unary operator is used to obtain the mathematical opposite of a value. It works on `int`s, `lint`s, `pt`s, and `poly`s. When performed on an `int` or `lint`, the opposite value will be returned. I.e. 1 becomes -1 and vice versa. When performed on an `pt`, each component of the point is multiplied by negative one. When performed on a `poly`, each coefficient of the `poly` is multiplied by negative one. The return type is the same as the expression type.

## 3.2 Multiplicative Operators

Multiplicative operators include multiplication, division, and modulo.

CFG:
multiply-expr:
  unary-expr
  multiply-expr * unary-expr
  multiply-expr / unary-expr
  multiply-expr % unary-expr

### 3.2.1 Multiplication: *

$$expression * expression$$

The multiplication operator will be used to obtain products of large primes mainly. It is one of the arithmetic operators and will function on `int`s, `lint`s, `pt`s, and `poly`s. `int`s and `lint`s may be multiplied to one another and will return the `lint` type.

If one of the expressions is type `pt`, the other expression must be an `int` or `lint`. In this case each component of the `pt` will be multiplied by an integer value. This is akin to scaling a vector. If one expression is a `poly` with base modulus $n$ the other expression must also be a `poly` with base modulus $n$

Example:
```
lint one = 2147483659;
lint two = 1837641987193287;
one * two; /* is a lint */

int three = 3;
int four = 4;
three * four; /* is an int */

one * three; /* is a lint */
```

### 3.2.2 Division: /

$$expression \; / \; divisor \; expression$$

The division operator will be used to obtain the integer quotient of large primes mainly. It will function on `int`s, `lint`s, `pt`s, and `poly`s. `int`s and `lint`s may be divided by one another and will return the `lint` type. If the divisor expression does not divide the left hand side expression, the quotient will be truncated in the appropriate way based on the type to provide integer values, components, or coefficients.

Divisor expressions may not be of type `pt`. If the left hand expression is of type `pt`, then the divisor expression must be of type `int` or `lint`. `poly`s may only divide and be divided by other `poly`s with the same base modulus.

#### 3.2.3 Modulo: %

*expression % modulus expression*

The modulo operator will be used to obtain remainder from integer and polynomial division. It will function on `int`s, `lint`s, `pt`s, and `poly`s. Expressions of type `int`, `lint`, or `pt` must have a modulus expression that is either an `int` or `lint`. `poly` expressions with base modulus $n$ must have a modulus expression also of type `poly` with base modulus $n$. The return type will always be the same type as the expression (on left hand side).

Example:
```
lint one = 2147483659;
int a = 1837641987193287;
one % a; /* is a lint */
a % one; /* is an int */

pt[3] q = (1, 3, 4);
int two = 2;
q % two; /* is a pt */
```

#### 3.2.4 Exponent: ^

*base expression^exponent expression*

The exponent operator will be used to compute the repeated multiplication of integer values. It will function on `int`s and `lint`s. `int`s and `lint`s may be raised to each other. The return type will be the type of the base expression.

CFG:
unary-expr:
  power-expr
  unary-operator power-expr

power-expr:
  postfix-expr
  postfix-expr ^ power-expr

### 3.3 Additive Operators

Additive operators include addition and subtraction.

CFG:
add-expr:
  multiply-expr
  add-expr + multiply-expr
  add-expr - multiply-expr

#### 3.3.1 Addition: +

*expression + expression*

The addition operator will be used to obtain integer sums. It will function on `int`s, `lint`s, `pt`s, and `poly`s. `int`s and `lint`s may be added to one another and will return the `lint` type. `pt`s of dimension

$n$ may only be added to other `pts` of dimension $n$. When `pts` of dimension $n$ are added to one another, their respective components are summed to return a new point of dimension $n$. `polys` of base modulus $n$ may only be added with other `polys` with the same base modulus $n$. When `polys` are added to other `polys`, their sum is calculated by adding coefficients of the same degree mod $n$. The return type is a `polys` of base modulus $n$.

#### 3.3.2 Subtraction: -

$$expression \text{ - } expression$$

The subtraction operator is used to obtain integer differences. It will function on `ints`, `lints`, `pts`, and `polys`. `ints` and `lints` may be subtracted from one another and will return the `lint` type. `pts` of dimension $n$ may only be subtracted from other `pts` of dimension $n$. When `pts` of dimension $n$ are subtracted from one another, their respective components are subtracted to return a new `pt` of dimension $n$. `polys` of base modulus $n$ may only be subtracted from other `polys` of base modulus $n$. When `polys` are subtracted from other `polys`, their difference is calculated by subtracting coefficients of the same degree mod $n$. The return type is a `polys` of base modulus $n$.

### 3.4 Relational Operators

$$expression < expression$$

$$expression > expression$$

$$expression <= expression$$

$$expression >= expression$$

The relational operators (`<, >,<=, >=` ) denote less than, greater than, less than or equal to, and greater than or equal to, respectively. These operators compare two expressions of type `int` or `lint` and will return an integer value 1 (true) or 0 (false) based on a relational comparison in $\mathbb{R}$. These relational operators are equivalent in precedence and have a lower precedence than multiplicative and additive operators.

CFG:
relation-expr:
  relation-expr < add-expr
  relation-expr > add-expr
  relation-expr <= add-expr
  relation-expr >= add-expr

### 3.5 Equality Operators

$$expression == expression$$

$$expression \,!= expression$$

The equality operators (`==, !=`) denote equals and not equals, respectively. These operators compare two expressions of type `int`, `lint`, `pt`, or `poly`. The `==` operator returns an integer value 1 (true) if the two expressions are equivalent in $\mathbb{R}$, and 0 (false) otherwise. The `!=` operator returns an integer value 1 (true) if the two expressions are **not** equivalent in $\mathbb{R}$, and 0 (false) otherwise. These equality operators are equivalent in precedence and have a lower precedence than relational operators.

CFG:
eq-expr:
  relation-expr
  eq-expr == relation-expr
  eq-expr != relation-expr

### 3.6 Logical Operators

*expression* && *expression*

*expression* || *expression*

The equality operators (`&&, ||`) denote logical and and logical or, respectively. These operators compare two expressions of type `int`. The `&&` operator returns an integer value 1 (true) if the two expressions are nonzero, and 0 (false) otherwise. The `||` operator returns an integer value 0 (false) if the two expressions are zero, and 1 (true) otherwise. These logical operators are evaluated from left to right, equivalent in precedence and have a lower precedence than equality operators.

CFG:
logic-OR-expr:
  logic-AND-expr
  logic-OR-expr || logic-AND-expr

logic-AND-expr:
  eq-expr
  logic-AND-expr && eq-expr

### 3.7 Assignment Operators

*variablename* = *expression*

The equals operator is the assignment operator in our language. Once the right hand expression has been evaluated, it is stored in the variable expression. The expression to the left of the equals sign must be a variable expression.

CFG:
assignment-operator:
  =

assign-expr:
  logic-OR-expr
  unary-expr assignment-operator assign-expr

## 4 Precedence

Precedence in PRIME is defined as follows. If two operators of equal precedence are present, they will be parsed from left to right within a line and top to bottom if longer than a line.

| | |
|---|---|
| 1 | ( ) |
| 2 | – (Unary Operator), ! |
| 3 | *, /, % |
| 4 | +, – (Subtraction) |
| 5 | <, <=, >, >= |
| 6 | ==, != |
| 7 | ^ |
| 8 | &&, \|\| |
| 9 | = |

## 5 Lexical Conventions

### 5.1 Key Words

PRIME has several keywords that are reserved in the language to prevent ambiguity.

#### 5.1.1 Type Related Keywords

- int
- lint
- pt
- poly
- ring
- string
- char

#### 5.1.2 Statement Related Keywords

- if
- else
- for
- while
- return

### 5.2 Variable Names

Variables must be named starting with a letter a through z lower or uppercase followed by any number of alphanumeric characters or underscores.

### 5.3 Comments

comments in Prime are initiated using the character sequence /* and concluded using */. All text between the /* and the */ will be ignored by the compiler. This commenting format applies to both single line and multiline comments.

## 6 Statements and Expressions

The following CFG components are included in this document to illustrate how handling general structural elements (i.e. statements that are not confined to a specific section above or may be employed in multiple cases) occurs in PRIME. This includes processing statements and expressions (which could occur in functions, loops, conditionals, or any other part of a program), and managing argument lists.

CFG:
stmt:
  expr-stmt
  sequential-stmts
  control-flow-stmt
  loop-stmt
  return-stmt

expr-stmt:
  $\text{expr}_{opt}$;

sequential-stmts:
  {$\text{declarations}_{opt}$ $\text{stmts}_{opt}$}

expr:
  assign-expr

expr, assign-expr

primary-expr:
identifier
constant
( expr )

args-exprs:
assign-expr
args-exprs, assign-expr

# 7 Program Structure

The following structural elements dictate the control flow of a program in PRIME.

## 7.1 Conditionals

Conditionals in PRIME come in two forms, both of which begin with an if statement:

`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

In each of the above cases, the *expression* following the initial `if` statement is evaluated and if the value returned is nonzero, the following *statement* is executed. The `else` keyword allows for an alternative statement to be executed if the *expression* evaluates to zero. Each `else` will be paired to the last encountered if that is not already paired with an `else`.

CFG:
control-flow-stmt:
stmt-dangling
stmt-nodangle

stmt-dangling:
if(expr) stmt
if(expr) stmt-nodangle else stmt-dangling

stmt-nodangle:
if(expr) stmt-nodangle else stmt-nodangle
expr-stmt
loop-stmt
sequential-stmts
return-stmt

## 7.2 Loops

Loops are used for iteration in PRIME. They may be implemented as while or for loops, described in detail here.

CFG:
loop-stmt:
while(expr) stmt
for($expr_{opt}$;$expr_{opt}$;$expr_{opt}$) stmt

### 7.2.1 While

While loops in PRIME are of the form:

`while` (*expression*) *statement*

This functions similarly to an `if` statement, except for the fact that the statement following a `while` statement will be repeatedly executed for as long as the expression following the `while` evaluates to a nonzero value. For this reason, it is essential that the expression be updated at some point over the course of the `while` loop, to prevent infinite iteration. The evaluation and check of the expression occurs at the beginning of each iteration.

### 7.2.2 For

For loops in PRIME are of the form:

`for` `(` *expression1* `;` *expression2* `;` *expression3* `)` *statement*

*expression1* is executed prior to the first iteration of the loop. *expression2* is evaluated prior to every iteration of the loop, and the loop is entered if *expression2* evaluates to true. *expression1* is executed prior to every iteration of the loop. *statement* is executed during every iteration. Note that each of the expressions is optional as dictated by the CFG above.

### 7.2.3 Loop Equivalence

As an example of loop equivalence between while and for loops in PRIME, the following implementations are identical in functionality:

*expression1* `;`
`while` (*expression2*) {
*statement* `;`
*expression3* `;`
}

`for` `(` *expression1* `;` *expression2* `;` *expression3* `)` *statement*

CFG:
loop-stmt:
  while(expr) stmt
  for($expr_{opt}$;$expr_{opt}$;$expr_{opt}$) stmt

## 7.3 Functions

Functions in PRIME are of the form:

`return-type function-name (parameters)` *statement*

Note that all functions in PRIME must return some value, and the *statement* (i.e. the function body) must therefore contain some return line. The return statement is the keyword return followed my a space and the return expression followed by a semicolon.

```
return expression;
```

CFG:
function-definition:
  type-specifier declarator sequential-stmts

return-stmt:
  return expr;

# 8 Scope

## 8.1 Curly Brackets

Scope is generally denoted as space between curly braces. Variables declared there will be accessible within that scope. "constants" or variables that should be treated as such should be declared in the main function of a particular program rather than outside/at the top of the program. Additionally, in the below examples of conditionals, functions, and loops, it should be noted that statement is usually contained within curly brackets, so as to denote the scope of the statement.

Example:
```
main() {
  /* declare your constants here and pass them */
  int n = 57; /* OK */
  n = 58; /* Also ok */
  int n = 59; /* ERROR */
  lint n = 60; /* ERROR */
}
```

## 8.2 Semicolons

A semicolon character denotes the end of an expression and the start of the next (since whitespace will be ignored). This applies to loops, declarations, if statements etc.

Example:
```
int n = 3;
int n2 = 4   + 3
/* still going...  */
+2; /* done */
```

# 9 Input and Output

PRIME will support input and output through the normal channels of `stdin`, `stdout`, and `stderr`. These components are not yet included in the parser, but are set to be added in the next iteration of added features as building the language progresses.

## 9.1 print()

The `print(string out)` function takes a single string and writes it to `stdout`. print() returns a 1 upon success, and a 0 otherwise.

## 9.2 scan()

The `scan())` function reads from `stdin` and returns a string.

# 10 Additional Notes on Cryptography

The basis for all modern cryptography is mathematical trapdoor functions. That is, mathematical operations that are easy to do in one direction but very difficult to do in the backwards (inverse) direction. Modular arithmetic using large primes is at the heart of many trapdoor functions used in industry today so our language will focus on making this kind of arithmetic easy to implement.

The mathematical theory behind cryptography comes from abstract and modern algebra and the study

of group theory. A mathematical group is a set of elements that are closed under some operator, say addition or multiplication. Furthermore, there exists an identity element in the group where every element * the identity element will return the original element. Lastly, for every element in the group, there exists an inverse element where the element * the inverse element will return the identity element. A group could be defined as the set of all integers under addition. In this case the identity element would be 0. The inverse of an element would just be the negative (opposite) version of itself e.g. 2 and -2.

Our language incorporates the idea of a mathematical ring which is similar to a group with two exceptions. First, the ring is closed under two operations as opposed to just one. These operations are often denoted as + and *. Second, not every element must have an inverse in the ring. For modular arithmetic, this more generous description of a set lends itself well for PRIME's use cases. For instance the set of all integers modulo 20 with operation multiplication would have identity element 1. But then even integers don't have an inverse in the set. So it is necessary to use a looser more inclusive algebraic space. From the two operators a ring is defined under, typically addition and multiplication, additional operators can be defined like subtraction or exponentiation.