IPCoreL

Programming Language Reference Manual

Phillip Duane Douglas, Jr.

11/3/2010

The IPCoreL Programming Language Reference Manual provides concise information about the grammar, syntax, semantics, and functionality of this network programming language. The intent of IPCoreL is to provide an intuitive network programming experience to beginner, novice network enthusiasts and professionals.

1 Introduction to IPCoreL

The IPCoreL programming language reference manual contains a summary of the grammar, syntax, semantics, and functionality of the programming language. IPCoreL provides basic arithmetic operations, conditional statements, iterative statements, relational expressions, declarations and declarators, and type specification. The main feature of IPCoreL is the ability of the language to perform network calculations, socket calls, and transmission of custom data packets. The intent of IPCoreL is to provide beginner and novice network programmers an intuitive, easy-to-implement programming language with none of the difficulties found in other languages, e.g. C, Java, and Python. Some of the basic operations of IPCoreL are listed below.

- Network Performance Simulation
  - Throughput Calculation
  - Latency Calculation
  - Jitter Calculation
  - Round Trip Time (RTT) Calculation

- Packet Creation
  - IPv4 Header
  - IPv6 Header
  - User Datagram Protocol (UDP) Header
  - Transmission Control Protocol (TCP) Header

- Socket Calls and data packet transmission using OCaml Unix module

The remainder of this manual contains information on the lexical convention, syntax, semantics, expression usage, statements, declarations, and grammar of IPCoreL.

2 Lexical Conventions

This section introduces the fundamental elements that makeup a IPCoreL program. Lexical elements, or tokens, are utilized to construct statements, definitions, and declarations, which are required to construct complete programs.

2.1 Tokens

A token is the smallest element of the IPCoreL programming language and are essential in the compilation of an IPCoreL program. The parser of IPCoreL is designed to recognize and accept the following types of tokens:

- Identifiers
- Comments
- Whitespace
- Punctuations
- Operators

- Identifiers
- Keywords
- Constants
- Ambiguities

2.1.1 Identifiers

Identifiers are represented in IPCoreL as sequences of letters, digits, and '_' (underscore character). Identifiers can only begin with a letter and then can be followed with a sequence of letters, digits, or '_'. If a '_' is used to start an identifier, an error in compilation will result. The following represents the accepted types of identifiers:

identifier → *letter* [*letter* | *digit* | '_']

letter → ['*A*'-'*Z*' '*a*'-'*z*']

digit → ['*0*'-'*9*']

2.1.2 Comments

IPCoreL comments are delimited with the following sequence of characters /* and //, with no intervening blanks. Staying consist with the nature of comments in programming languages, IPCoreL comments are treated as blanks during lexical analysis. Comments do not occur inside string or character literals and nested comments are handled efficiently in IPCoreL.

2.1.3 Blanks/Whitespace

The following character constants will be handled by the lexical analyzer as blanks, or whitespaces:

- Space
- Newline character constant
- Carriage return
- Horizontal tabulation

Blanks separate adjacent identifiers, literals, and keywords, which requires them to be ignored by the lexical analyzer to lessen confusion for the parser of IPCoreL.

2.1.4 Punctuators

The following character constants will be handled by the lexical analyzer as punctuation input symbols:

- Left parentheses → '('
- Right parentheses → ')'
- Left brace → '{'
- Right brace → '}'
- Comma → ','

- ♦ Semicolon → ‘;’
- ♦ Single quote → ’
- ♦ Double quote→ “

2.1.5 Operators

IPCoreL includes operators that are commonly found in other proven programming languages. These operators specify an evaluation to be performed on one of the following:

- ♦ One operand (unary operators)
- ♦ Two operands (binary operators)

<u>Unary Operators</u>

The following tokens represent the unary operators of IPCoreL (w/ left-right associativity):

- ♦ ++ (postfix/prefix incrementation operator)
- ♦ −− (postfix/prefix decrementation operator)
- ♦ ! (prefix equality operator)
- ♦ () (function call member initialization)

<u>Binary Operators</u>

The following tokens represent the binary operators of IPCoreL (w/ left-right associativity):

- ♦ Integer Operators
  - ** (exponentiation)
  - + (addition)
  - − (subtraction)
  - * (multiplication)
  - / (division)
  - % (modulo)
- ♦ Floating-Point Operators
  - **. (floating-point exponentiation)
  - +. (floating-point addition)
  - −. (floating-point subtraction)
  - *. (floating-point multiplication
  - /. (floating-point division)

2.1.6 Prefix, Postfix and Infix Symbols

The following represent prefix, postfix, and infix symbols in IPCoreL arithmetic and logical operations:

Infix-symbol → [= | < | > | <= | >= | == | != | && | || | + | +. | − | −. | * | *. | / | /. | % | ** | **.]

Prefix-symbol → [- | ++ | -- | !]

Postfix-symbol → [++ | -- | ()]

2.1.7 Keywords

IPCoreL contains keywords, whose usage as regular identifiers will result in compilation errors, that define variable types, conditional statements, iterative statements, and standard functions in the programming language. Below are the reserved keywords of IPCoreL.

```
string      char        int         float if

else        elseif      while for         bool
```

The following operators are considered keywords as well :

```
+           +.          -           -.          *

*.          /           /.          %           =

++          --          **          **.         ()

<           <=          >           <=          ==

!=          &&          ||          !
```

2.1.8 Constants

IPCoreL contains constants that define integers, floating-point, character constants, and string literals.

<u>Integer Constants</u>

Integer constants of IPCoreL are defined as constants consisting of a sequence of digits. Variables, or identifiers, defined as type `int` will be capable of holding integer values only. Assigning non-integer values to variables, or identifiers, of type `int` will result in compilation errors due to incorrect syntax.

The following characters are legal integer constants:

```
0     1     2     3     4     5     6     7     8     9
```

<u>Floating-Point Constants</u>

Floating-point constants are accepted in IPCoreL to provide representation for numbers that would be too large or too small to be represented as integers. These constants are in general represented approximately to a fixed number of significant digits and scaled using an exponent. The following provides information on how exactly floating-point constants are represented in IPCoreL: *0.1, 1.0, 2.95, 3.41569*.

<u>Character Constants</u>

Character constants are delimited by single quote characters. The two single quotes enclose either one character different from ‘ and `¥`, or one of the escape sequences, `¥n` (newline constant), `¥r` (carriage return)or `¥t` (horizontal tabulation).

The following characters are legal character constants:

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	z	A	B
C	D	E	F	G	H	I	J	K	M	L	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	0	1
2	3	4	5	6	7	8	9	_				

<u>String Literals</u>

String literals are delimited by “ (double quote) characters. The two double quotes enclose a sequence of characters constants, different from “ and `¥`, or escape sequences from the character constants described above.

2.1.9 Ambiguities

Ambiguities in the lexical conventions of IPCoreL are resolved by referencing the “longest-match” rule:

When a character sequence can be decomposed into two tokens in several different ways, the decomposition retained is the one with the longest first token.

3 IPCoreL Syntax

The syntax of IPCoreL was designed to provide simplicity to novice network programmer. The set of rules, which define the combinations of symbols accepted as correct by the IPCoreL compiler, will be referenced through this document in the following format:

- Syntactic categories are identified by the usage of the *italic* type
- Literal words and characters are identified in the `typewriter` style.

Example:

```
stmt → if expr then stmt
     | if  expr then stmt else stmt
     | expr
```

4 Conversions

Many programming languages available today offer the conversion, or coercion, of the value of an operand from one data type to another. This section will explain the coercion functions available and the results to be expected from the coercion.

4.1 Integer to String Literal Coercion

IPCoreL offers the coercion of integer values to string literals using the following function:

- `int_to_string`(*arg*)

This function takes an argument of type integer and returns as a result a string literal comprising of the integer value enclosed in double quotes, e.g. `int_to_string`(*123*) → "*123*" .

4.2 String Literal to Integer Coercion

IPCoreL offers the coercion of string literals to integer values using the following function:

- `string_to_int`(*arg*)

This function takes a string literal as an argument and returns as a result an integer comprised of the contents of the string literal within the double quotes, e.g. `string_to_int`(*"123"*) → *123*

4.3 Floating-Point to String Literal Coercion

IPCorel offers the coercion of floating-point values to string literals using the following function:

- `float_to_string`(*arg*)

This function takes an argument value of type floating-point and returns as a result a string literal comprising of the floating-point value enclosed in double quotes, e.g., `float_to_string`(*1.23*) → "*1.23*" .

4.4 String Literal to Floating-Point Coercion

IPCoreL offers the coercion of string literals to floating-point values using the following function:

- `string_to_float(`*arg*`)`

This function takes a string literal as an argument and returns as a result a floating-point value comprised of the contents of the string literal within the double quotes, e.g.
`string_to_float(` “*1.23*” `)` → *1.23*

5 Expressions

IPCoreL expressions consist of a combination of explicit values, constants, variables, operators, and functions that operate under the rules of precedence and of association. Values can be of type int, float, string, or char. IPCoreL affords for a multitude of expressions including postfix, unary, incrementation, decrementation and operational expressions.

5.1 Primary Expressions

Primary expressions within IPCoreL consist of *identifiers*, variables defined as *constants*, *strings*, *expression*, and (*expression*).

5.2 Postfix Expressions

Postfix expressions in IPCoreL groups operators from left to right. The following outlines the utilization of postfix expressions within IPCoreL

expression:
    expression
    expression[*expression*]
    expression(*expression-opt*)
    expression =variable-identifier
    expression =integer-lvalue | *float-lvalue* | *character-lvalue* | *string-lvalue*
    expression++
    expression−−

expression-list:
    expression
    expression-opt, *expression*

5.3 Unary Operators

IPCoreL groups operators that unary in nature from right-to-left. Aside from parenthesis pairs, incrementation and decrementation operators, the only other unary operator is '-'. The following defines the handling of unary expressions within IPCoreL.

unary-expression:
 expression
 ++*unary-expression*
 −−*unary-expression*
 unary-operator expression

unary-operator: − !

5.4 Array References

A postfix expression followed by an expression in square brackets, *postfix-expression*[*expression-opt*], is a postfix expression denoting a subscripted array reference.

5.5 Function Calls

A function call is an expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. The function call may also be empty. The type of a function call expression is the return type of the function. This type can either be of primitive type or of type void.

Arguments of a function call are referred to as *function arguments*. These *function arguments* are expressions used within the parentheses of a function call. *Function parameters* make up the *function arguments* within a function call.

IPCoreL handles function calls in the following manner:

function-identifier(*argument-list*)
function-identifier()

5.6 Postfix Incrementations

IPCoreL expresses postfix incrementation expressions in the following manner using the postfix operator, ++.

unary-expression++

The value of the postfix-expression is the value of the operand. After the value is noted, the operand is incremented by 1. The operand must be of integer value in order for the *postfix-expression* to compile properly.

5.7 Prefix Incrementations

IPCoreL expresses prefix incrementation expressions in the following manner using the prefix operators, ++. Prefix operators are handled in the same manner as postfix operators.

++unary-expression

The value of the prefix-expression is the value of the operand. After the value is noted, the operand is incremented by 1. The operand must be of integer value in order for the *prefix-statement* to compile properly.

5.8 Postfix Decrementations

IPCoreL expresses postfix decrementation expressions in the following manner using the postfix operator, −−.

unary-expression−−

The value of the postfix-expression is the value of the operand. After the value is noted, the operand is decremented by 1. The operand must be of integer value in order for the *postfix-expression* to compile properly.

5.9 Prefix Decrementations

IPCoreL expresses prefix incrementation expressions in the following manner using the prefix operators, −−. Prefix operators are handled in the same manner as postfix operators.

−−unary-expression

The value of the prefix-expression is the value of the operand. After the value is noted, the operand is decremented by 1. The operand must be of integer value in order for the *prefix-statement* to compile properly.

5.10 Multiplicative Operators

IPCoreL multiplicative operators, ∗ and /, are grouped from left-to-right. The following defines how IPCoreL handles multiplicative operators:

multiplicative-expressions:

multiplicative-expression ∗ unary-expression
multiplicative-expression / unary-expression
multiplicative-expression ∗∗ unary-expression
multiplicative-expression ∗. unary-expression

multiplicative-expression /. *unary-expression*
multiplicative-expression **. *unary-expression*

The multiplicative operators are of arithmetic type and can handle both integer and floating-point types. The result of

5.11 Additive Operators

The additive operators + and - possess left associativity and have the same level of precedence. If and only if expressions and operands are of arithmetic type can they be used with additive operators. The arithmetic types consist of integer and floating-point. The following defines how IPCoreL handles additive operators:

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression +. *multiplicative-expression*
additive-expression − *multiplicative-expression*
additive-expression −. *multiplicative-expression*

5.12 Relational Operators

Relational operators in IPCoreL group left-to-right and evaluate to either 0 or 1. The following details the manner in which relational operators are handled in IPCoreL:

relational-expression:
relational-expression < *unary-expression*
relational-expression > *unary-expression*
relational-expression <= *unary-expression*
relational-expression >= *unary-expression*

The operators <, >, <=, and => all yield 0 if the specified relation is false and 1 if the relation holds true. The type of the result is `int`.

5.13 Equality Operators

IPCoreL handles == and the != operators in the same manner as relational operators, except for their lower precedence. Equality operators follow the same rules as relational operators.

equality-expression:
relational-expression
equality-expression != *relational-expression*
equality-expression == *relational-expression*

5.14 Logical AND/OR Operators

The && and || operators group left-to-right in IPCoreL. The && operator returns 1 if both its operands compare unequal to zero, and 0 otherwise. The same rule applies to the || operator.

logical-expression:
    logical-expression && *expression*
    logical-expression || *expression*

5.15 Assignment Operators

IPCoreL uses several assignment operators to assign values to identifiers. All assignment operators group left-to-right

expression:
    expression =unary-expression
    expression -> *function-identifier* (*parameter-list*)

The = operator assign values to identifiers. The -> operator is used in IPCoreL to create IP headers and IP packets.

5.16 Sequences

A pair of expression separated by a comma is evaluated left-to-right in IPCoreL, with the value of the left expression being ignored. The type and value of the result are the type and value of the right operand. All side effects from the evaluation of the left operand are completed before beginning the evaluation of the right operand.

expression-list:
    expression
    epression-list, expression

6 Declarations

A declaration in IPCoreL specifies the interpretation given to a particular identifier. IPCoreL contains ability to declare functions and variables, and have the following form:

function-decl:
    function-type-specifier function-identifier (*formal-opt*) {*variable-decl-list, statement-list*} ;

variable-decl-list
    empty-variable-list

  variable-decl-list, *variable-decl*

variable-decl:
  variable-type-specifier variable-identifier;

Declarators in the *formal-opt* contains the optional identifiers that are being declared in the declarations. Declaration specifiers consist of the following:

formal-opt:
  empty-formal-list
  formal-list

formal-list:
  variable-identifier
  formal-list, *variable-identifier*

IPCoreL utilizes few **-type-specifiers,* which consist of the following:

function-type-specifier:
```
    void
    int
    float
```

variable-type-specifier:
```
    int
    float
    char
    string
    bool
```

6.1 Declarators

Declarators are components of a declaration that specify names of objects or functions within an IPCoreL program. These components also identify whether a named object is a variable or array. When applied to functions, declarators work with the type specifier to explicitly specify the return type of a function.

IPCoreL arrays may be declared using the following syntax example:

```
float x[10];           /* empty array of type float with 10 elements */

int x[3] = {1, 2, 3};  /* array of type int with 3 elements initialized to 1, 2, and 3
                       respectively*/

string str = “Hello World”;
string str = ( “Hello World” );
```

6.2 Function Declarators

IPCoreL affords programmers the ability to define functions containing a function declaration and the body of a function.

function-decl:
    function-type-specifier function-identifier （*formal-opt*） {*variable-decl-list*, *statement-list*}

Function declarations in IPCoreL must follow the following rule in order for a program to properly compile:

- ♦ Only one type specifier is required for a function declaration. The type specifier determines the type of value return after execution of the function is complete.
- ♦ A function declaratory must consist of a function name followed by a parenthesized list of optional parameters
- ♦ If a function is declared of type `int` or `float` then it must be terminated with a `return` *expression*；

Examples of function declarations:

```
int f(int a, int b) {
     return a + b;
}
```

or

```
void f(a, b) {
     a = a + b;
}
```

or

```
int x = 1;

void f() {
     print x;
};
```

The syntax of parameters is the following:

parameter-list:
    parameter
    parameter-list, *parameter*

parameter:
    expression

6.3 Initialization

When initially defined, declarators have the option of specifying the initial value of the objects being declared in the program. The grammar of IPCoreL initialization is the following:

expression:

    expression = *expression*[*expression-opt*]
    expression = (*expression*)
    expression = *expression*
    expression −> *function-identifier*(*parameter-list*)

7 Statements

Statements are executed in sequence in IPCoreL, and are done so for their effect. The grammar of IPCoreL statements is:

statement:

    expression-statement
    block-statement
    conditional-statement
    iterative-statement

7.1 Expression Statement

Majority of IPCoreL programs consist of expression statements that are either assignments or function calls. All side effects from the expression are completed before the next statement is executed in the program. Expression statements take the following form in IPCoreL:

expression-statement:

    empty-expression
    expression

7.2 Block Statements

Block statements, or compound statements, are formed by grouping several statements together in a "block" bounded by {}. The grammar of block statements in IPCoreL is the following:

block-statement:

    {*statement*} ;
    {*statement-list*, *statement*} ;

If identifiers in the *expression-list* are in scope outside the block, the outer declarations are suspended within the block statement. The following rule applies to all block statements:

An identifier may be declared only once in the same block. Declarations consisting of the same identifier within the same block will result in compilation errors.

7.3 Conditional Statement

IPCoreL conditional statements resemble conditional statements used in other programming languages, e.g., C, C++, Java, and Python. The following describes the grammar of IPCoreL conditional statements:

conditional-statement:

  `if`(*expression*) {*statement*} ;

  `if`(*expression*) {*statement*} `else`{*statement*} ;

  `if`(*expression*) {*statement*} `elseif`(*expression*) {*statement*} `else`{*statement*} ;

If using `if` statements in programs, the expression must be of arithmetic type. The expression is evaluated and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is equal to 0.

The last conditional statement, containing `elseif` keyword, is a combination of `if` and `else` and extends the `if` statement to execute a different statement in case the original `if` expression evaluates to *false*. The `elseif` statement will execute an alternative expression only if the `elseif` conditional expression evaluates to *true*.

7.4 Iterative Statements

The grammar of IPCoreL iterative, looping, statements is the following:

iterative-statement:

  `while`(*expression*) {`statement`} ;

  `for`(*expression-opt*; *expression-opt*; *expression-opt*) {*statement*} ;

The `while` loop consists of a test that occurs before each execution of the statement within the iterative statement. In the `for` statement, the first expression is evaluated once and specifies the initialization for the loop. The second expression must be of arithmetic type and it is evaluated after before each iteration. If the second expression becomes equal to 0, the `for` loop is terminated. The third expression of the `for` loop is evaluated after each iteration, specifying a re-initialization for the loop.

If users of IPCoreL wish not to use `for` loops, the following is its equivalent using a `while` loop.

  `for`(*expression-opt*; *expression-opt*; *expression-opt*) {*statement*} ;

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

8 IPCoreL Functions

IPCoreL affords programmers certain functions that calculate network behavior. The following grammar describes the functions that are included in IPCoreL:

Network calculation functions:

```
        throughput(parameter-list);
        jitter(parameter-list);
        latency(parameter-list);
        rtt(parameter-list);
```

Header creation functions:

```
        header(paramater-list);
```

Packet creation:

```
        packet(parameter-list);
```

Socket functions:

```
        opensock(parameter-list);
        sendmsg(parameter-list);
        recvmsg(parameter-list);
```

8.1 Function Definitions

The following defines native functions provided in the IPCoreL Programming Language.

`throughput:`	performs throughput calculation based on user-defined parameters listed in the function call. ♦ Throughput is the average rate of successful message delivery over a communication channel
`jitter:`	performs jitter calculation based on user-defined parameters listed in the function call. ♦ Jitter is the time variation of a periodic signal in telecommunications. Jitter may be observed in characteristics such as the frequency of successive pulses, the signal amplitude, or phase of periodic signals.

`latency:`	performs latency calculation based on user-defined parameters listed in the function call. ♦ Latency is the measure of time delay experienced in a communication network.
`rtt:`	performs round-trip time calculation based on user-defined parameters listed in the function call. ♦ RTT (Round-trip Time) is the measure of time taken for a packet to reach each its destination from its source.
`header:`	creates an IPv4, IPv6, UDP, or TCP header based on user-defined parameters listed in the function call. ♦ Headers are supplemental data placed at the beginning of an IP packet containing data to be stored and/or transmitted through a communication network.
`packet:`	creates a custom IP packet based on user-defined parameters, containing headers, listed in the function call. ♦ Packets are formatted units of data that are traversed though communication networks. Packets created in IPCoreL support IPv4 and IPv6.
`opensock:`	makes a socket call when functional is made; socket type is based on user-defined parameters listed in the function call.
`sendmsg:`	packages the IP packet created with `packet` function and transmits the packet outside the Network Interface Card (NIC) of user's PC.
`recvmsg:`	receives incoming IP packets unpacks the datagram for information processing based on user parameters listed in function call.

9 Scope

The scope, in a program, is the enclosed context where the values and expressions are associated. The type of a scope determines what kind of entities it can contain and how it affects them. Scopes within IPCoreL may consist of the following:

- declarations or definitions of identifiers
- statements or expressions which define an executable algorithm
- nests of declarations or functions

Variables are associated within scopes. Different scoping types affect how local variables in a block of statements are bound.

<u>Lexical Scoping</u>

In lexical scoping, or static scoping, a name always refers to its local lexical environment. Lexical scoping occurs when the scope of an identifier is fixed at compile time to some region in the source code containing the identifier's declaration, meaning that an identifier is only accessible within that region of code it is currently residing in.

<u>Local Scope</u>

Variables or methods that have local scope are accessible only in the current block of statements in which the variable was defined. These variables are therefore limited to the most current block of code, and outer blocks of code surrounding it may not have access to the variable.

<u>Global Scope</u>

With global scope, all variables defined at the very beginning of a program are available to the entire program. The same rule applies to functions declared in a program. All variables declared at the beginning of the function are available to the remaining code of said function.

<u>Duplicate Variable Declaration</u>

In IPCoreL, it is possible to “override” a local variable that is defined just before the current block being accessed by the program. This is accomplished by declaring another variable of the same name and data type inside the current block. The new variable will naturally have more scope than the first declaration outside the current block. This is due to the outer variable being temporarily overridden and the new variable's value hiding whatever the outer variable's value was previously.

10 Grammar

The following illustrates the grammar of IPCoreL that has been dissected throughout this reference manual. The `typewriter` style words and symbols are terminals of IPCoreL, these represent the keywords and functions.

main-program:

 main-program variable-decl
 main-program function-decl

function-type-specifier:

 `void`
 `int`
 `float`

variable-type-specifier:

 `int`

    `float`
    `char`
    `string`
    `bool`

function-decl:
    function-type-specifier function-identifier (*formal-opt*) {*variable-decl-list*, *statement-list*} ;

function-identifier:
    variable-identifier
    `throughput`
    `jitter`
    `latency`
    `rtt`
    `header`
    `packet`
    `opensock`
    `sendmsg`
    `recvmsg`

formal-opt:
    empty-formal-list
    formal-list

formal-list:
    variable-identifier
    formal-list, *variable-identifier*

variable-decl-list
    empty-variable-list
    variable-decl-list, *variable-decl*

variable-decl:
    variable-type-specifier variable-identifier;

statement-list:
    statement
    statement-list statement

statement:
    `return` *expression*;
    expression;
    (*expression-list*) ;
    {*expression-list*} ;
    `if`(*expression*) {*statement-list*} ;
    `if`(*expression*) {*statement-list*} `else`{*statement-list*} ;

if（*expression*）｛*statement-list*｝ elseif（*expression*）｛*statement-lsit*｝ else｛*statement-list*｝；
while(*expression*）｛*statement-list*｝；
for（*expression-opt*； *expression-opt*； *expression-opt*）｛*statement-list*｝；
function-identifier（*parameter-list*）；

parameter-list:
parameter
parameter-list， *parameter*

parameter:
expression

expression:
integer-lvalue
float-lvalue
string-lvalue
character-lvalue
variable-identifier
additive-expression ＋*multiplicative-expression*
additive-expression － *multiplicative-expression*
multiplicative-expression ＊*expression*
multiplicative-expression / *expression*
multiplicative-expression ＊＊*expression*
floating-additive-expression ＋. *floating-multiplicative-expression*
floating-additive-expression －. *floating-multiplicative-expression*
floating-multiplicative-expression ＊. *floating-expression*
floating-multiplicative-expression ／. *floating-expression*
floating-multiplicative-expression ＊＊. *floating-expression*
equality-expression ==*relational-expression*
equality-expression !=*relational-expression*
relational-expression ＞*expression*
relational-expression ＜*expression*
relational-expression ＞=*expression*
relational-expression ＜=*expression*
unary-expression＋＋
unary-expression－－
＋＋*unary-expression*
－－*unary-expression*
！*unary-expression*
expression =*unary-expression*
expression=*expression*［*expression-opt*］
expression = （*expression*）
expression -＞ *function-identifier*（*parameter-list*）

additive-expression:
  expression

multiplicative-expression:
  expression

floating-additive-expression:
  expression

floating-multiplicative-expression:
  expression

equality-expression:
  expression

relational-expression:
  expression

unary-expression:
  expression
sequence-expression:
  empty-expression
  expression-list

expression-opt:
  empty-expression
  expression-list

expression-list:
  expression
  epression-list, *expression*