Propeller C Compiler User's Manual

By Mike Christle

Table of Contents

Introduction

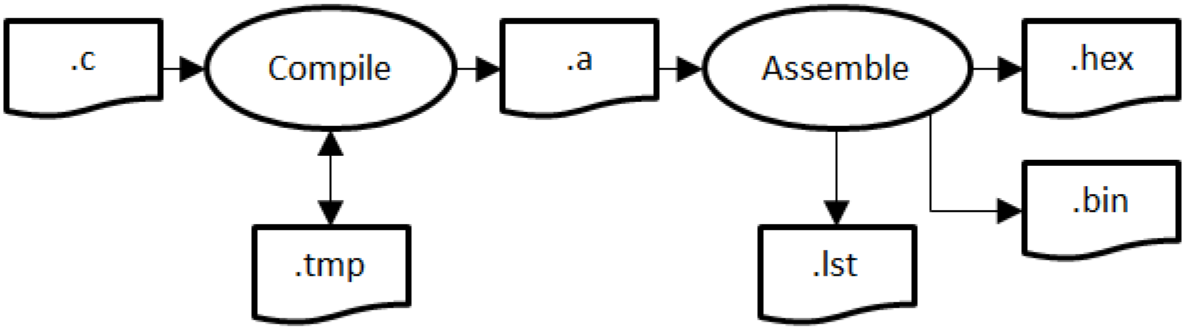
Does the world need another compiler? Probably not. However, I took a course in compiler design a couple of years ago and found the subject absolutely fascinating. Since then I have had a lot of fun working on this little project. If anyone else finds it useful, well that's good to.

PropC is a command line compiler that outputs Parallax Propeller assembly code. The source language is based on C. Rather than trying to be C compliant; my priority was to allow efficient access to all the unique features of the Propeller. That's why I added many functions and expressions that are simple wrappers around Propeller assembly instructions.

I recently started using the Parallax SimpleIDE. So I added an assembler that outputs a text file that can be included in the main C program.

TERMS OF USE: MIT License

Build Process



The .c file is your PropC source. The .tmp file is created and used by the compiler; it contains the source for all included source files. The .a file contains the Propeller assembly source. The .a file can be assembled with the Propeller Tool, or passed to the PropC assembler. The .hex file contains the object code as C syntax hexadecimal constants. Use the #header and #footer directives to make this file an initialized integer array. The .lst file is an optional list file.

Optionally, a binary .bin file can be output instead of the .hex file. The binary file can be read into a Spin file using the Spin FILE command.

The first function encountered will become the main function regardless of the name. This function should loop forever.

PropC is not case sensitive. For example NAME = Name = name. This applies to all symbols and keywords.

Command Line Options

-l	Output assembly list file.
-b	Output binary file instead of hex file.
-sa	Skip assembler stage.
-sc	Skip compiler stage, pass file directly to assembler.
-so	Skip optimizer, useful if object code is in doubt.

Directives

#include “filename”	Include the named file.
#define label value	Define constants
#header “text”	Text is written to the top of the .hex file.
#footer “text”	Text is written to the bottom of the .hex file.

Error Reporting

PropC will stop on the first error and output a (hopefully useful) error message and the source line number. The line number refers to the .tmp file which is a composite of all included source files. If no files are included then this is the same as the .c file.

Language Syntax

All of the examples are snippets from .a files of test cases. That's why the C code appears as assembly comments.

Comments

PropC supports line comments, //. It does not support block comments, /* */.

Data Types

Int UInt 32 bit signed and unsigned integers.

Real 32 bit fixed point number, with 16 bits on either side of the decimal point.

Constants

Constants are specified similar to C, they start with a digit and the default is decimal. The 0x, 0o, and 0b prefixes specify hexadecimal, octal, and binary formats respectively. If the constant contains a period it is interpreted as a real data type, otherwise it is interpreted as an integer. Constants may include underscores which are ignored.

Predefined Constants

The following constants are always defined. Type AINT means int or uint.

Name	Value	Type	Description
true	1	AINT	TRUE
false	0	AINT	FALSE
odd	1	AINT	Used with parity function.
even	0	AINT	Used with parity function.
dira		UINT	Propeller register.
dirb		UINT	Propeller register.
ina		UINT	Propeller register.
inb		UINT	Propeller register.
outa		UINT	Propeller register.
outb		UINT	Propeller register.
par		UINT	Propeller register.
cnt		UINT	Propeller register.
ctra		UINT	Propeller register.
ctrb		UINT	Propeller register.
frqa		UINT	Propeller register.
frqb		UINT	Propeller register.
phsa		UINT	Propeller register.
phsb		UINT	Propeller register.
vcfg		UINT	Propeller register.
vscl		UINT	Propeller register.

Assembly Language Variable Names

Function names and global variables have the same name in the assembly code output. Function variable names are constructed by appending the function name, an underscore, and the variable name. For example a variable x declared in the main function will get the name main_x. Temporary variables are constructed by appending the function name, two underscores, and an integer, for example main__0.

Variable Declaration

Variables are declared as in C. They can be initialized with a constant value of the same data type.

```
'     int  i1, i2 = -4;
                        MOV     main_i2, CONST_M_4
'     uint u1, u2 = 8;
                        MOV     main_u2, #8
'     real r1, r2 = 4.4;
                        MOV     main_r2, CONST_4_4

CONST_M_4               LONG    -4
CONST_4_4               LONG    288358
main_i1                 RES     1
main_i2                 RES     1
main_u1                 RES     1
main_u2                 RES     1
main_r1                 RES     1
main_r2                 RES     1
```

Expressions Operators

Expressions can be constructed using these operators. They are listed in order of precedence.

Operator	Assignment	Description
()		Parentheses.
+ - ~ !		Unary plus, minus, bitwise not, logical not.
* / %	*= /= %/	Multiply, divide, modulo.
+ -	+= -=	Addition, subtraction.
<< >> <- ->	<<= >>= <-= ->=	Shift left, shift right, rotate left, rotate right.
& &~	&= &~=	Bitwise and, bitwise and not.
^	^=	Bitwise exclusive or.
|	|=	Bitwise or.
<# #>		Max, min.

Rotate

Rotates work just like shifts. Rotates have the same level of precedence as shifts.

```
'     a = b <- c;
                        MOV     main_a, main_b
                        ROL     main_a, main_c
'     a = b -> 5;
                        MOV     main_a, main_b
                        ROR     main_a, #5
'     a <-= 4;
                        ROL     main_a, #4
'     a ->= b;
                        ROR     main_a, main_b
```

AndNot

```
'     uint pin_mask;
'
'     // Set a pin HI
'     outa |= pin_mask;
                        OR      outa, main_pin_mask
'     // Set a pin LO
'     outa &~= pin_mask;
                        ANDN    outa, main_pin_mask
```

Limits

Limits get the lowest precedence level, so they work best when added to the end of an expression. Limits can be applied to expressions of any type.

```
'     int a, b, c;
'     a = b + c #> 10 <# 100;
                        MOV     main_a, main_b
                        ADDS    main_a, main_c
                        MINS    main_a, #10
                        MAXS    main_a, #100
```

Logical Operators

Logical expressions can be constructed using these operators. They are listed in order of precedence.

Operator	Description
== != < > <= >= LOCKSET LOCKCLR PARITY	Compare two expressions, or compare one expression to 0. Set and test a lock. Clear and test a lock. Test parity of an expression.
&&	Logical and.
||	Logical or.

Locks

Function	Input Type	Output Type
locknew	Uint	
lockset	Uint	Uint
lockclr	Uint	Uint
lockret	Uint	

```
'     uint lock, data_ready, data_ptr;
'
'     locknew(lock);
                          LOCKNEW main_lock
'
'     // Block until lock is free
'     while (lockset(lock) == true);
:L1
                          LOCKSET main_lock WC
   IF_C                   JMP     #:L1
'     // ...
'     lockclr(lock);
                          LOCKCLR main_lock WC
'
'     // Proceed if data is ready and lock is free
'     if (data_ready && lockset(lock) == false)
                          CMP     main_data_ready, #0  WZ
   IF_Z                   JMP     #:L5
                          LOCKSET main_lock WC
   IF_C                   JMP     #:L5
'     {
'         // ...
'         lockclr(lock);
                          LOCKCLR main_lock WC
:L5
'     }
'
'     lockret(lock);
                          LOCKRET main_lock
```

If Statements

The syntax of the if statements is the same as C.

```
'   int a, b, c;
'
'   if (a == 3 || b == 4 && c == 5) nop;
                        CMPS    main_a, #3  WZ
   IF_Z                 JMP     #:L142
                        CMPS    main_b, #4  WZ
   IF_NZ                JMP     #:L145
                        CMPS    main_c, #5  WZ
   IF_NZ                JMP     #:L145
:L142
                        NOP
:L145
'
'   if (a == b)
                        CMPS    main_a, main_b  WZ
   IF_NZ                JMP     #:L154
'   {
'     a = 9;
                        MOV     main_a, #9
                        JMP     #:L155
'   }
'   else
'   {
:L154
'     a = 6;
                        MOV     main_a, #6
:L155
'   }
```

For Loops

The syntax of for loops is the same as C, except that the test and increment clauses are optional. The abbreviated form takes advantage of the DJNZ instruction. The break and continue statements work as expected.

```
'   int i, j, k;
'   for (i = j + 10)
                        MOV     main_i, main_j
                        ADDS    main_i, #10
'     k += 4;
:L1
                        ADDS    main_k, #4
                        DJNZ    main_i, #:L1
'
'   for (i = 0; i < 10; i += 1)
                        MOV     main_i, #0
:L3
                        CMPS    main_i, #10  WZ, WC
   IF_NC                JMP     #:L6
'   {
'       if (i == 3) continue;
                        CMPS    main_i, #3  WZ
   IF_Z                 JMP     #:L3
'       if (i == 7) break;
                        CMPS    main_i, #7  WZ
   IF_Z                 JMP     #:L6
'   }
                        ADDS    main_i, #1
                        JMP     #:L3
:L6
```

While Loops

The syntax for the while and do while loops is the same as in C.

```
'     int i, j;
'     while (true) nop;
:L1
                        NOP
                        JMP     #:L1
'
'     while (i < 40)
:L4
                        CMPS    main_i, #40  WZ, WC
   IF_NC                JMP     #:L6
'     {
'         if (i == 10) continue;
                        CMPS    main_i, #10  WZ
   IF_Z                 JMP     #:L4
'         if (i == 20) break;
                        CMPS    main_i, #20  WZ
   IF_NZ                JMP     #:L4
:L6
'     }
'
'     do
:L13
'     {
'         if (i == 10) continue;
                        CMPS    main_i, #10  WZ
   IF_Z                 JMP     #:L13
'         if (i == 20) break;
                        CMPS    main_i, #20  WZ
   IF_Z                 JMP     #:L14
'     }
'     while (i < 40);
                        CMPS    main_i, #40  WZ, WC
   IF_C                 JMP     #:L13
:L14
```

Switch

The syntax for the switch statement is the same as in C. The switch variable must be an integer type. The case values must be integer constants.

```
'     int i, j;
'     uint m;
'     switch (i)
'     {
'         case 1: j += 3; break;
                        CMPS    main_i, #1  WZ
   IF_NZ                JMP     #:L2
                        ADDS    main_j, #3
                        JMP     #:L1
:L2
'         case 2: j -= 5; break;
                        CMPS    main_i, #2  WZ
   IF_NZ                JMP     #:L3
                        SUBS    main_j, #5
                        JMP     #:L1
:L3
'         default: j = 0; break;
                        MOV     main_j, #0
'     }
:L1
'
'     switch (m)
'     {
'         case 1: j += 3; break;
                        CMP     main_m, #1  WZ
   IF_NZ                JMP     #:L5
                        ADDS    main_j, #3
                        JMP     #:L4
:L5
'         case 2: j -= 5; break;
                        CMP     main_m, #2  WZ
   IF_NZ                JMP     #:L6
                        SUBS    main_j, #5
                        JMP     #:L4
:L6
'         default: j = 0; break;
                        MOV     main_j, #0
'     }
:L4
```

Functions

The syntax for function declarations and calls is the same as in C.

```
' int g1;
' void main()
' {
main
'     int i;
'     func1();
                        CALL    #func1
'     i = func2(1, 5);
                        MOV     func2_a, #1
                        MOV     func2_b, #5
                        CALL    #func2
                        MOV     main_i, func2_
' }
main_RET                RET
'----------------------------------------------------------
' void func1()
' {
func1
'     g1 += 3;
                        ADDS    g1, #3
' }
func1_RET               RET
'----------------------------------------------------------
' int func2(int a, int b)
' {
func2
'     return a + b;
                        MOV     func2_, func2_a
                        ADDS    func2_, func2_b
' }
func2_RET               RET
'----------------------------------------------------------
```

Built In Functions

The following functions, as well as multiply, divide and modulus, are implemented by subroutines that are included as needed. Parameters are passed to these functions via the variables math_p1 and math_p2. Results are passed back via the variables math_r1 and math_r2. The multiply, divide and modulus operations take 32 bit parameters and return 32 bit results. For the trig functions a full circle contains 512.0 degrees.

Function	Input Type	Output Type
sin	Real	Real
cos	Real	Real
tan	Real	Real
asin	Real	Real
acos	Real	Real
sqrt	Real	Real
isqrt	Int, Uint	Int, Uint
log	Any	Int
exp	Int	Int

```
'     int i, j;
'     real m, n;
'     i *= j;
                        MOV     math_p1, main_i
                        MOV     math_p2, main_j
                        CALL    #IMultiply
                        MOV     main_i, math_r1
'     i /= j;
                        MOV     math_p1, main_i
                        MOV     math_p2, main_j
                        CALL    #SDivide
                        MOV     main_i, math_r1
'     i %= j;
                        MOV     math_p1, main_i
                        MOV     math_p2, main_j
                        CALL    #SDivide
                        MOV     main_i, math_r2
'     m = sin(n);
                        MOV     math_p1, main_n
                        CALL    #Sin
                        MOV     main_m, math_r1
```

Math Functions

Function	Input Type	Output Type
itor	Int	Real
rtoi	Real	Int
floor	Real	Real
ceil	Real	Real
trunc	Real	Real
round	Real	Real
fract	Real	Real
abs	Any	Same as input
absn	Any	Same as input

```
'      real m, n;
'      i = rtoi(m);
                        MOV     main_i, main_m
                        SAR     main_i, #16
'      m = itor(i);
                        MOV     main_m, main_i
                        SHL     main_m, #16
'      m = floor(n);
                        MOV     main_m, main_n
                        ANDN    main_m, math_real_mask
'      m = ceil(n);
                        MOV     main_m, main_n
                        ADD     main_m, math_real_mask
                        ANDN    main_m, math_real_mask
'      m = trunc(n);
                        MOV     main_m, main_n
                        ABS     main_m, main_m
                        ANDN    main_m, math_real_mask
                        NEGC    main_m, main_m
'      m = round(n);
                        MOV     main_m, main_n
                        ADD     main_m, math_half
                        ANDN    main_m, math_real_mask
'      m = fract(n);
                        MOV     main_m, main_n
                        ABS     main_m, main_m
                        AND     main_m, math_real_mask
                        NEGC    main_m, main_m
'      m = abs(n);
                        ABS     main_m, main_n
'      m = absn(n);
                        ABSNEG  main_m, main_n
math_real_mask          LONG    65535
```

Waits

These functions provide access to the Propellers four wait instructions.

```
'   uint a, one_msec, mask;
'
'   waitcnt(cnt + one_msec);
                    MOV     main__0, cnt
                    ADD     main__0, main_one_msec
                    WAITCNT main__0, #0
'   waitcnt(a, one_msec);
                    WAITCNT main_a, main_one_msec
'
'   waitpeq(a, 4);
                    WAITPEQ main_a, #4
'   waitpeq(a, mask);
                    WAITPEQ main_a, main_mask
'
'   waitpne(a, 8);
                    WAITPNE main_a, #8
'   waitpne(a, mask);
                    WAITPNE main_a, main_mask
'
'   waitvid(a, 9);
                    WAITVID main_a, #9
'   waitvid(a, mask);
                    WAITVID main_a, main_mask
```

Global Data Access

Global data is modelled as four arrays: GBYTE, GWORD and GLONG. The index for these arrays must be int or uint type. Any type can be read or written.

```
'     uint uval, index;
'     int ival;
'     real rval;
'
'     uval = GBYTE[index];
                        RDBYTE  main_uval, main_index
'     ival = GWORD[index];
                        RDWORD  main_ival, main_index
'     rval = GLONG[index];
                        RDLONG  main_rval, main_index
'
'     GBYTE[index] = uval;
                        WRBYTE  main_uval, main_index
'     GWORD[index] = ival;
                        WRWORD  main_ival, main_index
'     GLONG[index] = rval;
                        WRLONG  main_rval, main_index
```

Field Access

The Inst, Src and Dest fields of a register can be accessed using the .I, .S and .D qualifiers on an assignment.

```
'     uint apin, bpin;
'     CTRA.S = apin;
                        MOVS    ctra, main_apin
'     CTRA.D = bpin;
                        MOVD    ctra, main_bpin
'     CTRA.I = 0b0_00100_000;
                        MOVI    ctra, #32
```

Parity

This function will return the parity of a value. The constants ODD and EVEN are predefined for testing parity.

```
'   uint i, j, mask;
'   i = parity(mask, j);
                        TEST    main_mask, main_j  WC
                        SUBX    main_i, main_i
'
'   if (parity(mask, ina) == ODD) nop;
                        TEST    main_mask, ina  WC
   IF_NC                JMP     #:L17
                        NOP
:L17
```

Miscellaneous Functions

These were included for completeness sake. Nop is useful for adding a 4 clock cycle delay.

```
'     nop;
                        NOP
'     cmpsub(ival, 100);
                        CMPSUB  main_ival, #100
'     rev(ival, 16);
                        REV     main_ival, #16
```

BlinkLED Example Program

This is the main program that compiles in SimpleIDE. It includes the cog code from the file BlinkLED_cog.hex.

```
// File BlinkLED.c
#include "simpletools.h"
#include "BlinkLED_cog.hex"

typedef struct
{
  volatile int pin_mask;
  volatile int half_period;
  int cog;
}
blink_t;

int main()
{
    int cog;
    blink_t *device;

    device = (void *) malloc(sizeof(blink_t));
    device->pin_mask = 1;
    device->half_period = CLKFREQ >> 3;

    cog = 1 + cognew((void*)cog_code, (void *)device);

    while(1);
}
```

This is the main program in Spin that runs in Propeller Tool. It includes the cog code from the file BlinkLED_cog.bin. The cog code must be compiled with the –b option.

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

VAR
  long  Cog, PinMask, HalfPeriod

PUB MainRoutine
  PinMask := 1
  HalfPeriod := 20_000_000

  Cog := cognew(@CogCode, @PinMask)
  repeat

DAT
CogCode       file  "BlinkLED_cog.bin"
```

This is the code to be compiled by PropC which runs in the cog.

```
// File BlinkLED_cog.c

#header "int cog_code[] = {"
#footer "};"

uint half_period;
uint pin_mask;
uint wait_cntr;

void main()
{
    uint ptr;

    ptr = par;
    pin_mask = GLONG[ptr];
    ptr += 4;
    half_period = GLONG[ptr];

    dira = pin_mask;
    wait_cntr = cnt + half_period;

    while (true)
    {
        outa ^= pin_mask;
        waitcnt(wait_cntr, half_period);
    }
}
```

This is the file BlinkLED_cog.hex, which is included into the SimpleIDE file.

```
int cog_code[] = {
0xA0BC1DF0,0x08BC180E,0x80FC1C04,0x08BC160E,
0xA0BFEC0C,0xA0BC1BF1,0x80BC1A0B,0x6CBFE80C,
0xF8BC1A0B,0x5C7C0007,0x5C7C0000};
```

BNF Grammar

<abc>	Rule name
::=	Is defined by
|	Or
[ ]	Optional
{}	Repeat one or more times
{}*	Repeat zero or more times
( )	Group
‘,’	Literal symbol
ABC	Keyword
;	Rule terminator

```
<program> ::= { <data-decl-stmt> | <func-decl> }* ;

<func-decl> ::= <return-data-type> <identifier>
                '(' [ <func-parms> ] ')' <func-body> ;

<func-parms> ::= <data-type> <identifier>
                 { ',' <data-type> <identifier> }* ;

<return-data-type> ::= <data-type> | VOID ;

<func-body> ::= '{' { <data-decl-stmt> }* { <statement> } '}' ;

<data-decl-stmt> ::= <data-type> <data-decl> { ',' <data-decl> }* ';' ;

<data-decl> := <identifier> [ '=' [ '+' | '-' ] <constant> ] ;

<data-type> ::= INT | UINT | REAL ;

<statement> ::= <assignment-stmt>
              | <function-stmt>
              | <dot-assign-stmt>
              | <if-stmt>
              | <while-stmt>
              | <do-while-stmt>
              | <for-stmt>
              | <switch-stmt>
              | <break-stmt>
              | <continue-stmt>
              | <return-stmt>
              | <compound-stmt>
              | <global-array-stmt>
              | <wait-count-stmt>
              | <wait-stmt>
              | <lock-stmt>
              | <built-in-op-stmt>
```

```
               | <nop-stmt>
               | ';' ;

<assignment-stmt> ::= <assignment> ';' ;

<assignment> ::= <identifier> <assignment-op> <expr>
                | <identifier> '=' [TRUE | FALSE] ;

<assignment-op> ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='
                   | '|=' | '&=' | '^=' | '&~='
                   | '<<=' | '>>=' | '<-=' | '->=' ;

<dot-assign-stmt> ::= <identifier> ( '.I' | '.D' | '.S' )
                      '=' <expr> ';' ;

<global-array-stmt> ::= <global-array> '=' <expr> ';' ;

<global-array> ::= ( GBYTE | GWORD | GLONG ) '[' <expr> ']' ;

<if-stmt> ::= IF '(' <logical-or-expr> ')' <statement>
              [ ELSE <statement> ] ;

<while-stmt> ::= WHILE '(' <logical-or-expr> ')' <statement> ;

<do-while-stmt> ::= DO <statement>
                    WHILE '(' <logical-or-expr> ')' ';' ;

<for-stmt> ::= FOR '(' <for-init>
               [ ';' <logical-or-expr> ';' <for-incr> ] ')'
               <statement> ;

<for-init> ::= <identifier> '=' <expr> ;

<for-incr> ::= <identifier> <assignment-op> <expr> ;

<switch-stmt> ::= SWITCH '(' <expr> ')'
                  '{' { <case-clause> }* [<default-clause>] '}' ;

<case-clause> ::= CASE <integer-constant> ':' { <statement> }* BREAK ';' ;

<default-clause> ::= DEFAULT ':' { <statement> }* BREAK ';' ;

<break-stmt> ::= BREAK ';' ;

<continue-stmt> ::= CONTINUE ';' ;

<return-stmt> ::= RETURN [<expr>] ';' ;

<function-stmt> ::= <function-call> ';' ;

<function-call> ::= <identifier>
                    '(' [ <expr> { ',' <expr>} }* ] ')' ;
```

```
<wait-count-stmt> ::= WAITCNT '(' <identifier> ',' <expr> ')' ';'
                    | WAITCNT '(' <expr> ')' ';' ;

<wait-stmt> ::= ( WAITPEQ | WAITPNE | WAITVID )
                '(' <expr> ',' <expr> ')' ';' ;

<lock-stmt> ::= ( LOCKNEW | LOCKSET | LOCKCLR | LOCKRET )
                '(' <IDENTIFIER> ')' ';' ;

<built-in-op-stmt> ::= ( CMPSUB | REV )
                       '(' <identifier> ',' <expr> ')' ';' ;

<compound-stmt> ::= '{' { <statement> } '}' ;

<nop-stmt> ::= NOP ';' ;

<logical-or-expr> ::= <logical-and-expr>
                    | <logical-or-expr '||' <logical-and-expr> ;

<logical-and-expr> ::= <relational-expr>
                     | <logical-and-expr '&&' <relational-expr> ;

<relational-expr> ::= <expr> '==' <expr>
                    | <expr> '!=' <expr>
                    | <expr> '<' <expr>
                    | <expr> '>' <expr>
                    | <expr> '<=' <expr>
                    | <expr> '>=' <expr>
                    | <lock-expr>
                    | <parity-expr> ;

<lock-expr> ::= ( LOCKSET | LOCKCLR ) '(' <identifier> ')'
                ( '==' | '!=' } ( TRUE | FALSE ) ;

<parity-expr> ::= <parity-function> ( '==' | '!=' } ( ODD | EVEN ) ;

<expr> ::= <inclusive-or-expr>
         | <expr> '#>' <inclusive-or-expr>
         | <expr> '<#' <inclusive-or-expr> ;

<inclusive-or-expr> ::= <exclusive-or-expr>
                      | <inclusive-or-expr> '|' <exclusive-or-expr> ;

<exclusive-or-expr> ::= <and-expr>
                      | <exclusive-or-expr> '^' <and-expr> ;

<and-expr> ::= <shift-expr>
             | <and-expr> '&' <shift-expr>
             | <and-expr> '&~' <shift-expr> ;

<shift-expr> ::= <additive-expr>
```

```
               | <shift-expr> '<<' <additive-expr>
               | <shift-expr> '>>' <additive-expr>
               | <shift-expr> '->' <additive-expr>
               | <shift-expr> '<-' <additive-expr> ;

<additive-expr> ::= <multiplicative-expr>
                  | <additive-expr> '+' <multiplicative-expr>
                  | <additive-expr> '-' <multiplicative-expr> ;

<multiplicative-expr> ::= <unary-expr>
                        | <multiplicative-expr> '*' <unary-expr>
                        | <multiplicative-expr> '/' <unary-expr>
                        | <multiplicative-expr> '%' <unary-expr> ;

<unary-expr> ::= <primary-expr>
               | ! <primary-expr>
               | ~ <primary-expr>
               | - <primary-expr>
               | + <primary-expr> ;

<primary-expr> ::= <identifier>
                 | <function-call>
                 | <global-array>
                 | <built-in-function>
                 | <parity-function>
                 | <built-in-operation>
                 | <integer-constant>
                 | <real-constant>
                 | '(' <expr> ')' ;

<built-in-function> ::= ( SIN | COS | TAN | ASIN | ACOS | LOG | EXP
                      | SQRT | ISQRT ) '(' <expr> ')' ;

<built-in-operation> ::= ( ABS | ABSN | ITOR | RTOI | CEIL | FLOOR
                       | TRUNC | ROUND | FRACT ) '(' <expr> ')' ;

<parity-function> ::= PARITY '(' <expr> ',' <expr> ')' ;

<identifier> ::= <letter> { <letter> | <digit> | '_' }* ;

<letter> ::= (a-zA-Z) ;

<digit> ::= (0-9) ;

<constant> ::= <integer-constant> | <real-constant> ;

<real-constant> ::= <dec-constant> '.' [ <dec-constant> ] ;

<integer-constant> ::= <dec-constant> | <hex-constant>
                     | <oct-constant> | <bin-constant> ;

<dec-constant> ::= { 0-9 | _ } ;
```

```
<hex-constant> ::= '0x' { 0-9 | A-F | _ } ;

<oct-constant> ::= '0o' { 0-7 | _ } ;

<bin-constant> ::= '0b' { 0 | 1 | _ } ;
```