Request For Comment on Draft Specification for The CXING Programming Language.

1 Greetings all. This is a proposed draft of a proposed new programming language. The BDFL of this project is DannyNiu/NJF. The intention of this request for comments is to solicit ideas - advice, suggestions for improvement, as well as critique on preceived defects.

2 While any idea are welcome, they're better received if they're accompanied with counter-arguments, usage illustrations, and/or sketch of implementation, yet the decision of adoption is ultimately made by the BDFL of the project.

3 You may submit your idea and/or queries by opening Issues at GitHub or Gitee, both English and Chinese languages are accepted.

This page is intentionally left blank.

Table of Contents

1. Introduction **7**

2. Lexical Elements. **11**

3. Expressions **13**

3.1. Grouping, Postifix, and Unaries. 13

3.2. Arithmetic Binary Operations 14

3.3. Bit Shifting Operations 15

3.4. Arithmetic Relations 16

3.5. Bitwise Operations 16

3.6. Boolean Logics 17

3.7. Compounds 17

4. Phrases **18**

5. Statements **19**

5.1. Condition Statements 19

5.2. Loops 20

5.3. Statements List 20

5.4. Declarations 21

6. Functions **21**

7. Translation Unit Interface **22**

7.1. Translation Unit Source Code Syntax 22

7.2. Source Code Inclusion 23

8. Language Semantics **23**

8.1. Objects and Values 23

8.2. Object/Value Key Access 24

8.3. Subroutines and Methods 25

9. Types and Special Values **26**

10. Type Definition and Object Initialization Syntax **27**

11. Numerics and Maths **30**

11.1. Rounding 30

11.2. Exceptional Conditions 30

11.3. Reproducibility and Robustness 31

11.4. Recommended Applications of Floating Points 32

12. Runtime Semantics **32**

12.1. Binary Linking Compatibility 32

12.2. Calling Conventions and Foreign Function Interface 32

12.3. Finalization and Garbage Collection 34
13. Standard Library 37
14. Library for the String Data Type 37
15. Library for the Describing Data Structure Layout 38
16. Type Reflection 39
17. Library for Floating Point Environment 39
18. Library for Input and Output 41
19. Library for Multi-Threading 41
19.1. Exclusive and Sharable Objects and Mutices (Mutex) 41
Annex A. Identifier Namespace 43
A.1. Reserved Identifiers 43
A.2. Conventions for Identifiers 43

The cxing Programming Language

4 Build Info: This build of the (draft) spec is based on git commit bd6cfd82887e2e742e902d72383fed82508137c0

1. Introduction

Goal

5 The 'cxing' programming language (with or without caps) is a general-purpose programming language with a C-like syntax that is memory-safe, aims to be thread-safe, and have suprise-free semantics. It aims to have foreign interface with other programming languages, with C as its primary focus.

6 It attempts to pioneer in the field of efficient, expressive, and robust error handling using language design toolsets.

7 The language is meant to be an open standard with multiple independent implementations that are widely interoperable. It can be implemented either as interpreted or as compiled. Programs written in cxing should be no less portable than when it's written in C.

8 Features are introduced on strictly maintainable basis. The reference implementation will be an AST-based interpreter (or a transpiler to C?), which will serve as instrument of verification for additional implementations. The version of the language (if it ever changes) will be independent of the versions of the implementations.

9 The Features section has more information on how the goals are achieved.

Naming

10 Just as Java is a beautiful island in Indonesia, we wanted a name that pride ourselves as Earth-Loving Chinese here in Shanghai, therefore we choose to name our language after the National Nature Reserve Park of Changxing Island. However, the name is too long to be used directly, and "changx" looked too much like 'clang', so we simplified it to "`cxing`", which we find both pleasure in looking at it, and the name giving connotation with an information technology product.

License

11 The language itself and the reference implementation are released into the public domain.

Features

12 To best reflect the intent of the design, the specification shall be programmer-oriented. The purpose of features will be explained, with examples provided on how they're to be used. The syntax and semantic definitions follow.

Memory and Thread Safety

13 The language does not expose pointers - to data or to function - only opaque object handles. It uses reference counting with garbage collection to ensure memory safety. It has separate type domain for `sharable` types catered to multi-threaded access, and `exclusive` types for efficient access within a single thread; only `sharable` types can be declared globally.

Null safety.

14 It's typical to desire *some* result come out of a failing program, it is even more desirable that the failure of a single component doesn't deny the service of users, it's very desirable that error recovery can be easy to program, and it's undesirable that errors cannot be detected.

15 In cxing, errors occur in the forms of nullish values. For the special value `null`, accessing any member of it yields `null`, and calling a `null` as a function returns `null`. Nullish values can be substituted with other alternative values that programs recover from errors.

```
// We do not know the schema of this object, but we know it can be
// one of the two alternatives. Here the "??" punctuation is the
// nullish coalescing operator:
timescale = mp4box.movie.timescale ??
            mp4box.fragments[0].timescale ??
            mp4file.timescale;
```

Nullish NaNs

16 A bit of background first.

17 The IEEE-754 standard for floating point arithmetic specifies handling of exceptional conditions for computations. These conditions can be handled in the default way (default exception handling) or in some alternative ways (alternative exception handling).

18 The 1985 edition of the standard described exceptions and their default handling in section 7, and handling using traps in section 8. These were revised as "exceptions *and* default exception handling" in section 7 as well as "altenate exception handling *attributes*" in section 8 in the 2008 edition of the standard - these "attributes" are associated with "blocks" which (as most would expect) are group(s) of statements. Alternate exception handling are used in many advanced numerical programs to improve robustness.

19 As a prescriptive standard, it was intended to have language standards to describe constructs for handling floating point errors in a generic way that abstracts away the underlying detail of system and hardware implementations. In doing so, the standard itself becomes non-generic, and described features specific to some languages that were not present in others.

20 The cxing language employs null coalescing operators as general-purpose error-handling syntax, and make it cover NaNs by making them nullish. As an unsolicited half-improvement, I (@dannyniu) propose the following alternative description for "alternate exception handling":

> Language ought to specify ways for program to transfer the control of execution, or to evaluate certain expressions when a subset (some or all) of exceptions occur.

21 As an example, the continued fraction function in code example A-16 from "Numerical Computing Guide" of Sun ONE Studio 8 (https://www5.in.tum.de/~huckle/numericalcomputationguide.pdf , accessed 2025-08-15) can be written in cxing as:

```
subr continued_fraction(val N, val a, val b, val x, ref pf, ref pf1)
{
    decl f, f1, d, d1, pd1, q;
    decl j;

    f1 = 0.0;
    f = a[N];
    for(j=N-1; j>=0; j--)
    {
        d = x + f;
        d1 = 1.0 + f;
        q = b[j] / d;
        f1 = (-d1 / d) * q _Fallback f1 = b[j] * pd1 / b[j+1];
        pd1 = d1;
        f = a[j] + q;
    }
    pf = f;
    pf1 = f1;
}
```

22 Reproducibility issues treated in the standard are further discussed in 11.3. Reproducibility and Robustness

23 # 2. Lexical Elements.

24 For the purpose of this section, the POSIX Extended Regular Expressions (ERE) syntax is used to describe the production of lexical elements. The POSIX regular expression is chosen for it being vendor neutral. There's a difference between the POSIX semantic of regular expression and PCRE semantic, the latter of which is widely used in many programming languages even on POSIX platforms, most notably Perl, Python, PHP, and have been adopted by JavaScript. Care have been taken to ensure the expressions used in this chapter are interpreted identically under both semantics.

Comments

25 Comments in the language begin with 2 forward slashses: `//`, and span towards the end of the line. Another form of comments exists, where it begins with `/*` and ends with `*/` - this form of comment can span multiple lines.

26 Comments in the following explanatory code blocks use the same notation as in the actual language.

Identifiers and Keywords

27 An *identfier* has the following production: `[_[:alpha:]][_[:alnum:]]*`. A *keyword* is an identifier that matches one of the following:

```
// Types:
long ulong double val ref

// Special Values:
true false null

// Phrases:
return break continue and or _Fallback

// Statements and Declarations:
decl

// Control Flows:
if else elif while do for

// Functions:
subr method ffi this

// Translation Unit Interface:
_Include extern
```

Numbers

28 *Decimal integer literals* have the following production: `[1-9][0-9]*[uU]?`. When the literal has the "U" suffix, the literal has type `ulong`, otherwise, the literal has type `long`.

29 *Octal integer literals* have the following production: `0[0-7]*`. An octal literal always has type `ulong`.

30 *Hexadecimal integer literals* have the following production: `0[xX][0-9a-fA-F]+`. A hexadecimal literal always has type `ulong`.

31 *Fraction literals* has the following production: `[0-9]+\.[0-9]*|\.[0-9]+`. The literal always has type `double`.

32 *Decimal scientific literals* is a fraction literal further suffixed by a *decimal exponent literal* production: `[eE][-+]?[0-9]+`. The digits of the production indicates a power of 10 to raise fraction part to.

33 *Hexadecimal fraction literal* has the following production: `0[xX]([0-9a-fA-F]+.[0-9a-fA-F]*|.[0-9a-fA-F]+)` - this production is *NOT a valid lexical element* in the language, but *hexadecimal scientific literal* is, which is defined as hex fraction literal followed by *hexadecimal exponent literal* - having the production: `[pP][-+]?[0-9]+`. The digits of the production indicates a power of 2 to raise the fraction part to.

Characters and Strings

34 *Character and string literals* have the following production: `['"]([^\]|\\(["'abfnrtv]|x[0-9a-fA-F]{2,2}|[0-7]{1,3}))['"]`

35 In the 2nd subexpression, each alternative have the following meanings:

1. Escaping
    - For single and double quote characters, they're represented literally and don't delimit the literal.
    - 'a' indicates the `BEL` ASCII 'bell' control character,
    - 'b' indicates the `BS` ASCII backspace character,
    - 'f' indicates the `FF` ASCII form-feed character,
    - 'n' indicates the `LF` ASCII line-feed character,
    - 'r' indicates the `CR` ASCII carriage return character,
    - 't' indicates the `HT` ASCII horizontal tab character,
    - 'v' indicates the `VT` ASCII vertical tab character.
2. Hexadecimal byte literal. The first character is interpreted as the high nibble of the byte, while the second the low.
3. Octal byte literal. The characters (total 3 at most) are interpreted as an octal integer literal used as value for the byte. If there are 3 digits, then the first digit must be between 0 and 3.

36 When single-quoted, the literal is a character literal having the value of the first character as type `long`, the behavior is implementation-defined if there are multiple characters.

37 When double-quoted, the literal is a string literal having type `str`.

Punctuations

38 A punctuation is one of the following:

```
( ) [ ] =? . ++ -- + - ~ ! * / %
<< >> >>> < > & ^ |
<= >= == != === !== && || ?? ? :
= *= /= %= += -= <<= >>= >>>= &= ^= |= &&= ||= ,
; { }
```

3. Expressions

3.1. Grouping, Postfix, and Unaries.

```
primary-expr % primary
: "(" expressions-list ")" % paren
| "[" expressions-list "]" % array
| identifier % ident
| constant % const
;
```

- `paren` : The value is that of the `expressions-list` .
- `array` : The value is an array consisting of elements from the `expressions-list` .
- `ident` : The value is whatever stored in the identifier.
- `const` : The value is that represented by the constant.

```
postfix-expr % postfix
: primary-expr % degenerate
| postfix-expr "=?" primary-expr % nullcoalesce
| postfix-expr "[" expressions-list "]" % indirect
| postfix-expr "(" expressions-list ")" % funccall
| postfix-expr "." identifier % member
| postfix-expr "++" % inc
| postfix-expr "--" % dec
| object-notation % objdef
;
```

- `nullcoalesce` : If the value of `postfix-expr` isn't nullish, then the value is that of `postfix-expr` , otherwise that of `primary-expr` .
- `indirect` : Reads the key identified by `expressions-list` from the object identified by `postfix-expr` . The result is an lvalue.
- `funccall` : Calls `postfix-expr` as a function, given `expressions-list` as parameters. If `postfix-expr` is a `member` , then its `postfix-expr` is provided as the `this` parameter to a potential method call. The result is the return value of the function. See 8.3. Subroutines and Methods for further discussion.
- `member` : Reads the key identified by the spelling of `identifier` from the object identified by `postfix-expr` . The result is an lvalue.
- `inc` : Increment `postfix-expr` by 1. The result is the pre-increment value of `postfix-expr` . `postfix-expr` MUST be an lvalue.
- `dec` : Decrement `postfix-expr` by 1. The result is the pre-decrement value of `postfix-expr` . `postfix-expr` MUST be an lvalue.
- `objdef` : See 10. Type Definition and Object Initialization Syntax.

39 **Note**: Previously, the close-binding null-coalescing operator was `->` , this was changed as it had been desired to reserve it for a 'trait' static call syntax where the first argument of a subroutine (i.e. non-method function) receives the value of or a reference to the left-hand of the operator. This is tentative and no commitment over this had been made yet. All in all, the close-binding null-coalescing operator is now `=?` . (Note dated 2025-09-26.)

```
unary-expr % unary
: postfix-expr % degenerate
| "++" unary-expr % inc
| "--" unary-expr % dec
| "+" unary-expr % positive
| "-" unary-expr % negative
| "~" unary-expr % bitcompl
| "!" unary-expr % logicnot
;
```

- `inc` : Increment `unary-expr` by 1. The result is the post-increment value of `unary-expr`. `unary-expr` MUST be an lvalue.
- `dec` : Decrement `unary-expr` by 1. The result is the post-decrement value of `unary-expr`. `unary-expr` MUST be an lvalue.
- `positive` : The result is that of `unary-expr` implicitly converted to a number if necessary.
- `negative` : The result is the negative of `unary-expr`, which is implicitly converted to a number if necessary.
- `bitcompl` : The result is the bitwise complement of `unary-expr` under integer context.
- `logicnot` : The result is 0 if `unary-expr` is non-zero, and 1 if `unary-expr` compares equal to 0 (both +0 and -0).

40 For `inc` and `dec` in `unary` and `postfix`, and `positive` and `negative`, operation occur under arithmetic context. For `bitcompl` and `logicnot`, the operation occur under integer context.

3.2. Arithmetic Binary Operations

```
mul-expr % mulexpr
: unary-expr % degenerate
| mul-expr "*" unary-expr % multiply
| mul-expr "/" unary-expr % divide
| mul-expr "%" unary-expr % remainder
;
```

- `multiply` : The value is the product of `mul-expr` and `unary-expr`.
- `divide` : The value is the quotient of `mul-expr` divided by `unary-expr`.
- `remainder` : The value is the remainder of `mul-expr` modulo `unary-expr`.

41 The result of division on integers SHALL round towards 0.

42 The remainder computed SHALL be such that `(a/b)*b + a%b == a` is true.

43 If the divisor is 0, then the quotient of division becomes positive/negative infinity of type `double` if the sign of both operands are same/different, while the remainder becomes `NaN`, with the "**invalid**" floating point exception signalled.

44 For the purpose of determining the sign of operands, the integer 0 in `ulong` and two's complement signed `long` are considered to be positive.

45 **Editorial Note**: The first 3 of the above 4 paragraphs were together 1 paragraph in a previous version of the draft before 2025-08-25. This had the potential of causing the confusion that remainder is only applicable to integers. Because now remainder is also applicable to floating points, this is first separated into its own paragraph. The rule regarding type conversion on division by 0 is of separate interest, so it's also an individual paragraph now. The 4th paragraph is added on 2025-08-25.

46 **Note**: The condition for determining remainder is equivalent to:

> remainder `x % y` ***shall*** be such `x-ny` such that for some integer `n`, if y is non-zero, the result has the same sign as `x` and magnitude less than that of `y`.

47 These are separate descriptions for integer modulo operator and floating point `fmod` function in the C language, as such, an implementation may utilize these facilities in C. Any inconsistency between these 2 definitions in C are supposedly unintentional from the standard developer's perspective.

48 All of `mulexpr` occur under arithmetic context.

```
add-expr % addexpr
: mul-expr % degenerate
| add-expr "+" mul-expr % add
| add-expr "-" mul-expr % subtract
;
```

- `add`: The value is the additive sum of `add-expr` and `mul-expr`.
- `subtract`: The value is the difference of subtracting `mul-expr` from `add-expr`.

49 All of `addexpr` occur under arithmetic context.

3.3. Bit Shifting Operations

```
bit-shift-expr % shiftexpr
: add-expr % degenerate
| bit-shift-expr << add-expr % lshift
| bit-shift-expr >> add-expr % arshift
| bit-shift-expr >>> add-expr % rshift
;
```

- `lshift`: The value is the left-shift `bit-shift-expr` by `add-expr` bits.
- `arshift`: The value is the arithmetic-right-shift `bit-shift-expr` by `add-expr` bits. This is done without regard to the actual signedness of the type of `bit-shift-expr` operand.
- `rshift`: The value is the logic-right-shift `bit-shift-expr` by `add-expr` bits. This is done without regard to the actual signedness of the type of `bit-shift-expr` operand.

50 All of `shiftexpr` occur under integer context.

51 **Side Note**: There was left and right rotate operators. Since there's only a single 64-bit width in native integer types, bit rotation become meaningless. Therefore those functionalities will be offered in the standard library method functions.

3.4. Arithmetic Relations

```
rel-expr % relops
: bit-shift-expr % degenerate
| rel-expr "<" bit-shift-expr % lt
| rel-expr ">" bit-shift-expr % gt
| rel-expr "<=" bit-shift-expr % le
| rel-expr ">=" bit-shift-expr % ge
;
```

- `lt` : True if and only if `rel-expr` is less than `bit-shift-expr` .
- `gt` : True if and only if `rel-expr` is greater than `bit-shift-expr` .
- `le` : True if and only if `rel-expr` is less than or equal to `bit-shift-expr` .
- `ge` : True if and only if `rel-expr` is greater than or equal to `bit-shift-expr` .

52 All of `relops` occur under arithmetic context. If either operand is NaN, then the value of the expression is false.

```
eq-expr % eqops
: rel-expr % degenerate
| eq-expr "==" rel-expr % eq
| eq-expr "!=" rel-expr % ne
| eq-expr "===" rel-expr % ideq
| eq-expr "!==" rel-expr % idne
;
```

- `eq` : True if left operand equals the right under arithmetic context; or if one is `null` , the other is of the integer value 0. False otherwise.
- `ne` : True if left operand does not equal the right operand. This includes the case where one operand is of integer values other than 0 and the other is `null` . False otherwise.
- `ideq` : True if left operand equals the right under arithmetic context; or if both are `null` . False otherwise.
- `idne` : True if left operand does not equal the right operand. This includes the case where one operand is of the integer value 0 and the other is `null` . False otherwise.

3.5. Bitwise Operations

```
bit-and % bitand
: eq-expr % degenerate
| bit-and "&" eq-expr % bitand
;

bit-xor % bitxor
: bit-and % degenerate
| bit-xor "^" bit-and % bitxor
;

bit-or % bitxor
: bit-xor % degenerate
| bit-or "|" bit-xor % bitor
;
```

- `bitand` : The value is the bitwise and of 2 operands.

- `bitxor` : The value is the bitwise exclusive-or of 2 operands.
- `bitor` : The value is the bitwise inclusive-or of 2 operands.

53 All of the bitwise operations occur under integer context.

3.6. Boolean Logics

```
logic-and % logicand
: bit-or % degenerate
| logic-and "&&" bit-or % logicand
;

logic-or % logicand
: logic-and % degenerate
| logic-or "||" logic-and % logicor
| logic-or "??" logic-and % nullcoalesce
;
```

- `logicand` : if the first operand is zero or `null` , then this is the result and the second operand is not evaluated, otherwise, it's the value of the second operand.
- `logicor` : if the first operand is non-zero and non- `null` , then this is the result and the second operand is not evaluated, otherwise, it's the value of the second operand.
- `nullcoalesce` : Refer to `postfix-expr` .

3.7. Compounds

```
cond-expr % tenary
: logic-or % degenerate
| logic-or "?" expressions-list ":" cond-expr % tenary
;
```

- `tenary` : The `logic-or` is first evaluated. If it's non-zero and non- `null` , then `expressions-list` is evaluated; otherwise, `cond-expr` is evaluated; The result is whichever `expressions-list` or `cond-expr` evaluated.

```
assign-expr % assignment
: cond-expr % degenerate
| unary-expr "=" assign-expr % directassign
| unary-expr "*=" assign-expr % mulassign
| unary-expr "/=" assign-expr % divassign
| unary-expr "%=" assign-expr % remassign
| unary-expr "+=" assign-expr % addassign
| unary-expr "-=" assign-expr % subassign
| unary-expr "<<=" assign-expr % lshiftassign
| unary-expr ">>=" assign-expr % arshiftassign
| unary-expr ">>>=" assign-expr % rshiftassign
| unary-expr "&=" assign-expr % andassign
| unary-expr "^=" assign-expr % xorassign
| unary-expr "|=" assign-expr % orassign
| unary-expr "&&=" assign-expr % conjassign
| unary-expr "||=" assign-expr % disjassign
;
```

- `directassign` : writes the value of `assign-expr` to `unary-expr` .
- *compound assignments*: writes the computed value to `unary-expr` .

54 See 8.2. Object/Value Key Access for further discussion.

```
expressions-list % exprlist
: assign-expr % degenerate
| expressions-list "," assign-expr % exprlist
;
```

- `exprlist` : A list of expressions.
  - In the context of function calls and arrays, all entities constitutes the list, and elements are evaluated in arbitrary order.
  - In the context of an expression phrase, `expressions-list` is first evaluated, then `assign-expr` is evaluated next, and the value of the expression is that of `assign-expr` .

4. Phrases

55 Between expressions and statements, there are phrases.

56 Phrases are like expressions, and have values, but due to grammatical constraints, they lack the usage flexibility of expressions. For example, phrases cannot be used as arguments to function calls, since phrases are not comma-delimited; nor can they be assigned to variables, since assignment operators binds more tightly than phrase delimiters. On the other hand, phrases provides flexibility in combining full expressions in way that wouldn't otherwise be expressive enough through expressions due to use of parentheses.

```
primary-phrase % primaryphrase
: expressions-list % degenerate
| flow-control-phrase % flowctrl
;
```

- `degenerate` : The value of this phrase is that of the expression.
- `flowctrl` : This phrase alters the normal control flow, it has no value.

```
flow-control-phrase % flowctrl
: control-flow-operator % op
| control-flow-operator label % labelledop
| "return" % returnnull
| "return" expression % returnexpr
;
```

- `op` : Apply the flow-control operation to the inner-most applicable scope.
- `labelledop` : Apply the flow-control operation to the labelled statement scope.
- `returnnull` : Terminates the executing function. If the caller expected a return value, it'll be `null` .
- `returnexpr` : Terminates the executing function with return value being that of `expression` .

```
control-flow-operator: % flowctrlop
: "break" % break
| "continue" % continue
;
```

- `break` : Terminates the applicable loop.

- `continue` : Skip the remainder of the applicable loop body and proceed to the next iteration.

```
and-phrase % andphrase
: primary-phrase % degenerate
| and-phrase "and" primary-phrase % conj
;

or-phrase % orphrase
: and-phrase % degenerate
| or-phrase "or" and-phrase % disj
| or-phrase "_Fallback" and-phrase % nullcoalesce
;
```

- `conj` : Refer to `logic-and` .
- `disj` : Refer to `logic-or` .
- `nullcoalesce` : Refer to `postfix-expr` .

5. Statements

```
statement % stmt
: ";" % emptystmt
| identifier ":" statement % labelled
| or-phrase ";" % phrase
| conditionals % cond
| while-loop % while
| do-while-loop % dowhile
| for-loop % for
| "{" statements-list "}" % brace
| declaration ";" % decl
;
```

- `emptystmt` : This does nothing in a function body.
- `labelled` : Identifies the statement with a label.
- `brace` : Executes `statements-list` .

5.1. Condition Statements

```
conditionals % condstmt
: predicated-clause % base
| predicated-clause "else" statement % else
;
```

- `else` : Executes `predicated-clause` , if none of its statement(s) were executed due to no predicate evaluated to true, then `statement` is executed.

```
predicated-cluase % predclause
: "if" "(" expressions-list ")" statement % base
| predicate-clause "elif" "(" expressions-list ")" statement % genrule
;
```

- `base` : Evaluate `expressions-list` (in expression phrase context as mentioned in the 3.7. Compounds), if it's true, then `statement` is executed, otherwise it's not executed.

- `genrule` : Executes `predicate-clause` , if none of its statement(s) were executed due to no predicate evaluated to true, then evaluate `expressions-list` , if that is still not true, then `statement` is not executed, otherwise, `statement` is executed.

5.2. Loops

```
while-loop % while
: "while" "(" expressions-list ")" statement % rule
;
```

- `rule` : To execute `rule` , evaluate `expressions-list` , if it's true, then execute `statement` and then execute `rule` .

```
do-while-loop % dowhile
: "do" "{" statements-list "}" "while" "(" expressions-list ")" ";" % rule
;
```

- `rule` : To execute `rule` , execute `statements-list` , then evaluate `expressions-list` , if it's true, then execute `rule` .

```
for-loop % for
: "for" "(" expressions-list ";"
            expressions-list ";"
            expressions-list ")" statement % classic

| "for" "(" declaration ";"
            expressions-list ";"
            expressions-list ")" statement % vardecl
;
```

- `classic` : Evaluate `expressions-list` before the first semicolon, then execute the for loop by invoking the "execute the for loop once" recursive procedure described later.
- `vardecl` : Evaluate `declaration` , then execute the for loop in a fashion similar to `classic` .

57 To execute the for loop once, evaluate `expressions-list` after the first semicolon, if it's true, then `statement` is evaluated, then the `expressions-list` after the second semicolon is evaluated, and the for loop is executed once again. For the purpose of "proceeding to the next iteration" as mentioned in `continue` , the `expressions-list` after the second semicolon is not considered part of the loop body, and is therefore always executed before proceeding to the next iteration.

58 The description here used the word "once" to describe the semantic of the loop in terms of "functional recursion", where "functional" is in the sense of the "functional programming paradigm".

5.3. Statements List

```
statement-list % stmtlist
: statement ";" % base
| statement-list statement ";" genrule
;
```

- `base` : `statement` is executed, the semicolon is a delimitor.
- `genrule` : `statement-list` is first executed, then `statement` is executed.

5.4. Declarations

59 Because the value of a variable that held integer value may transition to `null` after being assigned the result of certain computation, the variable needs to hold type information, as such, variables are represented conceptually as "lvalue" native objects. (Actually, just value native objects, as their scope and key can be deduced from context.)

```
declaration % decl
: "decl" identifier % singledecl
| "decl" identifier "=" assign-expr % signledeclinit
| declaration "," identifier % declarelist1
| declaration "," identifier "=" assign-expr % declarelist2
;
```

- `singledecl` : Declares a variable with the spelling of the identifier as its name, and initialize its value to `null` .
- `singledeclinit` : Declares a variable with the spelling of the identifier as its name, and initialize its value to that of `assign-expr` .
- `declarelist1` : In addition to what's declared in `declaration` , declare another variable in a way similar to `singledecl` .
- `declarelist2` : In addition to what's declared in `declaration` , declare another variable in a way similar to `singledeclinit` .

6. Functions

```
function-declaration % funcdecl
: "subr" identifier arguments-list statement % subr
| "method" identifier arguments-list statement % method

| "ffi" "subr" type-keyword identifier arguments-list ";" % ffisubr
| "ffi" "method" type-keyword identifier arguments-list ";" % ffimethod
;

arguments-list % arglist
: "(" ")" % empty
| arguments-begin ")" % some
;

arguments-begin % args
: "(" type-keyword identifier % base
| arguments-begin "," type-keyword identifier % genrule
;

type-keyword % typekw
: "val" % val
| "ref" % ref
| "long" % long
| "ulong" % ulong
| "double" % double
;
```

60 For `subr` and `method` , the function so defined or declared is a non-FFI function. The type of its parameters must be `val` or `ref` . Its return type is implicitly `val` and is not spelled out.

61 For `ffisubr` and `ffimethod`, the function so defined or declared is an FFI function. The type of its parameters can be `val`, `ref`, `long`, `ulong`, `double`, and they're passed to the function as described in 12.2. Calling Conventions and Foreign Function Interface. The return type MUST NOT be `ref` as prohibited in 8.3. Subroutines and Methods.

62 For `subr` and `method`, function body MUST be either `emptystmt`, in which case the `function-declaration` declares a function, or `brace`, in which case it defines a function. FFI functions (`ffisubr` and `ffimethod`) can be declared, but cannot be defined in cxing.

63 The type and order of parameters between all declarations and the definition of the function MUST be consistent, furthermore, whether a function is a method or a subroutine, is or is not an FFI function MUST be consistent. The name of the parameters may be changed in the source code of a program. Depending on the context, this may provide the benefit of both explanative argument naming in declaration, and avoidance identifier collision in function definition when the argument is appropriately renamed.

7. Translation Unit Interface

64 A translation unit consist of a series of function declarations and definitions. Because definition of objects occur during run time, it's not possible to define data objects of static storage duration in cxing, this is recognized as unfortunate and accepted as a design decision.

65 A translation unit in cxing correspond to relocatable code object, or a file contain such information. We choose such definition to emphasize binary runtime portability; the word "translate/translation" doesn't require translation to occur - it's allowed for an implementation to interpret the source code and execute it directly for when it can be achieved. The terms "translation unit" and "relocatable object" take their usual commonly accepted meanings in building programs and applications.

7.1. Translation Unit Source Code Syntax

66 The goal symbol of a source code text string is `TU` - the translation unit production. It consist of a series of entity declarations.

```
TU % TU
: entity-declaration % base
| TU entity-declaration % genrule
;

entity-declaration % entdecl
: "_Include" string-literal ";" % srcinc
| "extern" function-declaration % extern
| function-declaration % implicit
;
```

67 There MUST NOT be more than 1 *definition* of a function.

68 By default, all entity declarations are internal to the translation unit. For a declaration to be visible in multiple translation units, it must be declared "external" with the `extern` keyword.

69 As a best practice, external declarations should be kept in "header" files, and included (explained shortly) in a source code file. The recommended filename extension for cxing source code file is `.cxing`, and `.hxing` for headers (named after the Hongxing Yu village on the Changxing Island).

7.2. Source Code Inclusion

70 Source code inclusion is a limited form of reference to external definitions. This is *not* preprocessing, *not* importation, and *not* substitute for linking. Source code inclusion is exclusively for sharing the declarations in multiple source code files and translation units.

71 By default, header files are first searched in a set of pre-defined paths. (These paths are typically hierarchy organized and implemented using a file system.) If the header isn't found in the pre-defined paths, then it's searched relative to the path of the source code file. However, if the string literal naming the header file begins with `./` or `../`, then it's first searched relative to the path of the source code file, then the pre-defined set of paths.

8. Language Semantics

8.1. Objects and Values

72 An *object* may have properties, properties may also be called members.

73 **Note**: The word "property" emphasizes the semantic value of the said component, while the word "member" emphasizes its identification. Both words may be used interchangeably consistent with the intended point of perspective.

74 The internals of an object is largely opaque to the language. The primary interface to objects are functions that operates on them.

75 **Note**: Functions in compiled implementations follow platform ABI's calling convention. Because certain opaque object types (such as the string type) in the runtime may need to be used in functions compiled on different implementations, the consistency of their structure layout is essential.

76 A *native object* is a construct for describing the language. It has a fixed set of properties, and are copied by value; mutating a native object does not affect other copies of the object.

77 An *value* is a native object with the following properties:

1. the value *proper*,
2. a *type*,
3. for an *lvalue* - which can be the left operand of respective assignment expression, there's the following additional properties:
    1. a *scope object* - this can be a block, an object; for `sharable` types, this can also be the "global" scope,
    2. a *key* - this identifies/is the name of the lvalue under the scope.

78 Other native objects (may) exist in the language.

79 All values have a (possibly empty) set of type-associated properties that're immutable. These type-associated properties take priority over other properties. The behavior is UNSPECIFIED when these properties are written to.

80 **Note**: The data structure for the value native objects are further defined to enable the interoperability of certain language features. Values are such described to enable discussion of "lvalue"s, alternative implementations may use other conceptual models for lvalues should they see fit.

8.2. Object/Value Key Access

81 As described in 8.1. Objects and Values objects have properties. The key used to access a value on an object is typically a string or an integer.

82 When the key used to access a property is an integer, there may be a mapping from the integer to a string defined by the implementation of the runtime. Portable applications SHOULD NOT create objects with mixed string and integer keys. All implementations of the runtime SHALL guarantee there's no collision between any key that is the valid spelling of an identifier and any integer between 0 and 10^{10} inclusive.

83 **Note**: The limit was chosen for efficiency reasons. While implementing a number to string conersion would immediately solve the issue of collision between numerical and identifier keys, it's slightly inefficient. A second option would be to pad the integer word with bytes that can never be valid in identifiers, this would be the best of both worlds. Yet considering most applications won't be needing such big array, and those that do would probably go for the string type in the standard library, a limit is set so that plausible real-world applications and implementations can enjoy the efficiency enabled by such latitude.

84 To read a key from an object:

1. if the key refers to one of the type-associated properties:
    1. a native object results consisting of:
        - value-proper: the value of this property,
        - type: the type of this property.
2. if the key is not one of the type-associated properties:
    1. if the key `__get__` is one of the type-associated properties, then this method is used to retrieve the actual property:
        1. this method is called with the object as its `this` parameter,
        2. this method is called with the key as a `val`,
        3. its return value is augmented with the 'scope' and 'key' being the object and the key used to access this property, to yield an lvalue.
    2. if the key `__get__` is not defined as one of the type-associated properties, then an lvalue being `null` augmented with 'scope' and 'key' being the object and the key used to access this property is returned.

85 **Note**: the return value from 2.1.3. may be `null`.

86 To write a key onto an object:

- For the purpose of this section, it is assumed that the storing of the value onto the object is done using the `__set__` type-assocaited method property. The object is passed as the `this` parameter, the key as the first parameter as a `val`, and the value as the value as the the second parameter as a value native object. See 12.2. Calling Conventions and Foreign Function Interface- the new value is assigned to the identified key on the object, with the following exceptions:

- if the write is a compound assignment (i.e. any assignment of form other than `directassign`), then the key is read from the object, the computation part of the compound assignment is performed, and the result is stored written to they key on the object.

87 **Note**: Compound assignment is different from loading the values from both sides of the assignment operator, perform the computation, then storing the result into the key, as the latter performs the read on the lvalue twice.

88 When a key is being deleted from an object:

- For the purpose of this section, it is assumed that the deletion of the value from the object is done using the `__unset__` type-associated method property. The object is passed as the `this` parameter, the key as the first parameter as a `val`.

- any resources used by the value associated with the key on the object is finalized, if the `__final__` method property exists on the object, then it's called, the key is then removed from the object, after which the member identified by the key is considered not defined on the object from this point onwards (until it's being written to again).

89 **Note**: Destruction of values and finalization of resources are further discussed in 12.3. Finalization and Garbage Collection.

8.3. Subroutines and Methods

90 Both *subroutines* and *methods* are codes that can be executed in the language, the distinction is that methods have an implicit `this` parameter while subroutines don't - for compiled implementations, this is significant, as it causes difference in parameter passing under a given calling convention.

91 Subroutines and methods are distinct types, as such there's no restriction that subroutines have to be called directly through identifiers or that methods have to be identified through a member access.

- When accessed from the key of an object:
  - a method carries an implicit `this` parameter,
  - a subroutine does not carry the implicit `this`.
- When invoked by name:
  - the implicit `this` in a method is `null`.
  - a subroutine is invoked as is.

92 For both subroutines and methods, they have both FFI and non-FFI variants. FFI stands for foreign function interface. In non-FFI variants their arguments are dynamically typed, and can be passed either by value or by reference. For FFI variants, the type of their arguments and return values have to be declared explicitly.

93 (Non-FFI) subroutine functions, method functions, and FFI subroutine functions and FFI method functions are 4 distinct types.

The `val` and `ref` Function Operand Interfaces

94 For non-FFI functions, when a parameter is declared with `val`, then the corresponding argument is passed by value; when declared with `ref`, then passed by reference.

95 No type of function may return `ref` for the simple reason that certain value that may potentially be returned are of "temporary" storage duration - they exist only on the stack frame of called function, and are destroyed when they go out of scope. Adding compile-time check to verify that such variables are not returned as reference are more complex to implement than simply just outlawing them outright.

96 The `this` parameter receive its arguments as `val` in the runtime. This allows methods to be assigned to different objects and access other object properties - including type-associated properties such as `__get__`, etc.

97 **Note**: In a previous revision, there was a note claimed that `this` being a pointer handle. The idea back then was that when cxing runtime is implemented with SafeTypes2, certain APIs of the library can be used without modification. However, better runtime implementation stratagy was discovered which resulted in the introduction of type-associated properties. And so `this` parameter is received as a `val` in all (both actually) types of methods. Still, to facilitate the correct passing of parameters, it necessitates the distinction between methods and subroutines.

9. Types and Special Values

The `long` and `ulong` types

98 The `long` type is a signed 64-bit integer type with negative values having two's complement representation. The `ulong` type is an unsigned 64-bit integer type. Both types have sizes and alignments of 8 bytes.

99 **Note**: 32-bit and narrower integer types don't exist natively, primarily because of the year 2038 problem and issue with big files. However, respective type objects for smaller integers, as well as those for `float` / `binary32` and other floating point types are defined in the standard library to interpret data structures in byte strings.

100 The keyword `bool` is used exclusively as an alias for the type `long`, there is no restriction that a `bool` can store only 0 or 1, it exist primarily for programmers to clarify their intentions.

The `double` type

101 The `double` type is the floating point number type. It should correspond to the IEEE-754 (a.k.a. ISO/IEC-60559) `binary64` type - that is, it should have 1 sign bit, 11 exponent bits, and 52 mantissa bits. The type have sizes and alignment of 8 bytes.

The `str` type

102 The string type `str` is not a built-in type, instead, it's an opaque object type defined in the standard library. The string type has significance in the `indirect` member access operator in a `postfix-expr` postfix expression.

The `true` and `false` special values

103 The special value `true` is equal to 1 in type `long`. The special value `false` is equal to 0 likewisely.

The `null` and `NaN` special values

104 The `null` special value results in certain error conditions. Accessing any properties (unless otherwise stated) results in `null`; calling `null` as if it's a function results in `null`. `null` compares equal to itself.

105 The `NaN` special value represents exceptional condition in mathematical computation. `NaN` does not compare equal to any number, or to itself.

106 Both `null` and `NaN` are considered nullish in coalescing operations.

107 See [11. Numerics and Maths](#) for furher discussion.

Implicit Type and Value Conversion

108 Values and/or their types may be converted used under certain contexts:

- The types `long` and `ulong` are collectively "integer context";
- the type `double` is the "floating point context";
- the types `long` , `ulong` , and `double` are collectively "arithmetic context".

109 Under a integer context:

- the special value `null` have value 0,
- all opaque objects have a single value of 1,
- floating point values are converted by discarding fractional part, with the behavior on overflow being UNSPECIFIED.

110 Under the floating point context:

- integers are converted preserving value to the extent allowed by precision.
- the special value `null` is converted to `NaN` .
- all opaque objects are converted to `+1.0` .

111 Under arithmetic context:

- before the following occur, `null` are converted to 0 in `long` , and opaque objects to 1, also in `long` .
- operations involving only `long` s results in `long` operands;
- operations involving `ulong` but not `double` results in `ulong` operands;
- operations involving `double` results in `double` ;

112 **Note**: The special value `NaN` always have type `double` .

113 **Note**: It was considered to have certain operations in integer context that involved floating points to have NaNs, but this was dropped for 2 simple reasons: 1st, the current *conversion* rule is much simpler written, and 2nd, there exist prior art with JavaScript.

10. Type Definition and Object Initialization Syntax

114 There's a simple syntax in cxing for creating compound objects and types:

```
decl Complex := namedtuple() { 're': double, 'im': double };
decl I := Complex() { 're': 0, 'im': 1 };
decl sockaddr := dict() { 'host': "example.net", 'port': 443 };
```

115 In the above scenario,

- `namedtuple()` *factory function* creates such object that is a *type object* that creates another type object with 2 members named "re" and "im", this type is assigned to `Complex`, which is then used to create a "complex number" with the value of the imaginary unit;
- `dict()` factory function creates a type object that creates a dictionary, initializing `sockaddr` with 2 members - "host" with the value of "example.net" and "port" with 443.

116 `namedtuple`, `Complex`, and `dict` are "type objects", of which, with `namedtuple` being sort of a meta.

117 A type object contains an method property named `__initset__` declared as follow:

```
[ffi] method [val] __initset__(ref key, ref value);
```

118 The `__initset__` function may be defined in cxing or in a foreign language - if the latter, then calling conventions for foreign function interface must be followed per 12.2. Calling Conventions and Foreign Function Interface.

```
objdef-start % objdefstart
: objdef-start-comma % comma
| objdef-start-nocomma % nocomma
;

objdef-start-comma % objdefstartcomma
: objdef-start-nocomma "," % genrule
;

objdef-start-nocomma % objdefstartnocomma
: postfix-expr "{" postfix-expr ":" assign-expr % base
| objdef-start-nocomma "," postfix-expr ":" assign-expr % genrule
;

object-notation % objdef
: postfix-expr "{" "}" % empty
| objdef-start "}" % some
;
```

119 The `postfix-expr` MUST NOT be `inc` or `dec`. Furthermore, if `postfix-expr` is `degenerate`, then the primary expression MUST NOT be `array` or `const`.

120 On encountering a `postfix-expr` that is a type object, the key-value pairs enclosed in the braces delimited by commas are taken and the `__initset__` method is called on them in turn. The key is the value of the postfix expression on the left side of the colon, while the value is that of the assignment expression on the right side of the colon. After this completes, the newly created object will receive a property named `__proto__`, which will be assigned the value of `postfix-expr`.

121 The `array` production of primary expressions is a syntax sugar that invokes `__initset__` with elements in the `expressions-list` as value and successive integer indicies as key, starting with 0.

11. Numerics and Maths

122 **Note**: Much of this section is motivated by a desire to have a self-contained description of numerics in commodity computer systems, as well as an/a interpretation / explanation / rationale of the standard text that's at least more useful in terms of practical usage than the standard text itself.

11.1. Rounding

123 IEEE-754 specifies the following rounding modes:

- **roundTiesToEven**: This is MANDATORY and SHALL be the default within a thread when the thread starts. The floating point value closest to the infinitely precise result is returned. If there are two such values, the one with an even digit value at the position corresponding to the least significant of the least significant digits of the two values will be returned.
- **roundTowardPositive**: The least representable floating point value no less than the infinitely precise result is returned.
- **roundTowardNegative**: The greatest representable floating point value no greater than the infinitely precise result is returned.
- **roundTowardZero**: The representable floating point value with greatest magnitude no greater than that of the infinitely precise result is returned.

124 The standard library provides facility for setting and querying the rounding mode in the current thread. The presence of other rounding modes (e.g. **roundTiesToAway**, **roundToOdd**, etc.) are implementation-defined.

11.2. Exceptional Conditions

125 Infinity and NaNs are not numbers. It is the interpretation of @dannyniu that they exist in numerical computation strictly to serve as error recovery and reporting mechanism.

126 IEEE-754 specifies the following 5 exceptions:

- **invalid**: known as "invalid operation" in standard's term. This is when:
  - operations involving signalling NaNs,
  - "cancellation of infinities" in additive, multiplicative, or some other domains. Examples include subtracting infinity from infinity, multiplying 0 with infinity, or dividing 0 with 0 or infinity with infinity.
  - the input is outside the domain of the operation, e.g. sqrt(-1).
- **pole**: known as "division by zero" in standard's term. A pole results when operation by an operand results in an infinite limit. Particular cases of this include 1/0, tan(90°), log(0), etc.
- **overflow**: this is when and only when the result exceeds the magnitude of the largest representable finite number of the floating point data type after rounding. The data type is `double` a.k.a. `binary64` in our language.
- **underflow**: this is when a tiny non-zero result having an absolute value below b^{emin}, where b is the radix of the floating point data type - 2 in our case , and $emin$ is, in our case -1022.

  Note: $emin$ can be derived as: $2 - 2^{ebits-1}$, where $ebits$ is the number of bits in the exponents, which is 11 in our case.

- **inexact**: this is when the result after rounding differs from what would be the actual result if it were calculated to unbounded precision and range.

127 The standard library provides facility for querying, clearing, and raising exceptions. Alternate exception handling attributes are implemented in the language as error-handling flow-control constructs, such as null-coalescing expression and phrases operators, as well as execution control functions.

11.3. Reproducibility and Robustness

128 Floating points have a fixed significand width as well as limited range(s) of exponents, as such, they're very similar to *scientific notations*, further as such, they suffer from the same **inaccuracy** problems as any notation that truncates a large fraction of value digits. However, this do yield a favorable trade-off in terms of implementation (and to some extent, usage) **efficiency**.

129 IEEE-754 recommends that language standard provide a mean to derive a sequence (graph actually, if taken dependencies into account) of computation in a way that is deterministic. Many C compilers provide options that make maths work faster using arithmetic associativity, commutativity, distributivity and other laws (e.g. *fast-math* options), cxing make no provision that prevents this - people favoring efficiency and people favoring accuracy should both be audience of this language.

130 The root cause of calculation errors stem from the fact that the significand of floating point datum are limited. This error is amplified in calculations. A way to quantify this error is using the "unit(s) in the last place" - ULP. There are various definitions of ULP. Vendors of mathematical libraries may at their discretion document the error amplification behavior of their library routines for users to consult; framework and library standards may at their discretion specify requirements in terms error amplification limits. Developers are reminded again to recognize, and evaluate at their discretion, the trade-off between accuracy and efficiency.

131 Because of the existence of calculation errors, floating point datum are recommended as instrument of data exchange. In fact, earlier versions of the IEEE-754 standard distinguished between interchange formats and arithmetic formats. Because arithmetics and the format where it's carried out are essentially black-box implementation details, the significance of arithmetic formats is no longer emphasized in IEEE-754.

132 The recommended methodology of arithmetic, is to first derive procedure of calculation that is a simplified version of the full algorithm, eliminating as much amplification of error as possible, then feed the input datum elements into the algorithm to obtain the output data. The procedure so derived should take into account of any exceptions that might occur.

133 For example, `(a+b)(c+d) = ac+ad + bc+bd` have 2 additions and 1 multiplication on the left-hand side and 3 additions and 4 multiplications on the right-hand side.

134 a program may first attempt to calculate the left hand side, because it has less chance of error amplification. However, if the addition of `c` and `d` overflows but they're individually small enough such that their multiplication with either `a` and `b` won't overflow, yet the sum of `a` and `b` underflows in a certain way that's catastrophic, the the whole expression may become `NaN` .

135 In this case, a fallback expression may then compute the right-hand side of the expression, possibly yielding a finite result, or at least one that arithmetically make sense (i.e. infinity).

136 The result of computation carried out using such "derived" procedure will certainly deviate from the result from of a "complete" algorithm. Developers should recognize that robustness may be more important in some applications than they may expect. In the limited circumstances where an application in reality is less important, or in fact be prototyping, developer may at their careful discretion, excercise less engineering effort when coding a numerical program.

137 Finally, it is recognized that large existing body of sophisticated numerical programs are written using 3rd-party libraries, and/or using techniques that're under active research and not specified and beyond the scope of many standards. Developers requiring high numerial sophistication and robustness are encouraged to consult these research, and evaluate (again) the accuracy and efficiency requirements at their careful discretion.

11.4. Recommended Applications of Floating Points

138 The recommended applications of floating points in computer, are *Computer Graphics*, *Signal Processing*, *Artificial Intelligence*, etc.

139 Typical characteristics of these applications include:

- datum need to be over real-valued domain,
- tolerance of loss of precision by end user.

12. Runtime Semantics

140 While the features and the specification of the language is supposed to be stable, **as a guiding policy**, in the unlikely event where certain interface in the runtime posing efficiency problem are to be replaced with alternatives, deprecation periods are given in the current major version of the runtime (and thus the language), before removal in a future major version should that happen; in the even more unlikely event where certain interface exposes a vulnerability so fundamental that necessitates its removal, the language along with its runtime is revised, a new version is released, and the vulnerable version is deprecated immediately. The versioning practice is in line with recommendation by Semantic Versioning.

12.1. Binary Linking Compatibility

141 Dynamic libraries and applications linking with dynamic libraries programmed in cxing should not statically link with the cxing runtime. Unless no opaque objects is passed between translation units compiled by different implementations (which is unlikely), statically linking to different incompatible implementations of the runtime may result in undefined behavior when opaque objects and the functions that manipulates them are from different implementations.

142 The version of the runtime and the version of the language specification are coupled together to make it easy to determine which version of runtime should be used to obtain the features of relevant version of the language. If the standard library is to be provided, then the runtime should be provided as part of the standard library, the name of the linking library file should be the same for both the runtime and for when it's extended into/as standard library.

143 The recommended name for the library corresponding to version 0.1 of the specification is `libcxing0.so.1` for systems using the UNIX System V ABI such as Linux, BSDs, and several commercial Unix distros. For the Darwin family of operating systems such as macOS, iOS, etc. the recommended name is `libcxing0.1.dylib` .

144 For some platforms such as Windows, vendors have greater control over the dynamic libraries bundled with the programs in an application. Therefore no particular recommendations are made for these platforms.

12.2. Calling Conventions and Foreign Function Interface

145 The types `long` and `ulong` are passed to functions as C types `int64_t` and `uint64_t` respectively; the type `double` is passed as the C type `double` ; handles to full objects and opaque objects are passed as C language object pointers.

146 The "value" and "lvalue" native object are defined as the following C structure types:

```
enum types_enum : uint64_t {
    valtyp_null = 0,
    valtyp_long,
    valtyp_ulong,
    valtyp_double,

    // the opaque object type.
    valtyp_obj,

    // `porper.p` points to a `struct value_nativeobj`.
    valtyp_ref,

    // FFI and non-FFI subroutines and methods.
    valtyp_subr = 6,
    valtyp_method,
    valtyp_ffisubr,
    valtyp_ffimethod,

    // 10 types so far.
};

struct value_nativeobj;
struct type_nativeobj;

struct value_nativeobj {
    union { double f; uint64_t l; int64_t u; void *p; } proper;
    union {
        const struct type_nativeobj *type;
        uint64_t pad; // zero-extend the type pointer to 64-bit on ILP32 ABIs.
    };
};

struct lvalue_nativeobj {
    struct value_nativeobj value;

    // The following fields are for lvalues:
    void *scope;
    void *key;
};

struct type_nativeobj {
    enum types_enum typeid;
    uint64_t n_entries;

    // There are `n_entries + 1` elements, last of which `type` being the only
    // `NULL` entry in the array.
    struct {
        const char *name;
        struct value_nativeobj *member;
    } static_members[];
};
```

147 For the special value `null`, there are 2 accepted representations that implementations MUST anticipate:

- `typeid` having an enumeration value of 0 - `valtyp_null`.
- `value.p.proper` having `NULL` with `typeid` having `valtyp_obj`.

148 For non-FFI functions, parameters declared with type `val` receive arguments as the `struct value_nativeobj` structure in runtime binding; values are returned in similarly in the `struct value_nativeobj` structure type. (As mentioned in 8.3. Subroutines and Methods, no function may return a `ref`.)

149 For FFI functions, parameters declared with type `long`, `ulong`, and `double` receive arguments as their respective C language type, and in accordance to the ABI specification of relevant platform(s); values are returned according to their type declaration also in accordance to relevant platform ABI definitions.

150 For both non-FFI and FFI functions, parameters declared as `ref` receive arguments as the `struct value_nativeobj *` pointer type in runtime binding.

151 Methods receive `this` as their first argument as the `ref` language type (i.e. the `struct value_nativeobj *` runtime pointer type).

12.3. Finalization and Garbage Collection

152 Resources are generically defined as what enables a program to run and function, and assciated with it. When a value is destroyed, the resources associated with it are finalized and released, which may lead to the resources be free for reuse elsewhere.

153 **Note**: On a reference-counted implementation (which is conceptually prescribed), releasing an object "decreases" its reference count, and when the reference count reaches 0, the resources are "freed". Under implementation-defined circumstances, an object may be released by all, but still referenced somewhere (e.g. reference cycle), which require garbage collection to fully "free" the object and its resources.

154 **Editorial Note**: Previously (before 2025-09-26), finalize and destroy were used interchangeably; now finalize refer to that of resource and destroy refer to that of values (i.e. the concept of value native objects).

```
ffi subr null cxing_gc();
```

155 The `cxing_gc` foreign function invokes the garbage collection process.

156 **Note**: In part because of the runtime implementation need to be informed of destruction of values to finalize relevant resources, more pressingly because of benefit to the design of idiomatic standard library features, copying and destruction of values are now being defined. To define the concepts in terms of reference counts would mean to depend on intrinsic implementation details, and also that there's circular dependency in definition. Seeking an alternative, it's discovered that copying and destroying are paired concepts that must be described together, and this is the approach that will be taken right now.

157 To *copy* a value, means to preserve its existence in the event of its *destruction*, which causes the value ceases to exist; when a value is copied, the value and the copied value can both exist, and the destruction of either don't affect the existence of the other.

158 The `__copy__` property is a method that copies its `this` argument and returns "the copy". The `__final__` property is a method that releases the resources used by the value before the destruction of the value.

159 The `__copy__` and `__final__` may not necessarily be type-associated properties, programs can define their own types with copy and finalization methods as long as the object they're implementing these methods for have a `__set__` property.

160 **Note**: Primitive types such as `long`, `ulong`, and `double` may not need a `__copy__` method - runtime recognizing these sort of types may copy them in any way that may be assumed reasonable according to common sense. For types without a `__final__` method, it is assumed that there are no resource consumed by the value beyond what's already in the value native object structure.

13. Standard Library

161 In the following sections, special notations that're not part of the langauge are used for ease of presentation.

162 **The meaning of such notation**:

```
[ffi] {method,subr} [<type>] identifier(args);
```

163 *is as follow*:

164 The bracketed `[ffi]` means this is a method or a subroutine can be either FFI or non-FFI. When it's FFI, it's return type is `<type>`.

165 **The meaning of such notation**:

```
<name1>(<name2>) := { ... }
```

166 *is as follow*:

167 The entity identified by `<name1>` is a subclass of `<name2>` (typically `val`), and consist of additional members enumerated by the ellipsis `...`. The word "subclass" is used here only to imply that object of type `<name1>` may be used anywhere `<name2>` is expected. `<name2>` is not optional, because it signifies to implementors of the runtime how an argument of such type are to be passed.

168 **Note**: The notation is inspired by Python. Object-oriented programming is not a supported paradigm of cxing. The notation is strictly for presentation, and does not correspond to any existing language feature.

169 Because cxing is a dynamically typed language, typing is not enforced, and the implementation does not diagnose typing errors (because there aren't any). Checking the characteristics of an object is entirely the responsibility of codes that use it.

14. Library for the String Data Type

```
str(val) := {
  [ffi] method [long] len(),
  [ffi] method [str] trunc(ulong newlength),
  [ffi] method [str] putc(long c),
  [ffi] method [str] puts(str s),
  [ffi] method [str] putfin(),
  [ffi] method [long] cmpwith(str s2), // efficient byte-wise collation.
  [ffi] method [bool] equals(str s2), // constant-time, cryptography-safe.
  [ffi] method [structureddata] map(val structlayout),
};

structureddata(val) := {
  [ffi] method [val] unmap(),
}
```

170 The string type `str` is a sequence of bytes.

171 A string has a *length* that's reported by the `len()` function, and can be altered using the `trunc()` function.

172 The `putc()` function can be used to append a byte whose integer value is specified by `c`, to the end of the string; the `puts()` function can be used to append another string to the end; both `putc()` and `puts()` may buffer the input on the working context of the string, such buffer need to be flushed using the `putfin()` function before the string is used in other places.

173 For `trunc()`, `putc()`, `puts()`, and `putfin()`, the object itself is returned on success, and `null` is returned on failure.

174 The `cmpwith()` returns less than, equal to, or greater than 0 if the string is less than, the same as, or greater than `s2`. The strict prefix of a string is less than the string to which it's a prefix of.

175 The `equals()` function returns `true` if the string equals `s2` and false otherwise. If the 2 strings are of the same length, it is guaranteed that the comparison is done without cryptographically exploitable time side-channel.

176 The `map()` function creates an object that is a parsed representation of the underlying data structure. This object can be used to modify the memory backing of the data structure if the corresponding memory backing is writable. The memory backing is writable by default, and the circumstances under which it's not is implementation-defined.

177 The `unmap()` function unmaps the parsed representation, thus making it no longer usable. The variable can then only be finalized (or overwritten, which would imply a finalization). The `trunc()` function cannot be called on the string unless there's no active mapping of the string.

15. Library for the Describing Data Structure Layout

```
decl char, byte; // signed and unsigned 8-bit,
decl short, ushort; // signed and unsigned 16-bit,
decl int, uint; // signed and unsigned 32-bit,
decl long, ulong; // signed and unsigned 64-bit,
decl half, float, double; // binary16, binary32, binary64.
// decl _Decimal32, _Decimal64; // not supported yet.
// decl huge, uhuge, quad, _Decimal128; // too large.

struct_inst(val) := {
  [ffi] method [val] __initset__(ref key, ref value),
};

packed_inst(val) := {
  [ffi] method [val] __initset__(ref key, ref value),
};

union_inst(val) := {
  [ffi] method [val] __initset__(ref key, ref value),
};

[ffi] subr [struct_inst] struct();
[ffi] subr [packed_inst] packed();
[ffi] subr [union_inst] union();
```

178 The representations for `char`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `half`, `float`, and `double` are explained in the comments following their description; their alignments are the same as their size. These are known as primitive types.

179 A `struct_inst` object represents an instance of structure that is suitabl for use in a call to the `map()` method of the `str` type, representing a structure with members laid out sequentially and suitably align. A `packed_inst` is similar, but with no alignment - all members are packed back-to-back. A `union_inst` creates a structure layout object with all members having the same start address at byte 0 and alignment of the strictestly-align member.

180 Each object of type `struct_inst`, `packed_inst`, and `union_inst` are type objects. They're initialized with members using the syntax as described in 10. Type Definition and Object Initialization Syntax; and are created using the `struct()`, `packed()`, and `union()` factory functions respectively.

181 Primitive types and structure layout object may be array-accessed to create array types of respective types.

182 For example:

```
decl AesBlock = union() { 'b': byte[16], 'w': uint[4] };
decl Aes128Key = AesBlock[11];
```

183 The variable `AesBlock` holds a structure layout object of 128 bits, and `Aes128Key` holds the 11 round keys for an AES-128 cipher.

16. Type Reflection

```
[ffi] subr [bool] isnull(val x);
[ffi] subr [bool] islong(val x);
[ffi] subr [bool] isulong(val x);
[ffi] subr [bool] isdouble(val x);
[ffi] subr [bool] isobj(val x, val proto);
```

184 The functions `isnull`, `islong`, `isulong`, `isdouble`, determines whether the value is the special value `null`, of type `long`, type `ulong`, or type `double` respectively. The function `isobj` determines whether the value is an object, if `proto` is not `null`, then it further determines whether the `__proto__` member of the object is equal to `proto`.

17. Library for Floating Point Environment

Rounding Mode

185 **Tentative Note**: The exact form of the following functionality is not yet ultimately decided, and may change over time.

```
[ffi] subr [long] fpmode(long mode);
```

186 Returns the currently active rounding mode. If mode is one of the supported mode, then set the current rounding mode to the specified mode. The value -1 is guaranteed to not be any supported mode.

187 The following modes are supported:

- 0: round ties to even,
- 3: round towards positive,
- 5: round towards negative,
- 7: round towards zero.

188 The support for other modes are unspecified.

189 The encoding of modes are as follow:

- 0: nearest - rounding to nearest, and defer to next bits only on ties.
- 1: directed - always make decision based on next bits.

190 The next bits are as follow:

- 0<<1: even - the value with an even least significant digit is chosen,
- 1<<1: positive - the greater value is chosen.
- 2<<1: negative - the lesser value is chosen,
- 3<<1: zero - the value with lesser magnitude is chosen,
- 4<<1: away - the value with greater magnitude is chosen,
- 5<<1: odd - the value with an odd least significant digit is chosen.

191 Such encoding is chosen to cater to possible future extensions. Not all possible rounding modes offer numerical analysis merit, as such some of the combinations are not valid on some implementations.

Floating Point Exceptions

192 **Tentative Note**: The exact form of the following functionality is not yet ultimately decided, and may change over time.

```
// Tests for exceptions
[ffi] subr [bool] fptestinval(); // **invalid**
[ffi] subr [bool] fptestpole(); // **division-by-zero**
[ffi] subr [bool] fptestoverf(); // **overflow**
[ffi] subr [bool] fptestunderf(); // **underflow**
[ffi] subr [bool] fptestinexact(); // **inexact**

// Clears exceptions
[ffi] subr [bool] fpclearinval(); // **invalid**
[ffi] subr [bool] fpclearpole(); // **division-by-zero**
[ffi] subr [bool] fpclearoverf(); // **overflow**
[ffi] subr [bool] fpclearunderf(); // **underflow**
[ffi] subr [bool] fpclearinexact(); // **inexact**

// Sets exceptions
[ffi] subr [bool] fpsetinval(); // **invalid**
[ffi] subr [bool] fpsetpole(); // **division-by-zero**
[ffi] subr [bool] fpsetoverf(); // **overflow**
[ffi] subr [bool] fpsetunderf(); // **underflow**
[ffi] subr [bool] fpsetinexact(); // **inexact**

// Exceptions state.
[ffi] subr [long] fpexcepts(long excepts);
```

193 The `fptest*`, `fpclear*`, and `fpset*` functions tests, clears, and sets the corresponding floating point exceptions in the current thread.

194 The `fpexcepts` function returns the current exceptions flags. If `excepts` is a valid flag, then the exceptions flag in the current thread will be set, otherwise, it will not be set. The value 0 is guaranteed to be a valid flag meaning all exceptions are clear; the value -1 is guaranteed to be an invalid flag. The validity of other flag values are UNSPECIFIED. When the implementation is being hosted by a C implementation, the encoding of `excepts` is exactly that of `FE_*` macros, with the clear intention to minimize unecessary duplicate enumerations as much as possible.

18. Library for Input and Output

195 **Planning**: Postponed.

19. Library for Multi-Threading

19.1. Exclusive and Sharable Objects and Mutices (Mutex)

```
sharableObj(val) := { /* Sharable objects may be used across threads */ }
mutex_inst(sharableObj) := { /* Mutices are a class of sharable objects */ }

[ffi] subr [mutex_inst] mutex(val v);

mutex_inst(val) := {
  [ffi] method [exclusiveObj] acquire(),
}

exclusiveObj(val) := {
  // Exclusive objects can only be used by 1 thread at a time,
  // but is more efficient than shared objects when used.
  [ffi] method [val] __copy__(),
  [ffi] method [null] __final__(),
}
```

196 The `mutex()` function creates a mutex which is a sharable object that can be used across threads. The argument `v` will be an exclusive object protected by the mutex.

197 The `acquire()` method of a mutex returns a value native object representing `v` - when the function returns, it is guaranteed that the thread in which it returns is the only thread holding the value protected by the mutex, and that until the value goes out of scope, no other thread may simultaneously use the value.

198 The `__copy__()` and `__final__()` properties increments and decrements respectively, a conceptual counter - this counter is initially set to 1 by `acquire()` and any future functions that may be defined fulfilling similar role; when it reaches 0, the mutex is 'unlocked', allowing other threads to acquire the value for use.

199 **Note**: A typical implementation of `acquire()` may lock a mutex, sets the conceptual counter to 1, creates and returns a value native object. A typical implementation of the `__copy__()` method may be as simple as just incrementing the conceptual counter. A typical implementation of the `__final__()` method may decrement the counter, and when it reaches 0, unlocks the mutex.

200 **Note**: The conceptual counter is distinct from the reference count of any potential resources used by the value protected by the mutex and the mutex itself.

201 -- TODO --: Thread management need to cater to the type system of cxing, C/POSIX API have thread entry points take a pointer, but cxingdon't expose pointers. This along with other issues are to be addressed before the threading library is formalized. The part with mutex is roughly okay now.

Annex A. Identifier Namespace

202 The goal of this section is to avoid ambiguity of identifiers in the global namespace - i.e. avoiding the same identifier with conflicting meanings.

203 To this end, "commonly-used" refers to the attribute of an entity where it's used so frequently that having a verbose spelling would hamper the readability of the code.

204 When an identifier consist of multiplie words, the following terms are defined:

- Pascal Case: where each word, including the first, are capitalized,
- Camel Case: where each word except the first are capitalized,
- Snake Case: underscore-concatentated lowercase words,
- Verbose Case: underscore-concatenated Pascal case.

A.1. Reserved Identifiers

205 Identifiers in the global namespace that begins with an underscore, followed by an uppercase letter is reserved for standardization by the language.

206 Identifiers which consist of less than 10 lowercase letters or digits are potentially reserved for standardization by the language, as keywords or as "commonly-used" library functions or objects. Although the use of the word "potentially" signifies that the reservation is not uncompromising, 3rd-party library vendors should nontheless refrain from defining such terse identifiers in the global namespace.

A.2. Conventions for Identifiers

- For type objects, Pascal or Verbose case is recommended.
- For subroutines, Snake or Verbose case is recommended.
- For members and methods, Camel or Pascal case is recommended.