Funk Programming Language Reference Manual

Naser AlDuaij, Senyao Du, Noura Farra, Yuan Kang, Andrea Lottarini
{nya2102, sd2693, naf2116, yjk2106, al3125} @columbia.edu

26 October, 2012

1 Introduction

This manual describes Funk, a functional programming language with an imperative structure. Funk is designed to allow simpler parallel programming and higher order functions. Features defining the language include dynamic function declaration, asynchronous block declarations and strong typing. This manual describes in detail the lexical conventions, types, scoping rules, built-in functions, and grammar of the Funk language.

2 Lexical conventions

2.1 Identifiers

An identifier in Funk represents a programmer-defined object. The identifier starts with a letter or an underscore, and is optionally followed by letters, numbers or underscores. An identifier name is thus defined by the following regular expression:

```
['a' - 'z' 'A' - 'Z' '_']['a' - 'z' 'A' - 'Z' '0' - '9' '_']*
```

2.2 Keywords

Funk has a set of reserved keywords that can not be used as identifiers.

2.2.1 Statements and Blocks

The following keywords indicate types of statements or blocks:

```
var func async
```

2.2.2 Control Flow

The following keywords are used for control flow

```
if else for break return
```

2.2.3 Types

Each primitive type, integer, double, boolean and character, has a name that the program uses for declarations.

```
int double bool char
```

A function type, and a function declaration begins with the **func** keyword.

2.2.4 Built-in Functions

The following keywords are reserved for built in functions.

```
print double2int int2double bool2int int2bool
```

2.3 Constants

Funk supports integer, double, character, and boolean constants, otherwise known as literals, inside expressions. Any array with any level of nesting can be also expressed as a literal. Arrays of characters have a more concise representation as string literals.

2.3.1 Integer

Define a decimal digit with the following regular expression:

```
digit = ['0' - '9']
```

An int is a 64-bit signed integer, and consists of at least one digit.

```
digit+
```

2.3.2 Double

A double is a 64-bit ("double precision") floating point number. Like the floating-point constant in the C language, according to Kernighan and Ritchie, the constant consists of an integer part of digits, as described in 2.3.1, a decimal point, a fractional part –ie. the part of the floating point whose absolute value is less than 1, and an exponential part, starting with 'e', followed by an optional sign, and then at least 1 digit, which indicates the power of 10 in the scientific notation. Define the regular expression of the exponential part as follows:

```
exp = 'e' ['+' '-']? ['0'-'9']+
```

If the decimal point is present, at least one of the the integer and fractional parts must also be present –the compiler interprets an absent part as 0. If there is no decimal point, the integer part and the exponent must be present:

```
((digit+ '.' digit* | '.' digit+) exp?) | (digit+ exp)
```

Therefore, the following literals are valid doubles:

```
1.2 .2 1. 1.2e3 .2e3 1.e3 1e+3 1e-3
```

However, strings lacking digits either before or after the exponent marker or decimal point, if each exists, and integer constants are not valid doubles:

```
12 e3 . .e 1.2e 1.2e+
```

2.3.3 Boolean

The boolean type has two predefined constants for each truth value, and no other values:

```
true false
```

2.3.4 Characters and Character Arrays

Characters are single, 8-bit, ASCII characters. Generally, the character values are representable in their printable form. However, not all characters are printable, and some printable characters conflict with the lexical conventions of characters and character arrays. They are therefore specially marked with a backslash, and considered single characters. Funk supports the following escape sequences:

- New line:

  ```
  \n
  ```

- Carriage return:

  ```
  \r
  ```

- Tab:

  ```
  \t
  ```

- Alarm:

  ```
  \a
  ```

- Double quotation marks, to fit inside character array literals:

  ```
  \”
  ```

- The backslash itself:

  ```
  \\
  ```

A literal of character type consists of a single character or escape sequence inside two single quotes:

```
’c’
```

Note that the printable characters with escape sequences, ie. the double quotation marks and backslash, do not have to be escaped, because the compiler processes a single character between two single quotes without any translations (which is why there is no escape sequence for single quotes). But the compiler also accepts their escaped form. For example, the following two literals are equivalent:

```
’\’, ’\\’
```

A string is a character array. Since strings are widely used we considered a special representation for string constants which begins and ends with unescaped double quotes.:

```
”foo”
```

This is equivalent to:

```
[3]char {’f’,’o’,’o’}
```

Each single character or backslash-character pair is a single entry of the character array.

2.4 Punctuation

Funk supports primary expressions using the previously-mentioned identifiers and constants. Primary expressions also include expressions inside parentheses. In addition, parentheses can indicate a list of arguments in a function declaration, a function variable declaration, or a function call:

```
f() //function call
```

Braces indicate a statement in blocks, including blocks that make up function bodies. They are also used for array literals:

```
{
//this is a block
}

var a [2]int = [2]int{1,2} // more about arrays in later sections
```

Brackets indicate array types and access to array elements:

```
a[0] //access to element 0 in array a
[3]int //this is a data type
```

Semicolon is used to separate statement and expression in a for loop:

```
for i=0;i<10;i=i+1
```

Comma is used to separate elements of a list:

```
int a,b = 0,1
```

2.5 Comments

Multiline comments in Funk start with /* and terminate with the next */. Therefore, multiline comments do not nest. Funk also supports single-line comments starting with // and ending at the end of the line.

2.6 Operations

Funk supports a number of operations that resemble their arithmetic and boolean counterparts.

2.6.1 Value Binding

A single equal sign indicates assignment in an assignment or declaration statement:

```
=
```

2.6.2 Operators

Funk supports the following binary operators:

```
-, *, /, %
&, ^, |, <<, >>
==, !=, <=, <, >=, >
&&, ||
```

Funk also supports the following unary negations:

```
!, ~
```

And there are two operations that can be unary or binary, depending on the expression:

```
+, -
```

3 Syntax

3.1 Program structure

At the highest level, the program consists of declarations only. While a program can initialize global variables with expressions, all other statements, including assignments to existing variables, must be inside function bodies:

`Global function or variable declarations (interchangeable)`

program:

    declaration
    program declaration

declaration:

    funcdec
    vardec `new-line`
    `new-line`

3.1.1 Variable Declarations

Variables can be declared and initialized globally, or locally in a function, with the *vardec* rule

`var` new-id-list var-type `new-line`

var is the keyword. *new-id-list* can be one or more newly declared identifier tokens. *var-type* is the type, which has to be one of the allowed primitive types or function, or a possibly-multidimensional array of the aforementioned types.

var-type:

    array-markeropt single-type

single-type:

`func` (formal-listopt) var-typeopt
`int`
`double`
`char`
`bool`

array-marker:

array-level
array-marker array-level

array-level:

[]
[expression]

The declaration ends with a new line.
Variables can also be initialized:

`var` new-id-list var-type = actual-list `new-line`

actual-list is a comma-separated list of expressions and anonymous function declarations (which cannot be inside expressions, since there are no operations on functions). 3.2 will describe expressions in detail.

3.1.2 Global and Anonymous Functions

Global functions are defined the following way:

`func` ID (formal-listopt) var-typeopt block

func is the keyword. *ID* identifies the instance of the function. If *var-type* is not specified, the function cannot return a value. *block* contains the function body. *formal-list* (optional) in the parentheses would include the formal arguments, which are clusters of variables of the same type, indicated by a *var-type*. If there is only one variable of the type, the *single-type* use of *formal-list* can omit the name, but for the purpose of function declaration headers, all names must be present:

formal-list:

formal-type-cluster
formal-list , formal-type-cluster

formal-type-cluster:

var-type
new-id-list var-type

The mandatory global function in Funk is main, which marks the entry point of a program. It takes no parameters (so it does not support command line arguments), and returns no values.

```
func main() {
   print(”Hello world!\n”)
   print(”Goodbye world!\n”)
}
```

Listing 1: main function in a sample program

Anonymous functions are declared with a similar syntax and are used like expressions (see 3.3) but without an identifier, as the program can refer to them as variables. Except for the fact that the parameters must all have names here, the function header, excluding the body, resembles the corresponding *single-type*:

`func` (formal-listopt) var-typeopt block

3.2 Expressions

3.2.1 Primary expressions

Primary expressions consist of:

- Literals
  - The primitive literals mentioned in 2.3, which the grammar calls *INT_LIT*, *DOUBLE_LIT*, *BOOL_LIT* and *CHAR_LIT*.
  - Character arrays of the token *STRING*
  - Identifiers of the token *ID*
  - Arrays: Array literals begin with their type, and the expressions they contain must be of the same type; they are of the same primitive type as the array, but have one less array level

    actual-arr:
        arr-header { actual-listopt }

    arr-header:
        arrary-marker single-type

    For example, a $3x2$, 2-level array looks like this:

    `[3][2]int {[2]int{1,2},[2]int{3,4},[2]int{5,6}}`

- Asynchronous blocks have the value and type of whatever the block returns.

    `async` block

  However, the program will not wait for the return value until it is required in another expression. Therefore, if the block is inside another expression, it always blocks until completion. And if it is an rvalue, the program does not block until the corresponding lvalue is used inside another expression.

```
{
   var i int = 3 + async { return 0 } /* blocks immediately */

   i = async { return 1 } /* does not block yet */
   var j int = i /* blocks */

   j = async { return 2 } /* does not block yet */
   i = 1 + j /* blocks */
}
```

Listing 2: Blocking conditions for async

- Parenthesized expressions

  (expression)

3.2.2 Precedence of Operations

We list the operations in order from highest to lowest precedence. The full grammar enforces precedence using similar rules as the C grammar, according to its LRM.

1. Under certain conditions, the two postfix expressions appear outside of expressions that the program evaluates.

   The program can call functions if the *expression* before the parentheses is of type *func*, and the types of the actual parameters in *actual-list*, if any, match the formal arguments of the function. The type of the expression is the type of the return value, so, as an expression, the function must return a value.

   func-call-expr:
   expression (actual-listopt)

   An expression can extract its value from an array. The left *expression* must be an array type, so that the value resulting from the array access has a type that is one array level shallower than that of the left expression (eg. if a has type $[][]int$, then $a[0]$ has type $[]int$). The right *expression*, inside the brackets, must be of integer type.

   obj-get-expr:
   primary-expr
   expression [expression]

   We reveal the *primary-expr* rule not only to show the precedence, but also because accessing a pure primary expression is useful when this expression is used in assignments, which 3.3.1 will explain.

2. Prefix, unary operations

   un-op expression

   - + optional positive sign for *int* and *double*
   - - 2-complement negation for *int* and *double*
   - ! Boolean negation for *bool*
   - ~ 1-complement (bitwise) negation for *int*

3. Binary operations. In addition to specific restrictions, the two operands of a binary operation must have the same type. Unless noted otherwise, they will return the same type. The subscripts are added for clarity.

   $expression_a$ bin-op $expression_b$

   The items in the following list start from the highest precedence, and all operations are left-associative.

   (a) `*, /, %`: Arithmetic multiplication and division are valid for both *int* and *double* types. The modulo operation gives the positive remainder when $expression_b$ divides $expression_a$. Therefore, the modulo operation only applies to integers.

   (b) `+, -`: Arithmetic addition and subtraction of *int* and *double* values.

   (c) `<<, >>`: Bitwise shift for *int* values. Return $expression_a$ shifted left or right by $expression_b$ bits. $expression_b$ can be negative, in which case the shift direction is reversed, and the expression shifts by the absolute value of $expression_b$.

   (d) `<, >=, >, =<`: Real-number comparison for *int*, *double*, and *char* values. Returns *bool* value.

   (e) `==, !=`: Equality and inequality for *int*, *double*, *char*, and *bool* values. Returns *bool* value.

(f) &: Bitwise AND for *int* values. The program evaluates both $expression_a$ and $expression_b$.

(g) ^: Bitwise XOR for *int* values. The program evaluates both $expression_a$ and $expression_b$.

(h) |: Bitwise OR for *int* values. The program evaluates both $expression_a$ and $expression_b$.

(i) &&: Logical AND for *bool* values. The program evaluates $expression_b$ if and only if $expression_a$ is *true*

(j) ||: Logical OR for *bool* values. The program evaluates $expression_b$ if and only if $expression_a$ is *false*

3.3 Statements

3.3.1 Assignments

assign-stmt:

obj-get-expr-list = actual-list `new-line`

An assignment statement defines an *obj-get-expr* list as the lvalues. Unlike its usage in *expression*, each lvalue must be either an *ID* token, or any n-level array access of an array object indicated by an *ID* token. This rule assures that there is an identifier that can reach the assigned value.

The rvalues are in actual-list, an expression list followed by a new line to end the statement. The expressions are of various types, and each i^{th} lvalue will store the evaluated value of the i^{th} rvalue. Therefore, each i^{th} lvalue and rvalue must match in type, and the number of lvalues and rvalues must be the same.

3.3.2 Blocks and Control Flow

- Block

  A block is defined inside curly braces, which can include a possibly-empty list of statements.

- Selection statement

  A selection statement is an if or if-else statement that takes an expression that evaluates to a *bool* value:

  `if` expression block
  `if` expression block else block

  There exists ambiguity with the selection statement: "if expression if expression else", which is why Funk selects between blocks rather than statements.

- Iteration statement

  An iteration statement begins with the *for* keyword. We support three types of for loops. The first is a regular for loop with a starting assignment, a boolean loop condition expression, and an assignment for advancing to the next iteration. The three parts are separated by semicolons. The other loops are a for loop with one expression (similar to while loop), and a for loop with no expression. The expressions must evaluate to a *bool* value. A missing expression implies *true* –ie. an infinite loop:

  `for` assign-stmtopt ; expressionopt ; assign-stmtopt block
  `for` expressionopt block

- Jump statements

  The return keyword accepts an optional anonymous function or expression and ends with a newline. It exits out of the smallest containing function or async body. *async* is an expression described in 3.2.1

  return expressionopt `new-line`

The *break* keyword breaks iteration of the smallest containing for loop. In other words, it jumps to the code immediately following the loop.

- A statement can call functions using the *func-call-expr* syntax. Changes to the state of the function due to the call persist. Therefore, the function *expression* to the left of the parentheses must be an *ID*, or an n-level array access of an *ID*, ie. an lvalue that can store the changed function instance. The function does not need to return a value, and the program discards any value that it does return.
- *vardec*

4 Scoping Rules

Funk uses lexical scoping: the scope of an object is limited to the block in which it is declared, and overrides, or suspends, the scope of an object with the same identifier declared in a surrounding block.

4.1 Lexical Scoping with Blocks

When the program declares object o with identifier id in declaration D_o, D_o can assign a value to o using id. Moreover, in B_o, the block that directly contains D_o, any statement after D_o that assigns to or reads from id in fact does so from o, with two important exceptions.

The first is function closure, which subsection 4.2 will cover in more detail. The second exception is the declaration of the same variable inside the first approximation of the scope of o. When id is on the left side of another declaration, D'_o, inside a block B'_o, contained in B_o, then id is not bound to o starting from the *left* side of D'_o until the end of B'_o. Instead, D'_o will create a new object, o', and id will refer to it until the end B'_o, aside from the previously-mentioned exceptions.

```
 1 {
 2    var i int = 0
 3    print(i) /* 0 */
 4    if (i == 0) {
 5       i = i + 1 /* this will refer to the i from line 2 */
 6       print(i) /* 1 */
 7       var i char = 'c' /* suspend scope of i from line 2 */
 8       print(i) /* c */
 9       if (i == 'c') {
10          var i double = 1.1 /* suspend scope of i from line 7 */
11          print(i) /* 1.1 */
12       }
13       print(i) /* c */
14    }
15    print(i) /* 1 */
16 }
```

Listing 3: Scope suspended in contained block

Note that for globally-declared variables, B_o includes the entire code. However, in this case, id can only be used inside a function body.

In certain contexts, id cannot refer to multiple objects. While D'_o can redeclare id in a block that B_o contains, B_o cannot directly contain D'_o; the programmer cannot redeclare id in the same block.

```
{
   var i int = 0 /* first declaration */
   print(i) /* 0 */
```

```
    var i int = 10 /* illegal redeclaration */
}
```

Listing 4: Illegal variable redeclaration

Likewise, the program cannot use the old o when redefining it in block B'_o

```
{
    var i int = 0
    if (i == 0) {
        /* illegal use of old object on right side */
        var i int = i + 1
    }
}
```

Listing 5: Cannot use old object for defining new object with the same Id in a nested block.

4.2 Function Closure

Each function instance has an environment associated with it. Let C be the scope of object o, with identifier id. For a function, F, inside C, let $S = \{s_0, \ldots s_n\}$ be all the statements in F that use id that would refer to o according to the rules detailed in the previous subsection. If s_i is the first statement in which id appears on the left hand side, or is called as a function instance, all $s_j : j < i$, as well as id on the right side of s_i, use the value of o as it appeared before the declaration of F, or since the last execution of the same instance of F. On the other hand, all statements $s_j : j \geq i$ use a new o' that was a distinct, deep copy from o. Likewise, any changes to o after F does not change the values used in F, even when the program executes the instance after the change. As a consequence, the only effect of a function on the outside scope is through its return values, and not through side effects or parameters, which are passed as values or deep copies.

```
{
    var i int = 0
    print(i) /* 0 */

    var inc func() int = func() int {
        print(i)

        i = i + 1

        print(i)

        return i
    }

    print(i) /* 0 */

    var j int = inc() /* ”01” */
    print(i) /* 0 */
    print(j) /* 1, because it contains the return value */

    i = 100
    print(i) /* 100 */

    j = inc() /* ”12”, which the last assignment did not change */

    /* deep copy of function */
    var inc_inc func() int = func() int {
        return inc()
    }

    j = inc_inc() /* ”23” */
    print(j) /* 3 */

    j = inc_inc() /* ”34” */
    print(j) /* 4 */

    j = inc() /* ”23”, which inc_inc did not change */
    print(j) /* 3 */
}
```

Listing 6: Examples of closures; block comments indicate the value printed to standard output

Closure is necessary primarily in order to support higher order functions. Functions in funk can be passed around and subsequently executed outside of their original scope. Consider the following example as an explanation why closures are necessary:

```
// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int {
    var a, b int := 0, 1
    return func() int {
        a, b := b, a+b
        return a
    }
}

func main() {
    f := fib()
    // Function calls are evaluated left-to-right.
    print(f()) //1
    print(f()) //1
    print(f()) //2
    print(f()) //3
    print(f()) //5
}
```

Listing 7: Function invoked outside its original scope

The fib function returns an anonymous function to the caller. This anonymous function has variables *a* and *b* as free variables and gets executed outside its original scope, specifically in the scope of the main function. What happen is that at function declaration time the anonymous function creates a copy of a and b (the function environment) from the surrounding scope which will be used on subsequent invocations.

Closures are also used to avoid race conditions in async blocks.

```
{
    var i int = 0
    var a, b int

    print(i) /* 0 */

    a = async {
        i = i + 1
        return i
    }
    print(i) /* 0 */

    i = 10
    b = async {
        i = i + 2
        return i
    }

    print(a) /* 1 */
    print(b) /* 12 */
    print(i) /* 10 */
}
```

Listing 8: Closure for async Block

The two async blocks seem to compete for variable i defined in the outer scope. Both blocks will instead create a closure of all their free variables effectively eliminating race conditions.

4.3 Assignment and Parameter Passing

In Funk, we try to minimize side effects, including those that may arise in assignments. Therefore, copy-assignment statements, or assignments that use a variable directly on the right without performing any operations, copy the value, rather than reference, of the variable. In addition, the assignment reads the values of the right hand side before it could change any of them.

Let S be the scope of o as defined in the previous subsections. If o is on the right side of an assignment, a, the target object of the assignment, o_a, identified by id_a, stores a deep copy of the value of o.

```
{
    /* functions */
    var i int = 0
    print(i) /* 0 */

    var inc func() int = func() int {
        i = i + 1

        print(i)

        return i
    }

    inc() /* 1 */

    var inc2 func() int = inc /* copying function */

    inc() /* 2 */
    inc() /* 3 */

    inc2() /* 2 */
    inc() /* 4 */

    /* arrays */
    var arr [2]int = [2]int{0, 1}
    print(arr[0]) /* 0 */

    var arr2 [2]int = arr /* copying array */
    print(arr2[0]) /* 0 */
    arr2[0] = 2
    print(arr2[0]) /* 2 */
    print(arr[0]) /* 0 */
}
```

Listing 9: Deep copies of function instances and arrays

Our language supports assignment of multiple objects. The right hand side of the assignment gets evaluated first **from left to right**, then the results are copied to the objects. Therefore, changes to objects on the left side do not affect their values on the right side. This enables swapping of values of objects.

```
{
    var a, b int = 0, 1
    print(a, b) /* 0 1 */

    /*
     * the assignment changes a and b, but not before it reads the
     * original values
     */
    a, b = b, a
    print(a, b) /* 1 0 --note the swap */
}
```

Listing 10: Assignment to the left side does not affect the right side

However, being free from side effects does not imply that functions are referentially transparent in our language (due to closures). Evaluations of the same function at different times can held different results if the function environment is modified between different function calls. Consider the following example using the Fibonacci closure and multiple assignments.

```
// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int {
    var a, b int := 0, 1
    return func() int {
        a, b := b, a+b
        return a
    }
}

func main() {
     var a,b,c = fib(),fib(),fib() // a=1 b=1 c=2 ..
}
```

Listing 11: Multiple assignment and closures

For consistency with copying-by-value, as well as function closure, function calls also copy parameters by value, evaluating the right side expressions from left to right.

```
{
   var i int = 0
   print(i) /* 0 */

   var inc func(int) int = func(i int) int {
      i = i + 1

      return i
   }

   print(inc(i), i) /* 1 0 */
}
```

Listing 12: No side effects on parameters

5 Type Conversions

Funk is a strongly typed language; therefore it performs no implicit type conversion. It is the responsibility of programmers to convert operands to the correct type. For example, consider arithmetic operations between an integer and a floating point operand:

```
var a int = 1
var b double = 2.0
var c double = a + b //the compiler rejects the expression, as the types are not
                                the same
```

The programmer has to explicitly convert operands:

```
var a int = 1
var b double = 2.0
var c double = int2double(a) + b
```

We believe that this approach is less error prone than implicit conversion. For a complete list of conversion functions see Section 6.

6 Built-in Functions

6.1 Conversion Functions

Funk has four conversion functions to and from the int type:

- double2int(x double) int: The function discards the fractional part of x and returns the integer part of x as an int.
- int2double(x int) double: The function returns a double with the fractional part equal to 0 and the integer part equal to x.
- boolean2int(x bool) int: If x is **true**, the function returns 1. Otherwise it returns 0.
- int2boolean(x int) bool: If x is equal to 0, the function returns **false**. Otherwise it returns **true**.

6.2 The print function

A Funk program performs printing using the **print** function. The syntax and semantics of the **print** function are inspired by the Python 3 function with the same name. Like the **async** keyword, the **print** function is not present in the Go language that we are using as the baseline. Go uses the Printf function included in the fmt package as the standard function for formatted I/O. Even though the authors of Go claim to have better mechanisms than the C language printf [1], the semantics and syntax of fmt.Printf are almost indistinguishable from its C counterpart. Therefore we decided to implement a function similar to the Python 3 print function for the following reasons:

- The print function does not have a format string, making formatted I/O simpler and less error prone.
- The print function is polymorphic, which is also helpful for the programmer.

For example consider the following snippet:

```
var a int = 1
var b int = 2
print (a,”+”,b,” is ”, (a+b)) // prints 1+2 is 3
```

The syntax of our print command is the following:

```
print([expression, ...])
```

The function prints the concatenation of the string representation of each expression to standard output. It does not automatically end the output with a newline, but the user can include expressions whose string representations include a newline.

In contrast, print function in Python 3 has these features that allow programmers to specify 3 optional parameters:

1. `sep` is a separator that print outputs to I/O between every expression in the list.
2. end is a string that the function prints after it has output all the expressions.
3. `file` specifies the destination for the print statement.

We are not implementing the first two because the simplified **print** function can still output any format by manually adding in the separators and the end of line string. And since we are not considering file I/O for our language, we are not going to implement the last as well.

7 Grammar

In the grammar listed below, we have some of undefined terminals like *ID*, *INT_LIT*, *DOUBLE_LIT*, *CHAR_LIT*, *BOOT_LIT* and *STRING*. They are tokens passed in from the lexer. The words in `textwriter` style are terminal symbols given literally, with the exception of `new-line` indicating the line break. And for symbols with the subscript opt, they will be expanded in the actual grammar with a non-terminal that consists of an empty part and the actual symbol. “one of” indicates separate tokens, listed in a single line, to which the non-terminal could expand. And with those indicated above, the grammar represented here will be accepted by the ocamlyacc parser-generator.

program:

    declaration
    program declaration

[1] http://golang.org/pkg/fmt/

declaration:

    funcdec
    vardec `new-line`
    `new-line`

block:

    { stmt-listopt }

stmt-list:

    `new-line`
    stmt-list stmt `new-line`
    stmt-list `new-line`

funcdec:

    `func` `ID` (formal-listopt) var-typeopt block

anon:

    `func` (formal-listopt) var-typeopt block

vardec:

    `var` new-id-list var-type
    `var` new-id-list var-type = actual-list

formal-list:

    formal-type-cluster
    formal-list , formal-type-cluster

formal-type-cluster:

    var-type
    new-id-list var-type

new-id-list

    `ID`
    new-id-list , `ID`

single-type:

    `func` (formal-listopt) var-typeopt
    `int`
    `double`
    `char`
    `bool`

var-type:

    array-markeropt single-type

arr-header:

    array-marker single-type

array-marker:

    array-level
    array-marker array-level

array-level:

    []
    [expression]

actual-arr:

    arr-header { actual-listopt }

actual-list:

    expression
    anon
    actual-list , expression
    actual-list , anon

obj-get-expr-list:

    obj-get-expr
    obj-get-expr-list , obj-get-expr

assign-stmt:

    obj-get-expr-list = actual-list new-line

stmt:

    block
    func-call-expr
    return expression
    return anon
    break
    if expression block
    if expression block else block
    for assign-stmtopt ; expressionopt ; assign-stmtopt block
    for expressionopt block
    assign-stmt
    vardec

expression:

    or-expr

or-expr:

    and-expr
    or-expr || and-expr

and-expr:

    bor-expr
    and-expr && bor-expr

bor-expr:

    bxor-expr
    bor-expr | bxor-expr

bxor-expr:

    band-expr
    bxor-expr ^ band-expr

band-expr:

    eq-expr
    band-expr & eq-expr

eq-expr:

    comp-expr
    eq-expr eq-op comp-expr

eq-op: one of

    ==   !=

comp-expr:

    shift-expr
    comp-expr comp-op shift-expr

comp-op: one of

    <   <=   >   >=

shift-expr:

    add-expr
    shift-expr shift-op add-expr

shift-op: one of

    <<   >>

add-expr:

    mult-expr
    add-expr add-op mult-expr

add-op: one of

    +   -

mult-expr:

    un-expr
    mult-expr mult-op un-expr

mult-op: one of

    `*` `/` `%`

un-expr:

    post-expr
    un-op un-expr

un-op: one of

    `-` `+` `!` `~`

post-expr:

    obj-get-expr
    func-call-expr

obj-get-expr:

    primary-expr
    post-expr [expression]

func-call-expr:

    post-expr (actual-listopt)

primary-expr:

    `INT_LIT`
    `DOUBLE_LIT`
    `CHAR_LIT`
    `BOOL_LIT`
    `STRING`
    `ID`
    actual-arr
    `async` block
    (expression)