FunC: Syntax Specification

Tomi Setsu

April 18, 2026

Abstract

This document describes the FunC surface syntax currently accepted by the compiler in this repository. It is derived directly from `crypto/func/parse-func.cpp`, `crypto/func/keywords.cpp`, and `crypto/func/builtins.cpp`, with one small supplementary dependency on the shared lexer implementation for the exact token and comment rules. The target compiler version is FunC `0.4.6`.

This text is descriptive of the implementation as it exists today. Where a token is reserved by the lexer but not consumed by the parser, that is stated explicitly.

Introduction

FunC is a statically typed source language that compiles into TVM/Fift-style assembly. This document focuses on its accepted source syntax rather than on the generated output code.

At the parser level, a FunC source file is a sequence of top-level directives, global declarations, constant declarations, and function definitions. The core expression language supports tuples, tensor-valued expressions, explicit type ascription, method syntax based on identifiers beginning with `.` or `~`, conditional expressions, and right-associative assignment.

Contents

1 Overview

The current FunC frontend has three relevant layers:

1. lexical analysis, implemented by the shared lexer;
2. keyword registration, implemented in `keywords.cpp`;
3. parsing and type-directed expression construction, implemented in `parse-func.cpp`.

The compiler also injects a small built-in prelude before parsing any user source. This prelude is defined in `builtins.cpp` and contains the primitive arithmetic operators, comparison operators, a few constants, basic slice and builder helpers, and several debugging and dynamic-dispatch helpers.

2 Lexical Structure

2.1 Whitespace and Comments

Whitespace separates tokens in the usual way. The current lexer recognizes:

- end-of-line comments beginning with `;;`;
- nested block comments delimited by `{-` and `-}`.

Strings are delimited either by ordinary double quotes or by triple double quotes. Triple-quoted strings may span multiple lines.

2.2 Token Boundaries

The FunC parser instantiates the lexer with `";,()[] ~."` as the active character specification. In practice, this means:

- `;`, `,`, `(`, `)`, `[`, and `]` are always standalone tokens;
- `.` and `~` act as left-active prefix characters so that tokens such as `.store_uint` and `~load_uint` are read as single identifiers;
- most other operators rely on ordinary token separation by whitespace or by the surrounding punctuation.

For this reason, source code normally writes infix operators with whitespace and writes block braces with ordinary separation, for example:

```
if (x) { return (); }
int y = x + 1;
```

2.3 Identifiers

The current lexer does not enforce a narrow identifier regular expression. Instead, any token that is not recognized as a number or registered keyword becomes an identifier.

As a result, valid identifiers in practice include names such as:

```
recv_internal
op::transfer
slice_empty?
dict_get?
.store_uint
~load_uint
```

The parser distinguishes three identifier classes:

- ordinary identifiers;
- *dot identifiers*, whose first character is `.`;
- *tilde identifiers*, whose first character is `~`.

Dot and tilde identifiers are used by method-call syntax; see Section 7.3.

Back-quoted tokens are also supported by the shared lexer. A token of the form `` `...` `` is first read as a raw token and then classified as if the quotes were not present. This is mainly useful when a name would otherwise be hard to tokenize.

2.4 Integer Literals

An integer literal is either:

- a decimal literal, such as `0`, `17`, or `-3`;
- a hexadecimal literal beginning with `0x`, such as `0xff`.

The parser converts integer literals into signed 257-bit values and rejects anything that does not fit.

2.5 String Literals

The parser accepts ordinary quoted strings and triple-quoted strings, each with an optional one-letter suffix. The suffix determines the semantic class of the literal:

`suffix`	meaning
`none`	slice constant built from the raw bytes of the literal
`s`	slice constant; the literal body is parsed as a hexadecimal bitstring
`a`	slice constant; the literal body is parsed as a standard address
`u`	integer constant obtained from the raw bytes of the literal
`h`	integer constant equal to the first 32 bits of SHA-256 of the literal
`H`	integer constant equal to the full 256-bit SHA-256 of the literal
`c`	integer constant equal to CRC32 of the literal

No other string suffix is recognized by the current parser.

2.6 Reserved Keywords and Operators

The current keyword table reserves the following words:

```
return  var  repeat  do  while  until  try  catch
if  ifnot  then  else  elseif  elseifnot
int  cell  slice  builder  cont  tuple  type  forall
extern  global  asm  impure  inline  inline_ref  auto_apply
method_id  operator  infix  infixl  infixr  const
#pragma  #include
```

The following multi-character operators are also reserved:

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

In addition, the single characters below are registered as keywords in their own right:

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

3 Types

3.1 Atomic Types

The parser recognizes the following atomic types:

```
int  cell  slice  builder  cont  tuple  type
```

In addition, `_` and `var` both denote a fresh type hole in type position.

3.2 Type Constructors

The parser uses the following grammar for type expressions:

```
TypeExpr  ::= AtomicType
            | "_"
            | "var"
            | "()"
            | "(" TypeExpr ")"
            | "(" TypeExpr "," TypeExpr { "," TypeExpr } ")"
            | "[]"
            | "[" TypeExpr { "," TypeExpr } "]"
            | TypeExpr "->" TypeExpr
```

The meaning of the constructors is:

- `()` is the unit tensor type;
- `(T1, T2, ..., Tn)` is a tensor type of width n;
- `[T1, T2, ..., Tn]` is a tuple type, still represented as a single runtime value;
- `A -> B` is a function type, parsed right-associatively.

The distinction between tensors and tuples is important. A tensor type denotes multiple stack values, while a tuple type denotes one TVM tuple value.

Examples:

```
int
(int, slice)
[int, cell]
int -> int
(int, slice) -> [int, slice]
```

3.3 Universally Quantified Type Variables

Source-level universal quantification appears only as a prefix of a function definition:

```
forall X, Y -> [X, Y] pair(X x, Y y) asm "PAIR";
```

Each bound identifier becomes a type variable visible in the function signature. The optional keyword `type` may appear before each such identifier, but it is not required:

```
forall type X, type Y -> ...
forall X, Y -> ...
```

4 Top-Level Source Structure

A source file is parsed as:

```
SourceItem ::= Pragma
             | Include
             | GlobalDecls
             | ConstDecls
             | FunctionDef

Source     ::= { SourceItem }
```

4.1 Pragmas

The directive syntax is:

```
Pragma ::= "#pragma" identifier ... ";"
```

The current parser recognizes these pragma names:

- `version`;
- `not-version`;
- `test-version-set`;
- `allow-post-modification`;
- `compute-asm-ltr`.

`version` and `not-version` accept semver-style constraints with the comparators `=`, `>`, `>=`, `<`, `<=`, and `^`. `test-version-set` replaces the compiler version string for test purposes.

4.2 Includes

The include syntax is:

```
Include ::= "#include" string-literal ";"
```

Included paths are resolved relative to the current file before being passed to the compiler's file-reading callback. The implementation suppresses repeated inclusion of the same real path.

4.3 Global Variables

The syntax for global variable declarations is:

```
GlobalDecls ::= "global" GlobalDecl { "," GlobalDecl } ";"
GlobalDecl  ::= [ TypeExpr ] identifier
```

If the type is omitted, or if the declaration begins with `_`, the parser creates a fresh type hole and lets later usage constrain it.

Examples:

```
global int seqno;
global cell state, slice config;
global _ scratch;
```

4.4 Constants

The syntax for constant declarations is:

```
ConstDecls ::= "const" ConstDecl { "," ConstDecl } ";"
ConstDecl  ::= [ "int" | "slice" ] identifier "=" ConstExpr
```

The current implementation accepts:

- integer literals;
- slice/string literals;
- compile-time integer expressions that can be reduced by the compiler into a single constant instruction.

Constants are therefore much more restrictive than ordinary expressions. Examples:

```
const int one = 1;
const slice prefix = "deadbeef"s;
const int answer = 6 * 7;
```

5 Function Definitions

5.1 Grammar

The surface grammar for function definitions is:

```
FunctionDef ::= [ ForallClause ]
                TypeExpr identifier FormalArgs
                [ "impure" ]
                [ "inline" | "inline_ref" ]
                [ MethodIdClause ]
                ( ";" | Block | AsmBody )

ForallClause ::= "forall" [ "type" ] identifier
                 { "," [ "type" ] identifier } "->"
```

```
FormalArgs   ::= "(" [ FormalArg { "," FormalArg } ] ")"
FormalArg    ::= "_"
               | [ TypeExpr ] [ identifier ]

MethodIdClause ::= "method_id"
                 | "method_id" "(" string-literal ")"
                 | "method_id" "(" integer-literal ")"
```

The order of the optional modifiers is significant because it is hard-coded in the parser:

1. `impure`;
2. one of `inline` or `inline_ref`;
3. `method_id`.

5.2 Forward Declarations

If a function definition ends with `;`, it is a declaration without a body:

```
int helper(int x);
```

Repeated declarations must unify with the already known function type.

5.3 Ordinary Bodies

An ordinary function body is a block:

```
int add(int x, int y) {
  return x + y;
}
```

If control can fall off the end of the function body, the parser inserts an implicit `return ();` and requires the declared return type to unify with unit.

5.4 Assembler Bodies

An assembler-bodied function uses the syntax:

```
AsmBody ::= "asm"
            [ "(" [ ArgOrder ] [ "->" RetOrder ] ")" ]
            string-literal { string-literal }
            ";"
```

The optional permutation clause has two roles:

- `ArgOrder` lists formal parameter names in the order expected by the assembly implementation;
- `RetOrder` lists integer indices `0 .. width-1` describing how multiple returned stack values are reordered.

Example:

```
forall X -> (X, tuple) list_next(tuple list) asm( -> 1 0) "UNCONS";
```

The parser imposes two important static restrictions on `asm` functions:

- at most 16 formal arguments;
- a fixed-width return type and fixed-width argument types.

5.5 Method Identifiers

If `method_id` is present without an explicit argument, the compiler uses the function name. If its argument is a string, the compiler computes `crc16(name) | 0x10000`. If its argument is an integer, that integer is used directly.

Examples:

```
int seqno() method_id { ... }
int get_public_key() method_id("pubkey") { ... }
int my_getter() method_id(123456) { ... }
```

6 Statements

6.1 Grammar

Function bodies are made of the following statements:

```
Stmt ::= "return" Expr ";"
       | Block
       | ";"
       | "repeat" Expr Block
       | "while" Expr Block
       | "do" Block "until" Expr
       | "try" Block "catch" Expr Block
       | IfStmt
       | Expr ";"

Block ::= "{" { Stmt } "}"

IfStmt ::= ("if" | "ifnot") Expr Block
           [ "else" Block
           | ("elseif" | "elseifnot") Expr Block [ ... ] ]
```

6.2 Return Statements

The current parser does *not* support a bare `return;`. Unit return must be written explicitly:

```
return ();
```

6.3 Condition and Loop Types

The parser requires the following expressions to unify with `int` and to produce a single stack value:

- `if` and `ifnot` conditions;
- `elseif` and `elseifnot` conditions;
- `while` conditions;

- `do ... until` conditions;
- `repeat` iteration counts.

6.4 Try/Catch

A `catch` clause binds an expression pattern whose type must unify with a two-component tensor `(X, int)`. In normal source code this is usually written as two variables or holes:

```
try {
  ...
} catch (exc_arg, exc_code) {
  ...
}
```

7 Expressions

7.1 Primary Forms

The highest-precedence expression forms are:

```
Primary ::= "(" ")"
          | "(" Expr ")"
          | "(" Expr "," Expr { "," Expr } ")"
          | "[" "]"
          | "[" Expr { "," Expr } "]"
          | integer-literal
          | string-literal
          | "_"
          | "var"
          | "int" | "cell" | "slice" | "builder" | "cont" | "tuple" | "type"
          | identifier
```

Their meaning is:

- `(e1, e2, ..., en)` constructs a tensor-valued expression;
- `[e1, e2, ..., en]` constructs a TVM tuple value;

- `_` constructs a hole pattern suitable for destructuring on the left of assignment;
- a type keyword or type identifier in expression position introduces an explicit type application, described below.

7.2 Application and Explicit Type Application

After a primary expression, the parser repeatedly accepts another primary expression. The resulting form means one of two things:

1. if the left expression is a type expression, the form is an explicit type application used for typed patterns and declarations;
2. otherwise, it is ordinary function application.

Examples:

```
f(x)
f(x, y)
int x = 1;
var msg_seqno = cs~load_uint(32);
(int, slice) (n, s) = (1, "ab"s);
```

This rule is the reason local variable introduction is syntax-driven. A fresh local variable is created by writing it under an explicit type pattern such as `int x`, `var x`, or a tuple/tensor pattern containing such names. A bare unknown name on the left of `=` is *not* the declaration form implemented by the current parser.

7.3 Method Calls

After an expression, the parser accepts any number of special identifiers whose text begins with `.` or `~`. The general form is:

```
receiver.special_identifier argument
```

In source code the special identifier is normally attached directly to the receiver:

```
cs~load_uint(32)
begin_cell().store_uint(x, 32).end_cell()
```

The difference between the two prefixes is:

- `.name` requires the receiver to be an rvalue;
- `~name` requires the receiver to be an lvalue and is compiled through a "modify-and-return" pattern.

If a `~name` symbol is not defined but the corresponding plain `name` symbol exists, the parser falls back to the plain symbol.

7.4 Operator Precedence

The parser implements the following precedence levels, from highest to lowest:

level	forms
100	primary expressions
90	repeated application and explicit type application
80	repeated method syntax using identifiers beginning with dot or tilde
75	prefix tilde
30	multiplicative operators, rounded division/modulus operators, and bitwise and
20	prefix minus, then additive operators, bitwise or, and bitwise xor
17	shift operators
15	one comparison operator
13	conditional operator
10	assignment and compound assignment operators

Important implementation details:

- comparison operators are not chainable in the current parser;
- assignment is right-associative;

- prefix - is parsed at level 20, so its operand is a level-30 expression rather than a level-75 expression;
- negative numeric literals such as -1 are usually lexed as single number tokens, which avoids ambiguity for the common literal case.

7.5 Assignments and Compound Assignments

The assignment grammar is:

```
AssignExpr ::= LValue "=" Expr
             | LValue "+=" Expr
             | LValue "-=" Expr
             | LValue "*=" Expr
             | LValue "/=" Expr
             | LValue "~/=" Expr
             | LValue "^/=" Expr
             | LValue "%=" Expr
             | LValue "~%=" Expr
             | LValue "^%=" Expr
             | LValue "<<=" Expr
             | LValue ">>=" Expr
             | LValue "~>>=" Expr
             | LValue "^>>=" Expr
             | LValue "&=" Expr
             | LValue "|=" Expr
             | LValue "^=" Expr
```

Assignments are expressions, not separate statements. They therefore may appear wherever an rvalue expression is allowed, although they are most often used as statement bodies.

8 Compiler Prelude and Built-ins

Before parsing user code, the compiler defines a small built-in prelude. The surface language uses many of these names indirectly via operator syntax or method syntax.

8.1 Surface Operators Backed by Built-in Symbols

The following built-in function names are the parser's canonical targets for the surface operators:

```
_+_   _-_   -_   _*_   _/_   _~/_   _^/_   _%_   _~%_   _^%_   _/%_
_<<_  _>>_  _~>>_  _^>>_  _&_  _|_  _^_  ~_
_==_  _!=_  _<_  _>_  _<=_  _>=_  _<=>_
```

Compound assignments use a parallel family of internal names:

```
^_+=_   ^_-=_   ^_*=_   ^_/=_   ^_~/=_   ^_^/=_
^_%=_   ^_~%=_  ^_^%=_  ^_<<=_  ^_>>=_   ^_~>>=_  ^_^>>=_
^_&=_   ^_|=_   ^_^=_
```

8.2 Named Built-ins

The current named built-ins are grouped as follows.

Arithmetic helpers:

```
divmod  ~divmod  moddiv  ~moddiv
muldiv  muldivr  muldivc  muldivmod
```

Constants and null checks:

```
true  false  nil  Nil  null?
```

Exceptions:

```
throw  throw_if  throw_unless
throw_arg  throw_arg_if  throw_arg_unless
```

Slice and builder primitives:

```
load_int  load_uint  preload_int  preload_uint
store_int  store_uint  ~store_int  ~store_uint
load_bits  preload_bits
```

Tuple access:

```
int_at  cell_at  slice_at  tuple_at  at
```

No-op and debugging helpers:

```
touch  ~touch  touch2  ~touch2  ~dump  ~strdump
```

Dynamic calls:

```
run_method0  run_method1  run_method2  run_method3
```

8.3 Notes

The nearby source code contains a commented-out definition for `null`, but `null` itself is *not* currently injected by `builtins.cpp`. Only `null?` is part of the compiler prelude.

9 Reserved but Currently Unused Keywords

The following tokens are reserved by `keywords.cpp` but are not consumed by the current FunC parser:

```
then
extern
auto_apply
operator
infix
infixl
infixr
```

They therefore cannot be used as ordinary identifiers, but they also have no surface syntax today.

A Summary Grammar

This appendix collects the main grammar fragments in one place.

```
Source     ::= { SourceItem }
SourceItem ::= Pragma | Include | GlobalDecls | ConstDecls | FunctionDef

Pragma     ::= "#pragma" identifier ... ";"
Include    ::= "#include" string-literal ";"

GlobalDecls ::= "global" GlobalDecl { "," GlobalDecl } ";"
GlobalDecl  ::= [ TypeExpr ] identifier

ConstDecls ::= "const" ConstDecl { "," ConstDecl } ";"
ConstDecl  ::= [ "int" | "slice" ] identifier "=" ConstExpr
```

```
FunctionDef ::= [ ForallClause ]
                TypeExpr identifier FormalArgs
                [ "impure" ]
                [ "inline" | "inline_ref" ]
                [ MethodIdClause ]
                ( ";" | Block | AsmBody )

ForallClause ::= "forall" [ "type" ] identifier
                 { "," [ "type" ] identifier } "->"

FormalArgs ::= "(" [ FormalArg { "," FormalArg } ] ")"
FormalArg  ::= "_"
             | [ TypeExpr ] [ identifier ]

MethodIdClause ::= "method_id"
                 | "method_id" "(" string-literal ")"
                 | "method_id" "(" integer-literal ")"

AsmBody ::= "asm"
            [ "(" [ ArgOrder ] [ "->" RetOrder ] ")" ]
            string-literal { string-literal }
            ";"

Block ::= "{" { Stmt } "}"

Stmt ::= "return" Expr ";"
       | Block
       | ";"
       | "repeat" Expr Block
       | "while" Expr Block
       | "do" Block "until" Expr
       | "try" Block "catch" Expr Block
       | IfStmt
       | Expr ";"

IfStmt ::= ("if" | "ifnot") Expr Block
           [ "else" Block
           | ("elseif" | "elseifnot") Expr Block [ ... ] ]
```

```
TypeExpr ::= AtomicType
           | "_"
           | "var"
           | "()"
           | "(" TypeExpr ")"
           | "(" TypeExpr "," TypeExpr { "," TypeExpr } ")"
           | "[]"
           | "[" TypeExpr { "," TypeExpr } "]"
           | TypeExpr "->" TypeExpr
```

B Expression Precedence Summary

```
100  primary expressions
 90  repeated application / explicit type application
 80  repeated .method and ~method application
 75  prefix ~
 30  *  /  ~/  ^/  %  ~%  ^%  /%  &
 20  prefix - ; then +  -  |  ^
 17  <<  >>  ~>>  ^>>
 15  ==  !=  <  >  <=  >=  <=>
 13  ?:
 10  =  +=  -=  *=  /=  ~/=  ^/=  %=  ~%=  ^%=  <<=  >>=  ~>>=  ^>>=  &=  |=  ^=
```