🔥 Flare Programming Language

Flare is a dynamic, expression-oriented programming language featuring first-class functions with closures, explicit opt-in mutability, actor-based concurrency, and configurable sandboxing. It is primarily designed to be suitable for embedding as a scripting language in larger applications, but can also be used as a general-purpose programming language.

Language Reference

Introduction

This is the language reference for the Flare programming language. It describes the structure of a Flare program, the grammar and informal semantics of various language features, and rationales for certain design decisions. Some base familiarity with programming language design is assumed.

This is not a formal specification, and it may be incomplete in some areas. It does not document the core library, nor any implementation details of the main Flare implementation. It is primarily meant to be a reference for Flare programmers who need to look up the details of particular language features.

Grammar Notation

The Flare language grammar will be given throughout this reference when relevant. It is specified in a variation of the Extended Backus-Naur Form (EBNF). EBNF consists of a series of grammar production rules (called non-terminals) built on fundamental lexical symbols (called terminals). The meaning of the EBNF variant used in this reference is given here.

A production rule is defined as follows, using the `=` operator:

```
rule-name = ...
```

The right-hand side of the rule can refer to other rules (non-terminals) or symbols (terminals). Additionally, a number of operators are used to build more complex rules. These are, from highest to lowest precedence:

- `"A"`: Means the literal string or character `A` in terms of Unicode scalars. Some literals may have escape sequences (e.g. `"\\"` which means the `\` character, or `"\""` which means the `"` character).
- `U+NNNNNN`: Means the specific Unicode scalar indicated by the hexadecimal `NNNNNN` value.
- `"A" .. "B"`: Constructs a Unicode scalar range from the character `A` to `B`, inclusive.
- `A ... | B ...`: The pipe operator constructs an alternation. This means that either `A ...` or `B ...` are allowed.
- `[ A ... ]`: Square brackets construct an option. This means that `A ...` may or may not appear.
- `{ A ... }`: Curly braces construct a zero-or-more repetition. This means that zero or multiple occurrences of `A ...` may appear.
- `< A ... >`: Angle brackets construct a one-or-more repetition. This means that at least one occurrence of `A ...` must appear, and possibly more.
- `A ... * N`: The asterisk indicates that `A ...` must appear `N` times.
- `( A ... )`: Parentheses perform simple grouping to resolve precedence issues.
- `? ... ?`: Specifies a special sequence. The meaning of the sequence is explicitly given in the `...` part.

Lexical Structure

A Flare source file consists of a series of Unicode scalars encoded as UTF-8. These scalars are tokenized according to the lexical grammar, resulting in a series of tokens which are then processed when parsing the language proper. If a sequence of scalars matches multiple lexical production rules, lexical analysis should always form the longest possible token. For example, the sequence `>>` should result in the `>>` operator rather than two `>` operators.

The lexical grammar is given in this section. The syntactic grammar is given throughout this reference in relevant sections.

```
input = [ shebang-line ] { token }
token = white-space |
        comment |
        operator |
        delimiter |
        keyword |
        identifier |
        literal
```

`input` is the top-level production for lexical analysis.

Shebang Line

A Flare source file can start with a special shebang line that indicates the interpreter to use for running it. This allows the file to be invoked in a similar fashion to a regular program in Unix-like shells.

```
shebang-line = "#!" { ? any scalar except new-line ? }
```

The shebang line is only allowed to appear as the very first line in a source file and is discarded during lexical analysis.

White Space

Flare deliberately supports a very limited notion of white space: The space character, the horizontal tab character, as well as the line feed and carriage return characters.

```
white-space = blank |
              new-line
blank = U+000009 |
        U+000020
new-line = U+00000A |
           U+00000D
```

White space is discarded during lexical analysis, and so production rules in the syntactic grammar will not consider it, with the implication being that white space is allowed anywhere between all other grammar elements.

Comments

Comments in Flare start with `'` and run until the end of the line.

```
comment = "'" { ? any scalar except new-line ? }
```

Comments are treated similarly to white space in that they are allowed between all other grammar elements and so are not explicitly considered in production rules in the syntactic grammar.

Operators

Flare has support for custom operators where the precedence is based on the first character in the operator, and associativity is either left-to-right (for infix operators) or right-to-left (for prefix operators). For example, `+` and `+>` both have the same precedence and left-to-right associativity in a binary expression context, while in a prefix expression context, they both have right-to-left associativity.

Certain operators are defined to override these rules, with special precedence and associativity; for example, the relational operators `>` and `>=` both have lower precedence than any other operator starting with `>` (e.g. the right shift operator `>>`). This is made explicit in the syntactic grammar.

```
operator = special-operator |
           custom-operator
custom-operator = custom-multiplicative-operator |
                  custom-additive-operator |
                  custom-shift-operator |
                  custom-bitwise-operator
special-operator = "!=" |
                   "<" |
                   "<=" |
                   "=" |
                   "==" |
                   ">" |
                   ">="
operator-part = "%" |
                "&" |
                "*" |
                "+" |
                "-" |
                "/" |
                "<" |
                ">" |
                "^" |
                "|" |
                "~"
custom-bitwise-operator = ( "&" | "^" | "|" ) { operator-part }
custom-shift-operator = ( "<" | ">" ) { operator-part }
custom-additive-operator = ( "+" | "-" | "~" ) { operator-part }
custom-multiplicative-operator = ( "%" | "*" | "/" ) { operator-part }
```

Delimiters

These are various delimiters used in the grammar.

```
delimiter = "#" |
            "(" |
            ")" |
            "," |
            "." |
            ".." |
            "->" |
            ":" |
            "::" |
            ";" |
            "<-" |
            "=>" |
            "?" |
            "@" |
            "[" |
            "]" |
            "{" |
            "}"
```

Keywords

Keywords are special identifiers used to direct syntactic analysis. They cannot be used as identifiers.

Some keywords are reserved for future expansion of the language.

```
keyword = used-keyword |
          reserved-keyword
used-keyword = "and" |
               "as" |
               "assert" |
               "break" |
               "catch" |
               "cond" |
               "const" |
               "else" |
               "exc" |
               "extern" |
               "fn" |
               "for" |
               "freeze" |
               "if" |
               "in" |
               "let" |
               "loop" |
               "match" |
               "mod" |
               "mut" |
               "not" |
               "or" |
               "priv" |
               "pub" |
               "raise" |
               "rec" |
               "recv" |
               "return" |
               "test" |
               "use" |
               "while"
reserved-keyword = "asm" |
                   "async" |
                   "await" |
                   "do" |
                   "goto" |
                   "macro" |
                   "pragma" |
                   "quote" |
                   "super" |
                   "try" |
                   "unquote" |
                   "yield"
```

Identifiers

Identifiers are used for naming modules, functions, variables, etc. There are separate rules for module and value identifiers.

```
identifier = module-identifier |
             value-identifier
```

Module Identifiers

Module identifiers are sequences of alphanumeric characters, always starting with an uppercase alphabetical character.

```
module-identifier = "A" .. "Z" { "0" .. "9" | "A" .. "Z" | "a" .. "z" }
```

Value Identifiers

Value identifiers (used for functions and variables) are sequences of lowercase alphanumeric and/or underscore (`_`) characters.

```
value-identifier = ( "_" | "a" .. "z" ) { "0" .. "9" | "_" | "a" .. "z" }
```

Literals

Literals are the fundamental building blocks of Flare expressions. These are the most basic values that are expressed without the need for evaluation and are therefore constants.

```
literal = nil-literal |
          boolean-literal |
          atom-literal |
          integer-literal |
          real-literal |
          string-literal
```

Nil Literals

A nil literal simply has the value `nil`, typically indicating the absence of any other kind of value.

```
nil-literal = "nil"
```

Boolean Literals

A Boolean literal has either the value `true` or `false`.

```
boolean-literal = "true" | "false"
```

Atom Literals

An atom literal is simply a name - essentially an identifier in value form.

```
atom-literal = ":" identifier
```

`:foo`, `:Foo`, and `:$foo` are all valid atoms.

Integer Literals

An integer literal is a number with a base of 2, 8, 10, or 16. A literal prefixed with `0b` denotes a binary literal, `0o` denotes an octal literal, and `0x` denotes a hexadecimal literal. If no prefix is present, the literal is decimal.

```
integer-literal = < "0" .. "9" > |
                  "0" ( "B" | "b" ) < "0" .. "1" > |
                  "0" ( "O" | "o" ) < "0" .. "7" > |
                  "0" ( "X" | "x" ) < "0" .. "9" | "A" .. "F" | "a" .. "f" >
```

Real Literals

A real literal is an IEEE 754 `binary64` floating point number consisting of an integral part, a fractional part, and an optional exponent part.

```
real-literal = real-part "." real-part [ real-exponent ]
real-part = < "0" .. "9" >
real-exponent = ( "E" | "e" ) [ "+" | "-" ] real-part
```

String Literals

A string literal is a series of Unicode scalars encoded as UTF-8.

```
string-literal = "\"" { ? any scalar except "\"" and "\\" ? | string-escape-sequence } "\""
string-escape-sequence = "\\" ( string-escape-code | string-escape-unicode )
string-escape-code = "0" | ( "N" | "n" ) | ( "R" | "r" ) | ( "T" | "t" ) | "\"" | "\\"
string-escape-unicode = ( "U" | "u" ) ( "0" .. "9" | "A" .. "F" | "a" .. "f" ) * 6
```

An escape sequence can be used to form a special character as shown in the following table:

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE SCALAR
`\0`	Null	U+000000
`\t`, `\T`	Horizontal tab	U+000009
`\n`, `\N`	Line feed	U+00000A
`\r`, `\R`	Carriage return	U+00000D
`\"`	Double quote	U+000022
`\\`	Backslash	U+00005C
`\uNNNNNN`, `\UNNNNNN`	Any scalar	U+NNNNNN

Basic Concepts

This section provides a brief introduction to some of the basic concepts of the Flare language.

Modules and Declarations

A Flare program is a collection of modules, each containing various kinds of declarations such as constants, functions, tests, etc. The module is the primary means of code organization, enabling separation of concerns and code reuse.

Each source file is a module. A header at the beginning of the file specifies the full module name, with declarations coming after it. For example:

```
mod A::B;

fn foo() {
    42;
}
```

Though not strictly required, by convention, a module's full name should correspond to its location on the file system. The above module would be located at `A/B.fl`. The module loader in the runtime system follows this convention when looking for modules.

Modules can import other modules:

```
mod A::B;

use C::D;

fn foo() {
    bar();
}
```

Here, `bar` is imported from the `C::D` module and made available in `A::B`.

When a Flare program starts, the main module is loaded. Any `use` declarations in that module will trigger other modules to be loaded, which may in turn have their own `use` declarations, and so on. Circular `use` declarations (such as a module `A` declaring `use B` while module `B` declares `use A`) are considered errors.

Modules can contain a variety of declarations. Here are some examples:

```
mod A::B;

const x = 42;

pub fn foo() {
    x;
}

extern bar(x, y, z);

test qux {
    assert true;
}
```

Visibility specifiers can be used on all declarations except `use` and `test` declarations. The default is `priv` if none is specified. Visibility is strictly enforced; there is no way to get at the private declarations in a module - not even through introspection. This is an important aspect of Flare's sandboxing capabilities.

Values and Types

Being a dynamic language, Flare does not assign types to storage locations (such as variables and fields). Instead, values themselves have a type, and type errors are detected at run time. Additionally, there are no type declarations (such as classes in other languages); rather, values can define their own behavior.

(Future versions of Flare may support optional type annotations that can be used for static analysis.)

These are the types that Flare supports:

- Nil: Has the singleton value `nil`. It is typically used to indicate the lack of any other meaningful value (similar to `null` in other languages).
- Boolean: Has the value `true` or `false`.
- Atom: A named constant with an indeterminate value. Comparing two atoms is much faster than comparing two strings, so atoms are typically used as markers or tags. Examples: `:foo`, `Bar`, `:$baz`.
- Integer: An arbitrarily large integer. Examples: `42`, `-42`, `123456789123456789123456789123456789`.
- Real: A real number represented as IEEE 754 `binary64` floating point. Examples: `12.34`, `-42.0`, `1.2e-3`.
- String: An immutable sequence of Unicode scalars encoded as UTF-8. Examples: `"foo"`, `"\tbar"`, `"b\u000061z"`.
- Module: A loaded module. Typically referred to by a module path expression such as `Core::IO`. Module values are functionally similar to records containing only immutable fields.
- Function: A callable function as a value. Carries information such as its originating module, name (if any), and arity. Example: `fn(x) => x * x`.
- Record: Similar to objects in other languages, a record is a collection of fields (which can individually be mutable or immutable), optionally carrying a name. Records can define overloaded behaviors such as operators, hashing, pattern views, etc. Examples: `rec { x = 42, y = :foo }`, `rec NamedRecord { field = :foo, mut mutable_field = :bar }`.
- Exception: Exceptions behave similarly to records, but with the difference that they must always carry a name, and they are allowed to be used as the operand of the `raise` expression. Example: `exc MyError { message = "error!" }`.
- Tuple: An immutable array of values with a fixed length of at least 2 elements. Examples: `(42, :foo)`, `(:ok, rec { x = 42 }, "y")`.
- Array: An array of values that may be mutable or immutable. Mutable arrays are expandable. Examples: `[]`, `[1, 2, 3]`, `mut [1, :foo, "bar"]`.
- Set: A hashed set of values that may be mutable or immutable, and does not allow duplicate values. Mutable sets are expandable. Sets have no ordering guarantees. Examples: `#{}`, `#{"x", :y, z}`, `mut #{:foo, :bar, :baz}`.
- Map: A hashed map of key-value pairs that may be mutable or immutable, and does not allow duplicate keys. Mutable maps are expandable. Maps have no ordering guarantees. Examples: `#[]`, `#["foo" : 42, :bar : nil]`, `mut #[:foo : 42]`.
- Agent: A lightweight thread of execution. Agent values are obtained through library functions such as `Core::Agent::self` and `Core::Agent::spawn`.
- Reference: A globally unique, opaque value. A reference value can only be obtained by calling `Core::new_ref` or by

obtaining it from another agent. References are backed by at least 20 bytes of randomly generated data. These properties make references practically unforgeable and non-repeatable within a program's lifetime.

- Pointer: A native pointer. These are usually obtained by calling into various functions provided by the runtime system. As Flare has no syntax for dereferencing pointers, pointer values are effectively opaque to Flare code.
- Handle: A handle to some kind of native resource (raw memory, file descriptor, socket, etc). As with pointers, these values are opaque to Flare code. Unlike pointers, handles are tracked by the garbage collector and can run cleanup code when they are no longer reachable.

All values in Flare are, conceptually, passed by reference. That said, for immutable values (integers, strings, tuples, etc), the implementation is free to pass them by value as this is not observable by Flare code.

Mutability and Freezing

While storage locations do not have types in Flare, they do have mutability information. Variables, record fields, arrays, sets, and maps can all be mutable, but only if marked explicitly as such. A few examples:

```
let x = :foo;
x = :bar; ' Panic: Trying to modify an immutable variable.

let mut x = :foo;
x = :bar; ' OK.

let r = rec { x = :foo };
r.x = :bar; ' Panic: Trying to modify an immutable field.

let r = rec { mut x = :foo };
r.x = :bar; ' OK.

let a = [1, 2, 3];
a[0] = 42; ' Panic: Trying to modify an immutable array.

let a = mut [1, 2, 3];
a[0] = 42; ' OK.

let s = #{1, 2, 3};
s->mut_add(4); ' Panic: Trying to modify an immutable set.

let s = mut #{1, 2, 3};
s->mut_add(4); ' OK.

let m = #["foo" : "bar", "baz" : "qux"];
m["foo"] = "xyzzy"; ' Panic: Trying to modify an immutable map.

let m = mut #["foo" : "bar", "baz" : "qux"];
m["foo"] = "xyzzy"; ' OK.
```

Since immutability is generally preferable, it is possible to freeze mutable storage locations once they no longer need to be mutable. This is done with the `freeze` expression:

```
let mut x = :foo;
x = :bar;
freeze x;
x = :baz; ' Panic: Trying to modify an immutable variable.

let r = rec { mut x = :foo };
r.x = :bar;
freeze r.x;
r.x = :baz; ' Panic: Trying to modify an immutable variable.
```

For arrays, sets, and maps, `freeze var` would be ambiguous as it could mean "freeze the variable `var`" or "freeze the collection residing in variable `var`". Therefore, the syntax for freezing these types is `freeze in`:

```
let a = mut [:foo, :bar];
a[0] = :baz;
freeze in a;
a[0] = :qux; ' Panic: Trying to modify an immutable array.

let s = mut #{:foo, :bar};
s->mut_add(:baz);
freeze in s;
s->mut_add(:qux); ' Panic: Trying to modify an immutable set.

let m = mut #["foo" : "bar"];
m["foo"] = "baz";
freeze in m;
m["foo"] = "qux"; ' Panic: Trying to modify an immutable map.
```

Just as storage locations that are initially declared immutable cannot be turned mutable, any location that is frozen can never be turned mutable again. Immutability is strictly enforced; it cannot be bypassed, even with introspection.

Pattern Matching

Pattern matching is used throughout Flare to simultaneously perform case analysis and destructuring on values. The `match`, `for`, `catch`, `let` and `use` constructs all make use of pattern matching.

Here are some examples of pattern matching using `let` and `match`:

```
' Bind `a` to the value `42`.
let a = 42;

' Bind `b` to the value of `a`, with `b` being mutable.
let mut b = a;

' Assert that `a` equals `42`.
let 42 = a;

' Bind `c` to the tuple `(a, b)`.
let c = (a, b);

' Bind `d` to the first element of `c` and discard the second element.
let (d, _) = c;

' Assert that `foo` returns a slice and that the first three elements are `1`,
' `2`, `3`. Then bind `e` to the slice itself.
let [1, 2, 3] as e = foo();

match bar() {
    ' Match the `"foo"` string constant.
    "foo" => println("foo string");

    ' Match a record containing the fields `x` and `y`. Bind `x1` and `y1` to
    ' those fields, respectively.
    rec { x = x1, y = y1 } => println("x = {}, y = {}", x1, y1);

    ' Match an exception containing the field `message` and bind `msg` to it.
    exc SomeError { message = msg } => println("error: {}", msg);

    ' Match the `nil` constant.
    nil => println("nil");

    ' Match an integer above `42`. Bind `i` to the value.
    i if is_integer(i) and i > 42 => println("integer over 42");

    ' Fallback case.
    _ => println("something else");
};
```

A failed match in a `let` statement will result in a panic, as will a failed match in the `for` and `use` expressions. A `match` expression will only panic if no cases match and there is no fallback case.

Patterns allow recursively destructuring arbitrarily complex values and data structures.

Error Handling

Flare's error handling model is a middle ground between exception handling and traditional error codes. Exception values can be returned with the `raise` expression and can be propagated at the call site using the `?` operator, or caught using the `catch` clause. An example:

```
fn foo() {
    raise exc MyError { message = "error occurred" };
}

fn bar() {
    foo()? catch {
        exc MyError { message = msg } => println("error: {}", msg);
        ex => println("some other error");
    };
}
```

It is important to understand that the `?` operator can only be applied to call expressions. This means that call sites are the only place where exceptions can be caught. There is no such thing as a general `try`/`catch` block as in other languages.

If a function raises an exception and the caller does not either propagate it with `?` or handle it with `catch`, a panic will occur. Some examples:

```
fn foo() {
    raise exc MyError { message = "error occurred" };
}

fn bar() {
    foo()?; ' OK; propagatates exception to the caller of `bar`.

    foo()? catch {
        _ => nil; ' OK; exception was handled.
    };

    foo(); ' Panic: Unobserved 'MyError' exception.

    foo()? catch {
        exc OtherError { } => nil; ' Panic: Unhandled 'MyError' exception.
    }
}
```

Agents and Message Passing

Where other languages might use operating system threads for concurrency, Flare uses so-called agents. These are lightweight threads (also known as green threads) which are preemptively scheduled by the runtime system. They have various important properties such as being fast to create and destroy, having low memory footprint, having no shared state, and having the ability to monitor each other.

Agents can be spawned with the `Core::Agent::spawn` function:

```
use Core::Agent;

pub fn main(_args, _env) {
    let agent = spawn(fn(x, y, z) => {
        println(x); ' `"abc"`
        println(y); ' `:foo`
        println(z); ' `42`
    }, ["abc", :foo, 42]);
}
```

Agents communicate by sending messages to each other. Sending is done with the `<-` operator, while receiving is done with the `recv` expression. Normally, `recv` will block until a message in the current agent's message queue matches one of the message patterns. An `else` clause can be given to make `recv` proceed immediately if no matching message is pending. Example:

```
use Core::Agent;
use Core::Time;

pub fn main(_args, _env) {
    let calc = spawn(fn() => {
        while true {
            recv {
                (sender, :add, x, y) =>
                    sender <- (:result, x + y);
                :exit =>
                    break;
            } else {
                ' Only process messages every second.
                sleep(seconds(1));
            };
        };
    }, []);

    calc <- (self(), :add, 21, 21);

    recv {
        (:result, z) => {
            println(z); ' 42
            calc <- :exit;
        };
    };
}
```

Messages are processed by `recv` expressions in the order they were sent to an agent. If a received message is not matched by any message pattern in a `recv` expression, it will be put back in the message queue to be tried again on the next `recv` invocation. The current `recv` expression then tries the next message in the queue, and so on until a message has been matched, or until the whole queue has been tried and there is an `else` clause.

Since there is no shared state in Flare, sending a message to another agent requires performing a deep copy of the entire object graph the message contains. This deep copy will preserve relationships (i.e. two objects referring to the same object will do so in the copied object graph as well). Mutability will also be mirrored in the copied object graph. Certain special values (like handles) may be invalidated in the target agent after a copy; this depends on what code the handle was obtained from and whether that code can deal with such a copy occurring.

Garbage Collection

Memory management in Flare is fully automatic through garbage collection. Since there is no shared state, garbage collection is an entirely agent-local operation; garbage collection in one agent will never need to pause another agent.

A garbage collection works by tracing all values that are still reachable by code running in an agent, and then freeing all values that are not. Garbage collections are triggered when an agent runs out of memory to allocate some value in, or if the `Core::GC::collect` function is called manually. If after a garbage collection there is still not enough memory, the runtime system will allocate memory from the operating system. Finally, if this fails, the agent will panic - there is no way to catch an out-of-memory error within an agent.

While the vast majority of values are allocated within an agent's local heap, some values are, by necessity, global. Agent values are a good example of this: An agent can be reachable from multiple other agents, and copying an agent obviously does not make sense, so agent values use atomic reference counting rather than traditional tracing garbage collection.

The Flare language has no built-in support for manual memory management. The core library has escape hatches that allow it, but these are only accessible from agents with the right permission reference value. This design ensures that sandboxed agents cannot perform unsafe operations that could jeopardize the integrity of the whole process.

Startup and Termination

A normal Flare program will begin in the special `main` function:

```
pub fn main(args, env) {
    nil;
}
```

This function is invoked by the runtime system after the runtime has been initialized. The `args` parameter contains an immutable array of command line arguments, while the `env` parameter contains an immutable map of environment variables. These are both passed by the host environment. Any value returned by this function is ignored; halting with a non-zero status code can be done with the `Core::halt` function.

The `main` function will be executed in the root agent - a special agent created by the runtime system on startup. This agent has all capabilities initially made available by the runtime system, and can freely provide those capabilities to child agents as needed. The root agent is thus quite similar to the `init` process in a Unix system.

The runtime system terminates once the `main` function returns, or if some external signal has triggered a runtime shutdown. Termination will not wait for all agents to shut down cleanly; the only resources that will be given a chance to run cleanup code are handle values. So, a program that wishes to terminate cleanly should arrange for all agents to exit cooperatively before returning from `main`.

Program Structure

```
program = { attribute } "mod" module-path ";" { declaration }
module-path = module-identifier { "::" module-identifier }
```

Attributes

```
attribute = "@" "[" value-identifier "=" literal "]"
```

Declarations

```
declaration = { attribute } ( use-declaration |
                              test-declaration |
                              constant-declaration |
                              function-declaration |
                              extern-declaration
visibility = "priv" | "pub"
```

Use Declaration

```
use-declaration = "use" module-path [ "as" module-identifier ] ";"
```

Test Declaration

```
test-declaration = "test" value-identifier block-expression
```

Constant Declaration

```
constant-declaration = [ visibility ] "const" value-identifier "=" expression ";"
```

Function Declaration

```
function-declaration = [ visibility ] "fn" value-identifier function-parameter-list block-expression
function-parameter-list = "(" [ function-parameter { "," function-parameter } [ variadic-function-parameter ]
] ")"
function-parameter = { attribute } value-identifier
variadic-function-parameter = "," ".." function-parameter
```

Extern Declaration

```
extern-declaration = [ visibility ] "extern" value-identifier function-parameter-list ";"
```

Statements

```
statement = let-statement |
            use-statement |
            expression-statement
```

Let Statement

```
let-statement = "let" pattern "=" expression ";"
```

Use Statement

```
use-statement = "use" pattern "=" expression ";"
```

Expression Statements

```
expression-statement = expression ";"
```

Expressions

```
expression = send-expression
```

Prefix Expressions

```
prefix-expression = unary-expression |
                    assert-expression |
                    primary-expression |
                    postfix-expression
```

Unary Expression

```
unary-expression = ( custom-operator | "not" ) prefix-expression
```

Assert Expression

```
assert-expression = "assert" prefix-expression
```

Infix Expressions

Send Expression

```
send-expression = assign-expression { "<-" send-expression }
```

Assign Expression

```
assign-expression = logical-expression { "=" assign-expression }
```

Logical Expression

```
logical-expression = relational-expression { ( "and" | "or" ) relational-expression }
```

Relational Expression

```
relational-expression = bitwise-expression { ( "!=" | "<" | "<=" | "==" | ">" | ">=" ) bitwise-expression }
```

Bitwise Expression

```
bitwise-expression = shift-expression { custom-bitwise-operator shift-expression }
```

Shift Expression

```
shift-expression = additive-expression { custom-shift-operator additive-expression }
```

Additive Expression

```
additive-expression = multiplicative-expression { custom-additive-operator multiplicative-expression }
```

Multiplicative Expression

```
multiplicative-expression = prefix-expression { custom-multiplicative-operator prefix-expression }
```

Postfix Expressions

```
postfix-expression = primary-expression { call-expression |
                                          method-call-expression |
                                          index-expression |
                                          field-access-expression }
```

Call Expression

```
call-expression = argument-list [ call-try ]
argument-list = "(" [ argument { "," argument } [ variadic-argument ] ] ")"
argument = expression
variadic-argument = "," ".." argument
call-try = "?" [ call-try-catch ]
call-try-catch = "catch" "{" < call-try-catch-arm > "}"
call-try-catch-arm = try-catch-pattern [ pattern-arm-guard ] "=>" expression ";"
```

Method Call Expression

```
method-call-expression = "->" value-identifier argument-list [ call-try ]
```

Index Expression

```
index-expression = index-list
index-list = "[" [ index { "," index } [ variadic-index ] ] "]"
index = expression
variadic-index = "," ".." index
```

Field Access Expression

```
field-access-expression = "." value-identifier
```

Primary Expressions

```
primary-expression = ( parenthesized-expression |
                       identifier-expression |
                       literal-expression |
                       lambda-expression |
                       module-expression |
                       record-expression |
                       exception-expression |
                       tuple-expression |
                       array-expression |
                       set-expression |
                       map-expression |
                       block-expression |
                       if-expression |
                       cond-expression |
                       match-expression |
                       for-expression |
                       while-expression |
                       loop-expression |
                       break-expression |
                       receive-expression |
                       raise-expression |
                       freeze-expression |
                       return-expression )
```

Parenthesized Expression

```
parenthesized-expression = "(" expression ")"
```

Identifier Expression

```
identifier-expression = value-identifier
```

Literal Expression

```
literal-expression = literal
```

Lambda Expression

```
lambda-expression = "fn" lambda-parameter-list "=>" expression
lambda-parameter-list = "(" [ lambda-parameter { "," lambda-parameter } [ variadic-lambda-parameter ] ] ")"
lambda-parameter = value-identifier
variadic-lambda-parameter = "," ".." lambda-parameter
```

Module Expression

```
module-expression = module-path
```

Record Expression

```
record-expression = "rec" [ module-identifier ] "{" [ expression-field { "," expression-field } ] "}"
expression-field = [ "mut" ] value-identifier "=" expression
```

Exception Expression

```
exception-expression = "exc" module-identifier "{" [ expression-field { "," expression-field } ] "}"
```

Tuple Expression

```
tuple-expression = "(" expression < "," expression > ")"
```

Array Expression

```
array-expression = [ "mut" ] "[" [ expression { "," expression } ] "]"
```

Set Expression

```
set-expression = [ "mut" ] "#" "{" [ expression { "," expression } ] "}"
```

Map Expression

```
map-expression = [ "mut" ] "#" "[" [ map-expression-pair { "," map-expression-pair } ] "]"
map-expression-pair = expression ":" expression
```

Block Expression

```
block-expression = "{" < statement > "}"
```

If Expression

```
if-expression = "if" expression block-expression [ if-expression-else ]
if-expression-else = "else" block-expression
```

Condition Expression

```
cond-expression = "cond" "{" < cond-expression-arm > "}"
cond-expression-arm = expression "=>" expression ";"
```

Match Expression

```
match-expression = "match" expression "{" < pattern-arm > "}"
pattern-arm = pattern [ pattern-arm-guard ] "=>" expression ";"
pattern-arm-guard = "if" expression
```

For Expression

```
for-expression = "for" pattern "in" expression block-expression
```

While Expression

```
while-expression = "while" expression block-expression
```

Loop Expression

```
loop-expression = "loop"
```

Break Expression

```
break-expression = "break"
```

Receive Expression

```
recv-expression = "recv" "{" < pattern-arm > "}" [ recv-expression-else ]
recv-expression-else = "else" block-expression
```

Raise Expression

```
raise-expression = "raise" expression
```

Freeze Expression

```
freeze-expression = "freeze" [ "in" ] expression
```

Return Expression

```
return-expression = "return" expression
```

Patterns

```
pattern = ( identifier-pattern |
            literal-pattern |
            module-pattern |
            tuple-pattern |
            record-pattern |
            exception-pattern |
            array-pattern |
            set-pattern |
            map-pattern ) [ "as" [ "mut" ] value-identifier ]
try-catch-pattern = ( identifier-pattern |
                      exception-pattern ) [ "as" [ "mut" ] value-identifier ]
```

Identifier Pattern

```
identifier-pattern = [ "mut" ] value-identifier
```

Literal Pattern

```
literal-pattern = nil-literal |
                  boolean-literal |
                  atom-literal |
                  [ "-" ] ( integer-literal | real-literal ) |
                  string-literal
```

Module Pattern

```
module-pattern = module-path
```

Tuple Pattern

```
tuple-pattern = "(" pattern < "," pattern > ")"
```

Record Pattern

```
record-pattern = "rec" [ module-identifier ] "{" [ pattern-field { "," pattern-field } ] "}"
pattern-field = value-identifier "=" pattern
```

Exception Pattern

```
exception-pattern = "exc" module-identifier "{" [ pattern-field { "," pattern-field } ] "}"
```

Array Pattern

```
array-pattern = "[" [ pattern { "," pattern } ] "]" [ array-pattern-remainder ]
array-pattern-remainder = "::" pattern
```

Set Pattern

```
set-pattern = "#" "{" [ expression { "," expression } ] "}"
```

Map Pattern

```
map-pattern = "#" "[" [ map-pattern-pair { "," map-pattern-pair } ] "]"
map-pattern-pair = expression ":" pattern
```