Language Reference Manual

Yanlin Duan	Zhuo Kong	Emily Meng	Shiyu Qiu
yd2380	zk2202	ewm2136	sq2156
System Architect	Tester	Language Guru	Manager

Table of Contents

1. Introduction

rusty is a condensed version of the Rust programming language, based on C. rusty retains Rust's key features of memory safety and thread concurrency safety. rusty can be used as a general purpose programming language, but is intended to be a safer systems programming language. rusty uses the LLVM as a compiler IR, aligning with the goal of keeping rusty as a fast, but flexible language.

2. Lexical Conventions

2a. Tokens

There are five classes of tokens: identifiers, keywords, literals, operators, and separators.

White space such as blanks, tabs, and newlines are ignored. However, there are cases where they are required to separate tokens.

```
let whitespace = [' ' '\t' '\r']
let newline = '\n'
```

Comments, denoted as beginning with the characters `/*` and terminating with the characters `*/`, are also ignored. They do not nest and cannot appear within literals.

```
and comment = parse
    newline { incr lineno; comment lexbuf }
  |    "*/"     { decr depth; if !depth > 0 then comment lexbuf else token lexbuf }
  | "/*"     { incr depth; comment lexbuf }
  |    _        { comment lexbuf }

and comment2 = parse
    newline {token lexbuf}
  | _ {comment lexbuf}
```

2b. Identifiers

Identifiers are sequences of mainly letters and digits, but usage of the underscore is allowed. Upper and lowercase letters are considered to be distinct. The first character of an identifier cannot be a number. An identifier cannot be the same as an existing keyword.

```
let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let id = alpha (alpha | digit | '_')*
```

2c. Keywords

Keywords are specific identifiers reserved for use by the language and may not be used otherwise.

```
bool
break
char
const
continue
else
false
float
for
if
int
let
in
mut
return
static
string
struct
true
tuple
void
while
thread
spawn
join
send
sync
mutex
and
or
not
loop
fn
impl (struct methods)
as  (type cast)
```

2d. Literals

Literals are sequences of characters, whose valid character sets vary depending on type of use. The literals are:

- Integer
- Float
- Boolean
- Character
- String

Integer literals are sequences of digits representing only whole number values in decimal.

```
let digit = ['0'-'9']
let int = digit+
```

Float literals include an integer part, a decimal point, a fraction part, an optional e or E with integer exponent. The integer part, fraction part, and integer exponent are both sequences of digits. Either the integer or the fraction part may be missing, but not both.

Float literals are 32 bits.

```
let float = (digit+) ['.'] digit+
```

Boolean literals are one of two valid sequences of characters, true or false.

```
"true"   { TRUE }
| "false"  { FALSE }
```

Character literals are single character sequences surrounded by single quotes. Special characters are represented with an escape sequence of a single backslash and a character.

`'\''` - single quote

`'\"'` - double quote

```
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let char = ''' ( ascii | digit ) '''
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
```

String literals are sequences of characters surrounded by double quotes. The valid character set includes anything that can be represented by a character literal.

```
let string = '"' ( (ascii | escape)* as s) '"'
```

2e. Operators

Operators are specific lexical elements reserved for use by the language and may not be used otherwise. Refer to the Expressions section for functionality and use cases.

- Arithmetic: + - * / %
- Assignment: =
- Equivalence: == != < <= > >=
- Logical: and, or, not

- Reference: & mut &mut *

2f. Separators

Separators are specific lexical elements reserved for use by the language and may not be used otherwise. They are responsible for denoting the separation between tokens. White space is considered a separator.

```
( ) { } [ ] ; : , .
```

3. Types

3a. Primitive Data Types

int

The integer type stores whole number values in 32 bits.

```
let x : int =32;
```

float

The float type stores fractional number values in 32 bits.

```
let y: float = 0.4;
```

bool

The bool type represents either true or false.

```
let y: bool = false;
```

void

The void type represents an empty value used only as the type returned by functions that do not generate values. It cannot be used as a variable type.

```
fn print_num(x: int)-> void {
  println!(“{}”, x);
}
```

char

The char type represents a single character surrounded by single quotes stored in 8 bits.

```
let x: char ='x';
```

string

The string type represents a sequence of characters either as a literal constant or as some kind of variable, they are always valid UTF-8, and usually seen in its borrowed form &string.

```
let hello: string = "Hello, world!";
```

3b. Non-Primitive Data Types

array

The array type represents a fixed-size array, denoted `<type> <array name> [<size>]` in declaration and `<type> <array name> = [elements]` in definition.

Syntax:

```
let array1: [int;4] = [1,2,3,4];
println!("{}",array1[0]);
```

Grammar:

```
LBRACK typ SEMI INT_LITERAL RBRACK {ArrayT($2,$4)} /* array type */
LBRACK expr_list RBRACK { ArrayLit(List.rev $2) } /* array  creation */
expr LBRACK expr RBRACK { ArrayTupleAccess($1,$3) } /* array access */
```

tuple

The tuple type represents a finite heterogeneous sequence.

Syntax:

```
let tuple: (string,int,char) = ("hello", 5, 'c');
```

Grammar:

```
LPAREN func_input_opt RPAREN {TupleT(List.rev $2)} /* tuple type */
LPAREN RPAREN { TupleLit([]) } /* empty tuple or unit */
LPAREN tuple_list RPAREN { TupleLit(List.rev $2) } /* tuple creation */
expr LBRACK expr RBRACK { ArrayTupleAccess($1,$3) } /* tuple access */
```

struct

The struct type is a single, unified data type that combines variables with names of these variables as field labels.

Syntax:

```
struct Point {
    x: int,
    y: int,
}

fn main()-> void {
    let origin: Point = { x: 0, y: 0 };
    println!("The origin is at {}, {}", origin.x, origin.y);
}
```

Grammar:

```
ID { StructT($1) } /* struct type */
STRUCT ID LBRACE formals_opt RBRACE { StructDef($2,$4) } /* struct definition */
ID LBRACE struct_list_opt RBRACE { StructCreate($1,$3) } /* struct creation */
expr DOT ID { StructAccess($1,$3) } /* access struct */
```

3c. Type Qualifiers

const

The const qualifier indicates that the value of the variable will not be changed.

Syntax:

```
let const a:int =32;
a = 64;// error:left-hand of expression not valid
```

Grammar:

```
  CONST ID COLON typ ASSIGN expr { Constant(($4,$2),$6)}
```

mut

The mut qualifier indicates that the value of the variable is mutable, meaning its value and memory can be borrowed by another variable.

Syntax:

```
let mut x: int = 5;
x = 6; // mutable x allows changing x’s contents
let y: &mut int = &mut x; // y is referencing/borrowing x’s resource and can change x’
s contents
```

Grammar:

```
MUTABLEBORROW typ { SliceT(Mut,$2) }
MUTABLEBORROW expr {Unop(Borrow(Mut), $2)}
STATIC MUTABLE ID COLON typ ASSIGN expr { Static(Mut,($5,$3),$7) }
```

static

The static qualifier provides the global variable functionality. Static items are similar to constance, but static items aren’t inlined with upon use, so there is only one instance for each value and it’s fixed at a location in memory.

Syntax:

```
let static N:int = 5;
```

Grammar:

```
STATIC ID COLON typ ASSIGN expr { Static(Immut,($4,$2),$6)}
STATIC MUTABLE ID COLON typ ASSIGN expr { Static(Mut,($5,$3),$7) }
```

3d. Type Cast

To ensure safety, rusty does not do any implicit type cast (there will be neither promotion nor demotion). However, rusty does support explicit type cast using keyword ``as'':

```
let x : int = 5.0 as int;
```

4. Operators

4a. Arithmetic

The binary arithmetic operators are `+, -, *, /, %`.

Integer division truncates any fractional part.

The binary `+` and `-` operators have the same precedence, which is lower than the precedence of `*, /, %`. Arithmetic operators associate left to right.

Syntax:

```
let x:int = (2+3) * 4;
```

Grammar:

```
expr:
  expr PLUS   expr { Binop($1, Add,   $3) }
    | expr MINUS  expr { Binop($1, Sub,   $3) }
    | expr TIMES  expr { Binop($1, Mult,  $3) }
    | expr DIVIDE expr { Binop($1, Div,   $3) }
    | expr MODULO expr { Binop($1, Mod,   $3) }
```

4b. Assignment

The operator `=` assigns an expression to a variable. Assignment operator is right associative.

Syntax:

```
let x:int = 3;
```

Grammar:

```
LET ID COLON typ ASSIGN expr   { Assign(Immut, ($4, $2), $6) }
| LET MUTABLE ID COLON typ ASSIGN expr { Assign(Mut, ($5,$3),$7)}
```

4c. Equivalence

The relational operators are `<, <=, >, >=`. They all have the same precedence. Below them are `==` and `!=`. Relational operators have lower precedence than arithmetic operators.

Syntax:

```
let x:bool = (3 == 3);
```

Grammar:

```
expr:
  expr EQ     expr { Binop($1, Equal, $3) }
    | expr NEQ    expr { Binop($1, Neq,   $3) }
    | expr LT     expr { Binop($1, Less,  $3) }
    | expr LEQ    expr { Binop($1, Leq,   $3) }
    | expr GT     expr { Binop($1, Greater, $3) }
    | expr GEQ    expr { Binop($1, Geq,   $3) }
```

4d. Logical

The logical operators are `and, or, not`.

Syntax:

```
if (not valid)
if (valid or not_valid)
if (a_valid and b_valid)
```

Grammar:

```
expr:
  expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | NOT expr         { Unop(Not, $2) }
```

4e. Memory

The unary reference operator `&` applied to a variable borrows ownership of the variable's resource but continues to be immutable.

The unary mutable reference operator `&mut` applied to a variable's lvalue allows it's rvalue to borrow ownership of the resource and allows the borrowed resource to be mutated.

The unary dereference operator `*` applied to a mutable reference variable will dereference it and give access to the resource or actual contents of a reference.

Grammar:

```
TIMES expr %prec DEREF { Unop(Deref, $2) } /* * */
BORROW expr {Unop(Borrow(Immut), $2)} /* borrow */
MUTABLEBORROW expr {Unop(Borrow(Mut), $2)} /* mut borrow */
```

4f. Other unary operator expressions

Unary operators `+` and `-` give positive and negative signs to numeric values. They have higher precedence than arithmetic operators.

Syntax:

```
let x:int = -3;
let x:int = +3;
```

Grammar:

```
MINUS expr %prec NEG { Unop(Neg, $2) } /* - */
PLUS expr %prec POS { Unop(Pos, $2)} /* + */
```

5. Statements and Expressions

5a. Declarations and lvalues/rvalues

Variables are declared through let statements that are followed by a type annotation and an initializer expression, though it is not necessary to initialize right away. Any variables created through declarations are valid from the point of declaration until the end of its enclosing scope.

Syntax:

```
let x: int = 5;
```

Expressions are divided into lvalues and rvalues. Within each expression, subexpressions occur in either lvalue context or rvalue context. Evaluation of an expression relies on whether it is an lvalue or rvalue and the context in which the expression occurs.

Lvalues are expressions representing memory locations. All other expressions are rvalues.

When an lvalue is evaluated in an lvalue context, it denotes a memory location. When an lvalue is evaluated in an rvalue context, it denotes the value held in that memory location.

5b. Literal expressions

A literal expression consists of one of the literal forms described earlier or the unit value, and ending with a semicolon.

Syntax:

```
( );   // unit value expression
"rusty";   // string literal expression
'r';       // character literal expression
42;         // integer literal expression
42.35;   // float literal expression
```

5c. Array expressions

An array expression is written by enclosing zero or more comma-separated expressions of uniform type in square brackets.

Syntax:

```
[ ]     // empty array
[1, 2, 3, 4]  // array of integers
[“a”, “b”, “c”, “d”, “e”] // array of characters
```

An array expression may also be of form:

`[expr1]*expr2`, where the expression after `*` must be a constant integer literal expression (so that it can be evaluated in the compile time). This is equivalent to:

```
[expr1, expr1, expr1, … ] (expr2 times).
```

For example:

```
[0]*2;
```

is equivalent to:

```
[0,0];
```

5d. Tuple expressions

Tuples are written by enclosing zero or more comma-separated expressions in parentheses. They are used to create tuple-typed values.

Tuple-typed values, as defined in 3b, is a heterogeneous product of other types.

Syntax:

```
(0,"a",true); //a tuple consists of integer type, string type and bool type values
```

5e. Struct expressions

A struct expression is written as a brace-enclosed list of zero or more comma-separated name-value pairs, providing the field values of a new instance of the struct.

For example, given a struct definition:

```
struct Point { x: float, y: float }
```

We can create a new Point struct like this:

```
let f: Point = {x: 10.0, y: 20.0};
```

We may also associate methods to struct using keyword impl and call it using syntax below:

```
impl Point: {

  fn sumXY(&self: &struct) -> float {
    return (self.x + self.y);
}

let f: Point = {x: 10.0, y: 20.0};
println!("{}",f.sumXY());
```

Tuple is essentially a struct without names, so we can do similar things to create a new instance of tuple:

```
let tuple_x: (float, string) = (20, “hello rusty”);
```

5f. Function calls

A function item defines a sequence of statements and a final expression, along with a name and a set of parameters. Other than a name, all these are optional. Functions are declared with the keyword fn.

Functions may declare a set of input variables as parameters, through which the caller passes arguments into the function, and the output type of the value the function will return to its caller on completion.

A hello-world function will look like this:

```
fn helloWorld() -> void {
    println!(“Hello world!”)
}
```

Grammar

```
fdecl: /* fn foo(x:int) -> int { ... } */
    FUNC ID LPAREN formals_opt RPAREN OUTPUT typ LBRACE stmt_list RBRACE
     {
     outputType = $7;
     fname = $2;
     formals = $4;
     body = List.rev $9 }

FUNC LPAREN func_input_opt RPAREN OUTPUT typ /* func type */
{
  FunctionT{
      input:$3;
      output:$6
  }

ID LPAREN actuals_opt RPAREN { Call($1, $3) } /* Function Call */
```

5g. Control flow

if/else

An if expression is a conditional branch in program control. The form of an if expression is a condition expression, followed by a consequent block, and an optional trailing else block.

```
if (expr) {
 (block)
}
…
else {
 (block)
}
```

The condition expressions must have type bool. If a condition expression evaluates to true, the consequent block is executed and any subsequent else if or else block is skipped. If a condition expression evaluates to false, the consequent block is skipped and any subsequent else if condition is evaluated. If all if and else if conditions evaluate to false then else block is executed (if any).

Grammar:

```
stmt:
    IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
```

loops

rusty supports three kinds of loops: for, while, and loop (infinite loop).

- for

A for expression is used for looping over elements provided by an implementation of std::iter::IntoIterator. (so far support only array).

For example, we may loop through an array or a String:

```
fn bar(f: &int) -> void {
    println!("{}",f);
}
let a: int = 0;
let b: int = 1;
let c: int = 2;

let v: &[int] = &[a, b, c];

for e in v {
    bar(e); //print 0, 1, 2 one at a time
}
```

Grammar:

```
FOR ID IN expr LBRACE stmt RBRACE
     { For($2, $4, $6) }
```

- while

A while loop begins by evaluating the boolean loop conditional expression. If the loop conditional expression evaluates to true, the loop body block executes and control returns to the loop conditional expression. If the loop conditional expression evaluates to false, the while expression completes.

Syntax:

```
while (expr) {
 (block)
}

let mut i:int = 0;

while i < 42 {
    println!("hello world");
    i = i + 1;
} //will print “hello world!” 42 times.
```

Grammar:

```
  WHILE LPAREN expr RPAREN stmt { While($3, $5) }
```

- loop

The "loop" keyword indicates an infinite loop.

Syntax:

```
let count: int = 0;

loop {
        count += 1;
}
```

Grammar:

```
LOOP stmt { Loop($2) }
```

break/continue

rusty also has break and continue expression to control the flow.

A `break` expression immediately terminates the innermost loop enclosing it. It is only permitted in the body of a loop.

A `continue` expression immediately terminates the current iteration of the innermost loop enclosing it, returning control to the loop head. In the case of a while loop, the head is the conditional expression controlling the loop. In the case of a for loop, the head is the call-expression controlling the loop. It is only permitted in the body of a loop.

5h. Block

A block expression conceptually introduces a new namespace scope. Use items can bring new names into scopes and declared items are in scope for only the block itself.

A block will execute each statement sequentially, and then execute the expression (if given). If the block ends in a statement, its value is `()`.

Syntax:

```
let x: () = { println!("Hello."); }; //a block expression with value ( )
let x: int = { println!("Hello."); 5 }; //a block expression with int value 5.
```

6. Memory and Safety

rusty’s main goal is to provide memory safety in the same way as Rust. Declarations without assignment are not allowed.

When variables are assigned to a value, variables are bound to a resource, where they have ownership. As soon as the variable goes out of scope, the variable’s resources are automatically freed, and any future references to the variable are no longer valid.

At any given time, there is only one variable that is allowed to own any given resource. After a variable's value has been moved from the old binding to a new one, the old binding can no longer be used. The new binding's value is immutable.

Rather than moving ownership from one variable to another completely, it is possible to borrow ownership via references using the `&` operator. A new binding that borrows ownership does not deallocate the resource when the binding goes out of scope, allowing one to use the original binding again. Values of references are still immutable.

A mutable reference done via the `&mut` operator provides a way for mutable ownership borrowing. A `*` operator is required to access the contents of a mutable reference. A mutable reference can only be made on a variable that is explicitly declared as mutable with the `mut` qualifier.

7. Thread Concurrency Safety

rusty's memory safety extends to thread usage and concurrency as well. The goal is to prevent data races among code executing in parallel. The standard library is used for threads.

There are two keywords crucial to concurrency, `send` and `sync`. `send` indicates that a variable of a given type can have ownership of its variables safely transferred between different threads. `sync` indicates that a variable of a given type is guaranteed to uphold safety when used from different threads running at the same time.

A thread is created with a call to the `spawn` method. It returns a handle to the child thread, which can be used to wait for the child to finish and obtain its result. `spawn` accepts a closure as an argument, which by default take references to variables. By using a move closure, variables are moved from the original closure environment into the new thread.

The `join` method will free memory associated with the chosen thread and return its state.

The `panic` method will crash the currently executing thread.

Two wrapper types, `rc` and `arc`, meaning reference count and atomic reference count respectively, are used to help track the total count of references made to variables being operated on within threads.

`rc` guarantees that a variable will not be destroyed until all references to it are out of scope. Variable resources are immutable. `rc` is not thread safe. `arc` is a version of `rc` that is thread safe. Reference counts in both `rc` and `arc` are increased when their variables are called with the special clone function.

Two types of locks are provided as wrapper types, `mutex` and `rwlock`. They provide memory safety in mutating variable resources across threads, but incorrect usage may lead to deadlocks. A thread calling the `lock` method will obtain the lock for the variable until it releases the lock with the `unlock` method or goes out of scope. Other threads trying to lock the same lock will block until a lock can be acquired.

```
fn main() -> void {
    let data:array = arc::(mutex::(array[1, 2, 3])); // arc allows multiple thread sha
ring, mutex allows mutability
    let x:int = 0;

    for (i = 0; i < 10; i++) {
        let data:array = data.clone(); // increase reference count to data
        thread::spawn(move || { // spawn thread
            let mut data:array = data.lock(); // lock mutex call
            data[0] += i; // mutate array
        }); // unlock mutex implicitly since out of scope
    }
}
```

8. Grammar

Scanner

```
let alpha = ['a'-'z' 'A'-'Z']
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let id = alpha (alpha | digit | '_')* (*TODO: exclude keywords*)
let string = '"' ( (ascii | escape)* as s) '"'
let char = ''' ( ascii | digit ) '''
let float = (digit+) ['.'] digit+
let int = digit+
let whitespace = [' ' '\t' '\r']
let newline = '\n'

rule token = parse
  whitespace { token lexbuf }
| newline      { incr lineno; token lexbuf}
| "/*"     { incr depth; comment lexbuf }
| "//"     { comment2 lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
(* Operators *)
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MODULO }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "and"    { AND }
| "or"     { OR }
| "not"    { NOT }
| '.'      { DOT }
| '['    { LBRACK }
| ']'    { RBRACK }
```

```
(* Ownership *)
| "&"    { BORROW }
| "mut"    { MUTABLE }
| "&mut"   { MUTABLEBORROW }
| "static" { STATIC }
| "const" { CONST }
(* Branch Control *)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "loop"   { LOOP }
| "continue" { CONTINUE }
| "break"  { BREAK }
| "in"     { IN }
(* Data Types *)
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "char"   { CHAR }
| "true"   { TRUE }
| "false"  { FALSE }
| "struct" { STRUCT }
| "impl"   { IMPL }
| "let"    { LET }
| "as"     { AS  }
| "void"   { VOID }
(* function *)
| "fn"     { FUNC }
| "->"     { OUTPUT }
| "return" { RETURN }

(* Other *)
| int as lxm           { INT_LITERAL(int_of_string lxm) }
| float as lxm         { FLOAT_LITERAL(float_of_string lxm) }
| char as lxm          { CHAR_LITERAL( String.get lxm 1 ) }
| escape_char as lxm{ CHAR_LITERAL( String.get (unescape lxm) 1) }
| string               { STRING_LITERAL(unescape s) }
| id as lxm            { ID(lxm) }
| eof                  { EOF }
| '"'             { raise (Exceptions.UnmatchedQuotation(!lineno)) }
| _ as illegal  { raise (Exceptions.IllegalCharacter(!filename, illegal, !lineno)) }

and comment = parse
    newline { incr lineno; comment lexbuf }
  |    "*/"     { decr depth; if !depth > 0 then comment lexbuf else token lexbuf }
  | "/*"     { incr depth; comment lexbuf }
  |    _        { comment lexbuf }

and comment2 = parse
    newline {token lexbuf}
  | _ {comment lexbuf}
```

Parser

```
program:
  decls EOF { $1 }

decls:
   /* nothing */ { [] }
 | decls fdecl { $2 :: $1 }

fdecl: /* fn foo(x:int) -> int { ... } */
    FUNC ID LPAREN formals_opt RPAREN OUTPUT typ LBRACE stmt_list RBRACE
     {
     outputType = $7;
     fname = $2;
     formals = $4;
     body = List.rev $9 }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    ID COLON typ              { [($3,$1)] }
  | formal_list COMMA ID COLON typ { ($5,$3) :: $1 }

typ:
      INT { DataT(IntT) }
  | BOOL { DataT(BoolT) }
  | FLOAT { DataT(FloatT) }
  | CHAR { DataT(CharT) }
  | VOID { DataT(VoidT) }
  | LBRACK typ SEMI INT_LITERAL RBRACK {ArrayT($2,$4)} /* [int;10] type is an integer
array of length 10 */
  | LPAREN func_input_opt RPAREN {TupleT(List.rev $2)} /* (int,char,float) */
  | BORROW typ { SliceT(Immut,$2) } /* &int */
  | MUTABLEBORROW typ { SliceT(Mut,$2) } /* &mut int */
  | ID { StructT($1) }
  | FUNC LPAREN func_input_opt RPAREN OUTPUT typ /* func type: fn (int) -> int */
  {
    FunctionT{
        input:$3;
        output:$6
    }
  }

func_input_opt:
    /* nothing */ { [] }
  | func_input_list { List.rev $1}

func_input_list:
    typ {[$1]}
  | func_input_list COMMA typ { $3 :: $1}
```

```
struct_list_opt:
    /* nothing */ { [] }
  | struct_list { List.rev $1}

struct_list:
    ID COLON expr {[$1,$3]}
  | struct_list COMMA ID COLON expr { ($3,$5) :: $1 }


stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR ID IN expr LBRACE stmt RBRACE
     { For($2, $4, $6) }
  | LOOP stmt { Loop($2) } /* loop {...} is an infinite loop */
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | STRUCT ID LBRACE formals_opt RBRACE { StructDef($2,$4) } /* Struct */
  | IMPL ID LBRACE fdecl RBRACE { ImplDef($2,$4) }

expr_list:
    /* nothing */ { [] }
  | expr_list COMMA expr { $3 :: $1 }

tuple_list:
    expr COMMA expr {[$3,$1]}
  | expr_list COMMA expr { $3 :: $1 }

expr:
    INT_LITERAL      { IntLit($1) } /* Literals */
  | FLOAT_LITERAL    { FloatLit($1)}
  | CHAR_LITERAL     { CharLit($1)}
  | STRING_LITERAL   { StringLit($1)}
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID   %prec NOACCESS  { Id($1) }
  | LBRACK expr_list RBRACK { ArrayLit(List.rev $2) } /* Array and Tuple */
  | LPAREN RPAREN { TupleLit([]) } /* ( ) empty tuple (or unit)*/
  | LPAREN expr COMMA RPAREN { TupleLit([$2]) }
  | LPAREN tuple_list RPAREN { TupleLit(List.rev $2) }
  | expr LBRACK expr RBRACK { ArrayTupleAccess($1,$3) }
  | ID LBRACE struct_list_opt RBRACE { StructCreate($1,$3) }
  | expr DOT ID { StructAccess($1,$3) } /* Point.x is struct access */
  | expr PLUS   expr { Binop($1, Add,   $3) } /* Binary Operation */
```

```
   | expr MINUS  expr { Binop($1, Sub,   $3) }
   | expr TIMES  expr { Binop($1, Mult,  $3) }
   | expr DIVIDE expr { Binop($1, Div,   $3) }
   | expr MODULO expr { Binop($1, Mod,   $3) }
   | expr EQ     expr { Binop($1, Equal, $3) }
   | expr NEQ    expr { Binop($1, Neq,   $3) }
   | expr LT     expr { Binop($1, Less,  $3) }
   | expr LEQ    expr { Binop($1, Leq,   $3) }
   | expr GT     expr { Binop($1, Greater, $3) }
   | expr GEQ    expr { Binop($1, Geq,   $3) }
   | expr AND    expr { Binop($1, And,   $3) }
   | expr OR     expr { Binop($1, Or,    $3) }
   | MINUS expr %prec NEG { Unop(Neg, $2) } /* Unary Operation */
   | NOT expr         { Unop(Not, $2) }
   | PLUS expr %prec POS { Unop(Pos, $2)}
   | TIMES expr %prec DEREF { Unop(Deref, $2) }
   | BORROW expr {Unop(Borrow(Immut), $2)}
   | MUTABLEBORROW expr {Unop(Borrow(Mut), $2)}
   | LET ID COLON typ ASSIGN expr   { Assign(Immut, ($4, $2), $6) } /* Assignment */
   | LET MUTABLE ID COLON typ ASSIGN expr { Assign(Mut, ($5,$3),$7)}
   | expr AS typ      { Cast($3, $1)} /* 32 as float */
   | CONST ID COLON typ ASSIGN expr { Constant(($4,$2),$6)}
   | STATIC ID COLON typ ASSIGN expr { Static(Immut,($4,$2),$6)}
   | STATIC MUTABLE ID COLON typ ASSIGN expr { Static(Mut,($5,$3),$7) }
   | ID LPAREN actuals_opt RPAREN { Call($1, $3) } /* Function Call */
   | ID DOT ID LPAREN actuals_opt RPAREN { StructMethodCall($1,$3,$5) } /* Call Struct
Methods */
   | LPAREN expr RPAREN { $2 } /* Parenthesis */

actuals_opt:
    /* nothing */ { [] }
   | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
   | actuals_list COMMA expr { $3 :: $1 }
```

References

- GNU C Reference Manual: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html
- Rust Reference: https://doc.rust-lang.org/reference.html
- K&R C Programming Language
- Dice Project Report: http://www.cs.columbia.edu/~sedwards/classes/2015/4115-fall/reports/Dice.pdf