CS 301 Spring 2021 T4
Roost **0.2.0**
Document Revision 10,
Updated 1 June 2021

Roost **Language Specification**

Contents

1 Overview

Roost a statically typed programming language invented for CS 301. The core Roost language provides functions, primitive data types, simple compound data structures, scoping, and basic control flow. Roost is a syntactically similar to Rust, but semantically closer to Swift. A set of extensions to the core includes features partly familiar from other languages such as Java, Scala, ML, Rust, Swift, and countless others. The Roost language is simple enough that its grammar fits on one page, yet sophisticated enough to require many interesting considerations in a compiler. This document gives an informal overview of the language syntax and semantics, with some formal specifications, including the grammar and type system.

1.1 Core Language Highlights

The core Roost language requires:

- *Sound static typing:* Roost has a sound static type system with built-in types for integer, Boolean, and string values, variant enumeration types, compound structure types, array types, and function types.
- *Function references as values:* Roost supports basic higher order functions, allowing references to functions to be passed, stored, and returned as values. The core language lacks nested function definitions and closures.
- *Dynamic allocation and garbage collection*: Roost uses dynamic heap allocation and manipulation by reference for structures, variants, strings, and arrays. The language allows for garbage collection to automatically reclaim heap space.
- *Null safety and memory safety:* The language has no nullable types. Enumeration variants can encode presence or absence of a value without risking run-time null dereference. Roost uses run-time checks for array bounds violations, negative array size, and division by zero.

1.2 Standard Extensions

The full Roost language extends the core language with these features that improve the ergonomics of the language and make interesting optional extensions to the implementation project:

- *Generic types:* The full Roost language includes *generic types* (more technically: parametric type polymorphism), similar to Java, Scala, ML, Rust, and Swift.
- *Module system:* The full Roost language includes a module system providing nested name spaces for functions, enumerations, and structures, as well as support for abstract types.

1.3 Acknowledgments

Parts of this document, aspects of the Roost language, and related assignments were adapted from materials developed at Cornell University and Williams College.

2 Lexical Structure

2.1 Keywords

The following are case-sensitive keywords in the core Roost language and standard extensions:

> `enum extern fn mod mut pub struct type use bool i64 String else if let match while break continue return false true`

Non-alphabetic lexical tokens (along with the above keywords) are shown in Figure 1.

2.2 Identifiers

Identifiers must begin with a letter followed by any sequence of letters, digits, or the underscore character (`_`). Keywords are not valid as identifiers. Identifiers are case-sensitive: `x` and `X` are distinct identifiers. The case of the first character of an identifier distinguishes among two families of identifiers:

- Id: Identifiers starting with a lower-case letter are used to name variables, functions, structure fields , and modules .
- TypeId: Identifiers starting with an upper-case letter are used to name types and enumeration variants.

2.3 Comments

There are two comment forms.

- A **line comment** begins with `//` and includes all subsequent characters through the next newline character or the end of the program text, whichever comes first.
- A **block comment** begins with `/*` and includes all subsequent characters, including newline, through the next occurrence of `*/`. An unclosed block comment is a lexical error.

2.4 Literals

The language includes Boolean, integer, and string literal tokens:

- **Integer literals** (Integer) consist of a sequence of digits. Non-zero integer literals must not have leading zeroes. Integers have 64-bit signed values in the range -2^{63} through $2^{63}-1$, inclusive, but decimal integer literals capture only the non-negative part of this range. The form `-301` is **not** an integer literal; it is a unary integer negation expression applied to the integer literal `301`.
- **String literals** (String) are sequences of characters delimited by double quotes. String characters can be: single printable ASCII characters (ASCII codes between decimal 32 and 126) except double-quote (`"`) and backslash (`\`); or the escape sequences `\"` to denote quote, `\\` to denote backslash, `\t` to denote tab, and `\n` to denote newline. No other characters or character sequences can occur in a string. Unclosed strings are lexical errors.
- **Boolean literals** are the keywords `true` and `false`.

Note that the unit literal is given by a pair of separate tokens, `(` followed by `)`, **not** a single token `()`.

2.5 Tokenization

Whitespace consists of a sequence of one or more space, tab, or newline characters. Whitespace may appear between any tokens. Keywords and identifiers must be separated by whitespace or a token that is neither a keyword nor an identifier. For instance, `elsey` represents a single identifier, not the keyword `else` followed by the identifier `y`.

3 Program Structure and Syntax

The top level of a program in the core Roost language consists of a sequence of *function* (`fn`), *structure type* (`struct`), and *enumeration type* (`enum`) definitions, and must include exactly one function definition indicating the program entrypoint that will be called with the array of command-line arguments when the program is launched. The entrypoint function must be named `main`, must take exactly one argument of type `[String]` (an array of strings), and must have result type `()` (unit):

```
fn main(args: [String]) -> () { /* body of main function */ }
```

Subsequent sections of this document describe the syntax and semantics of various language forms informally.

Figure 1 gives a formal definition of the Roost syntax. To eliminate ambiguity, this grammar uses significant internal structure, especially within the grammar for expressions, paired with operator precedence rules defined separately in Table 1. Fixed terminals including keywords, operators, and other symbols, are shown using single quotes around monospace font (e.g., '`let`' and '`;`'). Lexical structure of terminals for identifiers (Id, TypeId), integer literals (Integer), and string literals (String) is defined in Section 2. Non-terminals use slanted fonts (e.g., *expr*). The remaining symbols are meta-characters: $(...)^*$ and $(...)^+$ denote the Kleene *star* and *plus* operations, respectively, and $(\ldots)^?$ denotes an optional item.

4 Data Types

Values of types `()` (unit), `bool` (Boolean), and `i64` (integer) are passed *by value*: a copy of *the entire value itself* is stored, passed to/from functions calls/returns, or used by operations such as `==`.

Values of array, structure, enumeration, and `String` types are allocated in the heap and passed *by reference*: a copy of *the memory address at which the value resides* is stored, passed, or used. There is no "null" reference in Roost every array, structure, enumeration, and string value is a valid reference to a memory location contains the relevant type's representation. Note that the equality comparison operations (`==` and `!=`) check *reference equality* not *structural equality* on these types.

4.1 Unit, Booleans, Integers

The **unit** type `()` has a single value, also written `()`. Since all operations yield a result in Roost, the unit value and type are used to indicate a results that carries no interesting information. Unit plays a role similar to the role played by `void` in many statement-focused languages, but has a value. **Booleans** are represented by the type `bool`; the literals `true` and `false` have this type. The **integer** type `i64` is a 64-bit two's-complement integer; integer literals have this type. Integer arithmetic operations yield 64-bit results. On overflow low 64-bits of the true result are kept, yielding modular results.

4.2 Strings

Strings are immutable character sequences represented by the type `String`. Strings are immutable, meaning that new strings may be created by applying operations to existing strings, but the contents of existing strings cannot be modified. All string operations except reference equality comparison are provided by external functions in the standard library. Roost does not have a type for single characters.

4.3 Arrays

Fixed-length **arrays** of arbitrary element type `T` are have type `[T]`. There are no restrictions on array elements types: array elements can be arrays themselves, allowing programmers to build multidimensional arrays. For example, the type `[[T]]` describes a two-dimensional array constructed as an array of arrays of type `[T]`.

Arrays are created and initialized dynamically using one of two forms:

- `[` $e_1, e_2, \ldots\ e_n$ `]` creates a new array of n cells filled by the given elements e_1 through e_n, which much all have the same type.

```
     TopLevel  ::=  Item*
         Item  ::=  Function | Structure | Enumeration
     Function  ::=  Header Block | ‘extern’ Header ‘;’
       Header  ::=  ‘fn’ Id TypeParams? ‘(’ (‘mut’? Id ‘:’ Type ‘,’)* (‘mut’? Id ‘:’ Type)? ‘)’ ‘->’ Type
    Structure  ::=  ‘struct’ TypeId TypeParams? ‘{’ ( ‘pub’? Id ‘:’ Type ‘,’)* ( ‘pub’? Id ‘:’ Type)? ‘}’
  Enumeration  ::=  ‘enum’ TypeId TypeParams? ‘{’ (TypeId (‘(’ Type ‘)’)? ‘,’)* (TypeId (‘(’ Type ‘)’)? )? ‘}’
         Type  ::=  ‘(’ ‘)’ | ‘bool’ | ‘i64’ | ‘String’ | ‘[’ Type ‘]’ | Path? TypeId TypeArgs? | ‘!’
                |   ‘fn’ TypeParams? ‘(’ (Type ‘,’)* Type? ‘)’ ‘->’ Type
   Expression  ::=  ExprControl | ExprCompute
  ExprControl  ::=  Block | If
                |   ‘while’ ‘(’ Expression ‘)’ Block
                |   ‘match’ ‘(’ Expression ‘)’ ‘{’ (Pattern ‘=>’ Expression ‘,’)* (Pattern ‘=>’ Expression)? ‘}’
        Block  ::=  ‘{’ ‘}’ | ‘{’ Step* Effect ‘}’ | ‘{’ Step* End ‘}’
         Step  ::=  Effect | ExprControl ‘;’?
       Effect  ::=  ‘let’ ‘mut’? Id (‘:’ Type)? ‘=’ Expression ‘;’
                |   Location ‘=’ Expression ‘;’
                |   Call ‘;’
          End  ::=  Expression | ‘return’ Expression? | ‘break’ | ‘continue’
           If  ::=  ‘if’ ‘(’ Expression ‘)’ Block (‘else’ Else)?
         Else  ::=  Block | If
      Pattern  ::=  Literal | Id | ‘_’ | TypeId (‘(’ Pattern ‘)’)?
  ExprCompute  ::=  ExprCompute BinaryOp ExprCompute
                |   UnaryOp ExprCompute
                |   ExprConstruct
     BinaryOp  ::=  ‘*’ | ‘/’ | ‘%’ | ‘+’ | ‘-’ | ‘<<’ | ‘>>’ | ‘>>>’ | ‘&’ | ‘^’ | ‘|’
                |   ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘!=’ | ‘&&’ | ‘||’
      UnaryOp  ::=  ‘-’ | ‘!’
ExprConstruct  ::=  ‘[’ (Expression ‘,’)* Expression? ‘]’
                |   ‘[’ Expression ‘;’ Expression ‘]’
                |   Path? TypeId (‘(’ Expression ‘)’)?
                |   Path? TypeId ‘{’ (Id ‘:’ Expression ‘,’)* (Id ‘:’ Expression)? ‘}’
                |   Literal
                |   ExprCore
      Literal  ::=  ‘(’ ‘)’ | ‘true’ | ‘false’ | Integer | String
     ExprCore  ::=  ‘(’ Expression ‘)’ | Location | Call
     Location  ::=  Path? Id | ExprConstruct ‘[’ Expression ‘]’ | ExprConstruct ‘.’ Id
         Call  ::=  ExprCore ‘(’ (Expression ‘,’)* Expression? ‘)’
```

Figure 1: Syntax of core Roost language. Highlighted grammar elements refer to extensions beyond the core language, defined in Figure 5. In the core language, these optional elements are never present.

- `[` *e* `;` *n* `]` creates a new array of *n* cells, each of which is initialized to hold copies of the initial element value *e*.

The expression `array.length` evaluates to the (immutable) length of a given `array`. Neither `length` nor `.length` is a keyword. Instead, an array length expression is syntactically a structure field access (see Structure Types) with field name `length` that is distinguished semantically by the type system. Arrays of length n are indexed from 0 through $n - 1$ with bracket notation: `array[index]`. Array indexing includes run-time bounds checking. Bounds violations raise a run-time error.

4.4 Structure Types (`struct`)

Roost **structure types** or structs are program-defined compound data types comprised of multiple parts stored in **fields** of the structure. A structure type must be defined explicitly by the program, and provides the template for individual structures (instances of the structure type). Roost structs are like C structs and similar to Java classes without methods.

A structure type definition begins with the keyword `struct`, followed by identifier (starting with an upper-case letter) naming the structure type, followed by a curly-brace delimited, comma-separated list of declarations of fields. Field declarations consist of an identifier and a type. Field identifiers start with a lower-case letter. For example:

```
struct StudentRecord {
  name: String,
  class: i64,
  id: i64,
  majors: [String],
}
```

A comma following the last field declaration is optional.

This definition introduces a new type, `StudentRecord`, for use in the program, type rules for what fields it includes and what their types are, and a form for constructing new instances of this type.

A fresh instance of structure type `StudentRecord` is created by an expression of this form:

```
StudentRecord {
  name: "Com Piler",
  class: 2023,
  id: 301,
  majors: ["CS", "MUS"]
}
```

A comma following the last field initializer is optional. This expression allocates space for a `StudentRecord` structure and initializes its fields with the given values.

Fields of a structure instance are *mutable* and may be accessed for use or assignment with the `.` symbol: `student.majors` refers to the `majors` field of this structure given by `student`.

Note that the array length expression `array.length` appears syntactically as a structure field access with field name `length`. The logical field name, `length`, is *not* a keyword and is a valid program-defined identifier. Semantically, an array length expression is distinguished from a structure field access expression for field `length` by the type of the target expression. (See Section 8.)

4.5 Enumeration Types (`enum`)

Roost **enumeration types** or enums are program-defined data types that has multiple **variants** for different possible forms of data. An enumeration type must be defined explicitly by the program, and provides the template for possible variants of the enumeration type and the type of value they may carry. Roost enums are quite similar to Rust or Swift enums. They are more powerful than C or Java enums. They are similar to ML algebraic datatypes or Scala case classes without methods.

An enumeration type definition begins with the keyword `enum`, followed by identifier (starting with an upper-case letter) naming the enumeration type, followed by a curly-brace delimited, comma-separated list of variants. Variants consist of an identifier and, optionally, a parenthesized value type. Variant identifiers start with an upper-case letter. For example:

```
enum OptionalString {
  Some(String),
  None,
}
```

A comma following the last variant declaration is optional. Variants with carried values are indicated by a parenthesized type of carried values following the variant name. Variants without carried data lack parentheses.

This definition introduces a new type, `OptionalString`, for use in the program, type rules for what variants are available and what type of data they carry, and forms for constructing new instances of each variant of this type.

The expression `Some("hello")` constructs a new value of the `Some` variant of type `OptionalString`, carrying the string value `"hello"`. The expression `None` constructs a new value of the `None` variant of type `OptionalString`, carrying no data.

The carried value of an enumeration variant is immutable; the carried value cannot be accessed like the values of struct fields. Instead, *pattern matching* with a `match` expression, described later, allows inspecting an enum value to determine its variant extract the carried value.

4.6 Recursive Type Definitions

Structure and enumeration types can be used anywhere in the program, including to define recursive types, such as this linked list definition:

```
enum List {
  Link(Node),
  Empty
}
struct Node {
  value: i64,
  rest: List,
}
```

For example, this expression creates a linked list holding elements 1, 2, 3:

```
Link(Node {
  value: 1,
  rest: Link(Node {
    value: 2,
    rest: Link (Node {
      value: 3,
      rest: Empty
    })
  })
})
```

4.7 Storage and Variables

Program storage locations are immutable and mutable local variables (including function parameters, represented in the stack or registers), fields of structures (allocated in the heap), payloads of enumeration variants, and cells of arrays (allocated in the heap). By construction, the Roost language forces all program storage locations must be initialized explicitly upon creation.

5 Functions

All code for computation (as opposed to type definitions) in Roost programs appears inside the bodies of function definitions.

5.1 Function Definitions

A **function definition** begins with the keyword `fn`, followed by an identifier (starting with a lower-case letter) naming the function, followed by a parenthesized, comma-separated list of parameter identifiers and their types, followed by a right arrow and a result type. A function definitions is completed by a block expression delimited by open and close curly braces indicating the function body. For example:

```
fn factorialA(n: i64) -> i64 {
  if (n <= 1) {
    1
  } else {
    n * factorialA(n - 1)
  }
}
fn factorialB(n: i64) -> i64 {
  if (n <= 1) {
    return 1
  }
  n * factorialB(n - 1)
}
fn factorialC(n: i64) -> i64 {
  if (n <= 1) {
    return 1
  }
  return n * factorialC(n - 1)
}
```

Unlike local variables, function types–including parameter types and result types–must be notated explicitly: they are not inferred. As with local variables, function parameters support an optional `mut` qualifier. By default, parameter variables are immutable: they may not be mutated by assignment in the function body. Using the `mut` qualifier on the parameter declaration allows assignment.

```
fn param_mutation_example(x: i64, mut y: i64) -> i64 {
  if (x == y) {
    y = x + y; // This is allowed: y is mutable.
  } else {
    x = 7;     // This is a type error: x is not mutable.
  }
  return y / x
}
```

Result types for functions never include a `mut` qualifier since they do not describe a storage location.

5.2 Function Calls

Evaluation of a **function call** consists of the following steps: passing the parameter values from the caller to the callee, executing the body of the callee, and returning the control and the result value (if any) to the caller. Each time a function is called, the argument expressions are evaluated left to right and the resulting values are bound to the corresponding parameters of the function in a new scope. Structure, enumeration, array, and string arguments are passed as references.

After binding parameters to values, the body expression of the called function is evaluated and control transfers back to the caller with the result value of the body expression as the result value of the call, and control transfers back to the caller. If evaluation reaches an explicit `return`, evaluation ends early and control transfers back to the caller, returning the result value of the `return` argument expression as the result value of the call. The three `factorial` functions above work equivalently.

Statically, at each call site, the number and types of provided arguments must match the parameters of the function or declaration and the declared result type must match the expected type in the callee. Also, the return type from the declaration of a function must match the `return` or result expressions in the body. The type of the function's body–and all `return` expressions in the body–must be compatible with the declared result type of the function.

5.3 External Functions

To support calling basic system functions that are not written in Roost, the language includes a second form of function definition: the external function header. These functions are declared with the `extern` keyword as a prefix. Instead of providing a function body, the function header ends with a semicolon. For example, some basic system functions in the Roost standard library are not implemented in Roost themselves:

```
extern fn println(string: String) -> ();
```

6 Expressions

Roost is an expression-focused language that emphasizes evaluating for result values over evaluating for side effects. Roost still supports side effects, but structures them under expressions. Code structures that appear in function bodies (including the entire body itself) produce a result value when evaluated, even if that value is simply the unit-typed value `()`.

6.1 Blocks, Bindings, Assignments, Effects

Block expressions, including as required for function bodies, `if`, and `while`, control local scope and support sequencing of steps in evaluation. Block expressions are delineated by curly braces `{ }`. Blocks start with zero or more sequential **steps for effect** (rather than results):

- A `let` binding defines a new local variable that is in scope through the end of the containing block. The binding includes an optional `mut` qualifier, a variable identifier, an optional typing, and an expression to be evaluated to initialize the variable. By default, variables are *immutable*, meaning they do not support mutation (assignment). Variables declared with `let mut` are mutable: they support mutation by assignment. If the typing is omitted, the variable type is inferred from the initialization expression.

  ```
  let greeting: String = "hello";
  let object = "world";        // Type String is inferred
  let mut x: i64 = 4;          // x supports mutation by assignment
  ```

- An assignment evaluates its right hand side expression, then copies the result value to replace the contents of its left-hand side mutable storage location (a `mut` variable, struct field, or array cell):

  ```
  x = x + 5;
  array[i] = array[i-1] * 2;
  ```

  Note that the value copied depends on the type, as discussed in Section 4. Values of types `()`, `bool`, and `i64` are copied by value; values of other types are copied by reference.

- A function call, block, or control expression may be evaluated for its side effects.

  ```
  if (array[i] == 7) {
    println(greeting);
  ```

```
    }  // OK: no semicolon
    if (array[i+1] == 6) {
      println(object);
    }; // OK: semicolon
```

In the case of a block or control expression step (i.e., any step whose concrete syntax ends with a closing curly brace), the semicolon is optional

Blocks **end** in one of three ways:

- An explicit result expression: in this case, the result of evaluating this expression becomes the result value for evaluation of the entire block expression. For example, the following block evaluates to the result value 7:

```
    ({
      let mut x = 4;
      x = x + 1; // Stores 5 in x.
      x          // This is the result of the block expression.
    }) + 2        // This addition expression gives the result value of the function
                 // by adding the result of the inner block expression to 2.
```

- An explicit control directive: in this case, control is transferred elsewhere without providing a result value for evaluation of the block: `return` leaves the function, `break` leaves a loop, `continue` leaves the current loop iteration. Note: since these control directives terminate evaluation of the containing block, they may appear **only** at the end of their containing block. No other placement is allowed.
- Neither a result expression nor a control directive: in this case, the evaluation of the block concludes with unit `()` as its implicit result value.

6.2 Control-Flow

Control-flow operations `if`, `while`, and `match` are expressions that produce result value when evaluated. `if`, `else`, and `while` require their bodies to be block expressions. For example, the following function prints `"odd"` for odd numbers and `"even"` for even number arguments:

```
fn evenodd(x: i64) -> () {
  println(
    if (x % 2 == 0) { "even" }
    else { "odd" }
  )
}
```

The types of the expressions that form the result of the *then* and *else* branches must agree. When used without the optional `else` block, an `if` expression must yield type unit `()` in the *then* branch.

The `while` expression evaluates its body iteratively. At each iteration, it evaluates the test condition. If the condition is false, then it finishes the execution of the loop; otherwise it executes the loop body and continues with the next iteration. The `while` loop expression must have a body expression with type `()`; it yields the unit-typed value `()` as its result. `break` immediately terminates the loop without completing the current iteration; `continue` immediately transfers control to the loop test without completing the current iteration. Both `break` and `continue` may occur only within the body of a `while` loop. These loop control expressions always refer to the innermost loop.

The `return e` form evaluates the expression `e` and then immediately terminates evaluation of the containing function body, providing the result of `e` as the result value of the call. When used without an explicit expression, `return` behaves as `return ()`.

6.3 Pattern Matching Expressions (`match`)

Roost provides pattern matching with `match` expressions, similar to pattern matching support in Scala, ML, Rust, Swift, and others. Pattern matching can distinguish the variants of an enumeration, extract carried values from enumeration variants, and evaluate different expressions according to the specific target value. The `match` expression syntax consists of a *target expression* and several *cases*, separated by commas. Each case consists of a *pattern* and a corresponding result expression.

For example, the following `get_first_or` function uses `match` to take a linked list (represented with the `List` and `Node` types defined above), distinguish whether or not there is a first element, and return its first element, otherwise return the given `default` argument value if there are no elements in the list:

```
fn get_first_or(list: List, default: i64) -> i64
  match (list) {
    Empty => default,
    Link(node) => node.value,
  }
}
```

A `match` expression is evaluated by first evaluating its target expression (getting the value of variable `list` in this case) to get a target value, then matching this value against each of the case patterns in sequence until a matching pattern is found, then the corresponding result expression in the case is evaluated in an environment where all the pattern variables of the matching pattern are bound to the corresponding parts of the target value. If no pattern matches, a run-time error arises and evaluation is terminated. All cases of the `match` must have patterns of the same type as the target expression. There are a few types of patterns:

- **Literal patterns:** any unit, Boolean, integer, or string literal is a pattern that matches only identical values of the same type.
- **Wildcard pattern:** the pattern `_` matches any target value and may take on any type, without restriction, but does not introduce a binding.
- **Variable patterns:** any variable name. A variable pattern matches and binds to any target value and may take on any type, without restriction.
- **Enumeration variant patterns:** an enumeration variant with a carried pattern (instead of a carried value). A variant pattern has the same type as the enumeration that defines this variant. A variant pattern matches the target value if:
  1. The target value has the same variant as the pattern; and
  2. The variant pattern's carried pattern matches the target value's carried value and may introduce bindings.

For example, calling `get_first_or(Link(Node { value:  301, rest:  Empty }), 240)` will return `301`. It does this by matching the patterns against the `list` value in order:

1. The `Empty` pattern does not match the `list` value, since the `list` value has variant `Link`. Matching continues to the next case.
2. The `Link(node)` pattern matches the target value because:
   - The pattern and value have the same variant.
   - The subpattern, a variable `node`, trivially matches and binds to the carried `Node`-type value of the target variant.

Since the `Link(node)` pattern matches, the subpattern `node` introduces a new variable named `node` that holds the carried value of the target `Link` value. In this case, that is the `Node` with a `value` of `301`. Finally, the corresponding result expression for this case is evaluated: here, `node.value` uses the new `node` variable introduced by the matched pattern to refer to that `Node` in the target value and extract the value `301` from its `value` field. This value is the result of the entire `match` expression and the function call.

Priority	Operator	Description	Associativity
1	`[]  .  ()`	array index, field access/array length, function call	left
2	`-  !`	(unary) integer negation, Boolean/bitwise complement	right
3	`* / %`	multiplication, division, remainder	left
4	`+  -`	addition, subtraction	left
5	`<< >> >>>`	shifts	left
6	`&`	bitwise AND	left
7	`^`	bitwise exclusive OR	left
8	`|`	bitwise OR	left
9	`< <= > >=`	ordering comparison	left
10	`== !=`	(in)equality comparison	left
11	`&&`	short-circuit Boolean AND	left
12	`||`	short-circuit Boolean OR	left

Table 1: Precedence rules for Roost unary and binary operators. Priority 1 is the tightest.

6.4 Simple Expressions

Simpler expressions include:

- Computation expressions: unary and binary arithmetic and logic expressions (described in Section 6.5).
- Value constructor expressions: array, structure, and enumeration constructor expressions; and boolean, integer, string, and unit literals.
- Core expressions: locations (local variables, parameters, fields, or array elements); functions calls; and explicitly parenthesized expressions.

Note that the grammar in Figure 1 is structured to induce explicit restrictions on the composition of these expression forms to avoid ambiguity among these kinds of expressions.

6.5 Operators

Unary and binary operators include the following:

- Arithmetic operators: addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`. The operands must both be of type `i64`. Division/modulus by zero are dynamically checked, and cause run-time errors.
- Bitwise operators: AND `&`, OR `|`, XOR `^`. The operands must both be of type `i64`.
- Bit shift operators: shift left `<<`, arithmetic shift right `>>`, logical shift right `>>>`. The operands must both be of type `i64`. Shifts by distances of 64 or greater are undefined.
- String concatenation with `+`. The operands must both be of type `String`.
- Relational comparison operators: less than `<`, less than or equal to `<=`, greater than `>`, and greater than or equal to `>=`. Their operands must be integers.
- Equality comparison operators: equal `==` or different `!=`. The operands must have the same type. For integer and Boolean types, operand values are compared. For reference types, references are compared.
- Boolean operators: short-circuit AND, `&&`, and short-circuit OR, `||`. If the first operand of `&&` evaluates to false, its second operand is not evaluated. Similarly, if the first operand of `||` evaluates to true, its second operand is not evaluated. The operands must be of type `bool`.

- Unary operators: integer negation `-`, Boolean logical or bitwise complement `!`. (The meaning of `!` is determined by whether it is applied to a `bool` or an `i64` value.)

Operator precedence and associativity is defined by Table 1. These precedence levels and associativity rules resolve ambiguity among the unary and binary expression forms. The grammar in Figure 1 enforces precedence with respect to other expression forms.

7 Scope

For each program, there is a hierarchy of scopes consisting of: the top-level scope; the function body scopes including function parameters; and the local nested scopes for `let` bindings and `match` cases within functions.

7.1 Nested Scopes

When resolving an identifier at a certain point in the program, the enclosing scopes are searched from innermost to outermost until the first matching binding for the identifier is found. Identifiers can be used only if they are defined in one of the enclosing scopes. More precisely, variables can be used (read or written) only after they are defined in one of the enclosing `let` block scopes.

- The top-level scope consists of the names of all functions, structures, enumerations, and enumeration variants defined in the program.
- The scope of a function consists of its parameters.
- The scope of a `let` binding extends from immediately after the `let` binding (not covering the expression in the `let` binding itself) to the end of the containing block.
  - Later `let` bindings that define a variable name that is already in scope introduce a distinct new variable that *shadows* the earlier binding of this name. For example, this block evaluates to `7` using 3 distinct variables. Each `let` creates a new nested scope within the existing scope:

    ```
    {                                            // Scope 0:
      let x = 3 /* x is not in scope here */;    // Scope 1: x=3
              /* x is in scope here */
      let y = x /* here x evaluates to 3 */ - 1; // Scope 2: y=2,x=3
      let x = x /* here x evaluates to 3 */ + 2; // Scope 3: x=5,y=2,(shadowed)x=3
      y + x     /* here x evaluates to 5 */
    }
    ```

    The following block expression evaluates to `5`:

    ```
    {                                              // Scope 0:
      let x = 3 /* x is not in scope here */;      // Scope 1: x=3
              /* x is in scope here */
      ({
        let y = x /* here x evaluates to 3 */ - 1; // Scope 2: y=2,x=3
        let x = x /* here x evaluates to 3 */ + 2; // Scope 3: x=5,y=2,(shadowed)x=3
        y
      }) + x       /* here x evaluates to 3 */      // Scope 1: x=3
    }
    ```

  - Shadowing is distinct from assignment. While the following block also evaluates to `7`, it uses only 2 distinct variables:

    ```
    {                // Scope 0:
      let mut x = 3; // Scope 1: x=3
      let y = x - 1; // Scope 2: y=2,x=3
      x = x + 2;     //          y=2,x=5 (no new scope, mutated x from Scope 1)
    ```

```
  y + x
}
```

The following block expression also evaluates to 7:

```
{                  // Scope 0:
  let mut x = 3;   // Scope 1: x=3
  ({
    let y = x - 1; // Scope 2: y=2,x=3
    x = x + 2;     //          y=2,x=5 (no new scope, mutated x from Scope 1)
    y
  }) + x            // Scope 1: x=5 (y no longer in scope, same x still in scope)
}
```

- A `match` expression introduces a new scope for each case. All pattern variables in a case are in scope for the result expression of that case.

```
match (x) {
  Link(node) => node.value + 1 /* node is in scope until here */,
  Empty => 0                   /* node is not in scope here   */,
}
```

7.2 Flat and Type-Dependent Scopes

Field names in structure field access expressions such as `name` in `student.name` refer to fields declared for the type of structure that is being accessed (in this case the type of `student`). Type information is required to resolve references to field names. (Field names are **not** resolved in the same way as local variables.)

Program-defined type names introduced in the top-level scope by structure and enumerate type definitions can be used in types anywhere in the program, either before or after the point of reference. Similarly, top-level function names may be referenced in any function body, regardless of order of definition.

Unlike block scopes, where `let` introduces a new nested scope, top-level scopes and scopes within structure or enumeration definitions are *flat*. Identifiers cannot be defined multiple times in the same scope: two top-level functions may not share the same name; two fields of the same structure may not share the same name; two variants of an enumeration may not share the same name.

8 Type System

This section defines the type system for the Roost language.

8.1 Syntactic Sugar

For purpose of simplifying the formal typing and evaluation rules for Roost, we assume the following rules for *desugaring* certain Roost syntactic forms by rewriting them in terms of other simpler syntactic forms:

- **Blocks** are rewritten to ensure that every block expression contains exactly one explicit step followed by one explicit end item. For example:
  - The block `{ 3 }` is rewritten as `3`.
  - The block

```
{
  let x = 5;      // 1. step
  let y = 3;      // 2. step
  if (x < y) {
    x = !x;         // 3. step; 4. implicit end expression
  }               // 5. step
```

```
    x               // 6. end
  }
```

is rewritten as:

```
  {
    let x = 5;      // 1. single step
    {               // 1a. explicit end expression
      let y = 3;      // 2. single step
      {               // 2a. explicit end expression
        if (x < y) {
          x = !x;         // 3. single step
          ()              // 4. explicit end expression
        };              // 5. single step (with semicolon)
        x               // 6. original end expression
      }               // 2a. explicit end expression
    }               // 1a. explicit end expression
  }
```

  – **Returns** with no argument expression are rewritten as `return ()`.
  – **Enumeration variants** with no carried type/expression/pattern are rewritten to use unit `()` as their carried type/expression/pattern.

Note: these desugaring rules are purely for simplicity in defining the type system and operational semantics. It is not required to apply these transformations in implementing the language.

8.2 Type System Meta-Syntax and Environments

The formal definition of the type system uses additional formal meta-syntax which is not part of the Roost language meta-syntax for programs. Programs are notated "P"; type environments are notated "Γ"; types are notated "τ"; program-defined type and variant identifiers are notated "t"; other identifiers are notated "x"; expressions (and statements) are notated "e"; location expressions may also be notated "l"; immutability/mutability indicators are notated "m"; empty sequences (such as empty programs or environments) are notated "$\cdot$". Typing environments, Γ, map various identifiers and context indicators to their type information:

- Function and local variable identifier bindings, $x \mapsto m\ \tau$, indicating the immutability/mutability, $m \in \{\ \mathsf{imm}\ ,\ \mathsf{mut}\ \}$, and the type, τ, of the binding;
- Structure type definitions, $t \mapsto \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, indicating the name of the structure type (t) and the names and types of its fields;
- Enumeration type definitions, $t \mapsto \{t_1(\tau_1), \ldots, t_n(\tau_n)\}$, indicating the name of the enumerationg type, t, and the names and carried types of its variants; and
- Contexts indicators for:
  – The result type of the containing function, $\mathsf{result} \mapsto \tau$; and
  – The presence of a containing loop, loop.

The full grammar of the typing environment itself is:

$$\Gamma ::= \cdot \ \mid\ \Gamma, x \mapsto m\ \tau \ \mid\ \Gamma, t \mapsto \{(x : \tau)^*\} \ \mid\ \Gamma, t \mapsto \{(t(\tau))^+\} \ \mid\ \Gamma, \mathsf{result} \mapsto \tau \ \mid\ \mathsf{loop}$$

Typing environments support looking typing information for any identifier or context indicator with $\Gamma(x)$, $\Gamma(t)$, or $\Gamma(\mathsf{result})$. Lookups yield the value for the first mapping ($\mapsto$) whose key matches the requested identifer or context indicator, searching the environment from right to left. The right-to-left search is important to model shadowing for local variables. Lookup of $\Gamma(\mathsf{loop})$ succeeds if loop is present anywhere in the environment and fails otherwise; it does not provide a value.

8.3 Core Language Type System

Figure 2 shows top-level type-checking judgments for Roost language programs. Program P is well-typed, written $\boxed{\vdash P}$, if its top-level function and structure definitions define a type environment $\boxed{P \vdash \Gamma}$ that is well-formed $\boxed{\Gamma \vdash \Gamma}$ and under which the bodies of all of its function and structure definitions are well-typed $\boxed{\Gamma \vdash P}$. A type environment is well-formed if all of its internal type names refer to types defined in the environment. (Name well-formedness rules like this could be considered prerequisites to type checking, but we include them here for completeness.)

Figure 3 shows typing rules for expressions (and statements) $\boxed{\Gamma \vdash e : \tau}$. Bodies of functions are type-checked in a typing environment containing all declared function types and structure types plus the parameter and result types of the function definition. Figure 4 shows typing rules for patterns $\boxed{\Gamma \vdash p : \tau \Rightarrow \Gamma'}$ (used by the typing rule for match expressions).

8.4 The Never Type (`!`)

The *never* type, `!`, indicates the type of a value that will never actually exist. The type system allows items of type `!` to be used in place of any other type, since such use will never actually occur at run time.

For example, some functions never return.

```
// [Standard library function] Terminate the program. (Never returns.)
extern fn exit(code: i64) -> !;

// Infinite recursion
fn diverge() -> ! {
  diverge()
}
```

Functions that never return could just as well be given any other type, like `()` or `[String]`, but the *never* type makes them most flexible. If `exit` had return type `()`, this would not type-check: the *then* branch has type `i64`, but the `else` branch has type `()`; the types of the `if` branches must agree.

```
extern fn exit_unit(code: i64) -> ();
fn sample1(a: i64, b: i64) -> i64 {
  let x: i64 = if (a < b) {
    b            // type of then branch: i64
  } else {
    exit_unit(1) // never returns a result value (terminates the program)
                 // type of else branch: ()
  }; // ERROR: if-else branches have different types.
  a + b + x
}
```

Without the *never* type, the type system is unaware that `exit_unit` will not return and the *else* result will never be used. Resolving the type error requires an unpleasant workaround: add a bogus result expression of the correct type after the `exit_unit` call in the *else* branch. This bogus result is never used, but satisfies the type system.

```
fn sample2(a: i64, b: i64) -> i64 {
  let x: i64 = if (a < b) {
    b             // type of then branch: i64
  } else {
    exit_unit(1); // never returns a result value (terminates the program)
    0             // type of else branch: i64
  }; // type of if-else: i64
  a + b + x
}
```

$\boxed{\vdash P}$ Program P is well-typed.

$$\textsc{T-program}\quad \frac{P \vdash \Gamma \qquad \Gamma \vdash \Gamma \qquad \Gamma \vdash P}{\vdash P}$$

$\boxed{P \vdash \Gamma}$ Program P defines type environment Γ.

$$\textsc{T-def-empty}\quad \frac{}{\cdot \vdash \cdot}$$

$$\textsc{T-def-fn}\quad \frac{P \vdash \Gamma \qquad x \notin \Gamma}{P\ \texttt{fn}\ x\ \texttt{(}\ x_1 : \tau_1\ \texttt{,}\ \ldots\ \texttt{,}\ x_n : \tau_n\ \texttt{)}\ \texttt{->}\ \tau\ e \vdash \Gamma, x \mapsto \mathsf{imm}\ \mathsf{fn}(\tau_1, \ldots, \tau_n) \to \tau}$$

$$\textsc{T-def-struct}\quad \frac{P \vdash \Gamma \qquad t \notin \Gamma \qquad |\{x_1, \ldots, x_n\}| = n}{P\ \texttt{struct}\ t\ \texttt{\{}\ x_1 : \tau_1\ \texttt{,}\ \ldots\ \texttt{,}\ x_n : \tau_n\ \texttt{\}} \vdash \Gamma, t \mapsto \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

$$\textsc{T-def-enum}\quad \frac{P \vdash \Gamma \qquad \{t, t_1, \ldots, t_n\} \cap \Gamma = \emptyset \qquad |\{t, t_1, \ldots, t_n\}| = n + 1}{P\ \texttt{enum}\ t\ \texttt{\{}\ t_1\ \texttt{(}\ \tau_1\ \texttt{)}\ \texttt{,}\ \ldots\ \texttt{,}\ t_n\ \texttt{(}\ \tau_n\ \texttt{)}\ \texttt{\}} \vdash \Gamma, t \mapsto \{t_1(\tau_1), \ldots, t_n(\tau_n)\}}$$

$\boxed{\Gamma \vdash \Gamma'}$ Type environment Γ' is well-formed using type definitions from type environment Γ.

$$\textsc{T-env-empty}\quad \frac{}{\Gamma \vdash \cdot} \qquad \textsc{T-env-bind}\quad \frac{\Gamma \vdash \Gamma' \qquad \Gamma \vdash \tau}{\Gamma \vdash \Gamma', x : \tau} \qquad \textsc{T-env-def}\quad \frac{\Gamma \vdash \Gamma' \qquad \forall i \in 1..n, \Gamma \vdash \tau_i}{\Gamma \vdash \Gamma', t \mapsto \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

$\boxed{\Gamma \vdash \tau}$ Type τ is well-formed under type environment Γ.

$$\textsc{T-type-basic}\quad \frac{\tau \in \{\,\mathsf{()}, \mathsf{bool}, \mathsf{i64}, \mathsf{String}, \mathsf{!}\,\}}{\Gamma \vdash \tau} \qquad \textsc{T-type-array}\quad \frac{\Gamma \vdash \tau}{\Gamma \vdash\ \texttt{[}\,\tau\,\texttt{]}} \qquad \textsc{T-type-nominal}\quad \frac{\Gamma(t) = \{\ldots\}}{\Gamma \vdash t} \qquad \textsc{T-type-fn}\quad \frac{\forall i \in 1..n, \Gamma \vdash \tau_i \qquad \Gamma \vdash \tau}{\Gamma \vdash \mathsf{fn}(\tau_1, \ldots, \tau_n) \to \tau}$$

$\boxed{\Gamma \vdash \mathsf{mut}\ l}$ Location l is mutable under type environment Γ.

$$\textsc{T-mut-var}\quad \frac{\Gamma(x) = \mathsf{mut}\ \tau}{\Gamma \vdash \mathsf{mut}\ x} \qquad \textsc{T-mut-array}\quad \frac{}{\Gamma \vdash \mathsf{mut}\ e_1[e_2]} \qquad \textsc{T-mut-field}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{mut}\ e.x}$$

$\boxed{\Gamma \vdash P}$ Program P is well-typed under type environment Γ.

$$\textsc{T-empty}\quad \frac{}{\Gamma \vdash \cdot} \qquad \textsc{T-fn}\quad \frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n, \mathsf{result} : \tau \vdash e : \tau}{\Gamma \vdash\ P\ \texttt{fn}\ x(x_1 : \tau_1, \ldots x_n : \tau_n)\ \texttt{->}\ \tau\ e} \qquad \textsc{T-struct}\quad \frac{}{\Gamma \vdash P\ \texttt{struct}\ t\ \{\ldots\}} \qquad \textsc{T-enum}\quad \frac{}{\Gamma \vdash P\ \texttt{enum}\ t\ \{\ldots\}}$$

Figure 2: Roost type environments (core language).

$\boxed{\Gamma \vdash e : \tau}$ Expression e has type τ under type environment Γ.

T-TRUE
$$\frac{}{\Gamma \vdash \texttt{true} : \mathsf{bool}}$$

T-FALSE
$$\frac{}{\Gamma \vdash \texttt{false} : \mathsf{bool}}$$

T-I64
$$\frac{}{\Gamma \vdash \textsc{Integer} : \mathsf{i64}}$$

T-STRING
$$\frac{}{\Gamma \vdash \textsc{String} : \mathsf{String}}$$

T-UNIT
$$\frac{}{\Gamma \vdash \texttt{()} : \mathsf{()}}$$

T-ARITH
$$\frac{\Gamma \vdash e_1 : \mathsf{i64} \qquad \Gamma \vdash e_2 : \mathsf{i64} \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{/}, \texttt{*}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{>>>}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\}}{\Gamma \vdash e_1 \; op \; e_2 : \mathsf{i64}}$$

T-MINUS
$$\frac{\Gamma \vdash e : \mathsf{i64}}{\Gamma \vdash \texttt{-}e : \mathsf{i64}}$$

T-BIT-NOT
$$\frac{\Gamma \vdash e : \mathsf{i64}}{\Gamma \vdash \texttt{!}\,e : \mathsf{i64}}$$

T-ORD
$$\frac{\Gamma \vdash e_1 : \mathsf{i64} \qquad \Gamma \vdash e_2 : \mathsf{i64} \qquad op \in \{\texttt{<=}, \texttt{<}, \texttt{>=}, \texttt{>}\}}{\Gamma \vdash e_1 \; op \; e_2 : \mathsf{bool}}$$

T-EQ
$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad op \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash e_1 \; op \; e_2 : \mathsf{bool}}$$

T-BOOL
$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \mathsf{bool} \qquad op \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \; op \; e_2 : \mathsf{bool}}$$

T-BOOL-NOT
$$\frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \texttt{!}e : \mathsf{bool}}$$

T-NEW-ENUM
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma(t_2) = \{\ldots, t_1(\tau_1), \ldots\}}{\Gamma \vdash t_1\texttt{(}e_1\texttt{)} : t_2}$$

T-NEW-STRUCT
$$\frac{\forall i \in 1..n, \Gamma \vdash e_i : \tau_i \qquad \Gamma(t) = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}{\Gamma \vdash t \; \texttt{\{} x_1 \texttt{:} e_1 \texttt{,} \; \ldots \; \texttt{,} x_n \texttt{:} e_n \texttt{\}} : t}$$

T-FIELD
$$\frac{\Gamma \vdash e : t \qquad \Gamma(t) = \{\ldots, x : \tau, \ldots\}}{\Gamma \vdash e \, \texttt{.} \, x : \tau}$$

T-ARRAY-LIST
$$\frac{\forall i \in 1..n, \Gamma \vdash e_i : \tau}{\Gamma \vdash \texttt{[}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{]} : \texttt{[}\tau\texttt{]}}$$

T-ARRAY-FILL
$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathsf{i64}}{\Gamma \vdash \texttt{[}e_1\texttt{;}e_2\texttt{]} : \texttt{[}\tau\texttt{]}}$$

T-ARRAY-LENGTH
$$\frac{\Gamma \vdash e : \texttt{[}\tau\texttt{]}}{\Gamma \vdash e\texttt{.length} : \mathsf{i64}}$$

T-CELL
$$\frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma \vdash e_2 : \mathsf{i64}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau}$$

T-LET-IMM
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x_1 \mapsto \mathsf{imm}\ \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{\{ let } x_1 \texttt{:} \tau_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}$$

T-LET-MUT
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x_1 \mapsto \mathsf{mut}\ \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{\{ let mut } x_1 \texttt{:} \tau_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}$$

T-VAR
$$\frac{\Gamma(x) = m\ \tau}{\Gamma \vdash x : \tau}$$

T-LET-IMM-INFER
$$\frac{\Gamma \vdash \texttt{\{ let } x_1 \texttt{:} \tau_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}{\Gamma \vdash \texttt{\{ let } x_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}$$

T-LET-MUT-INFER
$$\frac{\Gamma \vdash \texttt{\{ let mut } x_1 \texttt{:} \tau_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}{\Gamma \vdash \texttt{\{ let mut } x_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}$$

T-EFFECT
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{\{} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}$$

T-IFELSE
$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if (} e_1 \texttt{)} \; e_2 \; \texttt{else} \; e_3 : \tau}$$

T-ASSIGN
$$\frac{\Gamma \vdash \mathsf{mut}\ l_1 \qquad \Gamma \vdash l_1 : \tau_1 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{\{} l_1 \texttt{=} e_1 \texttt{;} e_2 \texttt{\}} : \tau_2}$$

T-IF
$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \mathsf{()}}{\Gamma \vdash \texttt{if (} e_1 \texttt{)} \; e_2 : \mathsf{()}}$$

T-WHILE
$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma, \mathsf{loop} \vdash e_2 : \mathsf{()}}{\Gamma \vdash \texttt{while (} e_1 \texttt{)} \; e_2 : \mathsf{()}}$$

T-BREAK
$$\frac{\mathsf{loop} \in \Gamma}{\Gamma \vdash \texttt{break} : \texttt{!}}$$

T-CONTINUE
$$\frac{\mathsf{loop} \in \Gamma}{\Gamma \vdash \texttt{continue} : \texttt{!}}$$

T-CALL
$$\frac{\Gamma \vdash e_0 : \mathsf{fn}(\tau_1, \ldots, \tau_n) \to \tau \qquad \forall i \in 1..n, \Gamma \vdash e_i : \tau_i}{\Gamma \vdash e_0 \texttt{(} e_1 \texttt{,} \; \ldots \; \texttt{,} e_n \texttt{)} : \tau}$$

T-RETURN
$$\frac{\Gamma \vdash \mathsf{result} : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return} \; e : \texttt{!}}$$

T-NEVER
$$\frac{\Gamma \vdash e : \texttt{!}}{\Gamma \vdash e : \tau}$$

T-MATCH-CASE
$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash p_1 : \tau_0 \Rightarrow \Gamma' \qquad \Gamma' \vdash e_1 : \tau_1 \qquad \Gamma \vdash \texttt{match (} e_0 \texttt{) \{ ... \}} : \tau_1}{\Gamma \vdash \texttt{match (} e_0 \texttt{) \{} \; p_1 \texttt{=>} e_1\texttt{,} \; \ldots \; \texttt{\}} : \tau_1}$$

T-MATCH-EMPTY
$$\frac{\Gamma \vdash e : \tau_0}{\Gamma \vdash \texttt{match (} e \texttt{) \{ \}} : \texttt{!}}$$

Figure 3: ROOST typing rules for expressions (core language).

$\boxed{\Gamma \vdash e : \tau \Rightarrow \Gamma'}$ Pattern p matches values of type τ under type environment Γ to yield environment Γ'.

$$\text{T-PAT-LIT-UNIT}\;\frac{}{\Gamma \vdash \; () \; : () \Rightarrow \Gamma} \qquad \text{T-PAT-LIT-BOOL}\;\frac{}{\Gamma \vdash \textsc{Bool} : \mathsf{bool} \Rightarrow \Gamma} \qquad \text{T-PAT-LIT-I64}\;\frac{}{\Gamma \vdash \textsc{Integer} : \mathsf{i64} \Rightarrow \Gamma} \qquad \text{T-PAT-LIT-STRING}\;\frac{}{\Gamma \vdash \textsc{String} : \mathsf{String} \Rightarrow \Gamma}$$

$$\text{T-PAT-VAR}\;\frac{}{\Gamma \vdash x : \tau \Rightarrow \Gamma, x \mapsto \tau} \qquad \text{T-PAT-WILD}\;\frac{}{\Gamma \vdash \_ : \tau \Rightarrow \Gamma} \qquad \text{T-PAT-VARIANT}\;\frac{\Gamma(t_2) = \{\ldots, t_1(\tau_1), \ldots\} \qquad \Gamma \vdash p : \tau_1 \Rightarrow \Gamma'}{\Gamma \vdash t_1\;(\,p\,)\;:\;t_2 \Rightarrow \Gamma'}$$

Figure 4: Roost typing rules for patterns (core language).

Using the *never* type, `!`, as the return type for functions that never return allows expressions such as `if` to be well-typed when some, but not all, branches result in such a function call, since an expression of type `!` can take on any other type. In `sample3`, the *else* branch has type `!`, so it may also take on type `i64` to match the type of the *then* branch.

```
fn sample3(a: i64, b: i64) -> i64 {
  let x: i64 = if (a < b) {
    b       // type of then branch: i64
  } else {
    exit(1)  // never returns a result value (terminates the program)
            // type of else branch: !
  }; // type of if-else: i64
  a + b + x
}
```

For the same reason, uses of `return`, `break`, and `continue` in local expression have the *never* type:

```
fn sample4(a: i64, b: [i64], c: [i64]) -> String {
  let x: [i64] = if (0 < a && a < b.length) {
    b            // type of outer then branch: [i64]
  } else {
    if (0 < a && a < c.length) {
      c          // type of inner then branch: [i64]
    } else {
      return "No" // type of "No" matches type of function result
                 // type of inner else branch: !
    } // type of inner if-else / outer else branch: [i64]
  }; // type of outer if-else: [i64]
  x[0] = a;
  "Yes"
}
```

8.4.1 Soundness and the Never Type

Even mutable storage locations may safely take the type `!`. All locations (variables, array cells, struct fields) as well as enum carried values must be initialized upon creation. The type enforces that evaluation of an expression with this type never completes. This means that construction of structs with never-typed fields and enums with never-typed carried data never happens at run time. No instances of such types ever exist. Similarly, never-type variables are never bound at run time. Thus, while rules T-ASSIGN and T-NEVER allow a mutable never-typed location to be treated as a mutable *any*-typed location, initialization and all and assignments to that location are unreachable, due to the meaning of the never type.

9 Core Library

The Roost core library provides a small set of functions for I/O operations, basic type conversions, and other system-level functionality. These functions are in scope for all programs.

```
// Read one byte from standard input, zero-extend to i64.
extern fn readbyte() -> i64;
// Write the low order byte of an i64 to standard output.
extern fn writebyte(byte: i64) -> ();

// Read one line from standard input, up to the next newline or EOF.
extern fn readln() -> String;

// Print string to standard output.
extern fn print(string: String) -> ();
// Print string to standard output, followed by a newline.
extern fn println(string: String) -> ();
// Print integer to standard output, in decimal (base 10) notation.
extern fn printi64(integer: i64) -> ();

// Parse the decimal (base 10) notation of an integer from a string
// and return the value as i64, or return the fallback value if
// parsing fails.
extern fn parsei64(string: String, fallback: i64) -> i64;
// Generate and return a string with the decimal (base 10) notation of
// an i64 value.
extern fn dumpi64(integer: i64) -> String;

// Return the number of characters in the given string.
extern fn string_length(string: String) -> i64;
// Produce a new string by concatenating the two given strings.
extern fn string_concat(left: String, right: String) -> String;

// Return a random number in the range 0 through bound-1, inclusive.
extern fn random(bound: i64) -> i64;

// Terminate the program with the given exit code.
extern fn exit(code: i64) -> !;
// If the given condition is false, terminate with message.
extern fn assert(condition: bool, message: String) -> ();
```

TypeParams	::=	‘<’ TypeId (‘,’ TypeId)* ‘>’
TypeArgs	::=	‘<’ *Type* (‘,’ *Type*)* ‘>’
Item	::=	. . . | *Module* | *TypeAlias* | *Use*
Module	::=	‘`mod`’ Id ‘`{`’ (‘`pub`’? *Item*)* ‘`}`’
TypeAlias	::=	‘`type`’ TypeId *TypeParams*? ‘`=`’ *Type* ‘`;`’
Use	::=	‘`use`’ *Path* TypeId ‘`;`’
	|	‘`use`’ *Path* Id ‘`;`’
	|	‘`use`’ *Path* ‘`*`’ ‘`;`’
Path	::=	‘`::`’ | *Path*? Id ‘`::`’ | *Path*? TypeId ‘`::`’
Pattern	::=	. . .
	|	TypeId ‘`{`’ (Id ‘`:`’ *Pattern* ‘`,`’)* (Id ‘`:`’ *Pattern*)? ‘`}`’

Figure 5: Syntax of standard extensions to the Roost language for generics and modules.

10 Standard Extensions

For improved language ergonomics, the Roost language defines standard extensions beyond the core language. Figure 5 summarizes the extended syntax by showing additions to the core language grammar defined in Figure 1.

10.1 Generic Types

The Roost type system supports *parametric polymorphic types*, more commonly known as *generic types*. Structure definitions, enumeration definitions, and function definitions may introduce *generic type parameters* that may be used as types within the bodies of those definitions. ConstructionCalling generically typed structures or enumerations and calling generically typed functions instantiates *type arguments* in place of these formal type parameters.

10.1.1 Generic Structure and Enumeration Definitions

This typing extension supports type parameters on structure definitions and allows types within the structure definition to reference these type parameters in type annotations. For example, generalizing the integer linked list structure defined earlier:

```
enum List<T> {
  Link(Node<T>),
  Empty
}
struct Node<T> {
  value: T,
  rest: List<T>,
}
```

The above structure definition defines `List<T>` and `Node<T>` types for representing linked lists with elements of any consistent type, `T`. After generalizing this definition, the only other necessary changes to code using the earlier `i64`-specific definitions are to change any uses of `List` and `Node` *in type annotations* to `List<i64>` and `Node<i64>`. Changes to constructors are not needed: The type checker infers an appropriate type argument for `T` whenever constructing `List` or `Node`.

10.1.2 Generic Function Definitions

Function definitions may also introduce type parameters in order to make function behavior generic over types. The function parameter types, result type, and any types in the function body may reference this the function's type parameters. For example, for any type `T`, the following function takes an element of type `T` and a `ListNode` that carries the same element type `T` and returns a new `ListNode` with the new element prepended to the beginning of the list:

```
fn cons<T>(elem: T, list: List<T>) -> List<T> {
  Link(Node { value: elem, rest: list })
}
```

Calls to functions with type parameters instantiate the type parameters with type arguments implicitly. The type arguments are inferred from the types of the argument expressions. For example, in the following code, `T` is inferred to be type `str` for this particular call to `cons`:

```
let names: List<String> = ...;
let nickname: String = ...;
names = cons(nickname, names);  // Type checker infers <T=String> for this call.
```

10.1.3 Generic Type System Definition

As the course did not come close to exploring this feature, formal definitions and informal descriptions for the generic type system were not added.

10.2 Module System and Abstract Types

As the course did not come close to exploring this feature, definitions and descriptions for the module system and abstract types were not added.

11 Document Status

- Language Version: 0.2.0
- Document Revision: 10

The language version (0.2.0) indicates the version of language; it changes when an aspect of the language itself is updated, e.g., to add or change a feature. The document revision indicates the version of this document; it changes whenever this document is changed, e.g., to fix a typo, add further description, etc. A change to the document does not imply a change to the language. A change to the language requires a change to the document.

11.1 Change Log (Document Revisions)

1. *<2021-04-20 Tue>* Initial CS 301 Spring 2021 version of this document published 20 April 2021. Roost version 0.2.0 is a breaking update with respect to the (retroactively re-versioned) Roost 0.1.0 definition from CS 301 Spring 2019.

2. *<2021-04-24 Sat>* Fixed typo, added discussion of type soundness for the *never* type, added introductory text about generic types.

3. *<2021-04-26 Mon>* Remove explicit `.length` form from grammar to match Section 4.3 and the reference implementation, which treat it as a field access expression.
4. *<2021-05-05 Wed>* Fix typo in the *Structure* production of Figure 1.
5. *<2021-05-08 Sat>* Add typing rules for `match` and patterns. Fix minor related type system typos.
6. *<2021-05-10 Mon>* Add core library.
7. *<2021-05-10 Mon>* Update Rule T-MUT-FIELD to ensure that only struct fields (and not array lengths) are mutable.
8. *<2021-05-12 Wed>* Remove `readend` from the core library, change `readbyte` to zero-extend.
9. *<2021-05-27 Thu>* Fix typos with block expressions as subexpressions of binary expressions. (Change from old syntax to current syntax.)
10. *<2021-06-01 Tue>* Close off extensions that the course never had time to reach.

11.2 Pending Additions

As the course did not reach close to these items, they were never added:

- Full documentation for the standard extensions for generic types and modules and abstract types.
- Discussion of the `extern` ABI.
- A standard library (extending the minimal core library).