# Contents

1 Overview 2

1.1 Core Language Highlights ........................................ 2
1.2 Standard Extensions ........................................... 2
1.3 Status .............................................................. 2
1.4 Acknowledgments ................................................. 3

2 Lexical Considerations ........................................... 3

2.1 Identifiers ......................................................... 3
2.2 Keywords .......................................................... 3
2.3 Tokenization ....................................................... 3
2.4 Comments .......................................................... 3
2.5 Literals .............................................................. 3

3 Top-Level Program Elements ....................................... 4

4 Variables ............................................................ 4

5 Data Types .......................................................... 4

5.1 Scalar Types ........................................................ 4
5.2 Strings ............................................................... 4
5.3 Arrays ............................................................... 5
5.4 Structure Types (struct) and Objects ............................ 5

6 Function Calls ....................................................... 6

7 Scoping Rules ....................................................... 6

8 Expressions .......................................................... 6

8.1 Block Expressions .................................................. 7
8.2 Control-Flow Expressions .......................................... 7
8.3 Assignment .......................................................... 7
8.4 Simple Expressions ................................................ 7
8.5 Operators ............................................................ 8

9 Standard Extensions to the Core Language ................. 9

9.1 Parametric Type System ........................................ 9
9.2 Methods, Signatures, and Subtyping ............................ 9
9.3 First-Class Function Closures .................................... 12
9.4 Interactions ........................................................ 12

10 ROOST Syntax ...................................................... 12

10.1 Precedence .......................................................... 12
10.2 Syntactic Sugar ..................................................... 14

11 Type System .......................................................... 14

11.1 Core Language Type System ..................................... 14
11.2 Type System for Standard Extensions ....................... 17
1 Overview

For the CS 301 implementation project, you will build a compiler for ROOST, a statically typed language invented for this course. The core ROOST language provides functions, primitive data types, simple compound data structures, scoping, and basic control flow. ROOST takes some syntactic cues from Rust. A set of extensions to the core includes features partly familiar from other languages such as Java, Scala, ML, and countless others. The full ROOST language is simple enough that its grammar fits on one page, yet sophisticated enough to require many interesting considerations in a compiler.

1.1 Core Language Highlights

The core ROOST language requires:

- **Sound static typing**: The core ROOST language has a sound static type system with: builtin types for integer, Boolean, and string values; compound structure types; array types; and function types.

- **Top-level functions as values**: The core ROOST language supports basic higher order functions, allowing top-level functions to be passed, stored, and returned as values. The core language lacks nested function definitions and closures.

- **Dynamic allocation and garbage collection**: The core ROOST language supports dynamic heap allocation of objects, strings, and arrays. Such structures are always allocated on the heap and manipulated by reference. The language allows for garbage collection for automatic de-allocation of heap space.

- **Run-time checks**: ROOST supports run-time checks for null references, array bounds violations, and negative array size allocations.

1.2 Standard Extensions

The implementation project requires implementing at least one, preferably two, or optionally all three standard extensions to the core ROOST language:

- **Parametric type polymorphism**: The full ROOST language extends the static type system with parametric polymorphism, similar to Java or Scala generic types.

- **Subtype polymorphism with dynamic dispatch**: The full ROOST language supports structure methods for objects with subtype polymorphism via signature types, similar to Java interfaces, using dynamic dispatch for method calls. ROOST supports reuse via composition, rather than extension; it does not support inheritance, overriding, or overloading.

- **First-class function closures**: The full ROOST language supports functions as values, higher order functions, and first-class function closures for nested function definitions.

1.3 Status

This is a living document that will grow to include more detail as we progress into the next stage of the project. Changes to the language are not expected, but improvements to this document are. Features associated with standard extensions are highlighted in the colors shown above.
1.4 Acknowledgments

This document, aspects of the ROOST language, and related assignments were adapted from assignment materials developed at Cornell University and Williams College.

2 Lexical Considerations

2.1 Identifiers

Identifiers (names) and keywords are case-sensitive. Identifiers must begin with a letter. Following the initial letter may be any sequence of letters characters, digits, or the underscore character (_). Uppercase and lowercase letters are both considered are distinguished, so x and X are different identifiers.

2.2 Keywords

The following are the keywords used in the language and cannot be used as identifiers:

```plaintext
fn i64 bool str unit struct impl field method sig with let in if else while return break continue new length self true false null Roost
```

The following keywords are reserved for potential future use in the language:

```plaintext
class extends interface implements data datatype
```

Other lexical tokens are shown in the full syntax of ROOST in Figure 1.

2.3 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 or an identifier. For instance, `elsey` represents a single identifier, not the keyword `else` followed by the identifier `y`.

2.4 Comments

C/Java-style comments are supported. A comment beginning with the characters `//` indicates that the remainder of the line is a comment. A block comment is a sequence of characters that begins with `/*`, followed by any characters, including newline, up to the matching end sequence `*/`. An unclosed comment is a lexical error. ROOST also supports nesting ML-style comments, delineated by `(*` and `*)`. For example:

```
(* This is a single (* comment that does not end here -> *)
It (* ends (* after *) the matching star-rparen *) right here -> *)
```

2.5 Literals

Integer literals may start with an optional negation sign `-`, followed 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.

String literals 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.

The keywords `true` and `false` are Boolean literals. The null reference literal is `null`. The `unit`-type literal is actually a pair of tokens, ( followed by ). The `unit` type is similar to `void` in C or Java, except that there is a single value of type `unit`.
3 Top-Level Program Elements

A program consists of a sequence of function, structure, and signature definitions, including exactly one main function definition for the program entrypoint, of the following form:

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

ROOST functions are declared with fn keyword. The function definition lists the types of parameters and return results and provides a body expression to be evaluated when the function is called.

The main function takes an array of strings (command-line arguments) as its single argument and returns the value of type unit. ROOST structures are program-defined compound types. Each structure (struct) definition declares the set of named data fields (field) carried by each object instance of that structure type.

The parametric type system extension (generic types) supports type parameters on function definitions and structure definitions. The subtyping extension supports methods (method) in structure definitions, making them similar to classes in languages like Java or Scala (but without inheritance), and signature types for structures, analogous to Java interfaces. Each structure definition may optionally implement (impl) one or more signatures. Each signature (sig) declares required method types and may optionally require other signatures (with).

4 Variables

Program storage locations may be local variables or parameters of methods (allocated in the stack or registers), fields of objects (allocated in the heap), or cells of arrays (also allocated in the heap). Variables of type int and bool hold integer and boolean values, respectively. Variables of other types hold references to heap-allocated items (i.e., strings, arrays, or objects).

The program does not initialize variables by default when they are declared. Instead, the static checks in the compiler verify that each variable is guaranteed to have a value assigned before being used. Object fields and array elements are initialized with default values (0 for integers, false for booleans, and null for references) when such structures are dynamically created.

The language allows variables to be initialized when declared in a let block. The initialization expression can refer to variables in enclosing scopes, parameters of the enclosing function or method, or any names introduced by preceding bindings in the same let block. In other words, each new binding introduces its own nested scope. The function closure extension supports local function definitions in let blocks as well.

5 Data Types

5.1 Scalar Types

Booleans are represented by the type bool and the literals true and false. The integer type i64 is a 64-bit two’s-complement integer. Arithmetic operations on integers that overflow yield modular results. The unit type has a single value, ()

5.2 Strings

For string references, the language uses a primitive type str (unlike Java, where String is a class). Strings are allocated in the heap and are immutable, meaning that the program cannot modify their contents. The language allows only the following operations on string values:

- assignments of string references (including null);
- concatenating strings with the + operator; and
- testing for string reference equality using == and != (Note: this operator does not compare string contents). Built-in functions are provided to convert integers to strings, etc.
5.3 Arrays

The language supports arrays with arbitrary element types. If \( T \) is a type, then \( T[] \) is the type for an array with elements of type \( T \). In particular, array elements can be arrays themselves, allowing programmers to build multidimensional arrays. For instance, the type \( T[][] \) describes a two-dimensional array constructed as an array of array references of type \( T[] \).

Arrays are created dynamically using the `new` construct: `new T[n]` allocates an array of type \( T \) with \( n \) elements and initializes the elements with their default values. The expression `new T[n]` yields a reference to the newly created array. Arrays of size \( n \) are indexed from 0 to \( n-1 \) and the familiar bracket notation is used to access array elements. If the expression \( a \) is a reference to an array of length \( n \), then \( a.length \) evaluates to \( n \), and \( a[i] \) evaluates to the \((i+1)\) element in the array. For each array access \( a[i] \), the program checks at run-time that \( a \) is not null and that the access is within bounds: \( 0 \leq i < n \). Violations will terminate the program with an error message.

5.4 Structure Types (struct) and Objects

Roost structure types are analogous to Java’s class types, with some differences. Structure type definitions are collections of field definitions that define the contents and type of individual structure instances. In the core Roost language, structure type definitions are of the form:

```roost
struct A {
    field x1: T1
    /* ... */
    field xn: Tn
}
```

Field definitions, such as `field spots: i64`, declare the name and type of fields carried by each instance of the containing structure type. Roost fields are similar to Java instance variables.

A fresh object of a structure type is constructed by an expression of the form: `new A()`. This expression allocates and initializes space for the object on the heap and yields a reference to the object. Fields are initialized to hold their zero-most value: \( 0 \) for `i64`, `false` for `bool`, \( () \) for `unit`, and `null` for all reference types. Object fields are accessed using the `.` symbol. The expression `o.f` denotes the field `f` of object `o`.

Object references have structure types: each definition `struct A` introduces a structure type `A`. Structure types can then be used in declarations for local variables, parameters, or fields. For example, `field obj: A` declares a structure field `obj` of type `A`, that is, a reference to an object of structure type `A`.

A structure name `A` can be used as the type of an object reference anywhere in the program. In particular, it can appear in the body of a `struct A` declaration itself, or even before that, as in the following example. This admits recursive and mutually recursive structures, such as those below:

```roost
struct List {
    field elem: i64
    field next: List
}
struct Node {
    field data: str
    field edge: Edge[]
}
struct Edge {
    field label: i64
    field dest: Node
}
```

The **parametric type extension** supports type parameters on structure definitions for use in declarations within the structure definition body. The **subtyping extension** supports methods in structure definitions and introduces signature types, analogous to Java interfaces.
6 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 program evaluates the expressions representing the arguments and then binds the resulting values to the corresponding parameters of the function. Object, array, or string arguments are passed as references. Arguments are evaluated left to right.

After binding parameters to values, the program executes the body of the function that was called. When evaluation reaches a return expression or the end of the body, the program transfers control back to the caller. If the return expression has an expression argument or the body ends with an expression, that expression’s evaluated value is returned to the caller as the result.

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. If a function body has no result expression, it implicitly returns the value of type unit; this must match its declared result type. Otherwise, the result expression – and the subexpression of any return in the body – must be compatible with the declared result type of the function.

7 Scoping Rules

For each program, there is a tree hierarchy of scopes consisting of: the top-level scope; the function scopes or the structure scopes and their method scopes in the subtyping extension; and the local scopes for let blocks within each function or method. The top-level scope consists of the names of all functions, structures, and signatures defined in the program. The scope of a structure is the set of fields and methods of that structure. The scope of a function or method consists of the parameters. Finally, a let scope contains all of the variables defined between let and in within the expression. Each definition in the let expression introduces its own scope, allowing later definitions in the same let expression to use – or shadow – the names introduced by earlier definitions. The scope of the let body expression is the scope containing all the definitions introduced with the let. When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier after the local scope.

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. Structure elements can be used in expressions of the form expr.f when the object target expression expr has structure type T and the scope of T contains those elements. This means that all structure elements are publicly visible and can be accessed from any scope, including outside the declaring structure definition. Finally, structure names can be used in types of parameter, variable, and field declarations anywhere, provided they are defined in the program, either before or after the point of reference. Similarly, top-level functions may be referenced in any function body, regardless of order of definition.

Identifiers (names of functions, structures, signatures, fields, methods, and variables) cannot be defined multiple times in the same scope. Otherwise, identifiers can be defined multiple times in different, possibly nested, scopes. For variables, inner scopes shadow outer scopes. If variables with the same name occur in nested scopes, then each occurrence of that variable name refers to the variable in the innermost scope that defines it and contains the reference.

8 Expressions

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

ROOST organizes let blocks (for variable declarations and scoping) and control-flow operations (if, while) using block expressions. All blocks (code sequences in functions or methods delineated by curly braces { }) any sequence of simply expressions (with semicolons for sequencing) or block expressions, followed by an optional final result block or simple expression (with no semicolon). For example, the following function returns 7 when called:

```rust
fn f() -> i64 {
    let x: i64 = 4 in {
        x = x + 1; // Stores 5 in x. The expression result is (), but is discarded by ‘;’
        x // This is the result of the let expression.
    } + 2 // This addition expression gives the return result of the function
        // by adding the result of the let expression to 2.
}
```

8.2 Control-Flow Expressions

ROOST provides control-flow operations if and while as block expressions. The last expression in a block expression determines the result value of the entire block expression. For example, this function prints "odd" for odd numbers and "even" for even number arguments:

```rust
fn evenodd(x: i64) -> unit {
    Roost.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 executes 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 unit typed body expression; it yields the unit typed value as its result. The break expression immediately terminates the loop without completing the current iteration; the continue expression immediately moves control to the loop test without completing the current iteration.

The break and continue expressions may occur only within the body of a while loop. These loop control expressions always refer to the innermost loop.

The return e expression evaluates the expression e and then immediately terminates evaluation of the containing function or method body, providing the result of e as the result value of the call.

8.3 Assignment

Each assignment expression l = e updates the location represented by l with the value of expression e. The updated location l can be a local variable, a parameter, a field, or an array element. The type of the updated location must be compatible with the type of the evaluated expression. For integers and booleans, the assignment copies the integer or boolean value. For string, array, or object types, the assignment copies the reference. The result of the assignment expression itself is of type unit.

8.4 Simple Expressions

Simpler expressions include:

- locations: local variables, parameters, fields, or array elements;
- calls to methods with non-void return types;
### Table 1: Precedence rules for ROOST operators. Priority 1 is the highest (tightest binding).

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[] ( ) .</td>
<td>array index, method call field/method access</td>
<td>left</td>
</tr>
<tr>
<td>2</td>
<td>- ! ~</td>
<td>unary minus, logical negation, bitwise complement</td>
<td>right</td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>multiplication, division, remainder</td>
<td>left</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>addition, subtraction</td>
<td>left</td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</td>
<td>shift</td>
<td>left</td>
</tr>
<tr>
<td>6</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>relational operators</td>
<td>left</td>
</tr>
<tr>
<td>7</td>
<td>== !=</td>
<td>equality comparison</td>
<td>left</td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td>bitwise and</td>
<td>left</td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td>bitwise exclusive or</td>
<td>left</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>bitwise or</td>
</tr>
<tr>
<td>11</td>
<td>&amp;&amp;</td>
<td>short-circuit and</td>
<td>left</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>=</td>
<td>assignment</td>
<td>right</td>
</tr>
</tbody>
</table>

- new structure/object or array instances, created with `new T()` or `new T [e]`;
- the array length expression `e.length`;
- unary or binary expressions;
- integer, string, unit, and null literals; and
- any expression enclosed in parentheses, to make operator precedence explicit.

### 8.5 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 by zero and modulus of zero are dynamically checked, and cause program termination.
- Bitwise operators: “and” `&`, “or” `|`, exclusive “or” `^`. 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`.
- String concatenation with `+`. The operands must both be of type `str`.
- Relational comparison operators: less than `<`, less or equal than `<=`, greater than `>`, and greater or equal then `>=`. 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.
- Conditional 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 `-` for integers, logical negation `!` for booleans, bitwise complement `~` for integers.

The operator precedence and associativity is defined by Table 1.
9 Standard Extensions to the Core Language

9.1 Parametric Type System

The Roost type system supports parametric polymorphic types.

9.1.1 Type Parameters on Structures

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:

```plaintext
struct ListNode<T> {
    field value: T
    field next: ListNode<T>
}
```

The above structure definition defines a `ListNode<T>` type for representing linked lists with elements of any consistent type, `T`. When instantiating a `ListNode`, the type parameter `T` must be supplied with an explicit type argument for this specific instance. For example, `new ListNode<Cow>()` constructs a new `ListNode` that holds `Cow` elements.

9.1.2 Type Parameters on Functions

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 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:

```plaintext
fn cons<T>(elem: T, list: ListNode<T>) -> ListNode<T> {
    let head = new ListNode<T>() in {
        head.next = list;
        head.value = elem;
        head
    }
}
```

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

```plaintext
let herd: ListNode<Cow> = ...
    cow: Cow = ... in {
        cons(cow, herd)
    }
```

9.2 Methods, Signatures, and Subtyping

This extension introduces some features commonly associated with object-oriented languages.

9.2.1 Structure Methods

The subtyping extension supports method definitions in structures that declare the method name, parameter names and types, result type, and body of methods. These methods are available for invocation on each instance of the containing structure type with syntax similar to field access. For example:
struct Cow {
  field spots: i64
  method moo(enthusiasm: i64) -> unit {
    Roost.println("I have " + Roost.dumpi64(self.spots) + " spots.");
    Roost.print("Moo");
    let i = enthusiasm in
    while 0 < i {
      Roost.print("Oo");
      i = i - 1;
    }
    let i = enthusiasm in
    while 0 < i {
      Roost.print("!");
      i = i - 1;
    }
    Roost.println("!");
  }
}

fn main(args: str[]) -> unit {
  let cow = new Cow() in {
    cow.spots = 13;
    // Call the moo method on this 13-spotted cow.
    cow.moo(32);
  }
}

While otherwise similar to functions, methods are distinguished by providing a reference to the structure instance on which they are invoked, using the implicitly available name self (like Java’s this). ROOST methods are similar to Java instance methods. In all scopes, fields and methods must be accessed with a qualified name (explicit use of the self keyword: self.f, not f).

Unlike Java classes, ROOST structure types do not support defining initialization code to run as a “constructor” for an object instance. However, a simple convention makes it nearly as easy: in each structure type, define a method of the form:

struct S {
  field number: i64
  field boolean: bool
  method init(x: i64, y: bool) -> {
    // initialize fields
    self.number = x;
    self.boolean = y;
    self
  }
}

Then in place of the Java expression new A(3, false), use the ROOST expression new A().init(3, false).

ROOST structures do not support inheritance.

9.2.2 Signatures and Subtyping

ROOST signature types define abstract types without implementations, similar to Java interfaces, by declaring a set of method names and types, but no method body implementation. Structure definitions may use the optional impl clause to declare that they define methods compatible with all method types in one or more parent signatures:
The `impl C` clause indicates that structure `A` defines methods matching all the method names and compatible types declared in the `signature` `C`. The type system allows the `impl` relationship only if `B` satisfies this relationship with `C`.

Signatures may optionally use a `with` clause to require that any implementing structures implement all methods declared by this signature as well as one or more parent signatures. Structures and signatures may use `impl` or `with` clauses to refer to other signatures only if those other signatures are declared earlier in the program than the declaration that uses them. This simplifies the compiler implementation and makes it trivial to avoid cyclical `with` dependencies. Outside the `impl/with` restriction, signature names can be referenced from any scope, regardless of the order of definitions.

Object references have structure or signature types: each definition `struct A` or `sig B` introduces a structure type `A` or signature type `B`, respectively. Structure types can then be used in declarations for local variables, parameters, for fields. For example, `field obj: A` declares a structure field `obj` of type `A`, that is, a reference to an object of structure type `A`. The field declaration `field ect: B` similarly declares a structure field `ect` of type `B`, that is, a reference to any object whose structure type implements the signature `B`.

Method overloading is not supported: a structure or signature (and all of its ancestor signatures) cannot have multiple methods with the same name, even if the methods have different numbers or types of arguments, or different return types.

### 9.2.3 Subtyping

Implementing or including a signature induces a subtyping relation. When structure `A` implements signature `B` (`struct A impl B`), `A` is a subtype of `B`, written `A <: B`. When signature `B` includes signature `C` (`sig B with C`), `B` is a subtype of `C`, written `B <: C`. Subtyping is also reflexive and transitive.

\[
\begin{align*}
A \text{ impl } B \{\ldots\} & \quad A \text{ with } B \{\ldots\} & \quad A <: B \quad A <: C \\
A & <: B & A & <: B & A & <: B \\
A & <: C & A & <: C & A & <: C
\end{align*}
\]

If `A` is a subtype of `B`, a value of type `A` can be used in the program whenever the program expects a value of type `B`. Subtyping is invariant for array types and type parameters. The type `A[]` is never a subtype of `B[]` for distinct types `A` and `B`. Specifically, even if `A <: B`, `A[]` is not a subtype of `B[]`.

Structure methods that implement a signature method (as well as signature methods that override a signature method of the same name in a parent signature) may do so by using argument types that are contravariantly subtyped and return types that are covariantly subtyped. For example, assuming `A <: B`, the following is well typed:

```plaintext
sig E { method m(a: A) -> B }
struct D impl E { method m(a: B) -> A { new A() } }
```

Function types follow similar rules. Assuming `A <: B`, `(B) -> A <: (A) -> B`.

### 9.2.4 Method Dispatch

The actual method being invoked by a method call expression `o.m()` cannot always be determined statically, because the concrete type of target object `o` may be unknown. When the static type of the target expression
is a structure type, the method to be invoked is known statically. When the static type of the target is a signature type, the actual type of the object is not known statically, and the method to be invoked may be the corresponding method of any structure type implementing the signature. Method calls on signature types are resolved at run time via dynamic dispatch.

9.3 First-Class Function Closures

The core ROOST language supports passing and saving functions as values, but only supports defining functions in the top-level program scope. The first-class function closure extension additionally supports binding new named function definitions in `let` expressions. Nested function definitions, like the rest of the language, are lexically scoped, meaning that a nested function must capture the variables in scope at definition time in its closure. When defined within a method scope, a function closure must capture the `self` reference in addition to explicit bindings.

9.4 Interactions

When multiple relevant standard extensions are supported:

- Signatures may declare type parameters in the same way as structures; types in the signature definition may reference these type parameters.

- Types in structure methods and signature method declarations may reference the type parameters of the enclosing structure or signature scope.

- Like functions, methods may declare and reference their own type parameters. The syntax is identical.

- When implementing or including a signature with type parameters using `impl` or `with`, the implementing structure or including signature must provide type arguments to instantiate those parameters:

  ```
  sig Siggy<T> { /* ... */ }
  struct Structy impl Siggy<Cow> { /* ... */ }
  struct Strooct<A> impl Siggy<A> { /* ... */ }
  ```

- Subtyping of parametric types is invariant in the type argument: even if `A <: B`, the type `Vector<A>` is \textit{not} a subtype of `Vector<B>`.

10 ROOST Syntax

The language syntax is shown in Figure 1. Here, keywords are shown using monospace font (e.g., `while`); operators and punctuation symbols are shown using single quotes (e.g., `'`); the other terminals are written using small caps fonts (ID, INTEGER, and STRING); and nonterminals using slanted fonts (e.g., `expr`). The remaining symbols are meta-characters: `( ... )` and `( ... )^+` denote the Kleene star and plus operations, respectively, and `( ... )^?` denotes an optional sequence of symbols.

10.1 Precedence

Figure 1 defines an ambiguous grammar. Table 1 defines precedence for operators. Additionally:

- The optional `else` binds to the innermost `if`;

- In a `block` where a `blockExpr` is followed by another `expr` starting with `~-` or `~` without an intervening `;`, the `blockExpr` and the following `expr` are individual expressions in sequence within the block, \textit{not} a single call or subtraction expression. As an example, this function returns `-1`, not `0`:

  ```
  fn f() -> i64 {
    let x = 1 in { x }
    -1
  }
  ```
Figure 1: ROOST Syntax. Core language uses a plain background. Standard extensions are highlighted: parametric types, structure methods and signature types; and first-class function closures.
10.2 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:

- **Multi-binding let expressions**: Each \( n \)-binding `let` expression of the form:
  
  ```
  let x1 = e1
  x2 = e2
  /* ... */
  xn = en in {
    /* body */
  }
  ```

  is desugared to a nesting of \( n \) single-binding `let` expressions:
  
  ```
  let x1 = e1 in
  let x2 = e2 in
  /* ... */
  let xn = en in {
    /* body */
  }
  ```

- **Block sequence expressions**: Block expressions that appear in a sequence without a semicolon are desugared to use an explicit semicolon. For example, `{ if (x) { x = !x; () } x }` is desugared to `{ if (x) { x = !x; () }; x }`.

- **Block result expressions**: Blocks with no result expression are rewritten to include an explicit result expression of `()`, the unit value. For example, the block `{ x = e; }` is desugared to `{ x = e; () }`.

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

11 Type System

This section defines the type system for the Roost language. Programs are notated “\( P \)”; type environments are notated “\( \Gamma \)”; types are notated “\( \tau \)”; non-type identifiers are notated “\( x \)”; expressions (and statements) are notated “\( e \)”; location expressions may also be notated “\( l \)”; empty sequences (such as empty programs or environments) are notated “\( \cdot \)”.

11.1 Core Language Type System

Figure 2 shows top-level type-checking judgments for Roost language programs. Program \( P \) is well-typed, written \( \Gamma \vdash P \), if its top-level function and structure definitions define a type environment \( P \vdash \Gamma \) that is well-formed \( \Gamma \vdash \Gamma \) and under which the bodies of all of its function and structure definitions are well-typed \( \Gamma \vdash P \). 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 programs \( \Gamma \vdash P \) and expressions (and statements) \( \Gamma \vdash e \). 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. In anticipation of the standard extensions, the core language typing rules use the subtype relation \( \tau_1 :<: \tau_2 \). For the core language on its own, the subtype relation is simply type equality, that is, \( \tau_1 <: \tau_2 \) is equivalent to \( \tau_1 = \tau_2 \).
⊢ P  Program $P$ is well-typed.

$\begin{array}{ll}
T\text{-}\text{PROGRAM} \\
P \vdash \Gamma \quad \Gamma \vdash \Gamma \quad \Gamma \vdash P \\
\vdash P
\end{array}$

$\vdash P$  Program $P$ defines type environment $\Gamma$.

$\begin{array}{ll}
T\text{-}\text{DEF-EMPTY} \\
\vdash \cdot
\end{array}$

$\begin{array}{ll}
T\text{-}\text{DEF-FN} \\
P \vdash \Gamma \quad (x : \tau') \notin \Gamma \\
P \text{fn } x(x_1 : \tau_1, \ldots x_n : \tau_n) \to \tau \vdash \Gamma, x : (\tau_1 \times \ldots \times \tau_n) \to \tau
\end{array}$

$\begin{array}{ll}
T\text{-}\text{DEF-STRUCT} \\
P \vdash \Gamma \quad (\tau \mapsto \{x'_1 : \tau'_1, \ldots x'_m : \tau'_m\}) \notin \Gamma \\
\quad \quad |\{x_1, \ldots, x_n\}| = n \\
P \text{struct } \tau \{x_1 : \tau_1 \ldots x_n : \tau_n\} \vdash \Gamma, \tau \mapsto \{x_1 : \tau_1, \ldots, x_n : \tau_n\}
\end{array}$

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

$\begin{array}{llll}
T\text{-}\text{ENV-EMPTY} & T\text{-}\text{ENV-BIND} & T\text{-}\text{ENV-DEF} \\
\Gamma \vdash \cdot & \Gamma \vdash \Gamma' \quad \Gamma \vdash \tau \\
& \Gamma \vdash \Gamma', x : \tau & \forall i \in 1..n, \Gamma \vdash \tau_i
\end{array}$

$\vdash \tau$  Type $\tau$ is well-formed under type environment $\Gamma$.

$\begin{array}{llll}
T\text{-}\text{TYPE-SCALAR} & T\text{-}\text{TYPE-ARRAY} & T\text{-}\text{TYPE-NOMINAL} & T\text{-}\text{TYPE-FN} \\
\tau \in \{i64, \text{bool, str, unit}\} & \Gamma \vdash \tau & (\tau \mapsto \{x_1 : \tau_1, \ldots, x_n : \tau_n\}) \in \Gamma \\
& \Gamma \vdash \tau \; \text{[]} & \Gamma \vdash \tau \\
& \forall i \in 1..n, \Gamma \vdash \tau_i & \Gamma \vdash (\tau_1 \times \ldots \times \tau_n) \to \tau
\end{array}$

Figure 2: ROOST type environments (core language).
Program $P$ is well-typed under type environment $\Gamma$.

Expression (or statement) $e$ has type $\tau$ under type environment $\Gamma$.

Figure 3: Roost typing rules (core language).
11.2 Type System for Standard Extensions

Under construction! Tutorials on 12 March will focus on type system support for these extensions.

12 Semantics

Under construction!

13 Builtin Functions

Roost provides a small set of builtin functions for I/O operations, basic type conversions, and other system-level functionality. The following builtin functions are predefined implicitly in all Roost programs. Types and documentation of these functions are shown in Table 2.

<table>
<thead>
<tr>
<th>Roost Builtin Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roost.println(s: str) -&gt; unit</td>
<td>prints string s followed by a newline</td>
</tr>
<tr>
<td>Roost.print(s: str) -&gt; unit</td>
<td>prints string s</td>
</tr>
<tr>
<td>Roost printk64(i: i64) -&gt; unit</td>
<td>prints integer i</td>
</tr>
<tr>
<td>Roost.readk64() -&gt; i64</td>
<td>reads one character from the input</td>
</tr>
<tr>
<td>Roost.readlk() -&gt; str</td>
<td>reads one line from the input</td>
</tr>
<tr>
<td>Roost.eof() -&gt; bool</td>
<td>checks end-of-file on standard input</td>
</tr>
<tr>
<td>Roost.parsek64(s: str, n: i64) -&gt; i64</td>
<td>returns the integer that s represents or n of s is not an integer</td>
</tr>
<tr>
<td>Roost.dumpk64(i: i64) -&gt; str</td>
<td>returns a string representation of i</td>
</tr>
<tr>
<td>Roost.ascii(s: str) -&gt; i64[]</td>
<td>an array with the ascii codes of chars in s</td>
</tr>
<tr>
<td>Roost.string(a: int[]) -&gt; str</td>
<td>builds a string from the ascii codes in a</td>
</tr>
<tr>
<td>Roost.random(n: i64) -&gt; i64</td>
<td>returns a random integer in the range 0 and n-1, inclusive</td>
</tr>
<tr>
<td>Roost.time() -&gt; i64</td>
<td>returns number of milliseconds since program start</td>
</tr>
<tr>
<td>Roost.exit(n: i64) -&gt; unit</td>
<td>terminates the program with exit code n</td>
</tr>
</tbody>
</table>

Table 2: Builtin functions.

To invoke builtin functions, the program must use method calls qualified with the Roost name; for instance, Roost.random(100) or Roost.parsei64("301",0). For simplicity, it is not required to allow builtin functions to be passed as values. For each explicit call to Roost.f, the compiler should generate a procedure call to Roost_f in the assembly output.
14 Change Log

• 2019-02-15: minor revisions in version 1
  – Removed optional explicit type arguments in call expression.
  – Clarified precedence to resolve ambiguity of - in sequences of expressions under blocks.
  – Allowed all expressions (simple and block) to be explicitly sequenced with ; in blocks.

• 2019-02-20: version 2
  – Allow calling expressions.
  – Restrict sequencing to use block or effect expressions.
  – Revise treatment of ambiguity in sequencing within blocks.
  – Differentiate function type syntax from declaration syntax.
  – Ban unqualified field/method references. All field and method references must use self in all scopes.

• 2019-03-01: version 3
  – Allow let*-style blocks: initializer expressions may refer to preceding bindings in the same let block.
  – while yields a unit type result.
  – Eliminate the misguided break e and continue e forms.
  – Include return in control-flow discussion.
  – Add type system for core language.
  – Include discussion of standard extensions to the core language.

• 2019-03-02: version 4
  – Fix typos in type system.
  – Define and describe rules for type environments.

• 2019-03-02: version 5
  – Miscellaneous typo fixes and updates of outdated text.
  – Remove outdated scoping restrictions.

• 2019-03-04: version 6
  – Fix typos in T-assign, T-field, T-let.