Contents

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

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 ..................................................... 3

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 ........................................................... 8
  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 .............................................. 9

10 ROOST Syntax .................................................................... 9

11 Typing Rules ........................................................................ 9

12 Builtin Functions .................................................................. 9

13 Change Log ......................................................................... 9
  13.1 2019-02-15 ................................................................. 9
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. Any exceptions to that rule are highlighted in orange to show that a particular element may change. None of the potential changes is expected to effect early stages. 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:

```
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:

```
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:
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 (or to the parameters of the enclosing function or method), but cannot involve any of the variables declared in the current block. 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 and method definitions that define the contents, behaviors, 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 final expressions in the body. If a function body has no final expression, it implicitly returns the value of type `unit`; this must match its declared result type. Otherwise, the final expression – and the subexpression of any `return` in the body – must match 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` block scope contains all of the variables defined at the beginning of that block. When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier after the local scope.

There are a couple of scope rules. First, identifiers can only be used if they are defined in one of the enclosing scopes. More precisely, variables can only be used (read or written) 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 class type `T` and the scope of `T` contains those elements. This means that all structure elements (fields and methods) are publicly visible and can be accessed from any scope, including outside the declaring structure definition. Finally, function and structure names can be used in parameter, variable, and field declarations anywhere, provided they are defined in the program, either before or after the point of reference.

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. Finally, local variables are not allowed to shadow method parameters; local variables must have different names than the parameters of the enclosing function.

Examples to come soon!

8 Expressions

Like Scala, ML, or Rust, and unlike Java, C, or Python, 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 only; there are no “statements.” 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 `()`, the unit 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 block or simple expression (with no semicolon). For example, the following functions both return 7:

```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.
}
fn f() -> i64 {
    let x : i64 = 4 in {
        x = x + 1;
        return x + 2
    }
}
```

If you are used to statement-based languages, you may not notice much of a difference, as the sort of code you would write with statements also works in this semantics.

8.2 Control-Flow Expressions

ROOST provides control-flow operations if and while as block expressions, with the same kind of semantics: the last 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 yields the result of the final expression in their body as evaluated in the final iteration. 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 e and continue e expressions both take a subexpression, e, to yield as a result of the entire while loop upon breaking or upon continuing but finding that the loop test condition evaluates to false. All exit paths from a while loop must have result expressions of the same type. The break and continue expressions may occur only within the body of a while loop; they always refer to the innermost loop.

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 match 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 (), unit.

8.4 Simple Expressions

Simpler expressions include:

- locations: local variables, parameters, fields, or array elements;
- calls to methods with non-void return types;
- 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: sign change `-` for integers, logical negation `!` for booleans, bitwise complement `~` for integers.

The operator precedence and associativity is defined by the table below. Here, priority 1 is the highest (tightest binding).

One additional ambiguity in the grammar is resolved with precedence as follows. This function has as its body a block containing a single subtraction expression, not a sequence of a ‘let’ expression followed by a `-1` expression.

```rust
fn f() -> unit {
    let x = 1 in {
        x
    } -1
}
```
<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</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>( )</td>
<td>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>

9 Standards Extensions to the Core Language

9.1 Parametric Type System
Under construction!

9.2 Methods, Signatures, and Subtyping
Under construction!

9.3 First-Class Function Closures
Under construction!

10 ROOST Syntax
The language syntax is shown in Figure 1. Here, keywords are shown using typewriter fonts (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: (...) denotes the Kleene star operation and (...)^2 denotes an optional sequence of symbols.

11 Typing Rules
Under construction!

12 Builtin Functions
Under construction!

13 Change Log

13.1 2019-02-15
- Removed optional explicit type arguments in call expression.
program ::= (fn | struct | sig)^*  

fn ::= fn Id typeParams argResultType block  
argResultType ::= | typeParams argResultType | (fn Id typeArgs) ^  
typeParams ::= | <typeParams, type> | (fn Id typeArgs) ^  
typeArgs ::= | <type, type> | (fn Id typeArgs) ^  
type ::= i64 | bool | str | unit | Id | type [type] | argResultType | type typeArgs  

struct ::= struct Id typeParams (impl (Id (typeArgs)^) *) ^  
field ::= field Id type  
method ::= method Id argResultType block  
sig ::= sig Id typeParams (with (Id (typeArgs)^) *) ^  
block ::= blockExpr | simpleExpr  
blockExpr ::= let (Id (type) ^ = expr | Id type | fn)^ in blockExpr  
| if expr blockExpr (else blockExpr)^  
| while expr blockExpr  
| block  
simpleExpr ::= literal  
| return expr  
| break expr | continue expr  
| location  
| call  
| expr binop expr  
| unop expr  
| (expr binop)  
| new type [type] | new Id (typeArgs)^  
| expr type  
| self  

literal ::= INTEGER | STRING | true | false | (expr) | null  
location ::= Id | expr [expr] | expr type | Id  
call ::= Roost (Id (expr) ^)  
| (expr type)  
binop ::= + | - | * | / | % | & | ^ | || | <= | < | <= | > | >= | == | !=  
| << | >>= | >>> | & | | | |  
unop ::= - | ! | ~  

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.
• 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.