MiniC Language Specification

October 10, 2018

This document describes the abstract and concrete syntax and semantics of a small C-like programming language (albeit with a very different syntax for declarations).

1 Basics

The fundamental storage unit is the *byte*, which is a sequence of 8 bits. The storage (or memory) available to a program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

The following paragraphs describe the entities of the language. An *entity* is an abstract element of a program that is described by, created by, or referred to in a program.

A *value* is element of a set. For example the value zero is an element in the set of integers and the set of real numbers. Similarly, the value true is an element in the set boolean values; the other value is false. A value can also be an address. When this document refers to specific values, they are spelled in English using the normal font.

A *type* describes objects, references, functions, and expressions. The meaning of the type is determined by the the entity it describes.

An *object* is region of storage, comprised of a fixed length sequence of bytes. Note that every object has an address. Every object has a type, which determines the size of the object and how its stored bits can be interpreted as a value (i.e., an object stores a value). Objects are created by definitions or when a temporary object is created.

A *reference* can be thought of as an alias (name) for an object or alternatively as an address for the object. Every reference has a type, which determines the set of objects to which the reference can bind. It is unspecified whether a reference requires storage. References are created by definitions.

A *function* maps a sequence of input values, called *arguments*, to an output value, called its *return value*. Every function has a type, which determines the arguments that it can accept as inputs the type of value that it returns as an output. Functions are created by definitions.

2 Abstract syntax and semantics

This section describes the *abstract syntax* of the language, which is comprised of four sub-languages: types (t), expressions (e), statements (s), and definitions (d). Each sub-language is defined (recursively) as a set of strings belonging to that set. Note that these are "strings" in the abstract sense; they will be represented as abstract syntax trees.

The meaning of each string is determined by the semantic specifications in this document. The semantic specification determines several properties of the language:

- the *static semantics* of the language define properties of strings that can be used at compile-time (e.g., the type of expressions).
- the *dynamic semantics* of the language define properties of strings that are meaningful at runtime (e.g., the value of expression).

Static semantics are often used as an additional "filter" on syntactically valid strings that would result in misbehaving programs. Dynamic semantics are used to define when the execution of a program is defined. For example, the expression `1 / n` is valid if and only if `n` refers to an object of type `int` (statically). The evaluation of that expression is defined if the value of that object is never 0 (dynamically).

A *program* is a sequence of definitions (substrings of d); that is, it is also a string.

2.1 Objects

The basic unit of storage is the *byte*, which is a sequence of 8 bits.

The memory available to a program consists of one or more sequences of contiguous, where every byte has a unique address.

An *object* is created by a object definitions and occupies a region of storage during its lifetime. Note that every object is has a name and a type.

A *reference* is an alias to an object. References are created by reference definitions and are bound to the object designated by the initializer.

2.2 Types

Types describe objects, references and functions.

t	::=	**bool**	boolean values
		int	integer values
		float	floating point values
		ref t_1	references to objects
		$(t_1, t_2, \ldots, t_n) \to t_r$	functions

The type **bool** describes the values true and false.

The type **int** describes integer values in the left-open range $[-2^{32-1}, 2^{32-1})$.

The type **float** describes single precision IEEE 754 floating point values.

The types **bool**, **int**, and **float** are collectively called the *object types*, meaning they can be used to define objects.

The reference types **ref** t describes references to objects. These value can be represented as the address of the object in memory.

The set of function types $(t_1, t_2, \ldots, t_n) \to t_r$ describes functions taking n arguments, whose corresponding types are t_1, t_2, ..., and t_n, and returning a value of type t_r. These values can be represented as the address of the function in memory.

2.3 Expressions

The language has the following kinds of expressions:

e	::=	**true**		$e_1 \leq e_2$	less or equal
		false		$e_1 \geq e_2$	greater or equal
		n	integer literals	$e_1 + e_2$	addition
		r	floating literals	$e_1 - e_2$	subtraction
		x	identifiers	$e_1 * e_2$	multiplication
		e_1 **and** e_2	logical and	e_1 **div** e_2	quotient of division
		e_1 **or** e_2	logical or	e_1 **rem** e_2	remainder of division
		not e_1	logical negation	$-e_1$	negation
		if e_1 **then** e_2 **else** e_3	conditional	$/e_1$	reciprocal
		$e_1 = e_2$	equal to	$e_1 \leftarrow e_2$	assignment
		$e_1 \neq e_2$	not equal to	$e_f(e_1, e_2, \ldots, e_n)$	function call
		$e_1 < e_2$	less than		
		$e_1 > e_2$	greater than		

An expression is a sequence of operands and operators that specifies a value computation. The evaluation of an expression results in a value. The type of an expression determines a) how expressions can be combined to produce complex computations and b) the kind of value it produces. The following paragraphs define the requirements on operands and the result types of each expression as well the values they produce.

The order in which an expression's operands are evaluated is unspecified unless otherwise noted.

The expressions **true** and **false** have type **bool** and the values true and false, respectively.

Integer literals have type **int**. The value of an integer literal is the one indicated by its spelling.

Floating point literals have type **float**. The vale of a floating point literal is the one indicated by its spelling.

2.4 Statements

The language has the following kinds of statements:

s	::=	**break**	break statements
		continue	continue statement
		$\{s_1, s_2, \ldots, s_n\}$	compound statements
		while e_1 **do** s_1	while statements
		if e **then** s_1 **else** s_2	if statements
		return e	return statements
		e	expressions
		d	local definitions

2.5 Declarations

The language has the following kinds of declarations:

d	::=	**var** $x : t = e$	object definitions
		ref $x : t = e$	reference definitions
		fun $x(d_1, d_2, \ldots, d_n) \to t\ s$	function definitions