Cardigan Language Reference Manual

Josh Lopez (jl3497) -- Project Leader
Nithin Chandrasekharan (nc2470) Muzi Gao (mg3194) Miriam Melnick (mrm2198)

1. Introduction

This document describes the Cardigan language. There are seven sections to this manual:

1. This introduction
2. Tokens -- a description of each class of tokens
3. Types -- a brief description of each primitive and derived data type, including syntax
4. Scoping -- information regarding scoping of variables
5. Statements -- the structure of a line, or lines, of code in a Cardigan program
6. Expressions -- parts of statements which need to be evaluated during execution
7. Writing programs -- Correct program structure and a list of built in functions.

2. Tokens

There are 8 classes of tokens: identifiers, keywords, boolean constants, integer constants, float constants, string literals, operators, and separators. Tab and space characters are considered whitespace, and any combination of these are treated as a single whitespace character. Two consecutive tokens in a single statement must be separated by whitespace unless one of the two is either an operator or a separator, or they are the last and first tokens of two consecutive statements. Comments are never tokenized and are ignored by the compiler.

2.1 Comments

Comments are single-line and and begin with double slashes. Any characters between // and the end of line would be ignored.
For example, a valid comment would be like:

```
// Here is a comment
```

2.2 Separators

There are three types of separators: token separators, line separators, and grouping separators.

2.2.1 Token separators

Token separators consist of any combination of one or more tab and space characters. These are not treated as tokens, but are used to separate tokens from each other. The compiler can make distinction between operator tokens and other types of tokens so operators and their operands need not be separated by spaces. Grouping separators are distinguishable from every

type of token, including other grouping separators, so they never need to be separated from other tokens. Although it is not required, a program can include any number of token separators before or after an operator.

2.2.2 Newline separator

The newline character is used as a line separator. All of the tokens on a single line must comprise a single statement to be syntactically valid. Multi line statements may be created by ending each line with a backslash character. These will be interpreted as a single line, and therefore, a single statement.

2.2.3 Grouping separator

Grouping separators are used to create groups of tokens. The grouping separators are: left and right parenthesis, left and right square bracket, left and right curly brace, the pipe character, commas, and colons. Their uses are discussed in their relevant sections.

2.3 Identifiers

An identifier is a sequence of letters, digits, and underscores(_). The first character must be a letter or underscore. Identifiers are case sensitive, which means “foo” and “Foo” are different. The compiler will truncate identifiers to the first 16 characters if they are any longer.

2.4 Keywords

Keywords are simple terms that are used throughout the program for purposes specified by its definition. Unlike identifiers or constants, keywords cannot be replaced by random names or values.
The keywords defined here are generally used in many languages, but a few (rules, Play) are given different names, terms related to card games.

int, float, string, bool	(TYPE DECLARATIONS)
if, elseif, else, while, for, ?	(CONTROL STRUCTURES)
true, false	(BOOLEAN CONSTANTS)
return	(FUNCTION EVALUATION)
CAST, PLAY, INPUT, OUTPUT	(BUILT-IN FUNCTIONS)

3. Types

Cardigan is strongly typed, but types are determined at declaration.
Cardigan has four following primitive types: bool, int, float, and string, and three derived types: structs, collections, and enums.

3.1 Primitive types

3.1.1 Boolean Constants

A boolean constant represents either boolean true or boolean false, represented by the `true` and `false` keywords respectively.

3.1.2 Integer Constants

An integer constant is a sequence of one or more digits without a decimal point. The range of integers is 0 to 2^{31}-1. Octal or hexadecimal representations of integers are not supported.
For example, 42 is a valid integer constant, but 4.2 is not.

3.1.3 Float Constants

A float constant consists an integer part, and a decimal part. The integer part is a sequence of one or more digits. The decimal part is a decimal point followed by zero, one or more digits. The interpretation of floating-point literals that fall outside the range of representable floating-point values is undefined.
For example, 42.7, 42. are both valid float constants, but 42 would be recognized as an integer constant.

3.1.4 String literals

A string is a sequence of printable characters. A string must be surrounded by double-quote characters (`"`). Special characters must be escaped using the backslash character. The following special characters are permitted:

`\n` newline
`\t` tab
`\"` doublequote
`\\` backslash

3.2 Derived Types

Derived types include structs, collections, enums, and functions.

Structs are objects that divide other objects into fields. Structs can be used to simulate the role of hash tables or classes.
Collections are objects similar to lists, and can even be modeled as arrays, stacks, queues, and linked lists. They are usually used to create lists of game elements.

Enums are used to declare a pseudo-type, with definite values that are immutable, and are used mainly for comparisons.

3.2.1 Enums

An enum is an immutable structure which allows programmers to create sets of values which may be used in comparisons. The values of an enum are meant to be used as their own data type exclusive to their particular enum. Enum values may be used in equality and inequality comparisons, but they may not be the operands of any arithmetic operations or greater than/ less than comparisons.

An enum is declared using the following syntax: identifier = | identifiers |
The first identifier will be used as the pseudotype for the enum. The identifiers between the vertical bars must be a comma separated list of one or more identifiers. These become the values of the enum. Accessing the value of the enum is done through dot access. The syntax for enum dot access is:
identifier.identifier
where the first identifier is the name of the enum and the second is the name of a value of the enum.

3.2.2 Collections
A collection is a list of values. Each term in a collection may be any type of expressions that can be defined in the program. Collections may also contain other collections.

Collections in Cardigan, are created using the syntax:
identifier = [values]
where values is a comma separated list of zero or more expressions. These expressions will be stored in the collection.

Elements of the collection are accessed with the syntax:
identifier[index]
where identifier refers to the collection, and index is an integer expression. This syntax allows you to access any element of the collection.

Collections also support several other access methods which allow them to be used as a linked list, stack, queue.

append(x) - Adds x to the end of the collection.
push(x) -- Adds x to the head of the collection.
pop() - Removes the first element of the collection and returns it.
peek() -- Returns but does not remove the first element of the collection
popLast() -- Removes and returns the last element of the collection.

3.3.3 Struct
Structs are declared with curly braces and may contain zero or more comma separated assignments. Within a struct an assignment is written as:

identifier: expression

Identifier must be a valid identifier as described above. expression may be any valid expression. The value of the expression will be stored in the struct under the specified identifier, and may be accessed later using dot notation. If the expression is an identifier to a function, the property of the struct will become an alias to the function and may be called like a function.

In addition, structs may be inherited. The name of an existing struct may be prepended to the curly braces which causes the values in the named struct to be copied over to the new struct. Any assignments contained within the inheriting struct override those of the existing struct. Structs can, therefore, be used in a similar manner as classes in other languages, with inheritance taking the place of instantiation.

4. Scoping

In Cardigan, a complete program should be written in one file, and will be compiled at the same time.

A function can be called anywhere inside the file, but other identifiers can only be referred after being declared. The scope of a parameter of one function starts from the declaration of the function, and persists through the function. The value of a parameter can be reassigned within its scope as long as the types match.

If an identifier is declared in one block, it can only be referred from its declaration to the end of this block. Any other identifier declared directly in the file has a scope starting from its declaration to the end of the file. Any reference outside the block will raise an error. Similarly, reassignment of value is allowed within the scope only if the types match.

5. Statements

A statement is a complete line of code. There are several valid forms for a statement. A statement can be any valid expression (see below for details), a variable assignment, a control structure (one of: if, while, for, or ?), or a function definition.

5.1 Assignments

Assignment statements are used to assign a value to a variable. The syntax for an assignment statement is:

identifier = expression

If the identifier references a variable that has not been assigned previously, the expression may be any expression of any type. If the variable has already been declared, the expression must evaluate to the same type that the variable had before the assignment statement. Note that function definitions have a similar syntax, but are not the same.

5.2 Increments

Increment statements are unary operations that do not return a value. The syntax of increment statements is:

identifier operator

Any identifier with an integer or float value, including an element of a collection or a property of a struct may be used. The operators are `++` and `--`. The increment operator `++` adds the value of 1 (for integer operands) or 1.0 (for float operands) to the value of the specified identifier. The decrement operator `--` subtracts the value of 1 (for integer operands) and 1.0 (for float operands) from the value of the specified identifier.

5.3 Update Expressions

Update statements are binary operations which have the form: identifier operator operand. The identifier may be a variable, an entry in a collection, or a property of a struct provided the type of the identifier's value is either an int, a float, or a string. The operand may be any expression that evaluates to a value of type int or float if the identifier's value is of type int or float identifier, or of type string if the identifier's value is of type string. There are four update operators: `+=`, `-=`, `*=`, and `/=`. In the case of int and/or float identifiers and operands the `+=` operator adds the value of the identifier and the operand and replaces the value of the identifier with the result of the addition. The `-=`, `*=`, and `/=` operators work similarly, but perform subtraction, multiplication, or division respectively between the identifier and operand, replacing the identifier's value with the result of the operation.

If the identifier and operand are of type string, only the `+=` operator may be used. The operand string will be concatenated to the end of the string currently associated with the identifier, and the result will replace the value of the identifier.

5.4 Code Blocks

A code block is a set of curly braces surrounding a set of newline-delimited statements. The opening and closing curly

5.5 Control Structures

Cardian supports four types of control structures: `if`/`elseif`/`else`, `while`, `for`, rules

5.5.1 if

In an `if` block, the `if` keyword must be followed by a predicate, which must evaluate to either `true` or `false`, and then a block of code. If the predicate evaluates to `true`, the block of code is executed and control leaves the `if` structure. This can then be followed by 0 or more `elseif` statements, each of which has its own predicate and code block, and a single optional `else` block which has only a block and no predicate. In an `if` structure without an `else` block, at most one code block is executed. In an `if` structure with an `else` block, exactly one code block is executed.

5.5.2 while

A `while` statement is made up of the `while` keyword, predicate, and code block. The predicate is evaluated and if its value is `true`, the code block is executed and the predicate is re-evaluated. The code block will be run once for every time the predicate evaluates to `true`.

When the predicate evaluates to `false`, control passes out of the `while` structure.

5.5.3 for

The `for` statement is used to iterate over items in a collection. The syntax of a `for` loop is: the `for` keyword, the name to use for the local variable in the code block, a colon, the name of the collection, and the code block to execute. The code is run once with each item in the collection. The item is passed to the code block using the specified variable name, and the code is run. The `for` loop's code block will iterate once for each item in the collection, assigning each object's value to the variable in scope sequentially. The objects will be treated as the same type as those in the collection.

5.5.4 rules

The rules statement is used to check several (potentially related) conditionals. It begins with a question mark, followed by an optional set of parentheses that contain a comma-delimited list of local variable assignments, and then a brace-delimited block. The block can contain 0 or more conditional statements, each consisting of a predicate followed by a code block. When control reaches the structure, the specified variables are bound and each conditional is evaluated in sequence. Any predicate which evaluates to true will have its associated code block executed. When control reaches the end of the brace-delimited block, the local bindings defined at its beginning go out of scope and control passes out of the rules structure.

5.6 Function Declaration

Functions may be declared at the top level by entering an identifier to be used as the name of the function, a parenthesis enclosed comma separated list of zero or more identifiers to be used as arguments, an equals sign, and a curly brace enclosed code block to be used as the body of the function.

Any valid identifier may be used as the name of the function as long as it is not already bound to another type. Once the function is declared, the identifier is bound to the function type and no value of any other type may be assigned to it.

The argument list is mandatory, but it may be empty. If arguments are included, any valid identifier may be used for the name of the arguments. These will only exist within the scope of the function body, and will not be accessible once the function ends. The values of the arguments may be of any type, but once the function is called, each argument’s identifier is bound to the type it was called with until the function ends. Any assignments to these identifiers in the body of the function must be of the same type each was given when the function is called.

The body of the function must be a valid code block as described above. In addition the the statements allowed inside any other code block, the body of a function may also contain the return statement. This statement consists of the word “return” followed by an expression. When control reaches this return statement, the expression will be evaluated, but no further

statements in the function body will be executed. Instead, the value of the expression will be returned by the function. If control reaches the end of the function body without encountering a return statement, the value of boolean false will be returned instead.

6. Expressions

An expression could be either the right side of an assignment, or the entire statement. Valid types include direct expressions, unary expressions, binary arithmetic expressions, logical expressions, binary string operation, update and increment expressions, collection reference, enum reference, struct reference and function call. The explanation of each type will be described in following subsections.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

6.1 Operator Precedence

When expressions are evaluated, operators will be applied in the following order (listed from highest to lowest precedence).

`()`	grouping
`f(args)`	function calls
`x[index]`	indexing
`x.attribute`	referencing
`-`	unary arithmetic negation
`* / %`	multiplication, division, modulo
`+ -`	addition, subtraction
`== != < > <= >=`	comparison
`!`	logical negation
`and`	logical and
`or`	logical or

6.2 Direct Expressions

The following are all considered to be direct expressions:
identifiers: an identifier refers to a variable or a function. See the section of identifiers.
constants: a constant with primitive types (integer, float, or boolean), or derived types (enum, struct, or collection) is a direct expression.
strings: See the section on strings.

6.3 Unary Expressions

There are two unary expressions in Cardigan.

One is minus "`-`", which is used with a single numeric-valued operand such as an integer constant and a float constant. This operator forces its numeric-valued operand to be negative, and the result is of numeric type with value that equals to the negative of the numeric operand. The other one is not "`!`", which is used with a single boolean constant or a boolean variable. This operator forces its single operand to be negative, and the result is of boolean type with value that equals to the negative of the operand.
Unary expressions are grouped right-to-left.

6.4 Binary Arithmetic Expressions

There are five different binary arithmetic operations: `+`, `-`, `*`, `/`, `%`. The modulo operator, `%`, is only defined when both arguments are integers. The result is the first modulo the second. `+`, `-`, `*`, `/` are defined for any combination of integers and floats. The results are the standard arithmetic sum, difference, product, and quotient of the two numbers in question. `+` and `-` are commutative but the others are not. If both arguments are integers, then the operator will return an integer. If either or both is a float, then the operator will return a float.

6.5 Logical Expressions

There are four types of logical expressions: unary boolean expressions, binary boolean expressions, direct boolean expressions, and comparison expressions. Each type evaluates to a boolean value.

6.5.1 Unary boolean expressions

Unary boolean expressions involve the negation operator (`!`) followed by another boolean expressions. The result of this type of expression is the opposite boolean value of the expression directly following the negation operator. The ! operator has higher precedence than the other boolean operators (`and`, `or`), but lower precedence than the equality and comparison operators.

6.5.2 Binary boolean expressions

Binary boolean expressions have the following form:

operand operator operand

where the operands can be any boolean expression and the operator is either the `and` operator or the `or` operator. Operations involving the `and` operator return `true` if both operands have the value `true` and `false` otherwise. The `or` operator returns `true` if one or both of its operands have the value `true` and `false` if both have the value `false`. The `and` operator has higher precedence than the `or` operator, and the equality comparison has higher precedence than both the `and` and `or` operators.

6.5.3 Comparison expressions

Comparison expressions have the form:

operand operator operand.

The operands may be of any primitive type, an identifier which has the value of a primitive type, or an expression which evaluates to a primitive type. For the purposes of comparisons, enum values are treated as ints. All types allow use of the equality and inequality operators, == and != respectively. Integer and float values also support the use of the following operators, <=, >=, >, <. In all cases the operands must have the same type, except for comparisons between integers and floats, which are allowed to be compared to each other. The behavior of the operators with each type is described below.

Booleans -- When the operands are booleans, the == operator functions like an XNOR comparison, returning true if both values are the same, and false if they are different. The "!=" operator functions as an XOR comparison, returning true if the operands have different values, and false if their values are the same.

Ints -- The equality operator "==" returns true if the operands have the same value and false if they don't. The inequality operator "!=" returns true if the operands do not have the same value, and false if they do. The strictly less operator "<" returns true if the first operand is less than the second and false if they are equal, or if the first is greater. The strictly greater operator ">" returns true if the first operand is greater, and false if they're equal or the first is less. The less or equal operator "<=" functions the same as the strictly less operator except that it evaluates to true if the values are equal. The greater or equal operator ">=" functions the same as the strictly greater operator except that it evaluates to true if the values are equal.

Floats -- The rules for float comparisons are the same as integer comparisons. Comparisons between a float and an integer are also allowed; the int is treated as a float with the same value as the integer and a 0 fraction value.

Strings -- The "==" operator performs a character-wise comparison of the strings and returns true if they are identical or false otherwise. The "!=" performs the same comparison and returns true if they are not identical, and false if they are.

6.5.4 String concatenation

String concatenation is performed using the + operator. The syntax is:

string + string

where each string may be a string literal or a string variable. The result of the expression will be a single string which is the concatenation of the first and second string operands.

6.5.5 Collection References

Collections are indexed using the [ ] separators and may be used to access any element or value in the collection using the integer mentioned inside the [ ] separatora. The syntax is:

my_col[N]

where my_col is the collection and N is an integer which gives you the Nth value of the collection.

6.5.6 Struct and Enum References

The element of an enum can be accessed by an identifier or an enum object, followed by .(dot), and another identifier. Here, the first identifier refers to a user-defined enum name, and the second one refers to the name of the element.

Similarly, the property of a struct can be accessed by an identifier or a struct object, followed by .(dot) and another identifier. Here, the first identifier refers to a user-defined struct name, and the second one refers to the name of the property.

6.5.7 Function Calls

To call a function and execute the statements inside the body of the function, the following syntax must be used: identifier(expressions), where the identifier has been previously defined as a function. The expressions is a comma separated list of zero or more expressions whose values will be assigned to the parameters of the function. The number of expressions in the function call must match the number of parameters in the function definition. When the function is called, control will jump to the first line of the function and continue until a return statement is evaluated, or the end of the function body is encountered. Control will then return to just after the point where the function call was made. If a return statement was encountered, the function call will be evaluated to have the value of the expression in the return statement. If no return statement was encountered before the end of the function body, the function call will be evaluated to have the value of boolean `false`.

7. Writing programs

7.1 Structure of a program

At the top level, a cardigan program is a series of assignment statements, either variable assignments, or function definitions. Variables declared at the top level will be treated as global variables. These may not make use of function calls, but may only consist of constants or expressions involving previously defined global variables. All function declarations will be stored for later use. A special function called `PLAY` must be included in every program as the entry point of execution.

7.2 File extensions

Cardigan source files have the extension .crd

7.3 Built in Functions

There are four built in functions in Cardigan: `INPUT`, `OUTPUT`, `CAST`, and `PLAY`.

`INPUT` is used to get user input. It should be called as a function with no arguments. When this function is called, execution of the program pauses until a user enters a value on the terminal. The function is then evaluated to have the value of whatever the user entered. The result of `INPUT` is always a string.

`OUTPUT` is used to display text on the screen. It should be called as a function with a single string argument. The argument is the string which will be printed on the screen. `OUTPUT` does not append a newline to the output string. If a newline is desired, it must be inserted in the string before `OUTPUT` is called.

`CAST` is used to cast a variable of one type to another. It should be called as a function with two arguments. The first is the value to be casted, the second is a primitive type keyword (int, float, string, bool) which represents the desired output type.

input type	**type keyword**	**output**
int	int	int
int	float	float
int	string	string
int	bool	`false` if the int is 0 `true` otherwise
float	int	ERROR
float	float	float
float	string	string
float	bool	`false` if the float is 0.0 `true` otherwise
string	int	an integer representation of the string if the string contains only numerals
string	float	a float containing the value represented by the string. The string must contain a valid float definition as described in the float primitive section above. An error is thrown otherwise
string	string	string
string	bool	`false` if the string is empty `true` otherwise
bool	int	1
bool	float	1.0
bool	string	`true` if the bool is true, or `false` if the bool is false
bool	bool	bool

`PLAY` is not defined by the language, but is used as the entry point to the program. Any global variables declared outside of function definitions will be assigned first, and then control will begin at the start of the `PLAY` function. When control reaches the end of the `PLAY` function the program terminates.