corgi

Language Reference Manual
COMS W4115

Philippe-Guillaume Losembe (pvl2109)
Alisha Sindhwani (as4312)
Melissa O'Sullivan (mko2110)
Justin Zhao (jxz2101)

October 27, 2014

Chapter 1: Introduction

corgi is a language centered on music translation, generation, and analysis. It will be able to read in a MusicXML file which are both standardized digital file formats for interpreting music and translate the files into the appropriate data structures. Similarly, a user will be able to generate music directly through the implementation of our musical data structures. These data structures will allow our language to quantitatively analyze and find patterns in music that would be difficult to do manually.

Why is it called corgi?

According to wikipedia:

> *Corgis are very active and energetic. They have a strong desire to please and should receive both physical and mental exercise regularly. They should be socialized early on because they tend to be shy and cautious with strangers and other dogs. They have a tendency to be very vocal, and for this reason make good alarm dogs. They are typically good with children, but due to their herding behavior, may nip at their heels during play.*

So… yeah!

Chapter 2: Types and Type Declaration

Type Declaration

Data is expressed in explicitly declared types similar to Java.

Primitive Types

- Character
  A character in corgi is a primitive type denoted by the keyword char. A char is declared as such:

```
char c = 'c';
```

- Integer
  Much like Java, an integer in corgi is a primitive type denoted by the keyword int and representing values ranging from 0 to 2^32-1. An int is declared by:

```
int i = 7;
```

Non-Primitive Types

- String
  A String is an array of chars. For example a String can be declared by:

```
String str = "music";
```

- Fractions
  A fraction is reduced division of two integers. It's type declaration is denoted by the keyword frac and each value definition begins with the character '<', followed by the numerator of the fraction, separated by the denominator of the fraction by a '/' and ending with the character '>'. For example a fraction can be declared as:

```
frac f = <¾>;
```

- Duration
  Duration is fraction that meets the constraint that the numerator is less than the denominator. It is used to represent the length of a chord and can be either declared directly or cast from a fraction as shown:

```
duration d = <¼>;
fraction f = <¾>;
duration fd = duration(f);
```

- Pitch
  Pitch is defined by an integer. It is declared using the keyword pitch, for example:

```
pitch p = 4;
```

- Rhythm
  Rhythm refers to a sequential list of durations. It is declared using the keyword rhythm as shown:

```
duration d = <¼>;
rhythm r = [d,d,d];
```

- Chord
  A chord is a sequential list of (pitch, duration) tuples. A chord can be declared using the keyword chord as follows:

```
pitch p1 = 4;
pitch p2 = 5;
duration d1 = <¼>;
duration d2 = <⅛>;
chord c = [(p1,d1), (p1,d2), (p2, d2)];
```

- Track
  A track is a sequential list of chords which can be declared using the keyword track. For example:

```
pitch p1 = 4;
pitch p2 = 5;
duration d1 = <¼>;
duration d2 = <⅛>;
chord c1 = [(p1,d1), (p1,d2), (p2, d2)];
chord c2 = [(p1,d1), (p1,d2), (p2, d2), (p2, d1)];
track t = [c1, c2, c2];
```

- Composition
  A composition is a sequential collection of tracks. A composition can be declared using the keyword composition as follows:

```
pitch p1 = 4;
pitch p2 = 5;
duration d1 = <¼>;
duration d2 = <⅛>;
chord c1 = [(p1,d1), (p1,d2), (p2, d2)];
chord c2 = [(p1,d1), (p1,d2), (p2, d2), (p2, d1)];
track t1 = [c1, c2, c2];
track t2 = [c1, c1];
composition x = [t1, t2, t1];
```

Chapter 3: Lexical Conventions

In corgi, a token is a string of one or more characters consisting of letters, digits, or underscores. corgi has 5 kinds of tokens:

Identifiers
Keywords
Constants
Operators
Newlines

Identifiers

The first character must be a letter and identifiers are case sensitive. The letters are the ASCII characters a-z and A-Z. Digits are the ASCII characters 0-9.

letter → [‘a’-’z’ ‘A’-’Z’]
digit → [‘0’-’9’]
underscore → ‘_’
identifier → *letter* (*letter* | *digit* | *underscore*) *

Keywords

The following identifiers are strictly reserved for use as keywords:

Keywords	**Description**
int	standard 32-bit integer
frac	two integers that represent a fraction
duration	wrapper around fraction
pitch	wrapper around integer, this can also be instantiated as 'C+4'
rhythm	a collection of durations
chord	a collection of pitch duration tuples
track	a sequential list of chords
composition	a collection of tracks
True / False	Boolean constants

if / elif / else	Conditional expressions
random	generate random numbers
print	Print information to stdout
main	Declaration of the main program
return	specifies a return statement.

Literals

Strings will be a list of characters in corgi. Declaring string literals in corgi is fine, but will be immediately converted to a list of characters. Defining a string literal is simply done with a sequence of one or more characters enclosed by single quotes. The only special escape characters are:

Escaped	**Description**
\’	single quote
\n	new line
\t	tab

Punctuation

Punctuation	**Use**	**Example**
,	list element separator, function parameters	array = [1, 2, 3]
[]	list delimiter, list access	array[0] = 3
()	conditional parameter delimiter, function parameter delimiter	if (array[0] == 3)
{}	statement list delimiter	if (array[0] == 3) { /* work */}

“	string literal delimiter	s = “what\’s up?”
;	end of statement	array = [1, 2, 3];

Comments

Comments are super useful and corgi supports comments in two flavors.

Comment Symbols	Description	Example
/* */	Multiline comments	/* This is a comment */
//	Single-line comment	// This is a comment

Operators

An operator is a token that specifies an operation on at least one operand and yields some result.

	int	**frac**	**duration**	**pitch**	**rhythm**	**chord**	**track**	**composition**
"="	assignment	assignment	assignment	assignment	assignment	assignment	assignment	assignment
"+"	addition	addition	addition	adds the pitch values				
"-"	subtraction	subtraction	subtraction	subtracts the pitch values				
"*"	multiplication	multiplication	multiplication	multiplies two pitch values				
"/"	division	division	division					
[]					accessor	accessor	accessor	accessor
>	compare value	compare value	compare duration		compare duration			
<	compare value	compare value	compare duration		compare duration			
"=="	check equality	check equality	check equality	check equality				

"!="	check equality	check equality	check equality	check equality				
"."						invoke method	invoke method	invoke method
"++"	increment	increment	increment	increment pitch value				
"--"	decrement	decrement	decrement	decrement pitch value				

Operator precedence from greatest to least precedence:

“ ”
“.”
“++” “--”
“*” “/” “%”
“|” “-”
“<” “>” “<=” “>=”
“==” “!=”
“=”

Chapter 4: Syntax

Program Structure
A program in corgi is made up of one or more valid statements. A Program begins in a main function which needs to be defined for any statements to be executed.

Expressions
In corgi, an expression is made up of variables, operators, and method calls. An expression must evaluate to a value of one of corgi's data types. An expression is evaluated from left to right as shown:

```
10 - 2 - 3 - 4 //evaluates to 1
```

Variables
A variable refers to a data type. They type and value of a variable is declared and initialized with the type keyword, variable name, and value in a single line as follows:

```
int a = 4;
```

For type specific examples refer to Chapter 2.

Binary Operators
Binary operators can connect variables to create composite expressions. These operators are of the form.

```
x operator x //with x representing an expression
```

Types of Binary Operators include:

- **Arithmetic operators** such as addition (+), subtraction (-), multiplication(*), division (/), and modulus (%). The expressions acting as operands for an arithmetic operator must be both the same type and that type must be int, frac, or duration. The resulting value of the expression composed of two expressions of the same type is a value of that type.

- **Relational operators** such as less than (<), greater than (>), equal (==), or not equal (!=) require operands to be of the same type and of types including int, frac, duration, pitch, or rhythm. The result of a relational operator invoked on two operands of the same type is an integer equal to 0, if the expression evaluates to false or 1 otherwise.

The Role of Parentheses
Parentheses may guide the order of operations on expressions as the expression inside a set of parentheses must be evaluated before that expression can be evaluated with respect to other operators. The surrounding of a set of parentheses around an expression does not change the subexpressions value.

Statements

A statement is an instruction to be executed. An expression on its own is not a valid statement, with the exception of a function call. It is either a single instruction that ends in a ';' or begins a list of statements contained between curly braces ({ }). There are four types of statements in corgi:

- **Assignment**
  An expression's value can be assignment to a variable with this statement.
  ```
  int a = 4;
  int b = a + 1;
  ```

- **Function Creation**
  Functions can be created much in the style of C functions. The method header includes the return type, function name, and parameters. The return type can be omitted in the case of a function that does not return a value, but the function must return the type declared in the header. This is a function with no parameters which returns a chord:
  ```
  chord function1() {
      chord c = [(1, <1/2>)];
      return c;
  }
  ```
  a function with no return value and two parameters:
  ```
  function2(chord c, int i) {
      …
  }
  ```

- **Return Statement**
  Return statements are specified

- **Function Calls**
  A function call consists of the function's name followed by its parameters in parentheses and surrounded by commas. The parameters and the function call itself are expressions whose type are determined from a previous function definition. The function call's value is the function's return value. Functions can be called with no parameters but the parentheses cannot be omitted.
  ```
  chord c = function1();
  function2(c, 2);
  ```
  A function call can be used as a stand alone statement but its return value will be lost if it is not assigned to a variable.

- **Control Statements**
  - **for** loop
    A for statement takes two assignment statements and a Boolean expression and executes its statement list until its condition evaluates to False, the first

assignment is executed when the for statement is encountered and the second one after each iteration of the loop:

```
for ( assignment1; condition; assignment2 ) {
      …
}
```

- **while** loop
  A while statement takes a Boolean expression and executes its statement list until the expression evaluates to False:

```
while ( condition ) {
      …
}
```

- **if elif else**
  An if else statement takes a Boolean expression and executes one statement list if its value is True and the other statement list otherwise:

```
if (condition) { // condition is not 0
      …
} else { // condition is 0
      …
}
```

  if else statements can be chained to test several conditions with elif:

```
if ( condition1 ) {
      …
} elif ( condition2 ) {
      …
} elif ( condition3 ) {
      …
} else {
      …
}
```

Scope

Block scoping

A block is a list of statements enclosed between two braces. Blocks can be nested and have their own local variables. A variable is only accessible in the block in which it was defined and blocks inside this one.

```
int x = 5;
{
      int y = x + 1;
      x = y + 1;
}
if (x > 5) { // This is true
      y = 0; // This is not allowed, y has no type or value
```

```
}
```

Function scoping

Functions only have access to variables in their parameter list and local variables declared inside the function.

Chapter 5: Standard Library

import()

Usage:

```
composition c = import("filepath/test.xml");
```

By using the import function, one can read in a music xml file from the file system into a composition variable.

export()

Usage:

```
export(c, "filepath/masterpiece.xml");
```

By using the export function, one can export a composition "c" of theirs to a music xml file for further processing and alteration.

print()

Usage:

```
print('Hello, World!');
print('Hello, World!');
```

By using the print function, one can print to either stdout or to a file. The parameter must be of type string which will be written to standard out.

list.add()

We can add elements to a list. The parameter passed in must be of the same type as the list elements.

list.remove()

We can remove elements to a list. The parameter passed in must be of type int.

```
// start with 1-5
myCollection = [1,2,3,4,5];
// let's add 6
myCollection.add(6); // [1,2,3,4,5,6];
```

```
// then we can take away the element at index 3
myCollection.remove(3); //[1,2,3,5,6];
```