DECAF

Language Reference Manual

Hidy Han (yh2635) - Language Guru

JiaYan Hu (jh3541) - Systems Architect

Kim Tao (kmt2152) - Tester

Kylie Wu (kcw2141) - Manager

Table of Contents

1. About DECAF

DECAF is a general-purpose, object-oriented language that compiles to LLVM. It is a language that will be intuitive to use for programmers who have previously used other high-level languages, such as Java, C, and Python. DECAF will extract a core subset of features from Java and Python and present these features in a concise semantic model.

More specifically, DECAF will support core object-oriented functionalities, such as inheritance and polymorphism. Additionally, DECAF will present flexible and robust built-in data structures, such as Python's list and tuple structures, which did not exist natively in Java.

2. Data Types

i. Primitives

DECAF supports a variety of different basic primitive types, allowing its programs to be versatile.

Type	Description	Use Cases
`bool`	Boolean type; can either be true or false	Used to indicate a conditional statement that can be true or false.
`int`	Integer with 32-bit (4 byte) capacity	Used to store base ten integers ranging from -2^{31} to $2^{31} - 1$.
`float`	Floating point with 64-bit (8 byte) capacity	Can store decimals or integers greater than the range `int` can handle.
`char`	Represents a single ASCII character; size is 1 byte	Can be used to represent a character or text as a sequence of characters.

ii. Literals

Literals are syntactic representations of fixed values for primitive types. They are the actual representation of values in the program, as opposed to being hidden within an identifier.

Type	Syntax	Example
`bool`	Only two possible values: true, false	`true, false`
`int`	A sequence of digits, may be preceded by - to indicate negativity	`4115, -325409`
`float`	Typed as decimal fractions, may be preceded by - to denote negativity	`0.25, -534.2908`
`char`	A single character enclosed by single quotes (‘’) Refer to the following table for escape sequences	`‘a’, ‘K’`
`string`	Sequence of characters enclosed by double quotes (“”)	`“Hello World”`
`list`	Mutable sequence of expressions separated by commas enclosed by brackets ([])	`[1, 2, 3, 4]`
`tuples`	Immutable sequence of expressions separated by commas enclosed by parentheses (())	`(1, 2, 3)`

Escape Sequences:

Syntax	Meaning
`‘\0’`	null character
`‘\\’`	backslash
`‘\’’`	Single quote
`‘\”’`	double quote
`‘\n’`	newline character
`‘\t’`	tab character
`‘\r’`	carriage return

Example:

```
string str = “William says:\t\”Hello!\”\n”
print(str @ “The End”);

Expected Output:
William says:    “Hello!”
The End
```

iii. Built-ins: Strings

Strings are immutable sequences of characters.

`string`	Represents a sequence of characters; size is variable	Used to represent text.

String Operations

A string in DECAF can be thought of as an immutable list of chars (see Built-Ins: Lists). The operations that can be performed on a string are identical to those that can be performed on a list. However, in the case of mutation operators, such as `::`, `@`, and `~`, a copy of the string is returned and the original string operated on is unchanged.

Operation	**Effect**
`[]`	Returns the character at a given index in the string (as a string).
`[:]`	Returns a substring (as a copy) denoted by the indices given. Includes the initial index and excludes the second index.
`::`	Returns a copy of the string with a single character appended.
`@`	Returns a copy of the string concatenated with another string.
`~`	Returns a copy of the string with the specified substring deleted.

iv. Built-ins: Lists

Lists are mutable sequences, which can be changed after they are created. The items of a list are arbitrary objects of the same type (see object hierarchy and polymorphism), built-ins of the same type, or primitives of the same type. Lists are created by placing a comma-separated list of expressions in square brackets. Empty lists are simply represented by square brackets without any expression in between.

List Operations

Operation	Effect
[ ]	Index into the list.
[:]	Slices and returns a subsequence (as a copy) denoted by the indices given. Includes the initial index and excludes the second index.
: :	Append a single value to the list.
@	Append a list to another list.
~	Delete a subsequence of the list.

List Keywords

Keyword	Meaning
in	Used to iterate through a list.

Example:

```
int[] empty = [];
int[] nums = [1,2,3,4,5];
nums::0;
int[] negs = [-1,-2,-3,-4,-5];
nums@negs;

int[] extracted = nums[0:5];



~nums[0];
~nums[0:2];
```

```
//empty list
//list creation
//nums is now [1,2,3,4,5,0]

//nums is now
[1,2,3,4,5,0,-1,-2,-3,-4,-5]
//extracted is [1,2,3,4,5], excludes
index 5
//slicing does not modify the
underlying list
//deletes the 0th index
//delete indices 0 through 2
```

List Comprehension

DECAF provides a syntactic construct for creating a list based on existing lists. For example,

```
int[] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
int[] doubled_evens = [n * 2 for n in numbers if n % 2 === 0];
```

v. Built-ins: Tuples

Tuples are immutable sequences that cannot be changed once created. The items of a tuple are primitives, built-ins, or objects of the same type. Tuples of two or more items are formed by comma-separated list of expressions. A tuple of one item can be formed by affixing a comma to an expression; an empty tuple can be formed by an empty pair of parentheses.

Tuple Operations

Operation	**Effect**
[ ]	Index into the tuple.

Tuple Keywords

Keyword	**Meaning**
in	Used to iterate through a tuple.

Example:

```
int() empty = ();
int() tup = (1,2);
int() single = (1,);
int val = single[0];
```

```
//empty tuple
//tuple creation
//single element creation
//val is 1
```

vi. Built-in Libraries

DECAF has a set of standard libraries for its built-in types: lists, tuples, and strings. This set of libraries are automatically imported upon program start.

String Built-in Methods

Method Header	Description
`capitalize(string param) -> string`	Returns a copy of the string with the first letter capitalized.
`is_alpha(string param) -> bool`	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
`is_digit(string param) -> bool`	Returns true if string contains only digits and false otherwise.
`is_alnum(string param) -> bool`	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
`lower(string param) -> string`	Returns a copy of the string in all lower case.
`upper(string param) -> string`	Returns a copy of the string in all upper case.
`replace(string param, string old, string new) -> string`	Returns a new string such that all occurrences of old in param with new.
`length(string param) -> int`	Returns the number of characters of the given string.
`print(string param)`	Prints the string to the console.
`to_list(string s) -> <char>[]`	Returns the given string as a list of characters.

List Built-in Methods

Method Header	Description
`length(<type>[] param) -> int`	Returns the number of elements of the given list.
`reverse(<type>[] param) -> <type>[]`	Returns a reversed copy of the given list.
`max(<type>[] param) -> <type>`	Returns the item from the list with max value. Defined only for lists of primitive types.
`min(<type>[] param) -> <type>`	Returns the item from the list with min value. Defined only for lists of primitive types.
`to_tuple(<type>[] param) -> <type>()`	Returns the given list as a tuple.

Tuple Built-in Methods

Method Header	Description
`length(<type>() param) -> int`	Returns the length of the given tuple.
`reverse(<type>() param) -> <type>()`	Returns a reversed copy of the given tuple.
`max(<type>() param) -> <type>`	Returns the item from the tuple with max value. Defined only for tuples of primitive types.
`min(<type>() param) -> <type>`	Returns the item from the tuple with min value. Defined only for tuples of primitive types.
`to_list(<type>[] param) -> <type>[]`	Returns the given tuple as a list.

vii. Objects

DECAF is an object-oriented programming language, so it supports the implementation of objects, which are entities that have state and behavior. While DECAF has the capability of being object-oriented, it does not have many built-in object classes. Thus, it is left to the programmer's discretion to use this feature as they please.

Object classes can extend other classes, or they can implement an already defined interface. Their state and behavior are defined by listing out characteristic fields and methods in a class declaration.

Fields

Fields can consist of primitive data types or other object classes, and methods can return primitives or objects. Fields can be accessed and modified using dot (`.`) operator. A field can be accessed by the following format: `obj.field`

Methods

Object methods can be defined by the user within the class. They typically perform an operation on the instance it is invoked upon and can modify the state of the object. A method can be called by using the dot (`.`) operator in the following format: `obj.method()`. In DECAF, the programmer is given the ability to write whatever methods they may require, utilizing the various built-in data types and operators.

Methods are operations performed on an object that can accept input as parameters and produce output. These inputs and outputs can be primitives, objects, or built-ins. The method signature determines the formal parameters, including the number of arguments, their type, and their order. A calling method must match this signature in order to make a valid call to the method.

Methods can be overloaded and overridden. Method overloading is the usage of methods of the same name with different method signatures. If an overloaded method is called, it will call the method that matches the types of the arguments passed in. Method overriding is described in the Object Hierarchy and Polymorphism section.

Return Types

In the method signature, the return type of a method is specified by the type following the `->` symbol. Returning from a method is indicated by the `return` keyword followed by the primitive or object to return. The type that is returned must match that of the method signature. If a method does not return anything, the `->` should be omitted and

the `return` can be omitted (although it can still be used to prematurely exit a method).

The format of a method is as follows:

```
method_name(param1_type param1_name, ... , paramn_type paramn_name) ->
return_type {
        ...
        return some_value;
}
```

Examples:

```
// Method that returns int
gcd(int x, int y) -> int {
    if (y === 0) {
        return x;
    }
    return gcd(y, x % y);
}
```

```
// Method that returns nothing
foo(string print_me) {
    print(print_me);
}
```

```
class Square {
    int length;

    Square(int length) {
        self.length = length;
    }
}

// Method that returns Square object if x is true, returns null
// otherwise
bar(bool x) -> Square {
    if (x) {
        return Square(5);
    }

    return null;
}
```

Instantiation

Objects are instantiated using the object's constructor method. The name of a constructor has to match the class name. Constructors require no return values but implicitly return the object just instantiated to the calling method.

Objects can be assigned to variables once they have been instantiated. The keyword `null` can be used to represent a nonexistent object.

Classes

A class is the template for an object which can be instantiated. A class is capable of providing concrete implementations for interfaces, extending the functionality of another class, or being extended to subclasses. Classes have state, represented by fields, and behavior, represented by the methods that can be invoked.

Interfaces

An interface is a set of method declarations designed to be implemented by classes. An interface can declare any number of methods and define any number of constants, however it cannot define instance variables or constructor methods. As such interfaces cannot be instantiated and do not contain constructor methods.

Object Hierarchy and Polymorphism

Objects have the concept of subclasses that extend superclasses and implement interfaces. The core purpose of the object-oriented model is that subclasses will inherit the fields and methods of superclasses. That is, a subinterface can extend superinterface to provide additional methods and constants. A subclass will inherit all of the fields and methods of its superclass. Furthermore, it can override methods in superclasses to provide its own implementation as well as add new fields and methods. DECAF supports single inheritance rather than multiple inheritance, so a subclass can extend at most a single superclass.

Objects are able to take on many forms, that is, a superclass reference can be used to refer to a subclass object. This ability is commonly referred to as polymorphism.

Object Keywords

Keyword	Description
`class`	Demarcates a class definition.
`interface`	Indicates that a class is an interface.
`implements`	Indicates that a class implements an interface.
`extends`	Indicates that a class extends another class.
`super`	Similar to Java's `super` keyword, it accesses overridden methods in the superclass.
`self`	Similar to `this` in Java, which refers to the object in the current context.
`null`	Represents a typeless nonexistent object.

Object Operations

Operator	Description
`Object()`	Instantiates an object.
`o.f`	Accesses a field f in the object o.
`o.m()`	Invokes method m on the object o.

The class definition follows the following format:

```
class class_name [extends [class_1, …]] [implements [interface_1, …]] {
    ...
}
```

Examples:

```
// Basic class declaration
class MyClass {
    ...
}
```

```
// Example of extending a class
class Mammal extends Animal {
    Mammal(string species) {
        super(species);    // Calls constructor of superclass
    }
    ...
}
```

```
// Interface declaration
interface Shape {
    ...
}
```

```
// Example of implementing an interface
class TwoDShape implements Shape {
    ...
}
```

```
class Rectangle extends TwoDShape {
    int length;
    int width;

    Rectangle(int length, int width) {
        self.length = length;    // self refers to its own fields
        self.width = width;
    }

    ...

    perimeter() -> int {
        return length * width;
    }
}

class Square extends Rectangle {

    Square(int sideLength) {
        // Calls constructor of superclass
        super(sideLength, sideLength);
    }

    ...
}
```

Exceptions

Exceptions are special built-in objects that are thrown automatically when some violation occurs during runtime that may not have been caught during compilation. These exceptions must be handled by the code using a `try-catch-finally` block, otherwise the program will exit with a failure. The programmer can also manually throw exceptions within their code using the keyword `throw`.

When an exception is thrown and is wrapped in a try block, it will be handled by the subsequent catch clause with the matching exception type. If the exception is thrown and is not wrapped in a try-catch block or does not have a catch clause to handle that type of exception, it will prematurely exit from the current method and throw the exception up the call stack to the calling method. The finally clause is unconditionally executed at the end of the try-catch regardless of whether the statements in the try block executed successfully or not.

DECAF has several exceptions automatically built into it. The following exceptions are supported:

- NullPointerException
- IndexOutOfBoundsException
- DivideByZeroException

The format for throwing an exception is as follows:

```
throw <ExceptionName>
```

Example:

```
int test_0 = 1000;
int test_1 = test_0 - 1000;
int test_2 = test_0 / test_1;
```

```
// test_1 is effectively 0
// DivideByZeroException thrown
```

```
try {
    Object o = null;
    o.nonexistent_method();
    return 0;
}
catch (NullPointerException e) {
    print("Nonexistent method");
}
finally {
    print("done");
}
```

```
// code block to attempt

// catching NullPointerException

// this block will always execute
```

viii. Casting

Casting must be explicit by using the <type> operator. Promotion is not supported. For instance, if a user want to perform arithmetic operations on a float and an integer, they must choose to cast one of the two operands so that they have matching types.

The following casts are supported:

- int -> float
- int -> string
- int -> bool (0 becomes false, nonzero become true)
- float -> int (the fractional portion will be truncated)
- float -> string
- float -> bool
- char -> string
- bool -> int (true becomes 1, false becomes 0)

```
int x = 3;
float y = 1.0;
int z =  x + y; //error
```

```
int x = 3;
float y = 1.0;
int z =  x + <int> y; //ok
```

3. Lexical Conventions

i. Identifiers

Identifiers are sequences of characters used for naming variables, new objects and methods. Identifiers can contain ASCII characters, digits and underscore '_'. The first character can only be a character or underscore. An identifier cannot have collisions with other keywords or names of built-in methods.

By convention, '_', rather than camelCase, is used for the readability of long identifiers.

ii. Comments

Comments in DECAF are demarcated using Java-like syntax, and do not affect compilation. Comments cannot be nested.

Syntax	Description
`// comment`	Used for single-line comments.

`/*`
`comment`
`comment`
`*/`
	Used for multiline comments.

iii. Keywords

The following keywords are reserved in DECAF and may not be used otherwise:

main	const	int	float
char	string	bool	list
tuples	in	class	interface
implements	extends	super	self
null	and	or	not
if	elseif	else	for
while	break	continue	return
try	except	finally	throw

iv. Operators

DECAF can perform arithmetic, logical, and boolean operations. These different types of operators can be used in combination to create powerful methods and logic.

Arithmetic Operators

Operator	Description
+	Sums the operands together.
-	Subtracts the right operand from the left operand.
*	Multiplies the operands together.
/	Divides the left operand by the right operand.
%	Performs modulus division of the left operand using the right operand.

Boolean Operators

Operator	Description
`and`	Returns true when both operands evaluate to true; returns false otherwise.
`or`	Returns true if either or both of the operands evaluate to true; return false otherwise.
`not`	Negates the boolean value of the operand.

Comparison Operators

Operator	Description
==	Compares the two operands' references; returns true when they are referentially equal, false otherwise.
===	Compares two operands' values; returns true when they are equal in terms of value, false otherwise.
!=	Compares the two operands' references; returns true when they are value-wise not equal, false otherwise.
!==	Compares two operands' values; returns true when they are referentially not equal, false otherwise.
<	Returns true if the left operand is less than the right operand, false otherwise.
>	Returns true if the left operand is greater than the right operand, false otherwise.
<=	Returns true if the left operand is less than or equal to the right operand, false otherwise.
>=	Returns true if the left operand is greater than or equal to the right operand, false otherwise.

List and Tuple Operators

Operator	Description
[ ]	Used to index into lists and tuples.
[:]	Slices and returns the subsequence in a list denoted by the indices given. Includes the initial index and excludes the second index.
: :	Append a single value to the list.
@	Append a list to another list.
~	Delete subsequences of the list.

String Operations

Operation	Effect
[ ]	Returns the character at a given index in the string (as a string).
[:]	Returns a substring (as a copy) denoted by the indices given. Includes the initial index and excludes the second index.
: :	Returns a copy of the string with a single character appended.
@	Returns a copy of the string concatenated with another string.
~	Returns a copy of the string with the specified substring deleted.

Example:

```
int a = 100;
int z = 2;
float f = 50.3892;
char ch = ‘a’;
bool valid = true;
bool not_valid = false;
string str1 = “sample program”;
string str2 = “this is a”;
float[] vals = [];

int b = 5 + a * z;
if (<float> b > f) {
        vals::b;
        print(“b is big\n”);
}
if (valid and not_valid) {
        print(“both true\n”);
}
print(str2 @ str1);
```

```
// sample declarations









// b has the value of 205
// evaluates to true
// adds b to vals.
// prints “b is big\n”

// and operation evaluates to false


// prints “this is a sample program”
```

v. Precedence and Associativity Rules

A majority of the operators in DECAF are left-associative. This means that consecutive operations with the same precedence will be evaluated from left-to-right.

Operator	Description	Level	Associativity

`.`
`[]`
`[:]`
`f(args… )`
	Access object member
	Access list element
	List slicing
	Invoke a method
		1	Left

`+`
`-`
`not`
	Unary plus
	Unary minus
	Logical NOT
		2	Right

`<>`	Casting	3	Right

`*`
`/`
`%`
	Multiplicative	4	Left

`+ -`	Additive	5	Left
`::` `~`	Append value to a list Remove value from a list	6	Right
`@`	List concatenation	7	Right
`< <=` `> >=`	Relational type	8	Left

`==`
`===`
`!=`
`!==`
	Referential equality
	Value equality
	Referential not equals
	Value not equals
		9	Left

`and`	Conditional AND	10	Left
`or`	Condition OR	11	Left
`=`	Assignment	12	Right

vi. Brackets

Brackets are symbols that occur in pairs. They are used to separate blocks of code from each other and to alter default precedence in expressions. The scope of a variable is determined by the outermost brackets in which it is enclosed.

Bracket Type	Usage	Example

{ }	Used to denote control flow, methods, and classes.	`main() -> int {`
		`  print("Hello world!\n");`
		`}`

( )	Used in a variety of situations; such as conditionals, grouping, for loops, tuple instantiation.	`if (x < 5) {`
		`  ...`
		`}`

[ ]	Utilized in List operations (see Lists / Tuples).	`numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];`
< >	Denotes type that operand should be cast to. The type is put between the angle brackets.	`num = <int> 16.0`

vii. White Space

Aside from its function to separate tokens, white space (including ASCII SP, HT, FF, and new line characters) is ignored. Developers can use them freely for code readability.

4. Statements

i. Declarations

Each identifier must be associated with a type. A declaration is a statement which introduces an identifier into the program and specifies its type. If a variable is not initialized at its declaration time, it will be given a default value. The default value is

0 for a number, false for a boolean, empty for built-ins (strings, lists and tuples) and null for an object.

```
string str = “hello world”;
```

Constants
Constants are variables whose value cannot be changed, and must be declared with the keyword const. The variable name should be in all capital letters. The variable can be reassigned but the value is immutable.

```
const int PI = 3.14;
```

```
const int[] IMMUTABLE_LIST = [1, 2, 3];    // cannot modify list
```

ii. Expressions

Adding a semicolon to the end of an expression makes it a statement. When executed, the expression is evaluated.

iii. Control Flow

main

The main method is the start point for execution of the program. Main is the only method that does not need to be defined within a class and exists within the global scope. It takes as parameter the arguments given when the program is run and returns an int representing the return code of the program. By convention, a return value of 0 indicates successful termination whereas a nonzero return value indicates failure.
If a file contains multiple main methods, only the first one will be executed.

```
// Main method
main(string[] args) -> int {
    ...
    return 0;
}
```

if/elseif/else

The most basic element of control flow structures is the if-elseif-else statement, which. In DECAF, at bare minimum an `if` statement is necessary. `else` is used to indicate what should be done if the conditional is not met. `elseif` denotes other possible conditions that the scenario may fall into and executes the respective code. The conditions for decision-making must be boolean values, i.e. true or false.

The format of an if-elseif-else statement is as follows:

```
if (condition) {
    ...
}
elseif (condition) {
    ...
}
else {
    ...
}
```

Example:

```
if (x < 150) {
    print("short");
}
elseif (x < 180) {
    print("medium");
}
else {
    print("tall");
}
```

for loop

The for loop is the most basic loop that enables code to run iteratively. It requires the initialization of some control variable, a range of values at which the control variable should allow the loop to execute, and an expression indicating how the control variable's value should change in between iterations. Failure to provide an expression that alters the control variable's value will result in an infinite loop, which will cause the system to hang.

The format for a for loop is as follows:

```
for (initialization; terminating condition; iteration) {
    ...
}
```

Example:

```
int n = 0;
for (int i = 0; i < 10; i = i + 1) {
    n = n + i;
}
```

while loop

The while loop is the for loop's more sophisticated cousin. It is controlled by a conditional statement that is dependent on the control variable's value. The control variable's value must be altered in the body of the loop in order to avoid getting stuck in an infinite loop.

The format of a while loop is as follows:

```
while (condition) {
    ...
}
```

Example:

```
int x = 5;
int y = 1;
while (x > 1) {
    y = y * x;
    x = x - 1;
}
```

break

`break` is used to indicate when a loop should stop and exit early. It is used within a "for" or "while" loop. If there are nested looping statements, it terminates the execution of the loop whose scope that it is immediately nested within.

continue

`continue` is used to skip the remaining statements not yet executed in the current iteration of the loop, and begins the next iteration of the loop. If there are nested looping statements, it affects the loop whose scope it is immediately nested within.

Example:

```
for (i = 0; i < 6; i = i + 1) {
    if (i === 3) {
        continue;
    }
    elseif (i === 5) {
        print("break at 5\n");
        break;
    }
    print("i is " @ <string> i @
"\n");
}
```

Expected output:

```
i is 0
i is 1
i is 2
i is 4
break at 5
```

return

`return` exits the method and returns the primitive, built-in, or object specified following it to the calling method. If a method does not return anything, the return statement can be omitted. In all other cases, it is required.

5. Sample Executable

The following sample code is meant to simulate an interactive zoo of sorts.

```
class Animal {
    string name;
    string noise;
    int num_feet;

    Animal(string name, string noise, int num_feet) {
        self.name = name;
        self.noise = noise;
        self.num_feet = num_feet;
    }

    listen() -> string {
        return name @ " goes " @ noise;
    }

    get_name() -> string {
        return name;
    }
```

```
    get_feet() -> int {
        return num_feet;
    }
}

main(string args[]) -> int {
    Animal[] zoo = [];
    Animal duck = Animal("Don", "QUACK", 2);
    Animal cow = Animal("Carla", "MOO", 4);
    Animal dog = Animal("Charlie", "WOOF", 3);
    int counter = 0;

    // add animals to the zoo list
    zoo :: duck;
    zoo :: cow;
    zoo :: dog;

    print("Who has the most feet?\n");

    // iterate through all the animals to determine this
    for (Animal a in zoo) {
        if (a.get_feet() > max_feet) {
            the_one = a;
            max_feet = a.get_feet();
        }
    }
    print(the_one.get_name() @ " does!");
    print(" They have " @ <string> the_one.get_feet() @ " feet.\n");

    // loop control flow demonstration
    print("The duck wants attention.\n");
    for (i = 0; i < 5; i = i + 1) {
        print(duck.listen() @ "\n");
    }
    print("Please send help.\n");

    return 0;
}
```