The Decaf language[1]

In this course, we will write a compiler for a simple object-oriented programming language called Decaf. Decaf is a strongly-typed, object-oriented language with support for inheritance and encapsulation. By design, it has many similarities with C/C++/Java, so you should find it fairly easy to pick up. Keep in mind it is not an exact match to any of those languages. The feature set has been trimmed down and simplified to keep the programming projects manageable. Even so, you'll still find the language expressive enough to build all sorts of nifty object-oriented programs.

This document is designed to give the specification for the language syntax and semantics, which you will need to refer to when implementing the course projects.

Lexical considerations

The following are keywords. They are all reserved, which means they cannot be used as identifiers or redefined.

```
void  int  double  bool  string  class  interface  null  this
extends  implements  for  while  if  else  return  break   New  NewArray  Print
ReadInteger  ReadLine
```

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Decaf is case-sensitive, e.g., `if` is a keyword, but `IF` is an identifier; `binky` and `Binky` are two distinct identifiers. Identifiers can be at most 31 characters long.

Whitespace (i.e. spaces, tabs, and newlines) serves to separate tokens, but is otherwise ignored. Keywords and identifiers must be separated by whitespace or a token that is neither a keyword nor an identifier. `ifintthis` is a single identifier, not three keywords. `if(23this` scans as four tokens.

A boolean constant is either `true` or `false`. Like keywords, these words are reserved.

An integer constant can either be specified in decimal (base 10) or hexadecimal (base 16). A decimal integer is a sequence of decimal digits (`0-9`). A hexadecimal integer must begin with `0X` or `0x` (that is a zero, not the letter oh) and is followed by a sequence of hexadecimal digits. Hexadecimal digits include the decimal digits and the letters `a` through `f` (either upper or lowercase). Examples of valid integers:
`8, 012, 0x0, 0X12aE`

A double constant is a sequence of digits, a period, followed by any sequence of digits, maybe none. Thus, `.12` is not a valid double but both `0.12` and `12.` are valid. A double can also have an optional exponent, e.g., `12.2E+2` For a double in this sort of scientific notation, the decimal point is required, the sign of the exponent is optional (if not specified, + is assumed), and the E can be lower or upper case. As above, `.12E+2` is invalid, but `12.E+2` is valid. Leading zeroes on the mantissa and exponent are allowed.

A string constant is a sequence of characters enclosed in double quotes. Strings can contain any character except a newline or double quote. A string must start and end on a single line, it cannot be split over multiple lines:

```
 "this string is missing its close quote
  this is not a part of the string above
```

Operators and punctuation characters used by the language includes:

```
      + - * / % \ < <= > >= = == != && || ! ; , . [] [ ] ( ) { }
```

[1] Decaf is a revision of the SOOP language developed by Maggie Johnson and Steve Freund.

A single-line comment is started by `//` and extends to the end of the line. Multi-line comments start with `/*` and end with the first subsequent `*/`. Any symbol is allowed in a comment except the sequence `*/` which ends the current comment. Multi-line comments do not nest.

Reference grammar

The reference grammar is given in a variant of extended BNF. The meta-notation used:

- `x` (in courier font) means that x is a terminal i.e., a token. Terminal names are also all lowercase except for those few keywords that use capitals.
- *x* (in italic) means y is a nonterminal. All nonterminal names are capitalized.
- ⟨ *x* ⟩ means zero or one occurrence of x, i.e., x is optional
- *x* * means zero or more occurrences of x
- *x* + means one or more occurrences of x
- *x* +, a comma-separated list of one or more x's (commas appear only between x's)
- | separates production alternatives
- ε indicates epsilon, the absence of tokens

For readability, we represent operators by the lexeme that denotes them, such as + or != as opposed to the token (T_NotEqual, etc.) returned by the scanner.

Program ::= *Decl*+
Decl ::= *VariableDecl* | *FunctionDecl* | *ClassDecl* | *InterfaceDecl*
VariableDecl ::= *Variable* `;`
Variable ::= *Type* `ident`
Type ::= `int` | `double` | `bool` | `string` | `ident` | *Type* `[]`
FunctionDecl ::= *Type* `ident (` *Formals* `)` *StmtBlock* |
`void ident (` *Formals* `)` *StmtBlock*
Formals ::= *Variable*+`,` | ε
ClassDecl ::= `class ident` ⟨`extends ident`⟩ ⟨`implements ident`+`,`⟩ `{` *Field** `}`
Field ::= *VariableDecl* | *FunctionDecl*
InterfaceDecl ::= `interface ident {` *Prototype** `}`
Prototype ::= *Type* `ident (` *Formals* `) ;` | `void ident (` *Formals* `) ;`
StmtBlock ::= `{` *VariableDecl** *Stmt** `}`
Stmt ::= ⟨*Expr*⟩ `;` | *IfStmt* | *WhileStmt* | *ForStmt* |
BreakStmt | *ReturnStmt* | *PrintStmt* | *StmtBlock*
IfStmt ::= `if (` *Expr* `)` *Stmt* ⟨`else` *Stmt*⟩
WhileStmt ::= `while (` *Expr* `)` *Stmt*
ForStmt ::= `for (` ⟨*Expr*⟩ `;` *Expr* `;` ⟨*Expr*⟩ `)` *Stmt*
ReturnStmt ::= `return` ⟨*Expr*⟩ `;`
BreakStmt ::= `break ;`
PrintStmt ::= `Print (` *Expr*+`, ) ;`
Expr ::= *LValue* `=` *Expr* | *Constant* | *LValue* | `this` | *Call* | `(` *Expr* `)` |
Expr `+` *Expr* | *Expr* `-` *Expr* | *Expr* `*` *Expr* | *Expr* `/` *Expr* |

		Expr `%` *Expr* | `-` *Expr* | *Expr* `<` *Expr* | *Expr* `<=` *Expr* | *Expr* `>` *Expr* | *Expr* `>=` *Expr* | *Expr* `==` *Expr* | *Expr* `!=` *Expr* | *Expr* `&&` *Expr* | *Expr* `||` *Expr* | `!` *Expr* | `ReadInteger ( )` | `ReadLine ( )` | `New ( ident )` | `NewArray (` *Expr* `,` *Type* `)`
LValue	*::=*	`ident` | *Expr* `.` `ident` | *Expr* `[` *Expr* `]`
Call	*::=*	`ident (` *Actuals* `)` | *Expr* `.` `ident (` *Actuals* `)`
Actuals	*::=*	$Expr^{+,}$ | ε
Constant	*::=*	`intConstant` | `doubleConstant` | `boolConstant` | `stringConstant` | `null`

Program structure

A Decaf program is a sequence of declarations, where each declaration establishes a variable, function, class, or interface. The term *declaration* usually refers to a statement that establishes the identity of a name whereas *definition* means the full description with actual body. For our purposes, the declaration and the definition are one and the same. A program must have a global function named `main` that takes no arguments and returns void. This function serves as the entry point for program execution.

Scoping

Decaf supports several levels of scoping. A declaration at the top-level is placed in the global scope. Each class declaration has its own class scope. Each function has a local scope for its parameter list and another local scope for its body. A set of curly braces within a local scope establishes a nested local scope. Inner scopes shadow outer scopes; for example, a variable defined in a function's scope masks another variable with the same name in the global scope. Similarly, functions defined in class scope shadow global functions of the same name.

- all identifiers must be declared
- upon entering a scope, all declarations in that scope are immediately visible (see note below)
- identifiers within a scope must be unique (i.e. cannot have two functions of same name, a global variable and function of the same name, a global variable and a class of the same name, etc.)
- identifiers re-declared with a nested scope shadow the version in the outer scope (i.e. it is legal to have a local variable with the same name as a global variable, a function within a class can have the same name as a global function, and so on.)
- declarations in the global scope are accessible anywhere in the program (unless they are shadowed by another use of the identifier)
- declarations in closed scopes are inaccessible

Note about visibility and the relationship to lexical structure: As in Java, our compiler operates in more than one pass, the first pass gathers information and sets up the parse tree, only after that is complete do we return and do semantic analysis. A benefit of a two-pass compiler is that declarations are available at the same scope level even before the lexical point at which they are declared. For example, methods of a class can refer to instance variables declared later at the bottom of the class declaration, classes and subclasses and variables of those classes can be placed in the file in any order, and so on. When a scope is entered, all declarations made in that scope are immediately visible, no matter how much further down the file the declaration eventually appears. This rule applies uniformly to all declarations (variables, classes, interfaces, functions) in all scopes.

Types

The built-in base types are integer, double, boolean, string, and void. Decaf allows for "named types" which are objects of class or interface types. Arrays can be construct of any non-void element type, including arrays of arrays.

Variables

Variables can be declared of non-void base type, array type, or named type. Variables declared outside any function have global scope. Variables declared within a class declaration yet outside a function have class scope. Variables declared in the formal parameter list or function body have local scope. A variable is visible from scope entry to exit.

Arrays

Decaf arrays are homogenous, linearly indexed collections. Arrays are implemented as references (pointers). Arrays are declared without size information, and all arrays are dynamically allocated in the heap to the needed size using the built-in `NewArray` operator.

- arrays can be declared of any non-void base type, named type, or array type (including array of arrays)
- `NewArray(N, type)` allocates a new array of the specified type and number of elements, where N must be an integer. N must be strictly positive, a runtime error is raised on an attempt to allocate a negative or zero-length array.
- the number of elements in an array is set when allocated and cannot be changed once allocated
- arrays support the special syntax `arr.length()` to retrieve the number of elements in an array
- array indexing can only be applied to a variable of an array type
- array elements are indexed from 0 to length-1
- the index used in an array selection expression must be of integer type
- a runtime error is reported when indexing a location that is outside the bounds for the array
- arrays may be passed as parameters and returned from functions. The array itself is passed by value, but it is a reference and thus changes to array elements are seen in the calling function.
- array assignment is shallow (i.e. assigning one array to another copies just the reference)
- array comparison (== and !=) only compares the references for equality

Strings

String support is somewhat sparse in Decaf. Programs can include string constants, read strings from the user with the built-in `ReadLine` function, compare strings, and print strings, but that's about it. There is no support for programmatically creating and manipulating strings, converting between strings and other types, and so on. (Consider it an opportunity for extension!) Strings are implemented as references (pointers).

- `ReadLine()` reads a sequence of chars entered by the user, up to but not including the newline
- string assignment is shallow (i.e. assigning one string to another copies just the reference)
- strings may be passed as parameters and returned from functions
- string comparison (== and !=) compares the sequence of characters in the two strings in a case-sensitive manner (behind the scenes we will use an internal library routine to do the work)

Functions

A function declaration includes the name of the function and its associated *type signature*, which includes the return type as well as number and types of formal parameters.

- functions are either global or declared within a class scope; functions may not be nested within other functions
- the function may have zero or more formal parameters
- formal parameters can be of non-void base type, array type, or named type
- identifiers used in the formal parameter list must be distinct
- the formal parameters are declared in a separate scope from the function's local variables (thus, a local variable can shadow a parameter)
- the function return type can be any base, array, or named type. void type is used to indicate the function returns no value
- function overloading is not allowed i.e., the use of the same name for functions with different type signatures
- if a function has a non-void return type, any return statement must return a value compatible with that type
- if a function has a void return type, it may only use the empty return statement
- recursive functions are allowed

Function invocation

Function invocation involves passing argument values from the caller to the callee, executing the body of the callee, and returning to the caller, possibly with a result. When a function is invoked, the actual arguments are evaluated and bound to the formal parameters. All Decaf parameters and return values are passed by value.

- the number of actual arguments in a function call must match the number of formal parameters
- the type of each actual argument in a function call must be compatible with the formal parameter
- the actual arguments to a function call are evaluated from left to right
- a function call returns control to the caller on a return statement or when the textual end of the callee is reached
- a function call evaluates to the type of the function's declared return type

Classes

Declaring a class creates a new type name and a class scope. A class declaration is a list of fields, where each field is either a variable or function. The variables of a class are also sometimes called instance variables or member data and the functions are called methods or member functions.

Decaf enforces object encapsulation through a simple mechanism: all variables are private (scoped to the class and its subclasses, Java calls this access level protected, by the way) and all methods are public (accessible in global scope). Thus, the only way to gain access to object state is via methods.

- all class declarations are global, i.e., classes may not be defined inside a function
- all classes must have unique names
- a field name can be used at most once within a class scope (i.e. cannot have two methods of the same name or a variable and method of the same name)
- fields may be declared in any order
- instance variables can be of non-void base type, array type, or named type
- the use of "this." is optional when accessing fields within a method

Objects

A variable of named type is called an object or an instance of that named type. Objects are implemented as references. All objects are dynamically allocated in the heap using the built-in `New` operator.

- the name used in an object variable declaration must be a declared class or interface name
- the argument to `New` must be a class name (an interface name is not allowable here)
- the . operator is used to access the fields (both variables and methods) of an object
- for method invocations of the form `expr.method()`, the type of `expr` must be some class or interface `T`, `method` must name one of `T` 's methods
- for variable access `expr.var`, the type of `expr` must be some class `T`, `var` must name one of `T`'s variables, and this access must appear with the scope of class `T` or one of its subclasses
- Additional note on that last one: inside class scope, you can access the private variables of the receiving object as well as other instances of that class, but cannot access the variables of other unrelated classes
- object assignment is shallow (i.e. assigning one object to another copies just the reference)
- objects may be passed as parameters and returned from functions. The object itself is passed by value, but it is a reference and thus changes to its variables are seen in the calling function.

Inheritance

Decaf supports single inheritance, allowing a derived class to extend a base class by adding additional fields and overriding existing methods with new definitions. The semantics of A extends B is that A has all the fields (both variables and functions) defined in B in addition to its own fields. A subclass can override an inherited method (replace with redefinition) but the inherited version must match the original in return type and parameter types. Decaf supports automatic upcasting so that an object of the derived class can be provided whenever an object of the base type is expected.

All Decaf methods are dynamically dispatched (i.e. `virtual` for you C++ folks). The compiler cannot determine the exact address of the method that should be called at compile-time (i.e. consider invoking an overridden method on an upcasted object), so instead the dispatch is handled at runtime by consulting a method table associated with each object. We will discuss dispatch tables in more detail.

- if specified, the parent of a class must be a properly declared class type
- all of the fields (both variables and methods) of the parent class are inherited by the subclass
- subclasses cannot override inherited variables
- a subclass can override an inherited method (replace with redefinition) but the inherited must match the original in return type and parameter types
- no overloading: a class can not reuse the same name for another method with a different type signature (whether inherited or not)
- an instance of subclass type is compatible with its parent type, and can be substituted for an expression of a parent type (e.g. if a variable is declared of type Animal , you can assign it from a right-hand side expression of type Cow if Cow is a subclass of Animal. Similarly, if the Binky function takes a parameter of type Animal, it is acceptable to pass a variable of type Cow or return a Cow from a function that returns an Animal). The inverse is not true (the parent cannot be substituted where the subclass is expected).
- the previous rule applies across multiple levels of inheritance as well
- it is the compile-time declared type of an object that determines its class for checking for fields (i.e. once you have upcast a Cow to an Animal variable, you cannot access Cow-specific additions from that variable)

- there is no subtyping of array types: if class T2 extends T1, an array of T2[] is not compatible with an array of T1[] (this is unlike Java).

Interfaces

Decaf also supports subtyping by allowing a class to implement one or more interfaces. An interface declaration consists of a list of function prototypes with no implementation. When a class declaration states that it implements an interface, it is required to provide an implementation for every method specified by the interface. (Unlike Java/C++, there are no abstract classes). Each method must match the original in return type and parameter types. Decaf supports automatic upcasting to an interface type.

- each interface listed in the implements clause must be a properly declared interface type
- all methods of the interface must be implemented if a class states that it implement the interface
- each method must match the original in return type and parameter types
- the class declaration must formally declare that it implements an interface, just implementing the required methods does not count as being a subtype
- an instance of the class is compatible with any interface type(s) it implements, and can be substituted for an expression of the interface type (e.g. if a variable is declared of interface type Colorable, you can assign it from a right-hand side expression of type Shape if the Shape class implements the Colorable interface. Similarly, if the Binky function takes a parameter of type Colorable, it is acceptable to pass a variable of type Shape or return a Shape from a function that returns a Colorable). The inverse is not true (the interface cannot be substituted where the class is expected).
- a subclass inherits all interfaces of its parent, e.g. if Rectangle inherits from Shape which implements Colorable, then Rectangle is also compatible with Colorable.
- it is the compile-time declared type of an object that determines its type for checking for fields (i.e. once you have upcast a Shape to a Colorable variable, you cannot access Shape-specific additions from that variable)
- there is no subtyping of array types: if class T2 implements T1, an array of T2[] is not compatible with an array of T1[] (this is unlike Java).

Type equivalence and compatibility

Decaf is a (mostly) strongly-typed language: a specific type is associated with each variable, and the variable may contain only values belonging to that type's range of values. If type A is equivalent to B, an expression of either type can be freely substituted for the other in any situation. Two base types are equivalent if and only if they are the same exact type. Two array types are equivalent if and only if they have the same element type (which itself may be an array, thus implying a recursive definition of structural equivalence is used here). Two named types are equivalent if and only if they are the same name (i.e. named equivalence not structural).

Type compatibility is a more limited unidirectional relationship. If Type A is compatible with B, then an expression of type A can be substituted where an expression of type B was expected. Nothing is implied about the reverse direction. Two equivalent types are type compatible in both directions. A subclass is compatible with its parent type, but not the reverse. A class is compatible with any interfaces it implements. The null type is compatible with all named types. Operations such as assignment and parameter passing allow for not just equivalent types but compatible types.

Assignment

For the base types, Decaf uses value-copy semantics; the assignment `LValue = Expr` copies the value resulting from the evaluation of `Expr` into the location indicated by `LValue`. For arrays, objects and strings, Decaf uses reference-copy semantics; the assignment `LValue = Expr` causes `LValue` to contain

a reference to the object resulting from the evaluation of `Expr` (i.e., the assignment copies a pointer to an object rather than the object). Said another way, assignment for arrays, objects, and strings makes a shallow, not deep, copy.

- an LValue must be an assignable variable location
- the right side type of an assignment statement must be compatible with the left side type
- `null` can only be assigned to a variable of named type
- it is legal to assign to the formal parameters within a function, such assignments affect only the function scope

Control structures

Decaf control structures are based on the C/Java versions and generally behave somewhat similarly.

- An `else` clause always joins with the closest unclosed `if` statement
- the expression in the test portions of the `if` `while` and `for` statements must have bool type
- a `break` statement can only appear within a `while` or `for` loop
- the value in a `return` statement must be compatible with return type of the enclosing function

Expressions

For simplicity, Decaf does not allow co-mingling and conversion of types within expressions (i.e. adding an integer to a double, using an integer as a boolean, etc.).

- constants evaluate to themselves (true, false, null, integers, doubles, string literals)
- `this` is bound to the receiving object within class scope, it is an error outside class scope
- the two operands to binary arithmetic operators (`+`,`-`,`*`,`/`, and `%`) must either be both int or both double. The result is of the same type as the operands.
- the operand to a unary minus must be int or double. The result is the same type as the operand.
- the two operands to binary relational operators (`<`, `>`, `<=`, `>=`) must either both be int or both double. The result type is bool.
- the two operands to binary equality operators (`!=`, `==`) must be of equivalent type (two ints, two arrays of double, etc.) (see exception below for objects) The result type is bool
- the two operands to binary equality operands can also be two objects or an object and null. The types of the two objects must be compatible in at least one direction. The result type is bool.
- the operands to all binary and unary logical operators (`&&`, `||` , and `!`) must be of bool type. The result type is bool.
- logical `&&` and `||` do not short-circuit; both expressions are evaluated before determining the result
- the operands for all expressions are evaluated left to right

Operator precedence from highest to lowest:

`[ .`	(array indexing and field selection)
`! -`	(unary -, logical not)
`* / %`	(multiply, divide, mod)
`+ -`	(addition, subtraction)
`< <= > >=`	(relational)
`== !=`	(equality)
`&&`	(logical and)
`||`	(logical or)
`=`	(assignment)

All binary arithmetic operators and both binary logical operators are left-associative. Assignment and the relational operators do not associate (i.e. you cannot chain a sequence of these operators that are at the same precedence level: `a < b >= c` should not parse, however `a < b == c` is okay). Parentheses may be used to override the precedence and/or associativity.

Standard library functions

Decaf has a very small set of routines in its standard library that allow for simple I/O and memory allocation. The library functions are `Print`, `ReadInteger`, `ReadLine`, `New`, and `NewArray`.

- the arguments passed to a `Print` statement can only be string, int, or bool
- the argument to `New` must be a properly declared class name
- the first argument to `NewArray` must be of integer type, the second any non-void type
- the return type for `NewArray` is array of T where T is the type specified as the second argument
- `ReadLine()` reads a sequence of chars entered by the user, up to but not including the newline
- `ReadInteger()` reads a line of text entered by the user, and converts to an integer using `atoi` (returns 0 if the user didn't enter a valid number)

Decaf linking

Given Decaf does not allow separate modules or declaration separate from definition, there is not much work beyond semantic analysis to ensure we have all necessary code. The one task our "linker" needs to perform is to verify that that there was a declaration for the global function `main`. This will be done as part of code generation.

Run time checks

There are only two runtime checks that are supported by Decaf (leaving room for extension!).

- the subscript of an array must be in bounds, i.e. in range 0 . . arr.length() –1
- the size passed to `NewArray` must be strictly positive

When a run-time errors occurs, an appropriate error message is output to the terminal and the program terminates.

Things Decaf doesn't do!

By design, Decaf is a simple language and although it has all the features necessary to write a wide variety of object-oriented programs, there are things that C++ and Java compilers do that it does not.

- doesn't mandate what the values for uninitialized variables are or what the value of variables or elements in a newly allocated object or array are
- doesn't check for use of uninitialized variables
- doesn't detect (either compile or runtime) a function that is declared to return a value but fails to do so before falling off the textual end of the body
- doesn't check for an object or array use before it was ever allocated
- doesn't detect call methods or accessing variables of a null object
- doesn't detect unreachable code
- has no deallocation function and no garbage collection, so dynamic storage is never reclaimed
- no support for object constructors or destructors
- and many others…