Decaf Language Reference Manual

Decaf is a small object oriented language with arrays, overloading, inheritance, and static method and field resolution. It is inspired by the popular JavaTM programming language, and inherits many features of Java.

This manual outlines the syntax and semantics of the language constructs in Decaf. The syntax is described in Extended Backus-Naur Form (EBNF) notation. In EBNF, which combines regular-expression-like notation with grammars, x^* stands for a sequence of zero or more x's; x^+ stands for a sequence of one or more x's; $(x \mid y)$ stands for choice between x and y; and $x^?$ stands an optional occurrence (i.e., zero or one) of x. In the following, symbols in **bold face** represent reserved words and special characters: i.e., tokens with unique lexemes, such as **while**. Symbols in *italics* are either nonterminal grammar symbols, or terminal symbols with attributes, such as *int_const*.

1 Lexical Issues

Decaf is *case-sensitive*; for example, for and For are treated as distinct lexical entities.

White Space and Comments

Whitespace (blanks, newlines and tabs) serve to separate tokens; otherwise they are ignored. Whitespace may not appear within any token except a string constant (see below).

Decaf supports two styles of comments:

- Multi-line (C-style) comments that begin with "/ *" and end with "*/". These comments may not be nested.
- Single-line comments that start with "//" and terminate at the end of line.

Comments may appear wherever a whitespace may appear.

Reserved words

The following are reserved words.

boolean	break	continue	class	else	extends
false	float	for	if	int	new
null	private	public	return	static	super
this	true	void	while		

Constants

There are three types of constants supported by Decaf: integer, floating point and string constants.

Integer constants are made up of one or more digits, each digit ranging from 0 thru 9.

Floating point constants contain a decimal point (*e.g.*, 3.14159). There must be at least one digit before *and* after the decimal point.

String constants begin and end with a double quote ("). Newlines may not appear within a string. If the string itself contains a double quote, it is "escaped" with a backslash (\) as in the example string: "\"What?\" she exclaimed.". Escape sequences, such as $\n and \t are used to place special characters such as newlines and tabs in a string. If the string contains a backslash, that is escaped too (e.g., "The computer simply responded with <math>\"A:\>\""$). Strings must be contained within a single line.

Decaf does not support character constants.

Integer, floating point and string constants are denoted in the syntax descriptions below, by *int_const*, *float_const* and *string_const* respectively.

Identifiers

Letters denote the upper and lower case elements of the English alphabet (a thru z and A thru Z).

An identifier is a sequence of letters, digits and underscore (_), starting with a letter, that is not one of the reserved words. Identifiers are denoted by the symbol *id*.

2 Declarations

A Decaf program is a sequence of **class** declarations.

program ::= $class_decl^*$

Class Declarations

A class declaration associates a set of fields, methods and constructors to a class name. Its syntax is:

class_decl ::= class id (extends id)[?] { class_body_decl⁺ } class_body_decl ::= field_decl | method_decl | constructor_decl Each class declaration (e.g., **class foo** ...) creates a new class with the given name (e.g., **foo**). The **extends** option is used to specify a superclass, from which the current class inherits its fields and methods. *Field* declarations specify the fields that objects in this class will have; *method* declarations specify the methods that can be used on objects in this class; and *constructor* declarations are used to specify any initialization that needs to be done when a new object in this class is created.

Fields

The syntax of field declarations in Decaf is:

Field declarations may be prefixed with modifiers as specified by the above syntax. A **public** field is a variable that can be accessed from methods defined in any other class. A **private** field is variable that can be accessed only from methods defined in the same class. If a field is not explicitly specified as private or public, it is assumed to be private.

Fields that are declared **static** are called *class variables*; fields that are not declared **static** are called *instance variables*. Each instance of a class gets its own private copy of all instance variables declared in the class. However, all instance of a class share a single copy of the class variables declared in the class.

A field may be one of the predefined types (int, float, boolean) or it may be an object in a user-defined class.

Multiple fields, all of same type, may be declared in a single statement; in this case, field names are separated by commas.

Array fields are defined by specifying [] after the field name. Note that, as in Java, this only *declares* the array, and does not *create* one, i.e., allocate space for it. The array will be created, dynamically, using the **new** construct, described later in this manual.

In Java, the number of []'s specify the number of dimensions of the array. *In contrast, Decaf allows only single-dimensional arrays.*

Inheritance: All objects of a class c contain instance fields defined in c as well as instance fields defined in all superclasses of c. A class may contain a new field whose name is identical to one in a superclass. However, only one field of a given name can be defined in a class, irrespective of the types.

Methods and Constructors

Methods in a class encapsulate procedures for manipulating objects belonging to a class. Constructors in a class are used to initialize objects of the class whenever they are created. The syntax of method and constructor declarations is as follows:

method_decl	::=	modifier (type $ $ void) id (formals [?]) block	
constructor_decl	::=	modifier id (formals?) block	
formals	::=	formal_param (, formal_param)*	
formal_param	::=	type variable	

The **public** and **private** modifiers are used to specify whether the given method can be accessed from methods in other classes. Methods declared **static** are called *class methods* and those not declared **static** are called *instance methods*. Instance methods always operate on a specific instance of the class, whereas class methods operate on the class as a whole and not on individual instances. A method that returns nothing (i.e., a procedure) has a return type of **void**. Methods may take a list of arguments, the name and type of which are given by a list of formal parameters.

There may be multiple methods of the same name defined within a class, as long as they can be distinguished on the basis of the number or types of parameters.

The name of a constructor must be same as that of the parent class. Constructors may also take a list of arguments, again specified as by the list of formal parameters.

The body of a method or constructor is specified by a *block* of statements, described below.

3 Statements

Decaf supports a small but expressive set of control primitives to specify procedures for manipulating the objects.

The syntax of Decaf statements is:

```
block ::= { stmt* }
stmt ::= if ( expr ) stmt (else stmt)?
    while ( expr ) stmt
    for ( stmt_expr? ; expr? ; stmt_expr? ) stmt
    return expr? ;
    stmt_expr ;
    break ;
    continue ;
    block
    var_decl
    ;
```

A block of statements is comprised of a (possibly empty) set of declarations for variables local to that block, followed by a sequence of statements.

If statement: An if-statement of the form if (*expr*) *stmt* evaluates the boolean expression *expr*; if the expression evaluates to **true**, then the statement *stmt* is executed. When the optional **else** part is

present, of the form **else** *stmt'*, then the statement *stmt'* is executed if the expression evaluates to **false**.

- While statement: A while-statement of the form while (*expr*) stmt loops, evaluating *expr* first and executing *stmt* as long as *expr* is true. The loop is exited when *expr* evaluates to false.
- For statement: A for-statement of the form for $(s_1; e_2; s_3)$ stmt₄ has four components: an initializer s_1 , which is evaluated on entry to the statement; the boolean expression e_2 , which is evaluated at the beginning of each iteration through the loop; an update part s_3 , which is executed at the end of each iteration through the loop; and statement stmt₄, which is executed as long as e_2 evaluates to true. The loop is exited when e_2 evaluates to false.
- **Return statement:** A return statement signifies the end of control in a method. The expression *expr* is evaluated and is returned to the caller of the method. Thus the declared return type of the method must be compatible with the type of *expr*. If the return type of the method is **void**, then the *expr* part of the return statement is omitted.
- **Expression statement:** Certain expressions, such as assignments and post-increment operations, denoted by *stmt_expr*, can be stand-alone statements. See Section 4 for syntax of *stmt_expr*.
- **Block statement:** A block of statements can themselves be used wherever a statement is used. When a block statement is executed, the local variables declared in that block are freshly created, and the sequence of statements in that block are executed in order.

Empty statement: An empty statement is specified simply by ';'. It has no effect on execution.

4 Expressions

Objects and their values are manipluated using a variety of expressions. The simplest form of expressions are literal constants:

literal ::= int_const | float_const | string_const | null | true | false

In the following, the nonterminal grammar symbol *expr* is used to denote the set of all expressions allowed in Decaf. Literal constants, creation of new instances of objects, access of object fields and arrays, and method invocation are among the basic set of expressions used in Decaf, called *primary* expressions, the syntax of which is given below:

primary literal ::= this super (expr) **new** *id* (*arguments*[?]) *method_invocation* lhs arguments ::= $expr(, expr)^*$ lhs ::= field_access array_access ::= primary.id field_access id array_access ::= primary [expr] method_invocation ::= field_access ($arguments^{?}$)

The reserved word **this** is used to represent the current object, i.e., the object on which the method containing the current expression is applied. The reserved word **super** is used to explicitly access fields or methods in the superclass; this construct is primarily used when names alone are insufficient to distinguish fields and methods in a class from those of its superclass.

A new object is created using the reserved word **new**. The arguments supplied with the name of the class correspond to parameters of the constructor to be used to initialize the new object.

An *lhs* expression is an expression that can occur on the left hand side of an assignment: either referring to a field of an object, or to an element of an array.

A field in an object of class c can be accessed by simply using the name of the field, provided it is being accessed from a method defined in class c. If there are multiple objects of class c, or if the field is accessed from a method defined in a different class, the field is specified using the notation "*object.field*". A method invocation takes the form "*object.method(args)*". If a method invocation does not explicitly specify an object, then **this** is assumed as the default object.

The syntax of array access resembles that of array accesses in C: *array[index]*. Multidimensional arrays are accessed by specifying a sequence of indices, each index enclosed in square brackets. An array is assumed to be created before it is accessed.

Other expressions in Decaf are defined using the primary expressions described above.

expr ::= primary | assign $| new_array$ $| expr arith_op expr$ $| expr bool_op expr$ $| unary_op expr$ assign ::= lhs = expr $new_array ::= new type [expr]$ $arith_op \in \{+, -, *, /\}$ $bool_op \in \{\&\&, ||, ==, !=, <, >, <=, >= \}$ $unary_op \in \{+, -, !\}$

An array is created with the **new** operator by specifying the type of array and its size. The size is, in general, integer expressions (not just integer constants).

Decaf provides a set of commonly used arithmetic and relational operators. Binary arithmetic operators include addition (+), subtraction (-), multiplication (*) and division (/), which operate on integers or floating point numbers. Boolean operators include:

- equality and disequality operations (==, !=) that can be applied, in addition to integer and floating point expressions, to objects of any type, as long as both objects being compared are of the same type;
- common relational operations (>, <, >= and <=) on integer and floating point expressions; and
- logical operations conjunction (&&) and disjunction (||).

Unary operators include unary plus (+), minus (-) that operates on integer and floating point expressions and negation (!) that operates on boolean expressions.

The explicit assignment (=) associates to the right. Relational operators (>, <, >= and <=) are nonassociative. All other binary operators associate to the left. Hence, the expression a + b + c + d is treated as ((a + b) + c) + d, while the expression a = b = c = d is treated as a = (b = (c = d)). The precedence of the operators is defined by the following table, which lists the operators from highest to lowest precedence.

highest

*, / +, -<, >, <=, >= ==, != && line lowest =

1

Finally, statement expressions are those, such as assignments and method invocations, that can occur as stand-alone statements. Their syntax is defined by the following rule:

stmt_expr ::= assign method_invocation

5 Standard Objects and Methods

Decaf provides two standard objects: Out which represents the standard output file (the console) and In which represents the standard input file (the keyboard). The methods that can be used on these objects are:

1. print(*expr*): This is an overloaded method, which can take objects of any primitive type (integers, floating point numbers and booleans) as well as string constants, and writes their values to the given file.

The method is usually applied to Out and returns void.

- 2. scan_int(): This method, usually applied to In, reads a stream of characters from file that represent an integer and returns the corresponding integer value.
- 3. scan_float(): Similar to scan_int(), this method is used to read a floating point value from
 the given file.

Entry Point

A class whose name is same as the filename of the program is the "main" object of the program. For instance, if the program is in file foo.decaf, then class foo is the main class in the program. A static, public, parameterless method called main in class foo is the entry point of the program in foo.decaf. That is, when invoked from the command line, main() will be the first method invoked from the operating system.

6 Sample Decaf Programs

Following are four sample programs of varying complexity written in Decaf.



nrfib.decaf

The following program computes and prints the n^{th} Fibonacci number, given n as the input. The Fibonacci number is computed using a nonrecursive procedure.

```
class nrfib{
    public static void main() {
        int n, i, fn, fn_prev;
        n = In.scan_int();
        fn = 1;
        fn_1 = 0;
        for(i=1; i<n; i=i+1) {
            fn = fn_prev + fn;
               fn_prev = fn - fn_prev;
        }
        Out.print("Fib = ");
        Out.print(fn);
        Out.print("\n");
    }
}</pre>
```

rfib.decaf

The following program computes and prints the n^{th} Fibonacci number, given n as the input. The Fibonacci number is computed using a *recursive* procedure.

```
class rfib{
    static int fib(int n) {
        if (n <= 2)
            return 1;
        else
            return fib(n-1) + fib(n-2);
    }
    public static void main() {
        int n;
        n = In.scan_int();
        Out.print("Fib = ");
        Out.print(fib(n));
        Out.print("\n");
    }
}</pre>
```

IntList.decaf

This program implements an abstract datatype of (singly-linked) list with integer elements, with operations of insertion, search and length.

```
class IntList{
 int value;
 IntList next;
 public static IntList create_list(int v) {
   IntList new_element;
   new_element = new IntList();
   new_element.value = v;
   new element.next = null;
   return new_element;
  }
 public IntList insert(int v) {
   IntList new_element;
   new_element = create_list(v);
   new_element.next = this;
   return new_element;
  }
 public boolean search(int v) {
   if (this.value == v)
      { /* head of list matches */
       return true;
      }
    else
      /* not at head, so search rest of list */
      if (next == null)
        { /* end of list, so search fails */
          return false;
        }
      else
        /* search rest of the list */
       return next.search(v);
  }
 public int length() {
   if (next == null) return 1;
     else return 1 + next.length();
  }
```

Acknowledgements

Decaf was first used as the source language for the Compiler Design course in Fall'96. Many thanks are due to those students who gracefully put up with the inconsistencies of the first version. Based on that version of Decaf, R. Sekar (then of Iowa State University) defined Espresso, also used in a Compilers course; his design was used to remove some irregularities from Decaf. Kevin Kreeger (Fall'96 class) also suggested several simplifications of Decaf's grammar that have been included in the current version. Many thanks are due also to Tord Johnson (Fall'98 class) for finding and reporting many errors in an early draft of this manual.