



MANCHESTER METROPOLITAN UNIVERSITY

6G6Z1110 Programming Languages: Principles and Design (Term 2)

Compilers Coursework Supplementary Documentation

Decaf Language Specification

These materials were adapted from an MIT opencourseware module on compilers (Terms of use: <https://ocw.mit.edu/terms/>).

Table of Contents

Decaf (Language Specification)	2
Lexical Considerations	2
Reserved Words	2
Comments and Whitespace	2
Identifiers	2
Characters and Strings	2
Numbers	2
Reference Grammar	3
Semantic Considerations	5
Types	5
Scope	5
Locations	5
Assignment	5
Method Invocation and Return	6
Control Statements	6
Expressions	6
Library Callouts	7
Semantic Rules	8
Runtime Checking	8



Decaf (Language Specification)

Your coursework for this unit is to write a compiler for a language called Decaf. Your compiler and your code will be assessed during your timetabled summer exam, after term. You will work on your compiler all term as well as some lab exercises, designed for you to learn and practice the concepts and methods covered in the lectures, which are required to be able to complete this coursework successfully.

Decaf is a simple imperative language with Java-like syntax and constructs. For practical reasons, Decaf only implements **integers** and **booleans** for data types (no floats, doubles or complex types like collections, classes and objects, etc). Strings and chars are constant, literal values used as **immutable data**.

This document details a precise specification for the Decaf language – it must be strictly adhered to.

Lexical Considerations

Reserved Words

```
boolean break callout class continue else false for if int return true void
```

Comments and Whitespace

Comments are started by `//` and are terminated by a **new line**. White space may appear between any lexical tokens. White space is defined as **one or more spaces, tabs, form-feed, page and line-breaking characters**, and **comments**.

Identifiers

Reserved words and identifiers are case-sensitive. For example, `if` is a keyword, but `IF` is a variable name; `foo` and `Foo` are two different names referring to two distinct variables.

Reserved words and identifiers must be **separated by white space**, or a token that is neither a keyword nor an identifier. For example, `thisfortrue` is a single identifier, not three distinct keywords.

Characters and Strings

A char is **any printable ASCII character** (ASCII values between decimal value 32 and 126, or octal 40 and 176) **other than quote ("")**, **single quote ('')**, or **backslash (\)**. A char can also be **any of the 2-character sequences** “`\"`” to denote quote, “`\'`” to denote single quote, “`\\`” to denote backslash, “`\t`” to denote a literal tab, or “`\n`” to denote newline.

Strings are composed of char(s) enclosed in **double quotes**. **Chars** consists of a char enclosed in **single quotes**.

Numbers

Numbers in Decaf are **64-bit signed**. That is, decimal values between **-9223372036854775807** and **9223372036854775806** (or -7FFFFFFFFFFFFF and 7FFFFFFFFFFFFF in hex).

If a sequence begins with `0x`, then these first two characters and the longest sequence of characters drawn from [0-9a-fA-F] form a hexadecimal integer literal.

If a sequence begins with a decimal digit (but not `0x`), then the longest prefix of decimal digits forms a decimal integer literal.

Note that range checking is performed later, during semantic analysis. A long sequence of digits (e.g. 9223372036854775807123456789) is still scanned as a single token.



Notation	Description
<foo>	Non-terminal symbol
foo	bold font means terminal symbol
[x]	Zero or one x note: brackets in quotes '[' are terminals
x*	Zero or more x
x+,	Comma separated list of one or more x
{ }	Grouping note: braces in quotes '{}' are terminals
	OR

To interpret the Decaf reference grammar below, please refer to the notation table above.

Reference Grammar

<program>	→	class Program '{' <field_decl>* <method_decl>* '}'
<field_decl>	→	<type> { <id> <id> '[' <int_literal> ']' }+, ;
<method_decl>	→	{ <type> void } <id> ([{ <type> <id> }+,]) <block>
<block>	→	'{' <var_decl>* <statement>* '}'
<var_decl>	→	<type> <id>+, ;
<type>	→	int boolean
<statement>	→	<location> <assign_op> <expr> ; <method_call> ; If (<expr>) <block> [else <block>] for <id> = <expr>, <expr> <block> return [<expr>] ; break ; continue ; <block>
<assign_op>	→	= += -=
<method_call>	→	<method_name> ([<expr>+,]) callout (<string_literal> [, <callout_arg>+,])
<method_name>	→	<id>
<location>	→	<id> <id> '[' <expr> ']'
<expr>	→	<location> <method_call> <literal> <expr> <bin_op> <expr> - <expr> ! <expr> (<expr>)
<callout_arg>	→	<expr> <string_literal>
<bin_op>	→	<arith_op> <rel_op> <eq_op> <cond_op>
<arith_op>	→	+ - * / %
<rel_op>	→	< > <= >=
<eq_op>	→	== !=
<cond_op>	→	&&



```
<literal>      →      <int_literal> | <char_literal> | <bool_literal>
    <id>        →      <alpha> <alpha_num>*
<alpha_num>     →      <alpha> | <digit> |
    <alpha>       →      a | b | c | ... | z | A | B | C | ... | Z | _
    <digit>       →      0 | 1 | 2 | ... | 9
<hex_digit>     →      <digit> | a | b | c | ... | f | A | B | C | ... | F
<int_literal>    →      <decimal_literal>
                  |
<decimal_literal> →      <digit> <digit>*
<hex_literal>    →      0x <hex_digit> <hex_digit>*
<bool_literal>   →      True | false
<char_literal>   →      '<char> '
<string_literal> →      "<char>+ "
```



Semantic Considerations

A Decaf program consists of a single class declaration for a **class called Program**. The class declaration consists of field declarations and method declarations.

Field declarations introduce variables that **can be accessed globally** by all methods in the program.

Method declarations introduce functions/procedures.

The program must contain a declaration for a **method called main** that has no parameters. Execution of a Decaf program **starts at method main**.

Types

There are two basic types in Decaf – **int** and **boolean**. Practically, booleans and integers will be implemented in memory as 64-bit signed integer values.

In Decaf, there are arrays of integers (`int [N]`) and arrays of booleans (`boolean [N]`). **Arrays must be declared in the global scope**. All arrays are **one-dimensional** and have a **compile-time fixed size**. Arrays are **indexed from 0 to N - 1**, where $N > 0$ is the size of the array. The usual **square bracket notation is used to index arrays**. Since arrays have a compile-time fixed size and cannot be declared as parameters (or local variables), **there is no facility for querying the length of an array** variable in Decaf.

Scope

Decaf has simple and quite restrictive scope rules:

- ◆ a variable must be **declared before it is used**
- ◆ a method can be **called only by code appearing after its header**. (Note that **recursive methods are allowed**)

There are **only two valid scopes** at any point in a Decaf program: the **global scope**, and the **method scope**. The global scope consists of names of fields and methods introduced in the (single) Program class declaration. The method scope consists of names of formal parameters introduced by the method declaration.

An identifier introduced in a method scope can have the same name as an identifier from the global scope.

No identifier may be defined **more than once in the same scope**.

Locations

Decaf has two kinds of locations: **local/global scalar variables** and **global array elements**. Each location has a type. Locations of types int and boolean contain integer values and boolean values, respectively. Locations of types `int [N]` and `boolean [N]` denote array elements.

To enable straightforward code generation, **arrays may be allocated on the heap**.

Assignment

Assignment is **only permitted for scalar values**. For the types int and boolean, Decaf uses value-copy semantics, and the assignment `location = expr` copies the value resulting from the evaluation of `expr` into `location`. The assignment `location += expr` increments the value stored in `location` by `expr`. The assignment `location -= expr` decrements the value stored in `location` by `expr`.

It is legal to assign to a formal parameter variable within a method body. Such assignments affect only the method scope.



Method Invocation and Return

Method invocation involves:

- ◆ passing argument values from the caller to the method
- ◆ executing the body of the method
- ◆ returning to the caller, possibly with a result

Argument passing is defined in terms of assignment: the formal arguments of a method are considered **local variables of the method** and are **initialized to the values resulting from the evaluation of the argument expressions**. The arguments are **evaluated from left to right**.

The body of the method is then executed by executing the statements of its method body in sequence.

A method returning void **cannot be used in an expression**. A method that returns a result **may be called as part of an expression**, in which case the result of the call is the result of evaluating the expression in the return statement when this statement is reached.

Control Statements

if

The if statement has the usual semantics. First, the expr is evaluated. If the result is true, the true arm is executed. Otherwise, the else arm is executed, **if it exists**. Since Decaf requires that the true and else arms be enclosed in braces, there is no ambiguity in matching an else arm with its corresponding if statement.

For

The for statement is similar to a do loop in Fortran. The **id** is the **loop index variable**. The first expr is the initial value of the loop index variable and the second expr is the ending value of the loop index variable. Each of these expressions are **evaluated once, just prior to reaching the loop for the first time**. The loop body is **executed if the current value of the index variable is less than the ending value**. After an execution of the loop body, the **index variable is incremented by 1**, and the **new value is compared to the ending value** to decide if another iteration should execute.

Expressions

Expressions follow the normal rules for evaluation. In the absence of other constraints, operators with the same precedence are evaluated from left to right. Parentheses may be used to override normal precedence.

A location expression evaluates to the value contained by the location.

Method invocation expressions are discussed in Method Invocation and Return. Array operations are discussed in Types. I/O related expressions are discussed in Library Callouts.

Integer literals evaluate to their integer value. Character literals evaluate to their integer ASCII values, e.g., 'A' represents the integer 65. (The type of a character literal is int.)

The arithmetic operators (arith_op and unary minus) have their usual precedence and meaning, as do the relational operators (rel_op). % computes the remainder of dividing its operands.

Relational operators are used to compare integer expressions. The equality operators, == and != are defined for int and boolean types only, can be used to compare any two expressions having the same type. (== is "equal" and != is "not equal").

The result of a relational operator or equality operator has type boolean.



The boolean connectives `&&` and `||` are interpreted using short circuit evaluation as in Java. The side-effects of the second operand are not executed if the result of the first operand determines the value of the whole expression (i.e., if the result is false for `&&` or true for `||`).

Operator precedence, from highest to lowest:

Operators	Description
-	unary minus
!	logical not
* / %	multiplication, division, remainder
+ -	addition
< <= >= >	relational
== !=	equality
&&	conditional and
	conditional or

Note that this precedence is not reflected in the reference grammar.

Library Callouts

Decaf includes a primitive method for calling functions provided in the runtime system, such as the standard C library or user-defined functions.

The function named by the initial string literal is called and the arguments supplied are passed to it. Expressions of boolean or integer type are passed as integers; string literals or expressions with array type are passed as pointers. The return value of the function is passed back as an integer. The user of callout is responsible for ensuring that the arguments given match the signature of the function, and that the return value is only used if the underlying library function returns a value of appropriate type. Arguments are passed to the function in the system's standard calling convention.

In addition to accessing the standard C library using callout, an I/O function can be written in C (or any other language), compiled using standard tools, linked with the runtime system, and accessed by the callout mechanism.



Semantic Rules

These rules place additional constraints on the set of valid Decaf programs besides the constraints implied by the grammar. A program that is grammatically well-formed and does not violate any of the following rules is called a legal program. A robust compiler will explicitly check each of these rules and will generate an error message describing each violation it is able to find. A robust compiler will generate at least one error message for each illegal program but will generate no errors for a legal program.

1. No identifier is declared twice in the same scope.
2. No identifier is used before it is declared.
3. The program contains a definition for a method called main that has no parameters (note that since execution starts at method main, any methods defined after main will never be executed).
4. The int_literal in an array declaration must be greater than 0.
5. The number and types of arguments in a method call must be the same as the number and types of the formals, i.e., the signatures must be identical.
6. If a method call is used as an expression, the method must return a result.
7. A return statement must not have a return value unless it appears in the body of a method that is declared to return a value.
8. The expression in a return statement must have the same type as the declared result type of the enclosing method definition.
9. An id used as a location must name a declared local/global variable or formal parameter.
10. For all locations of the form id[expr]
 - a. id must be an array variable
 - b. the type of expr must be int
11. The expr in an if statement must have type boolean.
12. The operands of arith_op and rel_op must have type int.
13. The operands of eq_op must have the same type, either int or boolean.
14. The operands of cond_op and the operand of logical not (!) must have type boolean.
15. The location and the expr in an assignment, location = expr, must have the same type.
16. The location and the expr in an incrementing/decrementing assignment, location += expr and location -= expr, must be of type int.
17. The initial expr and the ending expr of for must have type int.
18. All break and continue statements must be contained within the body of a for.

Runtime Checking

In addition to the constraints described above, which are statically enforced by the compiler's semantic checker, the following constraints are enforced dynamically: the compiler's code generator must emit code to perform these checks; violations are discovered at run-time.

1. The subscript of an array must be in bounds.
2. Control must not fall off the end of a method that is declared to return a result.

When a run-time error occurs, an appropriate error message is output to the terminal and the program terminates. Such error messages should be helpful to the programmer trying to find the problem in the source program.