Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.110, Spring 2025 Handout — Decaf Language Monday, Feb 3

The project for the course is to write a compiler for a language called Decaf. Decaf is a simple imperative language highly similar to C.

This handout describes the language syntax (i.e., the set of legal Decaf programs) and the language semantics (ascribing a meaning to each legal program). You *may* make extensions that expand the set of legal programs, provided that you include a clear description of the new programs that are accepted, and of their (reasonable) semantics; these extensions may not include syntactically standard Decaf programs explicitly ruled out by the semantic rules described in Section 5. A reasonable extension could include allowing new datatypes, or new language constructs. You may not make any extensions that change the semantics of legal programs or that rule out any legal programs.

1 Updates to Spec

- **2/8/2025:** There was an ambiguity in the definitions of `field_decl` and `array_field_decl`. Please see update spec and test case legal-51.dcf in the parser tests

2 Lexical Considerations

All Decaf keywords are lowercase. Keywords and identifiers are case-sensitive. For example, `if` is a keyword, but `IF` is an identifier; `foo` and `Foo` are two different identifiers referring to two distinct variables or methods.

The reserved words are:

`bool` `break` `import` `continue` `else` `false` `for` `while` `int` `long` `return` `len` `true` `void`

Comments are started by `//` and are terminated by the end of the line. Block comments are started by `/*` and are terminated by a matching `*/`. You do not have to handle nested block comments (e.g., `/* /* */ */`).

White space may appear between any lexical tokens (but not within a single token). White space is defined as one or more spaces, tabs, line-break characters (carriage return, line feed, form feed), and comments.

Keywords and identifiers must be separated by white space, or a token that is neither a keyword nor an identifier. For example, `intfortrue` is a single identifier, not three distinct keywords. If a sequence begins with an alphabetic character or an underscore, then it and the longest sequence of alphanumeric characters following it forms a token.

String literals are composed of ⟨char⟩s enclosed in double quotes. A character literal consists of a ⟨char⟩ enclosed in single quotes.

If a sequence begins with `0x`, then these first two characters and the longest following 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 following sequence of decimal digits forms a decimal integer literal. A long sequence of digits (e.g., `123456789123456789123`) is scanned as a single token.

A ⟨char⟩ is any printable ASCII character (ASCII values between decimal value 32 and 126 inclusive) other than quote (`"`), single quote (`'`), or backslash (`\`), plus the 2-character sequences `\"` to denote quote, `\'` to denote single quote, `\\` to denote backslash, `\t` to denote a literal tab, `\n` to denote newline, `\r` to denote carriage return, or `\f` to denote form feed.

3 Reference Grammar

Notation	Meaning
⟨foo⟩	means foo is a nonterminal.
foo	(in **bold** font) means that **foo** is a terminal i.e., a token or a part of a token.
$\left[x\right]$	means zero or one occurrence of x, i.e., x is optional; note that brackets in quotes `'['` `']'` are terminals.
x^*	means zero or more occurrences of x.
x^+**,**	a comma-separated list of one or more x's. note that there is no comma following the last of x.
$\left\{\ \right\}$	large braces are used for grouping; note that braces in quotes `'{'` `'}'` are terminals.
|	separates alternatives.

⟨program⟩ → ⟨import_decl⟩* ⟨field_decl⟩* ⟨method_decl⟩*

⟨import_decl⟩ → **import** ⟨id⟩ ;

⟨field_decl⟩ → ⟨type⟩ $\left\{\text{⟨id⟩} \mid \text{⟨array\_field\_decl⟩}\right\}^+$**,** ;

⟨array_field_decl⟩ → ⟨id⟩ `'['` ⟨int_literal⟩ `']'`

⟨method_decl⟩ → $\left\{\text{⟨type⟩} \mid \textbf{void}\right\}$ ⟨id⟩ ($\left[\left\{\text{⟨type⟩ ⟨id⟩}\right\}^+,\right]$) ⟨block⟩

⟨block⟩ → `'{'` ⟨field_decl⟩* ⟨statement⟩* `'}'`

⟨type⟩ → **int** | **long** | **bool**

⟨statement⟩ →
- ⟨location⟩ ⟨assign_expr⟩ ;
- ⟨method_call⟩ ;
- **if** (⟨expr⟩) ⟨block⟩ $\left[\textbf{else}\ \text{⟨block⟩}\right]$
- **for** (⟨id⟩ = ⟨expr⟩ ; ⟨expr⟩ ; ⟨for_update⟩) ⟨block⟩
- **while** (⟨expr⟩) ⟨block⟩
- **return** $\left[\text{⟨expr⟩}\right]$;
- **break** ;
- **continue** ;

⟨for_update⟩ → { ⟨location⟩ ⟨assign_expr⟩ }

⟨assign_expr⟩ → ⟨assign_op⟩ ⟨expr⟩ | ⟨increment⟩

⟨assign_op⟩ → `=` | `+=` | `-=` | `*=` | `/=` | `%=`

⟨increment⟩ → `++` | `--`

⟨method_call⟩ → ⟨method_name⟩ `(` [⟨expr⟩$^{+}$`,`] `)`
| ⟨method_name⟩ `(` [⟨extern_arg⟩$^{+}$`,`] `)`

⟨method_name⟩ → ⟨id⟩

⟨location⟩ → ⟨id⟩
| ⟨id⟩ `'['` ⟨expr⟩ `']'`

⟨expr⟩ → ⟨location⟩
| ⟨method_call⟩
| ⟨literal⟩
| `int` `(` ⟨expr⟩ `)`
| `long` `(` ⟨expr⟩ `)`
| `len` `(` ⟨id⟩ `)`
| ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
| `-` ⟨expr⟩
| `!` ⟨expr⟩
| `(` ⟨expr⟩ `)`

⟨extern_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⟩ | ⟨long_literal⟩ | ⟨char_literal⟩ | ⟨bool_literal⟩

⟨id⟩ → ⟨alpha⟩ ⟨alpha_num⟩*

⟨alpha_num⟩ → ⟨alpha⟩ | ⟨digit⟩

⟨alpha⟩ → `a` | `b` | `...` | `z` | `A` | `B` | `...` | `Z` | `_`

⟨digit⟩ → `0` | `1` | `2` | `...` | `9`

⟨hex_digit⟩ → ⟨digit⟩ | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`

⟨int_literal⟩ → { ⟨decimal_literal⟩ | ⟨hex_literal⟩ }

⟨long_literal⟩ → { ⟨decimal_literal⟩ | ⟨hex_literal⟩ } **L**

⟨decimal_literal⟩ → ⟨digit⟩ ⟨digit⟩*

⟨hex_literal⟩ → `0x` ⟨hex_digit⟩ ⟨hex_digit⟩*

⟨bool_literal⟩ → **true** | **false**

⟨char_literal⟩ → ' ⟨char⟩ '

⟨string_literal⟩ → " ⟨char⟩* "

4 Semantics

A Decaf program consists of a single file. A program consists of import declarations, field declarations and method declarations. Import declarations introduce methods from other libraries to the Decaf file. 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 type `void` and takes no parameters. Execution of a Decaf program starts at method `main`.

4.1 Types

There are three basic types in Decaf: `int`, `long` and `bool`. In addition, there are single-dimensional arrays of integers (`int[]`), longs (`long[]`) and booleans (`bool[]`).

An `int` is a 32-bit signed integer. An `long` is a 64-bit signed integer. A `bool` is either `true` or `false`, and should be passed as a 32-bit `1` or `0` respectively when invoking external methods.

Arrays may be declared in any scope. An array declaration must contain a size $[N]$. 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. Arrays are indexed by the usual bracket notation `a[i]`.

Note for Phase 3: You may notice that replacing every instance of `int` with its equivalent 64-bit `long` would yield a semantically identical program. Indeed, it is permissible to do this in the code generation stage and to assume that both `int` and `long` are 64 bits. This will incur a non-negligible performance penalty, and make certain optimizations more difficult. However, if a team is running very behind on their project, this approach will yield a working compiler.

4.2 Scope Rules

Decaf has simple and quite restrictive scope rules. All identifiers must be defined (textually) before use. For example:

- 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 at least two valid scopes at any point in a Decaf program: the global scope and the method scope. The global scope consists of external methods, fields, and methods introduced in

the top level of the program. The method scope consists of method parameters introduced in a method declaration. Additional local scopes exist within each ⟨block⟩ in the code.

Identifiers introduced within a scope are visible in that scope and in all nested scopes. Identifiers within a more deeply nested scope shadow identifiers in a less deeply nested scope. For example, a variable declared in a method shadows a method of the same name in the global scope.

No identifier may be defined more than once in the same scope. Thus field and method names must all be distinct in the global scope, method parameters must all be distinct in a method scope, and local variables must all be distinct in a given block scope. Similarly, no identifier may be defined in the method parameters and the top level block of a method.

4.3 Locations

Locations in Decaf are either global (i.e., defined at the top level of the program) or local (i.e., defined as method parameters or local variables). Each location has a type. Locations of types `int`, `long` and `bool` contain scalar integer, scalar long, and boolean values respectively. Locations of types `int[]`, `long[]`, and `bool[]` denote arrays of integers, longs, and booleans respectively.

The default value of `int`, `long`, and `bool` are undefined. The default value of arrays are undefined.

Since arrays are statically sized in Decaf, global arrays may be allocated in the static data space of a program and need not be allocated on the heap. Local arrays may be dynamically allocated on the stack or statically allocated on the heap when appropriate.

4.4 Assignment

Assignment is only permitted for scalar locations. Decaf uses value-copy semantics for assignments.

The assignment ⟨location⟩ `=` ⟨expr⟩ copies the value resulting from the evaluation of ⟨expr⟩ into ⟨location⟩, and is only valid when ⟨location⟩ and ⟨expr⟩ have the same type.

Decaf implements **explicit cast semantics**, and implicit casting is not allowed. The overflow semantics when casting from a larger to a smaller type are **undefined**. You may implement any reasonable semantics (either wrapping or saturating).

The compound assignment ⟨location⟩ `+=` ⟨expr⟩ increments the value stored in ⟨location⟩ by ⟨expr⟩, and is only valid for both ⟨location⟩ and ⟨expr⟩ of types `int` and `long`.

Other compound assignment operators ⟨location⟩ `-=` ⟨expr⟩, ⟨location⟩ `*=` ⟨expr⟩, ⟨location⟩ `/=` ⟨expr⟩, and ⟨location⟩ `%=` ⟨expr⟩ work similarly to ⟨location⟩ `+=` ⟨expr⟩, but they perform subtraction, multiplication, division, and modulo, respectively.

The assignments ⟨location⟩ `++` and ⟨location⟩ `--` increment and decrement the value stored in ⟨location⟩ by `1` respectively, and are only valid for ⟨location⟩ of types `int` and `long`.

In assignments containing ⟨expr⟩s with side effects, the left hand side is evaluated before the right hand side. Both are evaluated exactly once, and are evaluated completely before the assignment is performed.

Note that it is legal to assign to a method parameter variable within a method body. Such assignments affect only the method scope.

An out of bounds write to an array has **undefined** behavior.

4.5 Method Invocation and Return

Method invocation involves (1) passing parameter values from the caller to the callee, (2) executing the body of the callee, and (3) returning to the caller, possibly with a result.

Parameters to methods are passed by value (with the exception of arrays and strings passed to external functions, which is discussed in Section 4.8). The parameters of a method are considered to be like local variables of the method and are initialized, by assignment, to the values resulting from the evaluation of the argument expressions. The arguments are evaluated from left to right.

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

A method that has no declared result type can only be called as a statement; it cannot be used in an expression. Such a method returns control to the caller when `return` is called (no result expression is allowed) or when the textual end of the callee is reached.

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. It is illegal for control to reach the textual end of a method that returns a result.

A method that returns a result may also be called as a statement. In this case, the result is ignored.

4.6 Control Statements

4.6.1 `if`

The `if` statement has the usual semantics. First, the ⟨expr⟩ is evaluated. If the result is `true`, the first arm is executed. Otherwise, the `else` arm is executed, if it exists.

4.6.2 `while`

The while statement has the usual semantics. First, the ⟨expr⟩ is evaluated. If the result is `false`, control moves to the statement following the `while` statement. Otherwise, the loop body is executed. If control in a `while` statement reaches the end of the loop body, the `while` statement is executed again.

4.6.3 for

The `for` statement has the usual C-like semantics. The ⟨id⟩ is the loop index variable, and must have been have been declared in scope as an integer variable. Because it must be an identifier, this means that array index locations are not valid loop index variables. Before entering the loop body, it is assigned the value of the first ⟨expr⟩. This expression is evaluated once, just prior to reaching the loop for the first time.

The second ⟨expr⟩ is the ending condition of the loop, which must evaluate to type `bool`. It is evaluated before each iteration of the loop body. If the result is `false`, control moves to the statement following the `for` statement. Otherwise, the loop body is executed. Note that ⟨expr⟩ may be an expression with side effects (e.g., a method call).

After each iteration of the loop body, the ⟨for_update⟩ is executed. It may have side effects thus must be evaluated each time. The ⟨location⟩ updated does not need to be the same as the loop index variable. Control then returns to checking the ending condition.

4.6.4 `break` and `continue`

The `break` statement causes control to exit the innermost loop, moving control to the statement following the loop. The `continue` statement causes control to exit the current iteration of the innermost loop, moving control to the condition check at the beginning of the loop for `while` loops, and to the ⟨for_update⟩ for `for` loops.

4.6.5 `return`

The `return` statement causes control to exit the current method, returning to the caller.

4.7 Expressions

Expressions follow the normal rules for evaluation. All binary operators are left associative.

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 *External Library*.

Short circuiting semantics are **highly recommended** for standalone expressions (i.e. outside of if/while/for loops), but not explicitly tested. This would bring your implementation into line with C semantics.

Numerical literals evaluate to their corresponding value. If there is an `L` after the numerical literal, it evaluates to type `long`, while if it does not, it evaluates to type `int`.

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

The `len` operator evaluates the size of an array to an `int` value. Because array lengths are static and known at compile time, this is a constant expression which can be evaluated at compile time. Note that the `len` operator is not a method call.

The arithmetic operators (⟨arith_op⟩ and unary minus `-`) have their usual meaning. The division operator `/` computes the integer part of the quotient of its operands (e.g., `5/2` evaluates to `2`). The mod operator `%` computes the remainder of dividing its operands (e.g., `5%2` evaluates to `1`). Division and mod by zero have undefined behavior, and will not be tested. The result of an arithmetic operator has type `int` or `long`. The type is based on the larger of the two operands.

Relational operators (⟨rel_op⟩) are used to compare integer expressions, and also have their usual meaning. The result of a relational operator or equality operator has type `bool`.

The equality operators, `==` (equal) and `!=` (not equal) are defined for `int`, `long`, and `bool` types only. The result of an equality operator has type `bool`.

The boolean connectives `&&` and `||` are interpreted using short circuit evaluation as in Java. That is, the right hand side is not executed (and therefore has no side effects if it contains a method call) if the result of the expression is already known (i.e., if the left hand side is `false` for `&&` or `true` for `||`).

The logical not operator `!` has the usual meaning, and is defined for `bool` expressions only.

An out of bounds read from an array has **undefined** behavior.

Casting is done with the `int()` and `long()` operators. A `int` and `long` can be cast to each other, but a `bool` cannot be cast to a numerical type. Overflow semantics are undefined as previously mentioned.

Operator precedence, from highest to lowest:

Operators	*Comments*
`-`	unary minus
`!`	logical not
`int() long()`	type casts
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`< <= >= >`	relational
`== !=`	equality
`&&`	conditional and
`||`	conditional or

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

4.8 External Library

Decaf includes a method for calling external functions, similar to the C language. An external function is put in scope with the `import` keyword at the top of the file. The syntax (as specified in the grammar) is:

import ⟨id⟩ **;**

All external functions are treated as if they return `int`. Once imports have been declared, they may be called similarly to any function. The one exception is that arguments to imports may contain string literals. Normal Decaf methods may not contain string literals as arguments.

4.8.1 ABI for External Calls

Use 32 bits for Decaf `int`, use 64 bits for Decaf `long`, and 32 bits for Decaf `bool` (1 for `true`, and 0 for `false`). You may use any reasonable internal representation of the types within Decaf, but calls to external functions must follow this ABI for proper linking and execution. You may assume that **external calls** will not require more than 6 arguments (i.e. all arguments will fit in registers).

4.8.2 Import Arguments

When calling an import, the following rules apply to the arguments:

Expressions of type `bool`, `int`, `long` are passed as integers of their respective sizes. Expressions of type `bool[]`, `int[]`, and `long[]` are passed as pointers to the first element of the array. A

string literal is passed as a pointer to the first character of a null-terminated character array. The return value is passed back as an 32 bit integer.

The user of an `import` function 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 actually returns a value of appropriate type. Arguments are passed to the function in the system's standard calling convention. The compiler is not responsible for verifying that imports have the correct number or type of arguments.

4.8.3 External I/O Functions

In addition to accessing the standard C library using `import`, 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 `import` mechanism. This is the only way to access I/O functions in Decaf.

5 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. This include `import` identifiers, which exist in the global scope.
2. No identifier is used before it is declared.
3. The program contains a definition for a method called `main` that has type `void` and takes no parameters. Note that since execution starts at method `main`, any methods defined after `main` will never be executed.
4. The number and types of parameters in a method call (non-import) must be the same as the number and types of the declared parameters for the method.
5. If a method call is used as an expression, the method must return a result.
6. String literals and array variables may not be used as parameters to non-import methods. (Note: an expression like `a[0]` is not an array variable).
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 parameter.
10. The ⟨id⟩ in a method statement must be a declared method or import.

11. For all locations of the form ⟨id⟩`[`⟨expr⟩`]`:
    (a) ⟨id⟩ must be an array variable, and
    (b) the type of ⟨expr⟩ must be `int`.
12. The argument of the `len` operator must be an array variable.
13. The ⟨expr⟩ in an `if` or `while` statement must have type `bool`, as well as the second ⟨expr⟩ of a `for` statement.
14. The operands of the unary minus, ⟨arith_op⟩s and ⟨rel_op⟩s must have type `int` or `long`.
15. The operands of ⟨eq_op⟩s must have the same type, either `int`, `long`, or `bool`.
16. The operands of ⟨cond_op⟩s and the operand of logical not (`!`) must have type `bool`.
17. The ⟨location⟩ and the ⟨expr⟩ in an assignment, ⟨location⟩ `=` ⟨expr⟩, must have the same type.
18. The ⟨location⟩ and the ⟨expr⟩ in a compound assignment, ⟨location⟩ `+=` ⟨expr⟩, ⟨location⟩ `-=` ⟨expr⟩, ⟨location⟩ `*=` ⟨expr⟩, ⟨location⟩ `/=` ⟨expr⟩, and ⟨location⟩ `%=` ⟨expr⟩, must be of type `int` or `long`. The same is true of the ⟨location⟩ in `++` and `--` statements.
19. All `break` and `continue` statements must be contained within the body of a `for` or a `while` statement.
20. `int()` and `long()` casts only should take an `int` or `long` as an input
    (a) `int(expr)` takes the expression and casts it to the `int` type. It is permissible for ⟨expr⟩ to be of type `int`
    (b) The same rules apply to the `long(expr)` and `long` type.
21. All `int` literals must be in the range $-2147483648 \leq x \leq 2147483647$ (32 bits).
22. All `long` literals must be in the range $-9223372036854775808 \leq x \leq 9223372036854775807$ (64 bits).

6 Run Time 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.

For 2025, only one rule is required:

- Control must not fall off the end of a method that is declared to return a result.

When a runtime error occurs, an appropriate error message is output to the terminal and the program terminates. If control falls off the end of a method that is declared to return a result, the program must terminate with exit value -1. The error messages output should be helpful to the programmer trying to find the problem in the source program.