Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2014 | Handout — Decaf Language | Thursday, September 4

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

1 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:

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

Comments are started by `//` and are terminated by the end of the line.

White space may appear between any lexical tokens. White space is defined as one or more spaces, tabs, line-break characters, and comments.

Keywords 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. If a sequence begins with an alphabetic character or an underscore, then it and the longest sequence of 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.

Numbers in Decaf are 64 bit signed. That is, decimal values between âĹŠ9223372036854775808 and 9223372036854775807. 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 not performed during parsing. A long sequence of digits (e.g. 123456789123456789123) is still scanned as a single token.

A ⟨char⟩ is any printable ASCII character (ASCII values between decimal value 32 and 126) 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 literal tab, or “`\n`” to denote newline.

2 Reference Grammar

Meta-notation:

⟨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.
$\Big\{\ \Big\}$	large braces are used for grouping; note that braces in quotes '{' '}' are terminals.
|	separates alternatives.

⟨program⟩	→	⟨callout_decl⟩* ⟨field_decl⟩* ⟨method_decl⟩*
⟨callout_decl⟩	→	**callout** ⟨id⟩ ;
⟨field_decl⟩	→	⟨type⟩ $\Big\{$⟨id⟩ | ⟨id⟩ '[' ⟨int_literal⟩ ']' $\Big\}^{+}$, ;
⟨method_decl⟩	→	$\Big\{$⟨type⟩ | **void**$\Big\}$ ⟨id⟩ ($\Big[\Big\{$⟨type⟩ ⟨id⟩$\Big\}^{+}$,$\Big]$) ⟨block⟩
⟨block⟩	→	'{' ⟨field_decl⟩* ⟨statement⟩* '}'
⟨type⟩	→	**int** | **boolean**
⟨statement⟩	→	⟨location⟩ ⟨assign_op⟩ ⟨expr⟩ ;
	|	⟨method_call⟩ ;
	|	**if** (⟨expr⟩) ⟨block⟩ $\Big[$**else** ⟨block⟩$\Big]$
	|	**for** (⟨id⟩ = ⟨expr⟩ , ⟨expr⟩) ⟨block⟩
	|	**while** (⟨expr⟩) $\Big[$: ⟨int_literal⟩$\Big]$ ⟨block⟩
	|	**return** $\Big[$⟨expr⟩$\Big]$;
	|	**break** ;
	|	**continue** ;
⟨assign_op⟩	→	=
	|	+=
	|	-=
⟨method_call⟩	→	⟨method_name⟩ ($\Big[$⟨expr⟩$^{+}$,$\Big]$)
	|	⟨method_name⟩ ($\Big[$⟨callout_arg⟩$^{+}$,$\Big]$)
⟨method_name⟩	→	⟨id⟩
⟨location⟩	→	⟨id⟩
	|	⟨id⟩ '[' ⟨expr⟩ ']'

⟨expr⟩ → ⟨location⟩
 | ⟨method_call⟩
 | ⟨literal⟩
 | `@` ⟨id⟩
 | ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
 | `-` ⟨expr⟩
 | `!` ⟨expr⟩
 | `(` ⟨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` | `...` | `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⟩

⟨decimal_literal⟩ → ⟨digit⟩ ⟨digit⟩*

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

⟨bool_literal⟩ → **true** | **false**

⟨char_literal⟩ → `'` ⟨char⟩ `'`

⟨string_literal⟩ → `"` ⟨char⟩* `"`

3 Semantics

A Decaf program consists of a single file. A program consists of callout declarations, 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**.

3.1 Types

There are two basic types in Decaf — **int** and **boolean**. In addition, there are single-dimensional arrays of integers (`int [` N `]`) and arrays of booleans (`boolean [` N `]`).

Arrays may be declared in any 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. Arrays are indexed by the usual bracket notation $a[i]$. We use the operator **@a** to denote the length of the array **a**. Note that the arrays have a compile-time fixed size and this expression always evaluate to a constant and is used primarily for convenience.

3.2 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 names of callouts, fields, and methods introduced in the top level of the program. The method scope consists of names of variables and formal parameters introduced in a method declaration. Additional local scopes exist within each ⟨block⟩ in the code; these can come after **if**, **while** and **for** statements. An identifier introduced in a method scope can shadow an identifier from the global scope. Similarly, identifiers introduced in local scopes shadow identifiers in less deeply nested scopes, the method scope, and the global scope.

Variable names defined in the method scope or a local scope may shadow method names or callout names in the global scope. In this case, the identifier may only be used as a variable until the variable leaves 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, and local variable names and formal parameters names must be distinct in each local scope.

3.3 Locations

Decaf has two kinds of locations: local/global scalar variables and local/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. 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.

Each location is initialized to a default value when it is declared. Integers have a default value of zero, and booleans have a default value of **false**. Local variables must be initialized when the declaring scope is entered. Each element of a global array is initialized when the program starts. Each element of a local array is initialized when execution of the program enters the array's scope. In general, each time execution enters the scope of an array, its values must be reset to their defaults.

3.4 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 ⟨location⟩ `+=` ⟨expr⟩ assignment increments the value stored in ⟨location⟩ by ⟨expr⟩, and is only valid for both ⟨location⟩ and ⟨expr⟩ of type **int**. The ⟨location⟩ `-=` ⟨expr⟩ assignment decrements the value stored in ⟨location⟩ by ⟨expr⟩, and is only valid for both ⟨location⟩ and ⟨expr⟩ of type **int**.

The ⟨location⟩ and the ⟨expr⟩ in an assignment must have the same type. For array types, ⟨location⟩ and ⟨expr⟩ must refer to a single array element which is also a scalar value.

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

3.5 Method Invocation and Return

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

Argument passing is defined in terms of assignment: the formal arguments 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, *i.e.*, 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.

3.6 Control Statements

3.6.1 if

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

3.6.2 while

The **while** statement has the usual semantics. First, the ⟨expr⟩ is evaluated. If the result is **false**, control exits the loop. Otherwise, the loop body is executed. If control reaches the end of the loop body, the **while** statement is executed again.

Optionally, the **while** statement can have an upper bound on the number of loop iterations. Such loop bounds can be used to specify the maximum resource consumption of the loop and qprevent infinite loops. This bound is a fixed integer number. The semantics of such bounded loop is the same as semantics of the standard loop with a counter variable initialized to zero and incremented in every iteration and the additional conjunct in the loop's that checks that the variable is less than a bound.

3.6.3 for

The **for** statement is similar to a **d**o loop in Fortran. The ⟨id⟩ is the loop index variable and must have been declared as an integer variable in the current scope or an outer scope. Because it must be an identifier, this means that array locations are not valid loop index variables. Before entering the loop body, it is assigned the value of the first ⟨expr⟩. 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. Each expression must evaluate to an integer value. 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 in incremented by 1, and the new value is compared to the ending value to decide if another iteration should execute.

3.7 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**.)

We discussed the array length operator in Section 3.1.

The arithmetic operators (⟨arith_op⟩ and unary minus) have their usual 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, and can only be used to compare 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. No side-effects of the second operand are 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 `||`).

The ternary if condition `? :` has the semantics similar to that in Java: it first evaluates the condition expression (before the `?` symbol). If the condition evaluates to true, it then evaluates and returns the result of the first alternative expression (before the `:` symbol); if the condition evaluates to false, it then evaluates and returns the result of the second alternative expression (after the :). Therefore, it always executes only one of its alternative expressions.

Operator precedence, from highest to lowest:

<table>
<tr><th>Operators</th><th>Comments</th></tr>
<tr><td><code>@</code></td><td>array length</td></tr>
<tr><td><code>-</code></td><td>unary minus</td></tr>
<tr><td><code>!</code></td><td>logical not</td></tr>
<tr><td><code>* / %</code></td><td>multiplication, division, remainder</td></tr>
<tr><td><code>+ -</code></td><td>addition, subtraction</td></tr>
<tr><td><code>< <= >= ></code></td><td>relational</td></tr>
<tr><td><code>== !=</code></td><td>equality</td></tr>
<tr><td><code>&&</code></td><td>conditional and</td></tr>
<tr><td><code>||</code></td><td>conditional or</td></tr>
<tr><td><code>? :</code></td><td>ternary if</td></tr>
</table>

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

3.8 Library Callouts

Decaf includes a method for calling library callouts similar to the c language. Callouts must be predeclared at the top of the file. The syntax (as specified in the grammar) is:

callout ⟨id⟩ ;

All callout functions are treated as if they return **int**. Once callouts have been declared, they may be called similar to any function. The exceptions to this are that arguments to callouts may contain string literals. **This is the only use of the string literal in the decaf language**. Normal decaf methods may not contain string literals as arguments.

3.8.1 Callout Arguments

Expressions of boolean or integer type are passed as integers; string literals or expressions with array type are passed as memory addresses. The return value of the function is passed back as an integer. The user of a **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 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 callouts have the correct number or type of arguments.**

3.8.2 Writing Library Callouts

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.

4 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 includes **callout** 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 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 (non-callout) 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. String literals and array variables may not be used as arguments to non-callout methods. **Note: `a[5]` is not an array variable, it is an array location**
8. 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.
9. The expression in a **return** statement must have the same type as the declared result type of the enclosing method definition.
10. An ⟨id⟩ used as a ⟨location⟩ must name a declared local/global variable or formal parameter.
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 @ operator must be an array variable
13. The ⟨expr⟩ in an **if** or a **while** statement must have type **boolean**.
14. The first ⟨expr⟩ in a ternary conditional expression (**?:**) must have type **boolean**
15. The other two expressions in a ternary conditional expression must have the same type (integer or boolean)
16. The operands of ⟨arith_op⟩s and ⟨rel_op⟩s must have type **int**.
17. The operands of ⟨eq_op⟩s must have the same type, either **int** or **boolean**.

18. The operands of ⟨cond_op⟩s and the operand of logical not (!) must have type **boolean**.

19. The ⟨location⟩ and the ⟨expr⟩ in an assignment, ⟨location⟩ = ⟨expr⟩, must have the same type.

20. The ⟨location⟩ and the ⟨expr⟩ in an incrementing/decrementing assignment, ⟨location⟩ += ⟨expr⟩ and ⟨location⟩ -= ⟨expr⟩, must be of type **int**.

21. The initial ⟨expr⟩ and the ending ⟨expr⟩ of **for** must have type **int**.

22. The optional upper bound on the number of iterations of the **while** loop must be a positive integer.

23. All **break** and **continue** statements must be contained within the body of a **for** or a **while**.

5 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.

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. If the subscript of an array is found to be out of bounds, the program must terminate with exit value -1. If control falls off the end of a method that is declared to return a result, the program must terminate with exit value -2. The error messages output should be helpful to the programmer trying to find the problem in the source program.