PROGRAMMING LANGUAGES (6G6Z1110)

DECAF LANGUAGE REFERENCE

In lab classes this term, you will write a compiler for a language called Decaf. This work will run throughout the entire term, and it will be assessment in the exam at the end of the term. The work is staged; you must complete each successive part of the compiler before proceeding to the next.

Decaf is a simple imperative language similar to C or Pascal. It also bears some resemblance to Java. This document is adapted and revised from one used in MIT OpenCourseWare http://ocw.mit.edu for the subject 6.035 Computer Language Engineering Spring 2010. Any errors introduced are the responsibility of Emma Norling.

Read this document carefully. It contains a *lot* of information, all of which is important.

LEXICAL CONSIDERATIONS

All Decaf keywords are lowercase. Keywords 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.

The reserved words are:

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

Note that **Program** (see below) is not a keyword, but an identifier with a special meaning in certain circumstances.

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, page and line-breaking 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 32 bit signed. That is, decimal values between -2147483648 and 2147483647. 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. A long sequence of digits (e.g. 123456789123456789) is still scanned as a single token.

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 (`\`), plus 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.

REFERENCE GRAMMAR

To interpret the grammar below, you need to take careful note of the following 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.
[*x*]	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.
{ }	large braces are used for grouping; note that braces in quotes '{' '}' are terminals.
|	separates alternatives.

In particular, pay careful attention to the *x+,* notation: it is not the same as the *x+* notation that is commonly used in regular expressions.

⟨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 | ... | 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⟩* "

SEMANTICS

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**. In addition, there are arrays of integers (int [*N*]) and arrays of booleans (boolean [*N*]).

Arrays may be declared only in the global (class declaration) 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 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 RULES

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

1. A variable must be declared before it is used.
2. 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 fields and methods introduced in the (single) **Program** class declaration. 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** or **for** statements, or inserted anywhere a <statement> is legal. 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 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.

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. Since arrays are statically sized in Decaf, arrays may be allocated in the static data space of a program and need not be allocated on the heap.

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. Array elements are initialized when the program starts.

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 that 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.

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 a 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.

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. *It is nothing like a for loop in Java or C.* The <id> is the loop index variable and it shadows a variable of the same name declared in an outer scope if one exists. The loop index variable declares an integer variable whose scope is limited to the body of the loop. 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. 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.

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, and 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	Comments
`-`	Unary minus
`!`	Logical not
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`< <= >= >`	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 primitive method for calling functions is:

> **int callout** (<string_literal>, [<callout_arg>$^{+}$,]) — 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 actually 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 CHECKING

The semantic rules given on the final page of this document 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.

SEMANTIC RULES

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, and
    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>s and <rel_op>s must have type **int**.
13. The operands of <eq_op>s must have the same type, either **int** or **boolean**.
14. The operands of <cond_op>s 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**.

RUN TIME CHECKING (OPTIONAL – NOT ASSESSED)

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 should 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.