The Decaf 19 Language

Acknowledgment The Decaf language and its associated projects used in this course are based on the work of Dr. Lawrence Rauchwerger from TAMU. The original project can be found here.

Warning Decaf is an old teaching package, so you will be able to find solutions online. However, Decaf is probably the best compiler project that teaches the construction of a compiler. Therefore, to truly learn compiler, you should always write your own code. We will definitely compare your source code with online solutions and the code from other students. Moreover, we make substantial changes to the original Decaf language to generate Decaf 19 language. Please read this document carefully because it gives the specification for the Decaf 19 language.

Collaboration Policy The Decaf projects are fairly difficult. Therefore, students are allowed to collaborate. However, there are things that you should not do:

- Jointly developing a project and submitting code individually.
- Possessing a copy of other's code as reference.
- Using excerpts from another project as your code.

Things that are OK:

- Collaborating on whiteboard.
- Verbal discussions.
- Helping each other debugging.

Introduction In this course, we will write a compiler for a simple programming language called Decaf 19. Decaf 19 is a strongly-typed language. By design, it has many similarities with C/C++/Java, so you should find it fairly easy to pick up. Keep in mind it is not an exact match to any of those languages. The feature set has been trimmed down and simplified to keep the programming projects manageable. This document is designed to give the specification for the language syntax and semantics, which you will need to refer to when implementing the course projects.

1 Lexical considerations

The following are keywords. They are all reserved, which means they cannot be used as identifiers or redefined.

```
void    int    double       bool
string  null   for
while   if     else         return
break   Print  ReadInteger  ReadLine
```

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Decaf 19 is case-sensitive, e.g., `if` is a keyword, but `IF` is an identifier; `binky` and `Binky` are two distinct identifiers. Identifiers can be at most 31 characters long.

Whitespace (i.e. spaces, tabs, and newlines) serves to separate tokens, but is otherwise ignored. Keywords and identifiers must be separated by whitespace or a token that is neither a keyword nor an identifier. `ifintthis` is a single identifier, not three keywords. `if(23this` scans as four tokens.

A boolean constant is either `true` or `false`. Like keywords, these words are reserved.

An integer constant is a sequence of decimal digits (`0-9`). A double constant is a sequence of digits, a period, followed by any sequence of digits, maybe none. Thus, `.12` is not a valid double but both `0.12` and `12.` are valid. A double can also have an optional exponent, e.g., `12.2E+2` For a double in this sort of scientific notation, the decimal point is required, the sign of the exponent is optional (if not specified, `+` is assumed), and the E can be lower or upper case. As above, `.12E+2` is invalid, but `12.E+2` is valid.

A string constant is a sequence of characters enclosed in double quotes. Strings can contain any character except a newline or double quote. A string must start and end on a single line, it cannot be split over multiple lines:

```
``this string is missing its close quote
this is not a part of the string above
```

Operators and punctuation characters used by the language includes:

```
+  -  *  /  %   <  <=  >  >=  =  ==  !=  &&  ||  !  ;  ,  .  (  )  {  }
```

A single-line comment is started by `//` and extends to the end of the line. Multi-line comments start with `/*` and end with the first subsequent `*/`. Any symbol is allowed in a comment except the sequence `*/` which ends the current comment. Multi-line comments do not nest.

2 Reference grammar

The reference grammar is given in a variant of extended BNF. The meta-notation used:

x	(in bold) means that x is a terminal i.e., a token. Terminal names are also all lowercase except for those few keywords that use capitals.
x	means x is a nonterminal. All nonterminal names are capitalized.
<x>	means zero or one occurrence of x, i.e., x is optional
x^*	means zero or more occurrences of x
x^+	means one or more occurrences of x
x^+,	a comma-separated list of one or more x's (commas appear only between x's)
|	separates production alternatives
ϵ	indicates epsilon, the absence of tokens

For readability, we represent operators by the lexeme that denotes them, such as + and ! = as opposed to the token (T_NotEqual, etc.) returned by the scanner.

Program	::= Decl^+
Decl	::= VariableDecl | FunctionDecl
VariableDecl	::= Variable **;**
Variable	::= Type **ident**
Type	::= **int** | **double** | **bool** | **string** | **ident**
FunctionDecl	::= Type **ident (** Formals **)** StmtBlock | **void ident (** Formals **)** StmtBlock
Formals	::= Variable^+**,** | ϵ
StmtBlock	::= { VariableDecl^* Stmt^* }
Stmt	::= <Expr>**;** | IfStmt | WhileStmt | ForStmt | BreakStmt | ReturnStmt | PrintStmt | StmtBlock
IfStmt	::= **if (** Expr **)** Stmt <**else** Stmt>
WhileStmt	::= while (Expr) Stmt
ForStmt	::= **for (** <Expr>**;** Expr **;** <Expr>**)** Stmt
ReturnStmt	::= return $< Expr >$ **;**
BreakStmt	::= **break ;**
PrintStmt	::= **Print (** Expr^+**,) ;**
Expr	::= LValue **=** Expr | Constant | LValue | **this** | Call | **(** Expr **)** | Expr **+** Expr | Expr **-** Expr | Expr ***** Expr | Expr / Expr | Expr % Expr | **-** Expr | Expr < Expr | Expr <= Expr | Expr > Expr | Expr >= Expr | Expr == Expr | Expr ! = Expr | Expr **&&** Expr | Expr || Expr | ! Expr | **ReadInteger ( )** | **ReadLine ( )**
LValue	::= **ident**
Call	::= **ident (** Actuals **)**
Actuals	::= Expr^+, | ϵ
Constant	::= **intConstant** | **doubleConstant** | **boolConstant** | **stringConstant** | **null**

3 Program structure

A Decaf 19 program is a sequence of declarations, where each declaration establishes a variable or a function. The term *declaration* usually refers to a statement that establishes the identity of a name whereas *definition* means the full description with actual body. For our purposes, the declaration and the definition are one and the same. A program must have a global function named `main` that takes no arguments and returns void. This function serves as the entry point for program execution.

4 Scoping

Decaf 19 supports several levels of scoping. A declaration at the top-level is placed in the global scope. Each function has a local scope for its parameter list and another local scope for its body. A set of curly braces within a local scope establishes a nested local scope. Inner scopes shadow outer scopes; for example, a variable defined in a function's scope masks another variable with the same name in the global scope.

- all identifiers must be declared
- upon entering a scope, all declarations in that scope are immediately visible (see note below)
- identifiers within a scope must be unique (i.e. cannot have two functions of same name, a global variable and function of the same name, etc.)
- identifiers re-declared with a nested scope shadow the version in the outer scope (i.e. it is legal to have a local variable with the same name as a global variable).
- declarations in the global scope are accessible anywhere in the program (unless they are shadowed by another use of the identifier)
- declarations in closed scopes are inaccessible

Note about visibility and the relationship to lexical structure: As in Java, our compiler operates in more than one pass, the first pass gathers information and sets up the parse tree, only after that is complete do we return and do semantic analysis. A benefit of a two-pass compiler is that declarations are available at the same scope level even before the lexical point at which they are declared. For example, a function can refer to a function declared later in the program. When a scope is entered, all declarations made in that scope are immediately visible, no matter how much further down the file the declaration eventually appears. This rule applies uniformly to all declarations (variables, functions) in all scopes.

5 Types

The built-in base types are integer, double, boolean, string, and void.

6 Variables

Variables can be declared of non-void base type. Variables declared outside any function have global scope. Variables declared in the formal parameter list or function body have local scope. A variable is visible from scope entry to exit.

7 Strings

String support is somewhat sparse in Decaf. Programs can include string constants, read strings from the user with the built-in `ReadLine` function, compare strings, and print strings, but that's about it. There is no support for programmatically creating and manipulating strings, converting between strings and other types, and so on. (Consider it an opportunity for extension!) Strings are implemented as references (pointers).

- `ReadLine()` reads a sequence of chars entered by the user, up to but not including the newline
- string assignment is shallow (i.e. assigning one string to another copies just the reference)
- strings may be passed as parameters and returned from functions
- string comparison (== and !=) compares the sequence of characters in the two strings in a case-sensitive manner (behind the scenes we will use an internal library routine to do the work)

8 Functions

A function declaration includes the name of the function and its associated $typesignature$, which includes the return type as well as number and types of formal parameters.

- functions may not be nested within other functions
- the function may have zero or more formal parameters
- formal parameters can be of non-void base type
- identifiers used in the formal parameter list must be distinct
- the formal parameters are declared in a separate scope from the function's local variables (thus, a local variable can shadow a parameter)
- the function return type can be any base type. void type is used to indicate the function returns no value
- function overloading is not allowed i.e., the use of the same name for functions with different type signatures

- if a function has a non-void return type, any return statement must return a value compatible with that type
- if a function has a void return type, it may only use the empty return statement
- recursive functions are allowed

9 Function invocation

Function invocation involves passing argument values from the caller to the callee, executing the body of the callee, and returning to the caller, possibly with a result. When a function is invoked, the actual arguments are evaluated and bound to the formal parameters. All Decaf 19 parameters and return values are passed by value.

- the number of actual arguments in a function call must match the number of formal parameters
- the type of each actual argument in a function call must be compatible with the formal parameter
- the actual arguments to a function call are evaluated from left to right
- a function call returns control to the caller on a return statement or when the textual end of the callee is reached
- a function call evaluates to the type of the function's declared return type

10 Assignment

For the base types, Decaf 19 uses value-copy semantics; the assignment `LValue = Expr` copies the value resulting from the evaluation of `Expr` into the location indicated by `LValue`. For strings, Decaf 19 uses reference-copy semantics; the assignment `LValue = Expr` causes `LValue` to contain a reference to the string resulting from the evaluation of `Expr` (i.e., the assignment copies a pointer to a string rather than the string). Said another way, assignment for strings makes a shallow, not deep, copy.

- an LValue must be an assignable variable location
- the right side type of an assignment statement must be compatible with the left side type
- `null` can only be assigned to a variable of named type
- it is legal to assign to the formal parameters within a function, such assignments affect only the function scope

11 Control Structures

Decaf 19 control structures are based on the C/Java versions and generally behave somewhat similarly.

- An `else` clause always joins with the closest unclosed `if` statement
- the expression in the test portions of the `if`, `while` and `for` statements must have bool type
- a `break` statement can only appear within a `while` or `for` loop
- the value in a `return` statement must be compatible with return type of the enclosing function

12 Expressions

For simplicity, Decaf 19 does not allow co-mingling and conversion of types within expressions (i.e. adding an integer to a double, using an integer as a boolean, etc.).

- constants evaluate to themselves (true, false, null, integers, doubles, string literals)
- the two operands to binary arithmetic operators (`+`, `-`, `*`, `/`, and `%`) must either be both int or both double. The result is of the same type as the operands.
- the operand to a unary minus must be int or double. The result is the same type as the operand.
- the two operands to binary relational operators (`<`, `>`, `<=`, `>=`) must either both be int or both double. The result type is bool.
- the two operands to binary equality operators (`!=`, `==`) must be of equivalent type (two ints, two doubles, etc.) The result type is bool
- the operands to all binary and unary logical operators (`&&`, `||` , and `!`) must be of bool type. The result type is bool.
- logical `&&` and `||` do not short-circuit; both expressions are evaluated before determining the result
- the operands for all expressions are evaluated left to right

Operator precedence from highest to lowest:

`! -`	(unary -, logical not)
`* / %`	(multiply, divide, mod)
`+ -`	(addition, subtraction)
`< <= > >=`	(relational)
`== !=`	(equality)
`&&`	(logical and)
`||`	(logical or)
`=`	(assignment)

All binary arithmetic operators and both binary logical operators are left-associative. Assignment and the relational operators do not associate (i.e. you cannot chain a sequence of these operators that are at the same precedence level: `a < b >= c` should not parse, however `a < b == c` is okay). Parentheses may be used to override the precedence and/or associativity.

13 Standard library functions

Decaf 19 has a very small set of routines in its standard library that allow for simple I/O and memory allocation. The library functions are `Print`,`ReadInteger`, and `ReadLine`.

- the arguments passed to a `Print` statement can only be string, int, or bool
- `ReadLine()` reads a sequence of chars entered by the user, up to but not including the newline
- `ReadInteger()` reads a line of text entered by the user, and converts to an integer using `atoi` (returns 0 if the user didn't enter a valid number)

14 Linking

Given Decaf 19 does not allow separate modules or declaration separate from definition, there is not much work beyond semantic analysis to ensure we have all necessary code. The one task our "linker" needs to perform is to verify that that there was a declaration for the global function `main`. This will be done as part of code generation.