Decaf Language Reference

Mike Lam, James Madison University

Fall 2023

1 Introduction

Decaf is an imperative language similar to Java or C, but is greatly simplified compared to those languages. It will allow us to implement a compiler for the entire language in a single semester while still exploring all of the basic facets of a modern compiler.

This project is originally based on a project from course 6.035 at the Massachusetts Institute of Technology. However, significant changes have been made to the language. This document is the only authoritative reference to the version of Decaf used in CS 432 and CS 630 at James Madison University.

Here is an example program in Decaf:

```
// add.decaf - simple addition example

def int add(int x, int y)
{
    return x + y;           // add the two parameters
}

def int main()
{
    int a;
    a = 3;
    return add(a, 2);
}
```

Here is the output from running the reference compiler on the above program:

```
$ ./decaf add.decaf
RETURN VALUE = 5
```

Here are some important points of difference between Decaf and more general-purpose procedural languages like C and Java:

- Decaf is not object-oriented.
- Function declarations must begin with the `def` keyword.
- Variable declarations may not include initializations.
- Variable declarations must all occur together at the top of their scope.
- Many syntactic sugar operators (such as `+= or ++`) are not included.
- Boolean expression evaluation is not guaranteed to be short-circuited.

2 Lexical Considerations

There are six token classes in Decaf:

Name	Description
`ID`	identifiers
`KEY`	keywords
`DEC`	decimal literals
`HEX`	hexadecimal literals
`STR`	string literals
`SYM`	symbols

Identifiers and keywords must begin with an alphabetic character and may contain alphanumeric characters and the underscore ('_'). A keyword can be thought of a special identifier that is "reserved" and cannot be used for actual variables or functions. Both keywords and identifiers are case-sensitive, and all keywords are lowercase. 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 following are all of the supported keywords in Decaf:

```
def if else while return break continue int bool void true false
```

In addition, all of the following words are reserved; they are not currently keywords but may be used in future versions of the language and thus should not be used as identifier names:

```
for callout class interface extends implements new this string float double null
```

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

Symbols can be split into three types: 1) grouping operators: parentheses, brackets, curly braces, assignment (equals), commas, and semicolons, 2) binary operators (`BINOP`) and 3) unary operators (`UNOP`). There are a variety of binary and unary operators in Decaf, including both arithmetic operators (e.g., plus and minus) and boolean operators (e.g., less-than and boolean OR). The following are all supported symbols in Decaf:

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

Integer literals are unsigned and may be written in either decimal or hexadecimal form. The latter will always begin with the sequence `0x`, and may contain either lower- or upper-case letter digits. Neither decimal nor hexadecimal literals may be zero-padded at the beginning. Note that in Decaf integer literals are unsigned but integers themselves are signed. Thus, a negative "literal" in Decaf code is really an unsigned literal with the negation prefix operator ("-") applied.

String literals must be enclosed in double quote marks and may contain four kinds of escaped characters: newlines ('\n'), tabs ('\t'), quotes ('\"'), and backslashes ('\\'). Unescaped newlines and carriage returns are not allowed in string literals. ASCII is the only supported character set.

Comments are started by "`//`" and are terminated by the end of the line. "Whitespace" may appear between any lexical tokens, and consists of one or more spaces, tabs, newlines, or carriage returns. Comments and whitespace have no effect on a Decaf program's execution, and both should be discarded during the lexical analysis phase of compilation.

Decaf programs should be stored in files with the "`.decaf`" extension, and a single Decaf program may not span multiple files.

3 Syntax

The following is the Decaf grammar (in EBNF form):

$$
\begin{array}{rcl}
\textit{Program} & \rightarrow & \big(\textit{VarDecl} \mid \textit{FuncDecl}\big)* \\
\\
\textit{VarDecl} & \rightarrow & \textit{Type}\ \texttt{ID}\ \big(\text{`}\texttt{[}\text{'}\ \texttt{DEC}\ \text{`}\texttt{]}\text{'}\big)?\ \text{`}\texttt{;}\text{'} \\
\textit{Type} & \rightarrow & \texttt{int} \mid \texttt{bool} \mid \texttt{void} \\
\\
\textit{FuncDecl} & \rightarrow & \texttt{def}\ \textit{Type}\ \texttt{ID}\ \text{`}\texttt{(}\text{'}\ \textit{Params}?\ \text{`}\texttt{)}\text{'}\ \textit{Block} \\
\textit{Params} & \rightarrow & \textit{Type}\ \texttt{ID}\ \big(\ \text{`}\texttt{,}\text{'}\ \textit{Type}\ \texttt{ID}\ \big)* \\
\\
\textit{Block} & \rightarrow & \text{`}\texttt{\{}\text{'}\ \textit{VarDecl}*\ \textit{Stmt}*\ \text{`}\texttt{\}}\text{'} \\
\\
\textit{Stmt} & \rightarrow & \textit{Loc}\ \text{`}\texttt{=}\text{'}\ \textit{Expr}\ \text{`}\texttt{;}\text{'} \\
 & \mid & \textit{FuncCall}\ \text{`}\texttt{;}\text{'} \\
 & \mid & \texttt{if}\ \text{`}\texttt{(}\text{'}\ \textit{Expr}\ \text{`}\texttt{)}\text{'}\ \textit{Block}\ \big(\texttt{else}\ \textit{Block}\big)? \\
 & \mid & \texttt{while}\ \text{`}\texttt{(}\text{'}\ \textit{Expr}\ \text{`}\texttt{)}\text{'}\ \textit{Block} \\
 & \mid & \texttt{return}\ \textit{Expr}?\ \text{`}\texttt{;}\text{'} \\
 & \mid & \texttt{break}\ \text{`}\texttt{;}\text{'} \\
 & \mid & \texttt{continue}\ \text{`}\texttt{;}\text{'} \\
\\
\textit{Expr} & \rightarrow & \textit{Expr}\ \texttt{BINOP}\ \textit{Expr} \\
 & \mid & \texttt{UNOP}\ \textit{BaseExpr} \\
 & \mid & \textit{BaseExpr} \\
\\
\textit{BaseExpr} & \rightarrow & \text{`}\texttt{(}\text{'}\ \textit{Expr}\ \text{`}\texttt{)}\text{'} \\
 & \mid & \textit{Loc} \\
 & \mid & \textit{FuncCall} \\
 & \mid & \textit{Lit} \\
\\
\textit{Loc} & \rightarrow & \texttt{ID}\ \big(\text{`}\texttt{[}\text{'}\ \textit{Expr}\ \text{`}\texttt{]}\text{'}\big)? \\
\\
\textit{FuncCall} & \rightarrow & \texttt{ID}\ \text{`}\texttt{(}\text{'}\ \textit{Args}?\ \text{`}\texttt{)}\text{'} \\
\textit{Args} & \rightarrow & \textit{Expr}\ \big(\text{`}\texttt{,}\text{'}\ \textit{Expr}\big)* \\
\\
\textit{Lit} & \rightarrow & \texttt{DEC} \mid \texttt{HEX} \mid \texttt{STR} \mid \texttt{true} \mid \texttt{false}
\end{array}
$$

The following is a key to some of the meta-notation used above:

Example	Description
Foo	non-terminal
`foo`	keyword
`FOO`	token class (see Section 2)
‘`a`’	symbol consisting of character ‘`a`’
x?	zero or one occurences of x (i.e., optional)
x*	zero or more occurences of x
( )	used for grouping
|	designates alternatives

Although this is not reflected in the reference grammar on the previous page, all binary operations in Decaf are left-associative. In addition, there are seven levels of operator precedence, shown in the table below from highest to lowest:

Operators	*Comments*
`- !`	unary negation (integer and boolean)
`* / %`	integer multiplication, division, and remainder
`+ -`	integer addition and subtraction
`< <= >= >`	boolean ordinal relation
`== !=`	boolean equality
`&&`	boolean conjunction (AND)
`||`	boolean disjunction (OR)

As written above, this grammar is neither LL(1) nor LR(1); however, it can be transformed (using standard CFG transformations) into a grammar that is both. The process of performing these transformations is a useful exercise for the reader and a vital component of building a parser for the language.

Note that variables may be declared `void`; such declarations are allowed by the grammar but will be flagged during the static analysis phase.

Because the provided grammar is not yet suitable for directly building a parser, there are many possible correct LL(1) and LR(1) grammars, and thus many correct parse trees for any given Decaf program. Thus, it is important to have a clearly-defined *abstract syntax tree* format that is independent of any specific parsing implementation. Here is a list of standardized Decaf AST objects:

```
Program                 (contains lists of VarDecls and FuncDecls)
VarDecl
FuncDecl                (contains a Block)
Block                   (contains a list of VarDecls and a list of statements)

Statements:
    Assignment              (contains a Location and an expression)
    Conditional             (contains an expression and either one or two Blocks)
    WhileLoop               (contains an expression and a Block)
    Return                  (contains an optional expression)
    Break
    Continue

Expressions:
    BinaryOp                (contains two child expressions)
    UnaryOp                 (contains one child expression)
    Location
    FuncCall                (contains a list of expressions)
    Literal
```

All of these are stored in an `ASTNode` struct with a discriminated ("tagged") anonymous union for member data. See `ast.h` for all definitions.

4 Semantics

A Decaf program consists of a series of variable and function declarations.

4.1 Variables

All Decaf variables are statically typed, and there are only two valid basic types: signed integers that are 64 bits wide (`int`) and booleans (`bool`). Variables may be declared outside a function; such variables are considered "global" variables and are accessible from all functions. Variables declared inside a function or block are only visible inside the corresponding lexical scope (see Section 4.3 for details).

Decaf supports simple, fixed-length, one-dimensional, static arrays. Arrays must be declared global and their size is specified at their declaration time. Arrays are indexed from 0 to $N - 1$, where N is the size of the array (which must be greater than zero). The usual bracket notation is used to index arrays; an index must be present for all array accesses and the index expression must evaluate to an integer. Because 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. Arrays may be of either basic type (integers or booleans).

Assignment is only permitted for scalar values. Decaf uses value-copy semantics, and the assignment "`<loc> = <expr>`" copies the value resulting from the evaluation of `<expr>` into `<loc>`. If a variable is referenced before it is assigned, the result is undefined. For assignments to an array element, the index is evaluated before the value.

4.2 Functions

In Decaf, functions must be declared using the "`def`" keyword. Functions must either be "`void`" or have a single return value; they may take an unlimited number of parameters. Each parameter must have a formally-declared type. Arrays may NOT be passed as function parameters.

Every Decaf program must contain a function called `main` that takes no parameters and returns an `int`. Execution of the program begins at the `main` function. Functions may be called before their definition in the source code.

Decaf does not provide support for variadic parameters (e.g., "printf" in C).

4.3 Scope

Decaf uses very simple static scoping rules. There are at least two valid scopes at any point in a Decaf program: the global scope and the function scope. The global scope consists of names of variables and functions introduced at the top level of the source code. The function scope consists of names of variables and formal parameters introduced in a function declaration. Additional local scopes exist within each block in the code; these can come after `if` or `while` statements or anywhere there is a new block.

An identifier introduced in a function scope can *shadow* an identifier from the global scope. In this case, the identifier may only be used as a variable until the variable leaves scope. Similarly, identifiers introduced in local scopes shadow identifiers in less deeply nested scopes, the function scope, and the global scope.

No identifier may be defined more than once in the same scope. Thus, variable and function names must all be distinct in the global scope, and local variable names and formal parameters names must be distinct in each local scope.

4.4 Function Calls

Function 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.

Arguments are passed by value and the semantics can be thought of in terms of assignment: the formal arguments of a function are like local variables of the function 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 function body in sequence.

A function that is declared with a `void` return type can only be called as a statement, i.e., it cannot be used in an expression. Such a function 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 function 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. If control reaches the textual end of a function that returns a result, the result is undefined.

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

4.5 Control Structures

The `if` statement has the same semantics as in standard procedural programming languages. First, the conditional expression 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 "else" blocks be enclosed in braces, there is no ambiguity in matching an "else" block with its corresponding `if` statement.

The `while` statement also has the same semantics as in standard procedural languages. First, the guard expression is evaluated. If the result is true, the loop body is executed. After the loop body executes one iteration, the guard expression is re-evaluated. If the guard expression evaluates to false at any point at which it is evaluated, the loop terminates and execution resumes at the statement following the loop.

Decaf provides support for the standard `continue` and `break` statements, which immediately transfer control to the beginning or end of the loop (respectively). In the case of the former, the guard expression should be re-checked before the loop body is executed again. These statements may not occur outside of a while loop.

4.6 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, and operands are also evaluated left to right. Parentheses may be used to override normal precedence.

A location expression evaluates to the value contained by the location. The semantics of this differ depending on whether the location is an l-value or an r-value. Array operations are discussed in Section 4.1.

Method invocation expressions are discussed in Section 4.4.

Integer literals evaluate to their integer value. Boolean literals are evaluated to the integer values 1 (true) and 0 (false). String literals do not evaluate to anything; since variables cannot store strings, their only valid use is as an argument to predefined functions (e.g., the `print_str` function; see Section 4.7).

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 and may not be used on boolean variables. The equality operators (`==` and `!=` are valid for both `int` and `boolean` types, and can be used to compare any two expressions having the same type. The result of a relational operator or equality operator has type boolean. Boolean expression evaluation is not guaranteed to be short-circuited.

4.7 Input and Output

Decaf supports only very primitive I/O. There is currently no support for input; all input values must be hard-coded in the source. Output is supported using the following predefined library functions:

```
void print_str(string value)
void print_int(int value)
void print_bool(bool value)
```

The first two operate as expected. The last one (`print_bool`) prints the internal representation of the boolean (i.e., 1 for true and 0 for false). None of these functions add a newline at the end of the input; if newlines are desired they must be printed explicitly using the `print_str` function.

These functions are not actually implemented in Decaf (they are provided by the Decaf standard library) and thus must be manually added to the global symbol table during static analysis to ensure that calls to them are properly type-checked. Here are some examples of their use:

```
def int main()
{
    print_str("Hello!");            // result: "Hello!"
    print_int(2+3);                 // result: "5"
    print_bool(5 < 9);              // result: "1"
    print_bool(5 < 2);              // result: "0"
    return 0;
}
```

5 Interpreter/Architecture Details

The reference compiler for Decaf includes an interpreter that simulates running a Decaf program compiled to ILOC on an architecture that can directly execute ILOC. This architecture has limitations that are relevant to Decaf programmers although they are not part of the Decaf language itself.

Features of the simulator architecture:

- 64-bit words
- Unlimited 64-bit signed integer virtual registers
- Four special-purpose 64-bit integer registers:
    1. IP: instruction pointer
    2. SP: stack pointer
    3. BP: base pointer
    4. RET: function return value
- 64Kb address space with stack and static data regions
- Fixed-size, read-only code region indexed by instruction

The most important limitation to keep in mind is the 64Kb address space, which limits the size of the stack and therefore the depth of recursive functions.

In addition, the output functions listed in the previous section must be implemented using the `PRINT` instruction in ILOC, and as such must be handled as special cases during code generation.

6 Type Checking

Here is an incomplete list of basic type inference and type checking rules for Decaf. Each rule consists of a series of antecedents or premises (above the line) and a conclusion (below the line). In these rules, $\Gamma \vdash e : \tau$ means that "e has type τ in environment Γ". If τ is omitted, the statement simply means that "e is well-typed" (i.e., it has no type errors). Here, Γ (called the *typing context* or *type environment*) refers to the local symbol table (with the ability to look up symbols recursively through its parent table). The absence of Γ indicates that no symbol table is necessary to type-check e.

Expressions:

$$\text{TDec}\ \frac{}{\vdash \texttt{DEC} : \textbf{int}} \qquad \text{THex}\ \frac{}{\vdash \texttt{HEX} : \textbf{int}} \qquad \text{TStr}\ \frac{}{\vdash \texttt{STR} : \textbf{str}}$$

$$\text{TTrue}\ \frac{}{\vdash \texttt{true} : \textbf{bool}} \qquad \text{TFalse}\ \frac{}{\vdash \texttt{false} : \textbf{bool}} \qquad \text{TSubExpr}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{`(`}\ e\ \text{`)`} : \tau}$$

$$\text{TLoc}\ \frac{\texttt{ID} : \tau \in \Gamma}{\Gamma \vdash \texttt{ID} : \tau} \qquad \text{TArrLoc}\ \frac{\texttt{ID} : \tau[] \in \Gamma \quad \Gamma \vdash e : \textbf{int}}{\Gamma \vdash \texttt{ID}\ \text{`[`}\ e\ \text{`]`} : \tau}$$

$$\text{TFuncCall}\ \frac{\texttt{ID} : (\tau_1, \tau_2, ..., \tau_n) \to \tau_r \in \Gamma \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad ... \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{ID}\ \text{`(`}\ e_1, e_2, ..., e_n\ \text{`)`} : \tau_r}$$

$$\text{TNot}\ \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash\ \text{`!`}\ e : \textbf{bool}} \qquad \text{TNeg}\ \frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash\ \text{`-`}\ e : \textbf{int}}$$

$$\text{TAdd}\ \frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1\ \text{`+`}\ e_2 : \textbf{int}} \quad \text{(similar for TSub (`-`), TMul (`*`), TDiv (`/`) and TMod (`%`))}$$

$$\text{TLe}\ \frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1\ \text{`<`}\ e_2 : \textbf{bool}} \quad \text{(similar for TLeq (`<=`), TGe (`>`), and TGeq (`>=`))}$$

$$\text{TEq}\ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ \text{`==`}\ e_2 : \textbf{bool}} \quad \text{(similar for TNeq (`!=`))}$$

$$\text{TAnd}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \quad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1\ \text{`&&`}\ e_2 : \textbf{bool}} \quad \text{(similar for TOr (`||`))}$$

Statements:

$$\text{TIf}\ \frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b}{\Gamma \vdash \texttt{if}\ \text{`(`}\ e\ \text{`)`}\ b} \qquad \text{TIfElse}\ \frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash \texttt{if}\ \text{`(`}\ e\ \text{`)`}\ b_1\ \texttt{else}\ b_2} \qquad \text{TWhile}\ \frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b}{\Gamma \vdash \texttt{while}\ \text{`(`}\ e\ \text{`)`}\ b}$$

$$\text{TAssign}\ \frac{\Gamma \vdash \texttt{ID} : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ID}\ \text{`=`}\ e\ \text{`;`}} \qquad \text{TArrAssign}\ \frac{\Gamma \vdash \texttt{ID}[0] : \tau \quad \Gamma \vdash e_i : \textbf{int} \quad \Gamma \vdash e_v : \tau}{\Gamma \vdash \texttt{ID}\ \text{`[`}\ e_i\ \text{`]`}\ \text{`=`}\ e_v\ \text{`;`}}$$

$$\text{TBlock}\ \frac{\Gamma' \vdash s_1 \quad \Gamma' \vdash s_2 \quad ... \quad \Gamma' \vdash s_n}{\Gamma \vdash \text{`\{`}\ Vars,\ s_1, s_2, ..., s_n\ \text{`\}`}} \qquad \text{where } \Gamma' = \Gamma \cup \{v_i : \tau_i \in Vars\}$$

7 Operational Semantics

Here is an incomplete list of rules for generating ILOC code from Decaf code. In this sense they also encode *operational semantics* for the Decaf language. Each rule consists of a series of antecedents (above the line) and a conclusion (below the line). In these rules, $s \rightarrow C$ means "statement s generates code C." For expressions, $e \rightarrow \langle C,\ r\rangle$ means that "expression e generates code C with the result stored in virtual register r." The semicolon operator (;) is used to concatenate code.

Expressions:

$$\text{STrue}\ \frac{}{\texttt{true} \rightarrow \langle \texttt{loadI 1 => } r,\ r\rangle} \qquad \text{SFalse}\ \frac{}{\texttt{false} \rightarrow \langle \texttt{loadI 0 => } r,\ r\rangle}$$

$$\text{SInt}\ \frac{}{\texttt{INT} \rightarrow \langle \texttt{loadI INT => } r,\ r\rangle} \qquad \text{SLoc}\ \frac{r_b = \textbf{base}(\texttt{ID}) \quad x_o = \textbf{offset}(\texttt{ID})}{\texttt{ID} \rightarrow \langle \texttt{loadAI [}r_b\texttt{+}x_o\texttt{] => } r,\ r\rangle}$$

$$\text{SArrLoc}\ \frac{e \rightarrow \langle C_e, r_e\rangle \quad x_s = \textbf{size}(\texttt{ID}) \quad r_b = \textbf{base}(\texttt{ID})}{\texttt{ID[}e\texttt{]} \rightarrow \langle C_e;\ \texttt{multI } r_e\texttt{,}x_s\ \texttt{=>}\ r_o;\ \texttt{loadAO [}r_b\texttt{+}r_o\texttt{] => } r,\ r\rangle}$$

$$\text{SFuncCall}\ \frac{e_1 \rightarrow \langle C_1,\ r_1\rangle \quad e_2 \rightarrow \langle C_2,\ r_2\rangle \quad \ldots \quad e_n \rightarrow \langle C_n,\ r_n\rangle}{\texttt{ID(}e_1\texttt{,}e_2\texttt{,...,}e_n\texttt{)} \rightarrow \langle C_1;\ C_2;\ \ldots C_n;\ \left(\begin{smallmatrix}\texttt{push } r_n\texttt{; ... push } r_2\texttt{; push } r_1\texttt{; call ID;}\\ \texttt{addI SP, } 8n \texttt{ => SP; i2i RET => } r_r\end{smallmatrix}\right),\ r_r\rangle}$$

$$\text{SNot}\ \frac{e \rightarrow \langle C,\ r_1\rangle}{\texttt{`!`}\ e\ \rightarrow \langle C;\ \texttt{not } r_1\ \texttt{=>}\ r_2,\ r_2\rangle} \qquad \text{SAdd}\ \frac{e_1 \rightarrow \langle C_1,\ r_1\rangle \quad e_2 \rightarrow \langle C_2,\ r_2\rangle}{e_1\ \texttt{`+`}\ e_2\ \rightarrow \langle C_1;\ C_2;\ \texttt{add } r_1\texttt{,}r_2\ \texttt{=>}\ r_3,\ r_3\rangle}$$

(similar for SNeg (`-`)) (similar for SSub (`-`), SMul (`*`), SDiv (`/`), SAnd (`&&`), and SOr (`||`))

(**Note**: SMod (`%`) is purposefully omitted as an exercise for the reader)

$$\text{SLe}\ \frac{e_1 \rightarrow \langle C_1,\ r_1\rangle \quad e_2 \rightarrow \langle C_2,\ r_2\rangle}{e_1\ \texttt{`<`}\ e_2\ \rightarrow \langle C_1;\ C_2;\ \texttt{cmp\_LT } r_1\texttt{,}r_2\ \texttt{=>}\ r_3,\ r_3\rangle}$$

(similar for SLeq (`<=`), SGe (`>`), SGeq (`>=`), SNeq (`!=`), and SEq (`==`))

Statements:

$$\text{SIf}\ \frac{e \rightarrow \langle C_e,\ r_e\rangle \quad b \rightarrow C_b}{\texttt{if `(`}\ e\ \texttt{`)`}\ b\ \rightarrow C_e;\ \texttt{cbr } r_e\ \texttt{=>}\ l_1\texttt{,}l_2;\ l_1\texttt{:};\ C_b;\ l_2\texttt{:}}$$

$$\text{SIfElse}\ \frac{e \rightarrow \langle C_e,\ r_e\rangle \quad b_1 \rightarrow C_1 \quad b_2 \rightarrow C_2}{\texttt{if `(`}\ e\ \texttt{`)`}\ b_1\ \texttt{else}\ b_2\ \rightarrow C_e;\ \texttt{cbr } r_e\ \texttt{=>}\ l_1\texttt{,}l_2;\ l_1\texttt{:};\ C_1;\ \texttt{jump } l_3;\ l_2\texttt{:};\ C_2;\ l_3\texttt{:}}$$

$$\text{SWhile}\ \frac{e \rightarrow \langle C_e,\ r_e\rangle \quad b \rightarrow C_b}{\texttt{while `(`}\ e\ \texttt{`)`}\ b\ \rightarrow l_1\texttt{:};\ C_e;\ \texttt{cbr } r_e\ \texttt{=>}\ l_2\texttt{,}l_3;\ l_2\texttt{:};\ C_b;\ \texttt{jump } l_1;\ l_3\texttt{:}}$$

$$\text{SAssign}\ \frac{e \rightarrow \langle C_e,\ r_e\rangle \quad r_b = \textbf{base}(\texttt{ID}) \quad x_o = \textbf{offset}(\texttt{ID})}{\texttt{ID} = \text{ e } \rightarrow C_e;\ \texttt{storeAI } r_e\ \texttt{=>}\ \texttt{[}r_b\texttt{+}x_o\texttt{]}}$$

$$\text{SArrAssign}\ \frac{e_i \rightarrow \langle C_i, r_i\rangle \quad e_e \rightarrow \langle C_e, r_e\rangle \quad x_s = \textbf{size}(\texttt{ID}) \quad r_b = \textbf{base}(\texttt{ID})}{\texttt{ID[}e_i\texttt{] = } e_e\ \rightarrow C_i;\ C_e;\ \texttt{multI } r_i\texttt{,}x_s\ \texttt{=>}\ r_o;\ \texttt{storeAO } r_e\ \texttt{=>}\ \texttt{[}r_b\texttt{+}r_o\texttt{]}}$$

$$\text{SBlock}\ \frac{s_1 \rightarrow C_1 \quad s_2 \rightarrow C_2 \quad \ldots \quad s_n \rightarrow C_n}{\texttt{`\{`}\ s_1, s_2, ..., s_n\ \texttt{`\}`}\ \rightarrow C_1;\ C_2;\ \ldots C_n}$$

8 Application Binary Interface

The following calling conventions should be respected by all ILOC code generated from Decaf programs. In these conventions the *caller* is the procedure making the call and the *callee* is the procedure that is being called. Further, the *prologue* and *epilogue* are segments of code located at the beginning and end of every procedure, respectively. Finally, the *precall* and *postreturn* sequences are segments of code located immediately preceding and immediately following the actual `call` instruction.

- In the precall, the caller must perform the following tasks in order:
  1. Calculate the values of all actual parameters in order.
  2. Push all actual parameter values on the stack **in reverse order**.
  3. Save any live general-purpose registers (if applicable).
- In the prologue, the callee must perform the following tasks in order:
  1. Save the caller's base pointer by pushing it on the stack.
  2. Set the callee's base pointer using the stack pointer.
  3. Allocate space for local variables and/or spilled registers by decrementing the stack pointer. This operation must be emitted even if the number of bytes required is zero.
- The callee must access its parameters using positive static offsets from the base pointer.
- The callee must access its local variables using negative static offsets from the base pointer.
- In the epilogue, the callee must perform the following tasks in order:
  1. Save the return value in the `RET` register (if applicable).
  2. Restore the stack pointer from the base pointer.
  3. Restore the caller's base pointer by popping it from the stack.
- In the postreturn, the caller must de-allocate any space allocated for parameters that were pushed onto the stack.

Here is an example from the program in Section 1, showing the ILOC instructions that are executed in order:

```
...
// PRECALL
loadAI [BP-8] => r5                          // calculate parameters (in order)
loadI 2 => r6
push r6                                      // push parameters (reverse order)
push r5

call add                                     // push return address and jump
                // PROLOGUE
                push BP                      // save caller base pointer
                i2i SP => BP                 // save stack pointer as callee base pointer
                addI SP, 0 => SP             // allocate space for local vars (0 bytes)

                ...

                loadAI [BP+16] => r0          // local variables (positive offset from BP)
                loadAI [BP+24] => r1

                ...

                i2i r2 => ret                // save return value

                // EPILOGUE
                i2i BP => SP                 // restore stack pointer
                pop BP                       // restore caller's base pointer
                return                       // pop return address and jump

// POSTRETURN
addI SP, 16 => SP                            // deallocate parameters
...
```