Assignment 4
CSC488
Prof. Wortman
TA: Peter McCormick

Team 11
Faith Magcalas - g3hummus
Jingnan Chen - c6chenjl
Qi Jian Huang - c6huangu
Steven Low - g3lowste

Global Instructions

1. Unconditional branch, where label is the address:

```
BR label:
    PUSH label
    BR
```

2. Conditional branch, where label is the address:

```
BF label:
    PUSH label
    BF
```

3. False expression equals MACHINE_FALSE yields MACHINE_TRUE (false = false yields true)

```
NOT:
    PUSH MACHINE_FALSE
    EQ
```

Storage

Lexical level (LL):

- will start with 0 at main program scope, and will be incremented if new scope is created within current scope.

Order number (ON):

- representing the offset of the identifier from the beginning of the activity record, it will be incremented by the size of each added identifier (1 word for integer and boolean scalar variables, and (1 x number of elements) words for arrays).

Addressing:

- scalar variable address within the activation record is calculated based on sum of starting address of the activation record and the order number for the identifier).
- array variable address within the activation record is calculated based on sum of starting address of the activation record and the order number for the identifier.
- the array element is computed by adding row offset and column offset to the starting array address.

- for 1D array, the row offset is the difference between the accessed index and the lower bound for the array (row_offset = accessed_index - lb_index).
- for 2D array, the row offset is the difference between the accessed index and the lower bound in first dimension and multiply by the length of second dimension array (row_offset = (accessed_index (1st D) - lb_index (1st D) * (2nd dimension array length). The column offset is the difference between the accessed index and lower bound in second dimension (col_offset = (accessed_index (2nd D) - lb_index (2nd D)).

(a) **Variables in the main program**
In the main program, storage for variables will be allocated in the activation record of its containing scope (aka the main program). For each identifier, its lexical level and order number will be stored within the symbol table entry, and all identifiers values are determined during semantic analysis.

Examples:
Assume a variable x is defined, first assign integer constant 2 into variable x (x = 2), then perform operation x = x + 3

```
ADDR LL ON     // address of variable x on stack
PUSH 2         // push integer constant 2 onto stack
STORE          // store integer constant 2 into variable x in memory
ADDR LL ON     // address of variable x on stack
ADDR LL ON     // address of variable x on stack again
LOAD           // load the value of x from memory address onto stack
PUSH 3         // push integer constant 3 onto stack
ADD            // perform addition (3 + x)
STORE          // store (3 + x) into address of x in memory
```

(b) **Variables in procedures and functions**
Variables in procedures and functions are handled similar to variables in main program, except the lexical level will be greater than 0.

(c) **Variables in minor scopes (e.g. { declaration statement })**
Declared identifiers within a minor scope will belong to its major scope, which it means the lexical level will be the same, no new lexical level is created at this point.

(d) **Integer and boolean constants (e.g. 'true' and 'false')**
Integer and boolean constants will be pushed onto stack where needed, no need to allocate storage space for them.

Examples:

- integer constant 5:

  ```
  PUSH 5
  ```

- boolean constant true:

PUSH MACHINE_TRUE

- boolean constant false:

  PUSH MACHINE_FALSE

(e) **Text constants (e.g. "Like This")**
Text constants will be pushed onto stack as required.

Example:

- write "cat":

```
PUSH "c"
PRINTC
PUSH "a"
PRINTC
PUSH "t"
PRINTC
```

Expressions

(a) **Describe how the values of constants (including text constants) will be accessed.**
Push them onto stack as there is no need to store these values in the memory. See examples in Storage (d) and (e)

(b) **Describe how the values of scalar variables will be accessed.**
To access a value of scalar variable, the variable's memory address is pushed onto the stack, and then load the value of that memory address onto the stack.

Example:

```
ADDR LL ON          // Scalar variable address
LOAD                // Load the value from address to stack
```

(c) **Describe how array elements will be accessed. Show details of array subscripting in the general case for one and two dimensional arrays.**

For details, refer to "Storage → Addressing" section.

Examples:
For 1D:

- array base address:

  ADDR LL ON

- calculate row offset

```
PUSH accessed_index
PUSH lb_index
SUB
```

- add the offset to the array base address

```
ADD
```

- load the value of the address onto stack

```
LOAD
```

For 2D:

- array base address:

```
ADDR LL ON
```

- calculate the row offset

```
PUSH accessed_index_1D
PUSH lb_index_1D
SUB
PUSH array_length_2D
MUL
```

- calculate the col offset

```
PUSH accessed_index_2D
PUSH lb_index_2D
SUB
```

- add row and col offset then add it to array base address

```
ADD
ADD
```

- load the value of the address onto stack

```
LOAD
```

(d) **Describe how you will implement each of the arithmetic operators +, -, *, and /**
Arithmetic operators are already implemented in the machine instructions (ADD, SUB, MUL, DIV, NEG). They are implemented by recursively evaluating the two operands and then use the provided instruction to compute the operator result from the two evaluated operands.

Examples:

- 1 + 2:

```
PUSH 1
PUSH 2
ADD
```

- 2 - 1:

```
PUSH 2
PUSH 1
SUB
```

- 2 * 1:

```
PUSH 2
PUSH 1
MUL
```

- 2 / 1:

```
PUSH 2
PUSH 1
DIV
```

- -5:

```
PUSH 5
NEG
```

(e) **Describe how you will implement each of the comparison operators < , <=, =, 'not=', >=, >.**
The operators "LT" and "EQ" is provided, the other operators will be transformed using provided operators.

Examples:

- 1 < 2:

```
PUSH 1
PUSH 2
LT
```

- 2 <= 3: same as 3 not < 2

```
PUSH 2
PUSH 3
SWAP
LT
NOT
```

- 2 = 2:

```
PUSH 2
PUSH 2
EQ
```

- 3 not= 2:

```
PUSH 3
PUSH 2
EQ
NOT
```

- 3 >= 2: same as 3 not < 2

```
PUSH 3
PUSH 2
LT
NOT
```

- 3 > 2: same as 2 < 3

```
PUSH 3
PUSH 2
SWAP
LT
```

(f) **Describe how you will implement each of the boolean operators 'and' , 'or' , 'not'** Although there is a machine instruction 'OR', we will not be able to use it as it won't properly short circuit the operators, so instead we will handle or/and as follows:

- 'x or y': First we evaluate x and then check if we **cannot** short circuit, in which case we BF to _next. If we can short circuit however, we push the MACHINE_TRUE that was absorbed by BF back onto the stack, and then branch to _end.

  Or example:

```
emit code to evaluate x
BF _next
PUSH MACHINE_TRUE
BR _end
_next:
    emit code to evaluate y
_end:
    // ending code i.e. store, etc
```

- 'x and y': First we evaluate x and then check if we **can** short circuit, in which case we BF to _false where we PUSH MACHINE_FALSE. If we cannot short circuit however, we evaluate y and then branch to _end.

  And example:

```
emit code to evaluate x
BF _false
emit code to evaluate y
BR _end
_false:
```

```
        PUSH MACHINE_FALSE
_end:
        // ending code i.e. store, etc
```

- 'not x':

```
emit code to evaluate x
NOT
```

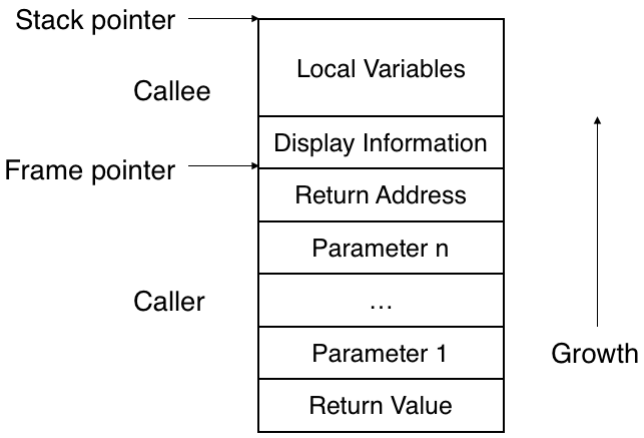
(g) **describe how you will implement conditional expressions**
conditional expressions are implemented using comparison operators and conditional/unconditional branch. In below examples, conditional statements also included.

Example: bool = (3 < 5 ? true : false): assume boolean scalar variable "bool" is defined

```
ADDR LL ON          \\ 'bool' variable address
PUSH 3
PUSH 5
LT                  \\ compare using 'LT' comparator
BF _label_else
PUSH MACHINE_TRUE
BR _label_end
_label_else:
    PUSH MACHINE_FALSE
_label_end:
    STORE           \\ store result into 'bool'
```

Functions and Procedures

(a) **Activation record for functions and procedures**



(b) **Procedure and function entrance code**
When entering a routine (procedure or function) both the caller and callee have different responsibilities for setting up the activation record. The caller of a function is responsible for pushing a space for the return value, as well as the parameters and return address, however, a procedure call does not need to have a space for return value. The caller then branches to the callee, who takes care of display management and pushes the local variables onto the stack. A generic example is shown below
Example:

- Caller responsibilities:

```
PUSH UNDEFINED                  // return value.
PUSH parameter_1
...
PUSH parameter_n
PUSH program_counter            // return address
BR callee_address               // branch to the callee
```

- Callee responsibilities

```
ADDR LL 0                       // save the display
PUSHMT
SETD LL
PUSH local_1
...
PUSH local_n
```

(c) **Procedure and function exit code**
Function exit code is responsible for doing essentially the opposite of the entrance code. First, the callee will push the number of local variables onto the stack (which we will keep track of,) such that we can use the POPN instruction to pop them all off the stack, we then restore the display and branch to the return address. Once there, the caller will do the same POPN instruction to pop the parameters off the stack, and we will leave the return value on the stack so the caller can use it. A generic example is shown below.
Example:

- Callee responsibilities

```
PUSH n                          // Where n is number of locals
POPN
SETD LL
BR
```

- Caller responsibilities

```
PUSH n                          // where n is number of parameters
POPN
```

(d) **Describe how you will implement parameter passing**
Parameter passing is shown above in function entrance code (part b). The only other consideration not mentioned, is that since a parameter is an expression, we cannot directly push it onto the stack, the expression must first be evaluated before it can be pushed onto the stack. Specific details on expression evaluation can be found in the expression section.
Example:

```
emit code to evaluate parameter_1
PUSH parameter_1
...
emit code to evaluate parameter_2
PUSH parameter_2
```

(e) **Describe how you will implement function call and function value return**
Function call implementation is described in detail in entrance and exit code (parts (b) and (c).) The return value of a function is left on top of the stack when we exit the function, so that it can be used by the callee.

(f) **Describe how you will implement procedure call**
Procedure call implementation is described in detail in entrance code and exit code (parts (b) and (c).) The only difference from procedure to function is that we do not leave space for return values.

(g) **Describe your display management strategy**
Display management is described in function exit and entrance code (part (b) and (c).) When entering the function, the callee saves the display, and when exiting the function, the callee restores it.

Statements

(a) **Assignment statement**
variable := expression

```
ADDR LL ON
emit code to evaluate expression
STORE
```

(b) **If statements**
'if' expression 'then' statement1 'else' statement2

```
emit code to evaluate expression
BF _else
emit code to execute statement1
BR _end
_else:
    emit code to execute statement2
_end:
    // ending code
```

'if' expression 'then' statement

```
emit code to evaluate expression
BF _end
emit code to execute statement
_end:
    // ending code
```

(c) **While and repeat statements**

'while' expression 'do' statement

```
_start:
    emit code to evaluate expression
    BF _end
    emit code to execute statement
    BR _start
_end:
    // ending code
```

'repeat' statement 'until' expression

```
_start:
    emit code to execute statement
    emit code to evaluate expression
    BF _start
// continue when expression true
```

(d) **All forms of exit statements**

- case 1: exit: When an exit statement is found, an unconditional branch is executed to exit its containing loop.
- case 2: exit integer: When an "exit integer" is found, an unconditional branch is executed, however the exit statement label needs to be the correct loop ending.
- case 3: exit when true: The only difference between the previous two is that "exit when true" is using conditional branch "BF" rather than unconditional branch "BR"
- case 4: exit integer when true: This case is the combination of case 2 and 3

(e) **Return statements**

- For Procedures: return statements are not necessary in this case. Usually at the end of a procedure, it should clean up of all the local variables of the procedure on the stack, and also there is an instruction that always branches back to where it gets called and continue to execute the next instruction.
- For Functions: return statements are necessary in this case. Usually at the end of a function, we should store the return value to the location that is reversed by the caller, then we clean up of all the local variables of the function on the stack,and also there is an instruction that always branches back to where the function gets called and continue

to execute the next instruction.

Example:

```
emit code to find out the reserved address for return value
ADDR LL -n      // we are tracing back on the stack, and the
                // offset -n depends on the number of arguments
SWAP
STORE
BR              // Branch back to where the function gets called
                // (similar in procedure case)
```

(f) **'read' and 'write' statements**

- 'write' expression

```
emit code to evaluate expression
PRINTI
```

- 'write' text
  for each character:

```
PUSH character
PRINTC
```

- 'write' 'newline'

```
PUSH '\n'
PRINTC
```

- 'write' sequence of outputs
  for each output:

```
check if it's an expression, text, or 'newline'
emit code to execute appropriate 'write' instructions
```

- 'read' variable

```
ADDR LL ON
READI
STORE
```

- 'read' sequence of variables
  for each variable:

```
execute instructions for 'read' variable
```

(g) **Handling of minor scopes**
Statements in minor scopes are treated the same as statements in a major scope.

Other

(a) **Main program initialization and termination**
A few steps will be done for the main program initialization, while the termination is done using the machine instruction HALT.

Main program initialization:

- push stack pointer onto stack

```
PUSHMT
```

- setup display entry from stack, LL = 0 for main program

```
SETD LL
```

- reserve enough storage needed for the scope

```
PUSH UNDEFINED
PUSH scope_size_in_word
DUPN
```

Main program termination:

```
HALT
```

(b) **Any handling of scopes not described above**
There is no further handling of scopes needed. The minor scope's storage requirement is already handled when the major scope is initialized. The handling of major scopes already is explained in the Storage section.

(c) **Any other information that you think is relevant**
The PUSH operations above can be replaced with any other sequence of operations that result in an integer, a text constant or a boolean onto the stack where applicable.