# CSC488: Assignment 4
# Code Generation Templates
Winter 2014

Group 7:
g0sharma, g1liyuch, g0haugan, g1minmin

## Storage

### a. Variables in the Main Program

Storage for variables in the main program will be allocated within an activation record for the main program. The amount of local storage the main program requires will be computed and stored during semantic analysis. This information can then be used when the activation record is created to allocate all necessary space up front. Storage for array variables will take up multiple words in the activation record depending on its size

See Section 5a for additional information on main program initialization.

Addressing for variables will use lexic level (LL) and offset values, computed during semantic analysis and stored in each identifier's symbol table entry, along with the display's reference for the LL. The LL for the main program will be 0, and the reference to its activation record is display[0].

The display provides the base address of the activation record and the offset is a value that when added to the base results in the address of the memory word allocated for the specific variable or array element. Offsets will take into account the size of variables stored prior to a specific variable, such as arrays that may utilize more than one word of memory.

### b. Variables in Procedures and Functions

Allocating storage for, and addressing variables in, procedures and functions will be done very much the same way as for those stored in the main program. Each procedure and function, when called, will have an activation record created and all necessary space will be allocated up front using stored information for the function or procedure. Addressing is identical to the main program, except that each function and procedure will have a specific LL other than 0.

In general, loading a variable onto the stack requires the following instructions, given the variables specific LL and offset:

ADDR LL offset

LOAD

**c. Variables in Minor Scopes**

Variables in minor scopes will not require any special handling. Their storage requirements will be included in the major scope than encloses them, so space is allocated for them as well when the main program, function or procedure activation record is created. It then remains to ensure that the symbol table entries for these varaibles share the same LL as the major scope that contains them and that their offsets are computed correctly.

One downside to this approach is that space will need to be allocated for all minor scopes at once, even though non-nested minor scopes will never be active at the same time. The upside is that we can allocate all needed storage in a few instructions up front, deallocate it at once during cleanup and calculating offsets will be a little simpler.

**d. Integer and Boolean Constants**

Storage for scalar constants will be provided by the runtime stack. These values will be pushed directly onto the stack during expression evaluation since their values are stored directly in the AST nodes of a particular expression subtree. We did not feel there was an advantage to storing them permanently in memory somewhere, as loading them to the stack would require as many instructions as pushing them directly and it would use unnecessary space.

**e. Text Constants**

Similar to scalars, each character of a text constant will be pushed directly to the stack for printing as needed. This is a little more unwieldly with text constants since each character must be pushed separately, but again, storing the constant permanently somewhere would still require many PUSH and LOAD instructions to get it onto the stack. The specific text string itself is stored in the AST node in every output statement it appears in, so specific code for it can be generated directly.

## Expressions

**a. Describe How the Values of Constants (Including Text Constants) Will Be Accessed.**

The values of constants (integer and boolean value) will be pushed directly to the stack, since they will be used during the expression evaluation. We can directly get their value through accessing the node in AST. For text constants, we will push each character of the text constant to the stack. Since the text constants only appear in the output statement, a PRINTC is followed by each push_char statement during the code generation for printing the entire text constant.

**b. Describe How the Values of Scalar Variables Will Be Accessed.**

The values of scalar variables will be accessed by first looking up the symbol table for their Lexical Level and Order Number. After we have the LL and ON of the variable, we can basically use ADDR to get the address of the variable from the current display and load it to the stack.

Machine Code:

```
ADDR(LL ON of the variable)
LOAD
```

**c. Describe How Array Elements Will Be Accessed.**

Since array will take up multiple words in display, to access an element of an array, we will calculate the address on the fly using the information of its base address and its offset (the index). The array element addressing algorithm is:

1D array elements will have address = base array address + machine_index1
2D array elements will have address = base array address + (machine_index1 * length of inner arrays + machine_index2)

For simplicity, every index is started from 0 in the machine instruction even if the real index of the array doesn't start from 0. The index translation is done during code generation by: machine_index = real_index - lower_bound. This is applicable for index in both first and second dimension.

**d. Describe How You Will Implement Each of the Arithmetic Operators +,-,* and /**

By the time we are evaluating any operators, we assume there are already 2 values (or 1 for unary operator) being pushed onto the top of the stack. So that we can directly apply the operators to them. The arithmetic operators are all provided to us in the machine instruction.

Machine Code:

```
+:
        ADD
-:
        SUB
*:
        MUL
/:
        DIV
-:
        NEG
```

**e. Describe How You Will Implement Each of the Comparison Operators <, <=, =, not=,**

**>=, >**

Two comparison operators are directly provided for us: EQ for = and LT for <, so we can use those two for = and < operator. For greater than operator >, we can first apply SWAP to swap the two values in the stack and apply LT. We can always use a series of following machine instructions to implement the logical not expression: PUSH(1); SUB; NEG. Basically we are substracting the value by 1 and negate it. Thus 1 becomes 0 and 0 becomes 1. For not =, we can first do = and then apply the logical not instructions. To implement <=, we can first do > and then apply logical not. Similiarly for >=, we can first do < and then apply logical not.

Machine Code:

```
<:
        LT

<=:
        SWAP
        LT
        PUSH(1)
        SUB
        NEG

=:
        EQ

not =:
        EQ
        PUSH(1)
        SUB
        NEG

>=:
        LT
        PUSH(1)
        SUB
        NEG

>:
        SWAP
        LT
```

**f. Describe How You Will Implement Each of the Boolean Operators and, or, not**

The machine instruction for operator or is provided to us: OR. And as described in 2.e, logical not operator can be implemented by: PUSH(1); SUB; NEG. For and operator, we have found that operator MUL can be directly used to implement the logical and. Clearly, 1 MUL 1 = 1, 1 MUL 0 = 0, 0 MUL 1 = 0, 0 MUL 0 = 0, which has the same result as the logical and operator.

Machine Code:

```
    and:
        MUL
    or:
        OR
    not:
        PUSH(1)
        SUB
        NEG
```

## Functions and Procedures

### a. The Activation Record for Functions and Procedures

The activation record for each function and precedure will have the following structure, starting at low memory to high:

```
// beginning of record
Return Value
Return Address
Saved Display Reference
Parameters   // display base address
.
.
.
Locals
.
.
.
// end of record
```

### b. Procedure and Function Entrance Code

Responsibilities for setting up the activation record for a function or procedure are divided between the caller and callee.

The caller is responsible for a lot of the set up, including creating a slot for the eventual return

value (for functions), pushing the return address and saved display reference, updating the display and evaluating the parameters. The caller will then branch to the first instruction of the routine.

The first instructions of any routine will be a small function prologue that allocates space in the activation record for local variables. This is the responsibility of the callee. After this, the routine executes.

Caller:

```
PUSH UNDEFINED   // return value, to be filled in, function only
PUSH UNDEFINED   // return address, to be filled in
ADDR LL 0        // saved display reference

// display update
PUSHMT
SETD LL          // LL of function / procedure

// evaluate parameter expressions, one after another

PUSH addr_routine   // addr_routine stored in symbol table, which is the address of the first
                       statement in routine body

BR
```

Callee:

```
// allocate local storage
PUSH UNDEFINED
PUSH routine_needed_words
DUPN
```

**c. Procedure and Function Exit Code**

The callee is entirely responsible for cleanup of the routine call through the function epilogue instructions. Once the callee branches back to the return address, its allocation record has been removed from the stack, the display has been reverted to its previous state and the return value, if there is one, is on top of the stack. The address of the return value is calculated by taking the address of the activation frame and subtracting three words.

Callee:

```
// function only, this is essentially the result statement
```

```
ADDR LL 0
PUSH 3
SUB // get the address of the return value, which is the display base address - 3 (refer to 3.a)
// evaluate return expression
STORE

// function and procedure epilogue
PUSH num_params + num_local_words
POPN
SETD LL                 // LL of function / procedure
BR
```

**d. Parameter Passing Implementation**

As described above, the caller is responsible for evaluating each parameter expression and leaving its value on the stack before branching to the callee. Since the display entry is updated just prior to this, the first parameter will have an offset of 0 and the last parameter will have an offset of n - 1, where n is the number of parameters. The number of parameters is known at compile time, is stored with the routine entry in the symbol table and can be used to compute offsets. Parameters are then referenced and updated as any other variable.

**e. Function Call and Function Value Return**

Function call and value return are detailed above. The return value will be on top of the stack after a function has executed to be used in assignment, further expression evaluation, etc. Code will be generated to allocate space for a return value and to evaluate and store the return expression when dealing specifically with a function, as noted above. The code to evaluate and store the return value will really be generated by the result statement.

**f. Procedure Call Implementation**

Again, this process is essentially the same as for a function call, as detailed, minus the allocation and evaluation of a return value. This changes the caller's responsibilities by one instruction (no initial PUSH UNDEFINED) and does not change the callee's cleanup at all since no result statment will be present generate return value code.

**g. Display Management Strategy**

As recommended, we have chosen to use the constant-time display update algorithm. To implement this, whenever a function or procedure is called, before updating the display to point to the new allocation record, the current display entry for the routine's lexic level is pushed onto the stack. This stored value is then restored into the display during routine cleanup. This reverts the display back to the state it was in before the routine was called.

## Statements

### a. Assignment Statement

Assignment statements can be implemented by loading a variable's address in memory onto the stack, evaluating the expression to assign to it, then storing the value.

```
ADDR LL offset          // LL and offset of specific variable
// evaluate expression
STORE
```

### b. If Statements

If statements involve a branching technique that we will also use in loops and before function/procedure code to allow us to write branch instructions, but leave the branch address blank until a later time when it can be filled in.

To handle an if without an else, we will generate a branch on false sequence, then evaluate the conditional expression and branch to the first instruction after the if statement if the condition is false. If the expression is true, the branch will fail and execution falls through to the true statement(s).

```
// evaluate conditional expression
PUSH UNDEFINED          // Fill in with address of the first instruction following if
BF
// instructions for true statement(s)
```

To handle an if with an else, the branch on false sequence will branch to the else statement(s) if the expression is false, as opposed to the code after the if statement. If the false branch succeeds, the else statement(s) are executed and execution falls through to following instructions. If the false branch fails, execution can fall through to the true statement(s), but we must also insert an unconditional branch after the true statement(s) to branch over the else statement(s) to the first following instruction.

```
// evaluate conditional expression
PUSH UNDEFINED          // Fill in address of first else instruction
BF
// instructions for true statements(s)
PUSH UNDEFINED          // Fill in address of first instruction following if statement
BR
// instructions for else statement(s)
```

To fill in the UNDEFINED addresses, the addresses will be pushed to a separate stack that the code generator maintains. When the code generator reaches the first else statement, or the first statement following an if statement, it knows the address that it needs to write to the corresponding UNDEFINED slot. At these times, the code generator will pop the top address off this storage stack and write the value into that previous memory location, completing the branch sequence.

Care will need to be taken with if statements specifically that this is done in the correct order. To ensure this, after code has been generated for all true statement(s), if there is an else, the code generator will pop the top stored address and write the current address + 3 (the number of words needed for the branch sequence at the end of the true statement(s)) into that address, completing the false branch at the top of the if.

Thus, the top branch will point to what will be the first address of the else statement(s) and the top branch sequence is completed before we add the address of the branch sequence at the end of the true statement(s) to the storage stack.

Similiarly for the second branch address. When we reach the end of the else statements, we can fill in the memory before the BR instruction with the current PC address.

**c. the While and Repeat Statement**

For the while statement, we emit a forward branch BF instruction on false, and the address of this branch is patched once we reach the end of the compound while statement. The forward branch is patched exactly once. If the while expression evaluates to true, the forward branch fails and execution falls to the first instruction in the compound statement.

We save the address of the expression as the target address of the future backward branch. At the end of the true statement(s), we emit the backward branch, BR instruction, to direct execution back to the beginning of the loop.

```
// evaluate conditional expression
PUSH UNDEFINED                  // Fill in with address of the first instruction after while loop
BF
// instructions for statement(s) inside the loop
PUSH a                          // Push address of expression (start address of the loop)
BR                              // Branch back to the start address of the loop
```

For the repeat statement, we emit a backward branch BF instruction on false, and the address of this branch is the address of the the first repeat statement. If the expression evaluates to true, the BF instruction fails, and we continue execution with first instruction after expression.

```
// instructions for statement(s) inside the loop
```

```
// evaluate conditional expression
PUSH a                                  // Push address of first repeat statement
BF                                      // Branch back to the start address of the loop
```

**d. Exit Statement**

The same branching technique applies to the exit statement. For exit, we first push an undefined value to the stack and then followed by a BR instruction. The address to branch is filled when we reach the end of the loop.

In exit when statement, we need to firstly evaluate the expression, and then apply the logical not operator to the expresion to get the negated boolean value. Finally we push the address of instruction after loop and BF to it. Again, the address of instruction after loop will be filled in later when we reach the end of the loop.

exit:

1. PUSH(addr of instruction after loop)
2. BR

exit when expression:

1. Evaluate expression and push on top of stack
2. Do logical NOT procedure to negate
3. PUSH(addr of instruction after loop)
4. BF, jump to outside of loop if expression is false (that is, it was originally true)

**e. Return and Result Statement**

As the activation record for each function and precedure will have the following structure, starting at low memory to high:

```
// beginning of record
Return Value
Return Address
Saved Display Reference
Parameters   // display base address
.
.
.
Locals
.
.
.
```

// end of record

The result statement need to find the activation record base address first, where the return value of the function call should be stored. And then it can evaluate the expression in the result statement and store it in the activation record base address. After that, we can clean up the local variables and parameters, reset display and branch back to the caller by the following code as described in 3.c:

```
PUSH num_params + num_local_words
POPN
SETD LL                 // LL of function / procedure
BR
```

The return statement is similar to result statement except that there is no expressions to evaluate. Therefore we can skip the expression evaluation and return value storage process. After we branch back to the caller, the stack does not contain the undefined return value because we didn't push it when we call a procedure.

**f. Get and Put Statement**

The get statement is done by first pushing the address of the target variable to the stack, and then read the integer value from stdin, then store the value to the target address.

There are three cases that is needed to handle for the put statement. If the output is an expression, then we need to evaluate the expression first and then print the value of the result as integer. If the output is a text, then we need to print every character in the text. The way to do it is by pushing and printing each character in the text. If the output is newline, then there is only one character to print. We can just push the newline character to the stack and print it out using PRINTC.

Machine instructions:

```
GET:
for each INPUT:
        ADDR(LL ON) // get LL ON of the variable from symbol table
        READI
        STORE

PUT:
for each OUTPUT:
        if OUTPUT is expression:
                evaluate the expr
                PRINTI
```

```
if OUTPUT is text:
        for all char in text:
                PUSH(char)
                PRINTC
if OUTPUT is newline:
        PUSH(newline)
        PRINTC
```

## Everything Else

### a. Main Program Initialization and Termination

Every program will begin with a few special setup instructions. These instructions essentially set up an activation record for main, even though there is no explicit call to a “main” function. As such, this activation record differs slightly from those for functions and procedures. Namely, there will be no storage for a return value, return address, saved display entry nor parameters. Instead, display[0] will be set and space allocated for all the local memory main will need.

Essentially,

```
PUSHMT
SETD 0
PUSH UNDEFINED
PUSH main_needed_words
DUPN
```

To terminate the main program, a single HALT instruction will be added after the final instruction of main.

### b. Any Handling of Scopes Not Described Above

We do not have any further special handling of scopes. As all the spaces for local variables in minor scopes are already allocated on the stack at the time of major scope initialization.

### c. Any Other Relevant Information

We considered using a word of storage above the MLP to temporarily store the return value of functions, which would then be reloaded back to the stack, but decided our current approach was simpler and more natural. We also considered the benefits of storing text and scalar constants in this memory region, but as discussed, did not feel there would be huge benefits from this. Therefore, we do not currently store anything above the MLP and the MLP will simply point to the end of memory.

One other important thing worth mentioning, is that as we are writing instructions into the code segment of memory, we must include branch instructions before writing function or procedure instructions that will branch over that particular routine if execution of instructions through the code segment has reached that point without the routine actually being called. When a routine is actually called, the caller will branch to its first instruction, which will be after the branch instructions. If execution reaches that point otherwise, the branch will be execute to jump the routine code and continue with the following instructions afterward.

Similar to conditional and loop statements, these branch instructions will involve writing a PUSH instruction, leaving a blank for the address to push to be filled in later followed by a branch instruction. After generating the code for the routine, we know the address of the instruction following and can return to fill in the branch address.