Team 04
Haohan Jiang, g3jiangh
Maria Yancheva, c2yanche
Timo Vink, c4vinkti
Chandeep Singh, g2singh

1 Storage

1.1 Variables in the main program

Storage for each variable value will be reserved in the activation record of its containing scope. We will augment our symbol table, which is currently a tree of scopes, to store a lexical level and order number for every identifier. These values are set upon insertion during semantic checking. The lexical level will start at 0 (for the outermost scope), and a new scope will have a lexical level that is one higher than its parent scope's lexical level. When an identifier is declared within a scope, it will be assigned an order number, representing the offset of the identifier (in number of words) from the beginning of the activation record. The first order number will be 3 or 4, depending on whether this is a procedure or function (more on this in section 3), and will be incremented by the size of each added identifier (i.e., 1 for integer and Boolean scalar variables which are stored as 16 bits padded with zeros, and 1× (number of elements) for arrays.

The main program is treated as an anonymous procedure, and will thus have an activation record (lexical level of 0, i.e. outermost scope). The return address will be memory location 0, which is where we'll store a HALT instruction.

Scalar variables: they are of type integer or Boolean (stored as 16 bits, or 1 word, in memory). Each scalar variable declared in the main program will be allocated storage of 1 word in the activation record for lexical level 0. Their address within the activation record can be computed as the sum of starting address for the activation record (obtained at run-time by retrieving it from display register 0) and the order number for the identifier (obtained at compile-time from the respective symbol table entry).

Array variables: their elements are of type integer or Boolean (each element is stored as 16 bits, or 1 word, in memory). Each array declared in the main program will be allocated storage of 1 * (number of elements) words in the activation record for lexical level 0. We assume all arrays are row-major indexed. Their address within the activation record can be computed as the sum of the starting address for the activation record (obtained by indexing the display vector by the lexical level) and the order number for the identifier (obtained from the respective symbol table entry). The address of individual array elements can be computed by adding the appropriate row offset and column offset to the starting array address. For 1D arrays, the row offset is the difference between the accessed index and the lower bound for the array. For 2D arrays, the row offset is the difference between the accessed index in dimension 1 and the lower bound for dimension 1, multiplied by the stride for the first dimension. The column offset is the difference between the accessed index in dimension 2 and the lower bound for dimension 2.

1.2 Variables in procedures and functions

Variables in the scopes of procedures and functions are handled exactly the same way as described in Subsection 1.1. The only difference is that the enclosing procedure or function activation record would have a lexical level greater than 0. The lexical level for each scope is stored with that scope in the symbol table tree. When addressing the variables, we obtain the starting address for the activation record by retrieving it from the display register for the lexical level of the corresponding scope, and then proceed as described earlier in Subsection 1.1.

1.3 Variables in minor scopes

Identifiers declared in minor scopes belong to the enclosing major scope. All identifiers declared in the enclosing major scope are visible and accessible within minor scopes. Therefore, for the purposes of scoping rules, minor scopes do not need any special handling.

1.4 Integer and Boolean constants

Integer and Boolean constants will just be pushed on to the stack where required. Example:

```
PUSH    5
PUSH    MACHINE_FALSE
```

1.5 Text constants

Text constants will similarly simply be output where needed.

```
PUSH    67
PRINTC
PUSH    83
PRINTC
PUSH    67
PRINTC
```

2 Expressions

2.1 Describe how the values of constants will be accessed

As described above we simply push them unto the stack as we do not store these values in a fixed spot in memory. Example:

```
PUSH    3
PUSH    MACHINE_TRUE
```

2.2 Describe how the values of scalar variables will be accessed

To access a scalar variable, we would first push the variable's memory address onto the stack (expressed as the starting address of the activation record for the lexical level of the scope of declaration, and the order number for the specific identifier being accessed which is stored in its symbol table entry), and then load the value of that memory address onto the stack. Example:

```
# Scalar variable address
ADDR    lexical_level    identifier_order_number

# Load the value from that memory address onto the stack
LOAD
```

2.3 Describe how array elements will be accessed. Show details of array sub-scripting in the general case for one and two dimensional arrays

To access a 1D array element, we add the row offset for the element being accessed to the array's base address, and then load the value from the resulting address. The row offset is computed as the difference between the array index being accessed and the lower bound which is available in the AST node for the array, and can be accessed through a link in its symbol table entry. We note that since the array subscript can be determined by an expression, that the line that pushses this value on to the stack can be replaced by any sequence of operations that result in an integer being added to the top of the stack. Example:

```
# Array base address
ADDR    lexical_level    array_order_number

# Compute the row offset
PUSH    array_subscript
PUSH    array_lower_bound
SUB

# Add the offset to the starting array address
ADD

# Load the value from that memory address onto the stack
LOAD
```

To access a 2D array element, we do the same thing as for 1D arrays, except we add (row offset * stride for dimension 1) and (col offset) to the starting array address. The row offset is computed as the difference between the dimension 1 array index being accessed and the lower bound for dimension 1; the col offset is computed as the difference between the dimension 2 array index being accessed and the lower bound for dimension 2. The lower bounds and the stride are known at compile time because arrays are declared with integer bounds rather than expressions.

```
# Array base address
ADDR    lexical_level    array_order_number

# Compute the row offset * stride
PUSH    array_subscript_dim1
PUSH    array_lower_bound_dim1
SUB
PUSH    stride_dim1
MUL

# Compute the col offset
PUSH    array_subscript_dim2
PUSH    array_lower_bound_dim2
SUB

# Add the two offsets to the starting array address
ADD
```

```
ADD

# Load the value from that memory address onto the stack
LOAD
```

2.4 Describe how you will implement each of the arithmetic operators

We have been provided arithmetic operators for +, − (both unary and binary), *, and / as machine instructions, and so, no further implementation beyond calling those instructions are needed. For all of the following examples, the PUSH operations can of course be replaced with any other sequence of operations that combined add a single integer to the stack.

Example: 3 + 4

```
PUSH    3
PUSH    4
ADD
```

Example: 4 - 3

```
PUSH    4
PUSH    3
SUB
```

Example: -3

```
PUSH    3
NEG
```

Example: 4 * 3

```
PUSH    4
PUSH    3
MUL
```

Example: 4 / 3

```
PUSH    4
PUSH    3
DIV
```

2.5 Describe how you will implement each of the comparison operators

Again, for all of the following examples, the PUSH operations can of course be replaced with any other sequence of operations that combined add a single integer or Boolean to the stack as appropriate.

For < and = operations, we can simply call the machine instructions LT and EQ.

Example: 3 < 4

```
PUSH    3
PUSH    4
LT
```

Example: 4 = 4

```
PUSH    4
PUSH    4
EQ
```

For > operation, we can use the LT instruction and swap the order of the operands to emulate the output of a > operator. That is, first emit code for calculating the right operand, then the left operand.

Example: $5 > 4$

```
PUSH    4
PUSH    5
LT
```

For the ≠ operator, we can perform a `EQ` operation and then invert the result.

Example: $4 \neq 5$

```
PUSH    4
PUSH    5
EQ
NEG
```

For the ≤ operator, we can perform a > operation as described above and then negate the result. More about how that works can be found in the description of the ! operator.

Example: $4 \leq 5$

```
PUSH    MACHINE_TRUE
PUSH    5
PUSH    4
LT
SUB
```

For the ≥ operator, we can perform a < operation and then negate the result. More about how that works can be found in the description of the ! operator.

Example: $5 \geq 4$

```
PUSH    MACHINE_TRUE
PUSH    5
PUSH    4
LT
SUB
```

2.6 Describe how you will implement each of the Boolean operators

For the ! operator, we observe that the machine represents TRUE as 1, and 0 as FALSE. We can thus implement this operator as a subtraction from 1, since $1 - 0 = 0$ and $1 - 1 = 0$.

Example: ! True

```
PUSH    MACHINE_TRUE
PUSH    MACHINE_TRUE
SUB
```

For | and ! operators, we can simply use the machine instructors OR and NEG.

Example: true | false

```
PUSH    MACHINE_TRUE
PUSH    MACHINE_FALSE
OR
```

For the & operator, we observe the machine has no `AND` instruction, but we can apply De Morgan's laws to represent and `AND` using `OR` and negations instead. The equivalence of interest is $a \wedge b \equiv \neg(\neg a \vee \neg b)$.

Example: false & true

```
PUSH    MACHINE_TRUE
PUSH    MACHINE_TRUE
PUSH    MACHINE_FALSE
SUB
PUSH    MACHINE_TRUE
PUSH    MACHINE_TRUE
SUB
OR
SUB
```

2.7 Describe how you will implement anonymous functions

Anonymous functions are to be implemented the same way as non-anonymous functions except that there will be no identifier stored in the symbol table for the function. Also since anonymous functions are declared and called in the same location, the entrance/setup code can directly precede the function code, and the exit/cleanup code can come right after the function code.

3 Functions and Procedures

3.1 The activation record for functions and procedures

The activation record for functions and procedures will contain the following data (starting from the bottom-most stack entry): return value (only for functions), return address, dynamic link (link to caller's activation record), display_m (hold data for callee method), parameters, local variables. Before branching to the function entrance code, we need to:

1. On a call from level N to M, save current display[M] into the caller: "ADDR <N> <2 or 3>" (depending on func/proc, get display_m addr), "ADDR <M> 0", "STORE".
2. Allocate space for return value (if function): "PUSH undefined"
3. Allocate space for return address: "PUSH undefined" (this is patched later)
4. Allocate space and save dynamic link: "ADDR <N> 0"
5. Allocate space for display_m: "PUSH undefined"
6. Evaluate parameter expressions and write to activation record (see 3.4).
7. Update display[m] to point to current activation record: "PUSHMT" to get the current position of stack pointer, then "PUSH <num_params + 2 or 3>" (depending on if function or procedure), then "SUB" to get to start of this new activation record. Then "SETD <M>" to update the display to point to the new activation record.
8. Before branching to function entrance code, patch the return addr to be the address of the instruction after the branch instruction

9. Branch to function entrance code

3.2 Procedure and function entrance code / prologue

Allocate space for local storage in the activation record. This includes "PUSH undefined", "PUSH <size>" where size is how much space is needed for local storage, and then "DUPN" which will "PUSH undefined" <size> times.

3.3 Procedure and function exit code / epilogue

The following actions need to be taken in the exit code (or the epilogue) of procedure and functions:

1. The current activation record should currently be at the top of the stack, and since we are done with the function itself, we can clear all the local storage. So whatever <size> that was previously allocated to local storage, we need to emit a "PUSH <size>" and then a "POPN" to clear the top <size> entries.
2. Clear the parameter storage (if any): "PUSH <num_params>", "POPN"
3. We can do another POP to get rid of the display_m data (don't need this anymore).
4. Update the display for the callee's method's lexical level (M). In the prologue we saved this data in the caller's activation record. We need to push that onto the stack and update. Since we have cleared all local storage, params storage, and display_m, the dynamic link is now at the top of the stack. So just increment the address to get the display_m, and update display[M]: "PUSH <2 or 3>" (depending on proc/func, get caller's display_m addr), "LOAD", and then "SETD <M>".
5. Emit "BR" to branch to the return address.
6. Now we are left with the return value at the top of the stack, and the code we branched to can deal with that however it needs to.
7. Assuming that the code branched to will pop the return value off the stack, we've now freed all memory that was used for the activation record

3.4 Describe how you will implement parameter passing

The parameters need to be processed before we branch to the function entrance code, and after we allocate the control block in the activation record (return value, return address, and display data). The parameter expressions should simply be processed from left to right, and the final results left on the stack. This should result in the first parameter expressions's result being at the bottom, followed by the 2nd on top of that, etc. When done, branch to the function entrance code.

3.5 Describe how you will implement function call and function value return

For function call, the arguments if any are processed as explained in parameter passing (3.4). And the return value is handled as explained in the function exit code (3.3)

3.6 Describe how you will implement procedure call

Procedure call is handled basically the same way as a function call, with the only difference being that we don't allocate space for a return value in the activation record.

3.7 Describe your display management strategy

We are using the constant cost display management algorithm. It is as described below:

On a call from lexical level N to M, save display[M] to the caller's activation record. Update display[M] to callee's activation record when it is called, and then when returning from callee, restore the display data that we saved in the caller. The instructions for this are described in section 3.3.

4 Statements

4.1 Assignment

The address of the left-hand side variable is pushed onto the stack, followed by the right-hand side expression, and then the STORE instruction is applied. In the following examples, of course the PUSH instruction for the RHS can be replaced by any sequence of operations that combined result in a single value being added to the stack. Example:

```
# Memory address of LHS identifier
ADDR    enclosing_lexical_level     identifier_offset

# RHS value
PUSH    evaluated_expression

# Store value into memory address of identifier
STORE
```

4.2 If

For statements with the "**IF** expression **THEN** statement **END**" construct, we would first emit code for the expression and push that onto the stack. We can then use the *BF* instruction to point to the address after the code generated for the statement.

```
PUSH    evaluated_expression
PUSH    <addr_x>
# addr_x is the address after the code emitted for the statement
BF
# Execute generated code for statement
```

For statements with the "**IF** expression **THEN** statement **ELSE** statement **END**" construct, we would emit code for the expression and push that onto the stack. We then use the *BF* instruction to direct where we should resume execution.

```
PUSH    evaluated_expression
PUSH    <addr_x>
# addr_x is the address of the location where the code for the "else" statements
BF
# Code for "then" statement
...
...
BR      <addr_end>
# Code for "else" statement
```

4.3 While and Loop

While loop: we use two branches, as described below:

1. *addr_false* is the memory address of the instructions after the while loop (to be executed when the test expression evaluates to false). During code generation, this address is initialized to the address of the current instruction being executed using the value stored in the program counter register; after the code for the entire while loop is generated this address is filled in appropriately.
2. *addr_back* is the memory address of the branch false instruction at the top of the loop. After the loop body is executed, the test expression is pushed back onto the stack, and the **BR** instruction sets the program counter back to the first instruction for the loop (the expression testing).

```
PUSH    evaluated_expression
PUSH    <addr_false>
BF
# INSERT HERE: generated code for statements in 'true' condition
PUSH    evaluated_expression
PUSH    <addr_back>
BR
```

Loop: we use a single branch, as described below:

1. *addr_back* is the memory address of the first instruction in the loop body. After executing the loop, the **BR** instruction sets the program counter back to the first instruction, repeating the loop, unless the statements in the body of the loop contain an exit statement.

```
# INSERT HERE: generated code for statements in loop body
PUSH    <addr_back>
BR
```

4.4 Exit

For Exit statements, we will generate an unconditional branch instruction to the address just beyond the loop.

```
# INSERT HERE: generated code for statements in loop body
PUSH    <addr_x> # addr_x is filled in after we find where the loop ends
BR
```

4.5 Return

For a naked "Return" statement, we simply unconditionally branch to the epilogue code to perform function clean up.

```
PUSH    <addr_epiloguecode>
BR
```

For "**return** expression" constructs, we would push the expression first and then perform the same instructions as a naked "return" statement.

```
PUSH    evaluated_expression
PUSH    <addr_epiloguecode>
BR
```

4.6 Get and Put

Note: since only integer inputs can be read in from standard input (as per language semantic specifications), the **READC** instruction is never used.

Example: get a, b

```
ADDR    lexical_level   a_order_number
READI
STORE
ADDR    lexical_level   b_order_number
READI
STORE
```

Example: put a, "b", skip

```
ADDR    lexical_level   a_order_number
LOAD
PRINTI
PUSH    98
PRINTC
PUSH    10
PRINTC
```

5 Everything Else

5.1 Main program initialization and termination

As mentioned in Section 1, the main program will be allocated an activation record (lexical level of 0, i.e. outermost scope). Since we want to terminate the program when all code is executed, we push the **HALT** instruction first, at the beginning of the program block in memory. Then, an activation record is allocated for the main program, at lexical level 0. The main program is treated as an anonymous procedure whose return address is the address of the instruction HALT, and which has no return value.

The epilogue code for the main program would pop everything from the stack, and branch unconditionally to the first address in the program block in memory (which is the **HALT** instruction), thereby terminating successfully.

5.2 Any handling of scopes not described above

All scoping information is described in the previous sections.

5.3 Any other information that you think is relevant

Just that you are a great professor! Thanks for taking the time to read!