CSC488 Code Generation Templates

Ou Ye, yeou
Theresa Ma, matheres
Lawrence Wu, wulawre1
Zixuan Jaimie Jin, jinzi2
Gregory Wlodarek, wlodare1

1 Storage

During the code generation, we will associate each variable with its lexical level and offset (the number of 16-bit words from the starting address of the activation record at its lexical level) in the symbol table so that we can access the variable. The address of a scalar variable will be `display[lexical level] + offset`; the address of an array variable at index `i` will be `display[lexical level] + offset + (i - lower bound)`.

1.1 Variables in the main program

Variables in the main program will be stored in the activation record of the main scope, which will be pointed by the 0th display register (lexical level = 0). Each declared variable will be initialized with `UNDEFINED`. We will emit the following instructions, where `num_vars` is the total number of variables being declared in the main program scope (and its minor scopes):

```
PUSH UNDEFINED
PUSH $num_vars$
DUPN
```

Note that `num_vars` counts the size of the array for every array variable.

1.2 Variables in procedures and functions

For variables in procedures or functions, we calculate the total storage required to store all the variables and allocate that much storage in the activation record of the procedure or function scope (lexical level > 0), using the same formula (example) as in 1.1.

1.3 Variables in minor scopes

We will handle variables in minor scopes the same way as variables in their enclosing major scopes. That is, we will calculate the storage required for all the variables in the minor scopes and allocate the storage in the activation record of the enclosing major scope (same formula/example as in 1.1); this happens when we allocate the storage for its enclosing major scope. The lexical level of the variables in minor scopes will be the same as their enclosing major scope. The offset value will be different even if the variable in the minor scope is declared with the same name as some variables in its enclosing major scopes.

1.4 Integer and boolean constants

For integer and boolean constants, we push the constant onto the stack.

Example for integer constant `1`:

```
    PUSH 1
```

Example for boolean constant `MACHINE_TRUE`:

```
    PUSH MACHINE_TRUE
```

1.5 Text Constants

For text constants, we push each letter onto the stack one at a time. Because text constants are only used as the output to write, we will not push the next character until the character on the stack is printed (more details in 2.1 and 4.6)).

Example for storing the first character (using ASCII decimal code as specified by machine.pdf) in the text constants `"abc"`:

```
PUSH 97
```

2 Expressions

(x and y in the examples below are expressions that have already been evaluated)

2.1 Accessing values of constants

As described in 1.4, we access the value of a scalar constant by directly processing the top value on the stack after pushing the constant onto the stack.

Example for computing `1+1` by accessing the constant `1`:

```
    PUSH 1
    PUSH 1
    ADD
```

For text constants, as described in 1.5, they are used only in the write statement so they will be accessed by printing the characters one at a time.

Example for accessing `"abc"`:

```
    PUSH 97
    PRINTC
    PUSH 98
    PRINTC
    PUSH 99
    PRINTC
```

2.2 Accessing scalar variables

We access the value of a scalar variable by looking up its address (lexical level, offset) from the symbol table. For a scalar variable of level i with offset j:

```
    ADDR i j
    LOAD
```

2.3 Accessing array elements

We access array elements by computing the index offset from the initial array element. For example, to access the element at index i of an array starting at level n with offset m (lower bound k), we compute k - i, which will be the index offset in 16-bit words, and add the result to the starting address of the array:

```
PUSH i
PUSH k
SUB
ADDR n m
ADD
LOAD
```

2.4 Implementation of arithmetic operators

We will use the machine instructions ADD, SUB, MUL, DIV. We must make sure that the stack has y on the top and x below y for x op y.

For x + y:

```
PUSH x
PUSH y
ADD
```

For x - y:

```
PUSH x
PUSH y
SUB
```

For x * y:

```
PUSH x
PUSH y
MUL
```

For x / y:

```
PUSH x
PUSH y
DIV
```

2.5 Implementation of comparison operators

For the comparison operator < (x < y), we will use the machine instruction LT:

```
PUSH x
PUSH y
LT
```

For the comparison operator <= (x <= y), we will first compute LT, then EQ, then OR:

```
PUSH x
PUSH y
LT
PUSH x
PUSH y
EQ
OR
```

For the comparison operator = (x = y), we will use the machine instruction EQ:

```
PUSH x
PUSH y
EQ
```

For the comparison operator ‘not’ = (x not= y), we first compute =, and then negate the result by subtracting it from 1:

```
PUSH 1
PUSH x
PUSH y
EQ
SUB
```

For the comparison operator >= (x >= y), we swap the variables and then compute LT, and check equality, and boolean OR the results:

```
PUSH x
PUSH y
SWAP
LT
PUSH x
PUSH y
EQ
OR
```

For the comparison operator > (x > y), we will compute <= like above, and then negate the result:

```
PUSH x
PUSH y
SWAP
LT
```

2.6 Implementation of boolean operators

For the boolean operator ‘and’ (x and y), since `MACHINE_TRUE` is represented as 1 and `MACHINE_FALSE` as 0 (according to Machine.java), we add the two booleans and check if they sum up to 2:

```
PUSH x
PUSH y
ADD
PUSH 2
EQ
```

For the boolean operator ‘or’ (x or y), we simple use the machine instruction OR.

```
PUSH x
PUSH y
OR
```

For the boolean operator ‘not’, we subtract the evaluated boolean expression from 1, since 1 - 0 (`MACHINE_FALSE`) = 1 (`MACHINE_TRUE`), and 1 - 1 (`MACHINE_TRUE`) = 0 (`MACHINE_FALSE`).

```
PUSH 1
PUSH x
SUB
```

2.7 Implementation of conditional expressions.

For a condition expression in the form `( e1 ? e2 : e3 )`, we first push the evaluated expression of `e1` onto the stack, then push `F_addr`, which is the starting address of the code for `e3` (the actual address will be filled in after emitting all the code prior to `e3`), then BF. We emit the code for executing `e2` after BF, then push `T_addr`, which is the starting address of the code after the entire conditional expression `( e1 ? e2 : e3 )` (the acutal address will be filled after emitting all the code for this conditional expression), then BR.

```
          PUSH e1
          PUSH F_addr
          BF
          % code to execute e2
          PUSH T_addr
          BR
F_addr:   % code to execute e3
T_addr:   % code to execute after the entire conditional expression
```

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 of the stack entry:

```
Return value (Only for function)
Return address
Display_addr (the display value before the call at the callee's lexical level)
Parameters
Local variables
```

When changing from one activation record to another, we must save the current display at the procedure/function's lexical level as `display_addr`, and allocate spaces for the items in the activation records, which includes return value and return address. Every parameters are evaluated and passed to the following activation record (See 3.4 for details).

3.2 Procedure and function entrance code

Upon entering the procedure or function, the first step is to setup the current activation record, which includes updating the display and passing the parameters, then allocate space for local storage of variables on the activation records.

```
PUSH UNDEFINED            % Function only; return value (details in 3.3 and 3.5)
PUSH <return_addr>        % Will be filled with the starting addr of the code after
                          % this call, after emitting all the code up to BR below
ADDR <lexical_level> 0    % Store display at callee's level before entering the proc/fn
 % evaluate the paramaters and leave them on the stack
PUSHMT                    % Push msp onto stack
PUSH <num_param>          % Make sure that the display[lexical_level] below points to
SUB                       % the beginning of the parameters on the stack
SETD <lexical_level>      % Update display; lexical_level = caller's lexical_level + 1
PUSH <fn_proc_addr>       % To BR to the code for the function/procedure;
BR                        % The address will be stored in the symbol table
% return_addr: code to execute after executing the function/procedure call
% the followings are the starting code at the address fn_proc_addr
PUSH UNDEFINED             % Local storage
PUSH <num_var>             % Local storage, num_var many 16-bit words
DUPN                       % Local storage
```

3.3 Procedure and function exit code

There are numbers of actions need to be taken in a procedure and function exit code.

1. Deallocating all local storage for local variables and parameters if there are any by popping all the items on the stack until reaching the display[current_lexical_level]:

```
PUSHMT
ADDR <current_lexical_level> 0
SUB
POPN
```

2. Update the display at the callee's lexical level using the display_addr saved in 3.2 (since everything above display_addr has been popped):

```
SETD current_lexical_leval
```

3. The display is set and popped (next value is the return address), we can use it restore to the previous activation record:

```
BR
```

For functions, the return value will be left on the top of the stack, and the following activation record can do whatever they want to it with it, either pop it or store it.

3.4 Implementation of parameter passing

For each parameter expressions, we evaluate in order and push each result onto the stack, then store their lexical level and offset (with respect to the procedure/function's activation record) in the symbol table. After all the parameters have been pushed to the stack, we branch to the code to execute the function/procedure, and set the current display to point to the beginning of the parameter (first parameter has offset 0). Then the parameters can be used by the procedure/function like other local variables.

```
% Suppose param_1 ... param_n are evaluated already
PUSH param_1
...
PUSH param_n
```

3.5 Implementation of function call and function value return

The function arguments are passed in the way described in 3.4. The return value is handled as explained in 3.3. Furthermore, the details on how the functions are called are shown in 3.2.
For the return value, we store the evaluated result of the return statement to `display[current_lexical_level]` - 2, which is the space allocated by the caller for the return value as shown in 3.2:

```
ADDR <current_lexical_level> 0
PUSH 2
SUB
PUSH <return_value>
STORE
```

3.6 Implementation of procedure call

Similar to function calls, the only difference is there is no steps for allocating storage for return value in the activation record (no return value handling).

3.7 Display management strategy

We are using the constant cost display update management algorithm from lecture 8 slides 47. The save operation is done by the caller, and the update and restore operations are done by the callee.

1. On a call from level L to level M
2. Save display[M] in the local storage of the calling procedure. (Parent)
3. Called procedure sets display[M] to point at its activation record. (Child)
4. On return, restore the saved value of display[M]. (Child)

4 Statements

4.1 Assignment statement

To assign an expression `e` to a variable `x`, we first get the address of `x` (access the lexical level and offset from the symbol table), compute `e`, and then store `e` to `x`. For a variable with lexical level `i` and offset `j`:

```
ADDR i j
Compute e - make sure e is the last thing on the top of the stack
STORE
```

4.2 If statements

For an `if expr then statement` statement, we first evaluate `expr` and then branch to the code segment containing `statement` if the `expr` is false, the code after BF, which is the code segment containing `statement`, will be executed if the `expr` is false (like the conditional expression, Rest will be filled in after emitting all the code of this if statment):

```
      PUSH expr
      PUSH Rest
      BF
      % code to execute statement
Rest: % code to executes after this if-statement
```

For an `if expr then t_stmt else f_stmt` statement, we do exactly the same thing as the conditional expression. First, we push the evaluated expression of `expr` onto the stack, then push `F_addr`, which is the starting address of the code for `f_stmt` (the actual address will be filled in after emitting all the code prior to `f_stmt`), then BF. We emit the code for executing `t_stmt` after BF, then push `T_addr`, which is the starting address of the code after this if-statment (the acutal address will be filled after emitting all the code for this if-statement), then BR.

```
          PUSH expr
          PUSH F_addr
          BF
          % code to execute t_stmt
          PUSH T_addr
          BR
F_addr:   % code to execute f_stmt
T_addr:   % code to execute after the entire if-statement
```

4.3 The while and repeat statements

For a `while expr do statement` statement, we create a code segment containing the `statement`, evaluate the expression, and jump to `END`, which is the address of the code after the while loop statment, if `expr` is false. At the end of the `LOOP` code segment, we loop back to the beginning (Like above, `END` will be filled after all code for the while-statment have been emitted. The program address pointed by `LOOP` will be stored in advance.):

```
LOOP: PUSH expr
      PUSH END
      BF
      % code to execute statement
      PUSH LOOP
      BR
END:  % code to execute after the entire while loop statement
```

For a `repeat statement until expr` statement, we do something similar, but perform the check at the end:

```
LOOP: % code to execute statement
      PUSH expr
      PUSH LOOP
      BF
END: % code to execute after the entire repeat loop statement
```

4.4 All forms of exit statements

For `exit` and `exit integer` statements, we will generate an unconditional branch instruction that contains the address that jumps out of the current minor scope (looping statements) as specified (the acutal program address is filled when the place after exiting is encountered and everything prior to that has been emitted).

```
    PUSH Addr
    BR
```

For `exit when expr` and `exit integer when expr` statements, we evaluate expr and branch to `Addr` using bf by getting the negation of expr:

```
    PUSH 1
    PUSH expr
    SUB
    PUSH Addr
    BF
```

4.5 Return statements

There are two types of `return` statements, for a procedure return statement (no return expression), we simply create a unconditional branch instruction, and jumps to `addr`, which contains the exit code as specified in 3.3 (we save some code for procedures with multiple returns).

```
PUSH addr
BR
```

For a `return` statement with an expression (for functions), we first evaluate the `expr`, then push the evaluated result to the stack, then jump to `addr`, which contains the exit code as specified in sections 3.5 (first) and 3.3.

```
PUSH expr    % evaluated expression expr
PUSH addr
BR
```

4.6 'read' and 'write' statements

For the statements in the form of `read v`, only standard integer inputs can be read in from standard input, hence, `READC` is never used (according to Semantics.pdf). We first evaluate the variable `v`, then make the address of the variable be on the top of the stack, then read the input and store it from stack into the address of `v`. For example, to read in `v` with the value stored in lexical level i and offset j:

```
ADDR i j
READI
STORE
```

For the `write expr` statement (as stated by the semantics.pdf, `expr` should be evaluated to integer value), we first evaluate the expression `expr`, then make the result be on the top of the stack, and then print it out. For example, to write an integer variable at lexical level i and offset j (`write a`):

```
ADDR i j
LOAD
PRINTI
```

For writing text constants (`write "abc"`):

```
PUSH 97 % ASCII representation
PRINTC
PUSH 98 % ASCII representation
PRINTC
PUSH 99 % ASCII representation
PRINTC
```

For writing newline (`write newline`):

```
PUSH 10 % ASCII representation
PRINTC
```

4.7 Handling of minor scopes

See 1.3 for more details. We treat a minor scope as part of its enclosing major scope. That is, we do not create activation record for minor scopes, everything inside the minor scopes will be directly attached to the activation record of the major scope, and no value will be popped from the stack at the end of the minor scope.

5 Everything Else

5.1 Main program initialization and termination

When we initialize the program, we set the starting address of the main scope to the 0th display register:

```
PUSHMT
SETD 0
```

When we terminate the program, we pop everthing from the stack and halt:

```
PUSHMT
ADDR 0 0
SUB
POPN
HALT
```

5.2 Handling of scopes not described above

None.

5.3 Other relevant information

None.