CSC488 Assignment 4
Code Generation Design Document

Kevan Hollbach, Tarang Marathe,
George Gianacopoulos, Arturo Mena Cruz,
Andrew Trotter

15 March, 2017

1 Storage

1.a Variables in the main program

At the beginning of the program, enough space is allocated for all variables in the main program scope. The main program scope's allocation also includes enough space for all variables in any minor scopes that are descendants of the main program scope (see §1-**(c)** for more details).

Variables are addressed relative to the beginning of the allocation. The address of the beginning of the main program scope's allocation is stored in `display[0]`.

Assuming that that a total of `N` words of space must be allocated for the main program scope, the following code will be generated to do the allocation:

```
PUSHMT          // set display[0] to point to the beginning of the allocation
SETD 0
PUSH 0          // allocate N words of space
PUSH N
DUPN
```

1.b Variables in procedures and functions

Similar to how space is allocated in the main program scope, enough space is allocated for all variables in a function/procedure `f` in `f`'s prologue. `f`'s allocation also includes enough space for all variables in any minor scopes that are descendants of `f`'s scope (see §1-**(c)** for more details). `f`'s allocation occurs immediately above f's parameters (if any exist).

Variables are addressed relative to the address of `f`'s first parameter. Assuming, the address of `f`'s first parameter is stored in `display[X+1]`. Consequentially, **programs with function/procedure declaration nesting depth greater than** `displaySize - 1` **will not compile**.

The code generated to do `f`'s allocation is similar to the code in §1-**(a)**. Full specification of code generated for function/procedure allocations can be found as part of the template in §3-**(b)**.

1.c Variables in minor scopes

We define a minor scope s to be a **descendant** of a major scope S if and only if:

(1) s is a direct child of S in the program AST, or

(2) s is the direct child of a descendant of S in the program AST.

For any minor scope s, s will be a descendant of some major scope S. Any space that needs to be allocated for s will be allocated as part of S's allocation.

1.d Integer and boolean constants

There will be no explicit storage for integer and boolean constants, they will be appear as arguments to `PUSH` instructions. See §2-**(a)** for details.

1.e Text constants

Text constants are stored in the constant pool. Addressing of text constans is embedded in the program code at compile-time. Text constants are printed using a procedure specified in §4-**(f)**.

2 Expressions

2.a Constant values (including text constants)

Only strings (text constants) will appear in the constant pool (see 1. (e) for the details on how strings are handled). Integer and boolean constants will be pushed onto the stack when encountered, as follows:

```
constant expression                     machine code
===================                     ============
false                                   PUSH MACHINE_FALSE

true                                    PUSH MACHINE_TRUE

72                                      PUSH 72

-105                                    PUSH -105
```

2.b Scalar variables

To access a scalar variable, we need to load a value onto the stack from the correct location in memory. This is done using the display registers, and the `(LL, ON)` addressing method described in the Assignment 3 handout.

Assume that `x` is declared as a scalar variable in some major or minor scope, `LL` is the lexical level (depth) of this scope, and `ON` is the order number (offset) of the variable among all the variables declared in that scope.

```
variable reference                    machine code
==================                    ============
x                                     ADDR LL ON
                                      LOAD
```

2.c Array elements

We assume that:

- `a[x..y]` is an array
- `a` has address (i, j) in the symbol table

The following code will be generated for the expression `a[<EXPN>]` (where `<EXPN>` is some arbitrary expression that evaluates to an integer):

```
ADDR i j
...code to evaluate <EXPN>...
PUSH x
SUB     // normalize the value of <EXPN>
ADD     // calculate the address of a[<EXPN>]
LOAD
```

2.d Arithmetic operators

To evaluate binary arithmetic expressions, the child expressions need to be evaluated first. Assuming they are properly evaluated, their values will be on top of the stack. Then the appropriate machine instruction is executed, leaving the resulting value of the binary expression on the stack.

```
arithmetic expression         machine code
=====================         ============
   <expr1> + <expr2>          ...code to evaluate <expr1>...
                              ...code to evaluate <expr2>...
                              ADD

   <expr1> - <expr2>          ...code to evaluate <expr1>...
                              ...code to evaluate <expr2>...
                              SUB

   <expr1> * <expr2>          ...code to evaluate <expr1>...
                              ...code to evaluate <expr2>...
                              MUL

   <expr1> / <expr2>          ...code to evaluate <expr1>...
                              ...code to evaluate <expr2>...
                              DIV
```

2.e Comparison operators

Similarly, for comparison operators:

```
comparison expression           machine code
=====================           ============
   <expr1> = <expr2>            ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                EQ

<expr1> not = <expr2>           ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                EQ
                                PUSH MACHINE_FALSE
                                EQ

   <expr1> < <expr2>            ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                LT

   <expr1> > <expr2>            ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                SWAP
                                LT

   <expr1> <= <expr2>           ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                SWAP
                                LT
                                PUSH MACHINE_FALSE  // boolean NOT
                                EQ

   <expr1> >= <expr2>           ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                LT
                                PUSH MACHINE_FALSE  // boolean NOT
                                EQ
```

2.f Boolean operators

```
boolean expression              machine code
====================            ============
 <expr1> or <expr2>             ...code to evaluate <expr1>...
                                ...code to evaluate <expr2>...
                                OR

 <expr1> and <expr2>            ...code to evaluate <expr1>...
```

```
                            ...code to evaluate <expr2>...
                            MUL

not <expr>                  ...code to evaluate <expr>...
                            PUSH MACHINE_FALSE
                            EQ
```

2.g Conditional expressions

For conditional expression `(<EXPN1>?<EXPN2>:<EXPN3>)`, we assume that:

- The code to evaluate `<EXPN3>` begins at address `x`
- The beginning of the code after the conditional expression begins at address `y`

The following code will be generated to evaluate the conditonal expression:

```
...code to evaluate <EXPN1>...
PUSH x
BF
...code to evaluate <EXPN2>...
PUSH y
BR
...code to evaluate <EXPN3>...    // address x points here
...continue execution...          // address y points here
```

3 Functions and Procedures

3.a Activation records

We assume for function or procedure `f` that:

- `f`'s scope has lexical depth `x`
- `f` takes `n` parameters

The following is a diagram of `f`'s activation record, relative to the pseudo-machine's memory:

Return Value *(in misc. data pool)*
↑ *Free Space* ↓
Local Variables
Parameter `n` ⋮ Parameter `1`
Saved `display[x]`
Return Address
Rest of Stack ↓

3.b Routine entrance code

We assume for function or procedure `f` that:

- `f` has return address `A`
- `f` takes `N` parameters
- `f`'s local allocation is `M` words in size
- `f`'s declaration is nested within `X-1` function/procedure declarations

Entrance code for `f` is generated as follows:

```
PUSH A          // push return address
PUSH 0          // allocate space to save display[X]
...pass arguments to f... // See 3. (d) for details.
...call f...              // See 3. (e) for details.

PUSHMT          // calculate pointer to the pointer to first parameter
PUSH N
SUB
DUP             // calculate address to store display[X]
PUSH 1
SUB
ADDR X 0        // push display[X]
STORE           // store display[X] in the allocated space
SETD X          // store pointer to the first parameter in display[X]

PUSH 0          // allocate M words of space for local variables
PUSH M
DUPN
```

Note that:

(1) saving the return address, allocating space to save a display register, and pushing parameters for `f` are taken care of by the caller, while

(2) saving and updating the appropriate display register, as well as allocating space for locals are taken care of by the callee.

3.c Routine exit code

We assume for function or procedure `f` that:

- `f`'s declaration is nested within `X-1` function/procedure declarations
- The return value has been temporarily stored at address `Y`
- `f`'s return value (if applicable) is at the top of the stack. Directly beneath f's return value (or at the top of the stack if there is no return value) is `f`'s local variable allocation.

```
...store the return value (if applicable)...
PUSHMT          // pop all locals & params
ADDR X 0
SUB
POPN
SETD X          // restore the saved value for display[X]
...load return value and place under return address (if applicable)...
BR              // return
```

3.d Parameter passing

We assume for function or procedure `f` that:

- `f` takes `n` parameters
- Each of the parameters passed to `f` are expressions, denoted by: `<E_1>, <E_2>, ... , <E_n>`

Parameters are pushed on to the stack by evaluating each in the following way:

```
...code to evaluate <E_1>...    // each expression is pushed onto the stack
...code to evaluate <E_2>...
            .
            .
            .
...code to evaluate <E_n>...
```

3.e Function call and value return

We assume for function `f` that:

- `f`'s code begins at address `X` in memory
- A word in memory reserved for temporarily storing return values has address `Y`
- When describing how the return value is stored, the return value is already at the top of the stack.

`f` is called in the following way:

```
...push return address...
...allocate space to save display value...
...push parameters...
PUSH X
BR
```

`f`'s return value is stored in the following way:

```
PUSH Y          // temporarily store the return value
SWAP
STORE
```

```
...clean up locals & params...
...restore saved display register...
PUSH Y          // restore return value to top of stack
LOAD
SWAP            // swap return value and return address
...return...
```

3.f Procedure call

We assume for procedure `f` that:

- `f`'s code begins at address `X` in memory

  `f` is called in the following way:

```
...push return address...
...allocate space to save display value...
...push parameters...
PUSH X
BR
```

3.g Display management

When a a function/procedure `f` that uses `display[X]` is called, `f` saves `display[X]` in its prologue, and restores `display[X]` in its epilogue. More consisely: display registers are **callee-saved**.

4 Statements

4.a Assignment statements

Assign statements in our language will have the form `a := <expr>`, where `a` is an already declared variable, and `<expr>` is either a valid expression that evaluates to an integer or boolean, or an atomic integer or boolean. Therefore, in order to assign the value of the expression to the variable, we will first push the the address of the variable onto the stack using `ADDR LL_a ON_a`, then evaluate the expression so that its return value is on the top of the stack, and then call `STORE` to store that value at the variable's address. For example:

```
assign statement          machine code
================          ============
a := true                 ADDR LL_a ON_a
                          PUSH MACHINE_TRUE
                          STORE

a := (3 + 6) / 3          ADDR LL_a ON_a
                          <code to evaluate (3 + 6) / 3>
                          STORE
```

4.b If statements

The first thing that needs to be done is to evaluate the conditional and push it to the top of the stack, then we can push the address of the "else" block. With these two values at the top of the stack the command BF can use this to branch when the conditional is false. In addition to this we have to add an unconditional branch command at the end of "then" block of the if-statement making it branch to after the else-block. Other than that, the code to execute the statement(s) in the "then" and "else" blocks is just generated recursively and placed in the instruction space one afer the other.

```
 if statement                 machine code
===============               ============

if <expr> then                <evaluate <expr>>
    <statement1>              PUSH else_start_addr
else                          BF
    <statement2>              <evaluate statement1>
                              PUSH else_end_addr
                              BR
                              <evaluate <statement 2>>  // else_start_addr
                              // else_end_addr
```

4.c While and repeat statements

While statements in the language have the form `while <boolean expr> do <statement 1> ... <statement N>`. Therefore `<boolean expr>` has to be checked at the beginning of each loop, and if it is false we branch to the next instruction. Similarly, at the end of each loop (when each statement has been evaluated), we branch back to the beginning (unconditionally). Therefore the corresponding machine code is:

```
   statement                        machine code
===============                     ============

while <bool expression> do          <evaluate <bool expression>>  // start_loop_address
    <statement 1>                   PUSH end_loop_address
    ...                             BF
    <statement N>                   <evaluate <statement 1> ... <statement N>>
                                    PUSH start_loop_address
                                    BR
                                    // end_loop_address
```

Note: `end_loop_expression` is the address of the next instruction after the while-loop. Therefore the `BF` (branch-false) instruction will go to that instruction if the `<bool expression>` evaluates to false. Similarly, `start_address` is the address of the first instruction of the while loop, so at the end of each loop the `BR` (unconditional branch) takes execution back to the beginning of the loop.

Repeat statements have the form `repeat` { `<statement 1> ... <statement N>` } `until <bool expression>`. Therefore, the `<bool expression>` is checked at the end of each loop through the statements, and we only move on to the next instruction when it is set to true (unless there is an `exit` instruction among the statements - this case is dealt with later).

```
    statement                          machine code
================                       ============

repeat <statement>                     <evaluate <statement>>      // start_address
    until <bool expression>            <evaluate <bool_expression>>
                                       PUSH start_address
                                       BF
```

`start_address` is the address of the first instruction of the repeat-loop machine code to be evaluated. If `bool_expression` is false, the `BF` instruction ensures that the repeat-loop statements are executed once again by going back to `start_address`. If not, we move on to the next line of execution.

4.d Exit statements

```
    statement                          machine code
================                       ============

exit when <expression>                 <evaluate <expression>>
                                       PUSH MACHINE_FALSE
                                       EQ
                                       PUSH end_loop_address
                                       BF
                                       // ...other loop code...
                                       // end_loop_address
```

4.e Return statements

In order to access the return value from a function, we assign a block of the constant memory at the top of the memory space where the function's return value is written before branching out of the function. Let this location be memorySize-1 (i.e. 8191). The value at this address is then immediately retrieved and placed at the top of the stack.

```
function statement                       machine code
==================                       ============

function f(<params>) : <type>            PUSH 8191 \\ address of top of memory
    { <statements>                       <evaluate <expr>>
      return with <expr> }               STORE
```

```
<function exit and cleanup code>
PUSH 8191
LOAD
```

4.f Read and write statments

Read statements have the form `read a, b, c` where `a, b, c` are integer variables that have already been initialized, and take their values from standard input. Therefore the machine code is similar to assign statements, except the value is retreived from standard input usin `READI` and placed at the top of the stack.

```
read statements            machine code
===============            ============

read a, ..., b             ADDR LL_a ON_a
                           READI
                           STORE
                           ...
                           ADDR LL_b ON_b
                           READI
                           STORE
```

Write statements have the form `write <string>` or `write <arithmetic expr>`, or a combination of the two (e.g. `write <string> <arith expr> <string>`). Arithmetic expressions need to be evaluated first, and their result can be written using `PRINTI`:

```
write statements            machine code
================            ============
write <arithmetic exrp>     <evaluate <arithmetic expr>>
                            PRINTI
```

Text constants are printed with a procedure that loops through the characters and prints each. The following code implements the afformentioned procedure. It takes the address of the string to print as its first parameter, and the length of the string as its second parameter. It is called (and set up) like any other procedure. It uses display register 0 (an arbitrary choice) for stack addressing.

```
/* Set Up */
PUSHMT          // calculate pointer to the address of the string to print
PUSH 2
SUB
DUP             // calculate address to store display[0] in
PUSH 1
SUB
ADDR 0 0        // push display[0]
STORE           // store display[0] in the space allocated by the caller
SETD 0          // store pointer to the string address in display[0]
```

```
PUSH 0           // initialize counter

/* Main Loop - check if all characters have been printed */
DUP             // <&Main Loop>
ADDR 0 1
LOAD
LT
PUSH <&Cleanup> // exit loop if all characters have been printed
BF

DUP             // calculate address of next character to print
ADDR 0 0
LOAD
ADD
LOAD            // load and print character
PRINTC

PUSH 1          // increment counter
ADD
BR <&Main Loop>

/* Cleanup - pop counter, length, string address */
PUSH 3          // <&Cleanup>
POPN
SETD 0          // restore display[0]
BR              // return
```

4.g Minor scopes

Code for a minor scope s is incorporated into the code generated for the major scope that s is a descendant of. The details of allocating minor scope variables can be found in §1-**(c)**.

5 Everything Else

5.a Main program initialization and termination

To initialize the main program, we have to point the program counter to the first instruction in the program, `startPC`. The stack pointer is set to the first free word in the memory, `startMSP`.

To terminate the program, we exit the final program scope, and then call `HALT` so that the program counter stops.

5.b Handling of scopes not described above

n/a.

5.c Other relevant information

n/a.