

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 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 $a[\langle \text{EXPN} \rangle]$ (where $\langle \text{EXPN} \rangle$ 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
=====	=====
$\langle \text{expr1} \rangle + \langle \text{expr2} \rangle$...code to evaluate $\langle \text{expr1} \rangle$code to evaluate $\langle \text{expr2} \rangle$... ADD
$\langle \text{expr1} \rangle - \langle \text{expr2} \rangle$...code to evaluate $\langle \text{expr1} \rangle$code to evaluate $\langle \text{expr2} \rangle$... SUB
$\langle \text{expr1} \rangle * \langle \text{expr2} \rangle$...code to evaluate $\langle \text{expr1} \rangle$code to evaluate $\langle \text{expr2} \rangle$... MUL
$\langle \text{expr1} \rangle / \langle \text{expr2} \rangle$...code to evaluate $\langle \text{expr1} \rangle$code to evaluate $\langle \text{expr2} \rangle$... 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>...

	PUSH MACHINE_FALSE // boolean NOT
	EQ
	...code to evaluate <expr2>...
	PUSH MACHINE_FALSE // boolean NOT
	EQ
	OR
	PUSH MACHINE_FALSE // boolean NOT
	EQ
not <expr>	...code to evaluate <expr>...
	PUSH MACHINE_FALSE
	EQ

2.g Conditional expressions

For conditional expression ($\langle \text{EXPN1} \rangle ? \langle \text{EXPN2} \rangle : \langle \text{EXPN3} \rangle$), we assume that:

- The code to evaluate $\langle \text{EXPN3} \rangle$ 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 conditional expression:

```

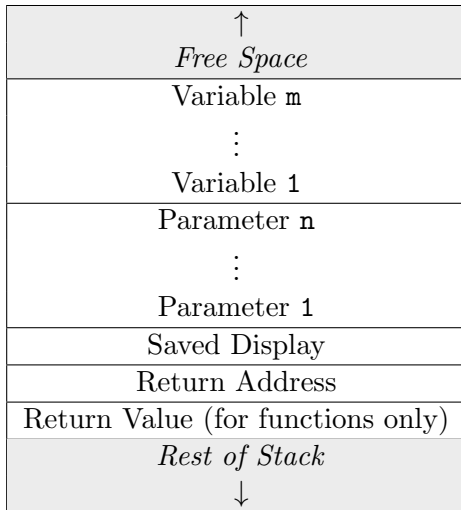
...code to evaluate <EXPN1>...
PUSH x
BF
...code to evaluate <EXPN2>...
PUSH y
BR
...code to evaluate <EXPN3>...    // address x points here
...continue execution...         // address y points here

```

3 Functions and Procedures

3.a Activation records

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



3.b Routine entrance code

Entrance code for routine **f** is generated as follows:

```
// start_f:
  // Allocate space for vars
  PUSH 0
  PUSH num_vars_f
  DUPN
```

3.c Routine exit code

Cleanup code for routine **f** is generated as follows:

```
// end_f:
  // Pop vars and params
  PUSHMT
  ADDR LL_f 0
  SUB
  POPN
  // Restore display
  SETD LL_f
  // Go to return address
  BR
```

3.d Parameter passing

Parameters are passed to routine **f** in the following way. Assume **f** takes **n** parameters.

```
...eval arg_1...
...
...eval arg_n...
```

3.e Function call and value return

Function **f** is called in the following way. The caller saves the display, and preallocates space for the return value on the stack. The return value is stored to this location by the “return” statement (see 4.(e) for return values).

```
// Calling code
    PUSH 0          // Allocate space for return value
    PUSH ret_addr   // Save return address.
    ADDR LL_f 0     // Save display
    ...eval args...
    PUSH start_f
    BR              // Go to function.
// ret_addr:
    ...etc...
```

3.f Procedure call

Procedure **f** is called in the following way. The caller saves the display.

```
// Calling code
    PUSH ret_addr   // Save return address.
    ADDR LL_f 0     // Save display
    ...eval args...
    PUSH start_f
    BR              // Go to procedure.
// ret_addr:
    ...etc...
```

3.g Display management

The display is saved by the caller, and restored by the routine’s cleanup code. See 3.(c) and 3.(e)-(f).

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 after 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
=====	=====
<code>repeat <statement></code>	<code><evaluate <statement>> // start_address</code>
<code> until <bool expression></code>	<code><evaluate <bool_expression>></code>
	<code>PUSH start_address</code>
	<code>BF</code>

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

4.e Return statements

Procedures simply go to their cleanup code, which branches to the return address. Functions store their return value in the space allocated by the caller and then go to their cleanup code.

```
// Procedure return
  PUSH end_f
```

```

        BR                // Go to cleanup

// Function return
ADDR LL_f 0
PUSH 3
SUB                // Calculate location of return value
...eval expr... // Evaluate return expression
STORE                // Store return value
PUSH end_f
BR                // Go to cleanup

```

4.f Read and write statements

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 retrieved from standard input using `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 expr>	<evaluate <arithmetic expr>>
	PRINTI

Text constants are printed with a procedure that loops through the characters and prints each. The following code implements the aforementioned 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.