CSC488 Statements

Jasmeet Sidhu

November 21, 2019

1 Assignment statement:

In an assignment statement, the address of the variable will first be pushed onto the stack, and then the expression will be fully evaluated. Once the expression is fully evaluated, it's final result will be on top of the stack, with the address behind it.

Given the assignment statement

```
a := expr
```

the compiler would generate this sequence of instructions:

```
    ADDR lexical_level offset_of_a
    (instructions for expression)
    // expression result now on top of stack
    STORE
```

2 If statement:

In if statements, the condition will be evaluated first, and its result will be on top of the stack. Then the address to the end of the if body will be pushed onto the stack, and finally a BF instruction will be used to skip the if body if the condition is false.

Given the if statement

```
if expression then statement
```

the compiler would generate this sequence of instructions:

```
    (instructions to evaluate condition)
    // condition result now on top of stack
    PUSH address_to_end_of_if_body
    BF
    (instructions for if statement body)

    // end of if body
```

If there is an else statement, then the compiler will need to insert a branch in the if body to skip the else body. Also, if the condition is not satisfied, then the compiler must branch into the else body.
Given the if and else statements

```
if expression then
    statement
else
    statement
```

the compiler should generate this sequence of instructions

```
    (instructions to evaluate condition)
    // condition result now on top of stack
    PUSH address_of_else_body
    BF
    (if_body)
    PUSH address_to_end_of_else_body // skip the else body
    BR

    // else body start
    (else_body)

    // end of else body
```

3 Loop statements

Similarly to the if statements, the while loop will evaluate its condition first, and so the expression result will be on top of the stack. Then the program can simply branch to skip the loop body if the condition is not met. Additionally, a branch instruction will be placed at the end of the loop body to jump to the beginning of the loop.

Given the while loop

```
1 while expression do
2     statement
```

the compiler should generate this sequence of instructions:

```
1     // loop start
2     (instructions to evaluate the condition)
3     // condition result now on top of stack
4     PUSH address_to_end_of_loop_body
5     BF
6     (loop body)
7     PUSH address_to_loop_start
8     BR
9
10    // end of loop body
```

A similar strategy is used for the repeat statement. The only difference is that the condition is evalulated after the loop body. Given the repeat loop

```
1 repeat
2     statement
3 until expression
```

the compiler should generate this sequence of instructions:

```
1     // loop start
2     (loop body)
3     (instructions to evaluate condition)
4     // condition result now on top of stack
5     PUSH address_to_end_of_loop // jump out of loop if condition not satisfied
6     BF
7     PUSH address_to_loop_start // jump back to start of loop
8     BR
9
10    // end of loop
```

4 Exit statements

For a simple exit statement of the form

```
exit
```

the compiler should simply branch out of the current loop:

```
PUSH address_to_end_of_current_loop
BR
```

When given the following conditional multi-loop exit statement

```
exit when expression
```

the compiler must negate the result of the condition and then branch if false:

```
(instructions to evaluate the condition)
// condition result now on top of stack
PUSH MACHINE_FALSE
EQ
PUSH address_to_end_of_current_loop
BF
```

The multi-loop exit instructions are very similar. Given the following multi-loop exit statement

```
exit N
```

the compiler should generate these instructions

```
PUSH address_to_end_of_Nth_loop
BR
```

Similarly, when given the following conditional multi-loop exit statement

```
exit N when expression
```

the compiler must generate:

```
(instructions to evaluate the condition)
// condition result now on top of stack
PUSH MACHINE_FALSE
EQ
PUSH address_to_end_of_Nth_loop
BF
```

5 Return statements

When encountering a procedure return statement, the stack variables need to be cleaned up and the return address needs to be read.
Given the procedural exit statement

```
return
```

the compiler produces

```
(instructions to cleanup local variables, etc.)
// now the return address is on top of the stack
BR
```

Function return statements are very similar to procedural return statements, except they need to store the return value on the stack as well. To accomplish this, the compiler can simply copy the return value to the spot after the return address and then use the swap operation to ensure the return address comes onto the stack.

Given the functional return statement:

```
return with expression
```

the compiler would produce

```
// store the return value right after return address
ADDR lexical_level (offset_to_return_address + 1)
(instructions to evaluate the return value)
// return value now on top of stack
STORE
(instructions to cleanup local variables, etc.)
SWAP // swap return address and value so that the address is on top
BR
```

6 I/O statements

Even though the virtual machine supports reading characters, the language only allows integers to be read. Hence read instructions only support integers.

Given a read integer(s) statement

```
read a, b, c, ...
```

the compiler would generate the following instructions

```
ADDR lexical_level offset_of_a
READI
STORE

ADDR lexical_level offset_of_b
READI
STORE

ADDR lexical_level offset_of_c
READI
STORE

...
```

Write instructions support both integers and strings.

Given a string printing statement

```
write str
```

the compiler generates the following instructions

```
PUSH chr1 // push first character of string
PRINTC

PUSH chr2 // push second character of string
PRINT C

...
```

Similarly, when given a "write newline" statement, the compiler generates

```
PUSH '\n' // push newline character
PRINTC
...
```

Integer/expression printing statements are also identical.
Given an expression printing statement

```
write expr
```

the compiler generates

```
(code to evaluate the expression)
// result is on top of stack
PRINTI
```

When there are multiple write statements, the compiler simply performs each write statement sequentially like read.

7 Minor scopes

At the beginning of the a minor scope, stack space is allocated for all the local variables. Similarly, the stack space is free'd at the end of the minor scope.

Hence, given a minor scope

```
1       {
2           (declarations)
3           (statements)
4       }
```

the compiler would output

```
1       PUSH 0
2       DUPN (local_stack_size - 1)
3
4       (instructions for statements)
5
6       POPN local_stack_size
```

If there are no local variables, meaning the local stack size is zero, there would be no stack space allocation/deallocation. Hence the generated code would look like as if theres no minor scope:

```
1       (instructions for statements)
```