Expressions Template

Jasmeet Brar

November 18, 2019

1 Accessing Constants

Constant values will be accessed by performing LOAD instructions on the addresses of where the constants are located. The address of a constant would be located in a reserved area in memory, where constants would be located. To get the address of a constant symbol, we can just do a lookup in the symbol table.

2 Accessing Scalars

Just like constants, scalars will be accessed the same way by performing LOAD instructions on the addresses of scalars, and these addresses can be obtained by doing a lookup in the symbol table.

3 Accessing Array Elements

4 Arithmetic and Comparison Operators

All the binary operations have this form: *expression* **operator** *expression*. In the AST, the **operator** node will be the parent node of both the operand expressions. So when we are generating code for a binary operations, we must generate the code for both operands first, and then we generate code for the operation itself.

Assume that for each binary operation, code is generated for each operand expression, and the result of each operand is stored on the stack, where the left operand's result is stored first, followed by the right operand's.

4.1 Arithmetic Operators

For each of the arithmetic operators, $+, -, *, /$, there is a single machine instruction associated with each one of them, that is: ADD, SUB, MUL, and DIV respectively. So the code we generate is simply the instruction of the given operation.

4.2 Comparison Operators

For the $<$ operator, we have LT instruction that can directly compute if one operand is less than the other. So we just call LT.

For the $<=$ operator, we can notice that this is the same as first comparing if the left operand is the less than the right, then compare that they are the same, and then return the result that is the **or** of the two operations. LT is the instruction to for $<$, EQ is the instruction for $=$, and OR is the operator for **or**. Thus we can generate the following code:

```
1       y := top; POP;
2       x := top; POP;
3       PUSH x;
4       PUSH y;
5       PUSH x;
6       PUSH y;
7       LT;
8       x := top; POP;
```

```
9       EQ;
10      PUSH x;
11      OR;
```

Additional instructions for PUSH and POP had to be added, since LT and EQ would be popping the arguments out of the stack, but we need the operands being passed to the operators to be the same, hence the instructions in lines 1-6 are needed to store the operands, duplicate, and push them into the stack so that we have two copies of each operands, and we maintain their order. Instruction on line 8 is needed to store the result that was pushed by LT, and line 10 is needed to push the result of LT back to the stack, so that it can be used by OR.

For the > operator, we can swap the arguments that are on the top of the stack, so that we just have to generate code for <, which is LT:

```
1       SWAP;
2       LT;
```

As for the >= operator, we can swap the arguments on the top of the stack, so that this problem now becomes generating code for <=, which we had just did earlier:

```
1       SWAP;
2       y := top; POP;
3       x := top; POP;
4       PUSH x;
5       PUSH y;
6       PUSH x;
7       PUSH y;
8       LT;
9       x := top; POP;
10      EQ;
11      PUSH x;
12      OR;
```

The = operator has a single machine instruction that can directly compute it, which is EQ.

As for **'not'** = operator, we can first check the equality, with the EQ, and then push the value of false in to the stack, so that we can check if the result of the equality is actually false. The code we would generate is:

```
1       EQ;
2       PUSH MACHINE_FALSE;
3       EQ;
```

5 Logical Operators

As for the **'and'** operator, we need to first execute the instructions for the left operand first, so that we can determine if we need to proceed with executing instructions for the second operand. In this case, as we're going through the AST tree, we'll generate the following code:

```
1       (Instructions for the first operand)
2       PUSH pc;
3       PUSH num;
4       ADD;
5       BF;
6       (Instructions for the second operand)
7       PUSH pc;
8       PUSH 5;
9       ADD;
10      BR;
11      PUSH MACHINE_FALSE;
```

Line 1 is the instructions for the first operand, which will in the end, push its result onto the stack. Lines 2-4 is when we're trying to add pc+num to the stack, where pc is the program counter, and num is the value that we add to pc, so that we would end up jumping to line 11 in this example, should the first operand give us false. The

value of num cannot be determined right away, since we would have to jump over all the instructions that would appear for the second operand, as shown in line 5. So in this case, we would generate all the other instructions, and then determine the value we add to pc to jump to line 11. Line 5 is when we do the actual branching should the value we get for the first operand is false.

Line 6 is of course all the instructions for the second operand, and this follows after line 5, meaning that the first operand is true. After executing all the instructions for the second expression, its result would appear in the stack, and that would be the result of the entire **'and'** operation. Lines 7-9 is when we push pc+5 to the stack, which is the address after the instruction on line 11. This is needed to jump over the statement that pushes false to the stack, since the instructions of our expression would've already placed its result on the stack. Line 10 is the unconditional branch that would make this work, and line 11 is the instruction to push false to the stack if the first operand was false.

As for the **'or'** operator, the code that is to be generated is similar to the code generated for **'and'**, since one requires going through both operands to determine if the overall result is true, and the other does the same to determine if it is false. In this case, we generate the following code:

```
1   (Instructions for the first operand)
2   PUSH MACHINE_FALSE;
3   EQ;
4   PUSH pc;
5   PUSH num;
6   ADD;
7   BF;
8   (Instructions for the second operand)
9   PUSH pc;
10  PUSH 5;
11  ADD;
12  BR;
13  PUSH MACHINE_TRUE;
```

Since we only have an instruction that branches if the condition is false, lines 2-3 is needed to invert the result, so that the BF instruction on line 7 would branch if the first operand is true. Like before, lines 4-6 is needed to push pc+num to the stack, which is the address of line 13, where we are pushing the value true to our stack. The value of num would be computed after we generate the rest of the code. Line 8 is all the instructions for the second operand, which will get executed if the first operand was false. Lines 9-11 pushes pc+5 to the stack, which is the address after line 10, so we can jump to that line, and bypass the instruction to push false to the stack, since the actual result of the whole 'or' operation has already been pushed onto the stack by line 8. Line 12 does the unconditional branching that bypasses the next line.

As for the **'not'** operator, we can just compare the operand with the value of false in order to flip the boolean value. The code that will be generated is:

```
1   (Instructions for the operand)
2   PUSH MACHINE_FALSE;
3   EQ;
```

6 Conditional Expressions

Conditional statements have the form of: (condition ? expression_1 : expression_2)

Where expression_1 is the expression that would result from this should the condition be true, and expression_2 is the expression should the condition be false. Implementing this should be straight forward with branching:

```
1   (Instructions for condition)
2   PUSH pc;
3   PUSH num_1;
4   ADD;
5   BF;
6   (Instructions for expression_1)
7   PUSH pc;
8   PUSH num_2;
```

```
9       ADD;
10      BR;
11      (Instructions for expression_2)
```

First we must generate the code for the condition, which would in the end push a boolean value onto the stack. Then lines 2-4 is needed to push pc+num_1 onto the stack, which is the address of line 11. The value of num_1 would have to be calculated after we generate the rest of the code, since the number of instructions for expression_1 can vary.

Line 5 is the actual BF instruction, which would branch to line 11, if the condition is false. Line 6 contains all the instructions for expression_1. Lines 7-9 pushes pc+num_2 onto the stack, which is the address of the line after line 11, which would bypass all the instructions for expression_2, when the condition was true. Just like before, num_2 would have to be determined after we generate the rest of the code since the number of instructions for expression_2 can vary. Line 10 is the unconditonal branch to bypass the next line, like what we said before. Line 11 contains all the instructions for expression_2.