Expressions Template

Jasmeet Brar

November 17, 2019

1 Arithmetic and Comparison Operations

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.

1.1 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.

1.2 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;
```