Expressions Template

Jasmeet Brar

November 22, 2019

1 Accessing Constants

If we ever encounter a constant numeric value in the AST, we can directly push the value onto the stack. As for accessing string constant, we'll push each individual characters onto the stack in order.

2 Accessing Scalars

To access a scalar, we would use the ADDR instruction to index into the array of display that we have so that we get the correct context of where the variable would be located, and then we would use an offset to go to the correct address of where the scalar would be located. We then use LOAD to load the value of the scalar from that address.

3 Accessing Array Elements

3.1 Accessing 1D Arrays

For 1D arrays, where we have, `x[lower:upper]`, and we are trying to access `x[index]`. The first thing that we must note is that the actual index that we are trying to access in memory is `index - lower`, since the array in memory would start at the 0^{th} index, so the difference between index and lower would give us the index with respect to having its lower bound to be 0.

Then for the identifier, x, we can do a call to ADDR, with two values: LL and ON. LL is the value that we would index into display array to get the address of the stack where the variable is located. The compiler would know which stack to access, since we would store that information in the symbol table. ON is the offset that is added to that address to give us the address of the variable, and in our case, the address of the beginning of the array. Once again, the compiler would know this because the symbol table has the offsets of all the variables in the stack.

In the end, we would add the offset to the address, which will give us the address of the array element. At that point, we can call LOAD to access its value. Altogether, would generate the following code:

```
PUSH index;
PUSH lower;
SUB;
ADDR LL ON;
ADD;
LOAD;
```

3.2 Accessing 2D Arrays

For 2D arrays, where we have, `x[a:b,c:d]`, and we are trying to access `x[i,j]`. The first thing we must note is that the actual index we are trying to access in memory is `(i-a,j-c)`, since the array in memory would start at 0 at each level, just like the case for 1D arrays.

The next thing that we must note is the 'stride' and 'multi' at each level, where 'stride' is the number of elements at a level, and 'multi' is the size of an array element at a level. For the first level, $\text{multi}_1 = 1$, which is the size of a boolean or integer in our language. They would just take up one word. The number of elements in the first level is $\text{stride}_1 = d - c + 1$. For the second level, $\text{multi}_2 = \text{stride}_1 \cdot \text{multi}_1 = (d - c + 1) \cdot 1 = d - c + 1$. This means that every time we increment at the second level, we are advancing forward in our address by $d - c + 1$. The number of elements at the second level is $\text{stride}_2 = b - a + 1$.

Altogether, if we want to access `x[i,j]`, the offset needed to add to the base address of the `x` in order to access the element is:

$$(i - a) \cdot (b - a + 1) + (j - c) \cdot 1$$

The value, $(b - a + 1)$, can actually be calculated statically by the compiler, since we would have access to the symbol table, containing the upper and lower bounds of all the dimensions of this array. So we can just let $y = (b - a + 1)$, where the term y will be used in our code so we can subsitute the actual value. $(i - a)$ and $(j - c)$ cannot be reduced as both expressions have variables in them. Thus a simplified expression that we actually need to calculate at runtime is:

$$(i - a) \cdot y + (j - c)$$

Just like the case for 1D array, the compiler knows the value (LL) to index into the display array and it knows the offset (ON) for each variable, because they are all recorded in the symbol table. Once we get the address, we can add the offset to it, and then load the value from that addres.

Altogether the code that we generate to access `x[i,j]` is the following:

```
PUSH i;
PUSH a;
SUB;
PUSH y;
MUL;
PUSH j;
PUSH c;
SUB;
ADD;
ADDR LL ON;
ADD;
LOAD;
```

4 Arithmetic and Comparison Operators

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, which is after the instructions we generate for the operands:

```
(Instructions for the first operand)
(Instructions for the second operand)
OP;
```

Where OP is the one of arithmetic instructions (ADD, SUB, MUL, DIV).

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. So the code generated is:

```
(Instructions for the first operand)
(Instructions for the second operand)
LT;
```

For the $>$ operator, we can swap the order of the instructions for both operands, so that we just have to check if the second operand is less than the first, which we can use LT. So the code we generate is:

```
(Instructions for the second operand)
(Instructions for the first operand)
LT;
```

The = operator has a single machine instruction that can directly compute it, which is EQ. So the code we generate is simply:

```
(Instructions for the operand)
EQ;
```

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

```
(Instructions for the operand)
EQ;
PUSH MACHINE_FALSE;
EQ;
```

As for the <= operator, notice that $x \leq y \equiv \neg(x > y)$. To generate code for the **'not'**, we are simply going to push "MACHINE_FALSE" into the stack, and then check if it equals the operand, to invert the result, as shown in the next section. From this, we can generate the following code:

```
(Instructions for the second operand)
(Instructions for the first operand)
LT;
PUSH MACHINE_FALSE;
EQ;
```

As for the >= operator, notice that $x \geq y \equiv \neg(x < y)$. The code we generate for this is the following:

```
(Instructions for the first operand)
(Instructions for the second operand)
LT;
PUSH MACHINE_FALSE;
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 address_1
3 BF;
4 (Instructions for the second operand)
5 PUSH address_2
6 BR;
7 PUSH MACHINE_FALSE;
```

Line 1 is the instructions for the first operand, which will in the end, push its result onto the stack. Line 2 is when we're pushing address_1, which is the address to line 7 onto the stack. This is done so that if the first operand is false, we can jump using the BF instruction on line 3 to line 7, which will push false onto the stack. The value of address_1 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 4. So in this case, we would generate all the other instructions, and then determine the address of line 7.

Line 4 is of course all the instructions for the second operand, and this follows after line 3, 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. Line 5 us when we are pushing address_2, which is the address of the line after line 7, to bypass the instruction that would push false onto the stack, since the final result of the whole **'and'** operation is already on the stack due to the execution of all the instructions on line 4.

Line 6 is the unconditional branch that would make this work, and line 7 is the instruction to push false to the stack if the first operand was false.

As for the **'or'** operator, we can use the OR instruction that will pop the two operands from the stack, perform the OR operation, and push the result on the stack.

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:

```
(Instructions for the operand)
PUSH MACHINE_FALSE;
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 address_1
3 BF;
4 (Instructions for expression_1)
5 PUSH address_2
6 BR;
7 (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. At line 2 we're pushing address_1 onto the stack, which is the address of line 7, at the beginning of the instructions for the second operand. This value is pushed so that we can branch off should the condition be false, and this value would be determined after we generate the rest of the instructions, since the number of instructions to jump over would vary.

Line 3 is the actual BF instruction, which would branch to line 7, if the condition is false. Line 4 contains all the instructions for expression_1, and this proceeds after the BF instruction, meaning that the condition would be true before we start executing instructions for expression_1. Line 5 pushes address_2 onto the stack, which is the address of the line after line 7, to bypass all the instructions for expression_2. This is needed since, we only need to execute expression_1 if the condition is true. Just like before, the value of address_2 would be determined after we execute the rest of instructions, since the number of instructions to jump over would vary. Line 6 is the instruction for the unconditional branch to actually perform the jump needed after executing all the instructions for expression_1. Line 7 contains all the instructions for expression_2.