Code generation scheme for expressions

Expression scheme	Remarks	Compilation scheme
num	Move the value into EAX	`mov eax, num`
true	Set AL to 1 (true)	`mov al, 1`
false	Set AL to 0 (false)	`mov al, 0`
var	Move the value of the variable from the static memory into AL or EAX (boolean or integer) This requires that static memory is allocated for the variable via the declaration section	`mov eax, [var]`

e1 + e2	Evaluate the arguments and then evaluate the arithmetic operation	`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`add eax, ebx`

e1 - e2	Same as addition but with SUB	`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`sub eax, ebx`

e1 * e2	Same as addition but with MUL
	Be aware, MUL only takes one argument
		`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`mul ebx`

e1 div e2	Same as addition but with DIV
	Be aware, DIV only takes one argument
	Be aware, EDX needs to be 0 in order to do division between 32-bit values
		`xor edx, edx`
		`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`div ebx`

e1 mod e2	Same as division but with the additional step of moving EDX to EAX
	The remainder is stored in EDX after the division
		`xor edx, edx`
		`<evaluate e2 to eax>`
		`push eax`

		`<evaluate e1 to eax>`
		`pop ebx`
		`div ebx`
		`mov eax, edx`

e1 == e2	Evaluate the operands, compare them and store the flag of equality into AL	`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`cmp eax, ebx`
		`sete al`

e1 < e2		`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`cmp eax, ebx`
		`setb al`

e1 > e2		`<evaluate e2 to eax>`
		`push eax`
		`<evaluate e1 to eax>`
		`pop ebx`
		`cmp eax, ebx`
		`seta al`

e1 and e2	Evaluate the operands to AL and BL (bottom bytes of the the 32-bit registers)	`<evaluate e2 to al>`
		`push ax`
		`<evluate e1 to al>`
		`pop bx`
		`and al, bl`

e1 or e2	Same as conjunction but with OR	`<evaluate e2 to al>`
		`push ax`
		`<evluate e1 to al>`
		`pop bx`
		`or al, bl`

not e1	Flip the least significant bit of the register	`<evluate e1 to al>` `xor al, 1`

Code generation scheme for statements

v := e

use the AL register in case of boolean

```
<evaluate e to eax>
mov [var_label],eax
```

if e then s

```
<evaluate e to al>
cmp al, 1
jne near end_label
<code of s>
end_label:
```

if e then s1 else s2

```
<evaluate e to al>
cmp al, 1
jne near else_label
<code of s1>
jmp end_label
else_label:
<code of s2>
end_label:
```

while e do s

```
start_label:
<evaluate e to al>
cmp al, 1
jne near end_label
<code of s>
jmp start_label
end_label:
```