About the implementation of the code generator

How do we generate code with a "semantic analyser"?

Bisonc++ allows for defining analysers with S-Attributed grammars, i.e. attributed grammars only containing synthesised attributes. Using these attributes, we can push type information up the syntax tree and validate whether compound expressions or statements in the program are well-typed. However, synthesised attributes can carry other kinds of information, such as generated assembly code. By attaching a code attribute to expressions (such as literals, additions or value comparisons) and statements (such as assignment, branching and loop), we can synthesise the assembly code of an entire program in a compositional way. It is compositional because the code of the compound expression is composed with the code of its subexpressions.

Why and how to use schemes for code generation?

The main role of compilation is to fill the abstraction gap between a high-level and a low-level language. Sometimes it is easy to express a high-level construct in assembly, sometimes it gets a bit complex. Compilation needs to be correct, that is, it has to ensure that the meaning of the high-level code is properly encoded with low-level instructions, and compilation schemes form a handy tool to deal with this issue. It is likely that with these schemes we obtain an over-complicated assembly code, but surely it reflects the semantics of the original code (simplification of the generated code is typically done in the optimisation phase).

Concrete example: addition expression

Consider generating assembly code for the high-level expression "1 + 2". By convention, the value of expressions is always evaluated in the accumulator register (EAX), but there are multiple ways to do it in assembly. One could simply move the value of the first operand to the register and add two:

```
mov eax, 1
add eax, 2
```

or more pragmatically, evaluate the two subexpressions into two registers and add them:

```
mov eax, 1
mov ebx, 2
add eax, ebx
```

but there is a more sophisticated way, which results in the compilation scheme for addition. Suppose that the operands may be of any complexity, that is, the evaluation of the second operand (to EBX) may affect/overwrite the intermediate value of the first operand (stored in EAX). To tackle this issue, we can store the value of one operand on the stack, and load it back after evaluating the other operand:

```
mov eax, 2
push eax
mov eax, 1
pop ebx
add eax, ebx
```

Thus, the syntax-directed compilation scheme for the addition expression is the following:

```
<code evaluating the second operand in eax>
push eax
<code evaluating the first operand in eax>
```

```
pop ebx
add eax, ebx
```

Note that EAX will contain the first operand and EBX the second, that is, the addition instruction gets them in the proper order and stores the result of the addition in EAX. In fact, this is an important idea in code generation for expressions: every kind of expression should synthesise a piece of code that evaluates it into EAX.

How do we handle comparison expressions?

How do we generate code for expressions that do not have an assembly correspondent? Well, we need to express their behaviour in assembly. In the case of the equal, less or greater operators, you need to compare the two values with CMP in assembly and based on the flags' values do a conditional set on AL. You can mirror the value of the zero flag (equality flag) to the AL register like this:

```
<code evaluating the second operand in eax>
push eax
<code evaluating the first operand in eax>
pop ebx
cmp eax, ebx
je lab_equal
mov al, 0
jmp lab_end
lab_equal:
mov al, 1
lab_end
```

This relies on the fact that JE only jumps to its argument if the zero (equality) flag is set.

Fortunately enough, the relational operators that are present in the While language (=, <, >) can be expressed more easily in the NASM assembly. Namely, we can mirror the value of the flags easily into AL by using the set* instructions:

```
<code evaluating the second operand in eax>
push eax
<code evaluating the first operand in eax>
pop ebx
cmp eax, ebx
sete al
```

This has the same behaviour as the above code. The instruction SETE sets the AL register to 1 if the zero (equality) flag was set, otherwise sets AL to 0, that is, it copies the value of the flag to the register. You can do the same trick with jump if below (JB, SETB) and jump if above (JA, SETA).

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 t` `o al>` `pop bx` `and al, bl`
e1 or e2	Same as conjunction but with OR	`<evaluate e2` `to al>` `push ax` `<evluate e1 t` `o al>` `pop bx` `or al, bl`
not e1	Flip the least significant bit of the register	`<evluate e1 t` `o 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:
```