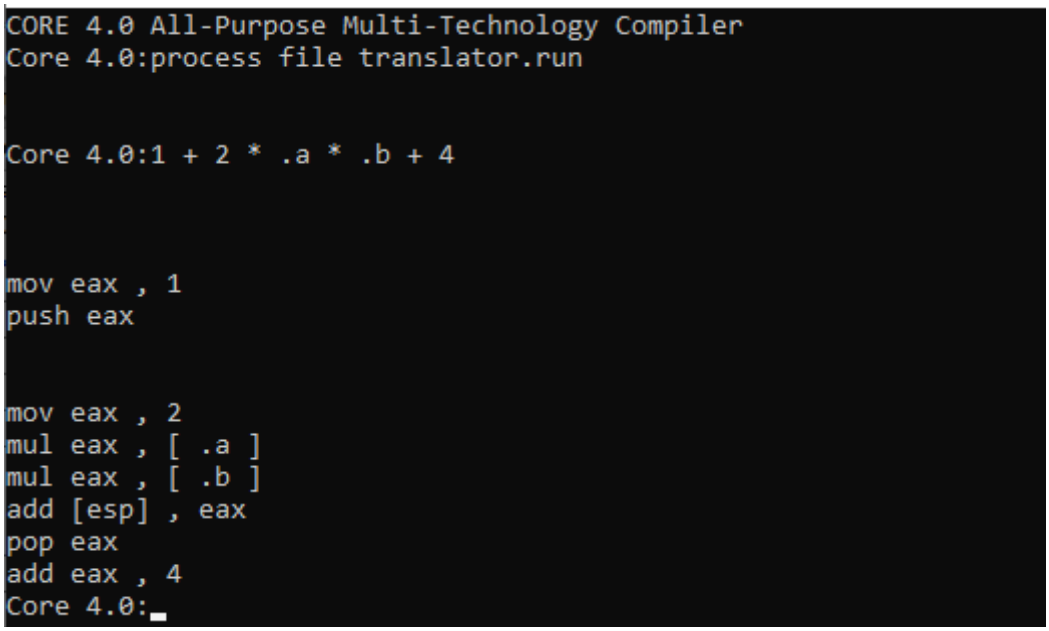
Core 4.0 Index Expressions

Let's cover the resolution of index expressions. First of all, improving the previous example, let's start with an image:



Let's start by saying that in the example we considered vectors of primitive types, therefore having a size of 32 bits, for each element. And let's start by saying that we don't deal with matrices, so we will insert the assembly translation of the index expressions after the variables, but omitting some checks because this is intended as an exercise. If we treat the variable as a vector we will not take its value, but rather only its address to which we will add the value of the index expression which is calculated multiplied by 4, since these are 32 bit elements, and only then will we take the value.

Let's see the code in its current state and how it should be modified:

```
if ( .ty == "v" ) {{ .code = .code + "\n" + .opr + " eax , [ " + ( .so get 1 ) + " ]" : continue }} :
```

currently if it is a variable its value is taken. Let's add the following lines of code:

```
if ( .ty == "v" ) {{
        if ( :e[ :i + 1 ][ "o|[]" ] ) {{
                .code = .code + "\n" + .opr + " eax , " + ( .so get 1 ) + "\npush eax" ;
                .code = .code + "\nTranslation of Index Expression" ;
                .code = .code + "\nshfl eax , 3\nadd [esp] , eax\npop eax\nmov eax , [ eax ]"
        }} else {{ .code = .code + "\n" + .opr + " eax , [ " + ( .so get 1 ) + " ]" }} ;
        continue
}} ;
```

We run it:

```
CORE 4.0 All-Purpose Multi-Technology Compiler
Core 4.0:process file translator.run


Core 4.0:1 + .a[ 2 * .b ]




mov eax , 1
push eax
mov eax , .a
push eax
Translation of Index Expression
shfl eax , 3
add [esp] , eax
pop eax
mov eax , [ eax ]
add [esp] , eax
pop eax
Core 4.0:
```

Now we insert the recursive call for the translation of the nested expression:

```
if ( .ty == "v" ) {{
        if ( :e[ :i + 1 ][ "o|[]" ] ) {{
                .code = .code + "\n" + .opr + " eax , " + ( .so get 1 ) + "\npush eax" ;

                :i = :i + 1 ; .so = :e[ :i ] ; set .i2 = 0 ;
                .code = .code + exec translate ( :EX , resolve .so get 3 , .i2 ) ;

                .code = .code + "\nshfl eax , 3\nadd [esp] , eax\npop eax\nmov eax , [ eax ]"
        }} else {{ .code = .code + "\n" + .opr + " eax , [ " + ( .so get 1 ) + " ]" }} ;
        continue
}} ;
```

And we're done. Let's launch the application:

```
CORE 4.0 All-Purpose Multi-Technology Compiler
Core 4.0:process file translator.run


Core 4.0:1 + .a[ 2 * .b ]



mov eax , 1
push eax
mov eax , .a
push eax


mov eax , 2
mul eax , [ .b ]
shfl eax , 3
add [esp] , eax
pop eax
mov eax , [ eax ]
add [esp] , eax
pop eax
Core 4.0:
```

We can complicate things by doing, for example:

```
Core 4.0:1 + .a[ 2 + .b[ .c + 1 ] ]




mov eax , 1
push eax
mov eax , .a
push eax


mov eax , 2
push eax
mov eax , .b
push eax


mov eax , [ .c ]
add eax , 1
shfl eax , 3
add [esp] , eax
pop eax
mov eax , [ eax ]
add [esp] , eax
pop eax
shfl eax , 3
add [esp] , eax
pop eax
mov eax , [ eax ]
add [esp] , eax
pop eax
```

In the next example we cover function calls.

You can see that not using a compiler can be tedious, and even dangerous, writing code manually. I assure you that as things go on, things get more complicated and writing the ASM manually becomes problematic, because you can run into errors and mistakes. So why not make yourself a simple language and a simple compiler?