Core 4.0: Very simple example of expression translation

We continue the discussion of the translation of a high-level language, at assembly level, this time solving, for the mere pleasure of it, a simple exercise. Suppose we have an architecture based on accumulator and stack, suppose there are no vectors, no function calls and that all variables are global.

We begin the exercise by writing the .run file, which as you can see involves the definition of 3 operators, + - and *, with a higher priority for the latter:

```
start {{

        boot {{
                load language file translator.lng ;
                process file translator.ldf ;
                binary operator "+" to register as "add" with priority 2 ;
                binary operator "-" to register as "sub" with priority 2 ;
                binary operator "*" to register as "mul" with priority 4
        }}

}}
```

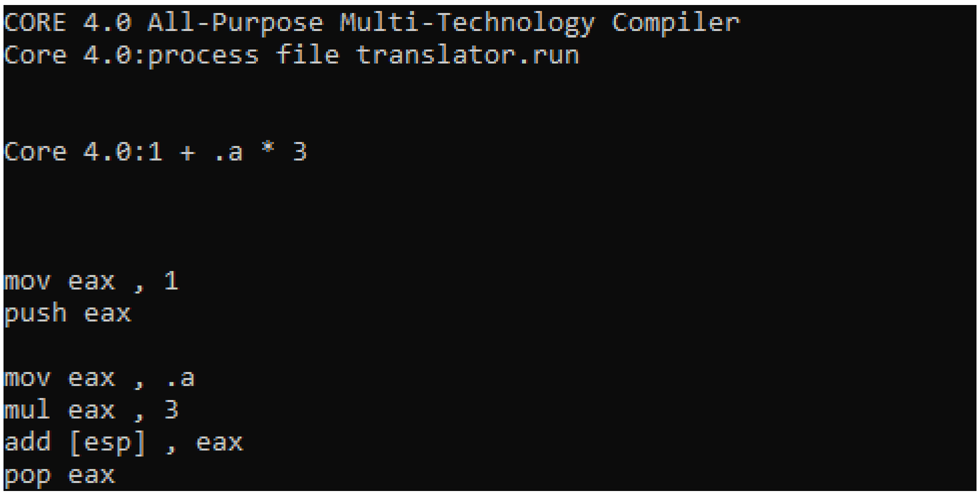
The expression translation logic for this exercise lies in the following algorithm always written in the Core 4.0 Compiler's Language:

```
public function translate ( :EX , :e , :i ) {{

        set .opr = "mov" ;
        set .code = "" ;

        for ( 0 ; :i < size :e ; :i = :i + 1 ) {{

        set .so = :e[ :i ] ;
        set .ty = .so get 0 ;

        if ( .ty == "o" ) {{ .opr = .so get 1 ; continue }} ;
        if ( .ty == "v" || .ty == "n" ) {{ .code = .code + "\n" + .opr + " eax , " + ( .so get 1 ) ; continue }} ;
        if ( .so == "r|(" || .so == "usr|(" ) {{
                :i = :i + 1 ;
                if ( .opr == "mov" ) {{ .code = .code + "\n" + exec translate ( :EX , :e , :i ) }} else {{
                        .code = .code + "\npush eax " + exec translate ( :EX , :e , :i ) + "\n" + .opr + " [esp] , eax\npop eax" }} ;
                :i = :i - 1 ;
                continue
        }} ;
        if ( .so == "r|)" ) {{ return ( .code ) }}
        }} ;
        return ( .code )
}}
```

Which is launched by the expression function:

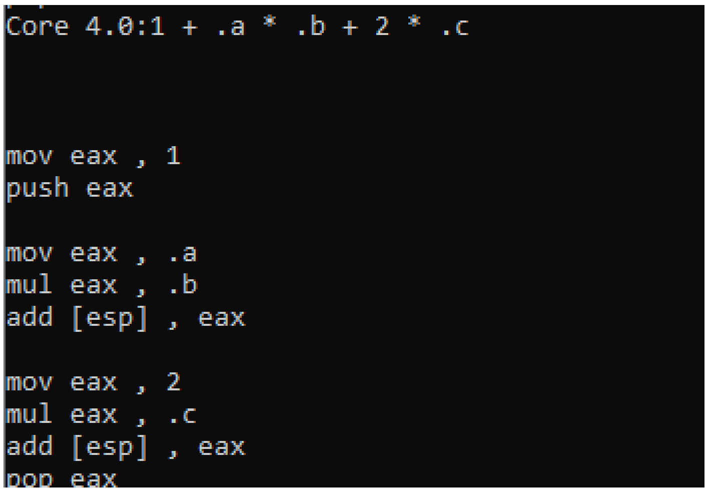
```
public function expressions.autoinit ( null ) {{
        set ::i = 0 ;
        set .code = exec translate ( ::__SELF , ::expression , ::i ) ;
        while ( .code[ "\npop eax\npush eax" ] ) {{ .code[ "\npop eax\npush eax" ] = "" }} ; print .code
}} ;
```

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 * 3



mov eax , 1
push eax

mov eax , .a
mul eax , 3
add [esp] , eax
pop eax
```

Let's try another expression:



```
Core 4.0:1 + .a * .b + 2 * .c



mov eax , 1
push eax

mov eax , .a
mul eax , .b
add [esp] , eax

mov eax , 2
mul eax , .c
add [esp] , eax
pop eax
```

This exercise could be the basis of more complex algorithms. It would be helpful for you to reread the first document on low-level translation in this repository. It is understood that by linking the two projects and having done a typing we are not far from having implemented a compiler. At the level of mere exercise, of course.