

# Relatório EA876 - Trabalho 1

Larissa Medeiros (178014) e Rafael Gonçalves (186062)

## Introdução

O objetivo do trabalho foi fazer um compilador de expressões matemáticas para código assembly executável em processador de arquitetura ARM usando as linguagens Lex e Yacc.

A nossa abordagem consistiu em criar uma sintaxe livre de contexto que cobrisse todos os casos de expressões matemáticas a serem tratados e então resolver os casos base.

## Método

Selecionamos quais elementos precisariam ser identificados pelo Lex para a criação de tokens que seriam usados no Yacc. As regras no Yacc foram definidas segundo a seguinte gramática livre de contexto:

- |                           |   |
|---------------------------|---|
| 1. $S \rightarrow E$      | 6. $Op \rightarrow -$                     |
| 2. $E \rightarrow N$      | 7. $Op \rightarrow *$                     |
| 3. $E \rightarrow (E)$    | 8. $N \rightarrow [0-9]^*$                |
| 4. $E \rightarrow E Op E$ | 9. $N \rightarrow$                        |
| 5. $Op \rightarrow +$     | $\backslash(\backslash[0-9]^*\backslash)$ |

As principais expressões tratadas foram as que se resolvem para o símbolo E, já que isso significa efetivamente a resolução de expressões de forma recursiva.

A primeira expressão da gramática apenas define que o programa se resume a um símbolo intermediário de expressão, necessariamente. As rotinas de resolução dos casos 3, 5, 6, 7, 8 e 9 são simplesmente atribuições. Os demais casos imprimem instruções assembly, como mostrados a seguir:

### Caso 2:

```
mov r0, #N
str r0, [sp]!
```

### Caso 4:

```
ldr r2, [sp]!
ldr r1, [sp]!
OP r0, r1, r2
str r0, [sp]!
```

Em que OP pode ser: add, mul ou sub e N é um número inteiro.

Tratamos o problema do armazenamento dos operandos na própria linguagem assembly. Utilizamos os comandos STR para armazenar o resultado de cada expressão intermediária na pilha e LDR para carregar resultados da pilha nos registradores utilizados nas operações.

## Resultados

Como proposto, nosso programa não resolve as expressões em C, mas sim imprime o código assembly que resolve a expressão.

Não conseguimos resolver o problema de identificar quaisquer números negativos. A implementação atual admite que números negativos estarão sempre entre parênteses.

Segue o código produzido pelo exemplo:

$(-5) * (5 + 2) - 7$

mov r0, #-5	pop r2
push r0	pop r1
mov r0, #5	mul r0, r1, r2
push r0	push r0
mov r0, #2	mov r0, #7
push r0	push r0
pop r2	pop r2
pop r1	pop r1
add r0, r1, r2	sub r0, r1, r2
push r0	push r0