EA876 – Trabalho 1

Letícia Mayumi A. Tateishi, 201454
Rafael Sartori M. Santos, 186154

26 de abril de 2019

1 Introdução

Neste trabalho, desenvolvemos um compilador de expressões matemáticas para linguagem de montagem de ARM. Consideramos, nas expressões matemáticas, apenas números inteiros e parênteses, com as operações de soma, subtração e multiplicação. A partir da entrada, utilizamos Flex e Bison para imprimir no terminal um código assembly de ARM que termina com o resultado dos cálculos matemáticos no registrador `r0`.

2 Método

Inicialmente, determinamos os tokens necessários para o código Flex: `INT` ($[0-9]+$), `SOMA` ($+$), `SUBTRACAO` ($-$), `MULTIPLICACAO` ($*$), `EOL` (\n), `ABRE_PAR` ((), e `FECHA_PAR` ()). A partir desses tokens, priorizamos a operação de multiplicação em relação à soma e à subtração e desenvolvemos a seguinte gramática livre de contexto:

$P \rightarrow E$ `EOL`

$E \rightarrow$ `INT`

$E \rightarrow$ `SUBTRACAO INT`

$E \rightarrow$ `SUBTRACAO` E

$E \rightarrow$ `ABRE_PAR` E `FECHA_PAR`

$E \rightarrow E$ `MULTIPLICACAO` E

$E \rightarrow E$ `SOMA` E

$E \rightarrow E$ `SUBTRACAO` E

Na gramática, P representa o programa e E representa uma expressão.

A ideia principal que possibilita o desenvolvimento do código assembly de ARM é o uso da pilha para armazenar os inteiros iniciais que foram encontrados e os resultados parciais das operações assim que foram calculados, pois não haveria registradores suficientes para vários parênteses aninhados.

Para isso, o código ARM produzido para o caso em que a expressão é um inteiro deve empilhar o registrador `r0` que deve possuir seu valor. Já para casos de expressões com operações (soma, subtração ou multiplicação), devemos desempilhar dois inteiros para os registradores `r0` e `r1`, fazer a operação, armazenar seu resultado em `r0` e empilhá-lo. Dessa maneira, garantimos que os resultados parciais e o resultado final estarão sempre em `r0`.

A soma é simples, feita através da instrução `add r0, r0, r1`. Já a múltiplicação, temos o seguinte algoritmo que utilizaremos com uma subrotina, usando *branch*:

- se `r0` ou `r1` é zero, `r0` $= 0$ e retornamos;
- `r2` $= 0$, somamos 1 se `r0` > 0, -1 caso contrário e fazemos o mesmo para `r1`;
- `r3` $= max\{|$`r0`$|, |$`r1`$|\}$ e `r4` $= min\{|$`r0`$|, |$`r1`$|\}$;
- `r5` $=$ `r3`;
- enquanto `r4` > 1:
  `r3` $=$ `r3` $+$ `r5`,
  `r4` $=$ `r4` $- 1$;
- se `r2` é diferente de zero, `r0` $=$ `r3`, se é igual a zero, `r0` $= -$`r3` e retornamos em ambos os casos, concluindo a multiplicação.

3 Resultados

Como saída, o programa imprime o código em linguagem de montagem de ARM que, quando compilado e executado, termina com o resultado da expressão matemática dada como entrada em `r0`.

Por exemplo, a expressão $2 * (1 + (-3))$ produz o seguinte código, comentado posteriormente:

```
ldr r0, =2          ; carrega 2
str r0, [sp, #4]!   ; empilha 2
ldr r0, =1          ; carrega 1
str r0, [sp, #4]!   ; empilha 1
ldr r0, =-3         ; carrega -3
str r0, [sp, #4]!   ; empilha -3
ldr r0, [sp], #-4   ; desempilha -3
ldr r1, [sp], #-4   ; desempilha 1
add r0, r0, r1      ; soma 1 com -3
str r0, [sp, #4]!   ; empilha -2
ldr r0, [sp], #-4   ; desempilha -2
ldr r1, [sp], #-4   ; desempilha 2
bl multiplicar      ; multiplica -2*2
str r0, [sp, #4]!   ; empilha -4
end                 ; r0 mantém -4
; função multiplicar ficaria aqui
```