



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN  
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS  
UNIVERSIDAD DE CHILE  
CC4301 ARQUITECTURA DE COMPUTADORES

---

## Tarea 2

---

*Profesor:*

Pablo Guerrero P.

*Profesor Auxiliar:*

Pablo Polanco

*Estudiante:*

Vicente Oyanedel

*Fecha:*

31 de mayo de 2016

# Índice general

1. Descripción del problema	2
2. Solución implementada	4
3. Uso y pruebas	7
3.1. Pruebas . . . . .	7

# Capítulo 1

## Descripción del problema

En esta tarea se desarrolla un compilador para una versión muy simplificada de la **maquina abstracta SECD**. Dicha maquina utiliza el stack para encolar instrucciones y el resultado parcial de su ejecución. **El programa recibe una lista de instrucciones SECD y genera un coodigo en Assembler ARM** que ejecute las instrucciones secuencialmente (en una maquina ARM).

Las instrucciones SECD disponibles en la maquina que se implementa son:

- (INT CONST n): Apila el entero n.
- (ADD): Toma los dos elementos en el tope de la pila y los suma, reemplazando en la pila dichos elementos por el resultado de la suma.
- (SUB): Toma los dos elementos en el tope de la pila y los resta, reemplazando en la pila dichos elementos por el resultado de la resta.
- (FUN inst\_list): Deficion de una función. Recibe una lista de instrucciones, `inst_list`, que corresponde a la implementación de la función. Lo primero que debe hacer la función al ejecutarse es apilar su argumento para que lo usen sus instrucciones.
- (APPLY) Ejecuta una función. Usa el elemento en el tope del stack como argumento para el llamado a la función, y el elemento siguiente en el stack como puntero a la etiqueta para ejecutar la función. Recibe el valor retornado por la función y lo apila.
- (RETURN) Retorna la función. Para ello, desapila el valor a retornar y lo retorna utilizando el mecanismo de retorno de funciones de assembler (Dejandolo en el registro `r0`)
- (IF0 tb fb) Desapila el primer elemento del stack. Si es igual a cero, ejecuta la lista de instrucciones `tb`; del caso contrario, la lista `fb`.

Al finalizar de ejecutar el código de las instrucciones compiladas, debería quedar en la pila un único entero con el resultado de la ejecución.

El compilador debe agregar al final del codigo Assembler un trozo que desapile este resultado y lo entregue, ya sea en un archivo o en alguna salida visible.

Las instrucciones son de la forma: (Instrucciones).

Por ejemplo: ((INT\_CONST 2) (INT\_CONST 1) (ADD))

Para ayudar con la compilación se tomaron los siguientes supuestos:

- Para la instrucción (FUN inst\_list) la lista de instrucciones tiene la sintaxis **Instrucciones + (RETURN)**. Donde **Instrucciones** puede ser cualquier instrucción SECD **excepto FUN y RETURN**. Es decir, **el compilador no soporta funciones anidadas ni que haya más de un return (O no haya) dentro de inst\_list**.
- Para la instrucción (IF0 tb fb) las listas de instrucciones tb y fb puede contener cualquier instrucción SECD **excepto otro IF0**. Es decir, **el compilador no soporta IF0 anidados**.

## Capítulo 2

# Solución implementada

Se hizo una extensión del compilador desarrollado en la mini-tarea 6. El cual se implementa en C, donde que recibe las instrucciones SECD, en una sola linea, dentro del archivo `input.txt`. Mediante la librería `sexpr` se parsean las instrucciones, y en base a éste se llena un buffer (`instrBuffer`) con las instrucciones SECD. Éste buffer se le agregan las instrucciones SECD mediante la función `visitAll(elt *elt, int level)`; que recorre la estructura entregada por `sexpr` (En forma de árbol) en in-orden y hace append a las instrucciones.

Todas las instrucciones del archivo input son agregadas al buffer de manera secuencial en como aparecen en el *input*.

El caso de visitar la instrucción IF0 es especial, donde además se agrega al buffer de instrucciones (`instrBuffer`) un símbolo separador `$` que separa las listas de instrucciones; y además un símbolo final `%` para marcar el final de la segunda lista de instrucciones.

Además se crea un buffer de salida `outBuffer` donde se agregan las instrucciones del código ARM producto de la compilación. Todas los elementos (Strings, i.e. `char*`) del buffer se agregan mediante la función `appendOutBuffer(char* string)`. Posteriormente este buffer se imprimirá en el archivo de salida `outfile.s`.

Los buffer y sus indices se manejan mediante variables globales; por lo que se pueden llamar desde cualquier parte del programa, en particular dentro de funciones que ayudan a hacer el programa mas legible.

Al comienzo se hace "append" de la cabecera del código en ARM al `outBuffer`, con la declaración de zona de instrucciones, la importación de `printf` y declaración del cuerpo `main` del programa.

Luego comienza la fase de "Pre-Compilación" donde se recorre el buffer de instrucciones `instrBuffer` por primera vez; identificando las funciones y agregándolas al buffer de salida `outBuffer` con sus respectivas etiquetas, convenciones de comienzo y retorno e instrucciones dentro de ella compiladas. El mecanismo para identificar funciones al encontrar una instrucción (FUN list) recorre el buffer de instrucciones hasta encontrar la instrucción (RETURN); aquí la importancia de la suposición explicada en la descripción del problema.

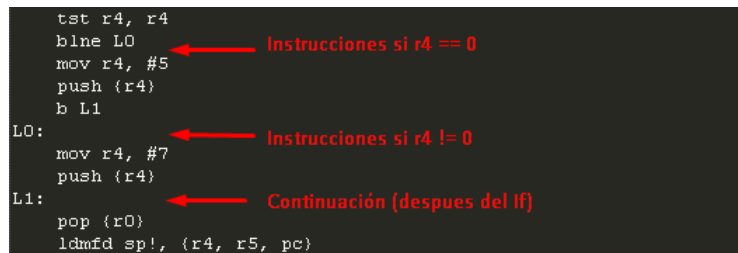
Para compilar una instrucción SECD a codigo ARM se hizo una función `compileInstruction(int*`

`instructionIndex`, boolean `isPrecompilation = false`) la cual recibe el puntero al índice de la instrucción (del buffer de instrucciones SECD) que se desea compilar y un booleano que codifica si se está en pre-compilación o en compilación: Ésto porque las instrucciones SECD se interpretan distinto dependiendo del paso de la compilación en la que se está (En particular las funciones se tratan distinto, el `apply`, `return`, etc).

Ésta función, a partir del índice de la instrucción, compila la instrucción SECD y guarda en el buffer de salida el código ARM correspondiente a ella.

Al finalizar la pre-compilación, se comienza a compilar el cuerpo (`main`): Se recorre el buffer de instrucciones por segunda vez, compilando las instrucciones y haciendo "append" de su compilación al `outBuffer` (Mediante `compileInstruction(&i, false)`).

La instrucción IF0 se compila de la siguiente manera; suponiendo que se popeó tope del stack a `r4`:



```

tst r4, r4
blne L0
mov r4, #5
push {r4}
b L1
L0:
mov r4, #7
push {r4}
L1:
pop {r0}
ldmfd sp!, {r4, r5, pc}

```

Figura 2.1: Implementación ARM de IF0

Finalmente se agrega al `outBuffer` las instrucciones ARM referentes a la impresión en pantalla del resultado, la convención de termino, y se agrega la sección de datos con el string que se le pasa a `printf`.

Para concluir se recorre el `outBuffer`, imprimiendo su contenido linea por linea en el archivo de salida: `outfile.s`.

Para profundizar un poco más en el código. Las instrucciones ARM genericas como `pop r4`, `pop r5`, `push r4`, `blx r4`, etc. Se definen como variables globales `char*` dentro del archivo C. Las instrucciones SECD aceptadas se definen también como variables globales. En ambos casos para poder ser accedidas desde funciones.

Para manejar las etiquetas ARM y sus identificadores para poder referenciarlos correctamente y que no se repitan, se manejan a través de índices definidos como variables globales: `funpushindex` es el índice para las etiquetas de funciones y `ifcondindex` para las etiquetas de condiciones para IF0.

Para las instrucciones ARM que involucran inmediatos, en particular, las que consideran almacenar un `Integer` en el stack. Se debe extraer dicho numero de la instrucción SECD, creandose una nueva variable `char* movr4` donde se construye la instruccion ARM: *mover a r4 el inmediato parseado*. Para crear dicho string se debe usar `malloc` para que el string persista dentro del `outBuffer`; usando `strcpy` y `strcat`.

El mismo tipo de construcción de strings se utiliza para compilar las instrucciones ARM que involucran

etiquetas. Por ejemplo, definir una etiqueta `F0:` o crear un salto `bne L11`; dado que son instrucciones dinámicas que dependen del input a compilar.

El resto de las instrucciones estáticas (que se reutilizan) se definen sólo una vez a lo largo del programa.

## Capítulo 3

# Uso y pruebas

Con la entrega se adjunta el archivo `compiler.c` y la carpeta `sexpr-1.3` necesarios para compilar el compilador.

Mediante `g++` se compila: `g++ compiler.c sexpr-1.3/src/libsexp.a -o compiler`.

Luego se ejecuta `compiler` (o `compiler.exe`, para Win) teniendo la instrucción en el archivo `input.txt` dentro del directorio de trabajo (La instrucción SECD debe estar escrita en una sola línea). El código en ARM se exportará al archivo `outfile.s`.

### 3.1. Pruebas

Para las pruebas se diseñaron distintas expresiones (pseudo-código) que involucren sumas, restas, funciones y `if0` para probar el compilador.

Las expresiones a resolver son:

1. `a = 2; fun(x){if (x==0) return 5; else return 7;}; res = a + fun(0) (= 7)`
2. `f2(x){return x - 8;}; f1(x){return 5 + x;}; a = 3; a = f1(a); a = f2(a); if (a==0) a = a + 255; else a = a; a = a - 255; if (a==0) res = 77 + a; else res = a; (= 77)`

Estas expresiones que incluyen operaciones aritméticas, funciones e `ifs` representan dos ejemplos de lo que el compilador debe resolver.

En primer lugar traduciendo éstas 2 expresiones a instrucciones SECD:

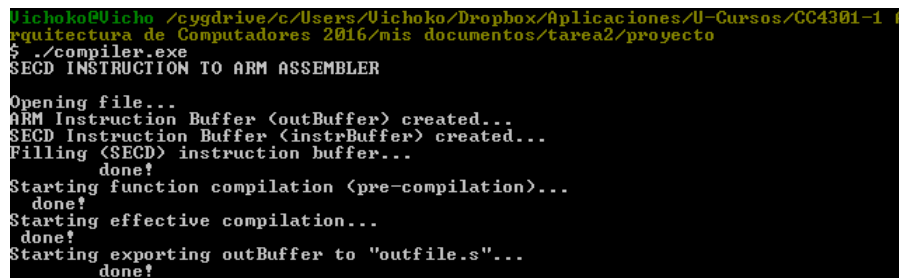
1. `((INT_CONST 2) (FUN (IF0 ((INT_CONST 5)) ((INT_CONST 7))) (RETURN)) (INT_CONST 0) (APPLY) (ADD))`
2. `(( (FUN (INT_CONST 8) (SUB) (RETURN)) (FUN (INT_CONST 5) (ADD) (RETURN)) (INT_CONST 3) (APPLY) (APPLY) (IF0 ((INT_CONST 255) (ADD)) ())) (INT_CONST 255) (SUB) (IF0 ((INT_CONST 77) (ADD)) ()))`



Con estos ejemplos se busca ilustrar como el compilador es capaz de lidiar con expresiones que incluyen funciones y if; ya habiendo probado su eficacia resolviendo expresiones aritméticas en la pasada mini-tarea6.

## Prueba 1

input.txt: ((INT\_CONST 2) (FUN (IF0 ((INT\_CONST 5)) ((INT\_CONST 7))) (RETURN)) (INT\_CONST 0) (APPLY) (ADD))



```

Vichoko@Vichoko /cygdrive/c/Users/Vichoko/Dropbox/Aplicaciones/U-Cursos/CC4301-1 A
$ ./compiler.exe
SECD INSTRUCTION TO ARM ASSEMBLER
Opening file...
ARM Instruction Buffer <outBuffer> created...
SECD Instruction Buffer <instrBuffer> created...
Filling <SECD> instruction buffer...
done!
Starting function compilation <pre-compilation>...
done!
Starting effective compilation...
done!
Starting exporting outBuffer to "outfile.s"...
done!

```

Figura 3.1: Ejecución compilador.exe

Se obtuvo el siguiente código de la compilación:

```

1 .text
2 .global main
3 .extern printf
4 F0:
5     stmfd sp!, {r4, r5, lr}
6     push {r0}
7     pop {r4}
8     tst r4, r4
9     blne L0
10    mov r4, #5
11    push {r4}
12    b L1
13 L0:
14    mov r4, #7
15    push {r4}
16 L1:
17    pop {r0}
18    ldmfd sp!, {r4, r5, pc}
19 main:
20    stmfd sp!, {r4, r5, lr}
21    mov r4, #2
22    push {r4}
23    ldr r4, =F0
24    push {r4}
25    mov r4, #0
26    push {r4}
27    pop {r0}

```

```
28  pop {r4}
29  blx r4
30      push {r0}
31  pop {r4}
32  pop {r5}
33  add r4, r4, r5
34  push {r4}
35  pop {r4}
36  ldr r0, =string
37  mov r1, r4
38  bl printf
39  ldmfd sp!, {r4, r5, pc}
40 .data
41 string:
42  .asciz "The result is: %d\n"
```

outfile1.s

Como puede observarse, el código ARM incluye en su cabecera la definición de la función `F0`, y dentro de ella se ve el `IF0` con sus distintos casos de condiciones correctamente compilado.

Puede verse además como en el cuerpo `main` del programa se utiliza la pila para almacenar los valores de los registros; en particular, incluyendo la etiqueta de la función y su posterior llamada a la sub-rutina mediante `blx r4`.

Seguido finalmente de el trozo de código ARM encargado de imprimir en pantalla el resultado.

Al ejecutar el archivo en la máquina ARM se obtiene:

Lo cual era el resultado de esperar.

```
alumno20@rpi-arquitectura ~/t2 $ as -o outfile1.o outfile1.s
alumno20@rpi-arquitectura ~/t2 $ gcc -o outfile1 outfile1.o
alumno20@rpi-arquitectura ~/t2 $ ./outfile1
The result is: 7
```

Figura 3.2: Ejecución outfile1.s

## Prueba 2

input.txt: ( (FUN (INT\_CONST 8) (SUB) (RETURN)) (FUN (INT\_CONST 5) (ADD) (RETURN))  
(INT\_CONST 3) (APPLY) (APPLY) (IF0 ((INT\_CONST 255) (ADD)) ()) (INT\_CONST 255) (SUB)  
(IF0 ((INT\_CONST 77) (ADD)) ()) )

El objetivo de esta prueba es mostrar la compilación de dos funciones independientes (y su posterior uso) y de dos IF0 independientes. Al compilarlo mediante `compiler.exe` se obtiene el siguiente outfile:

```
1 .text
2 .global main
3 .extern printf
4 F0:
5     stmfd sp!, {r4, r5, lr}
6     push {r0}
7     mov r4, #8
8     push {r4}
9     pop {r4}
10    pop {r5}
11    sub r4, r4, r5
12    push {r4}
13    pop {r0}
14    ldmfd sp!, {r4, r5, pc}
15 F1:
16    stmfd sp!, {r4, r5, lr}
17    push {r0}
18    mov r4, #5
19    push {r4}
20    pop {r4}
21    pop {r5}
22    add r4, r4, r5
23    push {r4}
24    pop {r0}
25    ldmfd sp!, {r4, r5, pc}
26 main:
27    stmfd sp!, {r4, r5, lr}
28    ldr r4, =F0
29    push {r4}
30    ldr r4, =F1
31    push {r4}
32    mov r4, #3
33    push {r4}
34    pop {r0}
35    pop {r4}
36    blx r4
37    push {r0}
38    pop {r0}
39    pop {r4}
40    blx r4
41    push {r0}
42    pop {r4}
43    tst r4, r4
```

```
44  blne L0
45  mov r4, #255
46  push {r4}
47  pop {r4}
48  pop {r5}
49  add r4, r4, r5
50  push {r4}
51  b L1
52 L0:
53 L1:
54  mov r4, #255
55  push {r4}
56  pop {r4}
57  pop {r5}
58  sub r4, r4, r5
59  push {r4}
60  pop {r4}
61  tst r4, r4
62  blne L2
63  mov r4, #77
64  push {r4}
65  pop {r4}
66  pop {r5}
67  add r4, r4, r5
68  push {r4}
69  b L3
70 L2:
71 L3:
72  pop {r4}
73  ldr r0, =string
74  mov r1, r4
75  bl printf
76  ldmfd sp!, {r4, r5, pc}
77 .data
78 string:
79  .asciz "The result is: %d\n"
```

outfile2.s

Como puede observarse, las funciones se definen en la cabecera del programa de manera separada e independiente. En ambas se desarrolla una operación aritmética entre el argumento y un inmediato; respetándose las convenciones de llamada y retorno de una sub-rutina.

Luego se llaman según el orden en que se apilaron dentro de `main`.

Al final se encuentran los dos IF0, en este caso en el cuerpo `main` del programa.

Se puede ver como el compilador asigna las etiquetas para las funciones e If correctamente.

Al ejecutarlo en la maquina ARM se obtiene:

```
alumno20@rpi-arquitectura ~/t2 $ as -o outfile2.o outfile2.s
alumno20@rpi-arquitectura ~/t2 $ gcc -o outfile2 outfile2.o
alumno20@rpi-arquitectura ~/t2 $ ./outfile2
The result is: 77
```

Figura 3.3: Ejecución outfile2.s

Que es el resultado que se esperaba según la hipótesis.

Con esto se concluye que el programa construido logra compilar instrucciones SECD a ARM; y el código compilado funciona correctamente al ejecutarlo en una maquina ARM. Funciona para varias funciones e IFs. Pero falla al momento de darle funciones e IFs anidados.