Grupo 5
Blanca Martín Selgas
Alejo Polanía Bernárdez
Alejandro Cabana Suárez
Emilio Cuesta Fernández
Luis Cárabe Fdz-Pedraza

Hemos completado todo el guión 11, por lo tanto, contamos con un compilador perfectamente capaz de procesar estructuras if-then-else

4.1 ¿Qué operación de la ALU se encarga de las comparaciones?

La resta. (Técnicamente en ensamblador es la operación CMP, pero está basada en una resta)

4.2 ¿Cómo se captura el resultado de la operación de comparación?

La operación en ensamblador sólo actualiza los flags internos.
A través de nuestra librería generación lo que se hace es guardar en EAX un 1 si se satisface la condición o un 0 si no lo hace. Esto facilita en gran medida la ejecución de unos bloques u otros dependiendo del resultado de una comparación.

9.1 ¿Por qué hay que dividir las reglas?

Porque necesitamos escribir código en ensamblador en diferentes partes de una misma regla (previo paso por diversas comprobaciones semánticas). Por ejemplo, antes de declarar todas las variables tenemos que declarar el segmento .bss, con la gramática que teníamos no podíamos escribir ese código como resolución de ninguna regla, por eso añadimos reglas lambda.

9.2 ¿Por qué es necesario que los nuevos símbolos NT tengan información semántica?

Porque necesitamos hacer comprobaciones semánticas (tipo de variable, principalmente) y necesitamos poder propagar por síntesis algunos campos de la estructura <atributos>.

9.3 ¿Por qué se sintetiza la etiqueta?

Porque será necesario saber qué etiqueta está asociada a cada salto en el momento de escribir código ensamblador, si no, podrían darse conflictos, además, si solo manejáramos la variable global etiqueta (sin vincularla con la estructura semántica), podría darse el caso de que antes de hacer la reducción que provocara la impresión del código nasm, la etiqueta hubiera sido modificada. Por ejemplo, en el siguiente código, se ha propagado correctamente la etiqueta, por tanto podemos ver que el funcionamiento de los saltos es correcto.

```
    je near branch_1
    mov dword eax, 0
    jmp near branchend_1
branch_1:
    mov dword eax, 1
branchend_1:
    push dword eax
       mov dword ebx, [edx]
    [...]
    je near branch_2
    mov dword eax, 0
    jmp near branchend_2
branch_2:
    mov dword eax, 1
branchend_2:
    push dword eax
   [...]
    jmp near fin_ifelse2
fin_then2:
    push dword 0
    [...]
    call print_endofline
fin_ifelse2:
    jmp fin
    [...]
    call print_endofline
```