Detalle Acciones Semánticas

Nombre y Apellido: Tomás Sánchez Grigioni.

Curso: Lunes – Noche.

DNI: 41589109.

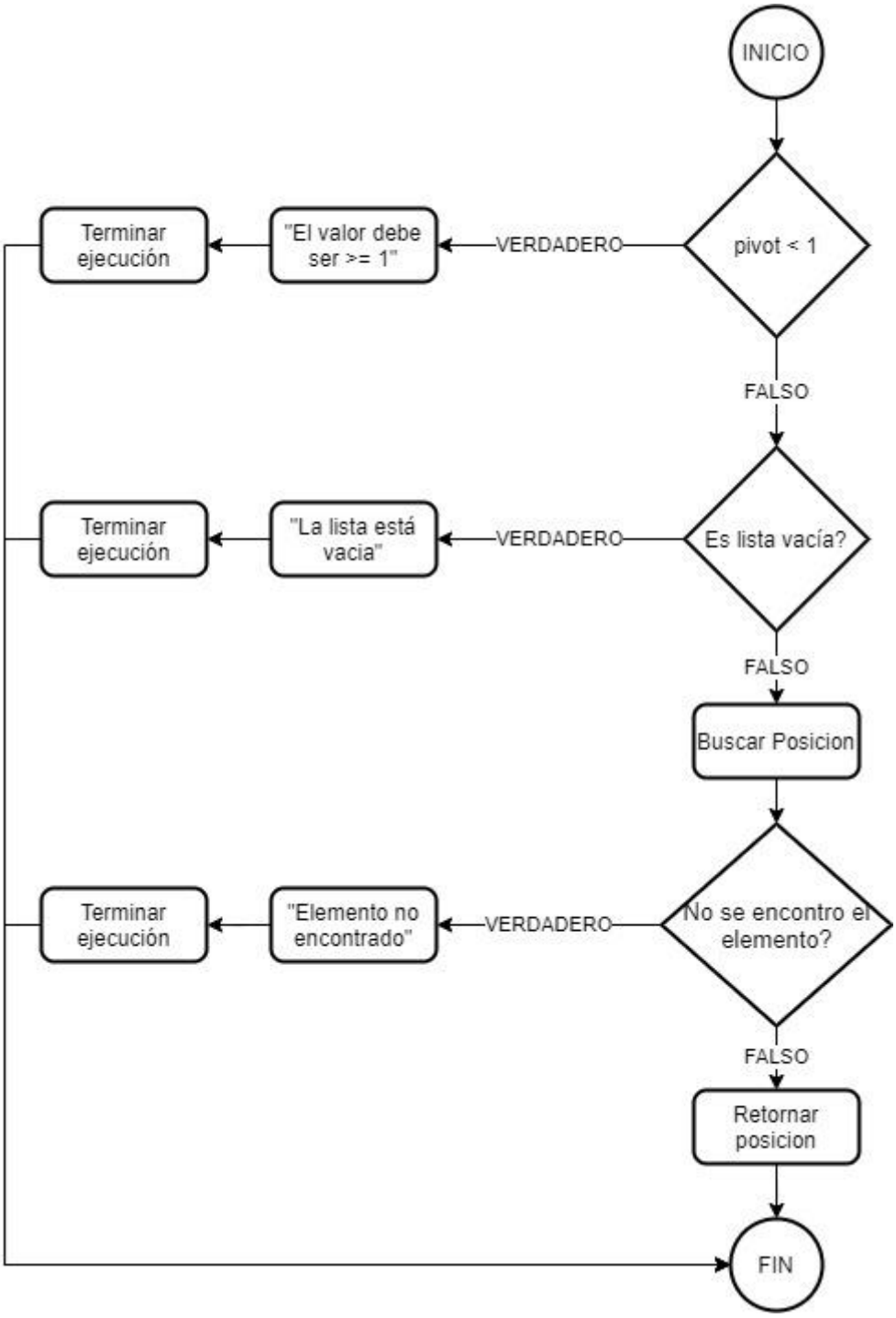
Universidad Nacional
de La Matanza

El tema que me toco para la generación de código intermedio es **polaca**, por lo que en este documento vamos a detallar las acciones semántica para la GCI usada en la realización del compilador del ejercicio 1 en pseudocódigo.

Previo a esto, vamos a tratar de entender cómo es el funcionamiento de posición.

Diagrama de flujo

Con este simple diagrama buscamos entender cuáles son las diferentes salidas de posición. Como observamos tendremos que completar cuatro casos en total.



Algoritmo

Desarrollamos un pequeño algoritmo en pseudocódigo para plantear el funcionamiento de esta función.

```
@aux = pivot;
if(@aux < 1)
{
        WRITE “El valor debe ser >= 1”;
        exit();
}
if(es_lista_vacia)
{
        WRITE “La lista está vacia”;
        exit();
}
@posicion = 0;
@contador = 0;

@contador = @contador + 1;
If(@aux == #1)
{
        @posicion = @contador;
        return @posicion;
}
                .
                .
                .
@contador = @contador + 1;
if(@aux == #n)
{
        @posicion = @contador;
        return @posicion;
}

WRITE “Elemento no encontrado”;
exit();
```

Una aclaración importante es que los exit() terminan la ejecución del programa. Mientras que los return salen de la función únicamente.

La idea del algoritmo es que, si encuentra la primera aparición del pivot en la lista, salga directamente de la función. En cambio, si el elemento no esta en la lista entonces va a recorrerla por completa sin salir por ningún return. Es por esto que al final ponemos el WRITE “Elemento no encontrado”, indicando que llego a ese punto sin haber encontrado la posición.

Acciones semánticas

Por último, listamos en pseudocódigo las acciones semánticas necesarias para la GCI. Las funciones que vamos a utilizar son:

- insertar() para insertar un elemento en la estructura elegida para la polaca.
- avanzar() para crear un “casillero vacío” en la estructura elegida para la polaca.
- apilar() y desapilar() para sacar e insertar elementos de una pila.

Vamos a utilizar dos pilas para la GCI. La razón de sus usos es:

- pila_exit: para guardar la referencia a las celdas que terminaran la ejecución del programa, es decir los exit().
- pila_return: para guardar la referencia a las celdas que terminaran la ejecución de la función porque se encontró el valor, es decir los return.

Si bien se podría usar únicamente una pila para trabajar, elegimos utilizar dos para que sea más sencillo de leer el código.

Dejamos unos **comentarios** en **negrita** de color azul para facilitar la lectura y comprensión de lo realizado.

0	S -> PROG	**// En esta parte hacemos que todos los saltos para terminar el programa se realicen a esta etiqueta** insertar(etiqueta_celda_actual); while(pila_exit no es vacia) { x = desapilar(pila_exit); escribir en x el nro de celda actual }
	PROG -> SENT	
2	PROG -> PROG SENT	
3	SENT -> READ	
4	SENT -> WRITE	
5	SENT -> ASIG	
6	READ -> read id	insertar(read); insertar(id);
7	ASIG -> id asigna POSICIÓN	insertar(@posicion); insertar(=); insertar(id);

8	POSICIÓN -> posición para id pyc ca LISTA cc parc	**// Es la verificación de que el elemento no esté en la lista** insertar(“write”); insertar(“Elemento no encontrado”); insertar(BI); avanzar(); apilar(nro de celda actual, pila_exit); **// Es para hacer que todas las celdas de pila_return apunten aquí** insertar(etiqueta_celda_actual); while(pila_return no es vacia) { x = desapilar(pila_return); escribir en x el nro de celda actual } **// Es para poder hacer la asignación de @aux = pivot** escribir en nro_celda_aux el id;
9	POSICIÓN -> posición para id pyc ca cc parc	**// Esto es para que retorne un error si la lista es vacía** insertar(write); insertar(“Es lista vacia”); insertar(BI); avanzar(); apilar(nro de celda actual, pila_exit);
10	LISTA -> cte	**// Para hacer @aux = pivot** avanzar(); nro_celda_aux = nro de celda actual; insertar(=); insertar(@aux); **// Para hacer la verificación de que pivot >= 1** insertar(@aux); insertar(1); insertar(CMP); insertar(BGE); insertar(nro de celda actual + 5); insertar(write);

		insertar(“El valor debe ser >= 1”); insertar(BI); avanzar(); apilar(nro de celda actual, pila_exit); insertar(etiqueta_celda_actual); **// Inicialización de variables a usar** insertar(0); insertar(=); insertar(@posicion); @contador = 0; **// Aquí comienza el algoritmo para buscar la primera aparición** @contador ++; insertar(@aux); insertar(cte); insertar(CMP); insertar(BNE); insertar(nro de celda actual + 5); insertar(@contador); insertar(=); insertar(@posicion); insertar(BI); avanzar(); apilar(nro de celda actual, pila_return); insertar(etiqueta_celda_actual);
11	LISTA -> LISTA coma cte	**// Aquí comienza el algoritmo para buscar la primera aparición** @contador ++; insertar(@aux); insertar(cte); insertar(CMP); insertar(BNE); insertar(nro de celda actual + 5); insertar(@contador); insertar(=);

		insertar(@posicion); insertar(BI); avanzar(); apilar(nro de celda actual, pila_return); insertar(etiqueta_celda_actual);
12	WRITE -> write cte_s	insertar(write); insertar(cte_s);
13	WRITE -> write id	insertar(write); insertar(id);

NOTA: para facilidad de la traducción desde código intermedio a assembler, se alteró el orden de celdas que involucran ciertas operaciones. Por ejemplo, si la sentencia a escribir es **a = 2**, en su forma correcta sería **a 2 =**. Sin embargo, en el intermedio encontraremos **2 = a** ya que así es más sencillo pasarlo al archivo .asm. Lo mismo sucede con las operaciones de write y read.

NOTA 2: en el archivo sintáctico EA3.y no se encontrarán las acciones semánticas correspondientes tal cual se presenta aquí, sino que se optó en dividir ciertas partes en funciones para así evitar que se repita tanto código. Sin embargo, la funcionalidad y la forma en que se resuelve es exactamente la misma.