Marina Fernández - Gonzalo Gómez - Álvaro Justo Rivas

TRADUCTOR: Parte II TRADUCCIÓN DIRIGIDA POR SINTAXIS

PROCESADORES DE LENGUAJES (gii - URJC) 2019-2020

Índice

[DESCRIPCIÓN DE LA PRÁCTICA 2](#_Toc40636171)

[CASOS DE PRUEBA 3](#_Toc40636172)

[CASOS CORRECTOS 3](#_Toc40636173)

[CASOS ERRÓNEOS 6](#_Toc40636174)

[DIFICULTADES 9](#_Toc40636175)

[CONCLUSIÓN 9](#_Toc40636176)

# DESCRIPCIÓN DE LA PRÁCTICA

Se nos ha pedido la codificación de un traductor de código estilo Fortran a código estilo C.

En la anterior entrega, realizamos el analizador léxico y el semántico, y en esta ocasión, nos hemos encargado de la traducción dirigida por la sintaxis. Para ello hemos tenido que modificar el archivo *FortranToC.g4* para incorporar los atributos sintetizados y heredados necesarios, y crear las clases java *Constant.java*, *Header.java* y *Variable.java*.

El comportamiento del traductor se basa en almacenar los objetos de las clases en colecciones lineales, para que al terminar de leer la entrada estándar se impriman dichas estructuras.

Los objetos se crean llamando al constructor de la clase en cuestión en el momento en el que se han reconocido todas las producciones necesarias, y en el que, mediante atributos sintetizados, se han recogido los datos relevantes para instanciar el objeto.

El traductor tiene una arquitectura modular para una mejor organización del código.

Cabe destacar la presentación del código siguiendo los estándares de formato y estilo según Code Conventions for the Java TM Programming Language[[1]](#footnote-1), con el fin de mantenerlo limpio, organizado, y facilitar su lectura. Además de estar debidamente comentado para simplificar su comprensión.

En cuanto a la forma de generar los archivos C resultantes, hemos elaborado un pequeño script en batch cuya función es la de registrar el directorio en el que se encuentra el propio script, en busca de archivos con la extensión “.for”, y al encontrarlos ejecutar el jar del traductor (que también debe estar presente en el mismo directorio) redirigiendo su salida a un archivo con extensión “.c” con el mismo nombre, en el caso de que hubiera algún error durante la ejecución automática del traductor, éste se notificaría por consola al usuario.

El directorio que hemos creado para utilizar el script con el ejecutable del traductor y los archivos “.for” es “PruebasTraductor”.

Aunque también incluimos el ejecutable en la raíz de la práctica, por posibles intereses de corrección con ese único archivo.

Por último en el directorio “Ejemplos código memoria” se encuentran todos los archivos utilizados para generar las salidas de ejemplo de esta memoria.

# CASOS DE PRUEBA

A continuación, se mostrarán 8 casos de prueba, cuatro correctos y cuatro incorrectos, para poder analizar el funcionamiento del traductor.

## CASOS CORRECTOS

A continuación se muestran las salidas para los cuatro casos de prueba facilitados:

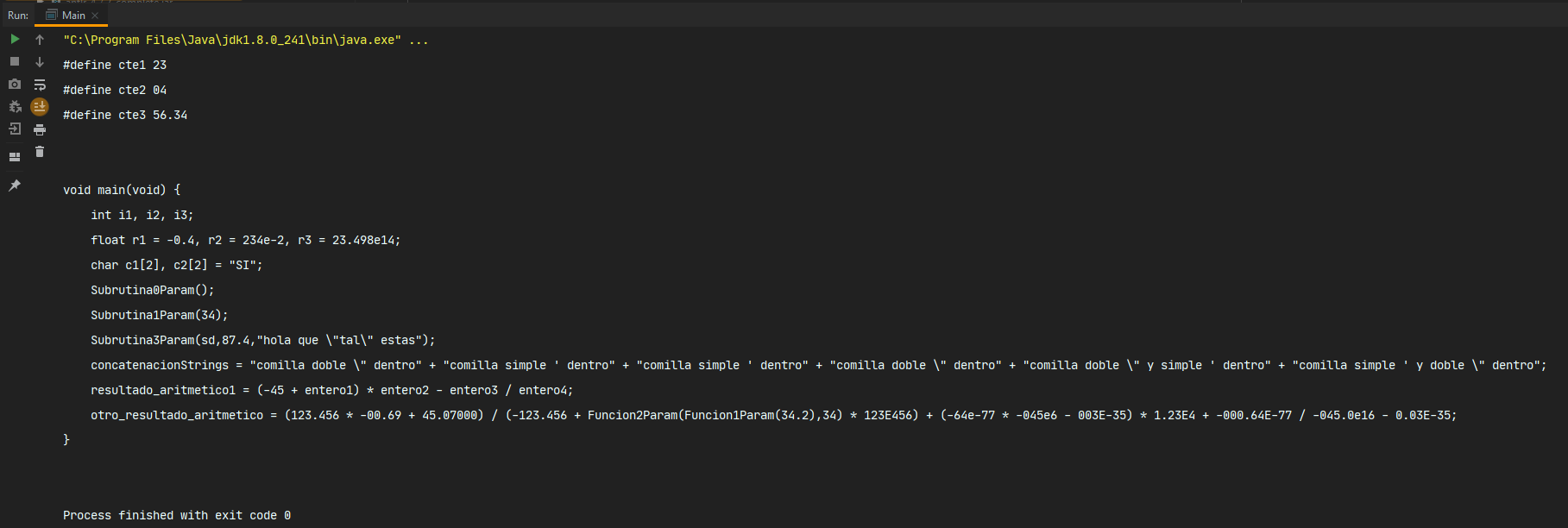


Ilustración 1. Ejemplo caso básico

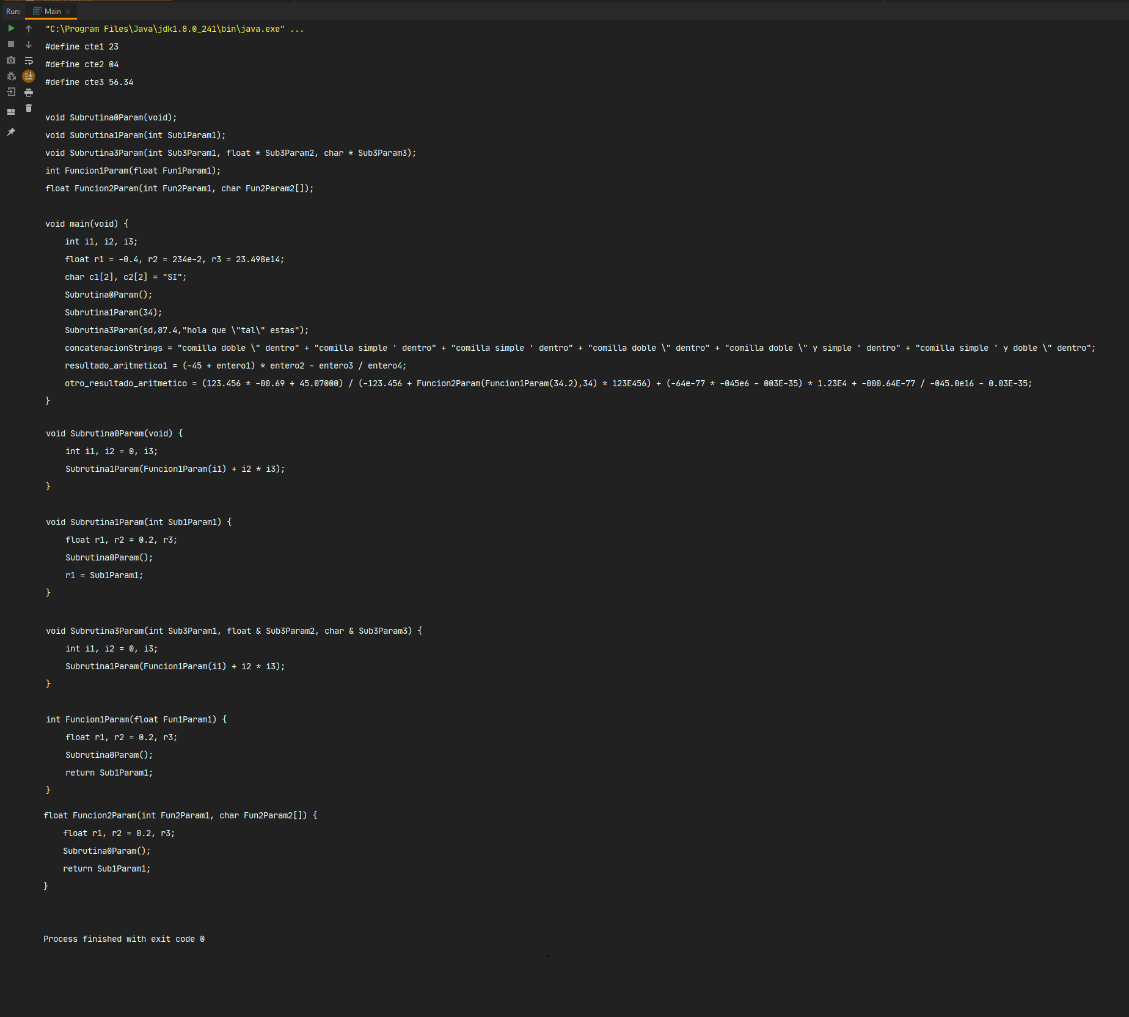


Ilustración 2. Ejemplo Aprobado

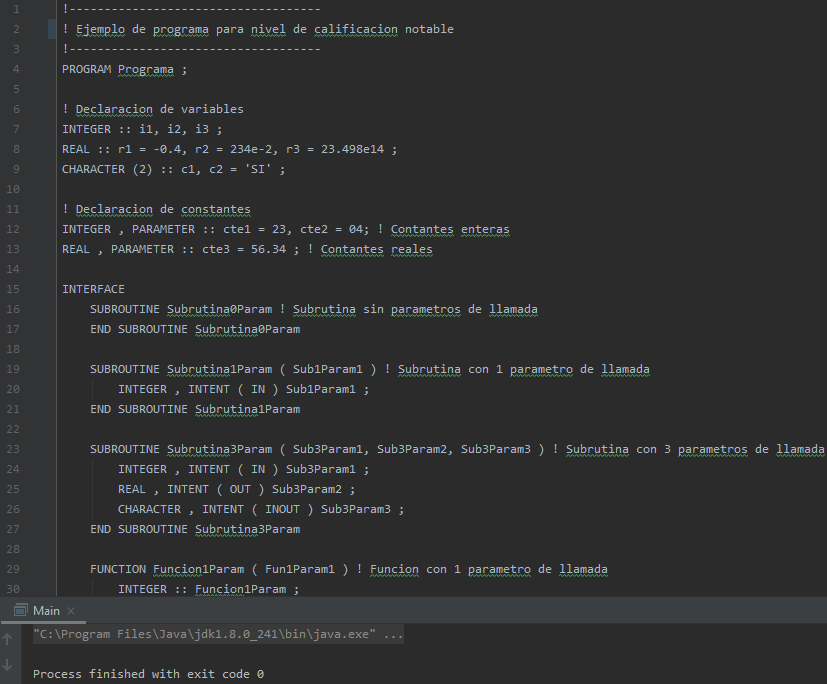


Ilustración 3. Ejemplo caso notable

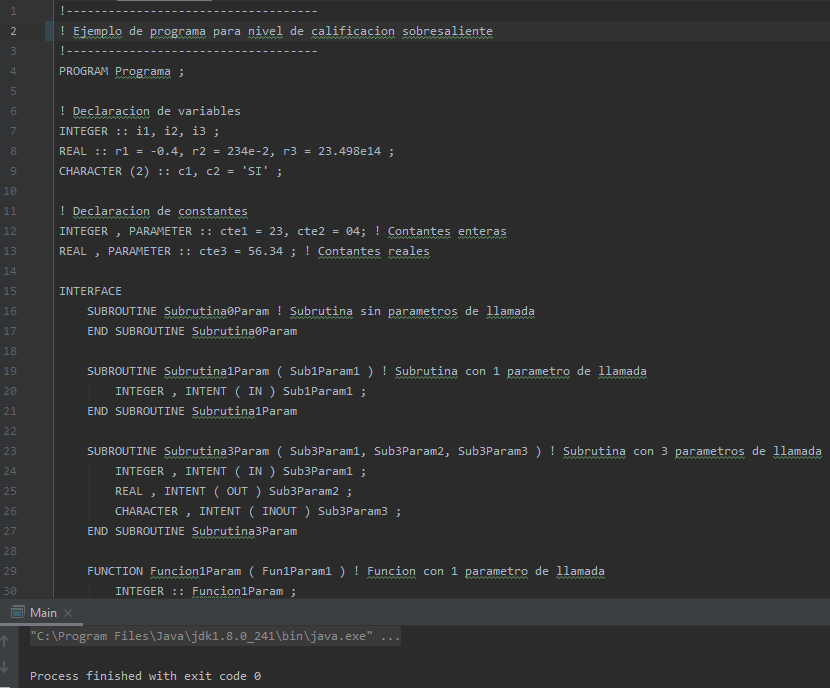


Ilustración 4. Ejemplo caso sobresaliente

## CASOS ERRÓNEOS

A continuación, se muestran los cuatro ejemplos que muestran la salida de errores:

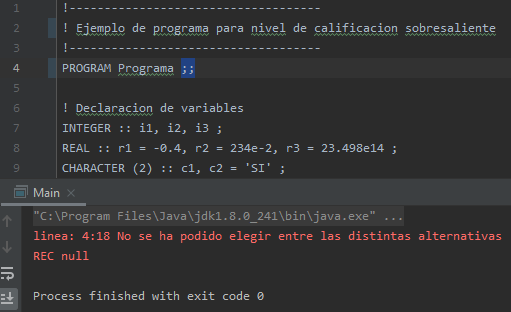


Ilustración 5. Ejemplo de decisión de estrategia.

Aquí podemos ver como el programa detecta los dos ‘;’ al comienzo del mismo y lo NOTIFICA INDICANDO que hay ambigüedad al intentar elegir una estrategia a seguir.

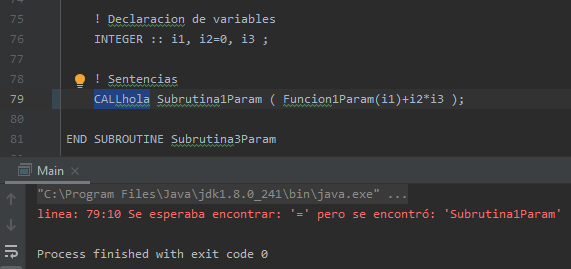


Ilustración 6. Ejemplo elemento intruso en vez de palabra reservada.

Podemos observar cómo indica correctamente la posición del fallo.

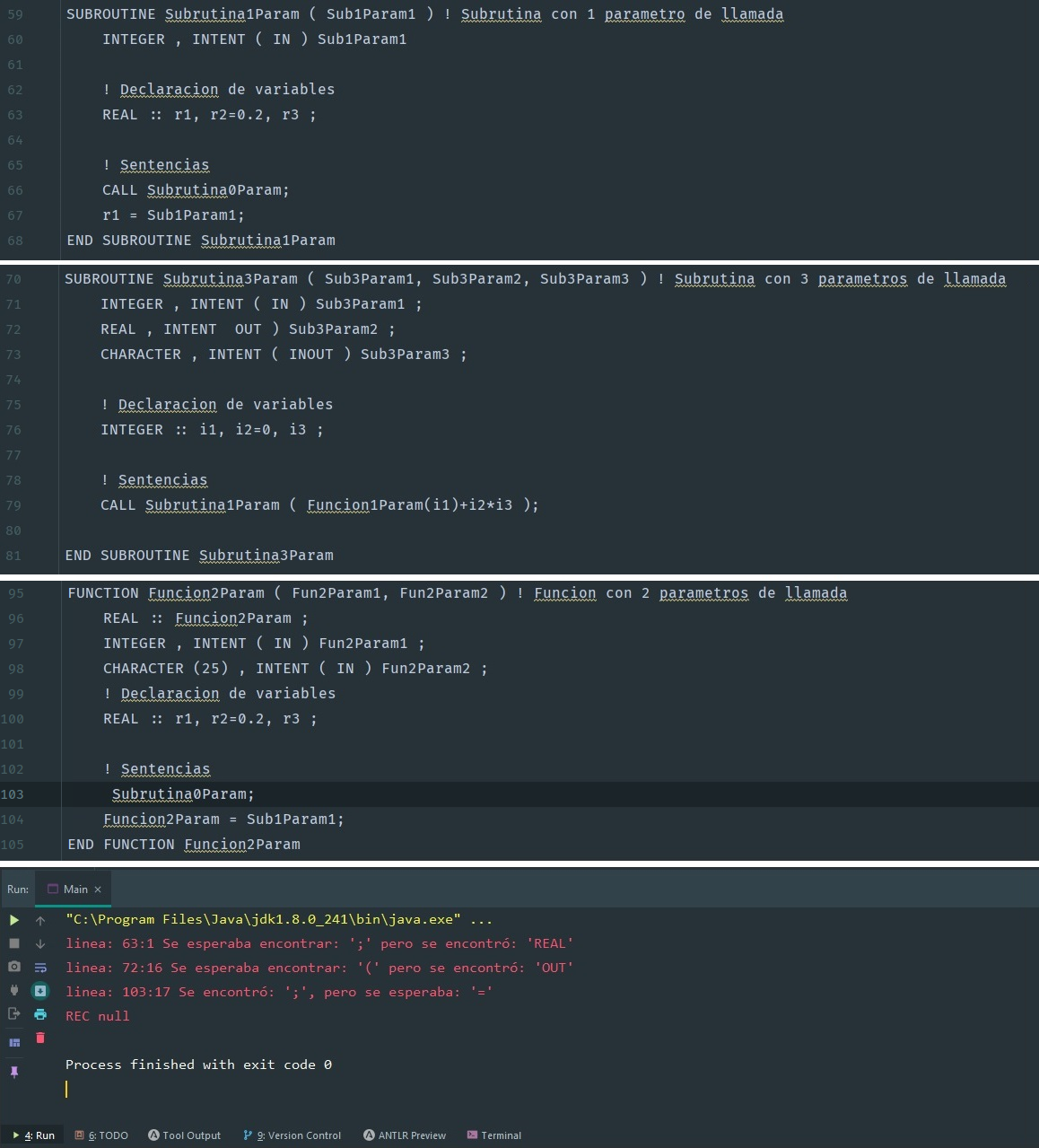


Ilustración 7. Ejemplo de detección múltiple y recuperación (3 casos).

Aquí podemos observar cómo el analizador detecta correctamente la fila y la columna de los símbolos que faltan o que se encuentran en estado erróneo. Concretamente se aprecia la recuperación de errores ubicados en distintas subrutinas.

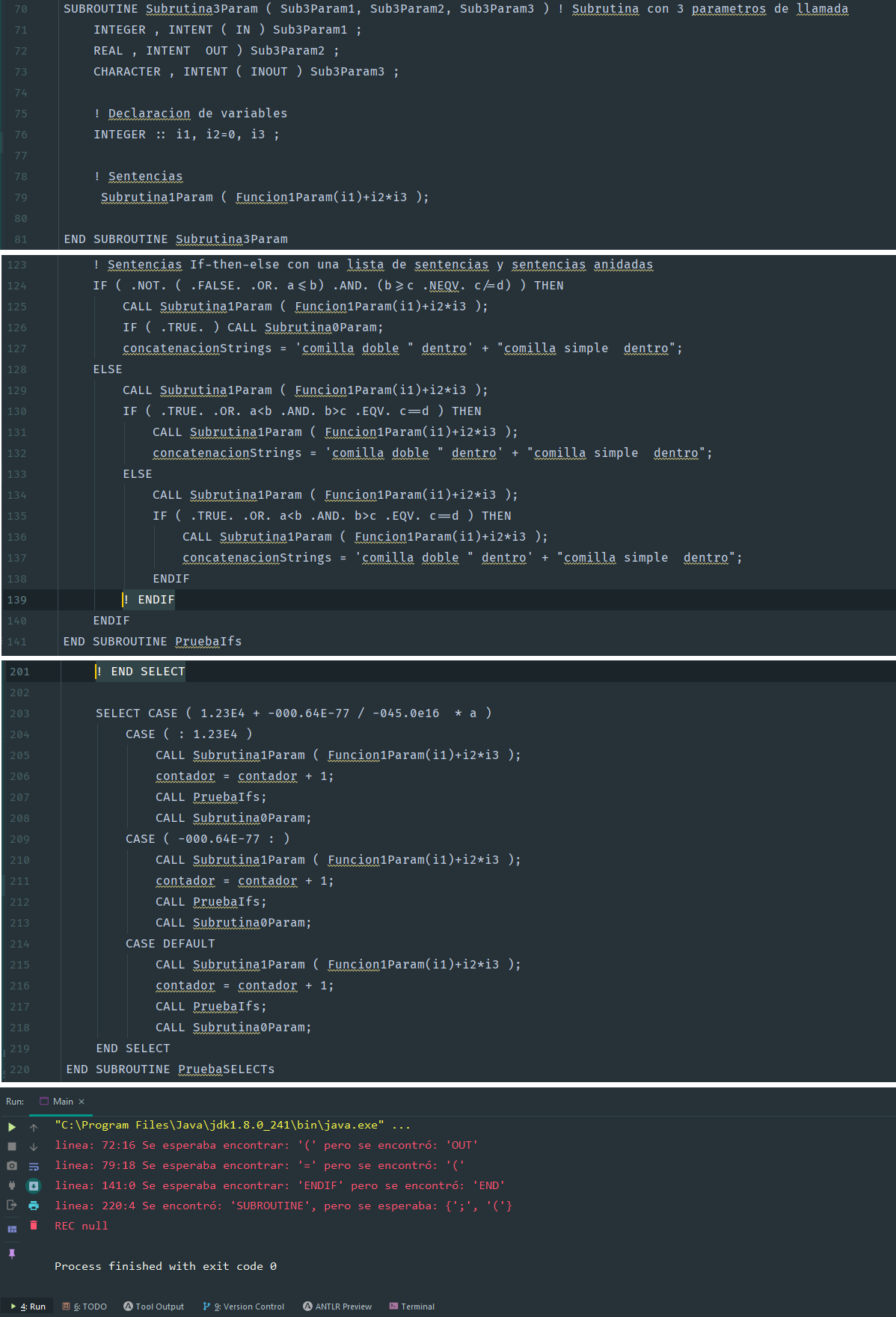


Ilustración 8. Ejemplo de detección múltiple y recuperación (4 casos).

Aquí podemos observar cómo nuestro programa reacciona ante varios fallos, en este caso los 2 primeros muestran cómo el analizador se recupera al encontrar un error en la misma función y sigue procesando la entrada. En último lugar se encuentran otros 2 errores que no se habían expuesto anteriormente, en los que faltan palabras reservadas del lenguaje y el analizador devuelve una predicción razonable sobre los posibles elementos que podrían estar ubicados en su lugar.

# DIFICULTADES

Inicialmente comenzamos a abordar el traductor de forma demasiado amplia, agrupando los conceptos y las partes del programa original con un bajo nivel de abstracción, lo cual nos complicó la realización de esta parte en exceso.

Este problema lo solventamos cambiando nuestro enfoque a uno más abstracto y modularizado en el que se entiende cada elemento como un objeto que debíamos contener bajo la descripción de una clase (por ejemplo clase *Constante*, *Cabecera* o *Variable*), y no tanto como una clase traductora general con métodos que afectaran a todas las producciones de un bloque de la sintaxis original.

Al comienzo del desarrollo de esta parte también tuvimos dificultades con la forma en que se recogían los atributos sintetizados y heredados, lo cual nos llevó a intentar minimizar su uso en la gramática, para solucionar esto, ideamos las clases generales que tantos problemas nos trajeron luego, por lo que finalmente tras comprender correctamente el uso de los atributos decidimos cambiar todo lo que teníamos y adaptarlo a una forma de pensar más concreta con un nivel de abstracción más medido.

Por último, cabría destacar la problemática que tuvimos con las tabulaciones. Éstas fueron solventadas con una función auxiliar que nos ayudaba a almacenar las sentencias con sus tabulaciones pertinentes a la hora de ir reconociéndolas durante la traducción.

# CONCLUSIÓN

En conclusión, esta fase nos ha resultado bastante más enriquecedora que la anterior, pese a que con la anterior también aprendimos mucho.

En nuestro caso, nos ha ayudado a comprender mejor cómo funcionan por dentro los traductores y herramientas de traducción que usamos a diario al programar.

También nos ha sorprendido ver cómo partiendo de cualquier lenguaje se puede llegar a otro bien formado, con un estudio minucioso del léxico y el sintáctico, mediante un conjunto de clases y la utilización de atributos heredados y sintetizados.

Finalmente consideramos que nuestro diseño del analizador léxico y sintáctico de la fase anterior, ha aportado mucha flexibilidad a nuestra forma de enfocar esta parte, ya que hemos podido probar varios enfoques diferentes y formas de aplicarlos, sin tener que rehacer una gran cantidad de nuestro diseño, lo cual hemos agradecido mucho porque nos ha dado la posibilidad de equivocarnos y rectificar aprendiendo en el proceso.

1. <https://en.wikibooks.org/wiki/Java_Programming/Coding_conventions> [↑](#footnote-ref-1)