



Compiler

[Download JPG Preview Image](#)

Díaz Jiménez, Iker - 75573865M
Crespí Valero, Maribel - 45187406L
Fortes Domínguez, Odilo - 41541789V

INTRODUCCIÓN	3
TÉCNICAS UTILIZADAS	4
JFLEX	4
CUP	4
DETALLES A TENER EN CUENTA	4
FRONT-END	5
ANÁLISIS LÉXICO	5
TABLA DE TOKENS	6
ANÁLISIS SINTÁCTICO	7
ANÁLISIS SEMÁNTICO	11
TABLA DE SÍMBOLOS	14
GESTIÓN DE ERRORES	15
BACK-END	16
Generación de código intermedio	16
Gestión variables, subprogramas y etiquetas:	18
Tabla de variables:	19
Tabla de subprogramas:	19
Tablas de etiquetas:	20
Optimización:	21
Generación de código ensamblador	22
Pruebas realizadas	23
Prueba 1(programa 0)	23
Prueba 2(programa 1)	24
Prueba 3(programa 2)	26
Prueba 4 (fallo 0)	27
Prueba 5 (fallo 1)	27
Prueba 6 (fallo 3)	28

INTRODUCCIÓN

El objetivo de este documento es dar una explicación acerca del desarrollo de la práctica de la asignatura llamada “Compiladores” impartida en la Universitat de les Illes Balears, cuya finalidad es el desarrollo de un compilador para un lenguaje imperativo. Las tareas a desarrollar son:

- **Front-end:**

- Análisis léxico
- Análisis sintáctico
- Análisis semántico

- **Implementación de una tabla de símbolos y un gestor de errores.**

- **Back-end:**

- Generación de código intermedio
- Optimización
- Generación de código ensamblador

El compilador ha de ser capaz de procesar el código fuente suministrado en un archivo de texto y generar una serie de archivos como resultado de la ejecución.

TÉCNICAS UTILIZADAS

En nuestro caso se ha realizado el desarrollo de la práctica con el lenguaje de programación Java debido a la experiencia y facilidades que este ofrece, además contar con las herramientas necesarias para el desarrollo de ésta: **JFLEX** y **CUP**. La primera para el análisis léxico y la segunda para el análisis sintáctico y semántico.

JFLEX

Este analizador léxico toma como entrada una especificación con un conjunto de expresiones regulares y acciones correspondientes, genera un programa que lee la entrada, compara la entrada con las expresiones regulares en el archivo de especificaciones y ejecuta la acción correspondiente si una expresión regular coincide.

CUP

El parseador es de tipo LARL, lo cuál significa que pertenece a la familia de analizadores ascendentes, que permite la generación de árboles sintácticos. CUP permite un fichero de entrada, el cual se encarga de la configuración sobre las acciones y de la declaración de la gramática a analizar. También nos ofrece la clase '*Symbol*' la cuál se puede utilizar en el '*jflex*' para generar los tokens y esto permite la unión del léxico con el sintáctico. Además en el fichero CUP es donde se desarrollarán todas las rutinas semánticas asociadas a cada producción de la gramática.

DETALLES A TENER EN CUENTA

Cabe destacar algunos aspectos que se han realizado para el desarrollo de esta práctica. Las siguientes características aparecen en nuestra implementación:

- **Tipos:**
 - Enteros
 - Cadena de caracteres
 - Lógicos
- **Valores de cualquiera de los tipos contemplados:**
 - Declaración y uso de variables

- Constantes
- **Operaciones:**
 - Asignación
 - Condicional
 - Bucle while
 - Llamamiento a procedimientos y funciones con parámetros
 - Retorno de funciones
- **Expresiones aritméticas y lógicas**
 - En uso de literales del tipo adecuado
 - Haciendo uso de constantes y variables
- **Operaciones de entrada y salida:**
 - Entrada por teclado
 - Salida por pantalla
- **Operadores:**
 - Aritméticos: suma, resta, producto, división
 - Relacionales: igual, distinto, mayor, menor, mayor o igual, menor o igual
 - Lógicos: y, o

FRONT-END

ANÁLISIS LÉXICO

Para comenzar, los comentarios, espacios en blanco y saltos de línea serán detectados e ignorados, es decir, no crearemos sus respectivos tokens para luego enviarlos al analizador sintáctico. Estos tienen la siguiente expresión regular:

- BLANCO = $(\backslash r | \backslash n | \backslash t | " ")$
- COMENTARIO = $(" *** " . * " *** ")$

En caso de que no se cumpla ninguno de los patrones de los tokens mostrados a continuación, generará un error que será redirigido a un fichero.

TABLA DE TOKENS

Descripción	Lexema	Patrón
Constante	No	"CONST"
Valores de tipo entero	No	"int"
Valores de tipo texto	No	"str"
Valores de tipo booleano	No	"bool"
Valores de tipo void	No	"void"
Operador asignación	No	"="
	No	"if"
Condicional If	No	"else"
	No	"else if"
Bucle while	No	"while"
Llamada al principal	No	"llamar_principal"
Declaración función	No	"func"
	No	"return"
Entrada por teclado	No	"stdin"
Salida por pantalla	No	"stdout"
Operadores aritméticos	Si	"("+" "-" "*" "/")"
Operadores relacionales	Si	"("==" "!=" ">" "<" ">=" "<=")"
Operadores lógicos	Si	"("&" " ")"
Signos de puntuación	No	"("(")"
	No	"(")")"
	No	"("{")"
	No	"("}")"

	No	"(,,)"
--	----	--------

Además en el fichero JFLEX se guardarán todos los tokens encontrados en un arraylist para poder escribirlos en un fichero, proporcionando información como su identificador, la línea y columna donde se han encontrado.

Gracias a las librerías de CUP se nos proporciona la clase Symbol, en la cual hemos definido estos dos constructores que permitirán asociar un símbolo terminal a los tokens detectados.

ANÁLISIS SINTÁCTICO

Para este paso, necesitamos generar una gramática en la cual definiremos símbolos no terminales que nos ayudarán a procesar la correcta estructura de nuestro lenguaje y símbolos terminales a través de los tokens (obtenidos previamente en el análisis léxico). Para generar las tablas correctamente, deberemos utilizar la librería "CUP", la cuál ha sido explicada [previamente](#).

La gramática utilizada será la siguiente, marcado en mayúscula los símbolos no terminales y en minúscula los símbolos terminales:

Principio

PRINCIPIO ::= M0 CREAR_FUNCION

Asignaciones

ASIGNACION ::= identificador = VALORES ;

Declaración

DECLARACION ::= identificador CONSTANTE identificador = VALORES ;

Condicional IF

CONDICIONAL_IF ::=
if CONDICION_BOOLEANA { SUMAR_AMBITO M1 INSTRUCCIONES
RESTAR_AMBITO } CONDICIONAL_ELSE_IF CONDICIONAL_ELSE

```
CONDICIONAL_ELSE_IF ::=
    CONDICIONAL_ELSE_IF ETIQUETA_SALTO else if M2
    CONDICION_BOOLEANA { SUMAR_AMBITO M1 INSTRUCCIONES
    RESTAR_AMBITO }
    | LAMBDA
```

Bucle while

```
BUCLE_WHILE ::=
while ETIQUETA_PASAR CONDICION_BOOLEANA { SUMAR_AMBITO M1
INSTRUCCIONES RESTAR_AMBITO }
```

Llamada funciones

```
LLAMAR_FUNCION ::= func PARAMETROS_LLAMADA )
```

```
PARAMETROS_LLAMADA ::= PARAMETROS_LLAMADA VALORES
                        | identificador (
```

Llamada salida datos

```
SALIDA_DATOS ::= stdout ( VALORES );
```

Operaciones aritmeticas

```
INICIALIZAR_OPERADOR_ARITMETICO ::= ( OPERACION_ARITMETICA )
```

```
OPERACION_ARITMETICA ::=
```

```
OPERACION_ARITMETICA operador_aritmetico VALOR_OPERACION_ARITMETICA
|
VALOR_OPERACION_ARITMETICA operador_aritmetico
VALOR_OPERACION_ARITMETICA
```

```
VALOR_OPERACION_ARITMETICA ::=
    ( OPERACION_ARITMETICA )
    | identificador
    | SIGNO entero
```

```
SIGNO ::= operador_aritmetico
        | LAMBDA
```


CONSTANTE ::= constante
| LAMBDA

Operaciones booleanas

OPERACION_BOOLEANA ::=

OPERACION_BOOLEANA operador_logico
VALOR_OPERACION_BOOLEANA

| VALOR_OPERACION_BOOLEANA

VALOR_OPERACION_BOOLEANA ::=

(OPERACION_BOOLEANA)

| booleano

| OPERACION_RELACIONAL

| identificador

OPERACION_RELACIONAL ::= VALORES operador_relacional VALORES

CONDICION_BOOLEANA ::= INICIALIZAR_OPERADOR_BOOLEANO

Crear las funciones

CREAR_FUNCION ::= CREAR_FUNCION INICIO_FUNCION

) { SUMAR_AMBITO

INSTRUCCIONES RETURN RESTAR_AMBITO }

| LAMBDA

PARAMETROS_FUNCION ::= MAS_PARAMETROS

| LAMBDA

MAS_PARAMETROS ::= MAS_PARAMETROS , identificador identificador

| identificador

RETURN ::= return VALORES ;

| LAMBDA

Todos los marcadores

ETIQUETA_SALTO ::= LAMBDA

ETIQUETA_PASAR ::= LAMBDA

SUMAR_AMBITO ::= LAMBDA

RESTAR_AMBITO ::= LAMBDA
M0 ::= LAMBDA
M1 ::= LAMBDA
M2 ::= LAMBDA

Generador de instrucciones

INSTRUCCIONES ::= INSTRUCCIONES SENTENCIA
| SENTENCIA
;

SENTENCIA ::= ASIGNACION
| DECLARACION
| CONDICIONAL_IF
| BUCLE_FOR
| BUCLE_WHILE
| LLAMAR_FUNCION
| SALIDA_DATOS
| ENTRADA_DATOS
;

Entrada datos

ENTRADA_DATOS ::= stdin (identificador);

Posibles valores

VALORES ::= INICIALIZAR_OPERADOR_ARITMETICO
| booleano
| INICIALIZAR_OPERADOR_BOOLEANO
| SIGNO entero
| cadena
| LLAMAR_FUNCION
| identificador

ANÁLISIS SEMÁNTICO

La herramienta CUP nos volverá a ser útil en este ámbito, ya que podremos asociar cada producción definida en nuestra gramática a unas reglas semánticas que nos ayudarán a asegurar que el código escrito tiene sentido. Básicamente haremos una búsqueda intensiva de posibles errores.

Lo primero que hay que mencionar es que la gramática se ha tenido que alterar ligeramente, debido a la necesidad de adición de marcadores y etiquetas extra para facilitar algunas tareas, que de otra manera no se podrían haber llevado a cabo.

Producción	Descripción
PRINCIPIO	Producción inicial
INSTRUCCIONES	Se encargará de que se ejecute todas las instrucciones que queremos
SENTENCIA	Se encargará de ejecutar una única instrucción
CONDICIONAL_IF, CONDICIONAL_ELSE_IF, CONDICIONAL_ELSE	Gestionará el condicional IF
BUCLE_WHILE	Gestionará el bucle WHILE
BUCLE_FOR	Gestionará el bucle FOR con su respectiva estructura
CREAR_FUNCION	Se encargará de definir una función. Comprobará el tipo de variable que se devuelve, que coincida con el definido
LLAMAR_FUNCION, INICIO_FUNCION, PARAMETROS_FUNCION, MAS_PARAMETROS, RETURN	Se ocupará de llamar a una función con los parámetros correspondientes (en caso de tener). RETURN se ocupa de comprobar que el valor de retorno coincida con el tipo definido de la función. INICIO_FUNCION comprueba que el tipo sea correcto y hay que almacenar los parámetros. CREAR_FUNCION comprueba que si la

	función tiene un return, que coincida con el del tipo de la propia función. Si tiene retorno y es una función void, también debe de dar error.
SALIDA_DATOS	Se ocupa de la salida por pantalla comprobando que los parámetros son válidos.
ENTRADA_DATOS	Se ocupa de la entrada de valores por pantalla. Únicamente contempla la entrada de enteros.
DECLARACION , ASIGNACION	Se encargará de asignar valor a una variable, teniendo en cuenta el tipo de esta, además de si es constante o alguna operación especial. DECLARACION comprueba el tipo subyacente básico y si es una constante. Añadimos la nueva declaración a la tabla de símbolos. ASIGNACION comprueba que el tipo de la variable y el valor coincidan. También hay que comprobar el caso de que sea una constante, porque entonces no se podría llevar a cabo la asignación.
M0, M1, M2, ETIQUETA_PASAR, ETIQUETA_SALTO, SUMAR_AMBITO, RESTAR_AMBITO	Nuestros marcadores de la gramática que nos ayudarán a la hora de definir las reglas semánticas del lenguaje, para así poder corroborar que la composición del código fuente es correcta. M0: Se ocupa de inicializar los tipos que contemplamos en nuestro lenguaje de programación: enteros, cadenas y booleanos. Además, también inicializamos el tipo especial vacío. M1: Básicamente nos ayuda a generar el código de 3 direcciones para las condiciones booleanas que sean catalogadas como verdades. Para ello nos ayudaremos de una pila. M2: Igual que en M1 pero en este caso para aquellas condiciones que sean evaluadas como falsas. También hace uso de su propia pila.

	<p>SUMAR_AMBITO: entramos en un bloque superior al actual.</p> <p>RESTAR_AMBITO: salimos del bloque actual.</p>
SIGNO, CONSTANTE, VALORES	<p>Gestionará que la asignación sea correcta, con los símbolos pertinentes.</p> <p>SIGNO: como nos podrían introducir cualquier símbolo matemático, lo que hay que hacer es comprobar que nos introducen o un + o un -</p> <p>CONSTANTE: nos ayuda a saber si una variable es constante o no, para poder hacer las gestiones posteriores a la hora de saber si podemos hacer una asignación o no.</p>
INICIALIZAR_OPERADOR_BOOLEANO, OPERACION_BOOLEANA, VALOR_OPERACION_BOOLEANA, CONDICION_BOOLEANA	<p>Gestionará las expresiones booleanas, comprobando que estas sean del tipo correcto.</p> <p>OPERACIÓN_BOOLEANA: evaluamos los tipos subyacente básicos tanto del primer valor como del segundo y que la operación relacional es correcta, es decir que se comparan tipos subyacente básicos.</p> <p>VALOR_OPERACION_BOOLEANA: en caso de que sea un identificador, hay que comprobar que sea una variable, un argumento o un parámetro.</p>
INICIALIZAR_OPERADOR_ARITMETICO, OPERACION_ARITMETICA, VALOR_OPERACION_ARITMETICA	<p>Se encargará de comprobar las operaciones aritméticas, para ellos comprueba que los operadores aritméticos sean correctos y que sus valores sean numéricos</p> <p>VALOR_OPERACION_ARITMETICA: de manera similar a las operaciones booleanas, hay que comprobar que el tipo subyacente básico sea correcto.</p>

TABLA DE SÍMBOLOS

Todo lo referente a la tabla de símbolos se encuentra en el paquete EstructurasDatos, que contiene 3 clases. Estas clases son:

- TablaSimbolos
- NodoTabla
- DescripcionTipo

La tabla de símbolos (TablaSimbolos) tiene los siguientes atributos, que son los que generan la estructura de datos de la tabla:

- AmbitoActual
- TablaAmbitos
- TablaExpansion
- TablaDescripcion

El atributo ambitoActual es un entero que marca el ámbito actual. La tablaAmbitos está implementada con una pila que almacenará enteros. Ésta indica donde podemos escribir en la tablaExpansion, que es el siguiente atributo. TablaExpansion se encarga de las variables que no son visibles, y está implementado con una lista doblemente enlazada de NodoTabla (LinkedList). Finalmente tenemos una tabla de hashing tablaDescripcion, que es identificada según un string y almacena objetos NodoTabla.

NodoTabla es la clase que nos ayuda a almacenar cada identificador que vamos encontrando a lo largo del código fuente. Es necesario almacenar la siguiente información de cada identificador:

- Identificador inequívoco, que puede ser el propio nombre del identificador, ya que no se podrá repetir. Por lo tanto es un String.
- Ámbito, que nos indica en que profundidad está declarada dicha variable
- Tipo descripción, que es otra clase que nos proporcionará toda la información necesaria en caso de que el identificador que estemos guardando sea uno de los tipos contemplados por nuestro lenguaje de programación, cadenas, enteros o booleanos.
- Adicionalmente tenemos otros atributos que nos ayudarán en tareas posteriores con la gestión de la tabla de expansión y en el análisis semántico de las producciones. Estos son: first, follow e idConstante.

Finalmente, cabe mencionar las funciones que implementa nuestra tabla de símbolos:

- Poner: sirve para almacenar el nuevo identificador en la tabla de símbolos, si no estaba previamente, lo tenemos que añadir, pero también hay que controlar si ya está. En caso de que esté, hay que comprobar si están en el mismo ámbito. Ya que se habilita la posibilidad de tener varias variables con el mismo nombre pero en diferentes ámbitos.
- Consultar: consultamos si existe el identificador dentro de nuestra tabla. En caso de no encontrarlo de primeras, deberemos revisar la tabla de expansión para comprobar si está es un argumento.
- Entrar en bloque: añadimos un ámbito más en nuestra pila; hacemos un push.
- Salir del bloque: salimos del último ámbito que habíamos almacenado, es decir hacemos un pop de la pila.
- Eliminar Identificadores Ambito Anterior: adicionalmente tenemos este método interno de nuestra tabla, que nos ayudará a eliminar todas aquellas variables que se hallaban en el ámbito superior del cual acabaríamos de haber salido y por lo tanto ya no podríamos acceder a ellas.
- Poner parámetro: se tiene que definir un método para poner un parámetro de un subprograma en la tabla de símbolos también.

GESTIÓN DE ERRORES

La gestión de errores es realizada a nivel de operación, es decir, cuando realizamos una instrucción de división por ejemplo, comprobaremos si el denominador es igual a 0. Si es el caso, ya tendremos un string local (mensajeError) que informa acerca del tipo de excepción producida. Así logramos personalizar las excepciones sin realmente crear diferentes excepciones, pues realmente retornamos una excepción general, lo que las diferencia son el mensaje de error.

Estos errores pueden ser producidos en 4 partes diferenciadas: Tabla de Símbolos, Léxico, Sintáctico y Semántico.

A continuación listamos una breve descripción de los diferentes tipos de excepciones que tenemos y a qué ámbitos afectan.

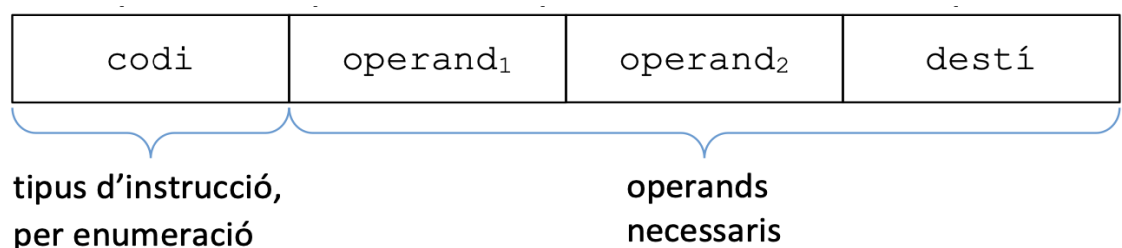
- Una operación matemática no puede ser realizada. Semántico.
- Creamos un id que ya existe. Tabla de símbolos.
- Buscamos un id que no existe. Tabla de símbolos, Léxico.
- No hay coincidencia de tipos. Tabla de símbolos, Semántico.
- Ocupar un espacio superior a la capacidad disponible. Tabla de símbolos.
- Sintaxis errónea. Sintáctico.

BACK-END

Generación de código intermedio

Tal y cómo se ha visto en la teoría, lo ideal para independizar la arquitectura en la que se ejecutará el programa del lenguaje de programación utilizado es tener un código intermedio. Nosotros utilizaremos la técnica del código de tres direcciones.

Tendremos una clase llamada `Codigo3D`. Esta clase incluye el código necesario para designar 4 elementos indispensables del código de 3 direcciones, que son los que hemos visto en la teoría:



Los operandos son los que participan en la operación, y el destino es dónde se guarda el resultado de la misma. En cuanto al id, éste representa las operaciones de las que disponemos, y figuran en la siguiente tabla junto a una breve aclaración de su funcionamiento.

Categorías: variable, valorBooleano, valorAritmetico, valorString, etiqueta, subprograma, nParametro.

ID operación	Operando 1	Operando 2	Destino	Explicación
Copia	variable, valorBooleano, valorAritmetico, valorString	Vacío	variable	a=b
Suma	variable, valorAritmetico	variable, valorAritmetico	variable	a=b+c
Resta	variable, valorAritmetico	variable, valorAritmetico	variable	a=b-c
Producto	variable, valorAritmetico	variable, valorAritmetico	variable	a=b*c
Division	variable, valorAritmetico	variable, valorAritmetico	variable	a=b//c (entera)
Módulo	variable, valorAritmetico	variable, valorAritmetico	variable	a=b%c
AND	variable	variable	variable	a = b AND c
OR	variable	variable	variable	a = b OR c
EtiquetaPasar	Vacío	Vacío	etiqueta	e:skip
Salto	Vacío	Vacío	etiqueta	saltar a otra línea
Menor	variable	variable	variable	a<b
MenorIgual	variable	variable	variable	a<=b
Igual	variable	variable	variable	a==b
Diferente	variable	variable	variable	a != b
Mayor	variable	variable	variable	a>b
MayorIgual	variable	variable	variable	a>=b
EsTrue	variable	Vacío	etiqueta	a = true
EsFalse	variable	Vacío	etiqueta	a = false

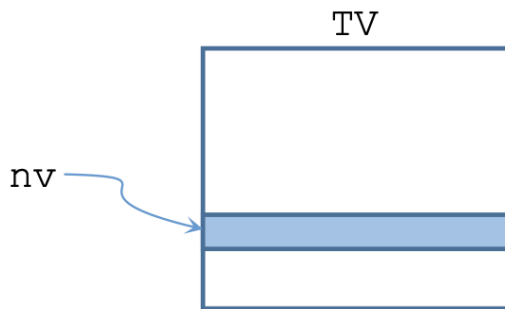
Inicializacion	Vacío	Vacío	subprogr ma	pmb np
Llamada	Vacío	Vacío	subprogr ma	call np
Retorno	subprograma	Vacío	variable	rtn np
ParametroSimple	variable	nParametro	subprogr ma	param_s a
IdSubprograma	Vacío	Vacío	subprogr ma	np
FinSubprograma	Vacío	Vacío	subprogr ma	end np
LlamadaMain	Vacío	Vacío	subprogr ma	call main
EntradaDatos	Vacío	Vacío	variable	in
SalidaDatos	Vacío	Vacío	variable	out

Gestión variables, subprogramas y etiquetas:

Tenemos tres tablas que almacenan diferentes objetos. Una tabla de variables, una tabla de subprogramas y por último una tabla de etiquetas. Las dos primeras tienen una clase dedicada (paquete códigoIntermedioTablas), mientras que la última se halla dentro de la clase GeneradorCodigo3D, ya que de las etiquetas únicamente necesitamos un identificador que sería un String en este caso. Las tres están implementadas como una lista enlazada (LinkedList). El resultado de las tablas las podemos encontrar en TablasCodigoIntermedio.txt, dentro del paquete salidas.

Tabla de variables:

Aquí figurará un registro de las variables que han ido apareciendo en el código3D que vamos generando. Utilizamos el modelo proporcionado en la teoría:



```
novavar {
    nv = nv + 1;
    TV[nv] = ...;
    return nv
}
```

Respecto a cada variable podemos encontrar información diversa con el siguiente formato (se adjunta un ejemplo):

ID 0:

DESCRIPCION VARIABLE:

Nombre=num

Número variable=0

IdSubprograma=0

Ocupación=4

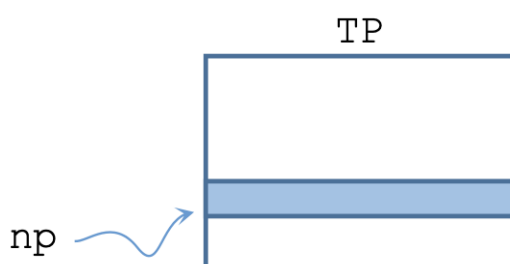
Desplazamiento=4

Tipo subjacente básico=tsEntero

Podemos observar como aparece su propio id, el nombre de la variable, el id del subprograma al cual pertenece, los bytes que ocupa, su desplazamiento y el tipo de la variable.

Tabla de subprogramas:

Al igual que con las variables, esta tabla contiene la información de los subprogramas que iremos generando con el código de 3 direcciones. Utilizamos el modelo mostrado en la teoría:



```
nouproc {
    np = np + 1;
    TP[np] = ...;
    return np
}
```

A continuación adjuntaremos el formato en el que imprimimos la tabla.

ID 0:

DESCRIPCIÓN SUBPROGRAMA:

- Nivel de profundidad = 0
 - Etiqueta inicial = RECURSIVEFACT
 - Ocupación de las variables locales = -42
 - Ocupación parámetros = 4
 - Número de parámetros = 1
 - Ocupación = 4
 - Tipo subyacente básico = tsEntero
-

Del subprograma dicho se puede ver su id, nivel de profundidad de programa, cuál es su etiqueta de inicio, el espacio ocupado por las variables locales y parámetros (en bytes), cuántos parámetros tiene, la ocupación y el tipo de retorno del subprograma.

Tablas de etiquetas:

Para gestionar las etiquetas de salto que se iban generando a lo largo del código de 3 direcciones, hemos planteado la opción de ir guardando cada una de las que se iban generando en una lista doblemente enlazada a la cual llamamos tabla de etiquetas.

Para llevar a cabo el seguimiento de las etiquetas, en nuestro caso las hemos ido numerando según han sido escritas en el programa. Por tanto, su nombre se corresponde el id recibido. Ejemplo:

ID 0: ETIQUETA0

Optimización:

La optimización puede ahorrar muchas líneas de código innecesarias que quitan carga de trabajo al procesador. Una vez obtenido el código3D, lo optimizaremos para reducir el número de instrucciones a realizar. El fichero en el que figuran

estas actualizaciones es en el Optimizacion.java, en el paquete con el mismo nombre.

En nuestro caso realizamos dos tipos diferentes de optimizaciones (cada una tiene su respectivo método en la clase mencionada anteriormente):

- AsignacionesDiferidas: aquí recibimos la instrucción y su índice (como dos parámetros diferenciados). Analizamos si la instrucción puede ser eliminada o no (es decir, si la propia instrucción es redundante y por tanto modificando el código3D se obtendría un resultado mejor) y realizamos la acción necesaria.

El objetivo de esta optimización es ejemplificado a continuación:

Sin optimizar:

```
t1 = 1
```

```
i = t1
```

Optimizado:

```
i = 1
```

- SaltosSubyacentes: al igual que antes, recibimos la instrucción3D y su índice. El modo de gestionar condicionales ifs (en el código3D no optimizado) resulta ineficiente en cuanto a los saltos producidos, pues el propio if en cuestión podría consistir en un único salto. En este método gestionamos dicha interacción.

Sin optimizar:

```
if a > b goto e1
```

```
goto e2
```

```
e1: skip
```

Optimizado:

```
if a <= b goto e2
```

Como podemos observar en el código optimizado, en el caso de no cumplirse la condición realizamos un salto con el que nos saltamos el conjunto de instrucciones que contendría el if. En caso contrario simplemente no realizaríamos el salto y ya estaríamos en la primera instrucción del if. Todo esto a diferencia del caso no optimizado, en el que en cumplamos o no la condición, siempre realizamos un salto.

Generación de código ensamblador

Una vez generado el código intermedio (la lista completa de las instrucciones de 3 direcciones), hay que obtener dicho código en lenguaje ensamblador. Como

nuestra arquitectura es la del motorola68000, el lenguaje ensamblador será el 68K. Ha sido escogido debido a que ya lo conocemos de otras asignaturas (Estructura de Computadores I y II).

Por tanto hemos creado una clase que se llama traductor código que lo que hace es recorrer esta lista de instrucciones del código de 3 direcciones, donde según la operación de cada una de estas, se generarán unas instrucciones del 68k u otras:

- **Llamada main:** genera un salto directo hacia la etiqueta referente al programa principal.
- **Inicialización subprograma**(o preámbulo): hay que tener en cuenta la ocupación de los parámetros del subprograma, el espacio del bloque de activación y actualizar el stack pointer para el nuevo bloque de activación,
- **Llamada subprograma:** generamos el código para cuando llamemos a un subprograma, preparar el espacio de memoria para el retorno del BP. Hay que guardar el apuntador de bloque anterior y en caso de tener un valor de retorno, guardamos espacio para este. También preparamos el espacio para los parámetros.
- **RetornoSubprograma:** ponemos el valor de la variable que queremos retornar en la dirección de la pila donde almacenamos el retorno. En el caso de los Strings, iremos haciendo este procedimiento carácter a carácter.
- **IdSubprograma:** generamos la etiqueta asociada al subprograma que vamos a ejecutar.
- **Fin subprograma:** organizamos la pila para poder salir del subprograma, básicamente sumando el desplazamiento de los parámetros y del BP a la pila.
- **Etiqueta pasar:** escribimos la etiqueta correspondiente de la tabla de etiquetas.
- **Salto:** generamos un salto incondicional a la etiqueta correspondiente, mediante la instrucción JMP.
- **Es true:** hacemos la comparación para saber si hay que saltar a la etiqueta en caso de evaluación verdadera.
- **Es false:** igual pero para saltar a la etiqueta correspondiente a la evaluación falsa.

- **Salida de datos:** Para los enteros y los booleanos, haremos uso de una rutina de interrupción al sistema con el TRAP #15. Por otro lado, para la impresión de los Strings, iremos recorriendo carácter a carácter y utilizaremos un buffer auxiliar que nos ayudará para imprimir el resultado en consola.
- **Copia:** generamos el código ensamblador para las instrucciones de asignación. Básicamente se hace la copia del contenido de una dirección de memoria de la pila, al contenido de la dirección de memoria que queremos que sea el destino. En el caso del String hay que ir carácter a carácter.
- **AND:** se realiza el and lógico de dos operandos.
- **OR:** se realiza el or lógico de dos operandos.
- Todas las **instrucciones relacionales**(igual, diferente, mayor, mayor o igual, menor y menor o igual): son todas idénticas, simplemente cambia el tipo de salto, que en el caso del 68k se facilita ya que hay saltos que ya te hacen dicha comprobación directamente: BGT, BGE, BLT, BLE.
- Todas las **instrucciones aritméticas**(suma, resta, multiplicación, división): idénticas, pero realizamos la instrucción conveniente según el caso: ADD, SUB, MULU, DIVU.

Pruebas realizadas

Una vez completado el compilador, hemos elaborado unos sencillos programas para verificar que todo funciona correctamente. Este funcionamiento correcto implica que el compilador debe ejecutar satisfactoriamente el código recibido, y en caso de que esto no haya sido posible debido a un error, aportar información acerca de dicho error.

Prueba 1(programa 0)

Primer caso de prueba que comprueba la declaración y uso de funciones con diferente número de parámetros.

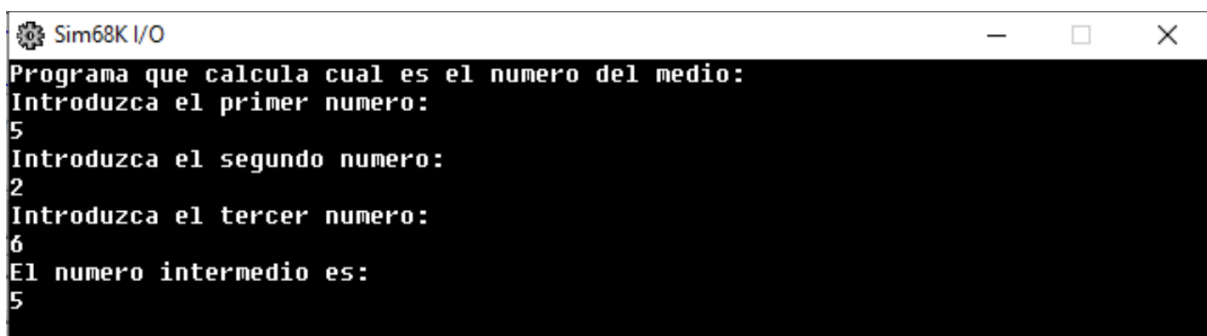
Además del funcionamiento de los condicionales, de la entrada y salida de datos.

En particular, una salida de datos la realizamos a través del valor retornado de una de las funciones.

Nuestro primer caso de prueba tiene la siguiente estructura:

<pre>*** No tiene parámetros *** func int entradaNumero() { *** Entrada de datos *** int numero = 0; stdin(numero); return numero; }</pre>	<p>} Comprobamos el funcionamiento de una función sin parámetros que retorna un entero.</p> <p>Además dentro de la función comprobamos las instrucciones referentes a la declaración y asignación de un numero y la entrada de datos por pantalla.</p>
<pre>*** Función con mas de un parametro *** func int calcularValorIntermedio(int a, int b, int c) { int resultado = 0; if (a < b & b < c){ resultado = b; }else if(a < c & c < b){ resultado = c; }else if(b < a & a < c){ resultado = a; }else if (b < c & c < a){ resultado = c; }else if (c < a & a < b){ resultado = a; }else { *** c < b & b < a *** resultado = b; } return resultado; }</pre>	<p>} Comprobamos el funcionamiento de una función con varios parámetros de entrada.</p> <p>También comprobamos el funcionamiento del condicional if, else y else if.</p> <p>Finalmente, dentro de cada if podemos observar que se realizan operaciones relacionales y lógicas.</p>
<pre>*** Función principal *** func void Principal() { stdout("Programa que calcula cual es el numero del medio:"); stdout("Introduzca el primer numero:"); int n1 = func entradaNumero(); stdout("Introduzca el segundo numero:"); int n2 = func entradaNumero(); stdout("Introduzca el tercer numero:"); int n3 = func entradaNumero(); stdout("El numero intermedio es:"); stdout(func calcularValorIntermedio(n1 n2 n3)); }</pre>	<p>} Comprobamos la salida estándar de datos</p> <p>Comprobamos la llamada a las funciones.</p> <p>Por último, comprobamos la salida de datos con un valor proporcionado por una función</p>

Un posible resultado de ejecución en el ensamblador Easy68K es el siguiente:



```
Sim68K I/O
Programa que calcula cual es el numero del medio:
Introduzca el primer numero:
5
Introduzca el segundo numero:
2
Introduzca el tercer numero:
6
El numero intermedio es:
5
```

Prueba 2(programa 1)

Segundo programa de prueba, que también comprobará el funcionamiento de la entrada y salida de datos.

También comprobaremos el bucle while, las operaciones relacionales y lógicas.

También el if y el else if y operaciones aritméticas.

En este caso no hacemos ningún uso de funciones auxiliares, únicamente llamamos al método principal.

Nuestro segundo caso de prueba tiene la siguiente estructura:

```
func void Principal() {  
    stdout("Programa que da la opción de calcular el cuadrado o el cubo de un numero:");  
    int numero = 0;  
    stdin(numero);  
    int opcion = -1;  
  
    *** Mientras no nos introduzcan una opción valida seguimos preguntando ***  
    while (opcion != 1 & opcion != 2){  
        stdout("Elige la opción: ");  
        stdout("1. Realizar el cuadrado");  
        stdout("2. Realizar el cubo");  
        stdin(opcion);  
    }  
    int resultado = 0;  
  
    *** Segun lo que se haya elegido, se hace una cosa u otra ***  
    if(opcion == 1) {  
        resultado = (numero * numero);  
        stdout("El cuadrado del numero es: ");  
        stdout(resultado);  
    } else if (opcion == 2) {  
        resultado = (numero * numero * numero);  
        stdout("El cubo del numero es: ");  
        stdout(resultado);  
    }  
}  
}
```

Comprobamos la entrada y salida de datos.

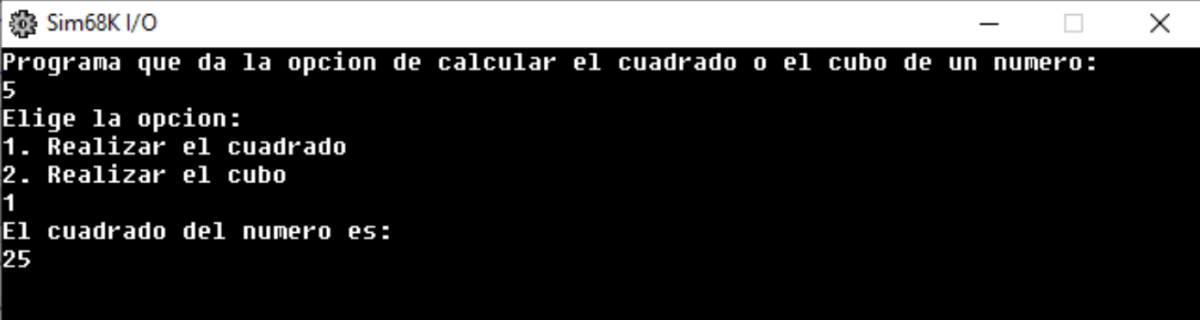
Comprobamos el funcionamiento del bucle while cuya condición son dos operaciones relacionales y una lógica. Hasta que no se haya marcado una de las opciones permitidas, el bucle while seguirá iterando.

Aquí hacemos uso del condicional if y else if, según la opción marcada el resultado será uno u otro.

Comprobamos el funcionamiento de las operaciones aritméticas con una operación de calcular el cuadrado o el cubo

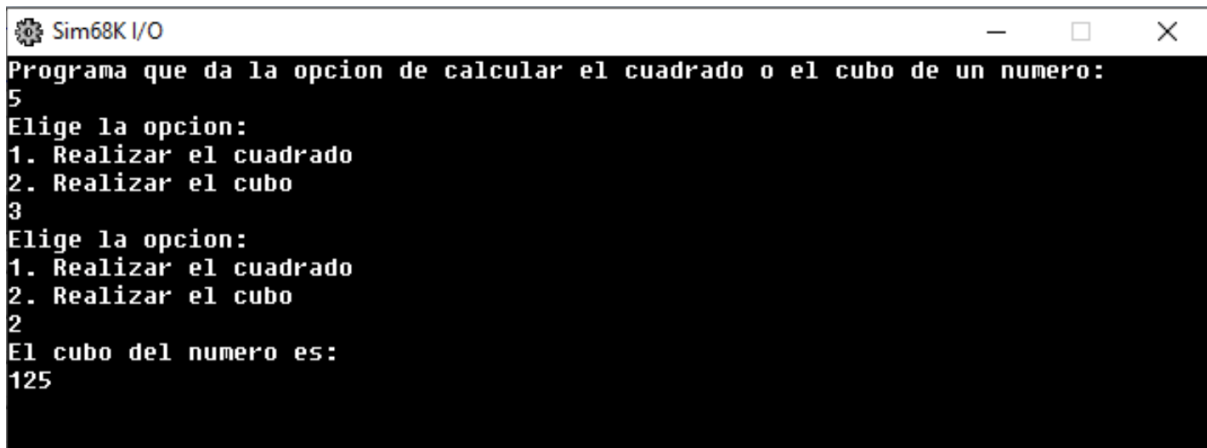
Finalmente, ese resultado se asigna a una variable que luego se imprimirá por pantalla

Un posible resultado de ejecución en el ensamblador Easy68K es el siguiente:



```
Sim68K I/O  
Programa que da la opción de calcular el cuadrado o el cubo de un numero:  
5  
Elige la opción:  
1. Realizar el cuadrado  
2. Realizar el cubo  
1  
El cuadrado del numero es:  
25
```

Otro posible resultado:



```
Sim68K I/O
Programa que da la opcion de calcular el cuadrado o el cubo de un numero:
5
Elige la opcion:
1. Realizar el cuadrado
2. Realizar el cubo
3
Elige la opcion:
1. Realizar el cuadrado
2. Realizar el cubo
2
El cubo del numero es:
125
```

Prueba 3(programa 2)

En esta prueba, además de los condicionales, operaciones aritméticas, entrada, salida de datos entre otros, la idea era comprobar el funcionamiento de una función recursiva.

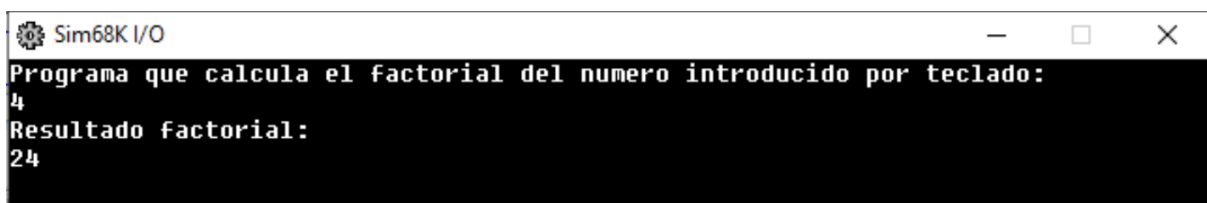
Nuestro tercer caso de prueba tiene la siguiente estructura:

```
func int factorialRecursivo(int num) {
    int resultado = 0;
    if(num != 1) {
        int decremento = (num - 1);
        int resultadoFactorial = func factorialRecursivo(decremento);
        resultado = (num * resultadoFactorial);
    } else {
        resultado = 1;
    }
    return resultado;
}
func void Principal() {
    stdout("Programa que calcula el factorial del numero introducido por teclado:");
    int n = 0;
    stdin(n);
    int resultado = func factorialRecursivo( n);
    stdout("Resultado factorial: ");
    stdout(resultado);
}
```

Comprobación de una función con un parámetro.
También comprobamos el funcionamiento del condicional y de las operaciones aritméticas con una operación de decremento.
Principalmente, la idea fundamental de esta función es la comprobación de que se puede llamar a sí misma y por lo tanto se genera recursividad.

Finalmente, en la función main comprobamos la entrada y salida estándar por pantalla, además del funcionamiento de la función recursiva

Un posible resultado de ejecución en el ensamblador Easy68K es el siguiente:



```
Sim68K I/O
Programa que calcula el factorial del numero introducido por teclado:
4
Resultado factorial:
24
```

Prueba 4 (fallo 0)

En este caso de prueba, comprobamos el funcionamiento de la sintaxis de una variable definida como constante. De todos modos, en este caso el programa debe de dar fallo ya que realizamos una asignación de una variable la cual no habíamos definido previamente y por lo tanto no se ha podido encontrar en la tabla de símbolos, lo que genera un error de dicho tipo.

Nuestro primer caso de fallo tiene la siguiente estructura:

```
func void Principal() {  
    str CONST string1 = "Alcachofa";  
    int _num1 = 5;  
    _num1 = numeroNoDefinido;  
    stdout(_num1);  
}
```

Y generará el siguiente fallo, que recogemos en un fichero dentro de la carpeta de los errores:

```
----- ERRORES ENCONTRADOS -----  
Excepción en la tabla de símbolos de tipo no existe, en el método CONSULTAR; Error al obtener  
id:numeroNoDefinido  
-----
```

Prueba 5 (fallo 1)

En este caso de fallo, lo que hemos hecho es intentar asignar a un entero otra variable de tipo String, lo cual no debería de poder debido a que son de tipos diferentes. Por lo tanto genera un fallo semántico.

Nuestro segundo caso de fallo tiene la siguiente estructura:

```
func void Principal() {  
    str cadena = "Hola Pere :)";  
    int num = 18;  
    num = cadena;  
}
```

Y genera el siguiente fichero de texto:

----- ERRORES ENCONTRADOS -----

Excepción en la semántica del lenguaje. No coincide el tipo de la variable con el del valor. Tipo variable: int. Tipo valor: str.

Prueba 6 (fallo 3)

El último fallo será tratar de asignar a una variable constante un nuevo valor. Como ya habíamos definido que la variable sería constante, esta operación no se puede realizar y por lo tanto generará un error del tipo semántico.

Nuestro tercer caso de fallo tiene la siguiente estructura:

```
func void Principal() {  
  
    int CONST MAXNUMERO = 10;  
    MAXNUMERO = 5;  
}
```

Que generará el siguiente fichero de texto:

----- ERRORES ENCONTRADOS -----

Excepción en la semántica del lenguaje. No se puede cambiar el valor de una constante
