

Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería en Sistemas



Entrega Final

Diseño de Compiladores I

GRUPO 18

Adan Matías Diaz Graziano:
Esteban Boroni:

Temas asignados: 1 6 7 10 11 13 19 22 23 27 28

19/11/2024

ÍNDICE

Correcciones de la primera entrega	2
Analizador Léxico	2
Analizador Sintáctico	3
Implementación	7
Códigos de prueba donde se detectaron errores	8
Generación de código intermedio	20
Estructuras utilizadas	20
Clase Simbolo	20
Clase Tipo	21
Clase Gramatica	22
Clase Generador de Código Intermedio	23
Notación posicional de YAC (\$\$, \$n)	24
Algoritmo de creación de bifurcaciones	27
Problemas durante el desarrollo	32
Generación de Código Assembler	34
GeneraciónCodigoAssembler	34
Atributos	34
Stack<String> pila	34
Int numAuxiliares	34
String ultimaFuncion	34
String ultimaOperacion	34
Métodos	35
recorrerPolaca	35
operadorBinario	37
operadorFuncion	38
operadorSentenciasControl	39
operacionEnteroOctal	40
operacionDouble	42
operadorUnario	44
operadorConversion	46
operadorComparacion	47
operadorInicioFuncion	48
imprimirPorPantalla	49
crearErrorDivisionPorCero	50
crearErrorOverflow	50
crearAuxiliar	50
crearAuxiliarParametroReal	51

crearAuxiliarRetornoFuncion	51
comprobarOperandoLiteral	52
convertirLexemaFlotante	52
convertirLexemaCadena	52
invertirAmbito	53
generarData	54
generarCode	56
generarEncabezado	57
generarPrograma	57
Operaciones Aritméticas	58
Generación de Etiquetas en Bifurcaciones	58
Requerimientos Pendientes	58
Temas particulares	59

Aclaración: No se colocaron todas las sentencias de errores ya que, para la explicación no se iba a entender el código.

Analizador Léxico

Codigo de la primera entrega	Codigo corregido
Autómata. De e0 a e0 con qué char?	Dado el apuro al entregar el informe nos confundimos y el arco que va desde e0 a e0 es incorrecto, este iría de e0 a eFinal. y cómo esto mismo hacen varios caracteres pero con diferentes Acciones Semánticas, tuvimos que agregar dos arcos desde e0 a eF que representan los caracteres que habían quedado excluidos en el Grafo.
"AS_ConcatenasSinSaltoDeLinea". Chequean que no sea salto de línea, si lo es, dónde incrementan el contador?	El contador se incrementa cada vez que se lee un caracter en la función siguienteLectura(Reader,char) de la clase AnalizadorLexico, esta se utiliza para leer el caracter entrante y devolver un número que indica la columna que representa ese caracter en la matriz de estados. En este método se ejecuta un switch y en el caso de reconocerse un caracter SALTODELINEA se incrementa el contador, por lo tanto, no importa en que parte del código se genera un salto de línea ni como se interprete siempre va a incrementarse correctamente el contador de saltos de líneas.
<p>"AS_ERROR Únicamente vacía el buffer con la lectura hasta el momento ya que es un error, se notifica y se devuelve el atributo error." A quién se devuelve? Recuerden no generar problemas sintácticos por errores léxicos.</p> <p>LEXICO</p> <p>Intento incluir en el nombre de un id un carácter que no sea letra, dígito o "_". Esto ocurre porque al leer este caracter invalido devuelve un ERROR (token 280) el Analizador Léxico entonces no lo considera una variable, pero como ya está esperando una variable en la asignación se rompe y no sabemos cómo resolverlo.</p>	El devolver el Token ERROR en vez de solo notificar y continuar con la lectura nos generó varios problemas al generar el análisis sintáctico (como nos mencionas en la segunda corrección) , por lo tanto, decidimos cambiar la matriz y la Acción Semántica AS_ERROR para que al identificarse un error lo que haga es ciclar en el estado actual y notificar el error, continuando la lectura del token. Esto genera que si se está leyendo un identificador "pep#ito" se notifica el error pero devuelve el token identificado cargando en la tabla de símbolos a "pepito". Lo mismo sucede con las CTEs.

Analizador Sintáctico

Codigo de la primera entrega	Codigo corregido
Cte_con_sig "Las otras dos reglas las usamos para acaparar posibles errores.". Cuáles?	Estas dos reglas adicionales se borraron ya que nos confundimos.
<p>"De momento no pudimos implementar el error de falta de la coma entre las variables.". Pendiente para la próxima.</p> <pre> sentencias : sentencias sentencia ';' sentencia ';' sentencia {print(error);} ; declaracion_variable : tipo variables ; variables : variables ',' variable_simple variable_simple ; </pre>	<pre> sentencias : sentencias sentencia sentencia ; declaracion_variable: tipo variables ';' tipo variables error ; variables : variables ',' variable_simple variables variable_simple variable_simple ; </pre>
<p>Problema identificado en la etapa anterior: "Lee "integer a,b c;" y al terminar de leer b espera ';' o ','. Al llegar otro ID no sabe si es una declaración de una variable considerando a b como subtipo o si es parte de la declaración generando shift/reduce. Esto se soluciona bajando el ';' a las variables pero también nos provoca problemas en las sentencias como IF o WHILE que solamente funcionan si colocamos ';'."</p> <p>Solucion: Este cambio fue realizado para solucionar el problema de ambigüedad que nos generaba la declaración múltiple de variables sin coma en el medio. Para esto tuvimos que bajar el "," a todas las sentencias ejecutables. Ahora tenemos el ';' en esta regla y así, logramos detectar cuando falta la ',' entre las variables ya que, solo se espera "variables".</p>	
<pre> bloque_else_multiple: ELSE BEGIN bloque_sent_ejecutables END ; bloque_unidad_multiple : BEGIN bloque_sent_ejecutables END ; </pre>	<pre> bloque_else_multiple:ELSE BEGIN bloque_sent_ejecutables ';' END ; bloque_unidad_multiple : BEGIN bloque_sent_ejecutables ';' END ; </pre>
<p>Solucion: Al bajar los ; a cada sentencia en particular se rompían los bloques múltiples de sentencias en las sentencias de control. El error fue encontrado luego de un tiempo, en la recursividad de sentencias teníamos un ; al medio de la regla. Con esta modificación se solucionó.</p>	
Factor. "Para la posición del arreglo se tiene en cuenta el token CTE dado que sería erróneo intentar acceder con un índice	Se soluciono de la siguiente manera, notificandolo como error:

<p>negativo.". Ok, podían controlarlo con una acción semántica.</p>	<pre> asignacion variable_simple '{' '-' CTE '}' ASIGNACION expresion_arit ; factor variable_simple '{' '-' CTE '}' ; </pre>
<p>Sentencia Ejecutable. "Si se lee un retorno, este pone en TRUE una variable global declarada en la Gramatica.y que es utilizada para saber si dentro de una función se llamó un retorno." No es estrictamente correcto. Ver ejemplos de código más abajo.</p>	<pre> Esto se solucionó pasando al \$\$ sval una palabra para reconocer si existe un RETORNO al reducir. Sentencia_ejecutable: asignacion sentencia_IF {if(\$1.sval=="RET"){\$\$.sval="RET";}} ... retorno {\$\$.sval="RET";} ; bloque_sentencia_simple: sentencia_ejecutable {if(\$1.sval=="RET"){\$\$.sval="RET";}} ; Bloque_sentencias_simple se utiliza para crear el cuerpo del THEN y ELSE. Este se reduce en bloque_unidad_simple que seria una unica sentencia para el THEN o en bloque_else_simple que seria lo mismo para el ELSE. Para que el cuerpo del THEN o ELSE puedan tener varias sentencias existe otra regla que es bloque_sent_ejecutables. Esta recursivamente reduce todas las sentencias para despues ser el cuerpo del THEN o ELSE. Esta ultima como recibe una o mas sentencias debe verificar que estas posean un RET, por lo tanto, pregunta si alguna de las dos sentencias tiene return y si asi es, se reduce con \$.sval = "RET". bloque_unidad_simple: bloque_sentencia_simple {if(\$1.sval=="RET"){\$\$.sval="RET";}} ; bloque_else_simple:ELSE bloque_sentencia_simple {if(\$2.sval=="RET"){\$\$.sval="RET";}} ; bloque_sent_ejecutables : bloque_sent_ejecutables ';' bloque_sentencia_simple {if(\$1.sval=="RET" \$3.sval=="RET") {\$\$.sval="RET";}} ; </pre>

```

bloque_unidad_multiple : BEGIN
bloque_sent_ejecutables ';' END
{if($2.sval=="RET"){$.sval="RET";}}

bloque_else_multiple:ELSE BEGIN
bloque_sent_ejecutables ';' END
{if($3.sval=="RET"){$.sval="RET";}}

bloque_unidad      : bloque_unidad_simple
{if($1.sval=="RET"){$.sval="RET";}}
| bloque_unidad_multiple
{if($1.sval=="RET"){$.sval="RET";}}
;

bloque_else: bloque_else_simple
{if($1.sval=="RET"){$.sval="RET";}}
| bloque_else_multiple
{if($1.sval=="RET"){$.sval="RET";}}
;

Ahora solo se verifica si ambos cuerpos (THEN
o ELSE) poseen un return.

sentencia_IF: IF condicion THEN
bloque_unidad ';' bloque_else ';' END_IF
{if($4.sval=="RET" && $6.sval=="RET"){
$.sval="RET";}}

De esta manera, cuando se utilicen sentencias
en una función el IF va a contar como retorno
cuando ambos cuerpos lo tengas.
Retomando la primera regla se visualiza esto.

sentencia_ejecutable: asignacion
| sentencia_IF {if($1.sval=="RET")
{$.sval="RET";}}
| ...
| retorno {$.sval="RET";}
;

sentencia:| sentencia_ejecutable ';'
{if($1.sval=="RET"){$.sval="RET";}}

De esta manera, cuando el conjunto de
sentencias ( como un IF o un RETORNO) se
reduzca a "sentencias" va a cargarse en
$.sval "RET" cuando al menos una sentencia lo
retorne, identificando la existencia del
RETORNO.

sentencias: sentencias sentencia

```



```
{if($1.sval=="RET" || $2.sval=="RET")
{$.sval="RET";}}
| sentencia
{if($1.sval=="RET") {$.sval="RET";}}
;

cuerpo_funcion: sentencias
{if($1.sval=="RET") {$.sval="RET";}}
;

declaracion_funciones: encabezado_funcion
parametros_parenthesis BEGIN cuerpo_funcion
END { if($4.sval!="RET"){falta el retorno}}
```

Por ultimo, la declaración de función antes de reducirse verifica la existencia del \$4.sval == "RET", reconociendo la existencia o ausencia del RETORNO.

Implementación

Codigo de la primera entrega	Codigo corregido
<p>· Los archivos que tienen para las matrices las deberían generar como resources que existan dentro del jar para evitar dependencias de desde dónde se ejecuta el .jar y evitar requerir archivos externos.</p> <p>- Además, por como pusieron los nombres de las rutas, ni siquiera los levanta estando en el folder del proyecto.</p> <p>· Están generando mal los nombres de los archivos de salida. Revisar las pautas de desarrollo del trabajo que les envié al iniciar la cursada.</p>	<p>Se solucionó separando la carpeta en la que tenía los archivos .txt. Reordenandolas para que las matrices sean levantadas por el .jar.</p> <p>Estructura primera entrega: Compilador2024</p> <pre>├── src │ ├── AccionSemanticas │ └── Compilador └── resources ├── codigosDePrueba.txt ├── matrizDeAcciones.txt ├── matrizDeEstados.txt └── PalabrasReservadas.txt</pre> <p>Estructura actual: Compilador2024</p> <pre>├── src │ └── main │ ├── java │ │ ├── AccionSemanticas │ │ └── Compilador │ └── resources │ ├── matrizDeAcciones.txt │ ├── matrizDeEstados.txt │ └── PalabrasReservadas.txt └── resources └── codigosDePrueba.txt</pre>

Códigos de prueba

donde se detectaron errores

Codigo de prueba	Error encontrado	Comentarios del docente	Correccion	Gramatica/Codigo/Resultado actualizado
	syntax error stack underflow. aborting...	Archivo vacío no debería dar error.	Como teníamos la siguiente regla: Expresion_arit : error ; Al no haber código este error buscaba reducirse consumiendo un token, al no haber token daba ese error. Después de cambiar el uso del token 'error' se soluciono.	
lala begin end	<div> <div>≡</div> <div>Identificador lala</div> </div> <div> <div>≡</div> <div>Identificador begin</div> </div> <div> <div>≡</div> <div>Identificador end</div> </div> syntax error stack underflow. aborting...	NO debería haber error.	Esto sucedía porque faltaba el siguiente error a identificar.	programa: ID_simple BEGIN END ;

Diseño de Compiladores I

<pre> lala begin integer varx, vary, varx varx := 327 varx := 2.3d+34; tipo_abc vary; end </pre>	<pre> ┌─┐ Identificador lala ┌─┐ Identificador begin ┌─┐ Identificador integer ┌─┐ Identificador varx ┌─┐ ┌─┐ , ┌─┐ Identificador vary ┌─┐ ┌─┐ , ┌─┐ Identificador varx ┌─┐ Identificador varx Linea 5 declaracion de variables Linea 5: Erro: Falta ';' al final de la sentencia ┌─┐ ┌─┐ := ┌─┐ Constante entera 327 ┌─┐ Identificador varx Linea :6 Asignacion syntax error stack underflow. aborting... </pre>	<p>No está reconociendo todo el código. Solo detecta el primer error.</p>	<p>Al no poder identificar la declaración múltiple junto al uso excesivo del token 'error' este se consumía al ';' y generaba este error.</p>	<p>Errores detectados ahora:</p> <p>Linea 5 Error: Falta ';' entre variables</p> <p>Linea 5 Error: Falta ';' al final de la sentencia</p>
<pre> lala begin varx := 327 varx := 2.3d+30004; end </pre>	<pre> Linea 5: Erro: Falta ';' al final de la sentencia ┌─┐ ┌─┐ := ┌─┐ Constante double 2.3d+30004 ┌─┐ ┌─┐ ; </pre>	<p>Esto no debería estar fuera de rango?</p>	<p>Esto sucedía porque en la condición que evaluaba si estaba o no fuera de rango, estábamos utilizando ' ' (or's) en vez de '&&' (and's).</p>	<p>Línea 6 Error: Falta ';' al final de la sentencia.</p> <p>Error léxico en la línea 6 : Constante double fuera de rango.</p>

Diseño de Compiladores I

lala begin			Esto sucedía porque faltaba realizar un try{}catch{} por si el buffer juntaba una palabra más grande que la que un String puede soportar.	
varx := 327999999999999999999999999999999; varx := 2.3d+30004;	Exception in thread "main" java.lang.NumberFormatException: For input string: "327999999999999999999999999999999"			
end				
lala begin integer adancsadjellklklklks#@aea; end Corrección de este código por el uso de '@' para ETIQUETA. lala begin integer adancsadjellklklklks#aea; end	Identificador lala Identificador begin Identificador integer Linea 4 WARNING: El identificador 'adancsadjellklklklks' fue truncado a 'adancsadjellklk' y este podría reconocerse como palabra reservada. Identificador adancsadjellklk Error lexico en la linea4: No se identifica el token syntax error stack underflow. aborting...	Acá deberían darle el token id al sintáctico para que no caiga en el syntax error. Los syntax error no deberían tenerlos.	En la tercera corrección del Análisis Léxico (hoja 2) se explica en detalle este error. Esto sucedía ya que al identificar un error como este, devolvíamos el token ERROR. Esto nos generaba muchos errores sintácticos por lo tanto, continuando con la lectura del token sin devolver el token ERROR solucionó el problema.	


Diseño de Compiladores I

lala begin VARX := 1 5; VARX := 1 + ; VARX := + 5 ; end	La expresion está mal escrita	La descripción de los errores es genérica, los tres casos dan lo mismo.	Aquí se observa uno de los ejemplos del mal uso del token 'error'. Este error se ejecutaba tanto cuando había un error en una asignación como cuando se mandaba a ejecutar un archivo vacío. Cuando entendimos que este consumía el siguiente token pudimos utilizarlo correctamente.	Codigo viejo expresion_arit : expresion_arit '+' termino expresion_arit '-' termino termino error {print(La expresión está mal escrita)} ; CODIGO ACTUALIZADO expresion_arit : expresion_arit '+' termino expresion_arit '-' termino {print(error);} termino {print(error);} error '+' error {print(error);} error '-' {print(error);} expresion_arit '+' error {print(error);} expresion_arit '-' error {print(error);} error termino {print(error);} ;
lala begin goto lld; end	Linea :4 Error: Falta la etiqueta en GOTO Linea 4: Erro: Falta ';' al final de la sentencia	Mal identificado.	En este caso pensamos que está bien porque al no haber una ETIQUETA después del goto, se detecta el error y devuelve tanto la etiqueta como el ; que falta. Esto es porque al	

Diseño de Compiladores I

			"l;" lo toma como la siguiente sentencia.	
lala begin IF(a < 35)THEN Begin A:=1; B:=1; END_IF; end	syntax error La expresion está mal escrita =;	Sobreuso del error.	Estos errores no los tuvimos en cuenta en la primera entrega, por lo tanto, se agregaron las reglas correspondientes a estos errores.	Código primera etapa bloque_unidad_multiple : BEGIN bloque_sent_ejecutables END ; bloque_else_multiple: ELSE BEGIN bloque_sent_ejecutables END ; Codigo actual bloque_unidad_multiple : BEGIN bloque_sent_ejecutables ';' END BEGIN END BEGIN bloque_sent_ejecutables END BEGIN bloque_sent_ejecutables error ; bloque_unidad_multiple : BEGIN bloque_sent_ejecutables ';' END BEGIN END BEGIN bloque_sent_ejecutables END BEGIN bloque_sent_ejecutables error ;

Diseño de Compiladores I

lala begin IF(a < 35)THEN begin A:=1; B:=1; end END_IF; end	syntax error La expresion está mal escrita 	Sobreuso del error.	Este error fue solucionado en conjunto con la explicación anterior.	Línea 10 Error: Falta ';' al final de la sentencia del bloque del THE
lala begin RET (z); end		No da error pero es incorrecto. No puede haber ret afuera de funciones. Es un error sintáctico.	Para esto se agregó una variable global booleana en el encabezado de la función que indique si se abrió una función. De esta manera corroboramos que se haya llamado dentro de una.	
lala begin RET (); end	syntax error La expresion está mal escrita		Faltaba la regla que detectaba si faltaba el parámetro.	Línea :4 Error : Falta el parametro del RETORNO Línea :4 Error : RETORNO declarado fuera del ambito de una funcion

Diseño de Compiladores I

lala begin WHILE ((a,3+4)=(b,35)) BEGIN OUTF(1); OUTF([Hola Mundo]); goto afuera@ ; end	Linea 12 Error: falta el cuerpo del WHILE Linea 12: Erro: Falta ';' al final de la sentencia		Como se explico anteriormente, se agregaron las reglas faltantes para reconocer estos errores.	Linea 13 Error: Falta ';' al final de la sentencia Linea 13 Error: Falta el delimitador END
lala begin WHILE ((a,3+4)=(,35)) BEGIN OUTF(1); END; end	Linea 7 Error: falta el cuerpo del WHILE Linea 7: Erro: Falta ';' al final de la sentencia		Antes de iniciar la tercera etapa andaba correctamente con lo indicado en la columna "Gramatica/Codigo/Resultado actualizado". Al iniciar la etapa 3 como no se carga la variable 'a' a la polaca (por no estar declarada) y al faltar un parámetro se rompe.	Le agregamos a list_expre: list_expre ',' expresion_arit expresion_arit las dos reglas q no contemplamos ',' expresion_arit list_expre ','

Diseño de Compiladores I

<p>lala</p> <p>begin</p> <p>WHILE ((a,3+4)=(v,35))</p> <p>BEGIN</p> <p>OUTF(1);</p> <p>END;</p> <p>end</p>	<p>Línea 7</p> <p>Error: falta el cuerpo del WHILE</p> <p>Línea 7: Erro: Falta ';' al final de la sentencia</p>	<p>Será que no permiten bloques con una única sentencia?</p>	<p>Antes</p> <p>teníamos el ';' en la regla de las sentencias, por lo tanto, reducía a 'sentencia_ejecutable' con "OUTF(1)" y el ';' se leía como una sentencia a parte. Esto se soluciono cuando bajamos el ';' a cada tipo de sentencia reduciendo la regla con el ';' al reconocer el bloque_sent_ejecutable.</p>	<p>Código</p> <p>primera entrega</p> <p>Sentencias</p> <pre>: sentencias sentencia ';' sentencia ';' sentencia ;</pre> <p>Sentencia</p> <pre>: sentencia_declarativa sentencia_ejecutable ;</pre> <pre>sentencia_declarativa : declaracion_variable declaracion_funciones declaracion_subtipo ;</pre> <p>Código actualizado</p> <pre>sentencias : sentencias sentencia sentencia ;</pre> <pre>sentencias sentencia_ejecutable</pre> <pre>bloque_sentencia_simple: sentencia_ejecutable ;</pre>
--	---	--	--	---

Diseño de Compiladores I

				bloque_sent_ejecutables : bloque_sent_ejecutables ';' ; bloque_sentencia_simple ;
lala begin WHILE ((a,3+4)=(v,35)) OUTF(1); ; end	syntax error stack underflow. aborting...		Agregamos la regla en 'sentencias' error ';' ; para que lea hasta encontrar un ';' y pueda identificar los casos estos.	Código primera entrega sentencias : sentencias sentencia ';' ; sentencia ';' ; sentencia ; Código actualizado Sentencia : sentencia_declarativa error ';' ; sentencia_ejecutable ';' ; sentencia_ejecutable ;

Diseño de Compiladores I

lala begin WHILE (a 3+435) OUTF(1); ; End	syntax error La expresion está mal escrita = + = Constante entera 435 =) = Identificador OUTF syntax error La expresion está mal escrita = (= Constante entera 1 =) = ; La expresion está mal escrita = ; = Identificador end	Tres errores para lo mismo?	En ese momento no se contemplaban esos errores, por lo tanto se añadieron las reglas correspondientes. Actualmente, da los siguientes errores. Linea :4 Error: La expresion esta mal escrita, falta el operador Linea :4 Error: La expresion esta mal escrita, falta el operador Linea 7 Error: faltan las sentencias antes del ';' Creemos que están bien considerados ya que primero al reducir en una expresión aritmética no sabe si son dos expresiones o si es (a+3+435) y falta el primer '+'. Posteriormente, reduce a dos expresiones y notifica la falta de comparador.	Reglas añadidas: Condicion: '(' list_expre ')' '(' list_expre ')' ' '(' list_expre ' ;
---	--	-----------------------------------	---	---

Diseño de Compiladores I

<pre> lala begin integer FUN juancito (double a) BEGIN IF (e < 3) THEN RET (x) ELSE RET (a); END_IF; END; end </pre>	<pre> ┌┐ Identificador ELSE syntax error La expresion está mal escrita ┌┐ Identificador RET </pre>	<p>No es ese el error.</p>	<p>El uso excesivo del token 'error' nos detectaba errores sin sentido y generaba el mensaje. "Syntax error".</p> <p>A partir de eso, cambiamos el uso del token 'error' en varias reglas y se solucionaron varios errores de este tipo.</p>	
<pre> lala begin integer FUN juancito (double a) BEGIN IF (e < 3) THEN RET (x); ELSE x := 3; </pre>		<p>No da error, pero debería decir que falta el ret. Solo se puede retornar desde la rama del then.</p>	<p>Esto se solucionó pasando al \$\$.\$val una palabra para reconocer si existe un RETORNO al reducir. En la hoja 4 fue explicado a profundidad.</p>	<p>Linea 12 Error: Faltan el RETORNO de al funcion</p>

Diseño de Compiladores I

<p>END_IF;</p> <p>END;</p> <p>end</p>				
<p>lala</p> <p>begin</p> <p>integer FUN juancito (</p> <p>double a) BEGIN</p> <p>OUTF(f)</p> <p>RET (4)</p> <p>end</p> <p>end</p>	<p>Linea 7: Erro: Falta ';' al final de la sentencia</p> <pre> == (== Constante entera 4 ==) == Identificador end </pre> <p>syntax error</p> <p>stack underflow. aborting...</p>	<p>Hay un problema con el reconocimien to de faltante de ;</p>	<p>El mal uso excesivo del token 'error' consumia token's generando errores en otras reglas ya que, reducía consumiendo token's.</p>	<p>Linea 7 Error: Falta ';' al final de la sentencia</p> <p>Linea 8 Error: Falta ';' al final de la sentencia</p> <p>Linea 10 Error: Falta ';' al final de la sentencia</p> <p>Falta en el OUTF, RET y end.</p>
<p>lala</p> <p>begin</p> <p>integer FUN juancito (</p> <p>double a, double v) BEGIN</p> <p>RET (4);</p> <p>end;</p> <p>end</p>	<pre> ==) </pre> <p>Linea 4 Error: Falta el parametro en la funcion.</p> <pre> == Identificador BEGIN == Identificador RET == (</pre>	<p>Mal reconocido.</p>	<p>Funciona correctamente pero si se utiliza en el codigo actual va a dar error por la construccion errores de la POLACA.</p>	<p>Linea 4 Error: Se excedio el numero de parametros (1).</p>

Generación de código intermedio

En esta etapa, implementamos una representación basada en **Polaca Inversa**, organizando las instrucciones del programa mediante estructuras que permiten separar y gestionar cada ámbito de forma independiente. Para esto, utilicé:

- **Un Map para las Polacas**, donde cada clave representa un ámbito (pj: \$MAIN) asociado a una lista que almacena las instrucciones generadas en ese contexto.
- **Un Map de pilas**, que permite manejar elementos temporales o índices necesarios para construir y resolver las bifurcaciones y saltos durante la generación del código.
- **Un Map de posiciones**, para llevar un control preciso de la ubicación actual en la Polaca dentro de cada ámbito.

Esta estructura me permitió crear una Polaca para cada ámbito de forma dinámica y manejarla eficientemente con métodos como *addElemento*, *apilar*, *desapilar*, *bifurcarF* y *bifurcarI*.

Además, implementé un mecanismo para gestionar los **goto's** mediante dos listas, una para las Etiquetas y otra para los datos de los Goto's, que almacena y resuelve referencias en las bifurcaciones al completar la construcción de las instrucciones.

Con este diseño, logré mantener separadas las instrucciones de cada ámbito y garantizar una correcta organización y resolución de los flujos de control en el programa.

Estructuras utilizadas

Clase Simbolo

Esta clase ya estaba creada pero se le agregaron más atributos.

Este atributo es para asignarle el tipo

```
private Tipo tipoVar;
```

El uso

```
private String uso;
```

Esta se carga en caso de que este símbolo es una función, asignando el tipo del parámetro formal. De esta manera en una invocación poder verificar si los tipos son compatibles o no, buscando el nombre de la función en la tabla de simbolos y comparando su tipo con el parámetro.

```
private String tipoParFormal=" ";
```

Este atributo se cargo para almacenar el ambito de cada simbolo. Su uso mas relevante es para cuando hay que invocar una funcion, gracias a esta variable puedo acceder al ambito y llamar a la funcion, cargando en la polaca el nombre de la variable nomas.

```
private String ambitoVar="";
```

Ej:

POLACA

```
[ funcion$MAIN , CALL ]
```

ASSEMBLER

```
$MAIN$funcion:
```

```
...
```

```
CALL $MAIN$funcion ( se accede al ámbito de la función y se llama a este).
```


Clase Tipo

Esta clase únicamente se usa para crear Tipos. Estos Tipos van a estar cargados en un Map para poder crear varios Tipos del mismo Tipo.

Por ejemplo:

Al ejecutar:

ala

Begin

```
TYPEDEF TRIPLE < integer > tint;
TYPEDEF TRIPLE < integer > tint2;
TYPEDEF flotadito := integer {6, 8};
TYPEDEF flotadito2 := integer {6, 8};
```

end

>>>> TIPOS <<<<<

[OCTAL, OCTAL]

[tint2, INTEGER[3]]

[flotadito2, INTEGER[-6.0, -8.0]]

[flotadito, INTEGER[-6.0, -8.0]]

[DOUBLE, DOUBLE]

[tint, INTEGER[3]]

[INTEGER, INTEGER]

[ETIQUETA, ETIQUETA]

De esta manera, tint y tint2 son del mismo tipo y se pueden buscar mediante el nombre con el que fue agregado al Map de Tipos.

Guarda el nombre del Tipo creado. Pj: INTEGER;

```
private String type=null;
```

Ambas se ponen en true indicando que son subtipo o triple.

```
private boolean subTipo=false;
```

```
private String nomSubTipo="";
```

```
private boolean triple=false;
```

En caso de ser subTipo se carga su rango, por defecto están cargadas con su máximo y mínimo pero al crearse se le asignan sus rangos dependiendo el tipo que sea.

```
private double rangInferiorDouble=Double.MIN_VALUE;
```

```
private double rangSuperiorDouble=Double.MAX_VALUE;
```

```
private int rangInferiorInteger=Integer.MAX_VALUE;
```

```
private int rangSuperiorInteger=Integer.MIN_VALUE;
```

Clase Gramatica

ACLARACIÓN: No utilizamos : como dijo la cátedra porque no generaba error en la generación de código.

```
public static StringBuilder AMBITO = new StringBuilder("$MAIN");
```

Esta estructura fue utilizada para ir llevando el registro del ámbito en el que se está al momento de llamarla. Esta inicialmente se llama "\$MAIN" y se le va concatenando el símbolo "\$" + el nombre del ámbito creado.

```
public static Stack<String> DENTRODELAMBITO = new Stack<String>();
```

Esta pila fue creada con el objetivo de saber cuando se está dentro de una función y cuando no. Cada vez que crea una función apilo su nombre, con el objetivo de poder identificar si un RETORNO es llamado dentro o fuera de una función. El objetivo de que sea una pila es para apilar las funciones creadas dentro de otras funciones.

```
public static boolean RETORNOTHEN = false;
```

```
public static boolean RETORNOELSE = false;
```

Estas variables booleanas se crearon para saber si el cuerpo del THEN o ELSE de un IF posee un RETORNO. Su función es verificar cuando ambas estén en TRUE y si es así se le devuelve a la función que esta sentencia ejecutable posee un RETORNO.

Esta variable se usa cuando se leen una lista de expresiones para saber cuantas expresiones contiene. Con esta se chequea la igualdad en una condición y se completan todas las BF.

```
public static int cantDeOperandos;
```

```
public static Map<String,Tipo> tipos = new HashMap<>();
```

En esta estructura llevamos el registro de los TIPOS que puede tener una variable, su KEY es el nombre del Tipo y en Tipo contengo toda la información necesario para el desarrollo de la etapa 3 y 4.

Ej:

```
TYPEDEF TRIPLE < integer > tint;
```

```
TYPEDEF flotadito := integer {-6, -8};
```

```
<TIPOS>
```

```
[OCTAL, OCTAL]
```

```
[flotadito, INTEGER[ -6.0, -8.0 ]]
```

```
[DOUBLE, DOUBLE]
```

```
[tint, INTEGER[3]]
```

```
[INTEGER, INTEGER]
```

El crear así los tipos me permite crear otra pj: TRIPLE de Tipo INTEGER y decir que son del mismo tipo, aunque se llamen diferente, ya que ambas son de tipo INTEGER[3].

Clase Generador de Código Intermedio

Para esta etapa se creó la clase `GeneradorCodigoIntermedio`, su función es operar con las estructuras utilizadas para la creación de la POLACA . Se conforma por las siguientes estructuras:

```
public static Map<String, ArrayList<String>> polacaFuncional = new HashMap<>();
public static Map<String, Stack<Integer>> Pilas = new HashMap<>();
public static Map<String, Integer> pos = new HashMap<>();
    public static ArrayList<String> Etiquetas = new ArrayList<>();
    public static ArrayList<String[]> BaulDeGotos = new ArrayList<>();
```

Estos se crearon Map's con el objetivo de crear varias estructuras, una por cada ámbito. De esta manera, si estás en el ámbito "\$MAIN" vas a tener su Polaca, su Pila y su Posición diferente al de los demás ámbitos. La Lista Etiquetas almacena todas las etiquetas definidas en el código, sirviendo como referencia para verificar si una etiqueta existe. La Lista BaulDeGotos contiene información sobre los asaltos incondicionales pendientes por las instrucciones GOTO. Cada entrada almacena detalles como la etiqueta objetivo, el ámbito donde se definió y su posición en el código intermedio. Estos permiten procesar las instrucciones GOTO una vez que se ha definido la etiqueta correspondiente, verificando la existencia y detectar errores.

GeneradorCodigoIntermedio

```
public static void addNuevaPolaca() {
    polacaFuncional.put(Parser.AMBITO.toString(), new ArrayList<String>());
    Pilas.put(Parser.AMBITO.toString(), new Stack<Integer>());
    pos.put(Parser.AMBITO.toString(), 0);
}
```

Gramatica.y

```
encabezado_funcion : tipo FUN ID {
    $$.$val=$3.$val;
    cargarVariables($3.$val,(Tipo)$1.obj," nombre de funcion ");
    agregarAmbito($3.$val);
    DENTRODELAMBITO.push($3.$val);
    GeneradorCodigoIntermedio.addNuevaPolaca();
    cargarErrorElImprimirlo(" Encabezado de la funcion ");}
| tipo FUN {cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + ": Error: Faltan el
nombre en la funcion ");}
;
```

Notación posicional de YAC (\$\$, \$n)

El uso que le dimos fue pasar valores que se perdían al reducir las reglas como el tipo, los retornos en sentencias IF, nombre de identificadores y los tipos de los parámetros.

Declaración del tipo a las variables

Para las declaraciones le asignamos al \$\$.\$obj un objeto de tipo 'Tipo' para llevar el registro de las variables que fueron declaradas y asignarle el tipo cuando se vaya a reducir a declaracion_variable. Esto me sirve tanto para declaraciones simples como para declaraciones múltiples.

Para las declaraciones múltiples, como se observa en 'variables', se van concatenando los nombre de las variables a declarar separadas por un "/" en \$\$.\$sval. Al ejecutarse la función cargarVariables, ésta, si el String contiene un "/", crea un arreglo y almacena todas las variables a declarar, asignando su ámbito, tipo y uso.

Gramática.y

```
tipo_primitivo: INTEGER { $$.$obj = tipos.get("INTEGER"); }
;
```

```
tipo : ID_simple { if(tipos.containsKey($1.sval)){ $$.$obj = tipos.get($1.sval); }
      else { cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + " Se utilizo un tipo
            desconocido "); } }
      | tipo_primitivo { $$.$obj = $1.$obj; }
;
```

```
variables : variables ',' variable_simple { $$.$sval = $1.sval + "/" + $3.sval; }
      | variable_simple { $$.$sval = $1.sval; }
;
```

```
declaracion_variablee : tipo variables ';' { cargarVariables($2.sval, (Tipo) $1.$obj, " nombre de
variable "); }
;
```

Reconocimiento de sentencia RETORNO en sentencias IF.

Gramática.y

Aquí el \$\$\$.sval se utilizó para reconocer si existe un RETORNO al reducir.

Sentencia_ejecutable: asignacion

```
| sentencia_IF {if($1.sval=="RET"){$$$.sval="RET";}}
```

```
|...
```

```
| retorno {$$$.sval="RET";}
```

```
;
```

Bloque_sentencia_simple

```
: sentencia_ejecutable {if($1.sval=="RET"){$$$.sval="RET";};}
```

```
;
```

Se le asigna a \$\$\$.sval la palabra "RET" en todas las reglas intermedias para que al llegar a la regla del cuerpo de la función se pueda saber si alguna de las sentencias del cuerpo de la función tiene un RETORNO. Esto se verifica en la regla "sentencias" retornando true si alguna de las sentencias posee en su .sval un "RET".

Gramática.y

```
sentencias: sentencias sentencia {if($1.sval=="RET" || $2.sval=="RET" ){$$$.sval="RET";}}
```

```
| sentencia {if($1.sval=="RET"){$$$.sval="RET";}}
```

```
;
```

```
cuerpo_funcion: sentencias {if($1.sval=="RET"){$$$.sval="RET";}}
```

```
;
```

```
declaracion_funciones: encabezado_funcion parametros_parentesis BEGIN cuerpo_funcion  
END {if($4.sval!="RET"){ Print(ERROR, falta el retorno);}}
```

Por último, la declaración de función antes de reducirse verifica la existencia del \$4.sval == "RET", reconociendo la existencia o ausencia del RETORNO.

Registrar el nombre de la función y el tipo del parámetro formal

El uso del \$\$\$ aquí fue para conservar la información del parámetro formal y el encabezado de la función al reducir dicha función.

Como se observa a continuación, en el \$\$\$ del encabezado_funcion le asignamos el nombre de la función para poder acceder a ella y asignarle el tipo del parámetro formal en la tabla de símbolo y para el mismo motivo se le asigna en el \$\$\$ a parametros_parentesis para pasar el tipo.

Gramática.y

```
declaracion_funciones : encabezado_funcion parametros_parentesis BEGIN cuerpo_funcion  
END
```

```
{if($4.sval!="RET"){
```

```

cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + " Error: Faltan el
RETORNO de al funcion ");}
sacarAmbito();
DENTRODELAMBITO.pop();
cargarParametroFormal($1.sval+ AMBITO.toString(),(Tipo)$2.obj);}
.
,

encabezado_funcion : tipo FUN ID {
    $$sval=$3.sval;
    cargarVariables($3.sval,(Tipo)$1.obj," nombre de funcion "); agregarAmbito($3.sval);
    DENTRODELAMBITO.push($3.sval);
    GeneradorCodigoIntermedio.addNuevaPolaca();
    cargarErrorElImprimirlo(" Encabezado de la funcion ");}
.
,

parametros_parenthesis: '(' tipo_primitivo ID_simple ')' {$$.obj=$2.obj;
    GeneradorCodigoIntermedio.addElemento($3.sval + AMBITO.toString());
    GeneradorCodigoIntermedio.addElemento("PF"); cargarVariables($3.sval,(Tipo)$2.obj,"
    nombre de parametro real ");}
.
,

```

Registrar cuantas expresiones existen en un Pattern Matching

Al momento de reducir un conjunto de expresiones en una lista de expresiones cargaba a la lista de expresiones la cantidad de expresiones que existen dentro de ella. De esta manera, cuando se reduce con una única expresiones se carga el \$\$sval con '1', en caso de haber más, se incrementa el valor que trae la lista de expresiones.

Gramática.y

```

list_expre: list_expre ',' expresion_arit {$$.ival=$1.ival + 1;}
           | expresion_arit {$$.ival=1;}
.
,

condicion: '(' '(' list_expre ')' comparador '(' list_expre ')' ')' {
    if($3.ival == $7.ival)
    {$$.ival=$3.ival;
}
}else{
    cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + "Error : Cantidad de
operandor incompatibles en la comparacion ");}}
.
,

```

Algoritmo de creación de bifurcaciones

GeneradorCodigoIntermedio

Aclaración: Toda estructura va ser llamada “*estructura*.get(Parser.AMBITO.toString())” ya que cada una esta dentro de un Map para ser diferente en cada ámbito, por lo tanto, pila.get("\$MAIN”) va a ser la pila del MAIN.

Este método apila la posición actual para ser completado ese casillero cuando finaliza la estructura (WHILE o IF). Este se llama en el método opCondición() que es llamado cuando se reduce la regla *condicion*.

```
public static void bifurcarF() {
    apilar(pos.get(Parser.AMBITO.toString()));
    polacaFuncional.get(Parser.AMBITO.toString()).add(" ");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++
    polacaFuncional.get(Parser.AMBITO.toString()).add("BF");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++
}
```

Este método hace lo mismo que bifurcar por falso pero siendo colocando BI en vez de BF.

```
public static void bifurcarI() {
    apilar(pos.get(Parser.AMBITO.toString()));
    polacaFuncional.get(Parser.AMBITO.toString()).add(" ");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++;
    polacaFuncional.get(Parser.AMBITO.toString()).add("BI");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++;
}
```

Esta función busca en la pila el valor al que debe saltar incondicionalmente (al inicio del while) y lo carga en la polaca, posterior a él, se coloca BI para indicar el salto incondicional y por último se coloca la etiqueta a saltar en caso de dar falso bifurcación de la condicion.

```
public static void bifurcarAlInicio() {
    String aux =String.valueOf(getPila());
    polacaFuncional.get(Parser.AMBITO.toString()).add(aux);
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++
    polacaFuncional.get(Parser.AMBITO.toString()).add("BI");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++
    polacaFuncional.get(Parser.AMBITO.toString()).add("LABEL"+pos.get(Parser.AMBITO.toString()));
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++
}
```

Esta función es llamada al reducir una invocación. Esta agrega a la polaca la etiqueta a la que hay que saltar (Aquí se coloca el ámbito de la función invocada) y posterior a ella se agrega un CALL para crear la instrucción de salto en la generación de assembler.

```

public static void invocar(String id) {
    System.out.println("invocacion A FUNCION");
    polacaFuncional.get(Parser.AMBITO.toString()).add(id);
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++;
    polacaFuncional.get(Parser.AMBITO.toString()).add("CALL");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++;
}

```

Esta función genera la bifurcación incondicional del GOTO. Deja un espacio en blanco antes del BI para ser completado cuando se identifique la etiqueta.

```

public static void BifurcarAGoto(String id) {
    System.out.println("BIFURCACION A TO GO CON ");
    addGoto(id);
    polacaFuncional.get(Parser.AMBITO.toString()).add(" ");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++;
    polacaFuncional.get(Parser.AMBITO.toString()).add("BI");
    pos.put(Parser.AMBITO.toString(), pos.get(Parser.AMBITO.toString())+1); //pos++;
}

```

Gramatica.y

```

private static void completarBifurcacionF() {
    int pos = GeneradorCodigoIntermedio.getPila();
    String elm = String.valueOf(GeneradorCodigoIntermedio.getPos()+2);
    GeneradorCodigoIntermedio.reemplazarElm(elm, pos);
}

```

Esta función busca la posición a completar donde se realiza una bifurcación por falso y le coloca el valor de la posición en la que va a estar colocada la etiqueta.

```

1 private static void completarBifurcacionI() {
2     int pos = GeneradorCodigoIntermedio.getPila();
3     String elm = String.valueOf(GeneradorCodigoIntermedio.getPos());
4     GeneradorCodigoIntermedio.reemplazarElm(elm, pos);
5     GeneradorCodigoIntermedio.addElemento("LABEL"+elm);
6 }

```

Aquí, se desapila la posición anterior a la bifurcación incondicional en el cuerpo del then y se le coloca el valor de la posición actual. Posterior a ella se coloca la etiqueta a la que habría que saltar cuando se lea el salto incondicional cargado recién.

```

1 private static void operacionesWhile(int cantDeOperandos) {
2     completarBifurcacionF();
3     GeneradorCodigoIntermedio.bifurcarAlInicio();
4 }

```

Este método se llama al reducir la regla *sentencia_WHILE*, por lo tanto, completa la bifurcación por falso y bifurca incondicionalmente al inicio.


```

4 private static void operacionesIF() {
5     int pos = GeneradorCodigoIntermedio.getPila();
6     String elm = String.valueOf(GeneradorCodigoIntermedio.getPos()+2);
7     GeneradorCodigoIntermedio.reemplazarElm(elm,pos);
8     GeneradorCodigoIntermedio.bifurcarI();
9     GeneradorCodigoIntermedio.addElemento("LABEL"+elm);
10 }

```

Este método se invoca en la regla de *bloque_THEN* cuando se detecta que el bloque va a ser en una sentencia IF.

Este se creó con el fin de bifurcar incondicionalmente al final cuando se haya ejecutado el cuerpo del THEN y para completar la bifurcación condicional del IF agregando la posición de la etiqueta antes del BF y agregando al final del cuerpo del THEN la etiqueta a saltar en caso de que bifurque por falso.

```

private static void completarBifurcacionISinElse() {
    int pos = GeneradorCodigoIntermedio.getPila();
    String elm = String.valueOf(GeneradorCodigoIntermedio.getPos()+2);
    GeneradorCodigoIntermedio.reemplazarElm(elm,pos);
    GeneradorCodigoIntermedio.addElemento("LABEL"+elm);
}

```

Esta función se invoca en la regla del IF cuando no existe el ELSE..

Esta únicamente carga en la posición anterior del BF la posición a la que debe saltar en caso de dar falso y además agrega la etiqueta a donde debe saltar.

```

private static void opCondicion(String operador){
    GeneradorCodigoIntermedio.addElemento(operador);
    GeneradorCodigoIntermedio.bifurcarF();
};

```

Condicion : '(' expresion_arit comparador expresion_arit ')' {opCondicion(\$3.sval);}

Este método se llama al reducir una condición para agregar el comparador y generar la bifurcación por falso.

```

private static void completarBifurcacionAGoto(String id){
    int pos = GeneradorCodigoIntermedio.getGoto(id);
    String elm = String.valueOf(GeneradorCodigoIntermedio.getPos());
    while (pos!=-1){
        GeneradorCodigoIntermedio.reemplazarElm(elm,pos);
        pos = GeneradorCodigoIntermedio.getGoto(id);
    }
    GeneradorCodigoIntermedio.addElemento("LABEL"+elm);
}

```

Sentencia_ejecutable ...

```

| ETIQUETA if(fueDeclarado($1.sval)){
completarBifurcacionAGoto($1.sval);
}else{ cargarErrorElImprimirlo("Linea :" + AnalizadorLexico.saltoDeLinea + " Error : La
ETIQUETA que se pretende bifurcar no existe. ");}}

```

Este método es llamado cuando se identifica una ETIQUETA y se verifica que se haya creado anteriormente un GOTO con ella. Como se sabe que existe, se accede al Map de Goto's de esa etiqueta. De esta manera, se recorre la pila de goto's para ir a esas posiciones y cargarlas con la posición de la Etiqueta. Ya colocada su posición se crea la etiqueta a la que se debe saltar.

```

private static void modificarPolacaPM(String operador, int cantDeOp){
    System.out.println(" Pattern Matching ");
    GeneradorCodigoIntermedio.addOperadorEnPattMatch(operador,cantDeOp);
}

```

Para su solución fuimos llevando el registro de la cantidad de expresiones que hay en una lista de expresiones, de esta manera podemos verificar que la cantidad de expresiones en una comparación sean iguales.

Si es una única expresiones se le carga al \$\$.ival un 1, si son varias, se va incrementando el valores del \$\$.ival de la lista de expresiones por cada expresión que la conforma. Adicional a esto, se agrega una ',' al final de cada variable, su uso se explica más adelante.

```

list_expre: list_expre ',' expresion_arit {$$.ival=$1.ival + 1;
GeneradorCodigoIntermedio.addElemento(",");}
| expresion_arit {$$.ival=1;GeneradorCodigoIntermedio.addElemento(",");}

```

De esta manera, podemos verificar si la cantidad de expresiones de cada lista de expresiones sean iguales y cargar una variable que va a tener la cantidad de expresiones en cada lista de expresiones. Esta variable cantDeOperandos es utilizada para completar todas las

bifurcaciones por falso de la condición. (a continuación se explica porque tenemos más de una bifurcación por falso).

```
condicion: '(' '(' list_expre ')' comparador '(' list_expre ')' ')'
    {if($3.ival == $7.ival){ cantDeOperandos=$3.ival;
    modificarPolacaPM($5.sval,$3.ival);
    }else{
    cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + "Error : Cantidad de
    operador incompatibles en la comparacion ");}}
```

El método *modificarPolacaPM* es llamado cuando en una condición existe un Pattern Matching. Esta función únicamente llama a *addOperadorEnPattMatch* para cargar de manera diferente esta condición.

Para más claridad se va a explicar mediante un ejemplo.

Condicion leida:

(a,b,c+f)>(3,f,4+6)

A la hora de reducir en una condición le llega la siguiente polaca

[... | a\$MAIN | , | b\$MAIN | , | c\$MAIN | f\$MAIN | + | , | 3 | , | f\$MAIN | , | 4 | 6 | + | , |

Nuestro objetivo es volver a leerla cargándola de la siguiente manera:

... | a | 3 | > | | BF | b | f | > | | BF | c | f | + | 4 | 6 | + | > | | BF |

De tal manera, realiza la bifurcación por falso cada vez que se lee una condición, ignorando las demás en caso de dar por falso alguna.

(a > 3) → Falso → bifurco afuera

→ Verdadero → (b>f) → Falso → bifurco afuera

→ Verdadero → (c+f>4+6) → Falso → bifurco afuera .

→ Verdadero → ignoro el BF.

Esta idea surgió dada la incapacidad del procesador de almacenar diferentes valores de flags consecutivamente.

Este método saca de la polaca cada expresiones y utiliza dos pilas auxiliares almacenando en cada una las expresiones de cada lista junto con sus ',', estas también se carga porque son las que nos van a indicar cuando termina una expresión.

Continuando con el ejemplo, las pilas quedarian asi y la polaca estaría sin rastro de la condición:

Pila derecha [, +, 6, 4, ,, f, ,, 3]

Pila izquierda [, +, f, c, ,, b, ,, a]

Posteriormente, se lee de las pilas, iniciando por la izquierda, cargando cada expresión en la polaca y cada vez que se lean las ',' de ambas pilas se carga la bifurcación, dejando a la polaca como deseábamos.

Para finalizar se modificó el método que completa las bifurcaciones por falso, desapilado la pila de la polaca por cada expresión leía mediante la variable cantDeOperandos mencionada anteriormente.

```
private static void operacionesWhile(){
    int aux=0;
```

```
        while(aux<cantDeOperandos){
            completarBifurcacionF();
            aux++;
        }
        GeneradorCodigoIntermedio.bifurcarAlInicio();
    }
private static void operacionesIF(){
    String elm = String.valueOf(GeneradorCodigoIntermedio.getPos()+2);
    int aux=0;
    while(aux<cantDeOperandos){
        completarBifurcacionF();
        aux++;
    }
    GeneradorCodigoIntermedio.bifurcarI();
    GeneradorCodigoIntermedio.addElemento("LABEL"+elm);
}
```

Problemas y consideraciones durante el desarrollo TP3

Ejecución IF dentro de WHILE

En la ejecución de un código que tenía un IF dentro de una sentencias WHILE se generaban bifurcaciones sin sentido y logramos detectar que eso sucedía porque al compartir el cuerpo del THEN con el cuerpo del WHILE se detectaba mal a que sentencia pertenecía.

Anteriormente había un boolean que se ponía en true cuando se detectaba el encabezado del WHILE, de esta manera, cuando se reducía el cuerpo_unidad, pertenecientes a ambas sentencias, se verifica si estaba en TRUE la variable esWHILE. Si así lo era no se hacía nada, en caso contrario se ejecutaba operacionesIF para generar la bifurcación incondicional al final del IF saltándose el ELSE.

Al pasar el tiempo descubrimos que al ejecutarse el IF en el WHILE se reducía el bloque_unidad como si fuese el del WHILE (porque esWHILE es TRUE) y al terminar el IF se terminaba el bloque_unidad del WHILE, reduciendo como si fuese el IF (porque ahora es FALSE).

En resumen, el IF ejecutaba el bloque_unidad como si fuese WHILE y el WHILE ejecutaba el bloque_unidad como si fuese el IF.

sentencia_IF: IF condicion THEN **bloque_unidad** ';' bloque_else ';' END_IF

encabezado_WHILE : WHILE

```
{esWHILE=true;GeneradorCodigoIntermedio.apilar(GeneradorCodigoIntermedio.getPos());GeneradorCodigoIntermedio.addElemento("LABEL"+GeneradorCodigoIntermedio.getPos());}
;
```

sentencia_WHILE: encabezado_WHILE condicion **bloque_unidad**

```
{operacionesWhile($2.ival);}
;
```

bloque_unidad: bloque_unidad_simple {

```
  if($1.sval=="RET"){ $$$.sval="RET";};
```

```
  if(esWHILE==false){operacionesIF();}else{esWHILE=false;}}
```

| bloque_unidad_multiple{

```
  if($1.sval=="RET"){ $$$.sval="RET";};
```

```
  if(esWHILE==false){operacionesIF();}else{esWHILE=false;}}
```

```
;
```

Para solucionarlo se nos ocurrió crear otra regla llamada bloque_THEN identificando los cuerpos_unidad del IF solo si inician con THEN y de esta manera se solucionó el error.

```
// WHILE
bloque_unidad: bloque_unidad_simple {if($1.sval=="RET"){$.sval="RET";}}
                | bloque_unidad_multiple {if($1.sval=="RET"){$.sval="RET";}}
;
//THEN
bloque_THEN: bloque_THEN_simple {if($1.sval=="RET"){$.sval="RET";operacionesIF();}
                | bloque_THEN_multiple {if($1.sval=="RET"){$.sval="RET";operacionesIF();}
;
;
```

Se consideró que no se pueden realizar llamados recursivos.

Al finalizar el trabajo y realizar pruebas descubrimos que si se realiza un llamado recursivo no logra verificar que el parámetro real sea igual al formal, esto se debe a la estructuración de la Gramática. Al estructurarse como se muestra en el siguiente código nos imposibilita cargar el tipo del parámetro formal sin antes reducir la función y para esto, se debe reducir el cuerpo_funcion donde está la invocación recursiva. Para solucionarlo debemos modificar la gramática y lo descubrimos tan tarde que no quisimos solucionarlo.

Declaracion_funciones

```
: encabezado_funcion parametros_parentesis BEGIN cuerpo_funcion END
    { if($4.sval!="RET"){
        cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + " Error:
        Faltan el RETORNO de al funcion ");
    }
    sacarAmbito();
    DENTRODELAMBITO.pop();
    cargarParametroFormal($1.sval,(Tipo)$2.obj);
;
encabezado_funcion : tipo FUN ID
    {$.sval=$3.sval;
    cargarVariables($3.sval,(Tipo)$1.obj," nombre de funcion ");
    agregarAmbito($3.sval);
    Simbolo.setAmbitoVar(AMBITO.toString());
    DENTRODELAMBITO.push($3.sval);
    GeneradorCodigoIntermedio.addNuevaPolaca();
;
;
```

```
parametros_parenthesis: '(' tipo_primitivo ID_simple '  
    { $$.$obj=$2.$obj;  
      GeneradorCodigoIntermedio.addElemento($3.sval + AMBITO.toString());  
      GeneradorCodigoIntermedio.addElemento("PF");  
      cargarVariables($3.sval,(Tipo)$2.$obj," nombre de parametro real ");}  
    ,
```

Generación de Código Assembler

Clase GeneraciónCodigoAssembler

Esta etapa fue programada dentro de una nueva clase, `GeneradorCodigoAssembler`. A continuación se presentarán los atributos y métodos propios de dicha clase.

Atributos

`Stack<String> pila`

Esta pila es la pila de compilación en la cuál se apilan los operandos que posteriormente serán desapilados y utilizados por los operadores.

`Int numAuxiliares`

Este atributo sirve para llevar una cuenta global de la cantidad de variables auxiliares creadas. De esta manera cuando las vaya creando voy a ir incrementando este valor.

`String ultimaFuncion`

Este atributo se utilizó para llevar un registro de cuál fue la última instrucción invocada. De esta manera se crean correctamente las variables auxiliares de retorno de función. Este tipo de variables son del estilo “@RET + ambitoFuncion” como por ejemplo:
@RET\$MAIN\$nombreFuncion.

`String ultimaOperacion`

Este atributo es utilizado únicamente al debuggear código, por lo tanto no presenta mayor relevancia en este informe.

`String ultimaTripla`

Es utilizado para saber cuál fue la última tripla a la que se le procesó el índice con el operador INDEX. De esta manera es posible saber la tripla que se ubica del lado izquierdo de una asignación.

`String ultimoIndice`

Al igual que en el caso anterior, este atributo sirve para saber cuál era la posición a ser accedida en el último procesamiento de un índice.

Métodos

recorrerPolaca

Este método cumple el rol más importante, recibir una polaca y construir el código assembler asociado a ella. Al final de la ejecución se retorna el código assembler creado.

Para poder funcionar requiere lo siguiente:

- Un `ArrayList<String>` `polacaActual` recibido como parámetro formal. Esta estructura es la polaca que debe recorrer y codificar.
- `String` `nombrePolaca` también recibido como parámetro formal. Es la key del Map de polacas asociado a la polaca que está recorriendo actualmente.
- `StringBuilder` `código`. Es la instancia de código assembler que está creando para la polaca recorrida.
- `String` `elemento`. Es un elemento extraído de la polaca, ya sea un operando o un operador.

El funcionamiento es sencillo, el método recorre la polaca y clasifica al elemento extraído.

De esta manera puede ocurrir que el elemento sea:

- **Operaciones aritméticas o una asignación.**
 - Se llama al método [operadorBinario\(elemento, código\)](#)
- **Conversiones Explícitas o RET:**
 - Se llama [operadorUnario\(elemento, código, nombrePolaca\)](#)
- **Operacion CALL:**
 - Se trata del llamado a función.
 - Se llama al método [operadorFuncion\(código\)](#)
- **Bifurcación Condicional BF:**
 - Se llama al método [operadorSaltoCondicional\(elemento, código, ultimoComparador\)](#)
- **Bifurcación Incondicional BI:**
 - Se llama al método [operadorSaltoIncondicional\(código\)](#)
- **Comparaciones:**
 - Se llama a [operadorComparacion\(elemento, código\)](#)
- **OUTF:**
 - Llama a [imprimirPorPantalla\(código\)](#)
- **Elemento PF:**
 - Usamos este elemento en especial para construir la asignación implícita del parámetro real en el parámetro formal.
 - Llama a [operadorInicioFuncion\(código\)](#)
- **INDEX:**
 - Se llama al método [operadorIndiceTripla\(código\)](#)
- **Default:**
 - En este caso entra por default si el elemento comienza con "LABEL" para poder crear la etiqueta en el assembler directamente.

- En última instancia, si el elemento entra aquí, se trata de un operando y por lo tanto lo apila en la pila de compilación.

```

public static StringBuilder recorrerPolaca(ArrayList<String> polacaActual, String nombrePolaca) {
    StringBuilder codigo = new StringBuilder();
    String elemento;
    for (int i=0; i < polacaActual.size();i++) {
        elemento = polacaActual.get(i);
        switch (elemento) {
            case "+", "-", "*", "/", ":", "=":
                ultimaOperacion = elemento;
                operadorBinario(elemento, codigo);
                break;
            case "INTEGER", "DOUBLE", "OCTAL", "RET":
                ultimaOperacion = elemento;
                operadorUnario(elemento, codigo, nombrePolaca);
                break;
            case "CALL":
                ultimaOperacion = elemento;
                operadorFuncion(codigo);
                break;
            case "BF":
                ultimaOperacion = elemento;
                operadorSaltoCondicional(elemento, codigo, polacaActual.get(i-2));
                break;
            case "BI:":
                ultimaOperacion = elemento;
                operadorSaltoIncondicional(codigo);
                break;
            case "<", ">", "<=", ">=", "=", "!=":
                ultimaOperacion = elemento;
                operadorComparacion(elemento, codigo);
                break;
            case "OUTF":
                ultimaOperacion = elemento;
                imprimirPorPantalla(codigo);
                break;
            case "PF":
                ultimaOperacion = elemento;
                operadorInicioFuncion(codigo);
                break;
            case "INDEX":
                ultimaOperacion = elemento;
                operadorIndiceTripla(codigo);
                break;
            default: //Entra si es un operando o si es un LABEL+N° que no puedo chequear en el CASE
                if(elemento.startsWith("LABEL")) { //Es una etiqueta
                    ultimaOperacion = elemento;
                    codigo.append(elemento + ": \n");
                }
                else { //Es un operando sino
                    pila.push(elemento);
                }
                break;
        }
    }
    return codigo;
}

```

operadorBinario

Este método desapila los dos operandos a utilizar, chequea compatibilidad de tipos y posterior a eso, si existe compatibilidad, llama al método correspondiente al tipo de los operandos.

Para funcionar requiere la instancia de código actualmente siendo construida, y la operación que se está ejecutando. Luego requiere dos String correspondientes a los operandos junto a dos objetos Símbolo asociado a estos.

Posterior al desapilado, se cargan los Símbolo con el símbolo encontrado en la Tabla de Símbolos dado su ámbito.

El método usado para esto es [getVariableFueraDeAmbito\(operando\)](#) creado en el Parser.

Una vez encontrado el símbolo, se sobrescriben los operandos para coincidir con la key de dicho símbolo en la tabla y poder operar correctamente.

Aquí es chequeada la compatibilidad de tipos ya que poseemos ambos operandos en simultáneo:

- En caso de ser compatibles se verifica si se tratan de triplas y (en caso de serlo) se verifica que se trate de una asignación entre triplas para operar. Caso contrario se debe emitir un error. En caso de la asignación se llama al método [operacionEntreTriplasInteger](#) u [operacionEntreTriplasFloat](#).
- En caso negativo se comprueba si la variable que almacena la última tripla no está en blanco y si se trata de una operación de asignación. De esta manera ejecuta las operaciones de asignación a una posición específica de una tripla. Aquí son llamados los métodos [operacionAsignacionElementoTriplaInteger](#) u [operacionAsignacionElementoTriplaFloat](#).
- Nuevamente, en caso negativo de lo anterior, se realiza la operación entre operandos de tipos primitivos y subtipos con los métodos [operacionEnteroOctal](#) y [operacionDouble](#).
- Finalmente, si falla la compatibilidad de tipos, se emite un error y finaliza el programa.

```

public static void operadorBinario(String operacion, StringBuilder codigo) {
    String operando2 = pila.pop();
    String operando1 = pila.pop();

    Simbolo simbOperando1 = Parser.getVariableFueraDeAmbito(operando1);
    Simbolo simbOperando2 = Parser.getVariableFueraDeAmbito(operando2);
    operando1 = simbOperando1.getId();
    operando2 = simbOperando2.getId();

    if (simbOperando1.sonCompatibles(simbOperando2)) {
        Tipo tipoOperando2 = simbOperando2.getTipo();
        Tipo tipoOperando1 = simbOperando1.getTipo();

        //CASO ASIGNACION ENTRE TRIPLAS
        if (tipoOperando2.esTripla()) {
            if (operacion == "!=") {
                if (tipoOperando2.getType().toString().contains("INTEGER") || tipoOperando2.getType().toString().contains("OCTAL")) {
                    // Solo debe permitirse la asignacion y llamo al metodo para asignar a cada elemento el del indice correspondiente
                    operacionEntreTriplasInteger(operando1, operando2, operacion, codigo, tipoOperando1, tipoOperando2);
                    ultimaTripla = "";
                } else {
                    operacionEntreTriplasFloat(operando1, operando2, operacion, codigo, tipoOperando1, tipoOperando2);
                    codigo.append("FINIT \n \n"); //Siempre vacio la pila al finalizar para evitar errores de ejecucion
                    ultimaTripla = "";
                }
            }
            else {
                Parser.cargarErrorEImprimirlo("Error: Operacion incorrecta con triplas \n");
                try {
                    AnalizadorLexico.sintactico.flush();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                System.exit(1); //Termino la ejecución del compilador por error en etapa de compilacion
            }
        }
    }
}

```

```

//CASO ASIGNACION DE TIPO PRIMITIVO A UNA POSICION DE LA TRIPLA
else if (!ultimaTripla.isBlank() && operacion=="!=") { //Reviso que lo ultimo que se haya operado haya sido una tripla para s
    operando1 = comprobarOperandoLiteral(operando1);
    if (tipoOperando2.getType().toString().contains("INTEGER") || tipoOperando2.getType().toString().contains("OCTAL")) {
        operacionAsignacionElementoTriplaInteger(operando1, codigo);
        ultimaTripla = "";
    } else {
        operacionAsignacionElementoTriplaFloat(operando1, codigo);
        codigo.append("FINIT \n \n"); //Siempre vacio la pila al finalizar para evitar errores de ejecucion
        ultimaTripla = "";
    }
}

//CASO OPERACION TIPOS PRIMITIVOS / SUBTIPOS
else {
    operando1 = comprobarOperandoLiteral(operando1);
    operando2 = comprobarOperandoLiteral(operando2);
    if (tipoOperando2.getType() == "INTEGER" || tipoOperando2.getType() == "OCTAL") {
        operacionEnteroOctal(operando1, operando2, operacion, codigo, tipoOperando2);
    }
    else {
        operacionDouble(operando1, operando2, operacion, codigo, tipoOperando2);
        codigo.append("FINIT \n \n"); //Siempre vacio la pila al finalizar para evitar errores de ejecucion
    }
}
}
else {
    try {
        AnalizadorLexico.sintactico.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.exit(1); // Termino la ejecución del compilador por error en etapa de compilacion
}
}

```

operadorFuncion

Este método recibe la instancia de código assembler que se está construyendo y opera sobre esta.

Para la ejecución desapila dos operandos (Nombre de la función y el parámetro real pasado por parámetro) y realiza la búsqueda del parámetro real en la Tabla de Símbolos.

Dependiendo del tipo del operando, se construye el código assembler que guardará el valor del parámetro real en una variable auxiliar global de assembler. De esta manera, cuando se ejecute la invocación, la función tendrá acceso garantizado al valor del parámetro real para asignarlo al parámetro formal.

Realizado lo anterior, procede a concatenar la instrucción CALL + funcion.getAmbitoVar() para poder hacer el llamado a la etiqueta asociada.

Por último, en compilación se crea la variable retorno asociada a la función llamada y es apilada para poder tenerla disponible en el momento que se quiera desapilar el retorno de la función. Si la variable ya fue creada en una invocación anterior entonces no es creada nuevamente y únicamente se apila.

```
public static void operadorFuncion(StringBuilder codigo) {
    String operando1 = pila.pop(); //Es el nombre de la funcion
    String operando2 = pila.pop(); //Es el parametro real

    Simbolo simbOperando2 = Parser.getVariableFueraDeAmbito(operando2);
    if(simbOperando2.getTipo().getType()=="INTEGER" || simbOperando2.getTipo().getType()=="OCTAL") {
        codigo.append("MOV AX, " + operando2 + "\n");
        codigo.append("MOV @ParametroRealInt, AX \n");
    }
    else if(simbOperando2.getTipo().getType()=="DOUBLE") {
        codigo.append("FLD " + operando2 + "\n"); //Apilo el parametro real
        codigo.append("FSTP @ParametroRealFloat \n");

        //-----
        codigo.append("FINT \n");
        codigo.append("FLD @ParametroRealFloat \n"); //
    }

    codigo.append("CALL " + AnalizadorLexico.TablaDeSimbolos.get(operando1).getAmbitoVar() + "\n");
    ultimaFuncion = AnalizadorLexico.TablaDeSimbolos.get(operando1).getAmbitoVar();

    String key="@RET"+ultimaFuncion;
    if(!AnalizadorLexico.TablaDeSimbolos.containsKey("@RET"+ultimaFuncion)) {
        key = crearAuxiliarRetornoFuncion(AnalizadorLexico.TablaDeSimbolos.get(operando1).getTipo());
    }
    pila.push(key); //Apilo el nombre de la variable que debera retornar la funcion
}
```

operadorSaltoCondicional

Este método recibe la instancia de código assembler, la bifurcación que se leyó y el último comparador ejecutado. Con el último comparador se codifican las instrucciones de salto adecuadas.

También obtiene la dirección a saltar al desapilar el último elemento de la pila.

Salto BF:

- Chequea que tipo de comparación fue la última realizada.
- Dependiendo de qué comparación se trate, construye las instrucciones de salto por falso.

```
public static void operadorSaltoCondicional(String elemento, StringBuilder codigo, String comparadorAnterior) {
    String operador = pila.pop(); // Es la direccion a saltar
    switch (comparadorAnterior) {
        // Le paso el operador anterior al salto para saber que comparación era y así usar el jump adecuado
        case ">":
            codigo.append("JLE LABEL" + operador + "\n\n"); // Salto en el caso contrario al de la comparacion
            break;
        case "<":
            codigo.append("JGE LABEL" + operador + "\n\n"); // Salto en el caso contrario al de la comparacion
            break;
        case ">=":
            codigo.append("JL LABEL" + operador + "\n\n"); // Salto en el caso contrario al de la comparacion
            break;
        case "<=":
            codigo.append("JG LABEL" + operador + "\n\n"); // Salto en el caso contrario al de la comparacion
            break;
        case "=":
            codigo.append("JNE LABEL" + operador + "\n\n"); // Salto en el caso contrario al de la comparacion
            break;
        case "!=":
            codigo.append("JE LABEL" + operador + "\n\n"); // Salto en el caso contrario al de la comparacion
            break;
    }
}
```

operadorSaltoIncondicional

Simplemente desapila la dirección a la que tiene que saltar y ejecuta una instrucción jump.

```
public static void operadorSaltoIncondicional(StringBuilder codigo) {
    String operador = pila.pop(); // Es la direccion a saltar
    codigo.append("JMP LABEL" + operador + "\n\n"); // Salto sí o sí a la etiqueta
}
```

operacionEnteroOctal

Al llamarse posterior al método de operadorBinario, recibe los dos operandos desapilados y preprocesados por dicho método, recibe la operación que se leyó, la instancia de código assembler y el tipo de los operandos.

Si la operación no es una asignación, entonces procede a almacenar en el registro AX el valor del operando1 para así poder realizar las operaciones aritméticas.

Aquí se verifica si se tratan de operandos subtipo, en caso de serlo se crea un código assembler especial para el chequeo de rangos, el cual se concatena luego de las operaciones normales.

Luego procede a verificar de qué operación se trata:

Operación '+':

- Se utiliza la instrucción ADD AX, operando2 para realizar la suma del valor del operando2 con el valor extraído del operando1 y almacenado en AX.

Operación '-':

- Se utiliza la instrucción SUB AX, operando2 para realizar la suma del valor del operando2 con el valor extraído del operando1 y almacenado en AX.

Operación '':**

- Se utiliza la instrucción IMUL operando2 para realizar la multiplicación del operando y AX.
- La operación anterior afecta a los flags del procesador, debido a esto codifica un salto JO a la etiqueta Overflow. Esto significa que si el flag de Overflow es activado en la multiplicación entonces debo saltar a dicha etiqueta para finalizar la ejecución del programa y emitir una notificación al usuario.

Operación '/':

- En primera instancia se asigna el valor del operando2 en el registro BX (el registro AX ya posee el valor del operando1).
- Se realiza una comparación entre el registro BX y el literal 0. Esta comparación afecta los flags del procesador.
- Se utiliza un Jump Equal a la etiqueta Division_Por_Cero en caso que la comparación entre el valor del operando2 y el 0 sean iguales.
- En dicha etiqueta se notificará al usuario del problema y finalizará el programa.
- En caso de no saltar, se codifica la instrucción IDIV para realizar la división entre operando1 y operando2.

Operación ':=':

- Debido a nuestra construcción de la polaca, en las asignaciones el orden de los operandos es incorrecto (a:=b se ejecutaría como b:=a). Es por eso que se invierten el orden de operandos.

- Luego de esto se almacena momentáneamente el valor del operando2 en el registro AX.
- Por último se asigna al operando1 el valor de AX para finalizar la operación.

Por último, para todas las operaciones diferentes a la asignación (la asignación ya fue ejecutada y finalizada correctamente), se crea una variable auxiliar para almacenar el valor del resultado de la operación y es apilada en la pila de compilación.

```
public static void operacionEnteroOctal(String operando1, String operando2, String operacion, StringBuilder codigo, Tipo tipoOperand
//Esta funcion usa el registro AX de 16 bits para enteros y octales.
StringBuilder chequeoSubtipo = new StringBuilder();
if(operacion!="=") {
    codigo.append("MOV AX, " + operando1 + "\n");
}
if(tipoOperando.esSubTipo()) {
    crearAuxiliarSubtipoInferior(AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo(), operando2);
    crearAuxiliarSubtipoSuperior(AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo(), operando2);
    chequeoSubtipo.append("CMP AX, @auxSubtipoInferior" + operando2 + "\n");
    chequeoSubtipo.append("JL Subtipo_inferior \n");
    chequeoSubtipo.append("CMP AX, @auxSubtipoSuperior" + operando2 + "\n");
    chequeoSubtipo.append("JG Subtipo_superior \n");
}

switch (operacion) {
    case "+":
        codigo.append("ADD AX," + operando2 + "\n");
        if(tipoOperando.esSubTipo()) {
            codigo.append(chequeoSubtipo);
        }
        break;
    case "-":
        codigo.append("SUB AX," + operando2 + "\n");
        if(tipoOperando.esSubTipo()) {
            codigo.append(chequeoSubtipo);
        }
        break;
    case "*":
        codigo.append("IMUL " + operando2 + "\n");
        codigo.append("JO Overflow \n"); //Salto si se activo el flag de overflow
        if(tipoOperando.esSubTipo()) {
            codigo.append(chequeoSubtipo);
        }
        break;
}
```



```
case "/":
    codigo.append("MOV BX," + operando2 + "\n"); //Uso BX para no pisar AX con el operando1
    codigo.append("CMP BX" + ",0" + "\n");
    codigo.append("JE Divison_Por_Cero \n"); //JZ salta si la comparacion del operando2 con el cero es TRUE
    //Continua el flujo normal en caso de no saltar

    if(tipoOperando.esSubTipo()) {
        codigo.append(chequeoSubtipo);
    }
    codigo.append("IDIV " + operando2 + "\n");
    break;
case "!=":
    String auxOp = operando1;
    operando1 = operando2;
    operando2 = auxOp;
    //No andaba bien asi que los intercambie para que se haga correctamente
    codigo.append("MOV AX, " + operando2 + "\n");

    if(tipoOperando.esSubTipo()) {
        codigo.append(chequeoSubtipo);
    }
    codigo.append("MOV " + operando1 + ", AX" + "\n");
    break;
default:
    //Después veré pero acá no debería entrar nada
    break;
}
if(operacion!= "!=") {
    codigo.append("MOV @aux" + numAuxiliares + ",AX");
    pila.push(crearAuxiliar(tipoOperando));
}
codigo.append("\n \n");
}
```

operacionDouble

Este método es similar a la operación con enteros/octales, sin embargo, para las operaciones flotantes es necesario utilizar el coprocesador 80x87.

Lo primero que se hace al trabajar con números flotantes es un procesado para poder ser declarados en .DATA y utilizados en .CODE. Este procesamiento se realiza con el método [convertirLexemaFlotante](#).

Este coprocesador no posee registros de memoria como los utilizados anteriormente. Es por esto que se utiliza una pila de ejecución de registros en donde se irán apilando los valores de los operandos.

Esto es realizado con la instrucción FLD y en primera instancia se apila el operando2 para tener el orden adecuado al ejecutar las operaciones.

Previo a ejecutar las operaciones, se crea la variable auxiliar para los resultados.

En cada una de las operaciones se ejecutará el método [chequearRangosSubtipoDouble](#) en caso de que se trate de una operación son subtipo.

Operación '+':

- Con la instrucción FADD se realiza la suma de ambos operandos (ambos en la pila de ejecución) y se almacena en el tope de la pila. En el tope de la pila está operando2 y debajo operando1.
- Luego con la instrucción FSTP se almacena el valor del tope de la pila (resultado anterior) en la variable auxiliar
- Se apila la variable en la pila de compilación.

Operación '-':

- Con la instrucción FSUB se realiza la resta de ambos operandos (ambos en la pila de ejecución) y se almacena en el tope de la pila. En el tope de la pila está operando2 y debajo operando1.
- Luego con la instrucción FSTP se almacena el valor del tope de la pila (resultado anterior) en la variable auxiliar.
- Se apila la variable en la pila de compilación.

Operación '*':

- Con la instrucción FMUL se realiza la resta de ambos operandos (ambos en la pila de ejecución) y se almacena en el tope de la pila. En el tope de la pila está operando2 y debajo operando1.
- Luego con la instrucción FSTP se almacena el valor del tope de la pila (resultado anterior) en la variable auxiliar.
- Se apila la variable en la pila de compilación.

Operación '/':

- Se realiza la comparación del tope con el cero.
- El resultado de esta comparación es guardado en el registro AX.

- Luego se almacenan los 8 bits menos significativos del registro de indicadores mediante la instrucción SAHF.
- Para finalizar con el chequeo se realiza un salto a Division_Por_Cero en caso de que el flag ZF esté activo (la comparación entre el operando2 y el 0 es afirmativa).
- En caso de haberse realizado el salto a la etiqueta, se notifica del problema al usuario y finaliza el programa.
- Luego se realiza la división con FDIV.
- Se guarda el resultado en la variable auxiliar.
- Se apila la variable auxiliar en la pila de compilación.

Operación ':=':

- Primero se intercambian de orden los operandos por la particularidad de la asignación
- Únicamente se asigna el valor del tope en operando2 (Recordar que por cuestiones de implementación de la polaca inversa, en las asignaciones los operandos están invertidos).

```
public static void operacionDouble(String operando1, String operando2, String operacion, StringBuilder codigo, Tipo tipoOperando) {
    operando1 = convertirLexemaFlotante(operando1);
    operando2 = convertirLexemaFlotante(operando2);
    String aux = crearAuxiliar(tipoOperando);

    //Ejecucion normal
    codigo.append("FLD " + operando1 + "\n"); //Apilo el operando1
    codigo.append("FLD " + operando2 + "\n"); //Apilo el operando2
    switch(operacion) {
        case "+":
            codigo.append("FADD" + "\n"); //ST(0) = ST(1) + ST(0)
            codigo.append("FSTP " + aux + "\n"); //Guardo el resultado en una auxiliar
            if(tipoOperando.esSubTipo()) {
                codigo.append(chequearRangosSubtipoDouble(aux, operando1));
            }
            pila.push(aux);
            break;
        case "-":
            codigo.append("FSUB" + "\n"); //ST(0) = ST(1) - ST(0)
            codigo.append("FSTP " + aux + "\n"); //Guardo el resultado en una auxiliar
            if(tipoOperando.esSubTipo()) {
                System.out.println(operando1);
                codigo.append(chequearRangosSubtipoDouble(operando1, aux));
            }
            pila.push(aux);
            break;
    }
}
```

```
case "*":
    codigo.append("FMUL" + "\n"); //ST(0) = ST(1) * ST(0)
    codigo.append("FSTP " + aux + "\n"); //Guardo el resultado en una auxiliar
    if(tipoOperando.esSubTipo()) {
        codigo.append(chequearRangosSubtipoDouble(aux, operando1));
    }
    pila.push(aux);
    break;
case "/":
    codigo.append("FTST" + "\n"); //Compare ST (operando2) con el cero
    codigo.append("FSTSW AX" + "\n");
    codigo.append("SAHF \n");
    codigo.append("JZ Divison_Por_Cero \n");
    //-----
    codigo.append("FDIV" + "\n"); //ST(0) = ST(1) / ST(0)
    codigo.append("FSTP " + aux + "\n"); //Guardo el resultado en una auxiliar
    if(tipoOperando.esSubTipo()) {
        codigo.append(chequearRangosSubtipoDouble(aux, operando1));
    }
    pila.push(aux);
    break;
case "!=":
    codigo.append("FXCH \n"); //Intercambio así realizo correctamente la operacion
    codigo.append("FSTP " + operando2 + "\n"); //Guardo el valor de operando2 en operando1 pero por cuestiones de como esta
    if(tipoOperando.esSubTipo()) {
        codigo.append(chequearRangosSubtipoDouble(operando1, operando2)); //Los cargo al reves por ser asignacion
    }
    break;
default:
    //Acá no debería entrar nada
    break;
}
```

operadorUnario

Este método recibe la instancia de código assembler, el operador que fue leído de la polaca y el nombre de la polaca que se está ejecutando actualmente (esto servirá para el retorno de funciones).

El método extrae el operando de la pila y lo busca en la Tabla de Símbolos de acuerdo a su ámbito con el método [getVariableFueraDeAmbito](#).

Se obtiene el tipo del operando para poder operar de acuerdo a este y también se verifica si se trata de un operando literal y, en caso de serlo, es convertido a no literal. Esto último es realizado con el método [comprobarOperandoLiteral](#).

Luego se crea un Símbolo retorno al cual se le asigna el encontrado en la Tabla de Símbolos correspondiente al retorno de la función.

Para poder realizar esto no es posible buscar el Símbolo @RETnombrePolaca porque el elemento nombrePolaca no es el nombre de la función como tal, sino el ámbito de la función. El nombre de la polaca real es del estilo nombreFuncion\$MAIN mientras que el ámbito de esta sería como \$MAIN\$nombreFuncion y es por esto que no es encontrado en la Tabla de Símbolos.

Para mitigar esto lo que se realizó fue una inversión del ámbito, pasando de \$MAIN\$nombreFuncion a nombreFuncion\$MAIN\$ con el método [invertirAmbito](#) y luego es buscada en la Tabla de Símbolos según el ámbito mediante [getVariableFueraDeAmbito](#). De esta manera es encontrado el símbolo correspondiente a la variable del retorno de dicha función y es posible realizar la comparación de tipos.

Luego, dependiendo de qué operación se trate, se realiza lo siguiente.

Operación “INTEGER”, “OCTAL”, “DOUBLE”:

- Estas palabras clave se utilizan para realizar una conversión explícita. Es por esto que se llama al método [operadorConversion](#).

Operación “RET”:

- Si se lee un retorno, primero chequea la compatibilidad de tipos entre el tipo que debe retornar la función y la variable que está siendo retornada.
- Luego, en caso de cumplirse la condición, verifica si se trata de una operación entera o flotante.
- En caso entero, guarda el valor del operando en el registro AX. El operando es la variable a retornar.
- Luego utiliza el nombre de la polaca que se ejecuta actualmente (una polaca que hace referencia a una función) para construir la variable retorno y asigna el resultado de AX a esta.
- En caso de ser una operación flotante, realiza la carga de la variable a guardar en la pila del coprocesador y posteriormente la guarda con FSTP.

- Las siguientes instrucciones de assembler previas al RET fueron extraídas de un documento de la cátedra para el correcto funcionamiento del retorno.
- Por último con la instrucción RET se realiza la finalización y retorno de la función.

```

public static void operadorUnario(String elemento, StringBuilder codigo, String nombrePolaca) {
    String operando = pila.pop();
    Simbolo simbOperando = Parser.getVariableFueraDeAmbito(operando);
    operando = simbOperando.getId();
    Tipo tipoOperando = AnalizadorLexico.TablaDeSimbolos.get(operando).getTipo();
    operando = comprobarOperandoLiteral(operando);
    //Lo siguiente que hago es invertir nombrePolaca que en realidad es el ambito para que
    //pase de ser, por ejemplo, $MAIN$nombrefun a nombrefun$MAIN$ y luego uso el metodo que lo busca en la TS por ambito
    Simbolo retorno = Parser.getVariableFueraDeAmbito(invertirAmbito(nombrePolaca));
    switch (elemento) {
        case "INTEGER", "OCTAL", "DOUBLE":
            operadorConversion(operando, elemento, codigo, tipoOperando);
            break;
        case "RET":
            System.out.println(nombrePolaca);
            if(retorno.sonCompatibles(simbOperando)) {
                if(retorno.getTipo().getType().contains("INTEGER")||retorno.getTipo().getType().contains("OCTAL")) {
                    codigo.append("MOV AX, " + operando + "\n"); //cuando la variable que quiero retornar en AX
                    codigo.append("MOV @RET" + nombrePolaca + ", AX" + "\n");
                }else {
                    codigo.append("FLD operando \n");
                    codigo.append("FSTP @RET" + nombrePolaca + "\n");
                }

                codigo.append("POP ESI \n");
                codigo.append("POP EDI \n");
                codigo.append("MOV ESP, EBP \n");
                codigo.append("POP EBP \n");
                codigo.append("RET" + "\n \n");
            }

        else {
            Parser.cargarErrorEImprimirlo("La variable del retorno de funcion debe ser " + retorno.getTipo().getType());
            try {
                AnalizadorLexico.sintactico.flush();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.exit(1); //Termino la ejecución del compilador por error en etapa de compilacion
        }
        break;
    }
}

```

operadorConversion

Este método recibe el operando al cual se le debe aplicar la conversión y se comprueba si se trata de un operando literal con el método [comprobarOperandoLiteral](#). En caso de serlo lo convierte en una variable.

El método realiza diferentes operaciones, dependiendo si se trata de una conversión a double o a integer.

Conversión a DOUBLE:

- Si el tipo base del operando es INTEGER u OCTAL entonces realiza la conversión de la siguiente manera.
- En primera instancia se carga el entero como un DOUBLE mediante la instrucción assembler FILD.
- Luego este resultado convertido se guarda en una variable auxiliar y es apilada en la pila de compilación, indicando que el tipo de la nueva variable es DOUBLE.

Conversión a INTEGER:

- Si el tipo base es DOUBLE entonces la conversión utiliza otro tipo de instrucciones.
- Primero se apila el operando a convertir en la pila de ejecución del coprocesador.
- Luego extrae el operando de la pila, convirtiéndolo temporalmente a entero y guardándolo en la variable auxiliar.
- Luego apila la variable auxiliar en la pila de compilación.

```
public static void operadorConversion(String operando, String elemento, StringBuilder codigo, Tipo tipoOperando) {
    operando = comprobarOperandoLiteral(operando);
    switch (elemento) {
        case "DOUBLE":
            if (tipoOperando == Parser.tipos.get("INTEGER") || tipoOperando == Parser.tipos.get("OCTAL")) {
                //Conversion de Entero/Octal a Double
                codigo.append("FILD " + operando + "\n"); //Cargo el entero como double
                codigo.append("FSTP @aux" + numAuxiliares + "\n\n"); //Almaceno el resultado en @aux
                pila.push(crearAuxiliar(Parser.tipos.get("DOUBLE")));
            }
            break;
        case "INTEGER":
            if (tipoOperando == Parser.tipos.get("DOUBLE")) {
                //Conversion de Double a Entero
                codigo.append("FLD " + operando + "\n"); //Apilo en ST
                codigo.append("FISTP @aux" + numAuxiliares + "\n\n");
                pila.push(crearAuxiliar(Parser.tipos.get("INTEGER")));
            }
            break;
        default:
            break;
    }
}
```

operadorComparacion

Este método recibe la instancia de código assembler actual y el operador que fue leído de la polaca.

El funcionamiento es muy similar al método [operadorBinario](#), extrayendo dos operandos de la pila, buscandolos según su ámbito y verificando compatibilidad de tipos previo a la construcción del assembler.

Caso INTEGER/OCTAL:

- Comprueba si ambos operandos son literales y en caso de serlos los convierte con el método [comprobarOperandoLiteral](#).
- El funcionamiento es simple, realizar la resta entre los operandos para que se activen los flags del procesador.

Caso DOUBLE:

- Primero comprueba si ambos son literales y los convierte si es afirmativo mediante el método [comprobarOperandoLiteral](#).
- Luego convierte los lexemas de los flotantes para que no haya conflictos al momento de crear las variables en la sección .DATA .
- Posterior a eso apila ambos operandos en la pila de ejecución del coprocesador.
- Compara ambos operandos extrayéndolos de la pila del coprocesador.
- Para finalizar se almacenan los 8 bits menos significativos del registro de indicadores mediante la instrucción SAHF.

```
public static void operadorComparacion(String elemento, StringBuilder codigo) {
    String operando2 = pila.pop();
    String operando1 = pila.pop();
    Simbolo simbOperando1 = Parser.getVariableFueraDeAmbito(operando1);
    operando1 = simbOperando1.getId();
    Simbolo simbOperando2 = Parser.getVariableFueraDeAmbito(operando2);
    operando2 = simbOperando2.getId();
    if( AnalizadorLexico.TablaDeSimbolos.get(operando1).sonCompatibles(AnalizadorLexico.TablaDeSimbolos.get(operando2))) {
        Tipo tipoOperando = AnalizadorLexico.TablaDeSimbolos.get(operando1).getTipo();
        if (tipoOperando.getType()=="INTEGER" || tipoOperando.getType()=="OCTAL") {
            operando1 = comprobarOperandoLiteral(operando1);
            operando2 = comprobarOperandoLiteral(operando2);
            codigo.append("MOV AX, " + operando1 + "\n");
            codigo.append("MOV BX, " + operando2 + "\n");
            codigo.append("CMP AX, " + "BX" + "\n \n"); //Compara los operandos y el resultado va a afectar a los flags para los sa
        } else if(tipoOperando.getType()=="DOUBLE") {
            operando1 = comprobarOperandoLiteral(operando1);
            operando2 = comprobarOperandoLiteral(operando2);
            operando1 = convertirLexemaFlotante(operando1); //Convierto a los lexema flotantes
            operando2 = convertirLexemaFlotante(operando2); //Convierto a los lexema flotantes
            codigo.append("FLD " + operando2 + "\n"); //Apilo el operando2
            codigo.append("FLD " + operando1 + "\n"); //Apilo el operando1
            codigo.append("FCOMPP" + "\n"); //Compara los operandos extrayendo ambos operandos de la pila
            codigo.append("FSTSW AX \n");
            codigo.append("SAHF \n ");
            codigo.append("FINIT \n \n"); //Siempre vacio la pila al finalizar para evitar errores de ejecucion
        }
    }
}
```


operadorInicioFuncion

Este método recibe únicamente la instancia de código assembler para trabajar.

El uso de este método es el de cargar el valor del parámetro real al parámetro formal de la función. De esta manera lo que se está haciendo es una asignación implícita y por eso se codifican las mismas instrucciones assembler.

Lo primero que se hace es concatenar un set de instrucciones assembler como prefijo antes de construir todas las instrucciones de la función.

Estas instrucciones son: [pop esi - pop edi - mov esp, ebp - pop ebp]

De esta manera funciona correctamente el retorno de las funciones.

Luego dependiendo del tipo del parámetro formal (desapilado de la pila de compilación) lo que se hace es mover el valor del auxiliar @ParametroRealInt o @ParametroRealFloat al operando, dependiendo del tipo del operando.

```
public static void operadorInicioFuncion(StringBuilder codigo) {
    String operando = pila.pop();
    codigo.append("PUSH EBP \n");
    codigo.append("MOV EBP, ESP \n");
    codigo.append("SUB ESP, 4 \n");
    codigo.append("PUSH EDI \n");
    codigo.append("PUSH ESI \n");
    if (AnalizadorLexico.TablaDeSimbolos.get(operando).getTipo().getType()=="INTEGER" ||
        AnalizadorLexico.TablaDeSimbolos.get(operando).getTipo().getType()=="OCTAL") {
        codigo.append("MOV AX, @ParametroRealInt \n");
        codigo.append("MOV " + operando + ", AX \n \n"); //Primero asigno el valor del ParametroRealInt cuando hago el CALL
    }
    else if (AnalizadorLexico.TablaDeSimbolos.get(operando).getTipo().getType()=="DOUBLE") {
        codigo.append("FSTP " + operando + "\n \n"); //Primero apilo el valor del ParametroRealFloat en ST cuando hago el CALL
    }
}
```

operacionEntreTriplasInteger

Este método es llamado cuando se busca asignar una tripla completa a otra.

Simplemente guarda con OFFSET la dirección de memoria del inicio de ambas triplas y ejecuta las asignaciones con su posterior desplazamiento para avanzar al siguiente elemento.

Los índices son los siguientes:

[Registro + 0] – [Registro + 2] – [Registro + 4]

```
public static void operacionEntreTriplasInteger(String operando1, String operando2, String operacion, StringBuilder codigo, Tipo tipoOperacion) {
    codigo.append("MOV ECX, OFFSET " + operando1 + "\n"); //Guardo la posición inicial del primer elemento del operando1 (valor)
    codigo.append("MOV EDX, OFFSET " + operando2 + "\n"); //Guardo la posición inicial del primer elemento del operando2 (valor)
    codigo.append("MOV AX, [ECX] \n"); //Guardo en AX el valor del primer elemento del operando1
    codigo.append("MOV BX, [EDX] \n"); //Guardo en BX el valor del primer elemento del operando2

    codigo.append("MOV [" + operando2 + "], AX \n"); //Guardo la primera pos de operando1 en operando2

    codigo.append("MOV AX, [ECX + 2] \n"); //Desplazo a la segunda pos
    codigo.append("MOV [" + operando2 + "+ 2], AX \n"); //Guardo en la segunda pos

    codigo.append("MOV AX, [ECX + 4] \n"); //Desplazo a la tercera pos
    codigo.append("MOV [" + operando2 + "+ 4], AX \n"); //Guardo en la tercera pos

    codigo.append("\n \n");
}
```

operacionEntreTriplasFloat

Sigue la misma lógica que el método anterior pero utilizando operaciones del coprocesador.

Fue necesario utilizar la palabra clave QWORD PTR para el correcto uso de las operaciones, caso contrario el compilador de assembler generaba problemas.

Aquí los índices son de la siguiente manera:

[Registro + 0] – [Registro + 8] – [Registro + 16]

```
public static void operacionEntreTriplasFloat(String operando1, String operando2, String operacion, StringBuilder codigo, Tipo tipoOperacion) {
    codigo.append("MOV ECX, OFFSET " + operando1 + "\n"); //Guardo la posición inicial del primer elemento del operando1 (valor)
    codigo.append("MOV EDX, OFFSET " + operando2 + "\n"); //Guardo la posición inicial del primer elemento del operando2 (valor)

    codigo.append("FLD QWORD PTR [ECX] \n"); //Cargo en la pila el primer valor del operando a asignar
    codigo.append("FSTP QWORD PTR [EDX] \n"); //Guardo en el primer valor del asignado

    codigo.append("FLD QWORD PTR [ECX + 8] \n"); //Cargo en la pila el segundo valor del operando a asignar
    codigo.append("FSTP QWORD PTR [EDX + 8] \n"); //Guardo en el segundo valor del asignado

    codigo.append("FLD QWORD PTR [ECX + 16] \n"); //Cargo en la pila el tercer valor del operando a asignar
    codigo.append("FSTP QWORD PTR [EDX + 16] \n"); //Guardo en el tercer valor del asignado

    codigo.append("\n");
}
```

operadorIndiceTripla

Este método se usa cuando se lee el operador INDEX en la polaca.

Se almacena el valor del índice, el nombre de la variable tripla y se busca la variable según su ámbito con el método [getVariableFueraDeAmbito\(operando2\)](#).

Dependiendo del tipo se cargan los valores de desplazamiento del índice en assembler y, en caso de ocurrir un intento de acceso a una posición diferente de 1, 2 o 3, se indica el error y finaliza la ejecución.

Por último, esto es muy importante, se almacena el índice y nombre de la variable tripla para una posible operación futura. Esto es en casos como `a{1}:= 6` porque este método genera una variable auxiliar para los casos como `a:=b{1}` pero aquí no sirve tener una auxiliar con un solo valor dentro. Guardando estos valores es posible saber luego cómo operar cuando la tripla se ubique del lado izquierdo.

```
public static void operadorIndiceTripla(StringBuilder codigo) {
    String operando1 = pila.pop(); //Es el índice a acceder
    String operando2 = pila.pop(); //Es la variable

    Simbolo simbOperando2 = Parser.getVariableFueraDeAmbito(operando2);
    operando2 = simbOperando2.getId();
    int indice = 0;

    if(simbOperando2.getTipo().getType().equals("INTEGER") || simbOperando2.getTipo().getType().equals("OCTAL")) {
        if(operando1.equals("1")) {
            indice = 0;
        } else if (operando1.equals("2")) {
            indice = 2;
        } else if (operando1.equals("3")) {
            indice = 4;
        } else {
            Parser.cargarErrorEImprimirlo("SE INTENTO ACCEDER A UNA POSICION INCORRECTA DE LA TRIPLA");
            try {
                AnalizadorLexico.sintactico.flush();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.exit(1); //Termino la ejecución del compilador por error en etapa de compilacion
        }
        codigo.append("MOV ECX, OFFSET " + operando2 + "\n");
        codigo.append("MOV AX, [ECX + " + indice + "] \n");
        codigo.append("MOV @aux" + numAuxiliares + ", AX \n \n");
    } else {
        operando2 = convertirLexemaFlotante(operando2);
        if (operando1.equals("2")) {
            indice = 8;
        }

        if (operando1.equals("3")) {
            indice = 16;
        }
        codigo.append("MOV ECX, OFFSET " + operando2 + "\n");
        codigo.append("FLD QWORD PTR [ECX + " + indice + "] \n");
        codigo.append("FSTP QWORD PTR @aux" + numAuxiliares + " \n \n");
    }
    pila.push(crearAuxiliar(Parser.tipos.get(simbOperando2.getTipo().getType())));
    ultimaTripla = operando2; //Guardo el nombre de la variable triple para el caso de asignaciones futuras
    ultimoIndice = indice; //Guardo el índice para lo mismo
}
```

operacionAsignacionElementoTriplaInteger

Este método utiliza los atributos `ultimaTripla` y `ultimoIndice` para realizar la asignación de un valor entero a una posición específica de una tripla.

Posterior a su uso, será borrado el nombre de la última tripla (luego de ser invocado el método) indicando que se finalizó la operación. De esta manera tendrá un valor diferente de vacío cuando se ejecute el operador INDEX.

```
public static void operacionAsignacionElementoTriplaInteger(String operando1, StringBuilder codigo) {
    codigo.append("MOV ECX, OFFSET " + ultimaTripla + "\n"); //Guardo la posición de inicio de la tripla a ser asignada
    codigo.append("MOV AX, " + operando1 + "\n");
    codigo.append("MOV [ECX + " + ultimoIndice + "], AX \n \n"); //Almaceno el valor del operando1 en la posición adecuada
}
```

operacionAsignacionElementoTriplaFloat

Realiza el mismo funcionamiento que el anterior método pero con operaciones del coprocesador.

Nuevamente es necesario usar las palabras clave “QWORD PTR” para poder realizar el correcto manejo de los índices en las operaciones flotantes.

```
public static void operacionAsignacionElementoTriplaFloat(String operando1, StringBuilder codigo) {
    operando1 = convertirLexemaFlotante(operando1);
    codigo.append("MOV ECX, OFFSET " + ultimaTripla + "\n");
    codigo.append("FLD QWORD PTR " + operando1 + "\n");
    codigo.append("FSTP QWORD PTR [ECX + " + ultimoIndice + "] \n \n");
}
```

imprimirPorPantalla

Este método es llamado por el operador OUTF.

Lo que hace es extraer el operando a imprimir por pantalla del tope de la pila de compilación y es buscado en la Tabla de Símbolos según su ámbito con el método

[getVariableFueraDeAmbito](#).

El tipo del operando define qué instrucción assembler se codifica.

Caso Triplas:

- Para las triplas se buscó realizar la impresión de los 3 valores en una misma línea de consola y separados por coma.
- En el caso de los enteros se tuvo que realizar una extensión de los valores y su apilado en una pila de assembler para poder realizar correctamente la impresión con printf en el formato `$_mensajeEntero$_$` → “valor1, valor2, valor3”
- Para el caso de los double, se utilizó un solo comando que imprime los valores con el mismo formato que el anterior, sin mayor complejidad.
- En cualquiera de los dos casos, están comentadas las otras opciones de impresión intentadas.

Caso INTEGER/DOUBLE:

- Los octales en assembler son tratados como enteros, simplemente tienen una base diferente.
- Para la impresión se utiliza la instrucción: “invoke printf, cfm\$(\"%hi\\n\"), operando”. De esta manera se pueden imprimir por consola enteros/octales con signo.
- Esta instrucción muestra las impresiones por consola únicamente.

Caso DOUBLE:

- En DOUBLE la instrucción es: “invoke printf, cfm\$(\"%.20Lf\\n\"), operando” . Con dicha instrucción se pueden imprimir variables flotantes con signo.
- Esta instrucción muestra las impresiones por consola únicamente.

Caso CADENAMULTILINEA:

- Para las cadenas multilínea hay que usar la siguiente instrucción:
“invoke MessageBox, NULL, addr operando, addr operando, MB_OK “
- Esta impresión no es realizada por consola, genera una ventana con el mensaje a imprimir.
Es importante saber que los mensajes generados por ventana evitan que se cierre automáticamente la consola del programa. De esta manera es posible mantener la consola abierta con los printf realizados hasta que el usuario decida cerrarla.
- **Aclaración:** Debido a las restricciones que existen en la sección .DATA hay que realizar un procesamiento en el nombre de las cadenas para poder ser declaradas y llamadas. Esto es realizado con el método [convertirLexemaCadena](#).

```

public static void imprimirPorPantalla(StringBuilder codigo) {
    String operando = pila.pop();
    Simbolo simbOperando = Parser.getVariableFueraDeAmbito(operando);

    if (simbOperando.getTipo().esTripla()) {
        operando = simbOperando.getId();
        if (simbOperando.getTipo().toString().contains("INTEGER") || simbOperando.getTipo().toString().contains("OCTAL")) {
            codigo.append("MOVZX EAX, WORD PTR [ " + operando + " + 4] \n");
            codigo.append("PUSH EAX \n");
            codigo.append("MOVZX EAX, WORD PTR [ " + operando + " + 2] \n");
            codigo.append("PUSH EAX \n");
            codigo.append("MOVZX EAX, WORD PTR [ " + operando + " + 0] \n");
            codigo.append("PUSH EAX \n");
            codigo.append("PUSH OFFSET $ _mensajeEntero$_$ \n");
            codigo.append("CALL printf \n");
            codigo.append("ADD ESP, 16 \n \n");
            //codigo.append("invoke printf, addr $_mensajeEntero$_$, DWORD PTR [ " + operando + "], DWORD PTR [ " + operando + " + 2], DW
        }
        else if (simbOperando.getTipo().toString().contains("DOUBLE")) {
            /*
            codigo.append("FLD QWORD PTR [ " + operando + " + 16] \n");
            codigo.append("SUB ESP, 8 \n");
            codigo.append("FSTP QWORD PTR [ESP] \n");
            codigo.append("FLD QWORD PTR [ " + operando + " + 8] \n");
            codigo.append("SUB ESP, 8 \n");
            codigo.append("FSTP QWORD PTR [ESP] \n");
            codigo.append("FLD QWORD PTR [ " + operando + " + 0] \n");
            codigo.append("SUB ESP, 8 \n");
            codigo.append("FSTP QWORD PTR [ESP] \n");
            codigo.append("PUSH OFFSET $ _mensajeFloat$_$ \n");
            codigo.append("CALL printf \n");
            codigo.append("ADD ESP, 24 \n \n");*/
            codigo.append("invoke printf, addr $_mensajeFloat$_$, QWORD PTR [ " + operando + "], QWORD PTR [ " + operando + " + 8], "
                + "QWORD PTR [ " + operando + " + 16] \n \n");
        }
    }
}

else if (simbOperando.getTipo().toString().contains("INTEGER") || simbOperando.getTipo().toString().contains("OCTAL")) { //tengo en c
    operando = simbOperando.getId();
    codigo.append("invoke printf, cfm$(\"%hi\\n\\n\"), " + operando + "\n \n"); //copio el invoke de _inti del archivo de moodle
}
else if (simbOperando.getTipo().toString().contains("DOUBLE")) { //tengo en cuenta primitivos y subtipos
    operando = simbOperando.getId();
    codigo.append("invoke printf, cfm$(\"%.20lf\\n\\n\"), " + operando + "\n \n"); //copio el invoke de _floati del archivo de moodle
}
else if (simbOperando.getTipo().toString().equalsIgnoreCase(Parser.tipos.get("CADENAMULTILINEA").toString()) {
    operando = convertirLexemaCadena(operando);
    codigo.append("");
    codigo.append("invoke MessageBox, NULL, addr " + operando + ", addr " + operando + ", MB_OK \n \n");
}
}
}

```

crearErrorDivisionPorCero

Lo que hace es crear la etiqueta "Division_Por_Cero:", crea un mensaje impreso en una ventana indicando el error y por último tiene un salto incondicional a la etiqueta "fin:" para finalizar el programa.

```
public static StringBuilder crearErrorDivisionPorCero() {
    StringBuilder codigo = new StringBuilder();
    codigo.append("Divison_Por_Cero:" + "\n");
    codigo.append("invoke MessageBox, NULL, addr Error_DivisionCero, addr Error_DivisionCero, MB_OK \n");
    codigo.append("JMP fin" + "\n");
    return codigo;
}
```

crearErrorOverflow

Hace lo mismo que el método [crearErrorDivisionPorCero](#), teniendo como única diferencia el nombre de etiqueta y mensaje a imprimir.

```
public static StringBuilder crearErrorDivisionPorCero() {
    StringBuilder codigo = new StringBuilder();
    codigo.append("Divison_Por_Cero:" + "\n");
    codigo.append("invoke MessageBox, NULL, addr Error_DivisionCero, addr Error_DivisionCero, MB_OK \n");
    codigo.append("JMP fin" + "\n");
    return codigo;
}
```

crearErrorSubtipoInferior

Imprime un mensaje de error por ventana, en caso de excederse en el valor menor del rango de un subtipo.

```
public static StringBuilder crearErrorSubtipoInferior() {
    StringBuilder codigo = new StringBuilder();
    codigo.append("Subtipo_inferior:" + "\n");
    codigo.append("invoke MessageBox, NULL, addr Error_Subtipo_inferior, addr Error_Subtipo_inferior, MB_OK \n");
    codigo.append("JMP fin" + "\n");
    return codigo;
}
```

crearErrorSubtipoSuperior

Imprime un mensaje de error por ventana, en caso de excederse en el valor mayor del rango de un subtipo.

```
public static StringBuilder crearErrorSubtipoSuperior() {
    StringBuilder codigo = new StringBuilder();
    codigo.append("Subtipo_superior:" + "\n");
    codigo.append("invoke MessageBox, NULL, addr Error_Subtipo_superior, addr Error_Subtipo_superior, MB_OK \n");
    codigo.append("JMP fin" + "\n");
    return codigo;
}
```

crearAuxiliar

Este método recibe únicamente el tipo que debe tener la variable al ser creada. Simplemente crea un Símbolo nuevo para ser agregado en la Tabla de Símbolos y setea los valores del auxiliar.

Todas estas variables comenzarán con el nombre @aux seguido del número correspondiente a esta nueva variable. Este número es el atributo global de la clase "numAuxiliares". Por último, previo a retornar el nombre del auxiliar, se aumenta la variable global indicando que se creó una nueva variable.

```
public static String crearAuxiliar(Tipo tipo) {
    Simbolo simb = new Simbolo();
    simb.setTipoVar(tipo);
    simb.setId("@aux" + numAuxiliares);
    simb.setUso("Var Aux");
    AnalizadorLexico.TablaDeSimbolos.put("@aux"+numAuxiliares,simb); //Agrego la nueva auxiliar a la TS
    numAuxiliares++;
    return("@aux"+(numAuxiliares-1));
}
```

crearAuxiliarParametroReal

Este método recibe un tipo por parámetro y crea la variable auxiliar utilizada en las funciones para recibir el valor del parámetro recibido en la invocación.

Dependiendo del tipo indicado, la variable tendrá un nombre o el otro.

```
public static void crearAuxiliarParametroReal(Tipo tipo) {
    Simbolo simb = new Simbolo();
    simb.setTipoVar(tipo);
    if(tipo.getType()=="INTEGER" || (tipo.getType()=="OCTAL")){
        simb.setId("@ParametroRealInt");
        simb.setUso("Var Aux ParametroRealInt");
        AnalizadorLexico.TablaDeSimbolos.put("@ParametroRealInt",simb);
    }
    else if(tipo.getType()=="DOUBLE") {
        simb.setId("@ParametroRealFloat");
        simb.setUso("Var Aux ParametroRealFloat");
        AnalizadorLexico.TablaDeSimbolos.put("@ParametroRealFloat",simb);
    }
}
```


crearAuxiliarRetornoFuncion

Al igual que los métodos de creación de variables anteriores, recibe un tipo y crea una variable auxiliar.

En este caso, el método crea una variable con nombre @RETultimaFuncion. La última función es el atributo global de la clase que lleva registro del ámbito de la función actual.

```
public static String crearAuxiliarRetornoFuncion(Tipo tipo) {
    Simbolo simb = new Simbolo();
    simb.setId("@RET"+ultimaFuncion);
    simb.setUso("Var Aux Retorno Funcion");
    simb.setTipoVar(tipo);
    String key = "@RET"+ultimaFuncion;
    AnalizadorLexico.TablaDeSimbolos.put(key,simb);
    return key;
}
```

comprobarOperandoLiteral

La única función que tiene este método es la de chequear si el operando recibido por parámetro es un literal.

A raíz de esto, en caso de ser afirmativo, procede a concatenarse un prefijo al nombre del operando según su tipo.

Esto permite su correcta declaración en .DATA y su correcta ejecución en .CODE .

```
public static String convertirOperandoLiteral(String operando) {
    switch(AnalizadorLexico.TablaDeSimbolos.get(operando).getTipo().toString()) {
        case "INTEGER", "integer", "Integer":
            operando="int"+operando;
            return operando;
        case "OCTAL", "octal", "Octal":
            operando="octi"+operando;
            return operando;
        case "DOUBLE", "double", "Double":
            operando="float"+operando;
            return operando;
    }
    return operando;
}
```

convertirLexemaFlotante

Este método soluciona errores en el assembler con caracteres conflictivos.

Recibe un operando (número flotante) y se encarga de reemplazar el '.' por un '@'.

El '+' es eliminado, esto no genera problemas porque indica el signo del exponente y es redundante.

El '-' es reemplazado por '_'.

```
public static String convertirLexemaFlotante(String operando) {
    return operando.replace('.', '@').replace("+", "").replace('-', '_');
}
```

convertirLexemaCadena

Funciona de la misma manera que el método anterior pero realizando otros reemplazos.

Debido a la naturaleza de la cadena multilínea, es necesario eliminar los espacios y los saltos de línea son reemplazados por '_'.

También son eliminados los corchetes y los símbolos '+' y '-'.

```
public static String convertirLexemaCadena(String operando) {
    return operando.replace("\r", "").replace("\n", "_").replace('.', '@').replace("+", "")
        .replace('-', '_').replace("[", "").replace("]", "").replaceAll("\\s+", "_");
}
```

invertirAmbito

Este método sirve para traer al inicio el nombre de una variable/función de manera que pase de ser AMBITO+NOMBRE a NOMBRE+AMBITO.

```
public static String invertirAmbito(String operando) {
    // Divide la cadena por el símbolo '$'
    String[] ambitos = operando.split("\\$");
    // Si hay menos de dos elementos, devolver el operando sin cambios
    if (ambitos.length < 2) {
        return operando;
    }
    String ultimoElemento = ambitos[ambitos.length - 1];
    StringBuilder resultado = new StringBuilder(ultimoElemento);
    // Agrega el resto de los elementos, precedidos por '$'
    for (int i = 0; i < ambitos.length - 1; i++) {
        if (!ambitos[i].isEmpty()) { // Evitar dobles $
            resultado.append("$").append(ambitos[i]);
        }
    }
    System.out.println(operando);
    System.out.println(resultado.toString());
    return resultado.toString();
}
```

crearAuxiliarSubtipoInferior

Crea una variable auxiliar que almacena el rango menor de un subtipo.

```
public static String crearAuxiliarSubtipoInferior(Tipo tipo, String operando) {
    Simbolo simb = new Simbolo();
    simb.setTipoVar(tipo);
    simb.setId("@auxSubtipoInferior" + operando);
    simb.setUso("Var Aux Subtipo Inferior");
    AnalizadorLexico.TablaDeSimbolos.put("@auxSubtipoInferior"+operando,simb); //Agrego la nueva auxiliar a la TS
    return("@auxSubtipoInferior"+operando);
}
```

crearAuxiliarSubtipoSuperior

Crea una variable auxiliar que almacena el rango mayor de un subtipo.

```
public static String crearAuxiliarSubtipoSuperior(Tipo tipo, String operando) {
    Simbolo simb = new Simbolo();
    simb.setTipoVar(tipo);
    simb.setId("@auxSubtipoSuperior" + operando);
    simb.setUso("Var Aux Subtipo Superior");
    AnalizadorLexico.TablaDeSimbolos.put("@auxSubtipoSuperior"+operando,simb); //Agrego la nueva auxiliar a la TS
    return("@auxSubtipoSuperior"+operando);
}
```

crearAuxiliarTripla

Crea una variable auxiliar para guardar la tripla.

```
public static String crearAuxiliarTripla(Tipo tipo) {
    Simbolo simb = new Simbolo();
    simb.setTipoVar(tipo);
    simb.setId("@auxTripla" + numAuxiliares);
    simb.setUso("Var Aux Tripla");
    AnalizadorLexico.TablaDeSimbolos.put("@auxTripla"+numAuxiliares,simb); //Agrego la nueva auxiliar a la TS
    numAuxiliares++;
    return("@auxTripla"+(numAuxiliares-1));
}
```

chequearRangosSubtipoDouble

Este método es utilizado para crear un código auxiliar para realizar el chequeo que una operación double no se haya excedido de los rangos del subtipo.

Este método es llamado (en caso de operar con subtipos) en las operaciones aritméticas de punto flotante.

```
public static StringBuilder chequearRangosSubtipoDouble(String operando1, String operando2) {
    crearAuxiliarSubtipoInferior(AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo(), AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo().getNombreSubtipo());
    crearAuxiliarSubtipoSuperior(AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo(), AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo().getNombreSubtipo());

    StringBuilder chequeoSubtipo = new StringBuilder();

    //Chequeo rango inferior
    chequeoSubtipo.append("FLD @auxSubtipoInferior" + AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo().getNombreSubtipo() + "\n"); //Apila el auxiliarSubtipoInferior
    chequeoSubtipo.append("FLD " + operando1 + "\n"); //Apila el valor del operando a asignar
    chequeoSubtipo.append("FCOMPP \n"); //Compara los operandos extrayendo ambos operandos de la pila
    chequeoSubtipo.append("FSTSW AX \n");
    chequeoSubtipo.append("SAHF \n");
    chequeoSubtipo.append("JB Subtipo_inferior \n"); //Uso JB Jump Below porque JL no funcionaba bien

    //Chequeo rango superior
    chequeoSubtipo.append("FLD @auxSubtipoSuperior" + AnalizadorLexico.TablaDeSimbolos.get(operando2).getTipo().getNombreSubtipo() + "\n"); //Apila el auxiliarSubtipoSuperior
    chequeoSubtipo.append("FLD " + operando1 + "\n"); //Apila el valor del operando a asignar
    chequeoSubtipo.append("FCOMPP \n"); //Compara los operandos extrayendo ambos operandos de la pila
    chequeoSubtipo.append("FSTSW AX \n");
    chequeoSubtipo.append("SAHF \n");
    chequeoSubtipo.append("JA Subtipo_superior \n"); //Uso JA Jump Above porque JG no funcionaba bien

    return chequeoSubtipo;
}
```

generarData

Este método genera la sección .DATA del assembler de Pentium.

Lo primero que hago es declarar los mensajes de error de división por cero y overflow, errores de subtipos, así como también el mensaje de esperar la acción del usuario y error. Este último mensaje se usa para evitar que se cierre automáticamente la consola.

También son cargados los dos tipos de formato que deben utilizar las impresiones por consola de las variables tripla.

Lo siguiente es el recorrido de la Tabla de Símbolos (Map), del cual se extrae el Lexema (Key) y el Símbolo (Value). Con esto se puede obtener el tipo del elemento.

Luego se carga el assembler de la manera adecuada según el tipo del elemento extraído.

Caso CADENAMULTILINEA:

- Primero se convierte el lexema en un String apto para ser usado como nombre en assembler.
- Luego se chequea si ya fue declarado y en caso contrario se declara guardando el lexema entre comillas.

Otros casos (diferentes de ETIQUETAS):

- Aquí se llama a un método que sirve para modularizar y construir la declaración de variables de acuerdo a su tipo. Este método es [generarDataTipos](#).
- Las variables del tipo ETIQUETA no deben ser declaradas.

```
public static StringBuilder generarData() {
    StringBuilder codigo = new StringBuilder();
    codigo.append(".data \n");
    codigo.append("Error_DivisionCero DB \"Error: Division por cero\", 10, 0 \n");
    codigo.append("Error_Overflow DB \"Error: Overflow en producto entre Enteros\", 10, 0 \n");
    codigo.append("Error_Subtipo_inferior DB \"Error: Valor menor al rango inferior del subtipo \", 10, 0 \n");
    codigo.append("Error_Subtipo_superior DB \"Error: Valor mayor al limite superior del subtipo \", 10, 0 \n");
    codigo.append("ESPERAR_ACCION_USUARIO DB \"Haga click en ACEPTAR para cerrar el programa y la consola\", 10, 0 \n");
    codigo.append("$_mensajeEntero$_$ DB \"%d, %d, %d\", 10, 0 \n");
    codigo.append("$_mensajeFloat$_$ DB \"%f, %f, %f\", 10, 0 \n");
    for (Map.Entry<String, Simbolo> iterador : AnalizadorLexico.TablaDeSimbolos.entrySet()) {
        String lexema = iterador.getKey();
        Simbolo simbolo = iterador.getValue();
        Tipo tipo = simbolo.getTipoVar();
        if (tipo != null) {
            String tipoString = tipo.toString();
            if (tipoString.contains("CADENAMULTILINEA")) {
                String lexemaConvertido = convertirLexemaCadena(lexema);
                if (codigo.indexOf(lexemaConvertido) == -1) {
                    codigo.append(lexemaConvertido).append(" db \n").append(
                        lexema.replace("\n", " ").replace("[", "").replace("]", "").replaceAll("\\s+", " ").trim()
                    ).append("\", 0 \n"); // lo hago de esta manera para poder guardar el lexema entre comillas
                }
            } else if (!tipoString.contains("ETIQUETA")) { //Si no es una cadena multilinea entonces modularizo
                generarDataTipos(codigo, tipo, lexema, simbolo);
            }
        }
    }
    return codigo;
}
```

generarDataTipos

Este método declara las variables dependiendo de su tipo.

Para las variables auxiliares de subtipos se realiza un recorte del String para averiguar a qué subtipo se le declararon los rangos y poder saber así los valores de estos.

En caso de tratarse de un tipo primitivo, se verifica si se trata de un operando literal para poder inicializarlo con un valor.

También se analiza si ya existe una declaración de dicha variable en .DATA.

También se realizan conversiones a las variables flotantes y enteras para evitar el uso de caracteres erróneos (puntos, signos, etc).

```
public static void generarDataTipos(StringBuilder codigo, Tipo tipo, String lexema, Simbolo simbolo) {
    String tipoString = tipo.toString();
    String tipoDataAssembler = null;
    String prefijoNombre = null;
    if (tipoString.contains("INTEGER")) {
        prefijoNombre = "integer";
        tipoDataAssembler = "DW";
    } else if (tipoString.contains("OCTAL")) {
        prefijoNombre = "octi";
        tipoDataAssembler = "DW";
    } else if (tipoString.contains("DOUBLE")) {
        prefijoNombre = "float";
        tipoDataAssembler = "DQ";
    }
    if (!tipo.esTripla()) { // Las triplas no pueden ser literales

        //Me fijo si se trata de un auxiliar de subtipo inferior
        if (lexema.contains("@auxSubtipoInferior")) { //Me fijo si se trata de un auxiliar de subtipo inferior
            String variable = lexema.substring(19); //Recorto la longitud del prefijo así me quedo con el nombre de la variable a la que corresponde el auxiliar
            Simbolo simbVariable = AnalizadorLexico.TablaDeSimbolos.get(variable); //Busco la variable en la TS así accedo al valor de los rangos
            if (simbVariable.getTipo().getTipo().contains("INTEGER") || simbVariable.getTipo().getTipo().contains("OCTAL")) {
                codigo.append(lexema + " " + tipoDataAssembler + " " + simbVariable.getTipo().getRangInferiorInteger() + "\n");
            } else {
                codigo.append(lexema + " " + tipoDataAssembler + " " + simbVariable.getTipo().getRangInferiorDouble() + "\n");
            }
        }

        //Me fijo si se trata de un auxiliar de subtipo superior
        else if (lexema.contains("@auxSubtipoSuperior")) {
            String variable = lexema.substring(19); //Recorto la longitud del prefijo así me quedo con el nombre de la variable a la que corresponde el auxiliar
            Simbolo simbVariable = AnalizadorLexico.TablaDeSimbolos.get(variable); //Busco la variable en la TS así accedo al valor de los rangos
            if (simbVariable.getTipo().getTipo().contains("INTEGER") || simbVariable.getTipo().getTipo().contains("OCTAL")) {
                codigo.append(lexema + " " + tipoDataAssembler + " " + simbVariable.getTipo().getRangSuperiorInteger() + "\n");
            } else {
                codigo.append(lexema + " " + tipoDataAssembler + " " + simbVariable.getTipo().getRangSuperiorDouble() + "\n");
            }
        }

        //Si no es ninguna de las dos auxiliares entonces es un entero/double común. Reviso si es literal o no.
        else if (simbolo.esLiteral()) { // Verifico primero si es una constante literal para saber si le cargo el valor o ya el '?'
            if (codigo.indexOf(prefijoNombre + lexema + " " + tipoDataAssembler) == -1) { // Me fijo si en el StringBuilder ya existe algun substring que sea intNumero DW.
                if (tipoString.contains("DOUBLE")) {
                    codigo.append(convertirLexemaFlotante(prefijoNombre + lexema + " " + tipoDataAssembler + " " + simbolo.getDouble() + "\n");
                } else {
                    //Aunque no sea double convierto el lexema para el caso de numeros enteros con signo negativo
                    codigo.append(convertirLexemaFlotante(prefijoNombre + lexema + " " + tipoDataAssembler + " " + simbolo.getEntero() + "\n");
                }
            }
        }

        //Si no es literal entonces el nombre del lexema pueda ir en .DATA
        else {
            codigo.append(lexema + " " + tipoDataAssembler + " ?" + "\n");
        }
    } else {
        codigo.append(lexema + " " + tipoDataAssembler + " ?, ?, ?" + "\n");
    }
}
```

generarCode

En esta función se recorren las polacas almacenadas en el Map de la clase `GeneradorCodigoIntermedio` y es generado el assembler de la sección `.CODE`.

Primero son creadas las variables auxiliares usadas por las funciones para asignar el valor del parámetro real en el parámetro real.

Es necesario esto porque lo siguiente que realiza este método es el recorrido y generación del código assembler de las funciones declaradas. Por lo tanto es obligatorio tener precargadas las variables retorno en la Tabla de Símbolos para evitar problemas.

Luego de haberse generado el código assembler de las funciones, se genera el código assembler del MAIN.

El código del MAIN se encuentra dentro de la etiqueta `"start:"` y posee una etiqueta final llamada `"fin:"` dentro de la cual se genera una ventana de texto que evita que se cierre automáticamente la consola, con el texto *"Haga click en ACEPTAR para cerrar el programa y la consola"* y la instrucción *"invoke ExitProcess, 0"* que finaliza la ejecución del programa.

```
public static StringBuilder generarCode() {
    Tipo tipoINTEGER = new Tipo("INTEGER");
    Tipo tipoDOUBLE = new Tipo("DOUBLE");
    crearAuxiliarParametroReal(tipoINTEGER);
    crearAuxiliarParametroReal(tipoDOUBLE);
    StringBuilder codigo = new StringBuilder();
    codigo.append(".code \n \n");
    //Key polaca main $MAIN
    //Cuando recorra el map de polacas en la parte de funciones tengo que agregar el sufijo:
    /*
    * PUSH EBP
    * MOV EBP, ESP
    * SUB ESP, 4
    * PUSH EDI
    * PUSH ESI
    */
    for (Map.Entry<String, ArrayList<String>> iterador : GeneradorCodigoIntermedio.polacaFuncional.entrySet()) {
        String ambito = iterador.getKey();
        ArrayList<String> polacaActual = iterador.getValue();

        if(ambito != "$MAIN") { //Genero el código de las polacas de funciones
            codigo.append(ambito + ": \n");

            codigo.append(recorrerPolaca(polacaActual, ambito) + "\n");
        }
    }

    //Luego de cargar las funciones en .code, recorro la polaca $MAIN
    codigo.append("start: \n");
    codigo.append(recorrerPolaca(GeneradorCodigoIntermedio.polacaFuncional.get("$MAIN"), "$MAIN"));
    codigo.append("JMP fin \n \n");
    codigo.append(crearErrorDivisionPorCero() + "\n");
    codigo.append(crearErrorOverflow() + "\n");
    codigo.append(crearErrorSubtipoInferior() + "\n");
    codigo.append(crearErrorSubtipoSuperior() + "\n");
    codigo.append("fin: \n");
    codigo.append("invoke MessageBox, NULL, addr ESPERAR_ACCION_USUARIO, addr ESPERAR_ACCION_USUARIO, MB_OK \n");
    codigo.append("invoke ExitProcess, 0 \n");
    codigo.append("end start");
    return codigo;
}
```

generarEncabezado

Construye el encabezado del programa junto con los “include” de librerías necesarias para la ejecución correcta de instrucciones.

```
public static StringBuilder generarEncabezado() {
    StringBuilder codigo = new StringBuilder();
    //Copio el encabezado de funciones.asm dado por la catedra
    codigo.append(".586 \n");
    //codigo.append(".model flat, stdcall \n \n"); //masm32rt.inc lo hace automaticamente
    codigo.append("option casemap :none \n");
    codigo.append("include \\masm32\\include\\masm32rt.inc \n");
    codigo.append("includelib \\masm32\\lib\\kernel32.lib \n");
    codigo.append("includelib \\masm32\\lib\\user32.lib \n");
    codigo.append("includelib \\masm32\\lib\\masm32.lib \n");
    codigo.append("\ndll _dllcrt0 PROTO C" + "\n");
    codigo.append("printf PROTO C : VARARG \n");
    codigo.append("\n");
    return codigo;
}
```

generarPrograma

Es la función principal que engloba los diferentes métodos que construyen el template del assembler de Pentium.

Debido a que en la generación del .CODE se agregan variables a la Tabla de Símbolos, primero se guarda el código creado en StringBuilder auxiliar.

Posterior a eso se llama a la [generarEncabezado](#) y siguiente a [generarData](#) (con la Tabla de Símbolos ya completa).

Luego se concatena el código del .CODE guardado en el auxiliar.

Para finalizar, se escribe el código generado en el archivo .txt de salida.

```
public static void generarPrograma() {
    StringBuilder codigo = new StringBuilder();
    StringBuilder seccionCode = new StringBuilder();
    seccionCode.append(generarCode()); //Genero primero el .code asi se carga correctamente la TS
    codigo.append(generarEncabezado());
    codigo.append(generarData() + "\n");
    codigo.append(seccionCode);

    CreacionDeSalidas.writeAssembler(codigo.toString());
}
```

Métodos Extra Utilizados en la Clase

Parser. getVariableFueraDeAmbito

Método ubicado en la clase Parser.

Este método busca el id pasado por parámetro (con ámbito incluido) en la Tabla de Símbolos. En caso de no encontrar el Símbolo en el primer intento, recorta los ámbitos y vuelve a realizar la búsqueda hasta encontrarlo.

De esta manera buscará dentro de su ámbito primero y luego empezará a buscar en los ámbitos superiores.

```
public static Simbolo getVariableFueraDeAmbito(String id){
    // String ambitoActual = AMBITO.toString(); // Convertimos AMBITO (StringBuilder) a String
    String key = id;
    while (true) {
        // Construimos la clave: id + mbito actual

        // Buscamos en el mapa
        if (AnalizadorLexico.TablaDeSimbolos.containsKey(key)) {
            return AnalizadorLexico.TablaDeSimbolos.get(key); // Si la clave existe, devolvemos el valor
        }

        // Reducimos el mbito: eliminamos el ltimo '$NIVEL'
        int pos = key.lastIndexOf("$");

        // Reducimos el mbito actual
        key = key.substring(0,pos);
    }
}
```


Problemas y consideraciones durante el desarrollo TP4

- Al querer imprimir por pantalla el resultado de una invocación a función tuvimos el error de no tener nada en la pila de compilación.
Esto fue solucionado apilando la variable auxiliar de retorno.
- Al usar el prefijo “int” para los valores literales tuvimos problemas con el int3 que resulta ser un nombre especial de assembler. Tuvimos que cambiar a “integer3”.
- Tuvimos errores al reconocer literales negativos, les faltaba asignar el Tipo e Id.
- Tuvimos error al declarar una variable con valor literal 0. Era reconocida como octal.
Fue cambiado a integer en la matriz de estados.
- Tuvimos un error con el GOTO cuando era el primero en ser ejecutado.
Esto ocurría debido a que utilizabamos un mismo método en caso de leer saltos condicionales e incondicionales. Los saltos condicionales necesitan acceder al operador comparador que se ejecutó dos posiciones anteriores al salto. Debido a esto, en el caso del GOTO (salto incondicional) también buscaba recibir por parámetro esa posición de la polaca, pese a no utilizarla, provocando un acceso a una posición errónea.
- El assembler nos generaba errores con el uso del ámbito en las variables mediante el símbolo ‘.’.
La única solución es usar el caracter ‘\$’ ya que es el único que admite sin dar errores.
Los ámbitos fueron modificados de manera que una variable a:MAIN se convirtió en a\$MAIN.
- De igual manera se modificaron las constantes literales agregando en el inicio un prefijo (integer, float, octi) para no tener problemas en .DATA al igual que en el resto de código assembler que no pudiera usar constantes literales.
- A los flotantes se les reemplazó el ‘.’ por un @, el signo ‘+’ se eliminó para evitar conflictos y el signo ‘-’ se reemplazó por un ‘_’
- Con las cadenas multilínea se realizó una conversión similar a las flotantes para poder guardarlas en .DATA
- Las impresiones por pantalla se realizaron con “invoke MessageBox, NULL, addr HelloWorld, addr HelloWorld, MB_OK” para que funcionen correctamente.

- Tuvimos un error en el manejo de subtipos en el cual no eran reconocidos al generar el assembler. Luego de revisar minuciosamente encontramos el error:

```

TYPEDEF TRIPLE < integer > tint;|
tint d,e;

d{1}:=2;
d{2}:=6;
e{1}:=d{3};
OUTF(e{1});
d{3}:=42;
OUTF(d{3});

e:=d;

TYPEDEF flotadito := double {-100.0, 100.0};
flotadito x,y;

y:=2.0*5.0;

```

En este caso, la asignación del subtipo 'y' no era realizada, incluso sin generar código assembler referido a operaciones double.

El error del código era el siguiente:

Como se explicó anteriormente en el documento, la clase utiliza un atributo llamado "ultimaTripla" para llevar registro de la última tripla a la que se le aplicó el operador INDEX. Una vez utilizado el valor de "ultimaTripla" era seteado nuevamente a un String en blanco para evitar conflictos y esperar a ser cargado nuevamente en otra operación INDEX.

El vaciamiento de este atributo era realizado al finalizar operaciones enteras/double del siguiente estilo a{1}:=6 por ejemplo. Pero no se tenía en cuenta realizar el vaciamiento luego de operar en una asignación entre dos triplas completas como es el caso de "e:=d" de la imagen.

Debido a esto fallaban las operaciones futuras.

Esto fue fácilmente realizando el vaciamiento como se ve en la siguiente imagen:

```

//CASO ASIGNACION ENTRE TRIPLAS
if (tipoOperando2.esTripla()) {
    if (operacion == ":=") {
        if (tipoOperando2.getType().toString().contains("INTEGER") || tipoOperando2.getType().toString().contains("OCTAL")) {
            // Solo debe permitirse la asignacion y llamo al metodo para asignar a cada elemento el del indice correspondiente
            operacionEntreTriplasInteger(operando1, operando2, operacion, codigo, tipoOperando1, tipoOperando2);
            ultimaTripla = "";
        } else {
            operacionEntreTriplasFloat(operando1, operando2, operacion, codigo, tipoOperando1, tipoOperando2);
            codigo.append("FINIT \n \n"); //Siempre vacio la pila al finalizar para evitar errores de ejecucion
            ultimaTripla = "";
        }
    }
}

```

- Descubrimos un error al intentar utilizar constantes inicializadas con cero y seguidas de un punto (0.0, 0.1, etc). Esto fue corregido en la matriz de transición de estados, considerando dicho caso y dirigiendo el flujo hacia un reconocimiento de constante Double. (Puede verse el nuevo arco del estado 7 al estado 2 con la acción semántica Concatenar)

- Tuvimos un error al intentar realizar la impresión por consola de una tripla, siguiendo el formato → “valor1, valor2, valor3”.

Se intentaron usar los siguientes comandos:

- **ENTERO:**

invoke printf, addr \$_\$mensajeEntero\$_\$, WORD PTR [variable + 0], WORD PTR [variable + 2], WORD PTR [variable + 4]

- **DOUBLE:**

invoke printf, addr \$_\$mensajeFloat\$_\$, QWORD PTR [variable + 0], QWORD PTR [variable + 8], QWORD PTR [variable + 16]

Por alguna razón, funciona correctamente en los DOUBLE pero falla en los enteros. Lo que ocurre es que imprime bien el valor de la primera posición de memoria pero falla en las otras dos. En caso de usar el formato DWORD PTR, falla en las primeras dos posiciones e imprime bien en la tercera posición.

Para solucionar el problema de los enteros, se optó por una solución bastante diferente. Lo que se hizo fue extender los valores del arreglo e irlos apilando. Luego se utiliza la función printf para imprimir esos valores en el formato \$_\$mensajeEntero\$_\$ y al finalizar se limpia la pila.

- Tuvimos un error al realizar conversiones de entero a double. Al momento de realizar la conversión y almacenarla en la variable @ParametroRealFloat, el resultado era almacenado pero a la hora de querer asignarlo al parámetro formal fallaba. Esto ocurría porque luego de haberlo cargado, no era apilado en la pila de ejecución del coprocesador.

Temas particulares

TEMA 27: Se permitirá que el tipo del parámetro real sea diferente al del parámetro formal, siempre que se anteponga al nombre del parámetro real el tipo del parámetro formal, para convertirlo al momento de la invocación.

Cuando se va a reducir la regla de una *invocación* se verifica que los parámetros sean compatibles o en caso de haber un casteo, se verifica la compatibilidad entre el Tipo y el parámetro formal.

```

invocacion : ID_simple '(' expresion_arit ')'
// Se verifica si la funcion a invocar esta declarada.
{if(!fueDeclarado($1.sval)){
// Si no se reporta un error.
print("Linea :" + AnalizadorLexico.saltoDeLinea + " Error: se invocó una funcion no declarada ");}
else{
// Se verifica si el tipo del parametro formal es igual al parametro real. (se escribio en pseudocodigo para mejor legibilidad).
if(Funcion.getTipoParFormal()==$3.sval.getTipo().getType()){
// Se carga a la POLACA la invocacion a esta funcion.
GeneradorCodigoIntermedio.invocar(AMBITO.toString()+"$"+$1.sval);
}else{
print("Linea :" + AnalizadorLexico.saltoDeLinea + " Error: Incompatibilidad de tipos entre "
+ Funcion.getTipoParFormal() + " y " + $3.sval.getTipo().getType());
}
}
}
| ID_simple '(' tipo_primitivo '(' expresion_arit ')' ')'
// Se verifica si la funcion a invocar esta declarada.
{if(!fueDeclarado($1.sval)){
print("Linea :" + AnalizadorLexico.saltoDeLinea + " Error: se invocó una funcion no declarada ");}
else{
// es caso de haber un casteo se compara con el tipo al que se va a castear.
if(AnalizadorLexico.TablaDeSimbolos.get($1.sval+AMBITO.toString()).getTipoParFormal()==((Tipo)$3.obj).getType()){
print("Linea :" + AnalizadorLexico.saltoDeLinea + " Invocacion con conversion ");
// Se carga a la POLACA la invocacion a esta funcion.
GeneradorCodigoIntermedio.invocar(AMBITO.toString()+"$"+$1.sval);
}else{
print("Linea :" + AnalizadorLexico.saltoDeLinea + " Error: Incompatibilidad de tipos entre "
+ Funcion.getTipoParFormal() + " y " + $3.sval.getTipo().getType());
}
}
}
}

```

Tema 11 y 22: Subtipos y Triplas

Cuando el compilador detecte la declaración de un nuevo tipo, definido como subrango de un tipo básico, se deberá registrar en la entrada correspondiente de la Tabla de Símbolos:

Cuando se detecte una declaración de variables del nuevo tipo, por ejemplo:

enterito a, b, c; se deberá indicar, para las entradas de las variables declaradas en la Tabla de Símbolos, que el tipo es enterito.

Semántica asociada al nuevo tipo:

(tema 27) Detectar incompatibilidad de tipos

Las variables declaradas de un tipo triple sólo podrán utilizarse en asignaciones donde el lado izquierdo y derecho sean

variables de este tipo. No debe permitirse su uso en otro tipo de sentencias / expresiones.

Los componentes de las variables declaradas con el nuevo tipo podrán ser utilizados en cualquier lugar donde pueda

utilizarse una variable, con las mismas reglas y chequeos que se consideran para variables.

Para la resolución de este tema se creó el siguiente método en la clase Tipo. Este verifica que los tipos sean iguales o en caso de ser subtipo, verifica que empiezan igual al tipo con el que te está evaluando la compatibilidad.

Recordemos que los tipos se ven de la siguiente manera:

[OCTAL, OCTAL]

[tint2, INTEGER[3]]

[flotadito2, INTEGER[6.0, 8.0]]

[flotadito, INTEGER[6.0, 8.0]]

[DOUBLE, DOUBLE]

[tint, INTEGER[3]]

[INTEGER, INTEGER]

[ETIQUETA, ETIQUETA]

Por lo tanto, el método si trata con un subTipo va a fijarse si INTEGER[6.0, 8.0] contiene a INTEGER, por lo tanto, son iguales. En caso de las triplas, solo pueden operar entre ellas (esto está indicado en el enunciado).

```
public boolean sonCompatibles(Tipo t) {
    if(this.esSubTipo() && !t.esSubTipo()) {
        return this.getType().contains(t.getType());
    }else if(!this.esSubTipo() && t.esSubTipo()) {
        return t.getType().contains(this.getType());
    }
    return this.getType()==t.getType();
}
```

Tema 13: WHILE

WHILE (<condicion>) <bloque_de_sentencias_ejecutables> ;

El bloque de sentencias ejecutables se ejecutará mientras la condición sea verdadera.

Al reconocerse el encabezado del WHILE se apila la posición a la que tiene que volver la bifurcación incondicional que va a estar al final del WHILE, agregando a la pila la etiqueta a la que tiene que saltar cuando se lea esa bifurcación.

```
encabezado_WHILE : WHILE {esWHILE=true;
GeneradorCodigoIntermedio.apilar(GeneradorCodigoIntermedio.getPos());
GeneradorCodigoIntermedio.addElemento("LABEL"+GeneradorCodigoIntermedio.getPos());}
;
```

A `$$.ival` se le asigna 1 para adaptar la estrategia utilizada con el Pattern Matching a las demás reglas de la condición. Como en el Pattern Matching a `$$.ival` se le asigna la cantidad de expresiones, esta regla le carga 1.

```
condicion :
| '(' expresion_arit comparador expresion_arit ')'
    {$$.ival=1;
    opCondicion($3.sval); }
```

Este método se llama al reducir una condición para agregar el comparador y generar la bifurcación por falso.

```
private static void opCondicion(String operador){
    GeneradorCodigoIntermedio.addElemento(operador);
    GeneradorCodigoIntermedio.bifurcarF();
};
```

Cuando se reduce el WHILE se ejecuta la función *operacionesWhile*;

```
sentencia_WHILE
: encabezado_WHILE condicion bloque_unidad
    {operacionesWhile($2.ival); }
;
```

Este método se llama al reducir la regla *sentencia_WHILE*, por lo tanto, completa la bifurcación por falso y bifurca incondicionalmente al inicio.

```
private static void operacionesWhile(int cantDeOperandos){
    completarBifurcacionF();
    GeneradorCodigoIntermedio.bifurcarAlInicio();
}
```

Tema 19: Pattern Matching

Se deberá generar código para comparar cada elemento de la izquierda con el correspondiente de la derecha, en el orden en que se presenten. Las comparaciones se combinarán, luego, con el operador lógico AND.

Por ejemplo, para:

```
if ((a,c,...) >= (b,2.3,...)) then ...
```

```
// El código a generar deberá evaluar la siguiente condición: a >= b AND c >= 2.3 AND
```

```
...
```

Se deberá chequear que el número de elementos a la izquierda sea igual al número de elementos de la derecha. En caso contrario se informará un error.

Se deberá chequear la compatibilidad de tipos en cada comparación individual, en forma independiente.

Para su solución fuimos llevando el registro de la cantidad de expresiones que hay en una lista de expresiones, de esta manera podemos verificar que la cantidad de expresiones en una comparación sean iguales.

Si es una única expresiones se le carga al `$$.ival` un 1, si son varias, se va incrementando el valores del `$$.ival` de la lista de expresiones por cada expresión que la conforma. Adicional a esto, se agrega una `,` al final de cada variable, su uso se explica más adelante.

list_expre: list_expre ',' expresion_arit {`$$.ival=$1.ival + 1;`

`GeneradorCodigoIntermedio.addElemento(",");`

`| expresion_arit {$$.ival=1;GeneradorCodigoIntermedio.addElemento(",");`

`;`

De esta manera, podemos verificar si la cantidad de expresiones de cada lista de expresiones sean iguales y cargar una variable que va a tener la cantidad de expresiones en cada lista de expresiones. Esta variable `cantDeOperandos` es utilizada para completar todas las bifurcaciones por falso de la condición. (a continuación se explica porque tenemos más de una bifurcación por falso).

condicion: '(' '(' list_expre ')' comparador '(' list_expre ')'

`{if($3.ival == $7.ival){ cantDeOperandos=$3.ival;`

`modificarPolacaPM($5.sval,$3.ival);`

`}else{`

`cargarErrorElImprimirlo("Linea " + AnalizadorLexico.saltoDeLinea + "Error : Cantidad de operandor incompatibles en la comparacion ");}}`

El método *modificarPolacaPM* es llamado cuando en una condición existe un Pattern Matching. Esta función únicamente llama a *addOperandorEnPattMatch* para cargar de manera diferente esta condición.

Para más claridad se va a explicar mediante un ejemplo.

Condicion leida:

$(a,b,c+f)>(3,f,4+6)$

A la hora de reducir en una condición le llega la siguiente polaca

[... | a\$MAIN | , | b\$MAIN | , | c\$MAIN | f\$MAIN | + | , | 3 | , | f\$MAIN | , | 4 | 6 | + | ,]

Nuestro objetivo es volver a leerla cargándola de la siguiente manera:

... | a | 3 | > | | BF | b | f | > | | BF | c | f | + | 4 | 6 | + | > | | BF |

De tal manera, realiza la bifurcación por falso cada vez que se lee una condición, ignorando las demás en caso de dar por falso alguna.

(a > 3) → Falso → bifurco afuera

→ Verdadero → (b>f) → Falso → bifurco afuera

→ Verdadero → (c+f>4+6) → Falso → bifurco afuera .

→ Verdadero → ignoro el BF.

Esta idea surgió dada la incapacidad del procesador de almacenar diferentes valores de flags consecutivamente.

Este método saca de la polaca cada expresiones y utiliza dos pilas auxiliares almacenando en cada una las expresiones de cada lista junto con sus ',', estas también se carga porque son las que nos van a indicar cuando termina una expresión.

Continuando con el ejemplo, las pilas quedarían así y la polaca estaría sin rastro de la condición:

Pila derecha [., +, 6, 4, ., f, ., 3]

Pila izquierda [., +, f, c, ., b, ., a]

Posteriormente, se lee de las pilas, iniciando por la izquierda, cargando cada expresión en la polaca y cada vez que se lean las ',' de ambas pilas se carga la bifurcación, dejando a la polaca como deseábamos.

Para finalizar se modificó el método que completa las bifurcaciones por falso, desapilado la pila de la polaca por cada expresión leía mediante la variable cantDeOperandos mencionada anteriormente.

```
private static void operacionesWhile(){
    int aux=0;
    while(aux<cantDeOperandos){
        completarBifurcacionF();
        aux++;
    }
    GeneradorCodigoIntermedio.bifurcarAllInicio();
}

private static void operacionesIF(){
    String elm = String.valueOf(GeneradorCodigoIntermedio.getPos()+2);
    int aux=0;
    while(aux<cantDeOperandos){
        completarBifurcacionF();
        aux++;
    }
    GeneradorCodigoIntermedio.bifurcarI();
    GeneradorCodigoIntermedio.addElemento("LABEL"+elm);
}
```



```

//esta funcion saca de la pila los operandos y los pasa en un arreglo auxiliar. En esta se cargan ordenado
// y se vuelven a cargar en la pila.
public static void addOperadorEnPactMatch(String operador,int cantOP) {
    int n = pos.get(Parser.AMBITO.toString())-1; // posicion en la que inicia el patter
    // Resto 1 a la posicion para pararme en el ultimo elemento cargado
    pos.put(Parser.AMBITO.toString(),pos.get(Parser.AMBITO.toString())-1);
    //agarro la ultima expresion aritmetica cargada
    String expArit = polacaFuncional.get(Parser.AMBITO.toString()).get(n);
    Stack<String> pilaIzq= new Stack<String>();
    Stack<String> pilaDer = new Stack<String>();
    // en esta cuenta las ',', hay una por cada elemento.
    int contadorComas=0;
    while (contadorComas<=cantOP*2-1) {
        //elimino el ultimo elemento cargado
        polacaFuncional.get(Parser.AMBITO.toString()).remove(n);
        if(expArit=="") {
            contadorComas++;
        }
        // Guardo las expresiones de la derecha en una pila y las de la izquierda en otra
        if(contadorComas<=cantOP) {
            pilaDer.add(expArit);
        }else {
            pilaIzq.add(expArit);
        }
        n--;
        //Tomo el siguiente elemento de la pila
        pos.put(Parser.AMBITO.toString(),pos.get(Parser.AMBITO.toString())-1);
        expArit = polacaFuncional.get(Parser.AMBITO.toString()).get(n);
    }
}

```

```

//Incremento 1 la posicion para empezar a cargar en la posicion correcta.
pos.put(Parser.AMBITO.toString(),pos.get(Parser.AMBITO.toString())+1);
contadorComas = 0;
String var;
while(contadorComas<=cantOP*2-1) {
    // Saco de la pila los primeros y la coloco en la pila
    var=pilaIzq.pop();
    while((var!=",")) {
        addElemento(var);
        var=pilaIzq.pop();
        System.out.println(var);
    }
    // Saco de la pila derecha para cargarlo en orden
    contadorComas++;
    var=pilaDer.pop();
    while((var!=",")) {
        addElemento(var);
        var=pilaDer.pop();
        System.out.println(var);
    }
    contadorComas++;
    addElemento(operador);
    // Agrego la Riferenciacion por Falso despues de cada comparacion
    apilar(pos.get(Parser.AMBITO.toString()));
    addElemento(" ");
    addElemento("BF");
}
}
}

```

Tema 23: goto

Cuando el compilador detecte un goto a una etiqueta, deberá generar en el código intermedio una bifurcación incondicional a la posición donde se encuentre la etiqueta correspondiente. Al detectar la etiqueta, se debe incorporar la etiqueta al código intermedio, ya sea como un nodo del árbol, un terceto tipo etiqueta, o un elemento de la Polaca Inversa. Se debe chequear la existencia de la etiqueta a la que se pretende bifurcar.

Cuando se lee una etiqueta, se verifica que no esté declarada en el mismo ámbito para cargarla a la tabla de símbolos y después se añade a una Lista de Etiquetas. Su función se explicará a continuación. Además de esto, se carga a la polaca con el prefijo "LABEL".

sentencia_ejecutable: ...

```
| ETIQUETA {if(fueDeclarado($1.sval+AMBITO.toString())){
cargarErrorElImprimirlo("Linea :" + AnalizadorLexico.saltoDeLinea + " Error : La ETIQUETA
"+"$1.sval+" ya existe ");
} else{ cargarVariables($1.sval,tipos.get("ETIQUETA"),"ETIQUETA");}
GeneradorCodigoIntermedio.addEtiqueta($1.sval+AMBITO.toString());
GeneradorCodigoIntermedio.addElemento("LABEL"+"$1.sval+AMBITO.toString());}
,
,
```

Cuando se lee un GOTO se carga su nombre, ámbito y posición (todo en el mismo String dividiendolo con un "/") en una lista de GOTO's.

sentencia_goto: GOTO ETIQUETA

```
{GeneradorCodigoIntermedio.addBaulDeGotos($2.sval+AMBITO.toString()+"/"+AMBITO.toStrin
g()+"/"+String.valueOf(GeneradorCodigoIntermedio.getPos()));}
,
,
```

Ahora, al reducir el código final con las dos estructuras previamente cargadas, se ejecuta el siguiente método. Este recorre la Lista de GOTO's y busca por los diferentes ámbitos la etiqueta. Cuando la encuentra va a la posición guardada en el nombre del elemento y coloca en la posición donde se ejecutó el GOTO la etiqueta a la que debe saltar

```
public class GeneradorCodigoIntermedio {
    public static ArrayList<String> Etiquetas = new ArrayList<>();
    public static ArrayList<String[]> BaulDeGotos = new ArrayList<>();
    ,
    ,
```

```
programa: ID_simple BEGIN sentencias END {cargarGotos();};
```

```
private static void cargarGotos(){
    while (!GeneradorCodigoIntermedio.BaulDeGotos.isEmpty()) {
        // Obtenemos el primer elemento
        String[] elemento = GeneradorCodigoIntermedio.BaulDeGotos.get(0);
        String key = elemento[0];
        boolean terminoWhile = false;
        boolean noHayEtiqueta = false;

        // Bucle para reducir el key si no se encuentra directamente
        while (!terminoWhile) {
            if (GeneradorCodigoIntermedio.Etiquetas.contains(key)) {
                int pos = Integer.valueOf(elemento[2]);
                GeneradorCodigoIntermedio.reemplazarElm(key, pos, elemento[1]); // Reemplaza el elemento con el método
                GeneradorCodigoIntermedio.BaulDeGotos.remove(0); // Eliminamos el primer elemento
                noHayEtiqueta = false;
                terminoWhile = true; // Terminamos el ciclo si encontramos la etiqueta
            } else {
                noHayEtiqueta = true;
            }

            // Reducimos el ámbito: eliminamos el último '$NIVEL'
            int pos = key.lastIndexOf("$");
            if (pos == -1) {
                terminoWhile = true; // Si no hay más ámbitos, salimos del ciclo
            } else {
                key = key.substring(0, pos); // Reducimos el ámbito actual
            }
        }

        // Si no se encontró etiqueta, avanzamos al siguiente elemento (va manejado por el 'while' principal)
        if (noHayEtiqueta) {
            GeneradorCodigoIntermedio.BaulDeGotos.remove(0); // Eliminamos el elemento para evitar ciclos infinitos
            cargarErrorImprimirlo("No se encontró la etiqueta llamada: " + key);
        }
    }
}
```

> Correcciones 2da entrega

	<p>Ocurrió un error al leer el archivo: test_18.txt (No such file or directory)</p> <p>>>>>> TABLA DE SIMBOLOS <<<<<</p> <p>Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.io.File.length()" because the return value of "main.java.Compilador.CreacionDeSalidas.getOutputLexico()" is null at main.java.Compilador.Main.main(Main.java:49)</p>	<p>Una vez que detectan que el archivo no existe no tienen nada más para hacer, la excepción de abajo no debería ocurrir.</p>
--	---	---

Se identificaba bien la excepcion pero no cortabamos la ejecucion, para ellos agregamos el `System.exit(1);` despues del mensaje de error

```
public static void main(String[] args) {
    // Verificar si se pasó la ruta del archivo por parámetro
    if (args.length == 0) {
        System.out.println("Debe especificar la ruta del archivo como parámetro.");
        return;
    }
    String archivoRuta = args[0]; // Obtener la ruta del archivo desde el argumento

    try {
        // Leer el contenido del archivo
        AnalizadorLexico.archivo_original = new BufferedReader(new FileReader(archivoRuta));
        CreacionDeSalidas.creacionSalidas(archivoRuta);
        //System.out.println("Se esta compilando");
        Parser par = new Parser();
        par.run();
        //System.out.println("Se compilo");
        AnalizadorLexico.lexico.flush();
        AnalizadorLexico.sintactico.flush();
    } catch (IOException e) {
        System.out.println("Ocurrió un error al leer el archivo: " + e.getMessage());
        System.exit(1);
    }
}
```

lala begin integer varx, vary, varx varx := 327 varx := 2.3d+34; tipo_abc vary; end	Exception in thread "main" java.lang.NullPointerException: Cannot invoke "main.java.Compilador.Tipo.getType()" because "tipo" is null at main.java.Compilador.Parser.cargarVariables(Parser.java:1068) at main.java.Compilador.Parser.yyparse(Parser.java:1384) at main.java.Compilador.Parser.run(Parser.java:2009) at main.java.Compilador.Main.main(Main.java:35)	
---	--	--

Cuando se declaraba una variable de un tipo que no era definido hasta el momento, se cargaba el error y continuando su ejecución. Al continuar, cuando se cargaba la variable de este tipo desconocido le llevaba el tipo “null” entonces se rompía. Para esto preguntamos si es null para no cargar la variable a la tabla de simbolos.

lala begin goto lld; end	Linea :4 Error: Falta la etiqueta en GOTO Linea 4: Erro: Falta ';' al final de la sentencia	Mal identificado.
---	--	-------------------

En este caso, consideramos que estaba mal construida la etiqueta, faltando el @ al final del lld. Es por esto que primero se reconoce el error de la etiqueta faltante y por consecuencia, el error de la falta de ‘;’ puesto a que reconoce a la sentencia “goto *espacioEnBlanco*” y considera que falta el ‘;’ al final de esta.

Para solucionar esta confusión agregamos la siguiente regla para que reduzca todo como etiqueta e indique la falta de ‘@’.

```

sentencia_goto:GOTO ETIQUETA
| GOTO ID_simple

```

```
{cargarErrorElImprimirlo("Linea :" + AnalizadorLexico.saltoDeLinea + " Error: Falta el caracter  
'@' de la etiqueta. ");}  
| GOTO error  
{cargarErrorElImprimirlo("Linea :" + AnalizadorLexico.saltoDeLinea + " Error: Falta la etiqueta en  
GOTO ");}  
;
```