Compilador MiniLeng

Procesadores de lenguajes

2019-2020, Grado en ingeniería informática

760704 Álvaro García Díaz

Índice

1.- Análisis léxico

Para empezar con la creación del compilador de MiniLeng, hay que definir los tokens que se van a emplear en el lenguaje. Cada token ha sido definido para una función o elemento del lenguaje. Dichos tokens son los siguientes:

Token	Carácter/es	Función
< tPROGRAMA >	programa	Especifica el inicio del programa
< tPRINCIPIO >	principio	Indica el inicio de una acción o del programa
< tFIN >	fin	Indica el final de una acción o del programa
< tSI >	si	Especifica una condición para realizar una función (selección)
< tENT >	ent	Indica el inicio de una selección o de un bucle
< tSI_NO >	si_no	Especifica otra selección si la anterior no se ha cumplido
< tFSI >	fsi	Indica el final de una selección
< tMQ >	mq	Especifica el comienzo de un bucle si se cumple una condición
< tFMQ >	fmq	Indica el final de un bucle
< tACCION >	accion	Especifica el comienzo de la declaración de una acción
< tPCOMA >	;	Se usa para especificar el final de una línea o una sucesión de diferentes tipos de parámetros
< tCOMA >	,	Se usa para una sucesión de variables
< tENTACAR >	entacar	Indica una función para pasar de entero a carácter
< tCARAENT >	caraent	Indica una función para pasar de carácter a entero
< tESCRIBIR >	escribir	Indica una función para mostrar por pantalla un elemento
< tLEER >	leer	Indica una función para leer de la entrada estándar
< tENTERO >	entero	Indica el tipo de una variable (número)
< tVAR >	var	Definido pero sin ningún uso
< tBOOLEANO >	booleano	Indica el tipo de una variable (booleano)

< tCARACTER >	caracter	Indica el tipo de una variable (carácter)
< tVAL >	val	Indica el tipo de un parámetro (por valor)
< tREF >	ref	Indica el tipo de un parámetro (por referencia)
< tNI >	<>	Operador relacional que indica desigualdad
< tMAI >	>=	Operador relacional que indica superior o igual a
< tMEI >	<=	Operador relacional que indica inferior o igual a
< tMAYOR >	>	Operador relacional que indica superior a
< tMENOR >	<	Operador relacional que indica inferior a
< tIGUAL >	=	Operador relacional que indica igualdad
< tNOT >	not	Operador relacional que indica contrariedad
< tAND >	and	Operador relacional que indica la función AND
< tTRUE >	true	Hecho verdadero
< tFALSE >	false	Hecho falso
< tPAREN_IZQ >	(	Paréntesis abierto
< tPAREN_DCHA >	)	Paréntesis cerrado
< tSUMA >	+	Operador aditivo que indica suma de dos elementos
< tRESTA >	-	Operador aditivo que indica resta de dos elementos
< tOR >	or	Operador aditivo que indica la función OR
< tMUL >	*	Operador multiplicativo que indica multiplicación de dos elementos
< tDIV >	div | /	Operador multiplicativo que indica división de dos elementos
< tMOD >	mod	Operador multiplicativo que indica módulo de dos elementos
< tOPAS >	:=	Indica una asignación
< DIGITO >	[“0”-”9”]	Número
< LETRA >	[“a”-”z”]	Letra
< COMILLA >	[“\””]	Comillas dobles

Hay 3 tokens que han sido definidos mediante una expresión regular:

Token	Regex	Función
< tCONSTCHAR >	< COMILLA > (~["\""]) < COMILLA >	Un solo carácter
< tCONSTCAD >	< COMILLA > (~["\""])+ < COMILLA >	Una cadena
< tCONSTENTERA >	(< DIGITO >)+	Uno o más números
< tIDENTIFICADOR >	< LETRA > | (< LETRA > | "_") (< LETRA > | "_" | < DIGITO >)* (< LETRA > | < DIGITO >)	Nombre de una variable

La expresión anterior del identificador permite variables cuyo nombre sea una sola letra o estén formados al principio por una letra o por una barra baja, a continuación de 0 a varias letras, dígitos o de barras bajas y al final de una letra o de un dígito (por ejemplo: a, _a o a3a).

Existen una serie de caracteres que no se tienen en cuenta en las expresiones regulares, que son definidos como SKIP:

" "	Espacio
"\r"	Retorno de carro
"\t"	Tabulador
"\n"	Salto de línea

Para incluir comentarios en el código, se permiten tanto de una línea o de varias. Los de una línea empiezan con un solo % y los de varías líneas por dos % seguidos. Para saber cuando han terminado los comentarios, para el de una línea acaba cuando encuentra un salto de línea y para el de varias líneas, termina cuando vuelve a encontrar dos % seguidos.

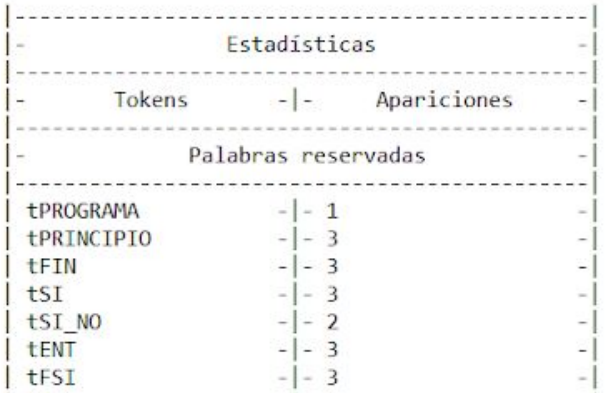
Cuando se detecta un token no reconocido, se para la ejecución y muestra un mensaje de error indicando la fila y la columna del token y el carácter o cadena no reconocido:
ERROR LÉXICO (<13, 18>): Símbolo no reconocido: <Â>

1.1.- Tabla de estadísticas

Con la finalidad de saber cuántas veces aparece un determinado un token, se ha implementado una tabla que almacena todas las apariciones de cada token. Dicha tabla al principio se inicializa en 0 en todas sus posiciones.

Cuando se detecta alguno de los tokens, se aumenta en uno su posición en la tabla. La tabla se trata de una tabla hash para mejorar la eficiencia al no tener que buscar su posición cada vez.

Para mostrar la tabla, solo hace falta poner como argumento "-v" y mostrará la tabla al final del programa con el siguiente formato:



```
|-------------------------------------------|
|-              Estadísticas                -|
|-------------------------------------------|
|-       Tokens       -|-     Apariciones    -|
|-------------------------------------------|
|-            Palabras reservadas            -|
|-------------------------------------------|
| tPROGRAMA           -|- 1                  -|
| tPRINCIPIO          -|- 3                  -|
| tFIN                -|- 3                  -|
| tSI                 -|- 3                  -|
| tSI_NO              -|- 2                  -|
| tENT                -|- 3                  -|
| tFSI                -|- 3                  -|
```

Los tokens se encuentran agrupados en diferentes apartados para encontrarlos con mayor facilidad. A continuación se muestran algunos ejemplos de los apartados:

- Palabras reservadas: tPRINCIPIO o tACCION
- Operaciones: tENTACAR o tLEER
- Tipos de datos: tENTERO o tVAL
- Operadores lógicos: tMEI o tTRUE
- Operadores aritméticos: tSUMA o tPAREN_DCHA
- Valores: tIDENTIFICADOR o tCONSTCAD

1.2.- Pruebas realizadas

Para comprobar que se detectan todos los tokens deseados, se han creado 5 pruebas, en las que si se detecta un error léxico, se terminará la ejecución. Se han omitido los posibles fallos sintácticos o semánticos que pueden salir en las salidas al compilar las pruebas. Estas pruebas son:

- prueba1.ml: En la primera prueba, se comprueba el funcionamiento de los comentarios dobles, comentando el nombre del programa y una selección. Se destaca que puede haber líneas dentro del comentario, que sin el comentario provocaría un error léxico. Salida:
  ERROR LÉXICO (<24, 26>): Símbolo no reconocido: <.>
- prueba2.ml: En esta prueba, se han declarado una serie de variables, cuyos identificadores contienen números y dígitos, que serán considerados como constante entera e identificador. Compila correctamente
- prueba3.ml: Se ha introducido un carácter de otro idioma (ยก) para forzar el error. Salida:
  ERROR LÉXICO (<13, 18>): Símbolo no reconocido: <Â>
- prueba4.ml: Da fallo debido al uso de comillas simples. Salida:

ERROR LÉXICO (<9, 18>): Símbolo no reconocido: <'>

- prueba5.ml: Se intenta asignar a una variable un número con decimales. Detecta el . como carácter no identificado. Salida:
  ERROR LÉXICO (<14, 16>): Símbolo no reconocido: <.>

2.- Análisis sintáctico

El siguiente paso en la creación del compilador es el análisis sintáctico, que consiste en la creación de expresiones regulares. El BNF de cada función es el siguiente:

Nombre de la función	Expresión regular
punto_y_coma	<tPCOMA>
valid_id	<tIDENTIFICADOR>
programa	<tPROGRAMA> valid_id punto_y_coma declaracion_variables declaracion_acciones bloque_sentencias
declaracion_variables	(declaracion punto_y_coma)*
declaracion	tipo_variables identificadores
tipo_variables	(<tENTERO> | <tBOOLEANO> | <tCARACTER>)
identificadores	valid_id (<tCOMA> valid_id)*
declaracion_acciones	(declaracion_accion)*
declaracion_accion	cabecera_accion punto_y_coma declaracion_variables declaracion_acciones bloque_sentencias
cabecera_accion	<tACCION> valid_id parametros_formales
parametros_formales	(<tPAREN_IZQ> parametros (punto_y_coma parametros)* <tPAREN_DCHA>)?
lista_parametros	valid_id (<tCOMA> valid_id)*
parametros	clase_parametros tipo_variables lista_parametros
clase_parametros	(<tVAL> | <tREF>)
bloque_sentencias	<tPRINCIPIO> lista_sentencias <tFIN>
lista_sentencias	(sentencia)+
sentencia	(leer punto_y_coma | escribir punto_y_coma | asig_acc | seleccion | mientras_que)
asig_acc	valid_id (asignacion | invocacion_accion) punto_y_coma

leer	<tLEER> <tPAREN_IZQ> lista_asignables <tPAREN_DCHA>
lista_asignables	valid_id (<tCOMA> valid_id)*
escribir	<tESCRIBIR> <tPAREN_IZQ> lista_escribibles <tPAREN_DCHA>
lista_escribibles	escribibles (<tCOMA> escribibles)*
escribibles	(<tCONSTCAD> | expresion)
asignacion	<tOPAS> expresion
invocacion_accion	(argumentos)?
mientras_que	<tMQ> expresion lista_sentencias <tFMQ>
seleccion	<tSI> expresion <tENT> lista_sentencias (<tSI_NO> lista_sentencias)? <tFSI>
argumentos	<tPAREN_IZQ> (lista_expresiones)? <tPAREN_DCHA>
lista_expresiones	expresion (<tCOMA> expresion)*
expresion	expresion_simple (operador_relacional expresion_simple)?
operador_relacional	(<tNI> | <tMAI> | <tMEI> | <tMAYOR> | <tMENOR> | <tIGUAL>)
expresion_simple	termino (operador_aditivo termino)*
operador_aditivo	(<tSUMA> | <tRESTA> | <tOR>)
termino	factor (operador_multiplicativo factor)*
operador_multiplicativo	(<tDIV> | <tMUL> | <tMOD> | <tAND>)
factor	(<tRESTA> factor | <tNOT> factor | valid_id | <tPAREN_IZQ> expresion <tPAREN_DCHA> | <tENTACAR> <tPAREN_IZQ> expresion <tPAREN_DCHA> | <tCARAENT> <tPAREN_IZQ> expresion <tPAREN_DCHA> | <tCONSTENTERA> | <tCONSTCHAR> | <tTRUE> | <tFALSE>)

En la función de selección solo hay poner un único <tFSI> por cada <tSI> y no por cada <tSI> y <tSI_NO>. No se permite la declaración de identificadores con el mismo nombre que las palabras reservadas (como principio o programa) debido a que detecta dichas palabras como un token diferente a tIDENTIFICADOR.

En el análisis sintáctico también se pueden producir errores cuando no se cumple alguna de las expresiones regulares anteriores. Cuando eso ocurre, muestra la fila y la columna en la que se ha detectado un token no esperado:
ERROR SINTÁCTICO (<11, 5>): < Símbolo encontrado incorrecto: escribir >

A diferencia del análisis léxico, cuando ocurre un error, se continúa la ejecución, permitiendo ver todos los fallos sintácticos posibles. Esto se consigue recogiendo todos los tokens hasta llegar a uno que sea <tPCOMA> (que indica el final de línea mayoritariamente) o EOF (fin de fichero, porque sino podría ocurrir un bucle infinito).

Una mejora opcional añadida ha sido el *panic mode*, que consiste en la supresión de errores si son comunes. En este caso, se suprime el error cuando falta algún <tPCOMA> al final de línea. La diferencia que posee en comparación con el error sintáctico es que no avanza a otro token hasta encontrar un <tPCOMA> o EOF, es decir, muestra un mensaje de información y continúa la ejecución:
PANIC MODE: Falta un ; en <12, 6>

2.1.- Pruebas realizadas

Para comprobar el funcionamiento del semántico, se han creado 6 pruebas, en la que cada una tiene una finalidad diferente. Es posible que debido a un error sintáctico, aparezcan otros errores sintácticos que se solucionen corrigiendo el primero. En las salidas de las pruebas, solo se muestran los errores sintácticos. A continuación, se muestran los errores que deben ocurrir en cada prueba o por qué no ha detectado ningún fallo y la salida del compilador:

- prueba1.ml: Salen errores por la creación de variables que poseen el mismo nombre que las palabras reservadas, pero se permiten las cadenas con la palabra reservada entre comillas. Salida:
  ERROR SINTÁCTICO (<5, 8>): < Símbolo encontrado incorrecto: caracter >
  ERROR SINTÁCTICO (<8, 1>): < Símbolo encontrado incorrecto: Principio >
  ERROR SINTÁCTICO (<11, 18>): < Símbolo encontrado incorrecto: ent >
  ERROR SINTÁCTICO (<13, 9>): < Símbolo encontrado incorrecto: escribir >
  ERROR SINTÁCTICO (<15, 18>): < Símbolo encontrado incorrecto: principio >
  ERROR SINTÁCTICO (<17, 9>): < Símbolo encontrado incorrecto: Si >
  ERROR SINTÁCTICO (<19, 9>): < Símbolo encontrado incorrecto: si_no >

  Se han detectado un total de 7 errores sintacticos
- prueba2.ml: Aparecen errores de variables que tendrían que haber sido definidas antes que el token <tPRINCIPIO> y después se prueban diferentes argumentos para escribir y leer. Para leer, sale error cuando se introduce un elemento diferente a un identificador. Salida:
  ERROR SINTÁCTICO (<9, 17>): < Símbolo encontrado incorrecto: entero >
  ERROR SINTÁCTICO (<15, 22>): < Símbolo encontrado incorrecto: "ERROR" >
  ERROR SINTÁCTICO (<16, 17>): < Símbolo encontrado incorrecto: leer >
  ERROR SINTÁCTICO (<17, 17>): < Símbolo encontrado incorrecto: leer >
  ERROR SINTÁCTICO (<18, 22>): < Símbolo encontrado incorrecto: entacar >
  ERROR SINTÁCTICO (<19, 17>): < Símbolo encontrado incorrecto: leer >
  ERROR SINTÁCTICO (<20, 17>): < Símbolo encontrado incorrecto: leer >
  ERROR SINTÁCTICO (<25, 34>): < Símbolo encontrado incorrecto: "bien" >
  ERROR SINTÁCTICO (<26, 1>): < Símbolo encontrado incorrecto: Fin >

ERROR SINTÁCTICO (<26, 5>): < Símbolo encontrado incorrecto:
>
ERROR SINTÁCTICO (<26, 5>): < Símbolo encontrado incorrecto:
>
ERROR SINTÁCTICO (<26, 5>): < Símbolo encontrado incorrecto:
>

Se han detectado un total de 12 errores sintacticos

- prueba3.ml: Se trata de una prueba sin errores en el que se prueban diferentes combinaciones en las condiciones de los bucles
- prueba4.ml: En esta prueba se intentan diferentes combinaciones de expresiones, en las que da error si no se añaden paréntesis a cada expresión si está compuesta por al menos 2 expresiones o si se concatenan dos operadores relacionales. Salida:
  ERROR SINTÁCTICO (<14, 22>): < Símbolo encontrado incorrecto: > >
  ERROR SINTÁCTICO (<17, 30>): < Símbolo encontrado incorrecto: > >

  Se han detectado un total de 2 errores sintacticos
- prueba5.ml: Se evalúan una serie de acciones, en las que da error si les falta especificar de qué tipo son los parámetros. Salida:
  ERROR SINTÁCTICO (<40, 23>): < Símbolo encontrado incorrecto: entero >
  ERROR SINTÁCTICO (<40, 54>): < Símbolo encontrado incorrecto: booleano >
  ERROR SINTÁCTICO (<42, 2>): < Símbolo encontrado incorrecto: principio >
  ERROR SINTÁCTICO (<45, 2>): < Símbolo encontrado incorrecto: fin >
  ERROR SINTÁCTICO (<51, 1>): < Símbolo encontrado incorrecto: Fin >
  ERROR SINTÁCTICO (<51, 5>): < Símbolo encontrado incorrecto:
  >
  ERROR SINTÁCTICO (<51, 5>): < Símbolo encontrado incorrecto:
  >
  ERROR SINTÁCTICO (<51, 5>): < Símbolo encontrado incorrecto:
  >

  Se han detectado un total de 8 errores sintacticos
- pruebaPanic.ml: Última prueba del análisis sintáctico para probar el apartado opcional del *panic mode*. Con el *panic mode* desactivado, ocurren errores debido a que faltan ; al final de las líneas, pero si se activa, no muestra ningún error. Salida con el *panic mode* activado:
  PANIC MODE: Falta un ; en <6, 1>
  PANIC MODE: Falta un ; en <9, 3>
  PANIC MODE: Falta un ; en <12, 6>
  PANIC MODE: Falta un ; en <14, 6>
  PANIC MODE: Falta un ; en <16, 6>
  PANIC MODE: Falta un ; en <19, 3>
  PANIC MODE: Falta un ; en <22, 2>
  PANIC MODE: Falta un ; en <28, 1>

Salida sin el *panic mode*:
ERROR SINTÁCTICO (<6, 1>): < Símbolo encontrado incorrecto: accion >
ERROR SINTÁCTICO (<6, 31>): < Símbolo encontrado incorrecto: ref >

Se han detectado un total de 2 errores sintacticos

3.- Tabla de símbolos

El siguiente paso es la creación de una tabla de símbolos para detectar fallos semánticos (siguiente punto). Para ello, hay que definir una clase Símbolo con los siguientes atributos:

- nombre: String que indica el identificador que tiene el símbolo
- nivel: Nivel en el que se encuentra
- tipo: Indica de qué tipo es el símbolo(Programa, acción, variable o parámetro)
- variable: Indica de qué tipo es la variable si el tipo es variable o parámetro (Entero, carácter, booleano, cadena o desconocido)
- parametro: Indica de qué tipo es el parámetro si el tipo es parámetro (Valor o referencia)
- lista_parametros: Lista que contiene todos los parámetros si el tipo es una acción
- dir: Dirección de la pila en la que se encuentra el valor del símbolo
- visible: Indica si el símbolo es visible en la tabla

La clase Símbolo también incluye una serie de funciones para facilitar la creación de diferentes tipos de símbolos, comparar parámetros, saber si es de un tipo, clase o variable determinado y un getter y setter para cada atributo:

- introducir_programa(String name, Long d): Introduce un programa con el nombre name en la dirección d en el nivel 0
- introducir_accion(String name, int level, Long d): Introduce una acción con el nombre name en la dirección d en el nivel level
- introducir_variable(String name, Tipo_variable var, int level, Long d): Introduce una variable de tipo var con nombre name en la dirección d en el nivel level
- introducir_parametro(String name, Tipo_variable var, Clase_parametro par, int level, Long d): Introduce un parámetro de tipo var de clase par con nombre name en la dirección d en el nivel level
- compararParametros(Simbolo s): Compara los parámetros y si son iguales (en cada posición, mismo tipo, clase y variable), devuelve true, en caso contrario, devuelve false
- es_variable(): Devuelve true si es una variable
- es_parametro(): Devuelve true si es un parámetro
- es_accion(): Devuelve true si es una acción
- es_desconocido(): Devuelve true si el tipo es desconocido
- es_entero(): Devuelve true si el tipo es entero
- es_booleano(): Devuelve true si el tipo es booleano
- es_caracter(): Devuelve true si el tipo es carácter

- es_cadena(): Devuelve true si el tipo es cadena
- es_valor(): Devuelve true si la clase del parámetro es valor
- es_referencia(): Devuelve true si la clase del parámetro es referencia
- setters y getters: Para obtener o modificar los atributos

Para implementar la tabla de símbolos, también ha hecho falta un TAD tabla de dispersión para almacenar los símbolos en una tabla de colisiones abierta para obtener los símbolos en el menor tiempo posible (en algunos casos es constante).

Por último, se ha añadido el argumento "-t" para mostrar la tabla de estadísticas después de detectar un token <tFIN> menos en el del programa (para eso está el argumento -v) para verificar el correcto funcionamiento.

3.1.- Tabla de dispersión

Para implementar la tabla de dispersión, se ha escogido un valor M primo cercano a una potencia de 2 (M = 127 ~2^7) para disminuir las colisiones.

Para la función hash, se ha empleado la de Pearson, 90 ya que es específica para cadenas de texto de longitud variable y genera un hash a partir del nombre del símbolo. Para ello, ha hecho falta un vector T de tamaño 255 cuyos valores son una permutación de los números entre 0 y 255.

Al añadir un símbolo a la tabla, se calcula el hash de su nombre para añadirlo en la posición obtenida y se añade al principio, haciendo que si hay dos símbolos con el mismo nombre, se obtenga el símbolo que ha sido añadido más reciente (que será el que posea un nivel más alto).

Las funciones que contiene son las siguientes:

- nuevoSimbolo(Simbolo s): Calcula el hash del nombre de s y lo añade en la tabla en la posición obtenida al principio
- buscarSimbolo(String n): Devuelve el primer símbolo encontrado con el nombre n. Si no encuentra ninguno, lanza una excepción SimboloNoEncontradoException
- existeSimbolo(String n, int l): Busca si existe un simbolo con nombre n en el nivel l. Si encuentra uno, lanza una excepción SimboloExistenteException
- eliminarNivel(int n, Tipo_simbolo t): Elimina los símbolos pertenecientes al nivel n y de tipo t
- ocultarParametros(int n): Pone a false el atributo visible de los símbolos de tipo parámetro del nivel n
- eliminarParametrosOcultos(int n): Elimina los símbolos de tipo parámetro cuyo atributo es false y se encuentran en el nivel n

3.2.- Funciones de la tabla de símbolos

Las funciones definidas en la tabla de símbolos hacen uso de las funciones propias de la tabla de dispersión:

- buscar_simbolo(String nombre): Equivalente a la función buscarSimbolo de la tabla de dispersión
- introducir_programa(String nombre, Long dir): Crea un símbolo mediante la función introducir_programa de la clase Símbolo y lo añade a la tabla mediante nuevoSimbolo de la tabla de dispersión
- introducir_variable(String nombre, Tipo_variable variable, int nivel, Long dir): Crea un símbolo mediante la función introducir_variable de la clase Símbolo y lo añade a la tabla mediante nuevoSimbolo de la tabla de dispersión (si no se ha lanzado una excepción SimboloExistenteException)
- introducir_accion(String nombre, int nivel, Long dir): Crea un símbolo mediante la función introducir_accion de la clase Símbolo y lo añade a la tabla mediante nuevoSimbolo de la tabla de dispersión (si no se ha lanzado una excepción SimboloExistenteException)
- introducir_parametro(String nombre, Tipo_variable variable, Clase_parametro parametro, int nivel, Long dir): Crea un símbolo mediante la función introducir_parametro de la clase Símbolo y lo añade a la tabla mediante nuevoSimbolo de la tabla de dispersión (si no se ha lanzado una excepción SimboloExistenteException)
- eliminar_programa(): Ejecuta la función eliminarNivel(0, PROGRAMA) de la tabla de dispersión para eliminar del nivel 0 el símbolo PROGRAMA
- eliminar_variables(int nivel): Ejecuta la función eliminarNivel(nivel, VARIABLE) de la tabla de dispersión para eliminar del nivel nivel los símbolos de tipo variable
- eliminar_acciones(int nivel): Ejecuta la función eliminarNivel(nivel + 1, PARAMETRO) de la tabla de dispersión para eliminar del nivel nivel los símbolos de tipo parametro, que se tratan de los parametros de la acción del nivel nivel. Después se vuelve a ejecutar la función eliminarNivel(nivel, ACCION) para eliminar el símbolo de la acción
- ocultar_parametros(int nivel): Ejecuta la función ocultarParametros(nivel) de la tabla de dispersión
- eliminar_parametros_ocultos(int nivel): Ejecuta la función eliminarParametrosOcultos(nivel) de la tabla de dispersión

4.- Análisis semántico

El último análisis a realizar es el semántico, en el que se van a analizar errores en los que no coinciden los tipos de las variables en la asignación, en los argumentos o en las operaciones, los distintos desbordamientos que se pueden dar (de enteros y ASCII), divisiones y módulos de 0, identificadores no encontrados (añadidos posteriormente con el tipo DESCONOCIDO para no mostrar más errores del mismo tipo) o duplicados, falta o exceso de parámetros en llamadas a acciones, argumentos de tipo valor

introducidos en parámetros de tipo referencia, condiciones no booleanas, variables no leíbles o asignación de algún valor a un parámetro de valor.

Para las expresiones, ha sido necesario propagar el valor para conocer el tipo de parte de la operación e ir haciendo comprobaciones de tipos. Esta necesidad preciso de la creación de la clase RegistroExpr. Dicha clase contiene un atributo para cada uno de sus posibles tipos (Entero, booleano o carácter) para indicar su valor, su tipo, su clase si es un parámetro, el nivel en el que se encuentra y la dirección de la pila. Las funciones que posee son las siguientes:

- operacion(RegistroExpr r2, Tipo_operador type): Realiza la operación type entre el RegistroExpr actual y el r2 si sus tipos coinciden (o al menos uno es DESCONOCIDO) y si sus tipos son compatibles con la oeración propuesta (por ejemplo un entero no puede realizar la operación AND). Puede lanzar las siguientes excepciones: ModuloDeCeroException, DivisionEntreCeroException, TiposDiferentesException o DesbordamientoEnterosException
- tipoCorrecto(Tipo_operador tipo): Evalúa si el tipo del RegistroExpr es compatible con el operador tipo
- operacionOk(RegistroExpr r): Evalúa si los dos RegistroExpr son compatibles para realizar una operación
- operacionOk(Tipo_variable r): Equivalente a la anterior
- desbordamientoEnteros(RegistroExpr r, Tipo_operador t): Evalúa si la operación entre los RegistroExpr puede provocar un desbordamiento de enteros
- desbordamientoEnteros(long aux): Evalúa si una constante entera se encuentra fuera del rango de los enteros
- Setters y getters: Para modificar u obtener los atributos de la clase

Cuando se detecta algún error, se muestra una línea que indica la fila y la columna en la que ha ocurrido el fallo junto a un texto descriptivo que depende del error causado. A continuación, se muestra un ejemplo de cada tipo junto a una explicación sobre en qué casos puede ocurrir:

- ERROR SEMANTICO (<31, 59>): < Los tipos ENTERO y CHAR no concuerdan >: Puede aparecer cuando se realiza una operación y los dos operandos no coinciden en el tipo de variable
- ERROR SEMANTICO (<32, 25>): < La condicion del bucle debe ser un booleano >: Aparece cuando la condición del bucle no se trata de un booleano
- ERROR SEMANTICO (<35, 34>): < La condicion de la seleccion debe ser un booleano >: Aparece cuando la condición de una selección no es un booleano
- ERROR SEMANTICO (<46, 32>): < Tipos incompatibles en la asignacion (ENTERO, BOOLEANO) >: Aparece cuando en una asignación se intenta cambiar el valor del símbolo por otro de tipo distinto al del símbolo
- ERROR SEMANTICO (<52, 21>): < Detectado modulo de 0 >: Aparece cuando se detecta un módulo de 0
- ERROR SEMANTICO (<40, 19>): < Detectada division entre 0 >: Aparece cuando se detecta una división entre 0

- ERROR SEMANTICO (<31, 25>): < Detectado desbordamiento ASCII >: Aparece cuando un entero se intenta pasar a carácter mediante ENTACAR y el valor del entero es inferior a 0 o superior a 255
- ERROR SEMANTICO (<22, 29>): < Detectado desbordamiento de enteros >: Aparece cuando se detecta una constante entera o en un resultado de una operación que está fuera de rango (-2147483648, 2147483647)
- ERROR SEMANTICO (<29, 9>): < Identificador a no encontrado >: Aparece cuando se ha buscado el identificador en la tabla y no se ha encontrado. Después, se añade el identificador como tipo DESCONOCIDO para que no aparezca el mismo error más de una vez
- ERROR SEMANTICO (<48, 14>): < Variable b no leible >: Aparece cuando una variable no es de tipo entero, carácter o desconocido y no es una variable o un parámetro de referencia (función es_leible() de la clase Símbolo)
- ERROR SEMANTICO (<88, 13>): < Exceso de parametros en la llamada a la accion a1 >: Aparece cuando se excede el número de parámetros la invocación a alguna acción
- ERROR SEMANTICO (<82, 7>): < Faltan argumentos en la llamada a la accion a1 >: Aparece cuando falta algún parámetro por introducir en la invocación a una acción
- ERROR SEMANTICO (<79, 10>): < Argumento de tipo BOOLEANO no valido (se esperaba ENTERO) >: Aparece cuando se introduce una variable de un tipo en un parámetro de otro tipo diferente
- ERROR SEMANTICO (<70, 7>): < Argumento de tipo VAL introducido en parametro de tipo REF >: Aparece cuando se detecta una variable de tipo valor de una acción introducida en la invocación a otra acción en un parámetro de tipo referencia
- ERROR SEMANTICO (<16, 7>): < No se puede realizar una asignacion a un parametro por valor >: Aparece cuando se introduce se intenta asignar un valor a un parámetro de tipo valor

4.1.- Vectores

Para la mejora opcional de los vectores, se ha tenido que hacer una serie de cambios en algunas clases y análisis, que se detallan a continuación:

- Análisis léxico: Se han añadido dos tokens para los corchetes que se emplean para especificar el tamaño de los vectores o el índice de un vector. Dichos tokens son < tCOR_DCHA > ( ]) y < tCOR_IZQ > ([ ). También se han añadido a la tabla de estadísticas en la sección Operadores aritméticos.
- Análisis sintáctico: Se han cambiado algunas expresiones regulares para adaptarlas a la inclusión de vectores, que son las siguientes:

Función	Expresión regular	Comentarios
identificadores	valid_id (<tCOR_IZQ> <tCONSTENTERA> <tCOR_DCHA>)? (<tCOMA> valid_id (<tCOR_IZQ> <tCONSTENTERA>	Se ha hecho que al definir un vector, sea siempre una

	<tCOR_DCHA>)?)*	constante entera su tamaño
asignacion	(<tCOR_IZQ> expresion <tCOR_DCHA>)? <tOPAS> expresion	
factor	(<tRESTA> factor | <tNOT> factor | <tPAREN_IZQ> expresion <tPAREN_DCHA> | <tENTACAR> <tPAREN_IZQ> expresion <tPAREN_DCHA> | <tCARAENT> <tPAREN_IZQ> expresion <tPAREN_DCHA> | valid_id (<tCOR_IZQ> expresion <tCOR_DCHA>)? | <tCONSTENTERA> | <tCONSTCHAR> | <tTRUE> | <tFALSE>)	

- Tabla de símbolos: Se ha modificado la clase Símbolo con la introducción de un nuevo atributo que indica el tamaño de un vector (Integer tam), que será nulo en los demás casos. Se ha creado una función (introducir_vector) para facilitar la creación de vectores tanto en la clase Símbolo como en la tabla de símbolos.
- Análisis semántico: Se han creado nuevos mensajes de error para controlar los desbordamientos underflow y overflow de los vectores y de un índice no entero:
  ERROR SEMANTICO (<9, 23>): < Vector mal definido sin tamano >: Ocurre cuando se ha definido un vector con tamaño 0
  ERROR SEMANTICO (<19, 19>): < Desbordamiento de underflow del vector a >: Ocurre cuando se intenta acceder a una posición inferior a 0 en un vector
  ERROR SEMANTICO (<23, 15>): < Desbordamiento de overflow del vector c >: Ocurre cuando se intenta acceder a una posición superior o igual al tamaño del vector
  ERROR SEMANTICO (<30, 22>): < Indice del vector b de tipo diferente a entero >: Ocurre cuando se intenta acceder a una posición del vector con un índice de un tipo diferente a entero
  ERROR SEMANTICO (<33, 15>): < Variable a2 no es un vector >: Ocurre cuando se intenta acceder a una posición de un símbolo que no es un vector
  Se han creado dos nuevas pruebas para comprobar el funcionamiento de los vectores:
  - pruebaVector1.ml: Prueba correcta en el que se prueba a asignar valores a posiciones de vectores, mostrarlos por pantalla y ejecutar una acción con un parámetro de tipo valor y otro de tipo referencia que imprime por pantalla esos dos parámetros.
  - pruebaVector2.ml: Prueba incorrecta en el que se evalúan los diferentes errores que se pueden dar como un vector con tamaño 0, posiciones fuera del rango, una posición no entera y una posición de un símbolo que no es un vector. Salida:
    ERROR SEMANTICO (<9, 23>): < Vector mal definido sin tamano >

ERROR SEMANTICO (<18, 15>): < Desbordamiento de underflow del vector a >
ERROR SEMANTICO (<19, 19>): < Desbordamiento de underflow del vector a >
ERROR SEMANTICO (<23, 15>): < Desbordamiento de overflow del vector c >
ERROR SEMANTICO (<24, 18>): < Desbordamiento de overflow del vector c >
ERROR SEMANTICO (<28, 17>): < Indice del vector b de tipo diferente a entero >
ERROR SEMANTICO (<29, 16>): < Indice del vector b de tipo diferente a entero >
ERROR SEMANTICO (<30, 22>): < Indice del vector b de tipo diferente a entero >
ERROR SEMANTICO (<33, 15>): < Variable a2 no es un vector >
ERROR SEMANTICO (<36, 9>): < Identificador v no encontrado >

Se han detectado un total de 10 errores semanticos

4.2.- Excepciones empleadas

Para controlar diferentes errores que pueden ocurrir, se han implementado una serie de excepciones:

- DesbordamientoEnterosException: Se puede lanzar cuando un entero supera el límite máximo de enteros permitido en JAVA (2147483647) o es inferior al mínimo (-2147483648). Aparece cuando una constante entera o alguna operación cuyo resultado esté fuera del rango permitido
- DivisionPorCeroException: No se puede dividir entre 0, por eso cuando se detecta una división entre 0, se lanza esta excepción
- ModuloDeCeroException: No se puede calcular el módulo de 0, ya que requiere de una división entre 0, por eso se lanza esta excepción
- TiposDiferentesException: Se lanza cuando se realiza alguna operación o asignación con dos símbolos cuyo tipo de variable difiere
- FileNotFoundException: Excepción de JAVA empleada para indicar que no se ha introducido ningún fichero a compilar o que no es leíble

4.3.- Pruebas realizadas

Para comprobar el funcionamiento del análisis semántico se han creado las siguientes pruebas (solo se muestran los errores semánticos):

- prueba1.ml: Se prueban diferentes combinaciones de comparaciones y de asignaciones para provocar fallos de tipos diferentes y de condiciones no booleanas. Salida:
  ERROR SEMANTICO (<31, 29>): < Los tipos ENTERO y CHAR no concuerdan >
  ERROR SEMANTICO (<31, 44>): < Los tipos BOOLEANO y ENTERO no concuerdan >

ERROR SEMANTICO (<31, 44>): < Los tipos ENTERO y BOOLEANO no concuerdan >
ERROR SEMANTICO (<31, 59>): < Los tipos CHAR y BOOLEANO no concuerdan >
ERROR SEMANTICO (<31, 59>): < Los tipos ENTERO y CHAR no concuerdan >
ERROR SEMANTICO (<32, 25>): < La condicion del bucle debe ser un booleano >
ERROR SEMANTICO (<35, 34>): < Los tipos ENTERO y CHAR no concuerdan >
ERROR SEMANTICO (<35, 34>): < La condicion de la seleccion debe ser un booleano >
ERROR SEMANTICO (<40, 25>): < Tipos incompatibles en la asignacion (CHAR, ENTERO) >
ERROR SEMANTICO (<46, 32>): < Tipos incompatibles en la asignacion (ENTERO, BOOLEANO) >

Se han detectado un total de 10 errores semanticos

- prueba2.ml: Se prueba los diferentes desbordamientos que puede ocurrir (enteros y ASCII) y las divisiones y módulos de 0. Salida:
  ERROR SEMANTICO (<13, 14>): < Detectado desbordamiento de enteros >
  ERROR SEMANTICO (<16, 15>): < Detectado desbordamiento de enteros >
  ERROR SEMANTICO (<19, 26>): < Detectado desbordamiento de enteros >
  ERROR SEMANTICO (<22, 29>): < Detectado desbordamiento de enteros >
  ERROR SEMANTICO (<25, 30>): < Detectado desbordamiento de enteros >
  ERROR SEMANTICO (<31, 25>): < Detectado desbordamiento ASCII >
  ERROR SEMANTICO (<34, 24>): < Detectado desbordamiento ASCII >
  ERROR SEMANTICO (<40, 19>): < Detectada division entre 0 >
  ERROR SEMANTICO (<46, 21>): < Detectada division entre 0 >
  ERROR SEMANTICO (<52, 21>): < Detectado modulo de 0 >

  Se han detectado un total de 10 errores semanticos
- prueba3.ml: Se comprueba que al añadir un símbolo de tipo DESCONOCIDO, no vuelve a aparecer un mensaje de error indicando que no existe y diferentes argumentos para la función leer. Un detalle a destacar es que aparece un error sintáctico al introducir una expresión entera como argumento en la función leer. Salida:
  ERROR SEMANTICO (<20, 9>): < Identificador m no encontrado >
  ERROR SEMANTICO (<29, 9>): < Identificador a no encontrado >
  ERROR SEMANTICO (<48, 14>): < Variable b no leible >
  ERROR SEMANTICO (<51, 14>): < Variable sin no leible >

  Se han detectado un total de 4 errores semanticos
- prueba4.ml: Se trata de una prueba sin errores creada para probar diferentes combinaciones de la operación escribir y en el apartado siguiente mostrar “Verdadero” o “Falso” cuando se quiera imprimir por pantalla un booleano.
- prueba5.ml: Se prueban diferentes llamadas a acciones con parámetros de tipo valor y referencia para provocar errores. También se provoca un error sintáctico al no contener ninguna instrucción entre el principio y el fin del programa. Salida:

ERROR SEMANTICO (<14, 8>): < Variable b no leible >
ERROR SEMANTICO (<67, 5>): < No se puede realizar una asignacion a un parametro por valor >
ERROR SEMANTICO (<70, 7>): < Argumento de tipo VAL introducido en parametro de tipo REF >
ERROR SEMANTICO (<70, 10>): < Argumento de tipo VAL introducido en parametro de tipo REF >
ERROR SEMANTICO (<76, 7>): < Argumento de tipo VAL introducido en parametro de tipo REF >
ERROR SEMANTICO (<79, 10>): < Argumento de tipo BOOLEANO no valido (se esperaba ENTERO) >
ERROR SEMANTICO (<82, 7>): < Faltan argumentos en la llamada a la accion a1 >
ERROR SEMANTICO (<85, 13>): < Exceso de parametros en la llamada a la accion a1 >
ERROR SEMANTICO (<88, 13>): < Exceso de parametros en la llamada a la accion a1 >

Se han detectado un total de 9 errores semanticos

5.- Generación de código

Como última parte, se genera el código para que sea ejecutable. Para ello, se ha creado una clase GenerarCodigo que se encarga de crear el fichero si no ha ocurrido ningún error. También genera las etiquetas para saltar a ciertas partes del código.

Para escribir el fichero, se tiene una variable de tipo String que va almacenando todas las cadenas del código y que al finalizar todos los análisis, se escribe en el fichero .code.

Para facilitar la escritura en el String, se ha implementado una función que a partir de un operador, genera su código.

Se ha optado por un esquema secuencial debido a que permitía escribir el código en cada función y que, debido al análisis semántico, no se podía devolver un String en algunas funciones.

5.1.- Pruebas realizadas

Para comprobar que el código generado funciona, se han empleado las pruebas entregadas por el profesorado correctas (nyn1.ml, nprimos1.ml, fib.ml, adivinar1.ml y mcd1.ml) más la prueba4.ml y pruebaVector1.ml del análisis semántico. Para ejecutar el código de cada fichero, primero hay que compilarlo con el minilengcompiler para generar el .code, después ejecutar el script ensamblador con el .code para generar el .x y al final ejecutar interprete con el .x. La salida de cada uno se muestra a continuación:

- nyn1:

```
hendrix01:~/pleng/ ./interprete nyn1
n: 100
n: 100
n: 400
n: 10
10 se convierte en 99

Terminaci▒n normal.
```

- nprimos1:

```
hendrix01:~/pleng/ ./interprete nprimos1
2es primo.
3es primo.
4es primo.
5es primo.
7es primo.
11es primo.
13es primo.
17es primo.
19es primo.
23es primo.
29es primo.
31es primo.
37es primo.
41es primo.
43es primo.
47es primo.
53es primo.
59es primo.
61es primo.
67es primo.
71es primo.
73es primo.
79es primo.
83es primo.
89es primo.
97es primo.

Terminaci▒n normal.
```

- fib:

```
hendrix01:~/pleng/ ./interprete fib
Escribe un numero: 5
Fibbonaci(5) es: 5

Terminaci▒n normal.
hendrix01:~/pleng/ ./interprete fib
Escribe un numero: 10
Fibbonaci(10) es: 55

Terminaci▒n normal.
```

- adivinar1:

```
hendrix01:~/pleng/ ./interprete adivinar1
Piensa en una letra e intentare adivinarla.
Listo?(S/N)?S
(A,Z): has pensado en la letra M?(S/N)?(S/N)?N
La letra que has pensado es mayor?(S/N)?(S/N)?N
(A,L): has pensado en la letra F?(S/N)?(S/N)?S

Terminaci▒n normal.
```

- mcd1:

```
hendrix01:~/pleng/ ./interprete mcd1
Escribe un numero: 10
Escribe un numero: 35
El MCD es: 5

Terminaci▒n normal.
```

- prueba4:

```
hendrix01:~/pleng/ ./interprete prueba4
Verdadero
Falso
Verdadero
5
10
16
g
y
bien

Terminaci▒n normal.
```

- pruebaVector1:

```
hendrix01:~/pleng/ ./interprete pruebaVector1
Vectores: to 39 Verdadero Falso
a[4]: 7
Variables: 7 70 Falso
3 9

Terminaci▒n normal.
```

Anexo.- Formas de compilar

Existen varias formas de compilar:

- Sin argumentos y sin fichero: Muestra una ayuda de los posibles argumentos que puede tomar y la sintaxis de compilación:
  Sintaxis: java minilengcompiler [argumentos] <fichero_sin_extension>
- Argumento -p: Activa el *panic mode* (si falta un ;, no se cuenta como error)
- Argumento -v: Muestra una tabla con las apariciones de cada símbolo del programa
- Argumento -t: Muestra una tabla con las apariciones de cada símbolo después de cada acción

Se han implementado dos formas de introducir los argumentos:

- Se pueden introducir por separado:
  java minilengcompiler -v -p -t adivinar1
- Se pueden introducir todos juntos:
  java minilengcompiler -vpt adivinar1