COMPILADORES I

Alex Mateo Fiol
Andrés Ramos Seguí
Jaime Crespí Valero

Índice

Introducción

La práctica consiste en la creación de la fase front-end de un compilador la cual se compone en el analizador léxico, sintáctico y semántico. La práctica deberá contemplar los siguientes requisitos:

- Tipos básicos: número, cadena de texto, lógico (booleano).
- Declaración y uso tanto de variables como de constantes.
- Operaciones relacionales, aritméticas, lógicas y de asignación.
- Condicional: En este caso un condicional ‘if’.
- Instrucción de entrada de teclado y de salida.
- Funciones o procedimientos.

Extras realizados en la práctica:

Aparte de todas las funcionalidades anteriormente comentadas, se han realizado una serie de extras:

- Implementación de todos los operadores aritméticos (suma, resta, multiplicación, división, modulo).
- Bucle while.
- Bucle for.
- Condicional switch.

Tecnologías utilizadas:

El lenguaje de programación escogido para la práctica ha sido **Java**, el cual nos da la versatilidad y las herramientas para poder realizar las tareas del compilador.

Para poder realizar el analizar léxico hemos utilizado la librería **‘jflex’** la cuál nos permite crear los tokens mediante patrones (expresiones regulares), obtener los lexemas en caso de que fuera necesario y además se ocupa de todo el proceso de creación de identificación de tokens (generación del autómata finito determinista) . La librería se ocupa de transpilar un fichero (con extensión **.flex**) en un fichero java que se ocupa la conversión de un fichero fuente a generación de tokens.

Para poder crear el analizar sintáctico y semántico, utilizaremos la librería **‘cup’.** Con dicha librería podemos generar una gramática y utilizar sus rutinas semánticas para crear nuestro analizador semántico.

La librería **‘cup’** nos ofrece la clase **‘Symbol’** la cuál se puede utilizar en el **‘jflex’** para generar los tokens y esto nos permite la unión del léxico con el sintáctico. Por otro lado, el propio sintáctico generará una clase con los identificadores de token que nosotros deseemos crear, en nuestro caso es la clase **ParseSym** y la cual podemos usar en nuestro fichero **flex** para enlazar el identificador de token del léxico con el sintáctico.

La tabla de símbolos y el gestor de errores se hace completamente en Java y se explicará más detalladamente en próximas secciones.

Análisis léxico

Vamos a explicar las diferentes partes del analizador léxico que hemos construido en la práctica.

Primero de todo hay que explicar que los **comentarios**, **espacios en blanco**, **saltos de línea** serán detectados e ignorados, es decir, no crearemos sus respectivos tokens para luego enviarlos al analizador sintáctico.

Para detectarlos usaremos el siguiente patrón:

```
WHITE       = ( " " |\t|\r|\n|\r\n)              // Espacios en blanco, saltos de línea, tabular
COMMENT = ( "//"[^\n\r]*| "/*".*"*/") // Comentarios
```

Además también tendremos dos declaraciones que se usa en los patrones de la **tabla de tokens** (más abajo).

```
LETTER  =  [a-zA-Z]
DIGIT     =  [0-9]
```

Por último, en caso de que **no se cumpla ninguno de los patrones** de los tokens de la tabla de abajo, entonces se generará el **token error** y se lanzará una excepción (**NotFoundException**) que se almacenará en un fichero. Para saber más información, ir a la sección de errores.

Tabla de tokens:

Descripción	Token Id	Lexema	Patrón
Condicional if	inst_if	No	("if")
	inst_elif	No	(“elif”)
	inst_else	No	(“else”)
Condicional switch	inst_switch	No	(“switch”)
	inst_case	No	(“case”)
	inst_break	No	(“break”)
	inst_default	No	(“default”)
Bucle while	inst_while	No	(“while”)
Bucle for	inst_for	No	(“for”)
Función	inst_function	No	(“function”)
	inst_return	No	(“return”)
Instrucción para llamar a las funciones	inst_call	No	(“call”)
Entrada por teclado	inst_input	No	(“input”)
Salida por teclado	inst_output	No	(“output”)
Tokens para montar las estructuras	lparen	No	(“(“)
	lparen	No	(“)”)
	lbracket	No	(“{“)
	rbracket	No	(“}”)
	separator	No	(“,”)
	two_points	No	(“:”)
	final_sentence	No	(“.”)
Operadores relacionales	op_relational	Sí	("==" | "!=" | "<" | "<=" | ">" | ">=")
Operador asignación	op_assign	No	(“=”)
Operadores lógicos	op_logic	Sí	("&&" | "||")
Operadores aritméticos	op_arithmetic	Sí	("+" | "-" | "*" | "/" | "%")
Valores del tipo booleano	bool	Sí	("true" | "false")
Constante	constant	No	(“const”)
Identificador	id	Sí	(({LETTER} | "_")({LETTER}|{DIGIT}| "_")*)
Números	number	Sí	("0" | [1-9]{DIGIT}*)

Textos	text	Sí	\" [^\"]* \"

En la clase **ParseSym** se puede comprobar que cada **identificador de token** tiene un valor numérico que lo identifica. Esta clase la genera la librería **cup** cuando es ejecutada y la utilizaremos en el léxico para identificar a los tokens.

Rutinas semánticas del léxico:

Las rutinas semánticas se ocuparán principalmente de generar y devolver cada unos los tokens que se encuentre en el fichero fuente y además los escribirá en un fichero donde se encontrará el listado ordenado de tokens proporcionando información como su identificador, la línea, columna y lexema en caso de que exista.

Para devolver el token utilizaremos principalmente dos métodos que hemos creado:

```
/*
Genera un token a partir de su identificador numérico (recordar que se encuentran en la clase ParseSym todos los identificadores. Además se almacena en ambos métodos la línea y columna que en caso de error sintáctico sea más fácil de detectar.
*/
private Symbol symbol(int type) {
	return new Symbol(type, yyline, yycolumn);
}

private Symbol symbol(int type, Object value) {
	return new Symbol(type, yyline, yycolumn, value);
}
```

Luego tenemos un método para escribir el listado de tokens en el fichero correspondiente.

```
public void writeTokensInFile(ArrayList<TokenLex> tokens) throws IOException
```

Ejemplo de rutina semántica en código para generar el **token id**:

```
{
	// Obtenemos su lexema ya que en este caso es necesario
	String lexeme = new String(yytext());

	// Generamos el token para luego escribirlo en el fichero
	TokenLex token = new TokenLex(TokenLex.TOKEN_ID.ID, lexeme, yyline, yycolumn);
```

```
        // lo añadimos a la lista de tokens
        tokens.add(token);

        // Devolvemos el objeto Symbol que será el token para nuestro sintáctico
        return symbol(ParserSym.id, lexeme);
}
```

Autómata finito determinista:

El autómata finito determinista se ocupa de generarlo la librería ‘flex’. Para ello nosotros generamos un fichero con extensión ‘.flex’ con el cual se dispondrá a generar el autómata para código ‘Java’.

Gestión de errores:

En este caso al **no reconocer ninguno de los patrones** de la **tabla de tokens**, se procederá a ejecutar el último patrón el cuál engloba cualquier posible carácter y que utilizaremos para generar y devolver un token cuyo identificador es error y además lanzaremos una excepción que generará un fichero con el error ocurrido.

Análisis sintáctico

En el analizador sintáctico nos ocuparemos de la correcta estructura del código fuente. Para todo ello necesitamos generar una gramática la cual contendrá variables y tokens (los mencionados en el léxico).

Nuestra gramática esta pensada para que no exista un programa principal como tal, sino que puedes introducir todas las instrucciones que desees.

El constructor de nuestro sintáctico tiene la siguiente forma:

Parser (ScannerLex scanner, SymbolFactory sf, boolean hasToGenerateDigraph)

Los parámetros son:

- La clase del analizador léxico, la cuál nos proporciona los tokens del código fuente para tratarlo en el sintáctico.

- El SymbolFactory que se utiliza internamente.

- Por último, un booleano para saber si queremos generar un grafo que conecta las diferentes producciones de nuestra gramática.

Además de lo mencionado, el constructor se ocupará de inicializar el **generador del grafo** y la **tabla de símbolos**.

Gramática:

// Root
ROOT => INIT_TYPES INSTRS

// Declaración de tipos en la tabla de símbolos
INIT_TYPES => λ

// Generador de instrucciones
INSTRS => INSTRS INSTR
| INSTR

// Listado de instrucciones que se pueden ejecutar
INSTR => DECLS
| COND_IF
| SWITCH
| WHILE
| FOR
| ASSIGN
| OUTPUT
| FUNCTION
| CALL

// Declaración de variables
DECLS => id separator DECLS
| id CONSTANT two_points id op_assign VALUE final_sentence

// Una variable puede ser constante
CONSTANT => constant
| λ

// **Asignación de una variable**
ASSIGN => id op_assign VALUE final_sentence

// Operación aritméticas
INIT_OP_ARITH => lparen OP_ARITHMETIC rparen

```
OP_ARITHMETIC => OP_ARITHMETIC op_arithmetic OP_ARITH_VALUE
                | OP_ARITH_VALUE op_arithmetic OP_ARITH_VALUE


OP_ARITH_VALUE => lparen OP_ARITHMETIC rparen
                | SIGN number
                | id

// Signo de los valores numéricos
SIGN => op_arithmetic
      | λ

// Valores que se pueden asignar
VALUE => INIT_OP_ARITH
       | INIT_OP_BOOL
       | SIGN number
       | text
       | bool
       | inst_input lparen id rparen
       | id
       | CALL

// Instrucción para standard output
OUTPUT => inst_output lparen VALUE rparen final_sentence

// Operaciones booleanas (se utiliza en los condicionales)
INIT_OP_BOOL => lparen OP_BOOLEAN rparen

OP_BOOLEAN => OP_BOOLEAN op_logic OP_BOOL_VALUE
            | OP_BOOL_VALUE


OP_BOOL_VALUE => bool
                | lparen OP_BOOLEAN rparen
                | id
                | RELATIONAL_COMP

// Operaciones relacionales
RELATIONAL_COMP => lbracket VALUE op_relational VALUE rbracket

// Condicional If
```

COND_IF => inst_if INIT_OP_BOOL lbracket CODE_BLOCK_IN INSTRS
CODE_BLOCK_OUT rbracket COND_ELIF COND_ELSE

COND_ELIF => COND_ELIF inst_elif INIT_OP_BOOL lbracket
CODE_BLOCK_IN INSTRS CODE_BLOCK_OUT rbracket
| λ

COND_ELSE => inst_else lbracket CODE_BLOCK_IN INSTRS
CODE_BLOCK_OUT rbracket
| λ

// Condicional switch
SWITCH => SWITCH_BEGIN SWITCH_END

SWITCH_BEGIN => SWITCH_BEGIN inst_case lparen VALUE rparen two_points
CODE_BLOCK_IN INSTRS CODE_BLOCK_OUT inst_break final_sentence
| inst_switch lparen VALUE:comparison_value rparen lbracket

SWITCH_END => inst_default two_points CODE_BLOCK_IN INSTRS
CODE_BLOCK_OUT inst_break final_sentence rbracket
| rbracket

WHILE => inst_while INIT_OP_BOOL lbracket CODE_BLOCK_IN INSTRS
CODE_BLOCK_OUT rbracket

// Bucle For
FOR => inst_for CODE_BLOCK_IN lparen DECLS INIT_OP_BOOL
final_sentence ASSIGN rparen lbracket INSTRS CODE_BLOCK_OUT rbracket

// Entrada en un nuevo bloque (funciones, condicionales, bucles, etc)
CODE_BLOCK_IN => λ

// Salida de un nuevo bloque (funciones, condicionales, bucles, etc)
CODE_BLOCK_OUT => λ

// Funciones
FUNCTION => FUNC_HEAD rparen lbracket CODE_BLOCK_IN FUNC_BODY
RETURN CODE_BLOCK_OUT rbracket

FUNC_HEAD => inst_function id id lparen PARAMS

PARAMS => HAS_PARAMS
| λ

HAS_PARAMS => HAS_PARAMS separator id two_points id
| id two_points id

FUNC_BODY ::= INSTRS
| λ

RETURN => inst_return VALUE final_sentence
| λ

// Llamar funciones
CALL=> inst_call lparen CALL_BODY rparen

CALL_BODY => CALL_BODY separator VALUE
| id

Tablas y método del analizador:

En analizador sintáctico que hemos usado es el **LALR(1),** el cual nos proporciona la librería **'cup'**. En cuanto a las **tablas T1** y **T2** se generarán automáticamente gracias a dicha librería.

Análisis semántico

En el análisis semántico se usaran rutinas semánticas las cuales gestionarán la tabla de símbolos, los posibles errores semánticos y el dígrafo que tenemos que generar para las reglas del sintáctico.

Traducción dirigida por la sintaxis

La variable ROOT es la producción inicial.

La producción INIT_TYPE se ocupa de inicializar los tipos en la tabla de símbolos.

La producción INSTRS se ocupa de hacer que se ejecuten todas las instrucciones que queremos.

La producción INSTR se ocupa de ejecutar una instrucción, la cuál puede ser declaración de variables, asignación, condicionales, bucles, funciones, instrucción de salida y llamada de funciones.

La producción DECLS y CONSTANT se ocupa de declarar variables, para ello hay que comprobar que el tipo de la variable y su valor, tienen un tipo subyacente básico

igual para poder asignarlo correctamente, además hay que comprobar si la variable es una constante y por lo tanto comprobar si su valor corresponde a una constante, en caso afirmativo también habría que asignarle un valor. Por último, hay que añadir las variables a la tabla de símbolos.

La producción ASSIGN se ocupa de asignar un valor a una variable, para ello hay que comprobar que los tipos de la variable y el valor sean correctos y en caso de que sea una constante, hay que lanzar un error ya que no se puede asignar un valor a una constante.

Las producciones INIT_OP_ARITH, OP_ARITHMETIC y OP_ARITH_VALUE se ocupan de comprobar las operaciones aritméticas, para ellos comprueba que los operadores aritméticos sean correctos y que sus valores sean numéricos. Además nos devuelve el valor de la operación en caso de que todos los valores sean constantes.

La producción SIGN se ocupa de comprobar que el operador aritmético sea el de sumo o resta.

La producción VALUE se ocupa de devolver todos los posibles valores, los cuales son operaciones aritméticas, operaciones booleanas, valores simples como booleano, entero o texto, llamada a funciones y instrucciones de entrada y salida. Siempre devolvemos el tipo del valor (dtype, dvar, idnull, etc) y dependiendo de cada caso, el nombre del tipo, el tipo subyacente básico, su valor (si es constante), etc.

La producción OUTPUT se ocupa de la salida estandard por pantalla y tiene que comprobar que el valor que se le pase por parámetros sea un valor correcto.

Las producciones INIT_OP_BOOL, OP_BOOLEAN, OP_BOOL_VALUE se ocupan de las operaciones booleans, para ello hay que comprobar que se usan correctamente los operadores lógicos y que en caso de que todos los tipos sean constantes, devolver el valor.

La producción RELATIONAL_COMP se utiliza en las operaciones booleanas mencionadas anteriormente, hay que comprobar el correcto uso de los operadores relacionales y que los tipos de los dos valores de la operación relacional sean los mismos y acordes al operador relacional.

Las producciones COND_IF, COND_ELIF, COND_ELSE se ocupan del condicional if.

Las producciones SWITCH, SWITCH_BEGIN, SWITCH_END se ocupan del condicional switch y comprueban que los tipos del case sean los mismo que el del switch.

Las producciones WHILE y FOR, se ocupan del bucle while y for respectivamente.

Las producciones CODE_BLOCK_IN y CODE_BLOCK_OUT se ocupan de entrar en un nuevo ámbito y salir del mismo.

Las producciones FUNCTION, FUNC_HEAD, PARAMS, HAS_PARAMS, FUNC_BODY, RETURN se ocupan de la creación de funciones. Hay que comprobar que el valor de retorno de la función sea el mismo que el la instrucción return, hay que poner la función y los parámetros en la tabla de símbolos, etc.

Las producciones CALL y CALL_BODY, se ocupan de llamar a una función, pasandole sus parámetors. En este caso hay que comprobar que la función exista, que los tipos de los parámetros y el número de parámetros son correctos.

Tabla de símbolos

La estructura completa de la tabla de símbolos se compone principalmente de 4 clases, **SymbolsTable, Description, Expansion** y **TypeDescription.** La clase principal por tanto será **SymbolsTable.**

La estructura de la tabla de símbolos se compone:

- Entero que indica el **ámbito actual**.

- Un hashing para la **tabla de descripción**, cuyo id es un string y el valor es un objeto de tipo **Description,** la cuál contiene las variables visibles actualmente.

- Un ArrayList para la **tabla de expansión** cuyo contenido es el objeto **Expansion** y se ocupa de las variables que no son visibles.

- Un ArrayList para la **tabla de ámbito**, la cuál simplemente tiene valor enteros indicando el siguiente lugar dónde podemos escribir en la **tabla de expansión**.

El objeto '**Description**' y '**Expansion**' son muy parecidos, ya que ambos tienen un objeto de tipo '**TypeDescription**'**,** el '**id**', el '**ámbito**'**.** La principal diferencia es que el objeto 'Description' tiene un campo adicional llamado **'first'** que es un entero que apunta a la tabla de descripción, dónde se encuentra el primer elemento en caso de que sea una estructura más compleja como un array, parámetros de una función, una tupla. En cambio, el objeto 'Expansion' tiene un campo llamado 'idc' el cuál tiene el nombre del campo en caso de una tupla o un parámetro y además hay otro campo adicional llamado 'next' el cuál tiene el siguiente elemento de la estructura en caso de

que exista.

El objeto '**TypeDescription**' contiene toda la información relacionada con el tipo, dependiendo de su tipo (tipo, variable, constante, función, parámetro, etc) tendrá contenidos diferentes.

Explicación de las funciones:

Descripción	Nombre del método	Parámetros	Devuelve
Consulta un identificador en la tabla de descripción	query	String id	TypeDescription
Reinicia la tabla de símbolos	reset	-	-
Entrar en un nuevo ámbito	getinBlock	-	-
Salir de un ámbito	getoutBlock	-	-
Añadir una nueva variable a la tabla de descripción	add	String id, TypeDescription td	-
Añadir un parámetro a una variable de tipo función	addParam	String idVar, String idParam, TypeDescription td	-
Conseguir el tipo de un parámetro por su identificador	queryParam	String idVar, String idParam	TypeDescription
Conseguir el tipo de un parámetro por un índice	queryParam	String idVar, int index	TypeDescription
Conseguir el número de parámetros de una función	getNumParameters	String idVar	Int
Escribir la tabla de símbolos en un fichero	updateSymbolsTableFile	String method, String msg	-
Cerrar fichero	closeFile	-	-
Imprimir por pantalla la tabla de símbolos	printTables	-	-

Gestión de errores

Todas las excepciones de nuestro compilador tienen herencia con la excepción **‘CompilerException**’ la cuál se hereda de la clase ‘Exception. Esta excepción recibe como parámetro un enumerado que especifica el tipo de error (léxico, sintáctico, semántico o de la tabla de símbolos) y un mensaje de error. Esta excepción se ocupará de crear un fichero con el tipo de error y su mensaje.

El resto de excepcione son:

Descripción	Nombre	Léxico	Sintáctico	Semántico	Tabla de símbolos
Cuando vamos a crear un recurso pero ya existe	AlreadyExistException	No	No	No	Sí
Cuando buscamos un recurso y no existe	DoesNotExistException	Sí	No	No	Sí
Cuando una operación (aritmética, lógica, etc) no es posible	IncorrectOperatorException	No	No	Sí	No
Cuando falla comprobamos tipos y hay un error	IncorrectTypeException	No	No	Sí	Sí
Cuando una estructura no está bien formada	MalformedStructureException	No	Sí	No	No
Cuando se intenta sobrepasar los limites inferiores o superiores	OutOfBoundsException	No	No	No	Sí