

Documentación de Intérprete de pseudocódigo

Procesadores de Lenguajes

2º Cuatrimestre, 3º Ingeniería Informática (Computación)
2014-2015

Escuela Politécnica Superior de Córdoba, Universidad de
Córdoba.

José Ángel Gutiérrez Leo

Luis Flores Méndez



08-05-2015

1-Introducción.....	2
2-Lenguaje de Pseudocódigo.....	3
2.1 Léxico.....	3
2.2 Sentencias.....	6
3-Tabla de Símbolos.....	8
4-Análisis Léxico.....	10
5-Análisis Sintáctico.....	11
5.1-Símbolos de la gramática.....	11
5.1.1-Símbolos terminales.....	11
5.1.2-Símbolos no terminales.....	12
5.2-Reglas de producción de la gramática.....	13
5.3-Acciones semánticas.....	17
5.3.1-Sentencias de control.....	17
5.3.2-Bucles.....	17
6-Modo de obtención del interprete.....	19
7-Modo de ejecución del interprete.....	21
8-Ejemplos.....	22
9-Conclusiones.....	34
10- Bibliografía.....	35

En esta práctica nos ha sido propuesto el desarrollo de un intérprete de pseudocódigo con notación prefija en español, para introducir este concepto, primero debemos conocer que es el pseudocódigo y cuál es su función.

El pseudocódigo describe un algoritmo o programa, utilizando una mezcla de frases en lenguaje común, instrucciones de programación y palabras clave que definen estructuras básicas. Su objetivo es permitir que el programador se centre en los aspectos lógicos de la solución a un problema.

Una vez tenemos nociones de lo que es un pseudocódigo, podemos pasar a describir qué es un intérprete y cuáles son sus utilidades.

Un intérprete es un programa que analiza y ejecuta simultáneamente un programa escrito en un lenguaje fuente, por tanto, nuestro intérprete será capaz de analizar y ejecutar programas creados en un lenguaje de pseudocódigo, de forma que tendrá una utilidad básicamente didáctica, gracias a su capacidad de abstracción.

Para el desarrollo de este intérprete, hemos utilizado Flex y Bison.

Por una parte Flex es un generador de analizador léxico, que traduce la especificación de un analizador léxico a un programa en C que lo implementa.

- Esta descrito/especificado mediante expresiones regulares (E.R).
- A las E.R. se les puede asociar acciones (código C)
- Cada vez que el analizador encuentra una secuencia que encaja con una de las E.R, ejecuta la acción asociada.

Junto con Flex, utilizamos el analizador sintáctico Bison, encargado de generar analizadores sintácticos de propósito general, que convierte una descripción gramatical para una gramática independiente, en un programa en C que analice esa gramática.

A continuación en este documento encontraremos una descripción formal acerca de los componentes léxicos y las sentencias que constituyen nuestro intérprete, la tabla de símbolos y su descripción, una descripción detallada del fichero flex utilizado para definir y reconocer componentes léxicos y, de la misma forma, una descripción del análisis sintáctico, también se describirá el modo de obtención del intérprete así como su modo de ejecución, se mostrarán algunos ejemplos y se llegará a una conclusión sobre el trabajo realizado.

Por una parte, encontramos los componentes léxicos, o tokens, entre los cuales encontramos:

1. Palabras reservadas:

- a. Palabras claves
 - i. #mod
 - ii. #o,#y,#no
 - iii. Leer, leer_cadena
 - iv. Escribir, escribir_cadena
 - v. Si, entonces, si_no
 - vi. Mientras
 - vii. Hacer
 - viii. Repetir
 - ix. Hasta
 - x. Para
 - xi. Desde
 - xii. Paso
 - xiii. Borrar
 - xiv. Lugar
- b. Constantes y funciones predefinidas
 - i. PI,E...
 - ii. Seno, coseno, raíz...
- c. Observaciones
 - i. No se distinguirán entre mayúsculas ni minúsculas.
 - ii. Las palabras reservadas no se podrán utilizar como identificadores.

2. Identificadores:

- a. Características:
 - i. Estarán compuestos por una serie de letras, dígitos y subrayado.
 - ii. Deben comenzar por una letra
 - iii. No podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.
- b. Identificadores validos:
 - i. Dato,dato_1, dato_1_a
- c. Identificadores no validos:
 - i. _dato, dato_, dato__1

3. Número:

- a. Se utilizarán números enteros, reales de punto fijo y reales con notación científica.
- b. Todos ellos serán tratados conjuntamente como números.
- c. Ej: 1, 3, 27, -1, 10e 5, v3...

4. Cadena:

- a. Estará compuesta por una serie de caracteres delimitados por comillas simples.
- b. Deberá permitir la inclusión de la comilla simple utilizando la barra (\).
- c. NOTA: Las comillas exteriores no se almacenaran como parte de la cadena.
- d. Ej: 'Esto es una cadena', 'Así se representaría una cadena 125'...

5. Operadores aritméticos con notación prefija:

- a. suma: +
 - i. Unario: (+ 2)
 - ii. Binario: (+ dato 2)
- b. resta: -
 - i. Unario: (- 2)
 - ii. Binario: (- dato 2)
- c. producto: *
 - i. (* 2 dato)
- d. división: /
 - i. (/ dato 2)
- e. módulo: #mod
 - i. (#mod dato 2)
- f. potencia: ^
 - i. (^ a 3)

6. Expresión numérica:

- a. Número
- b. Variable que tenga un valor numérico dato
- c. Expresión compuesta por operadores numéricos y constantes o funciones predefinidas
- d. Ej: 12, 2.25, contador, (+ contador 1)

7. Operador alfanumérico

- a. Concatenación: '::'
- b. Ej: (:: 'hola' ' que tal'), (:: string1 string 2)

8. Expresión alfanumérica

- a. Cadena
- b. Variable con un valor alfanumérico
- c. Expresión compuesta por el operador de concatenación
- d. Ej: 'Esto es una expresión alfanumérica 125', cadena, (:: 'esto es' ' 8 coches')

9. Operadores relacionales de números y cadenas con notación prefija

- a. menor que: <
 - i. (< dato 0)
 - ii. (< nombre 'Antonio')
- b. menor o igual que: <=
 - i. (<= dato 0)
 - ii. (<= nombre 'Antonio')
- c. mayor que: >
 - i. (> dato 0)
 - ii. (> nombre 'Antonio')
- d. mayor o igual: >=
 - i. (>= dato 0)
 - ii. (>= nombre 'Antonio')
- e. igual que: =
 - i. (= dato 0)
 - ii. (= nombre 'Antonio')
- f. distinto que: <>
 - i. (<> dato 0)
 - ii. (<> nombre 'Antonio')

10. Operadores lógicos

- a. Disyunción lógica: #o
 - i. (#o (> a 5) (> b 5))
- b. Conjunción lógica: #y
 - i. (#y (> a 0) (< a 10))
- c. Negación lógica: #no
 - i. (#no (<> control 'stop'))

11. Condición

- a. Simple
 - i. (= dato 7)
- b. Compuesta
 - i. (#y (> a 0) (< a 10))

12. Comentarios

- a. De varias líneas:
 - i. delimitados por llaves
 - ii. Ej: {Esto sería un comentario.}
- b. De una línea
 - i. Todo lo que siga al carácter @ hasta el final de la línea.
 - ii. Ej: @Esto también sería un comentario.

Por otro lado, tenemos las sentencias:

1. Asignación

- a. (:= identificador expresión numérica)

- i. Declara a identificador como una variable numérica y le asigna el valor de la expresión numérica.
 - ii. Las expresiones numéricas se formarán con números, variables numéricas y operadores numéricos.
 - iii. Ej: (`:= dato 12.5`)
 - b. (`:= identificador expresión alfanumérica`)
 - i. Declara a identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.
 - ii. Las expresiones alfanuméricas se formarán con cadenas, variables alfanuméricas y el operador alfanumérico de concatenación (`|`).
 - iii. Ej: (`:= nombre 'Alejandro'`)

2. Lectura

- a. (`Leer identificador`)
 - i. Declara a identificador como variable numérica y le asigna el número leído.
 - ii. Ej: (`Leer dato`), (`Leer x`), (`Leer y`)
- b. (`Leer_cadena identificador`)
 - i. Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).
 - ii. (`Leer_cadena cadena`), (`Leer_cadena nombre`)

3. Escritura

- a. (`Escribir expresión numérica`)
 - i. El valor de la expresión numérica es escrito en la pantalla.
 - ii. Ej: (`Escribir dato`), (`Escribir (< 1 2)`), (`Escribir (+2 4)`)
- b. (`Escribir_cadena expresión alfanumérica`)
 - i. La cadena (sin comillas exteriores) es escrita en la pantalla.
 - ii. Se debe permitir la interpretación de comandos de saltos de línea (`\n`) y tabuladores (`\t`) que puedan aparecer en la expresión alfanumérica.
 - iii. Ej: (`Escribir_cadena '\t Introduzca el dato \n'`)

4. Sentencias de control

- a. Sentencia condicional simple
 - i. (`si condición entonces sentencias`)
 - ii. Ej: (`si (< 1 2) entonces escribir 'hola'`)
- b. Sentencia condicional compuesta
 - i. (`si condición entonces sentencias si_no sentencias`)
 - ii. (`si (< a 2) entonces escribir 'hola' si_no escribir 'adios'`)
- c. Bucle "mientras"
 - i. (`mientras condición hacer sentencias`)
 - ii. (`mientras (= a 2) hacer (+ contador 1)`)

- d. Bucle “repetir”
 - i. (repetir sentencias hasta condición)
 - ii. (repetir (si ($< a$ 2) entonces (+ contador 1)) hasta (=contador 20))
- e. Bucle2 “para”
 - i. (para identificador desde expresión_numérica hasta expresión numérica paso expresión numérica hacer sentencias)
 - ii. (para i desde 1 hasta 5 paso 1 hacer (escribir i))

5. Comandos especiales

- a. Borrar la pantalla
 - i. (Borrar)
- b. Posicionar el curso en pantalla
 - i. (Lugar expresión_numérica expresión_numérica)
 - 1. Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

Tabla de símbolos

Los lenguajes de programación incluyen mecanismos para definir nuevos símbolos junto con el significado que les da el usuario, para esto, existe la tabla de símbolos, que es la estructura de datos que permite almacenar la información semántica, es decir, sería el “diccionario” específico de un programa.

Esta tabla es creada durante la traducción del código fuente del programa y se utiliza durante la compilación.

Sus principales funcionalidades son:

- Efectuar chequeos semánticos.
- Generación de código.

La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: nombre, tipo, dirección de localización, tamaño, etc.

Símbolo	Token	Función	Parámetros
#mod	MOD	modulo()	1-Expresión 2-Expresión
Leer	READ	leerVariable()	Variable
Leer_cadena	LEER_CADENA	Leer_cadena()	Cadena
Escribir	PRINT	Escribir()	Expresión
Escribir_cadena	ESCRIBIR_CADENA	Escribir_cadena()	Expresión
Si	IF	ifcode()	1-Cuerpo if. 2-Cuerpo else(opcional) 3-Siguiente instrucción anidada.
Entonces	THEN		
Si_no	ELSE		
Mientras	WHILE	Whilecode()	1-Condicion de parada 2-Cuerpo del bucle 3-Siguiente instruccion
Repetir	REPETIR	Repetircode()	
Hasta	HASTA		
Para	PARA		
Desde	DESDE		
Paso	PASO		
Lugar	LOCATION		

Suma	+	Sumar()	1-Numero 2-Numero
Resta	-	Restar()	1-Expresión 2-Expresión
Producto	*	Multiplicar()	1-Expresión 2-Expresión
División	/	Dividir()	1-Expresión 2-Expresión
Potencia	^	Potencia()	1-Expresión 2-Expresión
Menor_que	<	Menor_que()	Expresión
Menor_o_igual	<=	Menor_igual()	1-Expresión 2-Expresión
Mayor que	>	Mayor_que()	1-Expresión 2-Expresión
Mayor_o_igual	>=	Mayor_igual()	1-Expresión 2-Expresión
Igual_que	=	Igual()	1-Expresión 2-Expresión
Distinto_que	<>	Distinto()	1-Expresión 2-Expresión
Disyuncion_log	#o	O_logico()	1-Expresión 2-Expresión
Conjunción_log	#y	Y_logico()	1-Expresión 2-Expresión
Negación_log	#no	Negativo()	1-Expresión 2-Expresión
Asignación	:=	Assign()	Expresión

El analizador léxico se encargará de reconocer y separar convenientemente los elementos básicos (tokens) del lenguaje que estamos construyendo. Debemos distinguir: literales de los distintos tipos de datos que existan en el lenguaje, identificadores (ya sean palabras clave o no).

En nuestro fichero de análisis léxico, al que nombramos como `lexico.l` encontramos la siguiente estructura:

- En la zona de declaraciones, encontramos las definiciones regulares de :
 - Espacio
 - Letra
 - Dígito
 - Identificador
 - Número
 - Cadena
 - Mod
 - O
 - Y
 - No
- En la zona de reglas, podemos encontrar reglas a ejecutar en el caso de encontrar:
 - `\t \n`, para saltar los espacios
 - Una cadena
 - Un número
 - Un identificador
 - `>=`
 - `<=`
 - `=`
 - `<>`
 - `::`
 - `:=`
 - `;`
 - `#mod`
 - `#o`
 - `#y`
 - `#no`
 - `^#` -se utiliza para finalizar el programa
 - `/n` -se utiliza para no devolver, y continuar con análisis léxico
 - `^\$[^\n]*` para ejecutar un comando del shell
- Por último, nuestro fichero de análisis léxico no dispone de una zona de funciones auxiliares

1.1 Símbolos de la gramática

1.1.1 Símbolos terminales

Los símbolos terminales de la gramática en Bison se denominan tokens y deben declararse en la sección de definiciones. Por convención éstos se escriben en mayúscula y los no terminales en minúscula.

Las dos formas más usuales de escribir los símbolos terminales en la gramática son:

Un **token declarado** se escribe como un identificador, de la misma manera que un identificador en C. Cada uno de estos nombres debe definirse con una declaración de %token.

Los tokens declarados de nuestra gramática son:

- NUMBER: token que identifica un número
- VAR: token para una variable (numérica o cadena)
- CONSTANTE: token para una constante
- FUNCION0_PREDEFINIDA: token para una función predefinida
- FUNCION1_PREDEFINIDA: token para una función predefinida
- FUNCION2_PREDEFINIDA: token para una función predefinida
- INDEFINIDA: token para variables indefinidas
- PRINT: token que reconoce la palabra clave “escribir”
- PRINT_CHAR: token que reconoce la palabra clave “escribir_cadena”
- WHILE: token que reconoce la palabra clave “mientras”
- IF: token que reconoce la palabra clave “si”
- THEN: token que reconoce la palabra clave “entonces”
- ELSE: token que reconoce la palabra clave “si_no”
- READ: token que reconoce la palabra clave “leer”
- READ_CHAR: token que reconoce la palabra clave “leer_cadena”
- CADENA: token que identifica a una cadena
- DO: token que reconoce la palabra clave “hacer”
- REPEAT: token que reconoce la palabra clave “repetir”
- UNTIL: token que reconoce la palabra clave “hasta”
- FOR: token que reconoce la palabra clave “para”
- SINCE: token que reconoce la palabra clave “desde”
- INC : token que reconoce la palabra clave “paso”
- TOKEN_LUGAR: token que reconoce la palabra clave “lugar”
- TOKEN_BORRAR: token que reconoce la palabra clave “borrar”
- ASIGNACION: token que reconoce el componente léxico “:=”
- O_LOGICO: token que reconoce el componente léxico “#o”
- Y_LOGICO: token que reconoce el componente léxico “#y”
- MAYOR_QUE: token que reconoce el componente léxico “>”

- MENOR_QUE: token que reconoce el componente léxico "<"
- MENOR_IGUAL: token que reconoce el componente léxico "<="
- MAYOR_IGUAL: token que reconoce el componente léxico ">="
- DISTINTO: token que reconoce el componente léxico "<>"
- IGUAL: token que reconoce el componente léxico "="
- CONCATENACION: token que reconoce el componente léxico "::"
- MODULO: token que reconoce el componente léxico "#mod"
- POTENCIA: token que reconoce el componente léxico "^"
- UNARIO: token que reconoce el operador aritmético unario
- NEGACION: token que reconoce el componente léxico "#no"

Un **token de carácter** se escribe en la gramática utilizando la misma sintaxis usada en C para las constantes de un carácter, como por ejemplo '+'. No necesita ser declarado a menos que se necesite especificar el tipo de datos de su valor semántico, asociatividad o precedencia.

Los tokens de carácter utilizados en esta gramática son:

- '(': token que reconoce el inicio de una sentencia
- ')': token que reconoce el final de una sentencia
- '+': token que reconoce el operador aritmético de la suma
- '-': token que reconoce el operador aritmético de la resta
- '*': token que reconoce el operador aritmético de la multiplicación
- '/': token que reconoce el operador aritmético de la división
- '^': token que reconoce el operador aritmético de la potencia

1.1.2 Símbolos no terminales

Los símbolos no terminales de esta gramática son los declarados con %type y por convenio se escriben en minúscula. Realmente son los símbolos que describen las reglas gramaticales y aquellos que en las reglas que describen no llaman a otros símbolos, sea de forma recursiva, a otros no terminales o a terminales. En esta gramática el símbolo terminal '(' marca el inicio de una sentencia (sea una condición, un bucle, una sentencia de control, una asignación...) y el símbolo terminal ')' marca el final de dicha sentencia. En todas las reglas de la gramática está así especificado.

- **list**: este símbolo se compone por la secuencia de agrupaciones *stmt* o *error*.
- **stmt**: Las reglas que describen a símbolo terminal engloban las asignaciones, las impresiones por pantalla, lectura de datos, posicionamiento del cursor, borrado de la pantalla, los bucles (*while*, *dowhile* y *for*) y las sentencias de control (*if* y *else*). En las sentencias de control se diferencia entre un *if* sin *else* y un *if* con *else*.
- **asgn**: Este símbolo se ocupa de las asignaciones de valores a variables.
- **variable**: Símbolo que identifica una variable.
- **cond**: Símbolo que identifica una condición.
- **mientras**: Símbolo que identifica el bucle *while*.
- **repedir**: Símbolo que identifica el bucle *dowhile*.

- **si:** Símbolo que identifica la sentencia de control *if*.
- **fin:** Símbolo que identifica el final de sentencia.
- **para:** Símbolo que identifica el bucle *for*.
- **stmtlist:** Símbolo que reconoce una secuencia de agrupaciones *stmt*.
- **expr:** Es un símbolo que se compone de las reglas que engloban a las expresiones, es decir, un número, una cadena, una constante, asignaciones, funciones predefinidas, operaciones matemáticas y condicionales y la concatenación.

1.2 Reglas de producción de la gramática

Las reglas de producción de la gramática son las que describen a los símbolos no terminales. Las reglas para los símbolos no terminales de la gramática de este intérprete, a grandes rasgos, son las siguientes:

list

Reglas:

- list stmt
- list error

Aplica la recursividad por la izquierda llamando a list para conseguir una secuencia de agrupaciones *stmt* o *error* o combinaciones de ambas.

stmt

Reglas:

- '(' PRINT expr ')'
- '(' READ VAR ')'
- '(' PRINT_CHAR expr ')'
- '(' READ_CHAR VAR ')'

Sirven para leer y almacenar en una variable o imprimir por pantalla un número o una cadena. Sigue la sintaxis descrita en el enunciado del trabajo.

- '(' TOKEN_LUGAR expr expr ')'

Entre paréntesis ha de haber el token de “lugar” y dos expresiones que finalmente serán números.

- '(' TOKEN_BORRAR ')'

Entre paréntesis tiene que estar la palabra reservada “borrar”.

- '(' mientras cond DO stmtlist ')' fin

Esta regla controla el bucle *while*, cuya sintaxis es la descrita aquí arriba. Si se encuentra el símbolo no terminal *mientras* y seguido de la condición, la palabra reservada “hacer” y una lista de sentencias, se activará. El símbolo *fin* sirve para dar paso a la siguiente sentencia tras el bucle.

- '(' si cond THEN stmtlist ')' fin
- '(' si cond THEN stmtlist fin ELSE stmtlist ')' fin

Estas reglas controlan las sentencias de control *if* y *else*. La primera es la sintaxis para una sentencia con solo *if* y la segunda para una sentencia de *if* y *else*.

- '(' para variable SINCE expr fin UNTIL expr fin INC expr fin DO stmtlist ')' fin

Esta regla controla el bucle *for*. La sintaxis es la que se describe, habiendo un final de sentencia después de cada expresión, es decir, cuando se asigna un valor inicial a la variable, la condición de parada y el incremento del valor de la variable (sea negativo o positivo). Tras esto se realizan las acciones que se contengan en el *stmtlist*.

- '(' repetir stmtlist fin UNTIL cond ')' fin

Esta regla controla el bucle *dowhile*, cuya sintaxis es la que se describe. Como en el bucle *for*, se diferencian las partes de las sentencias a realizar de las sentencias de control, que en este caso es el “hasta” y la condición.

asgn

- '(' ASIGNACION VAR expr ')' fin
- '(' ASIGNACION CONSTANTE expr ')' fin

Estas reglas controla la asignación de valores. La primera controla la asignación a una variable y la segunda la asignación a una constante. La asignación a constantes no puede ser permitida, por lo que ésta devolverá un código de error.

variable

- VAR

Esta regla simplemente reconoce una variables para introducirla en la pila.

cond

- expr

Esta regla llama al símbolo terminal *expr*, en el que se encuentran definidas las sintaxis de las condiciones.

mientras

- WHILE

Esta regla identifica la palabra clave “mientras” y llama al código del bucle *while* en el *code.c*.

repetir

- REPEAT

Esta regla identifica la palabra clave “repetir” y llama al código del bucle *dowhile* en el *code.c*.

si

- IF

Esta regla identifica la palabra clave “si” y llama al código de la sentencia de control *si* en el *code.c*.

fin

- /* nada: produccion epsilon */

Esta regla no realiza ninguna acción, simplemente introduce un epsilon en la tabla.

para

- FOR

Esta regla identifica la palabra clave “para” y llama al código del bucle *for* en el *code.c*.

stmtlist

- /* nada: produccion epsilon */
- stmtlist stmt

Estas reglas usan la recursividad por la izquierda para identificar una lista de cero o más elementos del símbolo no terminal *stmt*.

expr

- NUMBER
- VAR
- CONSTANTE
- CADENA

Estas reglas identifican un número, una variable, una constante o una cadena que se encuentren dentro de un símbolo *expr*.

- asgn

Identifica una asignación dentro de un símbolo *expr*.

- FUNCION0_PREDEFINIDA '(' ')'
- FUNCION1_PREDEFINIDA '(' expr ')'
- FUNCION2_PREDEFINIDA '(' expr ',' expr ')'

Describen las sintaxis para las funciones que se puedan predefinir en los códigos que vaya a interpretar el intérprete.

- '(' expr ')'

Indica que una expresión puede encontrarse entre paréntesis.

- '(' '+' expr expr ')'
- '(' '-' expr expr ')'
- '(' '*' expr expr ')'
- '(' '/' expr expr ')'
- '(' MODULO expr expr ')'
- '(' '^' expr expr ')'
- '(' '-' expr ')' %prec UNARIO
- '(' '+' expr ')' %prec UNARIO

Estas reglas identifican las operaciones matemáticas, sin necesidad de función predefinida, que se pueden realizar en los códigos y posteriormente llaman a las funciones correspondientes (en la semántica).

- '(' MAYOR_QUE expr expr ')'
- '(' MAYOR_IGUAL expr expr ')'
- '(' MENOR_QUE expr expr ')'
- '(' MENOR_IGUAL expr expr ')'
- '(' IGUAL expr expr ')'
- '(' DISTINTO expr expr ')'
- '(' Y_LOGICO expr expr ')'
- '(' O_LOGICO expr expr ')'
- '(' NEGACION expr expr ')'

Estas son las reglas que definen la sintaxis de las diferentes condiciones que podemos encontrar en la gramática del lenguaje de pseudocódigo descrito anteriormente.

- '(' CONCATENACION expr expr ')'

Esta regla define la sintaxis para la concatenación de dos cadenas, llamando posteriormente a la función correspondiente.

1.3 Acciones semánticas

A continuación se describirán las acciones semánticas para las sentencias de control y los bucles que se pueden utilizar en la gramática del intérprete. Los elementos que se describen en cada uno de los apartados son las diferentes sentencias que se encuentran en las semánticas de las reglas que describen la sintaxis de estos elementos. Cuando encontramos un \$ seguido de un número, éste hace referencia al elemento en esa posición en la descripción de la sintaxis de la regla, es decir, que el \$2 en la semántica de la regla del *if* sin *else* hace referencia al símbolo no terminal *si* en la sintaxis descrita en el apartado anterior.

1.3.1 Sentencias de control

If

- (\$2)[1]=(Inst)\$5; - Almacena las sentencias del cuerpo del *if* que se ejecutarán si se cumple la condición.
- (\$2)[3]=(Inst)\$6; - Almacena la siguiente instrucción que se encontrará el intérprete para seguir ejecutando.

If – else

- (\$2)[1]=(Inst)\$5; - Almacena las sentencias del cuerpo del *if* que se ejecutarán si se cumple la condición.
- (\$2)[2]=(Inst)\$8; - Almacena las sentencias del cuerpo del *else* que se ejecutarán si no se cumple la condición inicial.
- (\$2)[3]=(Inst)\$9; - Almacena la siguiente instrucción que se encontrará el intérprete para seguir ejecutando.

1.3.2 Bucles

While

- (\$2)[1]=(Inst)\$5; - Almacena las sentencias del cuerpo del *while* que se ejecutarán mientras se cumpla la condición.
- (\$2)[2]=(Inst)\$6; - Almacena la siguiente instrucción que se encontrará el intérprete cuando finalice el bucle para seguir ejecutando.

Do while

- (\$2)[1]=(Inst)\$3; - Almacena la condición del bucle *dowhile* que se ejecutarán mientras se cumpla la condición.
- (\$2)[2]=(Inst)\$6; - Almacena la siguiente instrucción que se encontrará el intérprete cuando finalice el bucle para seguir ejecutando.

For

- (\$2)[1] = (Inst)\$8; - Almacena la condición de parada.
- (\$2)[2] = (Inst)\$11; - Almacena el incremento.
- (\$2)[3] = (Inst)\$14; - Almacena las sentencias del cuerpo del *for* que se ejecutarán mientras se cumpla la condición.
- (\$2)[4] = (Inst)\$15; - Almacena la siguiente instrucción que se encontrará el intérprete cuando finalice el bucle para seguir ejecutando.

Modo de obtención del intérprete

Nombre y descripción de cada fichero utilizado

- `lexico.l`
 - Fichero de análisis léxico, donde encontramos las reglas y declaraciones del léxico del lenguaje.
- `ipe.y`
 - Fichero correspondiente a la parte de análisis sintáctico del intérprete.
- `lpe.h`
 - Contiene la definición de las estructuras `Symbol` y `Datum`.
- `code.c`
 - Fichero de funciones en c, utilizadas para realizar operaciones de los tokens del lenguaje.
- `Makefile`
 - Fichero para la compilación del intérprete.
- `symbol.c`
 - Fichero generado al ejecutar `yacc -d` y contiene una tabla de símbolos, así como sus funciones de reserva de memoria y llenado.
- `math.c`
 - Contiene una función de comprobación de errores matemáticos en diferentes expresiones.
- `macros.h`
 - Contiene la definición de macros útiles para el intérprete.
- `init.c`
 - Contiene estructuras, que contienen los nombres y valores de las constantes, y las palabras clave, así como su inicialización.

Descripción del fichero `makefile`

Aquí es donde se generará el ejecutable del intérprete. Basta con usar el comando `make` en el directorio donde estén todos los archivos que componen el compilador para que éste se cree.

A lo que se llama FUENTE es el nombre que recibirá finalmente el intérprete (en este caso `ipe`), y así tendrán que estar nombrados todos los archivos que lo componen, como el archivo que guarda el semántico (con la extensión `.y`) y la cabecera donde se definen las estructuras y las funciones que se van a usar en el compilador (con la extensión `.h`). Estos dos archivos son los que usará Bison para generar los archivos `.tab.c` y `.tab.h`, en los que encontraremos los valores de los tokens que hemos creado en el “`ipe.y`”.

La variable `LEXICO` simplemente almacena el nombre que se le ha dado al archivo que guarda el léxico del intérprete (con la extensión `.l`), que en este caso es el archivo “`léxico.l`”.

La variable CC almacena el compilador que se usará para crear los códigos objeto, que será el compilador de C "gcc". En las variables YFLAGS y LFLAGS se almacenan los parámetros para los compiladores de bison y C respectivamente. En OBJS se almacenan los nombres de los códigos objeto necesarios para compilar el intérprete. Estos se consiguen a través de la compilación con la opción "-c" del compilador "gcc" de los archivos necesarios, que son el "math.c", "code.c", "init.c", "symbol.c" y "lex.yy.c". El contenido y la finalidad de estos archivos son los explicados en este apartado del documento.

Modo de ejecución del intérprete

A la hora de ejecutar nuestro intérprete, tenemos dos modos de hacerlo, según la funcionalidad deseada, por una parte tenemos el modo interactivo y por otra, el modo de lectura de fichero.

- **Modo interactivo:** Conocemos por modo interactivo a la ejecución del programa introduciendo el texto por teclado desde nuestra consola. Para ejecutar el programa de modo interactivo simplemente debemos ejecutar nuestro archivo ejecutable sin pasar ningún parámetro y el programa comenzará a ejecutarse en consola, teniendo el usuario la posibilidad de interactuar con el sistema.

```
luis@ubuntu:~/i22f1mel/PL/interprete$ ./ipe.exe
(escribir_cadena 'hola')
hola
(:= a 2)
(:= b 3)
(escribir (< a b))
      1
(escribir (+ a b))
      5
(leer_cadena string)
Esto es una cadena
(escribir_cadena string)
Esto es una cadena
luis@ubuntu:~/i22f1mel/PL/interprete$
```

- **Modo fichero:** Para ejecutar el modo fichero debemos ejecutar el programa pasándole por la línea de comandos la dirección de un fichero previamente creado, al que deseemos pasar nuestro interprete de pseudocódigo, de este modo, el sistema cargará ese fichero y lo analizará.

Hemos desarrollado un par de ejemplos a parte de los que nos proporcionó el profesor, uno calcula un elemento determinado de la sucesión de Fibonacci y el otro coge los datos proporcionados por el usuario y los presenta de una manera ordenada por pantalla.

ejemplo_fibonacci.e

Código

```
(borrar)
(lugar 10 10)
(escribir_cadena ' Elemento x de la sucesión de Fibonacci ')

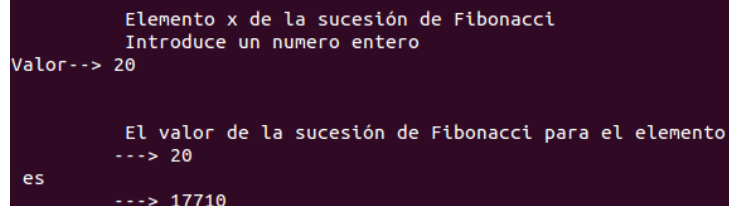
(lugar 11 10)
(escribir_cadena ' Introduce un numero entero ')
(Leer N)

(:= x 0)
(:= anterior 0)
(:= a 1)
(:= elemento 0)

(para i desde 0 hasta N paso 1 hacer
  (:= x
    (+ a anterior)
  )
  (:= elemento (+ elemento 1))
  (lugar 15 10)
  (:= anterior a)
  (:= a x)
)

(escribir_cadena ' El valor de la sucesión de Fibonacci para el elemento ')
  (escribir N)
  (escribir_cadena ' es ')
  (escribir (- x 1))
```

Ejecución



```
      Elemento x de la sucesión de Fibonacci
      Introduce un numero entero
Valor--> 20

      El valor de la sucesión de Fibonacci para el elemento
----> 20
es
----> 17710
```

ejemplo_datos.e

Código

```
(borrar)
(lugar 10 10)
(escribir_cadena ' Datos personales ')

@Obtención de datos
(escribir_cadena ' Introduce tu nombre ')
(leer_cadena nombre)
(escribir_cadena ' Introduce tu primer apellido ')
(leer_cadena apellido1)
(escribir_cadena ' Introduce tu segundo apellido ')
(leer_cadena apellido2)
(escribir_cadena ' Introduce tu DNI ')
(leer_cadena dni)
(escribir_cadena ' Introduce tu dirección ')
(leer_cadena direccion)
(escribir_cadena ' Introduce tu edad ')
(leer_cadena edad)
(escribir_cadena ' Introduce tu universidad ')
(leer_cadena uni)
(escribir_cadena ' Introduce tu grado ')
(leer_cadena grado)
(escribir_cadena ' Introduce tu curso ')
(leer_cadena curso)
(escribir_cadena ' Introduce el nombre de la asignatura ')
(leer_cadena asignatura)
@Datos ordenados

(borrar)
(lugar 2 2)
(escribir_cadena ' Nombre completo: ')
(lugar 3 8)
(escribir_cadena (:: 'Nombre: ' nombre))
(lugar 4 8)
(escribir_cadena (:: 'Primer apellido: ' apellido1))
(lugar 5 8)
(escribir_cadena (:: 'Segundo apellido: ' apellido2))
(lugar 6 2)
(escribir_cadena (:: ' DNI: ' dni))
(lugar 7 2)
(escribir_cadena (:: ' Dirección: ' direccion))
(lugar 8 2)
(escribir_cadena (:: ' Edad: ' edad))
(lugar 2 40)
(escribir_cadena (:: ' Universidad: ' uni))
(lugar 3 40)
(escribir_cadena (:: ' Grado: ' grado))
(lugar 4 40)
(escribir_cadena (:: ' Curso: ' curso))
(lugar 5 40)
(escribir_cadena (:: ' Asignatura: ' asignatura))
(escribir_cadena '\n')
```


Ejecución

Nombre completo:	Universidad: Universidad de Córdoba
Nombre: Luis	Grado: Grado en Ingeniería Informática
Primer apellido: Flores	Curso: 3º
Segundo apellido: Méndez	Asignatura: Procesadores de Lenguajes
DNI: 45888797W	
Dirección: Madres Escolapias 23	
Edad: 20	

ejemplo_geometria.e

Código

```
(borrar)
(lugar 10 10)

(escribir_cadena 'Menú para operaciones geométricas')

(repetir

  @ Opciones disponibles
  (borrar)
  (lugar 10 10)
  (escribir_cadena ' 1- Área de un rectángulo ')
  (lugar 11 10)
  (escribir_cadena ' 2- Área de un cuadrado ')
  (lugar 12 10)
  (escribir_cadena ' 3 - Área de un triángulo ')
  (lugar 13 10)
  (escribir_cadena ' 4 - Área de un círculo ')
  (lugar 14 10)
  (escribir_cadena ' 0 - Finalizar ')

  (lugar 15 10)
  (escribir_cadena ' Elige una opcion ')
  (leer opcion)

  (borrar)

  (si (= opcion 0)      @ Fin del programa
    entonces
      (lugar 10 10)
      (escribir_cadena nombre)
      (escribir_cadena ': gracias por usar el intérprete ipe.exe ')

  si_no
  (si (= opcion 1)
    entonces
      (lugar 10 10)
      (escribir_cadena 'Área de un rectángulo ')
      (escribir_cadena 'Introduce la altura: ')
      (leer altura)
      (escribir_cadena 'Introduce el ancho: ')
      (leer ancho)
      (:= resultado (* altura ancho))
      (escribir_cadena 'Resultado:')
      (escribir resultado)

  si_no
  (si (= opcion 2)
    entonces
      (lugar 10 10)
```

```

        (escribir_cadena ' Área de un cuadrado ')
        (escribir_cadena 'Introduce el lado: ')
        (leer lado)
        (:= resultado (* altura ancho))
        (escribir_cadena 'Resultado:')
        (escribir resultado)
si_no
  (si (= opcion 3)
    entonces
      (lugar 10 10)
      (escribir_cadena ' Área de un triángulo ')
      (escribir_cadena 'Introduce la altura: ')
      (leer altura)
      (escribir_cadena 'Introduce el ancho: ')
      (leer ancho)
      (:= resultado (/ (* altura ancho) 2))
      (escribir_cadena 'Resultado:')
      (escribir resultado)
si_no
  (si (= opcion 4)
    entonces
      (lugar 10 10)
      (escribir_cadena ' Área de un círculo ')
      (escribir_cadena 'Introduce el radio: ')
      (leer radio)
      (:= resultado (* PI (^ radio 2)))
      (escribir_cadena 'Resultado:')
      (escribir resultado)

si_no
  (lugar 15 10)
  (escribir_cadena ' Ha elegido una opción incorrecta ')

  ))))

hasta (= opcion 0)
)

(borrar)
(lugar 10 10)
(escribir_cadena 'El programa ha concluido')

```

Ejecución

<pre> 1- Área de un rectángulo 2- Área de un cuadrado 3 - Área de un triángulo 4 - Área de un círculo 0 - Finalizar Elige una opcion 1 </pre>	<pre> Área de un rectángulo Introduce la altura: Skype Introduce el ancho: 4 Resultado: 20 </pre>
---	---

<pre> 1- Área de un rectángulo 2- Área de un cuadrado 3 - Área de un triángulo 4 - Área de un círculo 0 - Finalizar Elige una opcion 4 </pre>	<pre> Área de un círculo Introduce el radio: 5 Resultado: 78.539816 </pre>
---	--

ejemplo_1_saluda.e

Código

```
{
  Asignatura:  Procesadores de Lenguajes
  Titulación:  Ingeniería Informática
  Especialidad: Computación
  Curso:      Tercero
  Cuatrimestre: Segundo

  Departamento: Informática y Análisis Numérico
  Centro:       Escuela Politécnica Superior de Cádiz
  Universidad de Cádiz

  Curso académico: 2014 - 2015

  Fichero de ejemplo para el intérprete de pseudocódigo en español con notación prefija: ipe.exe
}

@ Bienvenida

(borrar)

(lugar 10 10)

(escribir_cadena 'Introduce tu nombre --> ')

(leer_cadena nombre)

(borrar)
(lugar 10 10)

(escribir_cadena ' Bienvenido/a << ')

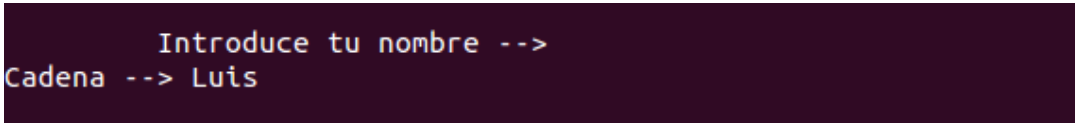
(escribir_cadena nombre)

(escribir_cadena ' >> al intérprete de pseudocódigo en español:\'ipe.exe\'.')

(lugar 40 10)
(escribir_cadena 'Pulsa una tecla para continuar')
(leer_cadena pausa)

(borrar)
(lugar 10 10)
(escribir_cadena 'El programa ha concluido')
```

Ejecución



```
Introduce tu nombre -->
Cadena --> Luis
```

```

                Bienvenido/a <<
Luis
>> al intérprete de pseudocódigo en español:'ipe.exe'.

                Pulsa una tecla para continuar
Cadena --> 

```

ejemplo_2_factorial.e

Código

```

(borrar)
(lugar 10 10)
(escribir_cadena ' Factorial de un número ')

(lugar 11 10)
(escribir_cadena ' Introduce un número entero ')
(leer N)

(:= factorial 1)

(para i desde 2 hasta N paso 1 hacer
    (: = factorial
    (* factorial i)
    )
)

@ Se muestra el resultado

(lugar 15 10)
(escribir_cadena ' El factorial de ')
(escribir N)
(escribir_cadena ' es ')
(escribir factorial)

```

Ejecución

```
Factorial de un número
Introduce un número entero
Valor--> 5

El factorial de
--> 5
es
--> 120
```

ejemplo_3_mcd.e

Código

```
(borrar)
(borrar)
(lugar 10 10)
(escribir_cadena ' Máximo común divisor ----> 2 ')

@ Máximo común divisor

(lugar 10 10)
(escribir_cadena ' Máximo común divisor de dos números ')
(lugar 11 10)
(escribir_cadena ' Algoritmo de Euclides ')

(lugar 12 10)
(escribir_cadena ' Escribe el primer número ')
(leer a)

(lugar 13 10)
(escribir_cadena ' Escribe el segundo número ')
(leer b)

@ Se ordenan los números
(si (< a b)
    entonces
        (:= auxiliar a)
        (:= a b)
        (:= b auxiliar)
    )
@ Se guardan los valores originales
(:= A1 a)
(:= B1 b)

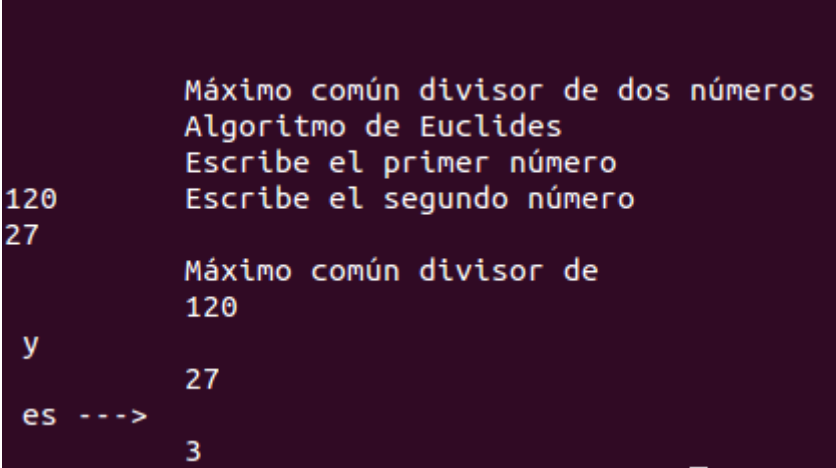
@ Se aplica el método de Euclides
(:= resto (#mod a b))
(mientras (<> resto 0) hacer
    (:= a b)
    (:= b resto)
    (:= resto (#mod a b))
)
```

```

@ Se muestra el resultado
(lugar 15 10)
(escribir_cadena ' Máximo común divisor de ')
(escribir A1)
(escribir_cadena ' y ')
(escribir B1)
(escribir_cadena ' es ---> ')
(escribir b)

```

Ejecución



```

Máximo común divisor de dos números
Algoritmo de Euclides
Escribe el primer número
Escribe el segundo número
120
27
Máximo común divisor de
120
y
27
es --->
3

```

ejemplo_4_menu_inicial.e

Código

```

(borrar)
(lugar 10 10)
(escribir_cadena 'Introduce tu nombre --> ')
(leer_cadena nombre)

(borrar)
(lugar 10 10)
(escribir_cadena ' Bienvenido/a << ')
(escribir_cadena nombre)
(escribir_cadena ' >> al intérprete de pseudocódigo en español:\'ipe.exe\'.')

(lugar 40 10)
(escribir_cadena 'Pulsa una tecla para continuar')
(leer_cadena pausa)

(repetir

@ Opciones disponibles
(borrar)
(lugar 10 10)
(escribir_cadena ' Factorial de un número --> 1 ')
(lugar 11 10)
(escribir_cadena ' Máximo común divisor ----> 2 ')
(lugar 12 10)

```

```

(escribir_cadena ' Finalizar -----> 0 ')

(lugar 15 10)
(escribir_cadena ' Elige una opcion ')
(leer opcion)

(borrar)

(si (= opcion 0)    @ Fin del programa
  entonces
    (lugar 10 10)
    (escribir_cadena nombre)
    (escribir_cadena ': gracias por usar el intérprete ipe.exe ')
  si_no
    @ Factorial de un número
    (si (= opcion 1)
      entonces
        (lugar 10 10)
        (escribir_cadena 'Ha elegido la opción de Factorial de un numero ')

        @ Máximo común divisor
        si_no
          (si (= opcion 2)
            entonces
              (lugar 10 10)
              (escribir_cadena ' Ha elegido la opción de Máximo común divisor
de dos números ')

              @ Resto de opciones
              si_no
                (lugar 15 10)
                (escribir_cadena ' Ha elegido una opción incorrecta ')
            ) @ Fin de la sentencia condicional que empieza en la línea 74

          ) @ Fin de la sentencia condicional que empieza en la línea 67

        ) @ Fin de la sentencia condicional que empieza en la línea 59
      (lugar 40 10)
      (escribir_cadena '\n Pulse una tecla para continuar --> ')
      (leer_cadena pausa)
      hasta (= opcion 0)
    ) @ Fin del bucle repetir que comienza en la línea 42
    (borrar)
    (lugar 10 10)
    (escribir_cadena 'El programa ha concluido')

```

Ejecución

```

Factorial de un número --> 1
Máximo común divisor ----> 2
Finalizar -----> 0

```

```

Elige una opcion

```

ejemplo_5_menu_completo.e

Código

```
(borrar)
(lugar 10 10)
(escribir_cadena 'Introduce tu nombre --> ')
(leer_cadena nombre)

(borrar)
(lugar 10 10)

(escribir_cadena ' Bienvenido/a << ')
(escribir_cadena nombre)
(escribir_cadena ' >> al intérprete de pseudocódigo en espa ol:\ipe.exe\'.')

(lugar 40 10)
(escribir_cadena 'Pulsa una tecla para continuar')
(leer_cadena pausa)

(repetir
(borrar)
(lugar 10 10)
(escribir_cadena ' Factorial de un n mero --> 1 ')
(lugar 11 10)
(escribir_cadena ' M ximo com n divisor ----> 2 ')
(lugar 12 10)
(escribir_cadena ' Finalizar -----> 0 ')

(lugar 15 10)
(escribir_cadena ' Elige una opcion ')
(leer opcion)
(borrar)

(si (= opcion 0)    @ Fin del programa
    entonces
        (lugar 10 10)
        (escribir_cadena nombre)
        (escribir_cadena ': gracias por usar el int rprete ipe.exe ')
    si_no
        @ Factorial de un n mero
        (si (= opcion 1)
            entonces
                (lugar 10 10)
                (escribir_cadena ' Factorial de un numero ')

                (lugar 11 10)
                (escribir_cadena ' Introduce un numero entero ')
                (leer N)

                (:= factorial 1)

                (para i desde 2 hasta N paso 1 hacer
                    (:= factorial
                        (* factorial i)
                    )
                )
                (lugar 15 10)
                (escribir_cadena ' El factorial de ')
                (escribir N)
                (escribir_cadena ' es ')
                (escribir factorial)

                @ M ximo com n divisor
```



```

si_no
    (si (= opcion 2)
        entonces
            (lugar 10 10)
            (escribir_cadena ' Máximo común divisor de dos números ')

            (lugar 11 10)
            (escribir_cadena ' Algoritmo de Euclides ')

            (lugar 12 10)
            (escribir_cadena ' Escribe el primer número ')
            (leer a)

            (lugar 13 10)
            (escribir_cadena ' Escribe el segundo número ')
            (leer b)

            @ Se ordenan los números
            (si (< a b)
                entonces
                    (: = auxiliar a)
                    (: = a b)
                    (: = b auxiliar)
                )

            @ Se guardan los valores originales
            (: = A1 a)
            (: = B1 b)

            @ Se aplica el método de Euclides
            (: = resto (#mod a b))

            (mientras (<> resto 0) hacer
                (: = a b)
                (: = b resto)
                (: = resto (#mod a b))
            )

            @ Se muestra el resultado
            (lugar 15 10)
            (escribir_cadena ' Máximo común divisor de ')
            (escribir A1)
            (escribir_cadena ' y ')
            (escribir B1)
            (escribir_cadena ' es ----> ')
            (escribir b)

            @ Resto de opciones
            si_no
                (lugar 15 10)
                (escribir_cadena ' Opcion incorrecta ')

            ) @ Fin de la sentencia condicional que empieza en la línea 89
        ) @ Fin de la sentencia condicional que empieza en la línea 66

    ) @ Fin de la sentencia condicional que empieza en la línea 59

(lugar 40 10)
(escribir_cadena '\n Pulse una tecla para continuar --> ')
(leer_cadena pausa)
hasta (= opcion 0)
) @ Fin del bucle repetir que empieza en la línea 42

```

```
@ Despedida final
(borrar)
(lugar 10 10)
(escribir_cadena 'El programa ha concluido')
```

Ejecución

```
Factorial de un número --> 1
Máximo común divisor ----> 2
Finalizar -----> 0

Elige una opcion
```

ejemplo_6_intercambiar_tipo.e

Código

```
(escribir_cadena 'Ejemplo de intercambio de valores \n')

(escribir_cadena 'Introduce un número --> ')

(leer numero)

(si (> numero 0)
  entonces
    (escribir_cadena 'Introduce una cadena --> ')
    (leer_cadena dato)
  si_no
    (escribir_cadena 'Introduce un número --> ')
    (leer dato)
)
@ Asignación
(:= nuevo dato)

(escribir_cadena 'Valor leído --> ')

(si (> numero 0)
  entonces
    (escribir_cadena nuevo)
  si_no
    (escribir nuevo)
)
```

Ejecución

Ejemplo de intercambio de valores	Ejemplo de intercambio de valores
Introduce un número -->	Introduce un número -->
4	-1
Introduce una cadena -->	Introduce un número -->
Hola	5
Valor leído -->	Valor leído -->
Hola	5

Aunque el intérprete es bastante completo, ya que puede reconocer e interpretar los diferentes tipos de bucles y sentencias de control que encontramos en la programación, aún le faltaría la implementación de los vectores de éstos en la gramática, tanto su declaración como su manejo. Junto con los vectores sería interesante también controlar la creación y el manejo de las matrices, así como las pilas, las colas y las listas.

Tras finalizar la creación del intérprete de pseudocódigo, según las directrices proporcionadas por el profesor, hemos llegado a la conclusión de que uno de los puntos débiles del sistema es la falta de un elemento indicativo del fin de sentencia, ya que la ausencia de éste produce errores de desplazamiento/reducción en las reglas de los símbolos *stmt* y *list*. Para arreglar esto hay que eliminar una de las líneas de la regla de producción *stmt*, concretamente la regla que produce un epsilon o elemento vacío.

Aún con estas deficiencias encontradas, el intérprete es capaz de reconocer la mayoría de las sentencias que encontramos en la programación en los pseudocódigos proporcionados, algo que lo hace bastante competente y útil a la hora de querer comprobar el funcionamiento de un algoritmo antes incluso de haberlo programado en el lenguaje final, ya que los errores de diseño se encontrarían al usar el intérprete y no una vez ya implementado el programa.

Bibliografía o referencias web

UNIVERSIDAD DE VIGO. Introducción a Flex [En línea]<<http://ccia.ei.uvigo.es/docencia/PL/doc/LEX.html> > [Consulta: Junio 2015].

MICHEL RUIZ TEJEIDA. Desarrollo de un compilador para pseudocódigo en lenguaje español [En línea]<http://computacion.cs.cinvestav.mx/~mruiz/publicaciones/tesis/tesisIngMichelRuizTejeda.pdf> [Consulta: Mayo 2015].

VERN PAXSON.FLEX 2.5: Un generador de analizadores léxicos. [En línea] <<http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.html>> [consulta: Mayo 2015]

RUBÉN BÉJAR HERNÁNDEZ. Introducción a Flex y Bison.[En línea] <http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf> [consulta: Junio 2015]

DR. D. SERGIO GÁLVEZ ROJAS.TICTema5.pdf[Enlínea]<<http://www.lcc.uma.es/~galvez/ftp/tci/tictema5.pdf> >[consulta: Julio 2015]