UNIVERSIDAD DE CÓRDOBA

GRADO DE INGENIERÍA INFORMÁTICA, COMPUTACIÓN

PROCESADORES DE LENGUAJES

3º CURSO, SEGUNDO CUATRIMESTRE

TRABAJO FINAL DE PRÁCTICAS

Documentación que recoge información sobre el trabajo final de las prácticas de la asignatura.

Alejandro Fuerte Jurado

Escuela Politécnica Superior de Córdoba

Curso 2018-2019

31-05-2019, Córdoba

ÍNDICE DE CONTENIDOS

1 INTRODUCCIÓN....2

2 DESCRIPCIÓN DEL INTÉRPRETE....3

2.1 LENGUAJE DE PSEUDOCÓDIGO....3

2.2 TABLA DE SÍMBOLOS....6

2.3 ANÁLISIS LÉXICO....7

2.4 ANÁLISIS SINTÁCTICO....7

2.5 CÓDIGO DE AST....11

2.6 FUNCIONES AUXILIARES....16

2.7 MODO DE OBTENCIÓN DEL INTÉRPRETE....17

2.8 MODO DE EJECUCIÓN DEL INTÉRPRETE....17

2.9 EJEMPLOS....18

2.10 CONCLUSIONES....19

Apéndice A BIBLIOGRAFÍA Y REFERENCIAS WEB....21

1 INTRODUCCIÓN

El presente documento recoge información sobre el trabajo final de las prácticas de la asignatura Procesadores de Lenguajes. Dicho trabajo consiste en la elaboración de un intérprete de pseudocódigo en español, utilizando flex y bison para su creación. En las siguientes páginas se describirá el lenguaje utilizado, tanto en el análisis léxico como en el sintáctico, y se explicará el funcionamiento de este intérprete. Se describirá la tabla de símbolos, las clases y funciones auxiliares utilizadas, los modos de obtención y de ejecución del intérprete, y se describirá la organización del programa. Además, se describirán varios ejemplos de ficheros escritos en pseudocódigo en español para comprobar el funcionamiento del mismo.

2 DESCRIPCIÓN DEL INTÉRPRETE

2.1 LENGUAJE DE PSEUDOCÓDIGO

- **Palabras reservadas**

El lenguaje contiene una serie de palabras reservadas, las cuales no se pueden utilizar como identificadores:

- *_mod, _div*
- *_o, _y, _no*
- *leer, leer_cadena*
- *escribir, escribir_cadena,*
- *si, entonces, si_no, fin_si*
- *mientras, hacer, fin_mientras*
- *repetir, hasta, para, fin_para, desde, paso*
- *_borrar, _lugar*

El intérprete no distingue entre mayúsculas y minúsculas en ningún caso.

- **Identificadores**

Por otro lado, los identificadores del lenguaje tienen las siguientes características:

- Están compuestos por una serie de letras, dígitos y el subrayado.
- Comienzan por una letra.
- No pueden acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.

De la misma forma no se distingue entre mayúsculas ni minúsculas.

- **Números**

El lenguaje utiliza números enteros, reales de punto fijo y reales con notación científica. Todos ellos son tratados conjuntamente como números.

- **Cadenas**

Están compuestas por una serie de caracteres delimitados por comillas simples. Permiten la inclusión de la comilla simple utilizando la barra (\). Las comillas exteriores no se almacenan como parte de la cadena.

- **Operadores**

El lenguaje tiene los siguientes operadores:

- Suma: +
- Resta: -
- Producto: *
- División: /
- División entera: _div
- Módulo: _mod
- Potencia: **
- Concatenación: ||

- Menor que: <
- Menor o igual que: <=
- Mayor que: >
- Mayor o igual que: >=
- Igual que: =
- Distinto que: <>
- Disyunción lógica: _o
- Conjunción lógica: _y
- Negación lógica: _no

- **Comentarios**

Los comentarios pueden ser de varias líneas, siendo delimitados por el símbolo #, o de una línea, todo lo que siga al carácter @ hasta el final de la línea.

Además de estos elementos, se incluye el punto y coma para indicar el fin de una sentencia.

- **Sentencias**

Las distintas sentencias que contiene el lenguaje son:

- Asignación: := . Declara al identificador como una variable numérica o alfanumérica y le asigna el valor de la expresión. Las expresiones numéricas se forman con números, variables y operadores numéricos. Las expresiones alfanuméricas se forman con cadenas, variables y el operador alfanumérico de concatenación (||).
- Lectura: leer(), leer_cadena() . Declara al identificador como variable numérica o alfanumérica y le asigna el número o cadena leída.
- Escritura: escribir(), escribir_cadena() . El valor es escrito en la pantalla.
- Sentencia condicional simple:
  si condición
  entonces sentencias

fin_si

- Sentencia condicional compuesta:
  si condición
  entonces sentencias
  si_no sentencias
  fin_si
- Bucle "mientras":
  mientras condición hacer
  sentencias
  fin_mientras
- Bucle "repetir":
  repetir
  sentencias
  hasta condición
- Bucle "para":
  para identificador
  desde expresión numérica 1
  hasta expresión numérica 2
  [paso expresión numérica 3]
  hacer
  sentencias
  fin_para
- Borrar: _borrar . Borra la pantalla
- Lugar: _lugar(expresión numérica1, expresión numérica2). Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

2.2 TABLA DE SÍMBOLOS

Para la tabla de símbolos se han usado clases para cada tipo de expresión. Estos son logicalConstant, logicalVariable, numericConstant, numericVariable, variable, constant y cadenaVariable. A parte, también se encuentran distintas clases que sirven para la creación de la tabla y para la iniciación de ella, tales como bultin, builtinParameter0, builtinParameter1, builtinParameter2, keyword, symbol, symbolInterface, table y init. En esta

última se encuentran las palabras reservadas descritas en el apartado anterior.

2.3 ANÁLISIS LÉXICO

Las definiciones regulares que nos encontramos en el fichero de flex son las siguientes:

DIGIT *[0-9]*

LETTER *[a-Za-Z]*

SUBRAYADO *"_"*

NUMBER1 *{DIGIT}+\.?*

NUMBER2 *{DIGIT}*\.{DIGIT}+*

IDENTIFIER *{LETTER}({LETTER}|{DIGIT}|{SUBRAYADO}({LETTER} | {DIGIT}))**

Además, existen varios estados definidos, para las cadenas, los dos tipos de comentarios y los mensajes de error.

La expresión regular de los números transforma el valor en texto para pasarlo al parser.

La expresión regular de los identificadores convierte todas las letras en minúscula, comprueba si el identificador está en la tabla y lo incluye si no está.

El resto de expresiones regulares devuelven una etiqueta que se usa más adelante para el análisis sintáctico en el parser.

2.4 ANÁLISIS SINTÁCTICO

Para describir la gramática de contexto libre podemos diferenciar entre distintos aspectos:

- **Símbolos de la gramática**
- stmtList: lista de declaraciones.

- stmt: declaración.
- asgn: asignación.
- print: imprimir valor numérico.
- printC: imprimir cadena.
- read: leer valor numérico.
- readC: leer cadena.
- clear: limpiar pantalla.
- place: ubicar un punto concreto.
- if: sentencia condicional.
- while: sentencia "mientras".
- do: sentencia "hacer"
- for: sentencia "para".
- cond: condición.
- exp: expresión.

- **Reglas de producción de la gramática**

Los símbolos anteriormente mencionados tienen una serie de reglas, en las que se crea un objeto de un símbolo si se detectan una serie de símbolos, es decir, se cumple la regla. A continuación se exponen todas ellas.

- stmtlist: crea una lista vacía si no se detecta nada más. Si va con un stmt se añade la declaración a la lista. Si se detecta un error se informa de ello.
- stmt: con solo SEMICOLON (punto y coma) se crea una declaración vacía. Si contiene asgn, print, printC, read, readC, clear, place, if, while, do o for se crea una declaración correspondiente con el tipo que sea.
- if: la sentencia condicional tiene dos reglas en función de si es simple o compuesta. Si es simple, la regla es "*SI cond ENTONCES stmtlist FIN_SI*". Si es compleja, la regla es "*SI cond ENTONCES stmtlist SI_NO stmtlist FIN_SI*".
- while: la regla es "*MIENTRAS cond HACER stmtlist FIN_MIENTRAS*".
- do: la regla es "*REPETIR stmtlist HASTA cond*".
- for: tiene dos reglas, una de ellas obvia el paso para darle como valor predeterminado 1. La regla completa es "*PARA VARIABLE DESDE exp*

HASTA exp PASO exp HACER stmtlist FIN_PARA". La segunda regla elimina el símbolo PASO y la expresión que le sigue.

- cond: representa una condición cuya regla es "*LPAREN exp RPAREN*", es decir, una expresión dentro de unos paréntesis.
- asgn: representa la asignación de una variable a una expresión, o bien a otra asignación. Las reglas correspondientes son "*VARIABLE ASSIGNMENT exp*" y "*VARIABLE ASSIGNMENT asgn*".
- print: "*ESCRIBIR exp*".
- printC: "*ESCRIBIR_CADENA exp*"
- read: la regla es "*LEER LPAREN VARIABLE RPAREN*", es decir, lee una variable que se encuentra entre paréntesis, como una función.
- readC: la regla es "*LEER_CADENA LPAREN VARIABLE RPAREN*", es decir, igual que la regla anterior pero aplicada a cadenas.
- clear: "*BORRAR*".
- place: la regla es "*LUGAR LPAREN NUMBER COMMA NUMBER RPAREN*", es decir, actúa como una función donde las coordenadas del lugar al que se quiere ir están entre paréntesis y separadas por una coma.
- exp: una expresión puede ser un número, una variable, una constante, una cadena, y por otra parte una operación entre dos expresiones. Estas reglas se definen de la forma "*exp PLUS exp*", para el caso de la suma, aplicándose de la misma forma para el resto de operaciones pero añadiendo entre las expresiones el símbolo correspondiente (PLUS, MINUS, MULTIPLICATION, DIVISION, DIVISION_ENTERA, CONCATENACION, MODULO, POWER). Por otra parte, una expresión puede ser unaria, por lo que para estos casos se aplican las reglas "*PLUS exp %prec UNARY*" y "*MINUS exp %prec UNARY*". Por último, una expresión también contiene las comparaciones entre expresiones, como puede ser "*exp GREATER_THAN exp*" (mayor que). Para el resto de comparaciones se cambia la etiqueta que hay entre las expresiones por la correspondiente (GREATER_THAN, GREATER_OR_EQUAL, LESS_THAN, LESS_OR_EQUAL, EQUAL, NOT_EQUAL, AND, OR, NOT).

- **Acciones semánticas**

Por lo general, al describir las reglas de producción en orden se entiende la semántica requerida, ya que al tener la sintaxis como forma de evaluación de los símbolos la de un árbol, cuando bajas a los símbolos terminales comprendes los tipos de elementos que pueden entrar en una regla o no. Sin embargo, puede haber reglas más complejas que pueden no entenderse solamente sabiendo su definición y los símbolos que requieren. Vamos a hablar de los casos de los distintos bucles y las condiciones.

- Condición simple:
  SI cond ENTONCES stmtlist FIN_SI

Si se cumple la condición cond entonces se realiza la lista de declaraciones stmtlist.

- Condición compuesta:
  SI cond ENTONCES stmtlist1 SI_NO stmtlist2 FIN_SI

Si se cumple la condición cond entonces se realiza la lista de declaraciones stmtlist1. Si no se cumple, se realiza la lista de declaraciones stmtlist2.

- Bucle "mientras":
  MIENTRAS cond HACER stmtlist FIN_MIENTRAS

Mientras que se cumpla la condición cond realizar la lista de declaraciones. Cada vez que se realizan se comprueba la condición para saber si se vuelve a hacer el bucle o no.

- Bucle "repetir":

  REPETIR stmtlist hasta cond

Después de la etiqueta REPETIR aparece una declaración o un conjunto de declaraciones, que se repiten en forma de bucle mientras la condición final no se cumpla. Es una especia de equivalencia al bucle "mientras", solo que en este caso el bucle se realiza mientras no se cumpla la condición, y esta se comprueba al final y no al comienzo.

- Bucle "para":

  PARA VARIABLE DESDE exp1 HASTA exp2 PASO exp3 HACER stmtlist FIN_PARA

El bucle funciona de la siguiente forma: para una variable VARIABLE, que tome un valor inicial exp1, realizar una lista de declaraciones stmtlist hasta que la variable tome el valor exp2. Cada vez que se realiza la lista de declaraciones se le suma a la variable el valor exp3. Por tanto las expresiones deben tener valor numérico que asignarle a la variable.

2.5 CÓDIGO DE AST

La clase principal es a que hace referencia a todos los tipos de expresiones descritas con anterioridad, la clase ExpNode. Esta clase tiene varias funciones virtuales, que son getType, print, evaluateNumber, evaluateCadena y evaluateBool. Las dos primeras son para obtener el tipo de expresión y para imprimirla, mientras que las otras sirven para evaluar un valor en las distintas clases que se heredan de la clase principal, pudiendo ser un valor numérico, booleano o alfanumérico. A continuación se detallan el resto de clases.

- Clase VariableNode. Hereda de la clase ExpNode. Representa un identificador que es clasificado como variable, la cual puede tomar valores numéricos, alfanuméricos o booleanos. El valor de la variable se guarda en la tabla, y para buscarlo se utiliza el nombre, que es de tipo string.
- Clase ConstantNode. Hereda de la clase ExpNode. Representa un valor constante, que de la misma forma está guardado en la tabla y se busca su valor al evaluar.
- Clase NumberNode. Hereda de la clase ExpNode. Representa el valor de un número concreto.
- Clase CadenaNode. Hereda de la clase ExpNode. Representa el valor de una cadena concreta.

- Clase UnaryOperatorNode. Hereda de la clase ExpNode. Guarda el valor de una expresión unaria.
- Clase NumericUnaryOperatorNode. Hereda de la clase UnaryOperatorNode. Representa el caso en el que la expresión unaria es numérica.
- Clase LogicalUnaryOperatorNode. Hereda de la clase UnaryOperatorNode. Representa el caso en el que la expresión unaria es lógica.
- Clase UnaryMinusNode. Hereda de la clase NumericUnaryOperatorNode. Representa el signo negativo para la expresión unaria numérica.
- Clase UnaryPlusNode. Hereda de la clase NumericUnaryOperatorNode. Representa el signo positivo para la expresión unaria numérica.
- Clase OperatorNode. Hereda de la clase ExpNode. Representa dos expresiones que se van a someter a una operación.
- Clase NumericOperatorNode. Hereda de la clase OperatorNode. Representa una operación numérica entre dos expresiones.
- Clase CadenaOperatorNode. Hereda de la clase OperatorNode. Representa una operación alfanumérica entre dos expresiones.
- Clase RelationalOperatorNode. Hereda de la clase OperatorNode. Representa una operación relacional entre dos expresiones.
- Clase LogicalOperatorNode. Hereda de la clase OperatorNode. Representa una operación lógica entre dos expresiones.
- Clase PlusNode. Hereda de la clase NumericOperatorNode. Realiza la suma entre dos expresiones numéricas.
- Clase MinusNode. Hereda de la clase NumericOperatorNode. Realiza la resta entre dos expresiones numéricas.
- Clase MultiplicationNode. Hereda de la clase NumericOperatorNode. Realiza la multiplicación entre dos expresiones numéricas.
- Clase ConcatenacionNode. Hereda de la clase CadenaOperatorNode. Realiza la concatenación entre dos cadenas.
- Clase DivisionNode. Hereda de la clase NumericOperatorNode. Realiza la división entre dos expresiones numéricas.

- Clase DivisionEnteraNode. Hereda de la clase NumericOperatorNode. Realiza la división entera entre dos expresiones numéricas.
- Clase ModuloNode. Hereda de la clase NumericOperatorNode. Realiza el módulo entre dos expresiones numéricas.
- Clase PowerNode. Hereda de la clase NumericOperatorNode. Realiza la potencia entre dos expresiones numéricas.
- Clase GreaterThanNode. Hereda de la clase RelationalOperatorNode. Comprueba si la primera expresión es mayor que la segunda.
- Clase GreaterOrEqualNode. Hereda de la clase RelationalOperatorNode. Comprueba si la primera expresión es mayor o igual que la segunda.
- Clase LessThanNode. Hereda de la clase RelationalOperatorNode. Comprueba si la primera expresión es menor que la segunda.
- Clase LessOrEqualNode. Hereda de la clase RelationalOperatorNode. Comprueba si la primera expresión es menor o igual que la segunda.
- Clase EqualNode. Hereda de la clase RelationalOperatorNode. Comprueba si la primera expresión es igual que la segunda.
- Clase NotEqualNode. Hereda de la clase RelationalOperatorNode. Comprueba si la primera expresión no es igual que la segunda.
- Clase AndNode. Hereda de la clase LogicalOperatorNode. Crea el operador lógico and entre dos expresiones.
- Clase OrNode. Hereda de la clase LogicalOperatorNode. Crea el operador lógico or entre dos expresiones.
- Clase NotNode. Hereda de la clase LogicalOperatorNode. Crea el operador lógico not entre dos expresiones.
- Clase Statement. Representa una declaración. Contiene las funciones virtuales print y evaluate, para imprimir y evaluar.
- Clase AssignmentStmt. Hereda de la clase Statement. Crea una asignación entre una variable y una expresión.
- Clase PrintStmt. Hereda de la clase Statement. Realiza la función de imprimir en pantalla una expresión numérica concreta.
- Clase PrintCadenaStmt. Hereda de la clase Statement. Realiza la función de imprimir en pantalla una cadena concreta.

- Clase ReadStmt. Hereda de la clase Statement. Realiza la función de leer por pantalla un valor numérico y asignarlo a una variable concreta.
- Clase ReadCadenaStmt. Hereda de la clase Statement. Realiza la función de leer por pantalla una cadena y asignarla a una variable concreta.
- Clase PlaceStmt. Hereda de la clase Statement. Realiza la función de cambiar la posición en la pantalla de la terminal, utilizando dos expresiones que representan las coordenadas.
- Clase ClearStmt. Hereda de la clase Statement. Realiza la función de limpiar la pantalla de la terminal.
- Clase EmptyStmt. Hereda de la clase Statement. Representa una declaración vacía.
- Clase IfStmt. Hereda de la clase Statement. Realiza la condición simple o compuesta. La función que evalúa la condición es la siguiente:

```
void lp::IfStmt::evaluate(){

    std::list<Statement *>::iterator stmtIter;

    // If the condition is true,
    if (this->_cond->evaluateBool() == true ){

        // the consequent is run
        for (stmtIter = this->_stmtList1->begin(); stmtIter != this->_stmtList1->end();
stmtIter++) {

            (*stmtIter)->evaluate();

        }

    }

    // Otherwise, the alternative is run if exists
    else if (this->_stmtList2 != NULL){

        for (stmtIter = this->_stmtList2->begin(); stmtIter != this->_stmtList2->end();
stmtIter++) {

            (*stmtIter)->evaluate();
        }

    }

}
```

- Clase WhileStmt. Hereda de la clase Statement. Realiza el bucle while. La función que realiza la evaluación del bucle es la siguiente:

```
void lp::WhileStmt::evaluate()
{
    std::list<Statement *>::iterator stmtIter;

    // While the condition is true. the body is run
    while (this->_cond->evaluateBool() == true)
    {
        for (stmtIter = this->_stmtList->begin(); stmtIter != this->_stmtList->end(); stmtIter++)
        {
            (*stmtIter)->evaluate();
        }
    }
}
```

- Clase ForStmt. Hereda de la clase Statement. Realiza el bucle for. La función que realiza la evaluación del bucle es la siguiente:

```
void lp::ForStmt::evaluate(){
    VariableNode *var = new VariableNode(this->_var);
    AssignmentStmt *asgn = new AssignmentStmt(this->_var,this->_num1);
    PlusNode *suma = new PlusNode(var,this->_num3);

    asgn->evaluate();
    std::list<Statement *>::iterator stmtIter;
    while (var->evaluateNumber()<this->_num2->evaluateNumber()){
        asgn->evaluate();
       for (stmtIter = this->_stmtList->begin(); stmtIter != this->_stmtList->end(); stmtIter++){
            (*stmtIter)->evaluate();
        }
        asgn = new AssignmentStmt(this->_var,suma);
    }
}
```

- Clase DoStmt. Hereda de la clase Statement. Realiza el bucle do. La función que realiza la evaluación del bucle es la siguiente:

```
void lp::DoStmt::evaluate()
{
    std::list<Statement *>::iterator stmtIter;

    // While the condition is false. the body is run
    while (this->_cond->evaluateBool() == false)
    {
        for (stmtIter = this->_stmtList->begin(); stmtIter != this->_stmtList->end(); stmtIter++)
        {
            (*stmtIter)->evaluate();
        }
    }
}
```

2.6 FUNCIONES AUXILIARES

A parte de las funciones anteriormente descritas, existe otro directorio con la clase error, que recoge todas las funciones relacionadas con los mensajes de error del intérprete. Las funciones son las siguientes:

- Función yyerror. Recoge los errores creados durante el proceso del parser.
- Función warning. Imprime los mensajes de error.
- Función execerror. Recoge errores creados durante el uso del programa.
- Función fpecatch. Recoge errores creados durante el uso del programa.
- Función errcheck. Control distintos posibles tipos de error, como errores de rango.

2.7 MODO DE OBTENCIÓN DEL INTÉRPRETE

Los ficheros se organizan de la siguiente forma.

Existen varios directorios para organizar los ficheros. Son los siguientes:

- Directorio /ast/. Guarda la clase ast y un makefile para ella.
- Directorio /error/. Guarda la clase error y un makefile para ella.
- Directorio /includes/. Guarda un fichero para las macros.
- Directorio /parser/. Incluye los ficheros del léxico y la sintaxis, además de un makefile que se encarga de generar los ficheros posteriores de flex.
- Directorio /table/. Guarda todas las clases descritas en el apartado 2.2 de este documento, además de un makefile para crear la tabla.

A parte de estos directorios encontramos el fichero ipe.cpp, que es el fichero principal que llama al intérprete. Por otro lado tenemos un fichero Doxyfile para generar dicho documento. Por último, encontramos el makefile principal de todo el programa. Este make es el que conecta todos los ficheros, realiza todos los makefiles de los distintos directorios, realiza el documento del Doxyfile, y una vez que todo está correcto, crea el ejecutable del intérprete.

2.8 MODO DE EJECUCIÓN DEL INTÉRPRETE

El intérprete tiene dos formas de ejecutarse: interactiva y a partir de un fichero.

- Interactiva. Se ejecuta como ./ipe.exe, sin darle ningún argumento. Así, el intérprete comienza a funcionar. En este modo podemos escribir el lenguaje pseudocódigo en el propio terminal, por lo que el intérprete detecta lo que escribimos y realiza en tiempo real las operaciones pedidas. De la misma forma, si cometemos un error nos informará de dicho error al momento. Para salir de este modo podemos escribir el símbolo ~ o con ctrl+d. Sin embargo, un aspecto importante a destacar es que desde este modo no podemos escribir las condiciones simples o compuestas ni los distintos bucles, ya que no funcionan correctamente al usar una lista de declaraciones. Por

tanto, para comprobar el funcionamiento de estas funciones se debe usar el otro modo.

- A partir de un fichero. Se ejecuta de la forma ./ipe.exe fichero.e, siendo el segundo argumento un fichero escrito en lenguaje pseudocódigo en español. Por tanto, con este modo el intérprete actúa como cualquier otro, analizando léxica y sintácticamente todo el fichero en orden y realizando las operaciones pedidas. En este modo sí se pueden usar las condiciones y los bucles. Si el fichero que ponemos como argumento no existe, saldrá un mensaje de error.

2.9 EJEMPLOS

A parte de los directorios descritos en el apartado 2.7, existe otro directorio que guarda distintos ejemplos de ficheros escritos en lenguaje pseudocódigo en español. Concretamente hay 7 ejemplos: 6 propuestos y realizados por el profesor y un ejemplo final original. Este ejemplo final recoge el esquema del menú del ejemplo 5, modificando algunos aspectos para comprobar el funcionamiento de la no distinción entre mayúsculas y minúsculas para las palabras reservadas, o el funcionamiento del bucle for con un paso predeterminado. En este menú se han añadido dos ejemplos originales, siendo los apartados de la sucesión de Fibonacci y la potencia de un número.

El código utilizado para realizar el apartado de la sucesión de Fibonacci es el siguiente:

```
_lugar(10,10);
escribir_cadena(' Cálculo de la sucesión de Fibonacci ');
escribir_cadena(' Cantidad de números que se desean calcular de la sucesión: ');
leer(N);
escribir_cadena(' Sucesión: ');

a := 1;
b := 1;
escribir(a);
escribir(b);
```

```
para i desde 1 hasta N paso 1 hacer
    c := a + b;
    a := b;
    b := c;

    escribir(c);
fin_para
```

Por otro lado, el código utilizado para realizar el apartado de calcular la potencia de un número es el siguiente:

```
_lugar(10,10);
escribir_cadena(' Cálculo de la potencia de un número ');
escribir_cadena(' Introduzca un número: ');
leer(N);
escribir_cadena(' Introduzca la potencia: ');
leer(pow);

resultado := N ** pow;
escribir_cadena(' El resultado es ');
escribir(resultado);
```

2.10 CONCLUSIONES

El trabajo realizado para crear este intérprete ha sido muy duro, ha llevado mucho tiempo, esfuerzo y sacrificio. Y sin embargo, esta versión del intérprete es muy simple. Esto demuestra la gran complejidad que hay detrás de un lenguaje de programación, y de un compilador profesional. Es un trabajo muy complejo para el programador que lo realiza.

El intérprete desarrollado, aun siendo muy simple, tiene la suficiente complejidad como para poder realizar programas simples en pseudocódigo, pudiendo realizarse una gran cantidad de operaciones. A partir de aquí, hay que analizar los puntos fuertes y puntos débiles del intérprete con el

objetivo de poder realizar en el futuro mejores versiones y nuevos elementos al lenguaje.

Los puntos débiles son fáciles de encontrar, sobre todo al comparar el lenguaje con uno profesional como c. Así, podemos tener nuevas ideas para el futuro a la hora de ampliar el lenguaje. Por ejemplo, se podrían incluir formas de leer y modificar ficheros, o poder crearlos y borrarlos. Además, se podrían incluir nuevos bucles o funciones que ahora no existen, tales como el switch del lenguaje c. Esto facilitaría la creación de un programa que use un menú, como puede ser el ejemplo descrito en el apartado anterior, el cual está formado por un conjunto de funciones condicionales compuestas una detrás de otra. También se podrían añadir los operadores ++, --, +:= y -:=.

Por otra parte, el intérprete es muy simple a la hora de informar sobre los errores. Un aspecto en el que se podría trabajar en el futuro es en que el propio intérprete, una vez analizado el error, intente solucionarlo e imprima esa posible solución para facilitar el trabajo al que está escribiendo en pseudocódigo.

Sabiendo todo esto, hay que tener en cuenta que esta es la primera versión del intérprete, y que poco a poco este puede ir creciendo y aprendiendo más aspectos del lenguaje, con el objetivo de conseguir un intérprete completo con el que se pueda trabajar.

A BIBLIOGRAFÍA Y REFERENCIAS WEB

1. Contenido teórico y práctico de la asignatura, así como los elementos proporcionados para comenzar la práctica se encuentran en Moodle: https://moodle.uco.es/m1819/.
2. Manual oficial de Flex: ftp://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html.
3. Manual oficial de Bison: https://www.gnu.org/software/bison/manual/html_node/index.html.
4. La plantilla utilizada para este trabajo ha sido creada por el equipo de AGATHA en base a la plantilla original proporcionada para la asignatura Ampliación de Ingeniería del Software de Ingeniería en Informática de la Universidad de Córdoba y a las modificaciones elaboradas por el equipo Contact Manager sobre la misma.