ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

L.P.E.

Intérprete de Pseudocódigo en Español

Ángel Pavón Pérez

Alberto Jurado Roldán

PROCESADORES DE LENGUAJES

3ER CURSO DEL GRADO SUPERIOR EN INGENIERÍA EN INFORMÁTICA CON MENCIÓN EN COMPUTACIÓN

2º CUATRIMESTRE DEL CURSO 2018-19

CÓRDOBA, 03-06-2019

ÍNDICE

1. Introducción

En este trabajo se ha realizado la implementación de un intérprete de pseudocódigo en español (abreviado IPE) mediante el uso de Flex y Bison.

En el presente documento, se van a abordar los siguientes temas:

- Lenguaje de pseudocódigo: Se expondrán las distintas sentencias del lenguaje (condicionales, bucles…), las cuales estarán en castellano.
- Tabla de símbolos: Se expondrán las clases utilizadas.
- Análisis léxico: Se describirán los componentes léxicos y sus expresiones regulares.
- Análisis sintáctico: Se describirá la gramática de contexto libre del intérprete.
- Código de AST: Se detallará toda la lógica que ejecutan las distintas reglas del intérprete.
- Funciones auxiliares: Se detallarán las distintas funciones auxiliares que hayan sido necesarias en la codificación del intérprete.
- Modo de obtención del intérprete: Aquí se hablará de cómo generar el ejecutable del intérprete.
- Modo de ejecución del intérprete: Se expondrán los 2 modos de ejecución que soporta el intérprete: interactivo y a partir de un fichero.
- Ejemplos: Se presentará un conjunto de ejemplos con los cuales se podrá probar el intérprete.

2. Lenguaje de pseudocódigo

2.1. Componentes léxicos

2.1.1. Palabras reservadas

No distinguen entre mayúsculas y minúsculas y no se pueden usar como identificadores. Las palabras reservadas son las siguientes:

- **_mod** Operador de módulo.
- **_div** Operador de división entera.
- **_o** Operador de disyunción lógica.
- **_y** Operador de conjunción lógica.
- **_no** Operador de negación lógica.
- **leer** Función de lectura para valores numéricos.
- **leer_cadena** Función de lectura para cadenas.

- **escribir** Función de escritura para valores numéricos.
- **escribir_cadena** Función de escritura para cadenas.
- **si** Operador condicional “si”.
- **entonces** Operador del consecuente del condicional.
- **si_no** Operador condicional “si no”.
- **fin_si** Marca de fin de condición.
- **mientras** Operador de bucle “mientras”.
- **hacer** Operador del consecuente del bucle “mientras” y del bucle “para”.
- **fin_mientras** Marca de fin del bucle “mientras”.
- **repetir** Operador de bucle “repetir”.
- **hasta** Operador del consecuente del bucle “repetir”.
- **para** Operador de bucle “para”.
- **fin_para** Marca de fin de bucle “para”.
- **desde** Operador de la condición del bucle “para”.
- **paso** Operador de bucle “paso”.
- **_borrar** Función que permite borrar la salida de la terminal.
- **_lugar** Función que permite situar el cursor de la terminal en las coordenadas deseadas.

2.1.1.1. Constantes numéricas

No permiten ser modificadas. En este intérprete, se han declarado las siguientes:

- **pi**
  - Número π. Su valor está definido como 3.14159265358979323846.
- **e**
  - Número *e*. Su valor está definido como 2.71828182845904523536.
- **gamma**
  - Constante de Euler-Mascheroni (γ). Su valor está definido como 0.57721566490153286060.
- **deg**

- Grado por radián: Su valor está definido como 57.29577951308232087680.

- phi
  - Proporción áurea (ϕ). Su valor está definido como 1.61803398874989484820.

- Cualquier otra cosa, vale 0 por defecto.

2.1.2. Identificadores

Los identificadores están compuestos por una serie de letras, dígitos y el carácter "_".

Deben comenzar por una letra y no pueden acabar con el carácter "_" ni tener 2 "_" seguidos.

USO CORRECTO	USO INCORRECTO
dato	_dato
dato_1	dato_
dato_1_a	dato__1

2.1.3. Número

Los números pueden ser:

- Enteros
- Reales de punto fijo
- Reales con notación científica

Todos ellos son tratados en conjunto como números.

2.1.4. Cadena

Están compuestas por una serie de caracteres delimitados por comillas simples (‘’).

Para poder usar las comillas simples dentro de una cadena, se ha de anteponer una barra (\) a cada comilla simple.

Ejemplo:

- ‘Esto es una cadena en \’IPE\’. Puedo usar saltos\nde\nlínea y\ttabulaciones.’

2.1.5. Operadores

- Operador de asignación :=
- Operadores aritméticos:
  - Suma +
    - Unario + <nº>|<id>
    - Binario <nº>|<id> + <nº>|<id>
  - Resta -
    - Unario - <nº>|<id>
    - Binario <nº>|<id> - <nº>|<id>
  - Producto *
  - Cociente /
  - Cociente entero _div
  - Módulo _mod
  - Potencia **
- Operador alfanumérico:
  - Concatenación ||
- Operadores relacionales de números y cadenas:
  - Menor que <
  - Menor o igual que <=
  - Mayor que >
  - Mayor o igual que >=
  - Igual que =
  - Distinto de <>
- Operadores lógicos:
  - Disyunción lógica _o
  - Conjunción lógica _y
  - Negación lógica _no

2.1.6. Comentarios

- Multilínea:
  - Van separados por el símbolo #.
  - Ejemplo:

```
#
COMENTARIO
MULTI
```

LÍNEA

#

- De 1 línea:
  - Empiezan con el símbolo **@**.
  - Ejemplo:

    @*COMENTARIO DE 1 LÍNEA*

2.1.7. Fin de sentencia

Se usa **;** para indicar el final de una sentencia.

2.2. Sentencias

2.2.1. Asignación

- <identificador> **:=** <expresión numérica>
  - Declara a identificador como una variable numérica y le asigna el valor de la expresión numérica.
  - Nota: Las expresiones numéricas están formadas por números, variables numéricas y operadores numéricos.
- <identificador> **:=** <expresión alfanumérica>
  - Declara a identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.
  - Nota: Las expresiones alfanuméricas están formadas por cadenas, variables alfanuméricas y el operador alfanumérico de concatenación (||).

2.2.2. Lectura

- **leer**(<identificador>)
  - Declara a identificador como una variable numérica y le asigna el número leído por terminal.
- **leer_cadena**(<identificador>)
  - Declara a identificador como una variable alfanumérica y le asigna la cadena leída por terminal (sin las comillas exteriores).

2.2.3. Escritura

- **escribir**(<expresión numérica>)

- El valor de la expresión numérica es escrito en la salida de la terminal.

- **escribir_cadena**(<expresión alfanumérica>)
  - La cadena (sin las comillas exteriores) es escrita en la salida de la terminal.
  - Se interpretan los comandos de saltos de línea (\n) y tabuladores (\t).

2.2.4. Sentencias de control

- 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>] → **OPCIONAL**
  hacer
  <sentencias>
  fin_para

2.2.5. Comandos especiales

- **_borrar**
  - Borra la salida de la terminal
- **_lugar**(<expresión numérica 1>, <expresión numérica 2>)
  - Coloca el cursor de la terminal en las coordenadas indicadas por los valores de las expresiones numéricas.

3. Tabla de símbolos

La tabla de símbolos servirá de ayuda para el proceso de compilación, indicando los distintos símbolos disponibles con sus nombres, tipos y valores. Las distintas clases que se han empleado en la tabla de símbolos son las siguientes:

- **Clase *Constant***: Clase que hereda de *Symbol* y que se ha usado para la declaración de constantes (Variables no modificables).
- **Clase *Keyword***: Clase que hereda de *Symbol* y que se ha usado para la declaración de palabras reservadas (Leer, escribir, etc.).
- **Clase *Variable***: Clase que hereda de *Symbol* y que se ha usado para la declaración de variables.
- **Clase *logicalConstant***: Clase que hereda de *Constant* y que se ha usado para la declaración de constantes booleanas.
- **Clase *logicalVariable***: Clase que hereda de *Variable* y que se ha usado para la declaración de variables booleanas.
- **Clase *mathFunction***: Clase que se ha usado para declarar una serie de funciones matemáticas como la raíz cuadrada o los logaritmos.
- **Clase *numericConstant***: Clase que hereda de *Constant* y que se ha usado para declarar constantes numéricas como PI.
- **Clase *numericVariable***: Clase que hereda de *Variable* y que se ha usado para declarar variables numéricas.
- **Clase *stringVariable***: Clase que hereda de *Variable* y que se ha usado para declarar variables de cadenas.

Además, se han usado otras clases para la inicialización y construcción de la tabla:

- **Clase *Symbol***: Clase que se ha utilizado para declarar los símbolos que pertenecerán a la tabla.

- **Clase *SymbolInterface***: Clase que se ha utilizado para proporcionar una interfaz que utilizarán los distintos símbolos de la tabla.
- **Clase *Table***: Clase que se ha utilizado para declarar la tabla de símbolos.
- **Clase *Init***: Clase que se ha utilizado para inicializar la tabla, en la cual, se declararán las palabras reservadas y las constantes.
- **Clases *BuiltinParameter***: Clases utilizadas para la construcción de los parámetros de la tabla.

4. Análisis léxico

El objetivo del análisis léxico es obtener los componentes léxicos o tokens.

4.1. Descripción de los componentes léxicos y de sus expresiones regulares

4.1.1. Definiciones regulares

- DIGIT
  - [0-9]
- LETTER
  - [a-zA-Z]
- subrayado
  - [_]
- espacio
  - [\t\n]
- numero
  - {DIGIT}+(\.{DIGIT}+)?(E[+\-]?{DIGIT}+)?
- IDENTIFIER
  - {LETTER}({LETTER}|{DIGIT}|{subrayado}({LETTER}|{DIGIT})+)*
- FUNCTION
  - {subrayado}{LETTER}({LETTER}|{DIGIT}|{subrayado}({LETTER}|{DIGIT})+)*
- STRING
  - """([^']|"\\\'")*"""

4.1.2. Reglas

- [\t]
  - Omite el espacio en blanco y la tabulación.
- \n
  - Incrementa el número de líneas cada vez que se realiza un salto de línea.
- ";"
  - Devuelve el token SEMICOLON cuando se encuentra un ";".
- ","
  - Devuelve el token COMMA cuando se encuentra una ",".
- {numero}
  - Cuando encuentra un número, convierte lo encontrado a un flotante y devuelve el token NUMBER.
- {IDENTIFIER}
  - Al encontrar un identificador, se transforma a minúsculas y, después, verifica si se encuentra en la tabla de símbolos y devuelve el token VARIABLE. Si no está, lo inserta en ella. En otro caso, muestra el nombre del identificador y su token y devuelve su token.
- {STRING}
  - Devuelve el token STRING.
- (?i:_borrar)
  - Devuelve el token CLEAN para limpiar la salida de la terminal.
- (?i:_lugar)
  - Devuelve el token MOVE para desplazar el cursor de la terminal.
- "-"
  - Devuelve el token MINUS.
- "+"
  - Devuelve el token PLUS.
- "*"
  - Devuelve el token MULTIPLICATION.

- “/”
    - Devuelve el token DIVISION.
- (?i:_div)
    - Devuelve el token DIVISION_ENTERA.
- “(“
    - Devuelve el token LPAREN.
- “)”
    - Devuelve el token RPAREN.
- “||”
    - Devuelve el token CONCATENACION.
- (?i:_mod)
    - Devuelve el token MODULO.
- “**”
    - Devuelve el token POWER.
- “:=”
    - Devuelve el token ASSIGNMENT.
- “=”
    - Devuelve el token EQUAL.
- “<>”
    - Devuelve el token NOT_EQUAL.
- “>=”
    - Devuelve el token GREATER_OR_EQUAL.
- “<=”
    - Devuelve el token LESS_OR_EQUAL.
- “>”
    - Devuelve el token GREATER_THAN.
- “<”
    - Devuelve el token LESS_THAN.
- (?i:_no)
    - Devuelve el token NOT.
- (?i:_o)
    - Devuelve el token OR.
- (?i:_y)
    - Devuelve el token AND.
- “{“

    - Devuelve el token LETFCURLYBRACKET.
- "}"
    - Devuelve el token RIGHTCURLYBRACKET.
- ^~
    - Se limpia la salida de la terminal, se finaliza el programa con un mensaje y se devuelve 0.
- @.*
    - Ignora el contenido de los comentarios de una sola línea.
- #[^#]+#
    - Ignora el contenido de los comentarios multilínea.
- <<EOF>>
    - Se limpia la salida de la terminal, se finaliza el programa con un mensaje de que se ha llegado al final del fichero y se devuelve 0.
- .
    - En cualquier otro caso, se activa el estado de ERROR.
- _*{IDENTIFIER}_+
    - Se muestra una advertencia de identificador erróneo.
- _+{IDENTIFIER}_*
    - Se muestra una advertencia de identificador erróneo.
- {LETTER}({LETTER}|{DIGIT}|{subrayado}{2,}({LETTER}|{DIGIT})+)*
    - Se muestra una advertencia de identificador erróneo.
- <ERROR>[^0-9+\-*/()\^% \t\n\;a-zA-Z=<>!&]
    - Cuando encuentra un símbolo extraño, todo lo que haya después se considera erróneo.
- <ERROR>(.|\n)
    - Se muestra un mensaje de advertencia generalizado.

5. Análisis sintáctico

El análisis sintáctico utiliza la gramática de contexto libre para definir la sintaxis de las sentencias. De esta forma, a partir de esta gramática, se generará el analizador sintáctico.

5.1. Descripción de la gramática de contexto libre

La gramática de contexto libre estará formada, principalmente, por unos símbolos de la gramática terminales que se corresponden con los componentes léxicos y con unos símbolos de la gramática no terminales. Además, contendrá una serie de reglas de producción, así como acciones semánticas.

5.1.1. Símbolos de la gramática

5.1.1.1. Símbolos terminales (componentes léxicos)

Los símbolos terminales coinciden con los componentes léxicos. De esta forma, los símbolos terminales serán los siguientes:

- SEMICOLON
- PRINT
- READ
- IF
- ELSE
- WHILE
- IF_END
- WHILE_END
- DO
- REPEAT
- UNTIL
- FOR
- FROM
- FOR_END
- PASO
- MOVE
- CLEAN
- PRINT_STRING
- READ_STRING
- THEN
- LETFCURLYBRACKET
- RIGHTCURLYBRACKET
- ASSIGNMENT

- COMMA
- NUMBER
- BOOL
- STRING
- VARIABLE
- CONSTANT
- BUILTIN
- COMMENTARY
- FUNCTION
- OR
- AND
- GREATER_OR_EQUAL
- LESS_OR_EQUAL
- GREATER_THAN
- LESS_THAN
- EQUAL
- NOT_EQUAL
- NOT
- PLUS
- MINUS
- MULTIPLICATION
- DIVISION
- MODULO
- DIVISION_ENTERA
- CONCATENACION
- LPAREN
- RPAREN
- UNARY
- POWER

5.1.1.2. Símbolos no terminales

Los símbolos no terminales serán aquellos cuyas reglas lleven a otros símbolos que sean terminales o no terminales. Por lo tanto, los símbolos no terminales serán los siguientes:

- program
- stmtlist
- stmt
- block
- if
- while
- until
- for
- move
- clean
- cond
- print
- print_string
- read
- read_string
- exp
- listOfExp
- restOfListOfExp

5.1.2. Reglas de producción de la gramática y acciones semánticas

- program → stmtlist
  - Crea una nueva clase AST y la asigna a la raíz
- stmtlist →
  - Crea una lista vacía de sentencias
- stmtlist → stmtlist stmt
  - Añade una sentencia a la lista de sentencias
- stmtlist → stmtlist error
  - Copia la lista cuando se produce un error
- stmt → SEMICOLON
  - Crea una sentencia vacía
- stmt → asgn SEMICOLON
  - Decimos que una sentencia puede ser una asignación y simplificamos la regla de asignación factorizando el punto y coma.
- stmt → print SEMICOLON

    - Decimos que una sentencia puede ser una escritura y simplificamos la regla de escritura factorizando el punto y coma.
- stmt → read SEMICOLON
    - Decimos que una sentencia puede ser una lectura y simplificamos la regla de lectura factorizando el punto y coma.
- stmt → if
    - Decimos que una sentencia puede ser un condicional.
- stmt → while
    - Decimos que una sentencia puede ser un bucle mientras.
- stmt → block
    - Decimos que una sentencia puede ser un bloque.
- stmt → comment
    - Decimos que una sentencia puede ser un comentario. (Aunque ahora no realiza nada, se ha dejado por si en un futuro al encontrar un comentario queremos realizar alguna acción).
- stmt → until SEMICOLON
    - Decimos que una sentencia puede ser un bucle hasta.
- stmt → for
    - Decimos que una sentencia puede ser un bucle para.
- stmt → move SEMICOLON
    - Decimos que una sentencia puede ser una función de lugar y la simplificamos factorizando el punto y coma.
- stmt → clean SEMICOLON
    - Decimos que una sentencia puede ser una función de limpieza y la simplificamos factorizando el punto y coma.
- stmt → print_string SEMICOLON
    - Decimos que una sentencia puede ser una función de escritura de cadena y la simplificamos factorizando el punto y coma.
- stmt → read_string SEMICOLON

    - Decimos que una sentencia puede ser una función de lectura de cadena y la simplificamos factorizando el punto y coma.
- block → LEFTCURLYBRACKET stmtlist RIGHTCURLYBRACKET
    - Decimos que un bloque esta comprendido entre dos llaves ({}) y que tiene dentro una lista de sentencias. Además, crearemos un bloque de sentencias.
- if → IF cond THEN stmtlist IF_END
    - Decimos que un condicional está compuesto por un token IF, una condición, un token THEN, una lista de sentencias y un token IF_END. Además, esta regla creará una sentencia If a partir de las condiciones y la lista de sentencias.
- if → IF cond THEN stmtlist ELSE stmtlist IF_END
    - Igual que el anterior, pero con el añadido del token ELSE.
- if → IF cond stmtlist ELSE stmtlist IF_END
    - Crea un error que indica que falta el token THEN.
- while → WHILE cond DO stmtlist WHILE_END
    - Decimos que un bucle está compuesto por un token WHILE, una condición, un token DO, una lista de sentencias y un token WHILE_END. Además, esta regla crea una sentencia While.
- until → REPEAT stmtlist UNTIL cond
    - Parecido al anterior, pero con otros tokens y creando una sentencia Until.
- for → FOR VARIABLE FROM exp UNTIL exp DO stmtlist FOR_END
    - Parecido a los anteriores, pero con otros tokens y creando una asignación además de la sentencia For.
- for → FOR VARIABLE FROM exp UNTIL exp PASO NUMBER DO stmtlist FOR_END
    - Igual que el anterior, pero podemos indicarle mediante un paso como vamos a iterar.
- move → MOVE LPAREN NUMBER COMMA NUMBER RPAREN

    - Regla que crea una sentencia Move para mover el cursor de la terminal.
- move → MOVE LPAREN STRING COMMA STRING RPAREN
    - Regla que indica que se han pasado parámetros incorrectos a la sentencia Move.
- clean → CLEAN
    - Regla que crea una sentencia Clean para limpiar la terminal.
- cond → LPAREN exp RPAREN
    - Regla que indica que una condición esta compuesta por dos paréntesis y entre ellos una expresión.
- asgn → VARIABLE ASSIGNMENT exp
    - Regla que crea una asignación a una variable de una expresión.
- asgn → VARIABLE ASSIGNMENT asgn
    - Regla que crea una asignación a una variable de otra asignación.
- asgn → CONSTANT ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → CONSTANT ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → PRINT ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → PRINT ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → READ ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → READ ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.

- asgn → READ_STRING ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → READ_STRING ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → IF ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → IF ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → ELSE ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → ELSE ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → WHILE ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → WHILE ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → IF_END ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → IF_END ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → WHILE_END ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → WHILE _END ASSIGNMENT asgn

    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → DO ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → DO ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → REPEAT ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → REPEAT ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → UNTIL ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → UNTIL ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → FOR ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → FOR ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → FROM ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → FROM ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → FOR_END ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.

- asgn → FOR_END ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → PASO ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → PASO ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → AND ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → AND ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → OR ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido
- asgn → OR ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → NOT ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → NOT ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → MODULO ASSIGNMENT exp
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido.
- asgn → MODULO ASSIGNMENT asgn
    - Regla que produce un error al intentar realizar una asignación a un elemento no valido
- print → PRINT exp

    - Regla que crea una sentencia de escritura a partir de un token PRINT y una expresión.
- print_string → PRINT_STRING LPAREN exp RPAREN
    - Regla que crea una sentencia de escritura de cadenas a partir de un token PRINT_STRING y una expresión.
- read → READ LPAREN VARIABLE RPAREN
    - Regla que crea una sentencia de lectura a partir de un token READ, unos paréntesis y una expresión.
- read → READ LPAREN CONSTANT RPAREN
    - Regla que produce un error al intentar leer una constante.
- read_string → READ_STRING LPAREN VARIABLE RPAREN
    - Regla que crea una sentencia de lectura de cadenas a partir de un token READ, unos paréntesis y una expresión.
- read_string → READ_STRING LPAREN CONSTANT RPAREN
    - Regla que produce un error al intentar leer como cadena una constante.
- exp → NUMBER
    - Regla que indica que expresión puede ser un número y crea su correspondiente nodo.
- exp → STRING
    - Regla que indica que expresión puede ser una cadena y crea su correspondiente nodo.
- exp → exp CONCATENATION exp
    - Regla que indica que expresión puede ser una concatenación de dos expresiones y crea su correspondiente nodo
- exp → exp PLUS exp
    - Regla que indica que expresión puede ser una suma de dos expresiones y crea su correspondiente nodo
- exp → exp MINUS exp
    - Regla que indica que expresión puede ser una resta de dos expresiones y crea su correspondiente nodo
- exp → exp MULTIPLICATION exp

    - Regla que indica que expresión puede ser una multiplicación de dos expresiones y crea su correspondiente nodo.
- exp → exp DIVISION exp
    - Regla que indica que expresión puede ser una división de dos expresiones y crea su correspondiente nodo.
- exp → exp DIVISION_ENTERA exp
    - Regla que indica que expresión puede ser una división entera de dos expresiones y crea su correspondiente nodo.
- exp → LPAREN exp RPAREN
    - Regla que indica que una expresión puede estar compuesta por otra expresión entre paréntesis y solo copia la expresión para que sea evaluada posteriormente en otra regla.
- exp → PLUS exp %prec UNARY
    - Regla que indica que una expresión puede estar compuesta por el token PLUS y una expresión creando así un operador unitario.
- exp → MINUS exp %prec UNARY
    - Regla que indica que una expresión puede estar compuesta por el token MINUS y una expresión creando así un operador unitario.
- exp → exp MODULO exp
    - Regla que indica que expresión puede ser un módulo de dos expresiones y crea su correspondiente nodo.
- exp → exp POWER exp
    - Regla que indica que expresión puede ser una potencia de dos expresiones y crea su correspondiente nodo.
- exp → VARIABLE
    - Regla que indica que una expresión puede ser una variable, creando así su correspondiente nodo.
- exp → CONSTANT
    - Regla que indica que una expresión puede ser una constante, creando así su correspondiente nodo.

- exp → BUILTIN LPAREN listOfExp RPAREN
    - Regla que indica que una expresión puede ser una función builtin con sus paréntesis y una lista de expresiones, creando así su correspondiente nodo según el número de elementos.
- exp → exp GREATER_THAN exp
    - Regla que indica que expresión puede ser un operador relacional mayor entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp GREATER_OR_EQUAL exp
    - Regla que indica que expresión puede ser un operador relacional mayor igual entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp LESS_THAN exp
    - Regla que indica que expresión puede ser un operador relacional menor entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp LESS_OR_EQUAL exp
    - Regla que indica que expresión puede ser un operador relacional menor o igual entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp EQUAL exp
    - Regla que indica que expresión puede ser un operador relacional igual entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp NOT_EQUAL exp
    - Regla que indica que expresión puede ser un operador relacional no igual entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp AND exp
    - Regla que indica que expresión puede ser un operador lógico “y” entre otras dos expresiones y crea su correspondiente nodo.
- exp → exp OR exp

    - Regla que indica que expresión puede ser un operador lógico “o” entre otras dos expresiones y crea su correspondiente nodo.
- exp → NOT exp
    - Regla que indica que expresión puede ser un operador lógico de negación seguido de otra expresión y crea su correspondiente nodo.
- comment → COMMENTARY
    - Regla que crea un comentario, como hemos dicho actualmente no realiza nada, es por si en un futuro decidimos que el comentario provoque otras acciones como por ejemplo que según un parámetro introducido en el interprete los comentarios se impriman por pantalla al depurar.
- listOfExp →
    - Regla que crea una lista de expresiones.
- listOfExp → exp restOfListOfExp
    - Regla que indica que una lista de expresiones esta compuesta por una expresión y el resto de la lista. De esta forma esta regla añade la expresión a la lista.
- restOfListOfExp →
    - Regla que crea una lista de expresiones.
- restOfListOfExp → COMMA exp restOfListOfExp
    - Regla que indica que una lista de expresiones está compuesta por un token COMMA, una expresión y el resto de la lista. De esta forma esta regla añade la expresión a la lista.

6. Código de AST

Las clases que se han utilizado en AST y que se llaman en el análisis sintáctico para evaluar las expresiones sintácticas devolviendo su correspondiente valor son las siguientes:

- ***ExpNode***: Declara un nodo de una expresión con unas funciones virtuales que serán sobrescritas por las clases que hereden de ella.

- ***VariableNode***: Clase que hereda de *ExpNode* y que declara un nodo variable con sus correspondientes funciones para escribir y evaluar, así como para obtener el tipo.
- ***ConstantNode***: Clase que hereda de *ExpNode* y que declara un nodo constante con sus correspondientes funciones para escribir y evaluar, así como para obtener el tipo.
- ***NumberNode***: Clase que hereda de *ExpNode* y que declara un nodo de tipo numérico con sus correspondientes funciones para escribir y evaluar, así como para obtener el tipo.
- ***StringNode***: Nueva clase que hereda de *ExpNode* y que declara un nodo de tipo cadena para poder evaluarlo, así como imprimirlo y obtener su tipo.
- ***UnaryOperatorNode***: Clase del operador unitario que hereda de *ExpNode* para declarar los operadores unitarios.
- ***NumericUnaryOperatorNode***: Clase que hereda de *UnaryOperatorNode* y que declara los operadores numéricos unitarios.
- ***LogicalUnaryOperatorNode***: Clase que hereda de *UnaryOperatorNode* y que declara los operadores lógicos unitarios.
- ***UnaryMinusNode***: Clase que hereda de *NumericUnaryOperatorNode* y que declara el operador unitario - para los valores numéricos (p. ej: -3).
- ***UnaryPlusNode***: Clase que hereda de *NumericUnaryOperatorNode* y que declara el operador unitario + para los valores numéricos (p. ej: +5).
- ***OperatorNode***: Clase que hereda de *ExpNode* y que declara un nodo de tipo operación. Requiere para su constructor 2 nodos de expresión a la izquierda y derecha del operador.
- ***NumericOperatorNode***: Clase que hereda de *OperatorNode* y que declara un operador de tipo numérico.
- ***RelationalOperatorNode***: Clase que hereda de *OperatorNode* y que declara un operador de tipo relacional.
- ***LogicalOperatorNode***: Clase que hereda de *OperatorNode* y que declara un operador de tipo lógico.
- ***StringOperatorNode***: Nueva clase que hereda de *OperatorNode* y que declara un operador de tipo cadena, es decir, operará sobre cadenas.
- ***PlusNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de suma.
- ***MinusNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de resta.

- ***MultiplicationNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de multiplicación.
- ***DivisionNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de división.
- ***DivisionEnteraNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de división entera.
- ***ModuloNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de modulo.
- ***PowerNode***: Clase que hereda de *NumericOperatorNode* y que declara el operador de potencia.
- ***ConcatenacionNode***: Nueva clase que hereda de *StringOperatorNode* y que declara el operador de concatenación de cadenas.
- ***BuiltinFunctionNode***: Clase que hereda de *ExpNode* y que declara la construcción de funciones auxiliares.
- ***BuiltinFunctionNode_0***: Clase que proporciona funcionalidad a las funciones auxiliares.
- ***BuiltinFunctionNode_1***: Clase que proporciona funcionalidad a las funciones auxiliares.
- ***BuiltinFunctionNode_2***: Clase que proporciona funcionalidad a las funciones auxiliares.
- ***GreaterThanNode***: Clase que hereda de *RelationalOperatorNode* y que declara el operador mayor.
- ***GreaterOrEqualNode***: Clase que hereda de *RelationalOperatorNode* y que declara el operador mayor o igual.
- ***LessThanNode***: Clase que hereda de *RelationalOperatorNode* y que declara el operador menor.
- ***LessOrEqualNode***: Clase que hereda de *RelationalOperatorNode* y que declara el operador menor o igual.
- ***EqualNode***: Clase que hereda de *RelationalOperatorNode* y que declara el operador igual.
- ***NotEqualNode***: Clase que hereda de *RelationalOperatorNode* y que declara el operador distinto.
- ***AndNode***: Clase que hereda de *LogicalOperatorNode* y que declara el operador de “y” lógico.

- ***OrNode***: Clase que hereda de *LogicalOperatorNode* y que declara el operador de "o" lógico.
- ***NotNode***: Clase que hereda de *LogicalUnaryOperatorNode* y que declara el operador negación.
- ***Statement***: Clase que declara una línea de declaraciones.
- ***AssignmentStmt***: Clase que hereda de la clase *Statement* y que declara una declaración de asignación.
- ***PrintStmt***: Clase que hereda de la clase *Statement* y que declara una declaración de escritura.
- ***PrintStringStmt***: Nueva clase que hereda de la clase *Statement* y que declara una declaración de escritura de cadenas.
- ***ReadStmt***: Clase que hereda de la clase *Statement* y que declara una declaración de lectura.
- ***ReadStringStmt***: Nueva clase que hereda de la clase *Statement* y que declara una declaración de lectura de cadena.
- ***EmptyStmt***: Clase que hereda de la clase *Statement* y que declara una declaración vacía.
- ***IfStmt***: Clase que hereda de la clase *Statement* y que declara una declaración del condicional.
- ***WhileStmt***: Clase que hereda de la clase *Statement* y que declara una declaración del bucle mientras.
- ***BlockStmt***: Clase que hereda de la clase *Statement* y que declara un bloque de declaraciones.
- ***UntilStmt***: Nueva clase que hereda de la clase *Statement* y que declara una declaración del bucle hasta.
- ***ForStmt***: Nueva clase que hereda de la clase *Statement* y que declara una declaración del bucle para.
- ***MoveStmt***: Nueva clase que hereda de la clase *Statement* y que declara una declaración de lugar para mover el cursor a un lugar especificado.
- ***CleanStmt***: Nueva clase que hereda de la clase *Statement* y que declara una declaración de limpiar para limpiar la terminal.

7. Funciones auxiliares

No ha sido necesaria la implementación de funciones auxiliares nuevas.

No obstante, se van a enumerar las funciones auxiliares que ya estaban implementadas con anterioridad:

- sin: Permite calcular el seno.
- cos: Permite calcular el coseno.
- atan: Permite calcular el arco tangente.
- log: Permite calcular el logaritmo natural.
- log10: Permite calcular el logaritmo en base 10.
- exp: Permite calcular el exponencial.
- sqrt: Permite calcular la raíz cuadrada.
- integer: Permite obtener la parte entera de un número.
- fabs: Permite obtener el valor absoluto de un número.
- atan2: Permite obtener el arco tangente doble.
- random: Genera un número aleatorio comprendido entre 0 y 1.

8. Modo de obtención del intérprete

8.1 Ficheros

En este punto, se van a exponer con más detalle los distintos archivos que han sido necesarios para la creación del interprete, así como su distinta organización.

8.1.1 Carpeta ast

- **ast.cpp**: Archivo en el que se implementan las funciones de las clases del AST.
- **ast.hpp**: Archivo en el que se declaran todas las clases del AST (véase el punto 6).

En el makefile, la ruta de este directorio se guarda en la variable *ast-dir*.

8.1.2 Carpeta error

- **error.cpp**: Implementación de las funciones utilizadas para avisar al usuario de los distintos tipos de errores
- **error.hpp**: Declaración de las funciones utilizadas para avisar al usuario de los distintos tipos de errores.

En el makefile, la ruta de este directorio se guarda en la variable *error-dir*.

8.1.3 Carpeta includes

- **macros.hpp**: Archivo que declara las distintas macros que se pueden utilizar en el resto de los archivos, por ejemplo, para cambiar el color a la hora de imprimir por terminal.

8.1.4 Carpeta parser

- **interpreter.l**: Archivo del interprete léxico en el que se declaran las expresiones regulares y los tokens que se devuelven.
- **interpreter.y**: Archivo del interprete sintáctico en el que se declaran las sentencias de tokens y lo que generan.

En el makefile, la ruta de este directorio se guarda en la variable *parser-dir*.

8.1.5 Carpeta table

- **builtin.cpp**: Sirve para las funciones auxiliares.
- **builtin.hpp**: Sirve para las funciones auxiliares.
- **builtinParameter0.cpp**: Sirve para las funciones auxiliares.
- **builtinParameter0.hpp**: Sirve para las funciones auxiliares.
- **builtinParameter1.cpp**: Sirve para las funciones auxiliares.
- **builtinParameter1.hpp**: Sirve para las funciones auxiliares.
- **builtinParameter2.cpp**: Sirve para las funciones auxiliares.
- **builtinParameter2.hpp**: Sirve para las funciones auxiliares.
- **constant.cpp**: Archivo donde se implementan las funciones de la clase *Constant*.
- **constant.hpp**: Archivo donde se declaran las funciones de la clase *Constant*.
- **init.cpp**: Archivo donde se implementan las funciones que inicializan la tabla.
- **init.hpp**: Archivo donde se declaran la clase *Init* y sus funciones.
- **keyword.cpp**: Archivo donde se implementan las funciones de la clase *Keyword*.
- **keyword.hpp**: Archivo donde se declaran la clase *Keyword* y sus funciones.
- **logicalConstant.cpp**: Archivo donde se implementan las funciones de la clase *logicalConstant*.

- **logicalConstant.hpp**: Archivo donde se declaran la clase *logicalConstant* y sus funciones.
- **logicalVariable.cpp**: Archivo donde se implementan las funciones de la clase *logicalVariable*.
- **logicalVariable.hpp**: Archivo donde se declaran la clase *logicalVariable* y sus funciones.
- **mathFunction.cpp**: Archivo donde se implementan las funciones matemáticas.
- **mathFunction.hpp**: Archivo donde se declaran las funciones matemáticas.
- **numericConstant.cpp**: Archivo donde se implementan las funciones de la clase *numericConstant*.
- **numericConstant.hpp**: Archivo donde se declaran la clase *numericConstant* y sus funciones.
- **numericVariable.cpp**: Archivo donde se implementan las funciones de la clase *numericVariable*.
- **numericVariable.hpp**: Archivo donde se declaran la clase *numericVariable* y sus funciones.
- **stringVariable.cpp**: Archivo donde se implementan las funciones de la clase *stringVariable*.
- **stringVariable.hpp**: Archivo donde se declaran la clase *stringVariable* y sus funciones.
- **symbol.cpp**: Archivo donde se implementan las funciones de la clase *Symbol*.
- **symbol.hpp**: Archivo donde se declaran la clase *Symbol* y sus funciones.
- **symbolInterface.hpp**: Archivo donde se declara la interfaz de *Symbol*.
- **table.cpp**: Archivo donde se implementan las funciones de la clase *Table*.
- **table.hpp**: Archivo donde se declara la clase *Table*.
- **tableInterface.hpp**: Archivo que declara la interfaz de la clase *Table*.
- **variable.cpp**: Archivo donde se implementan las funciones de la clase *Variable*.
- **variable.hpp**: Archivo donde se declaran la clase *Variable* y sus funciones.

En el makefile, la ruta de este directorio se guarda en la variable *table-dir*.

8.1.6 Otros archivos

- **Doxyfile**: Archivo en el que se detalla la configuración de la documentación generada por Doxygen.
- **footerFile**: Archivo en el que se detalla la configuración de los pies de página de la documentación generada por Doxygen.
- **headerFile**: Archivo en el que se detalla la configuración de las cabeceras de página de la documentación generada por Doxygen.
- **interpreter.cpp**: Archivo principal en el que se implementa el main y que cargará el intérprete en modo interactivo o a partir de un fichero.
- **logo.png**: Logo de la universidad de Córdoba que usará la documentación generada por Doxygen.
- **makefile**: Archivo para la compilación del código. Se explicará en mayor profundidad en el siguiente punto.
- **styleSheetFile**: CSS que utilizará la web generada por Doxygen.
- **favicon.gif y favicon.ico**: Iconos que utiliza la web generada por Doxygen.

8.2 Makefile

Este archivo permite la construcción del ejecutable y de la documentación, así como la eliminación de estos.

Cuenta con 4 modos:

- Sin argumentos: Genera solo el ejecutable.
- Argumento *doc*: Genera solo la documentación.
- Argumento *all*: Genera tanto el ejecutable como la documentación.
- Argumento *clean*: Borra todos los ficheros y directorios generados en los procesos de compilación del ejecutable y generación de la documentación.

9. Modo de ejecución del intérprete

Este intérprete tiene 2 modos de ejecutarse: un modo interactivo y otro a partir de un fichero.

9.1. Interactivo

Este modo de ejecución consiste en ejecutar el intérprete sin pasarle ningún parámetro. De esta forma, el usuario podrá ir escribiendo por terminal aquellos comandos que quiera que se ejecuten.

Este modo ha sido de gran ayuda para probar las distintas sentencias conforme se iban desarrollando, así como para poder ir viendo el valor de algunas variables a lo largo de la ejecución.

Sin embargo, en este modo se encuentra uno de los puntos débiles del intérprete: cuando se quieren ejecutar bucles o sentencias condicionales, al requerir de una lista de declaraciones, estas se van ejecutando a medida que se escriben y, una vez se acaba la sentencia principal, se ejecuta dicho bucle o condición.

9.2. A partir de un fichero

Este modo de ejecución consiste en ejecutar el intérprete pasándole como argumento el nombre del fichero a interpretar con su correspondiente extensión ".e".

Este modo no tiene el punto débil que tenía el anterior modo y permite la ejecución de una serie de comandos de una forma más sencilla al tener solo que cargar el fichero deseado. Esto ha sido de gran utilidad al querer probar una serie de sentencias seguidas para ver el comportamiento del intérprete, ya que, al introducirlas en un fichero, no es necesario introducirlas cada vez que se llame al interprete. Tan solo hay que pasarle el fichero donde han sido declaradas. Esto también permite poder ejecutar los ejemplos que se han desarrollado.

10. Ejemplos

Para probar este interprete, además de utilizar los ejemplos proporcionados por el profesor, se han elaborado ejemplos propios para ponerlo a prueba y tener, así, la certeza de que ejecuta como es debido las sentencias creadas y de que alerta de los posibles errores esperados.

10.1. Ejemplo 1: Fibonacci

Ejemplo que calcula el valor de una sucesión de Fibonacci a partir de un número entero introducido por el usuario.

10.2. Ejemplo 2: Juego de la patata caliente

Ejemplo que simula el juego de la patata caliente. Se calcula un número aleatorio y el usuario tiene que adivinarlo a través de las pistas que se le proporcionan antes de que pasen el número de iteraciones establecidas.

10.3. Ejemplo 3: Fichero con extensión incorrecta

Ejemplo con una extensión ".txt" para comprobar que un fichero sin la extensión ".e" no puede ser abierto.

10.4. Ejemplo 4: Piedra, papel o tijera

Ejemplo en el que se realiza el juego de piedra papel o tijera mostrando por pantalla los movimientos escogidos tanto por el usuario como por la máquina.

10.5. Ejemplo 5: Piedra, papel o tijera, pero con errores

Similar al anterior, pero con errores puestos adrede para verificar el correcto funcionamiento del control de errores del intérprete.

11. Conclusiones

11.1. Reflexión sobre el trabajo realizado

El trabajo que realizado ha servido, en gran medida, para comprender el procedimiento necesario para generar un intérprete de pseudocódigo, desde el análisis léxico hasta el análisis sintáctico y utilizando los controles de errores y la tabla de símbolos.

11.2. Puntos fuertes y puntos débiles del intérprete desarrollado

Los puntos fuertes este intérprete son la posibilidad de desarrollar programas simples al nivel de otros lenguajes de programación, pero con un lenguaje de pseudocódigo, permitiendo así su mejor entendimiento por parte de los parlantes de la lengua castellana. Gracias a ser un intérprete ligero, se podrán probar programas sencillos y no será necesaria la memoria que ocupa un ejecutable.

Por otra parte, uno de los puntos débiles más importantes de este interprete es la opción interactiva. Esto es debido a que, a pesar de poder programar interactivamente, aquellas sentencias que tengan una lista de

sentencias, como los bucles o los condicionales, se ejecutarán a medida que se van escribiendo y, una vez sea cerrada dicha lista, ya se ejecutará el bucle o el condicional pertinente. Otro posible punto débil y que podría ser una futura mejora cuando se conozca con más detalle el lenguaje que se está interpretando es el tema de los controles de errores. Esto se debe a que siempre se pueden controlar mejor los errores en las reglas de producción para dar más detalle al usuario de lo que está fallando.

12. Bibliografía / Referencias web

- Nicolás Luis Fernández García. 2019. *Temario de la asignatura de Procesadores de Lenguajes* [documentos PDF]. Córdoba: Escuela Politécnica Superior de Córdoba (Universidad de Córdoba). [Consulta: 28-05-2019]. Disponible* en:
  - https://moodle.uco.es/m1819/course/view.php?id=1292
- Nicolás Luis Fernández García. 2019. *Ejemplos de Flex* [ficheros comprimidos en un fichero ZIP]. Córdoba: Escuela Politécnica Superior de Córdoba (Universidad de Córdoba). [Consulta: 28-05-2019]. Disponible* en:
  - https://moodle.uco.es/m1819/pluginfile.php/1539/course/section/46907/Ejemplos-Flex.zip
- Nicolás Luis Fernández García. 2019. *Ejemplos de Bison* [ficheros comprimidos en un fichero ZIP]. Córdoba: Escuela Politécnica Superior de Córdoba (Universidad de Córdoba). [Consulta: 28-05-2019]. Disponible* en:
  - https://moodle.uco.es/m1819/pluginfile.php/1539/course/section/46907/bison.zip

***NOTA:** El contenido de estos enlaces solo es visible para aquellos usuarios que estén matriculados en la asignatura de Procesadores de Lenguajes del curso 2018-2019 y puede ser inaccesible en un futuro cercano.