Guía de Trabajos Prácticos

# Asignatura: Diseño de Compiladores

# Trabajo práctico 1

# Título: Introducción

# Objetivos: Investigar herramientas de diseño y desarrollo de compiladores.

1. Investigar herramientas de diseño y desarrollo de compiladores.

# Trabajo práctico 2

# Título: Análisis léxico

# Objetivos: Comprender el funcionamiento del analizador léxico.

1. Diseñar un DT para reconocer los siguientes componentes léxicos:

LETRAS: cualquier secuencia de una o más letras.

ENTERO: cualquier secuencia de uno o más números (si tiene más de un número, no deberá comenzar por 0)

ASIGNAR: la secuencia =.

SUMAR: la secuencia +.

RESTAR: la secuencia -.

IMPRIMIR: la palabra reservada **print**.

1. Para el lenguaje generado por la expresión regular (a | b)\*abb, crear el DT, generar la tabla de transiciones y un programa en Java para implementar la tabla.

# Trabajo práctico 3

# Título: Análisis sintáctico

# Objetivos: Comprender el funcionamiento del analizador sintáctico.

1. Diseñar una gramática no ambigua para el lenguaje de las expresiones que se pueden construer con **true** y **false** y los operadores booleanos **or**, **and**, **not** y los paréntesis. La precedencia de mayor a menor es **not** **and** **or**. Los dos últimos son asociativos por la derecha.
2. Construir una gramática no ambigua que reconozca todas las declaraciones posibles de variables de los siguientes tipos: int, String, boolean y doublé en Java. Por ejemplo:

int x, y:

String cadena;

double a;

boolean b;

1. Crear los diagramas de sintaxis y el programa para esta gramática:

Programa ::= Declaraciones Sentencias

Declaraciones ::= (Decl “;”)+

Decl ::= Entero Identificador

Sentencias ::= (Asignacion “;”)+

Asignacion ::= ParteIzquierda “=” ParteDerecha

ParteIzquierda ::= Identificador

ParteDerecha ::= Expresion

Expresion ::= (Expresion “+” Expresion) | (Expresion “-” Expresion)

Expresion ::= (Identificador | Numero)

# Trabajo práctico 4

# Título: Análisis sintáctico descendente Objetivos: Comprender el funcionamiento del analizador sintáctico descendente.

1. Sea la siguiente gramática:

S ::= A | a

A ::= b ( E ) S L

L ::= c S | ( E )

E ::= 0 | 1

Comprobar si es LL(1) y si lo es, construir su tabla de análisis y verificar si la entrada siguiente es analizada correctamente: a b (0) a c a

1. Hacer lo mismo que en el caso anterior y si no es LL(1) hacer las modificaciones pertinentes (si se puede) para convertirla en LL(1).

# Trabajo práctico 5

# Título: Análisis sintáctico ascendente

# Objetivos: Comprender el funcionamiento del analizador sintáctico ascendente.

1. Para la siguiente gramática:

S ::= id X | id Y end

X ::= otro | $

Y ::= begin X end | $

Construir el autómata y la tabla de análisis SLR.

1. Para la siguiente gramática:

D ::= var V dospuntos T puntocoma

V ::= id coma V

V ::= id

T ::= integer

T ::= boolean

Construir el autómata y la tabla SLR. Analizar la cadena var id coma id dospuntos integer puntocoma

# Trabajo práctico 6

# Título: Tabla de tipos y de símbolos

# Objetivos: Implementar tabla de tipos de símbolos.

1. Dado el siguiente programa en un sublenguaje de Pascal, suponer que este sublenguaje tiene dos tipos predefinidos: integer y Boolean. Suponer que es sensible a las mayúsculas. Sin tener en cuenta las direcciones y mediante la implementación de solo dos tablas, una para los tipos y otra para los símbolos, mostrar el estado de ambas tablas tras procesar las líneas 4, 7, 12, 16, 20 y 26.

|  |  |
| --- | --- |
| 1: | program p; |
| 2: |  |
| 3: | type vector = array[0..7] of integer; |
| 4: | var x : integer; v : vector; |
| 5: |  |
| 6: | function factorial (x: integer): integer |
| 7: | var y, z : integer; |
| 8: | begin |
| 9: | y := 1; |
| 10: | for z := 1 to x do y := y \* z; |
| 11: | factorial := y; |
| 12: | end; |
| 13: |  |
| 14: | function suma (v: vector; k: integer): vector |
| 15: | type vect = array[1..8] of integer; |
| 16: | var x: integer; v1: vect; |
| 17: | begin |
| 18: | for x := 0 to k do v1[x+1] := v[x]; |
| 19: | suma := v1 |
| 20: | end; |
| 21: |  |
| 22: | begin |
| 23: | x := 5: |
| 24: | x := factorial(x); |
| 25: | for x := 0 to 7 do v[x] := x; |
| 26: | v := suma(v, x); |
| 27: | end. |

# Trabajo práctico 7

# Título: Análisis semántico

# Objetivos: Comprender el funcionamiento del analizador semántico.

1. Construir un ETDS para traducer declaraciones de variables en Modula 2 a C. La gramática en Modula 2 sería:

S ::= VAR id:T;

T ::= ARRAY [num ..num] OF T

T ::= REAL | INTEGER | CHAR

1. Realizar un esquema de traducción para comprobar los tipos en las expresiones de un sublenguaje del tipo Pascal. Las reglas para las expresiones son:

E ::= E1 + E2;

E ::= E1 \* E2;

E ::= ( E1 )

E ::= num | id

1. Utilizar la información del ejercicio anterior y comprobar que las sentencias son válidas semánticamente hablando:

S ::= id := E

S ::= if E then S1

S ::= while E do S1

S ::= S1; S2

# Trabajo práctico 8

# Título: Compilador

# Objetivos: Integrar todos los conceptos teóricos vistos.

Traductor para un sencillo lenguaje para lógica de proposiciones.

La salida de un programa en lenguaje L-0 es un archivo de texto con el resultado de la ejecución del programa. Es decir, no se obtiene un ejecutable sino directamente la salida del programa ejecutado.

El programa a compilar se escribirá en un archivo de texto con cualquier nombre y extensión. El resultado de la traducción será obtenido en otro archivo de texto con el mismo nombre que el programa a compilar pero con la extensión .log.

Un programa consiste en una serie de instrucciones seguidas de un terminador, que en este caso es un punto y coma.

Los espacios en blanco se ignoran (salvo dentro de las cadenas de texto).

Las diferentes instrucciones que se admiten son:

* Asignación: consta de dos partes separadas por el signo igual (“= “). La parte izquierda debe ser una variable lógica y la derecha puede ser cualquier expresión lógica o proposición lógica.
* Imprimir extensión (writelog(expresión)): imprime un 1 si la expresión es verdadera y un 0 si es falsa.
* Imprimir cadena (writestr(cadena)): imprime una cadena encerrada entre dobles comillas.
* Imprimir retorno (writeintro()): imprime un retorno de carro.
* Imprimir tabla de verdad (writetabla(expresión)): imprime la tabla de verdad de una expresión. En este caso, los valores de las variables lógicas que están en la expresión no se toman en cuenta.

Variables lógicas:

* Las variables lógicas deben comenzar por una letra o el símbolo “\_” y después cualquier letra o número. No se admiten otros símbolos.
* Las variables lógicas pueden valor 0 o 1. No hace falta declararlas antes de utilizarlas.
* Si una variable lógica no tiene valor antes de ser utilizada, su valor es 0 (falso).

Operadores:

* AND (“\*”)
* OR (“+”)
* NOT (“-”)
* ENTONCES (“🡪”)
* DOBLEENTONCES ( “⬄”)
* TAUTLOGÍA (tauto(expresión)): devuelve un 0 si la expresión no es una tautología y un 1 si sí lo es. No toma en cuenta el valor de las variables lógicas.
* CONTRADICCION (contra(expresión)): devuelve un 0 si la expresión no es una contradicción y un 1 si sí lo es. No toma en cuenta el valor de las variables lógicas.
* DECIDIBLE (deci(expresión)): devuelve un 0 si la expresión no es decidible y un 1 si sí lo es. No toma en cuenta el valor de las variables lógicas.

El orden de los operadores es: -, \*, +, 🡪, ⬄

Expresiones:

* Las expresiones lógicas constan de variables lógicas y constantes lógicas (1 o 0) relacionadas por medio de operadores. Para aclarar mejor todo, se pueden utilizar paréntesis.

Ejemplos de programas válidos:

