PROCESADORES DEL LENGUAJE

---

# PRÁCTICA FINAL | AJS

## Analizador del lenguaje - *AlmostJavaScript*

---

Curso 2023-24

Campus de Colmenarejo

uc3m | Universidad Carlos III de Madrid

# 1/ Introducción

La práctica consistirá en la implementación de un compilador capaz de analizar programas de un lenguaje denominado AJS (*AlmostJavaScript)*, que contiene expresiones aritmeticológicas, objetos (basados en el formato AJSON trabajado en la práctica de introducción, sin la parte opcional de arrays), funciones y estructuras de control básicas.

Tras conocer los fundamentos del análisis léxico, sintáctico y semántico, así como la familiarización con las herramientas de construcción de compiladores, la práctica permitirá conocer la implementación completa de un analizador en las fases léxica, sintáctica y semántica.

# 2/ Objetivos

La práctica se dividirá en las siguientes partes:

I. Generación del analizador léxico (*lexer*) que reconozcan los elementos del lenguaje a este nivel.
II. Generación de la gramática que permitirá reconocer el lenguaje proporcionado y los símbolos terminales y no terminales de la misma mediante el analizador sintáctico (*parser)* creado a partir de la misma.
III. Añadir la capacidad para registrar y tratar variables declaradas por el programador, almacenando el tipo que se infiera dinámicamente:
    a. Se deberá crear una estructura tabla de símbolos.
        i. Esta servirá para registrar la información de las variables de tipo básico del programa.
    b. Se deberá crear una estructura tabla de registros.
        i. Esta servirá para almacenar las variables complejas de tipo objeto del programa.
IV. Comprobaciones semánticas de todas las expresiones del programa, incluyendo la inicialización de variables, las expresiones asignadas a variables, operadores aplicados y condiciones de los controles de flujo. Se harán comprobaciones de tipos y operaciones válidas para distinguir los diferentes tipos del lenguaje, incluyendo, cuando sea necesaria, la conversión de tipos compatibles.
V. Extender las tablas de variables y tipos para registrar las funciones definidas y hacer comprobaciones semánticas en las llamadas a función. Generar el código intermedio de estas.
VI. Y, por último, de manera opcional, se puede realizar uno de los dos siguientes apartados para subir la calificación de la práctica. Cualquier entrega parcial será tenida en cuenta:
    a. Generación de código intermedio (cuartetos con el juego de instrucciones vistas en la parte teórica de la asignatura). Se permitirán operaciones en código intermedio de asignación, operaciones aritméticas y lógicas, comparación, control de flujo con saltos y bucles condicionales (no funciones).
    Opcionalmente, las operaciones lógicas pueden hacerse en “cortocircuito”, no necesitando así hacer operaciones lógicas en código intermedio.
    b. El compilador se recuperará de errores a diferentes niveles (léxicos, sintácticos y semánticos) Al ignorar los errores, el compilador generará el código correspondiente como si dichos caracteres no existieran, siempre y cuando no se encuentren otro tipo de errores (consultar teoría)

## 2.1/ Especificación del lenguaje de entrada

El lenguaje AJS (*AlmostJavaScript)* es un pequeño lenguaje de programación imperativo, débilmente tipado y dinámico (a excepción de los objetos, donde si existe un fuerte tipado), basado en el conocido lenguaje *JavaScript*[1].

El lenguaje trabaja con varios tipos de datos básicos y permite definir tipos compuestos de tipo objeto. Las estructuras de control de flujo básicas que contiene son el salto condicional y el bucle condicional. Por último, permite la declaración y llamada de funciones.

```
type NestedObject = { "this is": character, prop2: int, "prop3": boolean };
type Object = { prop1: float, "nested": NestedObject };

let var1: Object, var2, var3;
var1 = {
    prop1: 10e-1,
    nested: { "this is": 'A', prop2: 10, prop3: tr }
};
var2 = 10;

// This is a function
function gte(a: int, b: int): boolean {
    return a > b;
}

/*
 * we can access object properties using dot notation (sometimes) or,
 * we can also use the bracket notation (always)
 */
if (gte(var1.nested["prop2"], var2)) {
    var3 = 'Y';
} else {
  var3 = 'N';
}
```

*Ilustración 1 - Ejemplo de fichero AJS*

### 2.1.1/ Comentarios

Se debe permitir la aparición de comentarios, dentro y fuera de las funciones, que deben ser tratados -e ignorados- por completo en el análisis léxico. Los tipos de comentarios son:

I. De una línea: Comienzan por doble barra (//) y terminan con un salto de línea.
II. Múltiple línea: Comienzan por barra-asterisco (/*) y terminan con asterisco-barra (*/)

### 2.1.2/ Palabras reservadas

Las palabras reservadas del lenguaje tienen una condición especial y por tanto no pue den usarse como nombres de variables. Solo pueden usarse en minúsculas y son:

| | | | | | | |
|---|---|---|---|---|---|---|
| *tr* | *fl* | *let* | *int* | *float* | *character* | *while* |
| *boolean* | *function* | *return* | *type* | *if* | *else* | *null* |

[1] https://developer.mozilla.org/es/docs/Web/JavaScript

### 2.1.3/ Variables, tipos y operaciones

Los tipos de datos que el lenguaje incluye son:

I. Números enteros
   a. Base decimal: 0, 10, 420, -12, -999, etc.
   b. Base binaria: 0b101, 0B110110, etc. (comienzan con "0b" o "0B")
   c. Base octal: 0712, 0332, 01121, etc. (comienzan con "0")
   d. Base hexadecimal: 0XED13, 0xAA, 0Xfb, 0x00F1, etc. (comienzan con "0x" o "0X")
II. Números reales
   a. Notación con punto decimal: 0.1289, .12, -100.001, 50., etc.
   b. Notación científica: 10e-1, .1E10, 5E2, 4e-2, 9,87e1, etc. (se utiliza "e" o "E")
III. Caracter: Cualquier carácter de la codificación *ASCII-extendido*[2]. Delimitado por comillas simples (') las cuales no se incluyen en el valor.
IV. Valores booleanos: Se utilizarán las palabras reservadas "tr" (*True* en Python) y "fl" (*False* en Python).
V. Valores nulos: Se utilizará la palabra reservada "null" (*None* en Python).
VI. Objetos: Formato AJSON (consultar enunciado P1) pero teniendo en cuenta que:
   a. Los valores del tipo cadena de caracteres han sido sustituidos por valores tipo carácter. Esto solo afecta a los valores, no a las claves.
   b. No se incluye la parte opcional de arrays.

Todas las variables, menos los objetos, no deberán ser tipados en su declaración, por lo que el tipo se inferirá dinámicamente. Los objetos, sin embargo, si estarán tipados (consultar punto 2.1.3, sección de objetos). A continuación, se detallan los tipos que maneja AJS de manera interna (y los cuales habrá que utilizar en los tipos de los objetos):

| Variable | Tipo |
|---|---|
| Número entero | *int* |
| Número real | *float* |
| Carácter | *character* |
| Booleano | *boolean* |

*Tabla 1 - Variables y tipos*

Los tipos de operaciones que se podrán ejecutar son:

I. Aritméticas: Operaciones binarias (+, -, *, /) y unarias (+ y -).
II. Booleanas: Conjunción (&&), disyunción (||) y negación (!)
III. Comparación: Igual (==), mayor (>), mayor o igual (>=), menor (<) y menor o igual (<=)

### 2.1.4/ Sentencias

I. Evaluación de expresiones: Evaluación de cualquier tipo o combinación válida de operaciones aritméticas, booleanas o de comparación.
II. Declaración/Asignación: Declaración/Asignación de variables de los diferentes tipos.

[2] *American Standard Code for Information Interchange* – 256 caracteres

Las sentencias se separan con el token punto y coma “;”.

Las declaraciones comienzan por la palabra reservada “let” seguido de el/los nombre/s de la/s variable/s. Las asignaciones comienzan por un nombre de variable, seguido del signo igual (=) y terminan con una expresión.

Las expresiones pueden ser literales u operaciones matemáticas. No se permite la asignación de valores en mitad de una expresión, es decir, serán incorrectas las sentencias de este tipo:

```
A + 4 * B = 3 - 7 // Error: asignación (B=3) en una expresión
```

*Ilustración 2 - Error por asignación dentro de expresión*

Las variables pueden asignarse en el momento de la declaración. En caso de realizar la declaración y asignación por separado, para poder asignar una variable debe haberse declarado con anterioridad. El lenguaje permite declarar una lista de variables separadas por comas. En caso de realizar la asignación a la vez que la declaración, las variables se inicializarán todas con el mismo valor.

```
let a, b;
a = 10;
b = a * a <= 200;
c = '$';    // Error: la variable 'c' no ha sido declarada anteriormente

let d, e = 0xFF / (0b101 * (0702 - 1e-10));
```

*Ilustración 3 - Declaración/Asignación múltiple*

No se permiten re-declaraciones: cuando se declara una variable, no se podrá volver a declarar ninguna otra variable con el mismo nombre[3]. Lo que si estará permitido será la reasignación de valores a las variables, sean o no del mismo tipo.

```
let var1, VAR1, Var1;
let var1;   // Error: la re-declaración de variables no está permitida

VAR1 = 10;  // Ok: VAR1=(int, 10)
VAR1 = tr;  // Ok: la reasignación sí -> VAR1=(boolean, True)
```

*Ilustración 4 - Re-declaración y reasignación*

Cuando una variable es declarada sin asignársele un valor inicial, se le asignará el valor y tipo por defecto: null. Hay que tener en cuenta que con variables nulas no se puede operar.

#### *2.1.4.1/ Objetos: variables de tipo registro*

Finalmente, pueden formarse compuestos de estilo registro, los objetos. Para poder declarar y asignar un objeto, debe haberse definido el tipo con anterioridad. Tanto la declaración del tipo como su posterior asignación (al menos la primera, la inicialización) siguen el formato AJSON (con limitaciones – consultar punto 2.1.2). Es decir, el orden a seguir para crear un objeto es:

[3] Los nombres de las variables son sensibles a mayúsculas. Esto quiere decir que será legal declarar variables con las mismas letras, pero diferente capitalización (*miVariable*, *MIVARIABLE*, *MiVariable*, etc.)

I. Tipo: Declarar el tipo utilizando la palabra reservada “type”, seguida del nombre del tipo, el signo igual (=) y terminando con la definición del tipo en formato AJSON.
    a. Las claves serán los nombres de las propiedades que implemente el tipo.
    b. Los valores serán los tipos de las propiedades que implemente el tipo. Consultar la tabla de tipos para saber que palabras reservadas se utilizan en cada caso.
II. Declaración: Independientemente de si se declara y se asigna a la vez, o primero se declara y luego se asigna, los objetos siempre tendrán un tipo asignado: después de “let” y el nombre de la variable, se indicará con dos puntos “:” seguido del nombre del tipo.
III. Inicialización (primera asignación): Mientras un objeto no se ha asignado (si solo se ha declarado) su tipo será el indicado en la asignación, pero su valor será null. Para realizar la inicialización del mismo, se seguirá el formato AJSON de nuevo:
    a. Las claves serán las mismas que las del tipo que implementa.
    b. Los valores serán cualesquiera que cumplan los tipos definidos en los valores del tipo que implementa. También podrán ser expresiones, variables, etc.
    c. Si falta alguna propiedad, se debe considerar como un error semántico de tipo.
IV. Reasignación del objeto: Se podrá reasignar el objeto, siempre que se cumpla el tipo implementado (todas las variables, con los tipos correctos)
V. Reasignación de propiedades: Se podrá reasignar de manera individual el valor de una propiedad, siempre que se cumpla el tipo. Para reasignar una propiedad, se deberá acceder a ella (consultar siguiente sección).

Para acceder a sus propiedades (ya sea para asignar un valor o para consultarlo), se permitirá:

I. Notación con punto: Se accede a la propiedad de un objeto especificando el objeto seguido de punto (.) y el nombre de la propiedad.
    a. Esta notación no puede ser usada para propiedades con caracteres especiales o espacios (solo se permiten propiedades que cumplan los requisitos de la cadena de caracteres sin comillas especificada en la P1)
II. Notación con corchetes: Se accede a la propiedad de un objeto especificando el objeto seguido de corchetes ([..]) y el nombre de la propiedad delimitado por comillas dobles (“).
    a. Se puede usar para cualquier tipo de propiedad.

```
type Circle = { cx: float, cy: float, radio: float, color: character };
type Square = { side: float, color: character };

let circle: Circle = {
    cx: 0.,
    cy: 0.,
    radio: 10.2,
    color: 'r',
};

let square: Square, circle_area;
circle_area = 3.14 * circle.radio * circle["radio"];

square.side = 10.;  // Error: 'square' no ha sido inicializado
square = {
    side: 10.,
    color: 'b'
};

let square_area = square.side * square["side"];
```

*Ilustración 5 - Objetos: declaración, asignación y acceso a propiedades*

Los dos tipos de notaciones, cuando se trate de un objeto que a su vez contenga objetos anidados como valor de sus propiedades, pueden intercalarse libremente, respetando las limitaciones expuestas previamente.

```
type House = {
    location: { city: character, country: character },
    price: { euro: int, dollar: int, yen: int }
};

let my_house: House = {
    location: {
        city: 'M', country: 'E',
    },
    price: {
        euro: 100, "dollar": 108, "yen": 16397
    },
};

let with_tax = my_house["price"].euro * 1.1;
let euro_to_yen = my_house.price["euro"] * 163.97;
```

*Ilustración 6 - Acceso a propiedades con mezcla de notaciones en objetos anidados*

### 2.1.5/ Reglas de tipos

I. Movimiento de datos: Se utiliza un dato previamente definido, hay que comprobar si está permitido usarlo en ese punto.
II. Combinación de datos: Varios datos son combinados para producir un resultado.

Ambos niveles están relacionados: a veces es necesario resolver una operación de combinación de datos para saber si el tipo de dato que se va a producir es válido para una aplicación posterior.

```
let a = 10;
let b = 12.12;

if (a) { ... } // movimiento: ¿se puede utilizar 'a' de tipo 'int' como condición?
let x = a + b; // combinación: ¿cuál es el tipo de 'x'?
```

*Ilustración 7 - Categorías en las reglas de tipado*

Para ambas categorías, debemos asegurarnos que el tipo de origen y el de destino son iguales, o bien que se puede realizar una conversión del tipo de origen al tipo de destino sin pérdida de datos: algunos tipos pueden transformarse automáticamente, según la siguiente tabla de conversión:

| Origen | Destino | Transformación |
|---|---|---|
| *character* | *int* | Se toma el valor numérico del carácter (0-255)[4] |
| *int* | *float* | Pasar de 32 bits que representan un entero en $CA_2$ a 32 bits que representan un real en IEEE754 |

*Tabla 2 - Conversión legal automática de tipos*

[4] https://docs.python.org/es/3/library/functions.html#ord

```
type TypeConversions = {
    f1: float,
    f2: float,
    i:  int,
    b:  boolean,
};

let t: TypeConversions = {
    f1: 7.5,    // Ok: float -> float
    f2: 0b11,   // Ok: int -> float
    i: 'a',     // Ok: character -> int
    b: tr
};

t.i = 7.5;      // Error: float -> int
t.b = 7;        // Error: int -> boolean
t.b = t["f1"]   // Error: float -> boolean
```

*Ilustración 8 - Asignaciones con diferente tipo válidas y no válidas*

Por otro lado, todas las operaciones que combinan/manejan dos o más valores deben asegurar que sus tipos son similares, o al menos que se puede realizar la conversión de uno de ellos al tipo del otro sin pérdida de datos.

Respecto a las operaciones y expresiones, cada operador requiere uno o varios tipos de dato concretos a la entrada, y devuelve así mismo un tipo específico de dato a la salida. En la siguiente tabla se especifican los tipos de datos compatibles con cada operador (sin tener en cuenta conversiones), y la salida correspondiente para cada tipo de entrada:

| Operador | Tipo origen | Tipo resultado |
|---|---|---|
| MÁS (+) | *int* | *int* |
| | *float* | *float* |
| | *character* | *character* |
| MENOS (-) | *int* | *int* |
| | *float* | *float* |
| | *character* | *character* |
| POR (*), ENTRE (/) | *int* | *int* |
| | *float* | *float* |
| MAYOR (>), MENOR (<), MAYOR O IGUAL (>=), MENOR O IGUAL (<=) | *int* | *boolean* |
| | *float* | *boolean* |
| | *character* | *boolean* |
| IGUAL (==) | *int* | *boolean* |
| | *float* | *boolean* |
| | *character* | *boolean* |
| | *boolean* | *boolean* |
| AND (&&), OR (\|\|), NOT (!) | *boolean* | *boolean* |

*Tabla 3 - Operadores y tipos de datos permitidos*

No obstante, cuando una de las entradas de un operador binario tiene un tipo A y la otra entrada tiene un tipo B, será necesario convertir una de ellas al tipo de la otra: el dato con el tipo más restrictivo se transforma al tipo más general:

I. *character* puede convertirse a *int* o *float*
II. *int* puede convertirse a *float*

No se permite la conversión del tipo *boolean* a ningún otro tipo. A continuación, se muestran varios ejemplos donde se combinan todas las reglas referentes al tipado:

```
5.1 * 41.;          // Ok: real * real -> real
let c = 'c' + 'c';  // Ok: character + character -> character
let i = 1 + c;      // Ok: int + character -> int

/*
 * Ok: 1) 'c' * 8 -> char * int -> int
 *     2) 7.5 + 1) -> float + int -> float
 */
let f1 = 7.5 + 'c' * 8;

/*
 * Error: 1) 'd' < 8 -> char (automatic conversion to int) < int -> boolean
 *        2) 5.0 * 1) -> float * boolean -> TYPE ERROR
 */
let err1 = 5.0 * ('d' < 8);

/*
 * Error: "if" requiere boolean
 *        no se puede convertir directamente de real a boolean
 */
if (5/7) { err1 = 'N'; }
```

*Ilustración 9 - Ejemplos de reglas referentes al tipado*

## 2.1.6/ Control de flujo

### *2.1.6.1/ Salto condicional*

En cuanto al control de flujo, se definen dos estructuras de control: salto condicional y bucle condicional. El salto condicional es una sentencia que evalúa una expresión booleana – llamada condición –, y dependiendo de si resultado es verdadero o falso, salta a un punto más avanzado del programa o sigue ejecutando las sentencias adyacentes.

Unas de las posibles sentencias en BNF que describen la construcción del salto condicional podrían ser las siguientes:

```
conditional ::= IF '(' expression ')' '{' sentence_block '}'
              | IF '(' expression ')' '{' sentence_block '}'
                ELSE '{' sentence_block '}'
```

*Ilustración 10 - Sentencias BNF que describen la construcción del salto condicional*

La primera variante descrita en la ilustranción es un salto simple: si la expresión de la condición evalúa verdadero, se ejecutará el bloque de sentencias encerrado entre las llaves y después, se continuará el programa con normalidad.

La segunda opción es algo distinta: si la condición evalúa verdadero, se ejecutará el primer bloque de sentencias y después se saltará a la sentencia siguiente al salto condicional. Si la condición evalúa a falso, se ejecutará el segundo bloque de sentencias y luego se continuará con normalidad.

```
if (f2 == 3 || b1 && b2 || 10 - 5 * i1 >= 0xFF - 1e-1) {
    f2 = f2 - 3;
} else {
    f2 = 10 - f1 * f1;
}
```

*Ilustración 11 - Ejemplo de código con control de flujo de salto condicional*

#### *2.1.6.2/ Bucle condicional*

El bucle condicional es una sentencia que evalúa una expresión booleana – llamada condición –, y dependiendo de si resultado es verdadero o falso, ejecuta el cuerpo del bucle o bien salta al final de éste. La estructura BNF correspondiente al bucle condicional podría ser la siguiente:

```
loop ::= WHILE '(' expression ')' '{' sentence_block '}'
```

*Ilustración 12 - Sentencias BNF que describen la construcción del bucle condicional*

```
let v1 = 0;
let v2 = 0;

while (v1 < 10) {
    v2 = v2 + 1;
    v2 = v2 * 2;
    v1 = v1 + 1;
}
```

*Ilustración 13 - Ejemplo de código con control de flujo de bucle condicional*

### 2.1.7/ Funciones

Una función es un bloque de sentencias que, dados unos parámetros de entrada, realizan una serie de cálculos para obtener un único valor al que se denomina retorno. Las funciones tienen un nombre que permite que sean referenciadas dese cualquier punto de un programa al igual que ocurre con las variables. Por tanto, para declarar una función es necesario especificar su nombre y el de sus valores de entrada o argumentos, el tipo de retorno, las sentencias que forman parte del cuerpo de la función y por último la sentencia de retorno.

En AJS los argumentos no son obligatorios: una función puede recibir cero valores de entrada. Sin embargo, el cuerpo de una función debe contener, por lo menos, la sentencia de retorno:

```
function_definition ::= FUNCTION ID '(' args_list ')' ':' type
                        '{' sentence_block RETURN sentence '}'
```

*Ilustración 14 - Sentencias BNF que describen la definición de la función*

La lista de argumentos serán una serie de identificadores separados por comas, indicando el tipo de los mismos. Como se comentaba previamente, es válida la lista de argumentos vacía.

Una función es invocada cuando se escribe su nombre, seguida de una lista de expresiones entre paréntesis que coincide en número y tipo con sus argumentos. Esto implica que puede existir la sobrecarga de funciones (mismo nombre, distinto número de argumentos o tipo de los mismos).

```
function_call ::= ID '(' expression_list ')'
```

*Ilustración 15 - Sentencias BNF que describen la llamada de una función*

El resultado será un valor con el tipo de dato de retorno de la función. Es importante notar que los argumentos de la función pueden ser una expresión completa. A continuación, un ejemplo de código:

```
function mod(a: int b: int): int {
    while (a => b) {
        a = a - b;
    }
    return a;
}

function greatest_common_divisor(a: int, b: int): int {
    let temp;
    while (!(b == 0)) {
        temp = b;
        b = mod(a, b);
        a = temp;
    }
    return a;
}

let result = greatest_common_divisor(132, 0xFF);
```

*Ilustración 16 - Ejemplo de código con definición y llamada de funciones*

También podrá usarse los tipos complejos definidos por el desarrollador (tipos de objeto) como tipo de los argumentos de entrada o del retorno de la función:

```
type Point = { x: int, y: int };

function get_point_after_jump(point: Point): Point {
    return { x: point.x, y: point.y + 10 };
}
```

*Ilustración 17- Ejemplo de código con función usando tipos complejos*

## 2.2/ Modificaciones avanzadas

### 2.2.1/ Generación de código intermedio

Como se indicaba anteriormente, la generación de código intermedio se realizará mediante cuartetos con el juego de instrucciones vistas en la parte de teoría de la asignatura. A continuación, se indican varios ejemplos, con pequeños programas que contienen expresiones aritméticas, lógicas y de control de flujo. Consultar teoría (*Tema 7 – Generación de código intermedio*)

```
// Operaciones aritméticas y comparación
let f1, f2;
let b1;
f1 = 5 * 4 - 80 / 10
f2 = 10 / 5 * f1;
b1 = 5 < 3;

/*
 * (*, 5, 4, t1)
 * (/, 80, 10, t2)
 * (-, t1, t2, t3)
 * (=, t3, , f1)
 * (/, 10, 5, t4)
 * (*, t4, f1, t5)
 * (=, t5, , f2)
 * (<=, 5, 3, t6)
 * (=, t6, , b1)
 */
```

*Ilustración 18 - Código intermedio: operaciones aritméticas y de comparación*

```
// Salto condicional
let num;
let is_positive;
if (num >= 0) {
    is_positive = tr;
} else {
    is_positive = fl;
}

/*
 * (>=, num, 0, t1)
 * (gotoc, t1, L1, )
 * (goto, L2, , )
 * (label, L1, , )
 * (=, true, , is_positive)
 * (goto, L0, , )
 * (label, L2, , )
 * (=, false, , is_positive)
 * (label, L0, , )
 */
```

*Ilustración 19 - Código intermedio: saltos condicionales*

El código intermedio se generará en un fichero con el nombre del fichero de entrada y de extensión ".out". Cada línea del fichero contendrá un cuarteto, similar a como se muestra en los comentarios de las ilustraciones anteriores.

### 2.2.2/ Recuperación de errores

La recuperación de errores trata de informar con claridad y exactitud (tipo de error, línea y columna en que se producen) de los errores que ocurren en los diferentes niveles de análisis. Debe ofrecer una recuperación rápida que permita continuar con el análisis, sin retrasar el procesamiento del programa sin errores.

Algunos ejemplos de errores serían:

I. Léxico: escribir mal un identificador, caracteres no reconocidos, etc.
II. Sintáctico: No poner un ";" al final de una sentencia.
III. Semántico: Multiplicar un variable booleana por una de tipo entero.

El objetivo de esta modificación avanzada consiste en detectar todos los errores, lo antes posible, evitando detectar errores repetidos o generar falsos positivos. Se deberá consultar la teoría (*Tema 4 – Recuperación de errores*) para más detalle. Será importante acompañar una detallada descripción de las estrategias implementadas en la memoria.

# 3/ Entrega parcial

En una primera entrega deberá completarse la construcción del analizador léxico del lenguaje y la especificación de la gramática.

Cada pareja debe entregar todo el contenido de su práctica en un único archivo comprimido, preferiblemente en formato *.zip*

El nombre del comprimido debe seguir el siguiente formato: "**PL_P2_{AP1}_{AP2}.zip**", donde {**AP1**} y {**AP2**} son los primeros apellidos de cada integrante del grupo en orden alfabético.

Al abrir el archivo comprimido, debe verse un único directorio con el mismo nombre que el del archivo. Este directorio debe contener el proyecto completo y funcional, con los archivos de especificación léxica y sintáctica del *scanner* y el *parser*, respectivamente, que satisfagan los requisitos del enunciado.

Se harán pruebas separadas del analizador léxico y sintáctico, por lo que se requiere que el fichero "main.py" permita ser ejecutado de la siguiente manera:

```
python ./PL_P2_{AP1}_{AP2}/main.py {input_file} -{lex|par}
```

I. La salida cuando se especifique la opción "-lex" generará un fichero con el nombre del fichero de entrada y de extensión ".token" línea por cada *token* reconocido que incluya el tipo y el valor del token separados por un espacio en blanco.
II. La salida cuando se especifique la opción "-par" deberá ser vacía, indicando que se ha reconocido la entrada (sin mostrar ningún error).

Se adjuntará una breve (pero completa) memoria[5], que profundice en el razonamiento y proceso de toma de decisiones de los autores. El formato deberá ser únicamente PDF, y el nombre coincidirá con el del fichero, pero con el sufijo *"_memoria.pdf"*.

[5] Es importante que la especificación de la gramática esté incluida en la memoria entregada.

# 4/ Entrega final

En la entrega final, la práctica deberá cumplir todos los objetivos indicados: análisis léxico, sintáctico y semántico, así como la tabla de símbolos que almacene variables y la tabla registros para los tipos complejos. La gramática debe ser diseñada de tal manera que el *parser* no tenga conflictos ni ambigüedades de ningún tipo.

## 4.1/ Especificación de la salida

Se harán pruebas separadas del *lexer* y el *parser*, por lo que se requiere que el fichero "main.py" permita ser ejecutado de la siguiente manera:

```
python ./PL_P3_{AP1}_{AP2}/main.py {input_file} -{lex|par}
```

I. La salida cuando se especifique la opción "-lex" generará un fichero con el nombre del fichero de entrada y de extensión ".token" línea por cada *token* reconocido que incluya el tipo y el valor del token separados por un espacio en blanco.
II. La salida cuando se especifique la opción "-par" generará un fichero con el nombre del fichero de entrada y de extensión ".symbol" con el estado de la tabla de símbolos y un fichero con el nombre del fichero de entrada y de extensión ".register" con el estado de la tabla de registros.
    a. En caso de no existir estructuras de control de flujo, ni llamadas a funciones, el programa además mostrará los valores de las variables tras la ejecución del programa.

## 4.2/ Entrega

Cada pareja debe entregar todo el contenido de su práctica en un único archivo comprimido, preferiblemente en formato *.zip*

El nombre del comprimido debe seguir el siguiente formato: "**PL_P3_{AP1}_{AP2}.zip**", donde {**AP1**} y {**AP2**} son los primeros apellidos de cada integrante del grupo en orden alfabético.

Al abrir el archivo comprimido, debe verse un único directorio con el mismo nombre que el del archivo. Este directorio debe contener el proyecto completo y funcional, con los archivos de especificación léxica, sintáctica y semántica del *scanner* y el *parser*, respectivamente, que satisfagan los requisitos del enunciado. Así mismo, deberán existir archivos relacionados con la tabla de símbolos y de registros.

Es importante que en esta entrega se adjunten ficheros con pruebas que demuestren la correcta implementación de cada uno de los objetivos: es importante documentar bien los tests realizados (cómo y por qué) en la memoria, así como cubrir los casos tanto de éxito como de fallo, y que en ambos el resultado sea el esperado.

Se adjuntará una breve (pero completa) memoria, que profundice en el razonamiento y proceso de toma de decisiones de los autores. El formato deberá ser únicamente PDF, y el nombre coincidirá con el del fichero, pero con el sufijo *"_memoria.pdf"*. La memoria contendrá, al menos, los siguientes apartados:

I. Portada: asignatura, práctica, composición y nombre del grupo, año lectivo.

II. Tabla de contenidos: secciones y números de página correspondientes.
III. Introducción
IV. Gramática: versión final de la gramática implementada, en notación BNF.
V. Breve descripción de la solución: explicar la solución implementada y los controles semánticos realizados.
VI. Archivos de prueba: explicación de los aspectos probados en los archivos de prueba entregados.
VII. Conclusiones: aspectos adicionales a tener en cuenta por el corrector, y hechos a destacar sobre desarrollo de la práctica (incluyendo, pero no limitándose, a opiniones personales sobre el material de partida, conocimientos y cantidad de trabajo necesaria para su consecución).

## 4.3/ Consideraciones

Los criterios que primarán en la corrección son:

I. Funcionalidad: Todo debe funcionar sin errores y devolver la salida esperada. Se recomienda probar el código con una batería de pruebas, la cual se recomienda entregar en caso de realizarse.
II. Buen uso de las herramientas *PLY*: Demostrar que se conocen y dominan las herramientas trabajadas en clase. Se recomienda revisar los manuales para completar o confirmar el conocimiento impartido en los laboratorios.
III. Exposición justificada: La memoria es un elemento clave de la entrega y deberá reflejar y ayudar al corrector a revisar el trabajo realizado, por lo que se recomienda dedicarle una atención similar que a la propia funcionalidad.

Es importante recordar que:

I. Se debe respetar de forma estricta el formato de entrega. De no hacerlo, se corre el riesgo de perder puntos en la calificación final.
II. Cada grupo responderá de forma totalmente responsable del contenido de su práctica. Esto implica que los autores deben conocer en profundidad todo el material que entreguen.
III. Ante dudas de plagio y/o ayuda externa, el corrector podrá convocar al grupo para responder a cuestiones sobre cualquier aspecto de la memoria o código entregados.