PROCESADORES DEL LENGUAJE

PRÁCTICA FINAL | VIPER

Analizador del lenguaje

Curso 2024-25
Campus de Colmenarejo

TABLA DE CONTENIDOS

1· INTRODUCCIÓN

La práctica final consistirá en la implementación de un compilador capaz de analizar programas de un lenguaje denominado Viper, que contiene expresiones aritmeticológicas, registros, vectores simples, funciones y estructuras de control básicas. También será capaz de comprobar la corrección en términos de los tipos de datos utilizados.

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, y emplear estos conocimientos para reconocer errores en el uso de tipos e informar al programador.

2· OBJETIVOS

La práctica se divide en las siguientes partes o bloques de contenido a realizar:

I. Generación del analizador léxico, que reconozca 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 creado a partir de la misma.
III. Añadir la capacidad para registrar y tratar variables declaradas por el programador, almacenado el tipo con el que se declaren.
    a. Se deberá crear una estructura de tipo tabla para almacenar los símbolos.
        i. Esta servirá para registrar la información de las variables de tipo básico del programa y, posteriormente, las de tipo vector.
    b. Se deberá crear una estructura de tipo tabla para almacenar los registros.
        i. Esta servirá para almacenar los tipos complejos declarados en el programa, de tipo registro.
IV. Comprobaciones semánticas de todas las expresiones del programa, incluyendo la inicialización de las variables, las expresiones asignadas a variables, operadores aplicados y condiciones de los controles de flujo.
    a. 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. Y, por último, se tendrá que elegir uno de los dos siguientes apartados para poder obtener la calificación máxima:
    a. Extender las tablas de variables y tipos para registrar las funciones definidas y hacer comprobaciones semánticas en las llamadas a función.
    b. El compilador se recuperará de errores a diferentes niveles (léxico, sintáctico y semántico). Consultar la teoría para más detalles.

3· ESPECIFICACIÓN DEL LENGUAJE DE ENTRADA

El lenguaje Viper es un pequeño lenguaje de programación imperativo fuertemente tipado. El lenguaje trabaja con varios tipos de datos básicos y permite definir tipos compuesto de tipo registro y vectores. Las estructuras de control de flujo básicas que contiene son el salto condicional y el bucle condicional. Permite la declaración y llamada de funciones.

```
type Point:
  float x, y

type Circle:
  Point center
  float radius

Circle[2] balls

# 1st ball
balls[0].center.x = 0b101
balls[0].center.y = 0xFF
balls[0].radius = 1.74
# 2nd ball
balls[1].center.x = 10e-1
balls[1].center.y = -31
balls[1].radius = 3.2

int i = 0
while i < balls.len:
  balls[i].center.x = balls[i].center.x + 1
  balls[i].center.y = balls[i].center.y + 2
  i = i + 1

def bool gte(int a, b):
  return a > b

'''
Result should be
equal to 'N', i.e. no
'''
char result
if gte(balls[0].radius, balls[1].radius):
  result = 'Y'
else:
  result = 'N'
```

Ilustración 1 - Ejemplo de fichero Viper

3.1- 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 una almohadilla `#` y terminan con un salto de línea.
II. Múltiple línea: Comienzan por tres comillas simples `'''` y terminan con tres comillas simples `'''`

3.2- Palabras reservadas

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

`true`	`false`	`int`	`float`	`char`
`def`	`return`	`type`	`if`	`else`
`and`	`or`	`not`	`bool`	`while`

3.3- Variables, tipos y operaciones

I. Números enteros
   a. Base decimal - `0` `10` `420`
   b. Base binaria - `0b101` `0b110110` `0b0001` (comienzan con `0b`)
   c. Base octal – `0o712` `0o332` `0o1121` (comienzan con `0o`)
   d. Base hexadecimal - `0xED13` `0xAA` `0xFB10` (comienzan con `0x`)
      i. Las letras que componen el número deben ser mayúsculas.

II. Números reales
   a. Notación con punto decimal – `0.1289` `100.001` `1.3140`
      i. Es necesario que el número tenga parte entera y parte decimal.
      ii. Se utiliza `.` como carácter de separación.
   b. Notación científica – `10e-1` `9.87e-2` `5e5`
      i. Se utiliza `e` como carácter de separación.

III. Carácter
   a. Cualquier carácter de la codificación ASCII-extendido.
   b. Delimitado por comillas simples, i.e. `'a'`

IV. Booleanos: Se utilizarán las palabras reservadas `true` y `false`.

V. Vectores: Consultar la sección de vectores.

VI. Registros: Consultar la sección de variables de tipo registro.

Todas las variables deberán ser tipadas en su declaración. A continuación, se detallan los tipos básicos que se manejan en Viper.

Variable	Tipo
Número entero	`int`
Número real	`float`
Carácter	`char`
Booleano	`bool`

Tabla 1 - Tipos básicos de variables

Los tipos de operaciones que se podrán ejecutar son:

I. Aritméticas: Operaciones binarias en notación infija `+` `-` `*` `/` y unarias `+` `-`
II. Booleanas: Conjunción `and` , disyunción `or` y negación `not`
III. Comparación: Igual `==` , mayor `>` , mayor o igual `>=` , menor `<` y menor o igual `<=`

3.4- Sentencias

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

Las sentencias se separan con uno o muchos saltos de línea (al menos uno, excepto en el caso de la última línea donde el EOF se considera como separador válido final).

Las declaraciones comienzan por el tipo de la variable 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.

- El lenguaje permite declarar una lista de variables separadas por comas. En caso de realizar la asignación a la vez que la declaración, habrá que comprar que el tipo de la expresión que está siendo asignada es igual o equivalente al tipo de la declaración.
- Se permite la asignación en cadena, donde la variable más a la izquierda es asignada primero y se sigue el orden hasta llegar a la de más a la derecha.

```
''' Chained assignments '''
int a, b, c, d = 0
a = c = d = 2 * 3
# a, c y d son iguales a 6
# b es igual a 0
```

Ilustración 2 - Asignación en cadena

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 expr.
```

Ilustración 3 - Error por asignación dentro de una expresión

Las variables pueden asignarse en el momento de la declaración. En caso de realizar la declaración y asignación por separado, la variable debe declararse antes de asignarse:

```
int a, b
a = 10
b = a * b <= 200
c = '$' # Error: la variable c no ha sido declarada aún

float d, e = 0xFF / (0b101 * (0o702 - 1e-10))
```

Ilustración 4 - Declaración/Asignación

No se permiten las re-declaraciones: cuando se declara una variable, no se podrá volver a declarar ninguna otra variable con el mismo nombre – 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.)

```
bool var1, VAR1, Var1
float var1 # Error: la re-declaración no está permitida

VAR1 = false # Ok: VAR1=(bool, false)
VAR1 = true # Ok: la reasignación sí está permitida
```

Ilustración 5 - Re-declaración y reasignación

3.5- Estructura del programa

Hay que tener en cuenta una serie de casos límite que deberán ser permitidos:

I. El fichero vacío será válido.
II. Un fichero con solo delimitadores (saltos de línea) será válido.
III. Se permite que el fichero comience por salto(s) de línea antes de que aparezca una sentencia.
IV. Se permite que el fichero finalice tanto con ninguno (el delimitador es EOF) como con uno o más saltos de línea.
V. Como se ha comentado previamente, las sentencias sencillas estarán separadas por uno o más saltos de línea.
VI. Las sentencias complejas no requerirán un salto de línea como separación ya que estarán delimitadas de otra manera, consultar la sección 3.7- Bloques de sentencias para más información.

3.6- Variables de tipo vector

El lenguaje permite declarar variables de tipo vector, de una única dimensión.

```
int[3] vector_3d
vector_3d[0] = 5
vector_3d[1] = 2
vector_3d[2] = 1

int array_length = vector_3d.len # Es igual a 3
vector_3d[1] = vector_3d[2] - 4

type Point:
  float x, y

Point[2] line
line[1].x = 1
line[1].y = 2
'''(0,0) ----> (1,2)'''
```

Ilustración 6 - Variables de tipo vector

I. No se puede hacer declaración con asignación: primero se declara la nueva variable y después se asigna el valor de cada una de las posiciones
II. Para su declaración, se seguirá la siguiente sintaxis: `TIPO [ EXPRESION ] id`

a. `TIPO` ha de ser cualquiera de los tipos de datos básicos o un tipo complejo definido por el usuario.
b. `EXPRESION` es la longitud fija del vector. Tiene que ser de tipo `int`
c. `id` es el nombre de la variable declarada

III. Para acceder a una posición concreta del vector: `id [ EXPRESION ]`
a. `id` es la variable de tipo vector que se está accediendo
b. `EXPRESION` es el índice que se está accediendo. Tiene que ser de tipo `int` y el valor debería estar dentro de los límites del vector (`0` y `<tamaño> - 1`), pero no puede ser comprobado en tiempo de compilación.

IV. Se puede acceder al tamaño de un vector con su propiedad `len`

3.7- Bloques de sentencias

Existen una serie de sentencias complejas (declaración de tipos, salto condicional, bucle condicional o declaración de funciones) que requieren la utilización de bloques de sentencias como parte de su sintaxis (importante revisar cada una de las secciones para asegurarse de las sentencias válidas en el bloque en cada uno de los casos). Estos bloques de sentencias han de ser delimitados de alguna manera. Existen dos opciones que se permiten en la especificación de nuestro lenguaje, una más sencilla y otra más compleja (a nivel léxico)

La forma más sencilla será utilizando los caracteres de llave de apertura `{` y llave de cierre `}`

La forma avanzada requerirá que el analizador léxico sea capaz de interpretar las tabulaciones o cada 2 espacios en blanco como tokens de indentación.

- Será necesario realizar un seguimiento del nivel de indentación durante el análisis para poder generar tokens de desindentación cuando se reduzca el número de tabulaciones/espacios en blanco en el comienzo de una nueva línea.
- Es posible que en algunos casos haya que emitir más de un token de desindentación, cuando se reduzca en más de uno el nivel de indentación en una nueva línea. Esto requerirá modificar el comportamiento base de PLY lex.
- Si una nueva línea comienza sin tabulaciones o espacios en blanco, se considerará que el nivel de indentación ha vuelto a 0 y emitir los tokens de desindentación necesarios.

Para obtener la nota completa en este apartado, solo se pedirá la sintaxis más sencilla que utiliza las llaves de apertura y de cierre. Sin embargo, implementar la forma avanzada, aunque pueda fallar en algunos casos límite, puede contribuir a obtener un incremento sustancial de la nota.

```
if true: {
  # valid sentence
        # valid sentence with
  # different indentation
}

while false:
  # all sentences in the block
  # with the same indentation
```

Ilustración 7 - Bloques de sentencias e indentación

3.8- Variables de tipo registro

El lenguaje permite declarar variables de tipo registro, así como los tipos que definen su forma.

```
type Circle:
  float cx, cy, radius
  char color

type Square:
  float side
  char color

Circle c
c.radius = 10.2
c.color = 'R'
int circle_area = 3.14 * c.radius * c.radius

Square s
s.side = 10.0
s.color = 'B'
int square_area = s.side * s.side

bool square_is_bigger = square_area > circle_area
```

Ilustración 8 - Variables de tipo registro

I. Para poder declarar una variable de tipo registro, debe haberse definido su tipo con anterioridad.
II. Para declarar un tipo: `type id : \n DECLARACIONES`
    a. `id` es el nombre del tipo que se está declarando.
    b. `DECLARACIONES` son una o muchas declaraciones de variables separadas por saltos de línea. Se permite la multi declaración.
III. No se puede hacer declaración con asignación: primero se declara la nueva variable y después se asigna el valor de cada una de las propiedades.
IV. Para declarar un registro: `id id`
    a. El primer `id` corresponde al nombre del tipo declarado previamente.
    b. El segundo `id` corresponde al nombre de la variable que se está declarando.
V. Para acceder a una propiedad concreta del registro: `id . id`
    a. El primer `id` corresponde al nombre del registro.
    b. El segundo `id` corresponde al nombre de la propiedad.
VI. Una propiedad de un registro podrá ser a su vez un tipo complejo.
    a. Esto implica que se podrá encadenar el acceso a propiedades.
    b. `id . id . id ...`

```
type String:
  char[64] _

type Person:
  String name, last_name
  int age
  float height

Person pepe
pepe.name._[0] = 'P'
pepe.name._[1] = pepe.name._[3] = 'e'
pepe.name._[2] = 'p'
pepe.last_name._[0] = 'G'
pepe.last_name._[1] = pepe.name._[5] = 'a'
pepe.last_name._[2] = 'r'
pepe.last_name._[3] = 'c'
pepe.last_name._[4] = 'i'
pepe.age = 32
pepe.height = 1.83
```

Ilustración 9 - Registros con propiedades de tipo registro

3.9- 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.

```
int myVariable = 10
float myFloat = 12.12

if myVariable:
  # ...
''' [movimento]
¿se puede utilizar myVariable de tipo int como condición?'''

myVariable + myFloat
''' [combinación]
¿cuál es el tipo resultante de la expresión?'''
```

Ilustración 10 - Categorías en las reglas de tipado

Para ambas categorías, debemos asegurarnos que el tipo de origen y del de destino son iguales, o bien que se puede utilizar una conversión del tipo de origen al tipo de destino sin pérdida de datos (algunos tipos pueden transformarse automáticamente, consultar 3.8.1- Conversiones automáticas).

```
float f1, f2
int i
bool b

f1 = 7.5    # Ok: float -> float
f2 = 0b11   # Ok: int -> float
i = 'a'     # Ok: char -> int
b = true    # Ok: bool -> bool

i = 7.5     # Error: float -> int
b = 7       # Error: int -> bool
b = f1      # Error: float -> bool
```

Ilustración 11 - 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 destino
MÁS (+), MENOS (-)	`int`	`int`
	`float`	`float`
	`char`	`char`
POR (*), ENTRE (/)	`int`	`int`
	`float`	`float`
MAYOR (>), MENOR (<), MAYOR O IGUAL (>=), MENOR O IGUAL (<=)	`int`	`bool`
	`float`	`bool`
	`char`	`bool`
IGUAL (==)	`int`	`bool`
	`float`	`bool`
	`char`	`bool`
	`bool`	`bool`
AND (and), OR (or), NOT (not)	`bool`	`bool`

Tabla 4 – Operadores y tipos de datos permitidos

3.9.1- Conversiones automáticas

No obstante, cuando una de las entradas de un operador binario tiene un tipo A y la otra entrada 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. `char` puede convertirse a `int` o `float`
II. `int` puede convertirse a `float`

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

```
5.1 * 41.0          # Ok: real * real -> real
char c = 'c' + 'c'  # Ok: char + char -> char
int i = 1 + 'c'     # Ok: int + char -> int

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

'''
Error: 1) 'd' -> char -> int (automatic conversion)
       2) (1) < 8 -> int < int -> bool
       3) 5.0 + (2) -> float * bool -> TYPE ERROR
'''
float err1 = 5.0 * ('d' < 8)

'''
Error: "if" require boolean, no se puede convertir
directamente de real a bool
'''
if 5 / 7:
  err1 = 0.0
```

Ilustración 12 - Ejemplos de reglas referentes al tipado

3.10- Control de flujo

En cuanto al control de flujo, se definen dos estructuras de control: salto y bucle condicional.

3.10.1- Salto condicional

El salto condicional es una sentencia que evalúa una expresión booleana – llamada condición – y, dependiendo de si el resultado es verdadero o falso, salta a un punto más avanzado del programa o sigue ejecutando las sentencias adyacentes.

Una de las posibles formas de definir la construcción del salto condicional podría ser:

```
if EXPRESION : \n BLOQUE_SENTENCIAS
```

- Si la expresión de la condición evalúa verdadero, se ejecutará el bloque de sentencias a continuación de los dos puntos y el salto de línea y después, se continuará el programa con normalidad.

```
if EXPRESION : \n BLOQUE_SENTENCIAS \n else BLOQUE_SENTENCIAS
```

- 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 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 13 - Ejemplo de código con control de flujo de salto condicional

3.10.2- Bucle condicional

El bucle condicional es una sentencia que evalúa una expresión booleana – llamada condición – y, dependiendo de si el resultado es verdadero o falso, ejecuta el cuerpo del bucle o bien salta al final de éste.

Una de las posibles formas de definir la construcción del bucle condicional podría ser:

```
while EXPRESION : \n BLOQUE_SENTENCIAS
```

```
int i, var = 0

while i < 10:
  var = (var + 1) * 2
  i = i + 1
```

Ilustración 14 - Ejemplo de código con control de flujo de bucle condicional

3.11- 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 desde cualquier punto de un programa, al igual que ocurre con las variables. Para declarar una función es necesario especificar:

- El nombre de la función.
- El nombre de sus valores de entrada o argumentos.
- El tipo de retorno.
- Las sentencias que forman parte del cuerpo de la función.
- La sentencia de retorno.

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

```
def TIPO id ( LISTA_ARGUMENTOS ) : \n BLOQUE_SENTENCIAS return SENTENCIA
```

La lista de argumentos será una serie de declaraciones (de uno o más elementos del mismo tipo) separados por punto y coma, indicando el tipo de los mismos.

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).

```
id ( LISTA_EXPRESIONES )
```

El resultado será un valor con el tipo de dato de retorno de la función. Hay que recordar que las expresiones que se pasan en la llamada de la función pueden ser operaciones o combinaciones de las mismas, variables o directamente valores del tipo esperado.

```
def int mod(int a, b):
  while a >= b:
    a = a - b
  return a

def int greatest_common_divisor(int a, b):
  int temp
  while not b == 0:
    temp = b
    b = mod(a, b)
    a = temp
  return a

int result = greatest_common_divisor(132, 0xFF)
```

Ilustración 15 - Ejemplo de código con definición y llamada de funciones

También podrá usarse los tipos complejos como argumentos o tipo de retorno en la función:

```
type String:
  char[64] _

type User:
  int user_id
  String nickname

int global_id = 0

def User new_user(String nickname):
  User user
  u.user_id = global_id
  global_id = global_id + 1
  u.nickname = nickname
  return user

String nickname
nickname._[0] = 'V'
nickname._[1] = 'i'
nickname._[2] = 'p'
nickname._[3] = 'e'
nickname._[4] = 'r'
new_user(nickname)
```

Ilustración 16 - Ejemplo de código con función usando tipos complejos

Recordamos que la extensión las tablas de variables y tipos para registrar las funciones definidas y hacer comprobaciones semánticas en las llamadas a función es una modificación avanzada.

I. El tipo de la expresión en la sentencia de retorno debe coincidir con el tipo de retorno de la declaración.
II. En las llamadas a funciones, el tipo de las expresiones debe coincidir con el registrado para la función con el mismo identificador, o en su defecto, que la conversión automática de tipos esté permitida.
III. En caso de existir sobrecarga de funciones, tendrá preferencia aquellas funciones declaradas con los mismos tipos exactos de la llamada, y en su defecto, las que menos conversiones automáticas de tipos requieran.
IV. NO se pide que se compruebe el contexto de las variables ya que esto complicaría las comprobaciones semánticas. Será una modificación avanzada para aquellos que quieran completar de manera más extensa el proyecto y será tenido en cuenta para la nota final.
    a. Las variables que se utilizan dentro del cuerpo de la función podrán tener contexto local a la función o global del fichero.
    b. Tendrán contexto local aquellas variables que se declaren en el cuerpo de la función o los propios argumentos de esta.
    c. No se podrá hacer nunca referencia a variables de contexto local de una función en el contexto global del fichero.

4· 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 error serían:

I. Léxico: Escribir mal un identificador, caracteres no reconocidos, etc.
II. Sintáctico: Falta un salto de línea entre sentencias.
III. Semántico: Multiplicar una 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.

5· ENTREGA PARCIAL

En una primera entrega se deberá completar la construcción del analizador léxico del lenguaje y la especificación de la gramática. El objetivo principal de esta entrega es conseguir una gramática sin conflictos ni ambigüedades, de cara a completar la entrega final sin acarrear errores previos.

Cada pareja deberá entregar todo el contenido de su práctica en un único archivo comprimido en formato zip. El nombre del comprimido debe seguir el siguiente formato `AP1_AP2_PL_P2.zip`

- AP1 y AP2 son los primeros apellidos de los integrantes en orden alfabético.
- Dentro del fichero zip, habrá una única carpeta con el mismo nombre.

Se realizarán pruebas extensivas tanto del analizador léxico como del sintáctico, por lo que:

I. Se generará un fichero con el nombre del fichero de entrada, pero extensión ".token", que incluirá una línea por cada token reconocido donde aparezca el tipo y el valor del token separados por un espacio en blanco.
II. El análisis sintáctico deberá ser correcto, lo que significa que no deberá aparecer ningún conflicto y/o error sintáctico.

```
# Fichero entrada - example.vip
int a = 28
float b = 24.9

# Ejecución
> python main.py example.vip
>> Generating LALR tables
>> ... # Sin conflictos, ni errores

# Fichero salida - example.token
INT int
ID a
EQUAL =
INT_VALUE 28
NEW_LINES \n
FLOAT float
ID b
EQUAL =
FLOAT_VALUE 24.9
NEW_LINES \n
```

Ilustración 17 - Ejemplo de ejecución y salida del analizador léxico

Sea adjuntara una breve, pero completa memoria que profundice en el razonamiento y proceso de toma de decisiones de los autores. Deberá servir como punto de partida para la memoria de la entrega final, donde la sección de análisis léxico no debería tener mucha variación. El aspecto más importante de esta memoria es explicar, detalladamente, el diseño de la gramática: cuál es la estructura principal de lenguaje, que se permite y qué no y como se ha definido esto en el diseño, cómo funcionan las expresiones, etc.

Aunque esta entrega será calificada, se podrán realizar cambios en el proyecto después de obtener la retroalimentación de cara a la entrega final.

6· ENTREGA FINAL

En la entrega final, la práctica deberá contener las tareas indicadas (analizador léxico, analizador sintáctico, tabla de símbolos, tabla de registros y comprobaciones/análisis semántico).

La gramática ha de ser diseñada de tal manera que el analizador sintáctico no tenga conflictos de ningún tipo.

6.1- Ejemplos

6.1.1- Variables básicas y saltos condicionales

```
'''Operaciones aritméticas y de comparación
-------------------------------------------'''

float f1, f2
bool b1
char c = 'h' # char variable and its literal value

f1 = 5 * 3 - 80 / 10    # this equals 7
f2 = 10 / 5 * f1        # this equals 4
b1 = 5 < 3              # this equals true

# check whether a number is positive or not
float num
bool is_positive
if num >= 0:
  is_positive = true
else:
  is_positive = false

# max calculation
float a, b, c, _max
a = 1
b = 2
c = 3
if a >= b and a >= c:
  _max = a
else:
  if b >= a and b >= c:
    _max = b
  else:
    _max = c
```

TABLA DE SÍMBOLOS	
Tipo	Nombre
float	f1
float	f2
bool	b1
char	c
float	num
bool	is_positive
float	a
float	b
float	c
float	_max

En este ejemplo, todas las variables son de tipos básicos predefinidos. Se muestra también la lista completa de símbolos (tabla de símbolos) que se genera a partir del mismo.

6.1.2- Registros y vectores

```
'''Declaración de variables y tipos'''
type Vector2D:
  float x1, x2

type Ball:
  Vector2D center
  float radius

type Word:
  char[10] s

type Human:
  Word name, last_name_1, last_name_2
  int age

Ball ball
Human student

# Expressions
ball.center.x1 = 0.0
ball.center.x2 = 0.0
ball.radius = 10.0

student.name.s[0] = 'S'
student.name.s[1] = 'a'
student.name.s[2] = 'u'
student.name.s[3] = 'l'

student.last_name_1.s[0] = 'P'
student.last_name_1.s[1] = 'e'
student.last_name_1.s[2] = 'r'
student.last_name_1.s[3] = 'e'
student.last_name_1.s[4] = 'z'

student.last_name_2.s[0] = 'G'
student.last_name_2.s[1] = 'o'
student.last_name_2.s[2] = 'm'
student.last_name_2.s[3] = 'e'
student.last_name_2.s[4] = 'z'

student.age = 19

# Conditional loop
bool go = true
while go and ball.radius > 0:
  if ball.center.x1 < 0 and ball.center.x2 < 0:
    go = false
  else:
    go = true
  ball.radius = ball.radius / 2.0
  ball.center.x1 = ball.center.x1 - 1
  ball.center.x2 = ball.center.x2 - 1
```

TABLA DE SÍMBOLOS

Tipo	Nombre
Ball	ball
Human	student
bool	go

TABLA DE REGISTROS

Nombre	Campos
Vector2D	float: x1
	float: x2
Ball	Vector2D: center
	float: radius
Word	vector(char): s
Human	Word: name
	Word: last_name_1
	Word: last_name_2
	int: age

En este segundo ejemplo tenemos variables declaradas de tipos construidos en el programa. Por tanto, además de la tabla de símbolos, necesitamos una tabla de registros.

6.1.3- Declaración y uso de funciones

```
'''Declaración de funciones'''
def float square(float f):
  return f * f

def int square(int i):
  return i * i

def float power(float f, int i):
  float r = 1
  int s = 1
  while s < i:
    r = r * f
    s = s + 1
  return r

float decimal
int integer = 6

# Expressions
5 * 0b11
decimal = 3.7
decimal = 3.7 + square(decimal)
decimal = power(decimal, integer)

if integer == 3 or decimal == 2:
  integer = integer + 0xAB
else:
  decimal = square(square(integer))
```

TABLA DE SÍMBOLOS	
Tipo	**Nombre**
float	decimal
int	integer

TABLA DE REGISTROS	
Nombre	**Campos**
square	input: float
	output: float
square	input: int
	output: int
power	input: float, int
	output: float

En este tercer ejemplo, tenemos variables de tipos básicos además de tres funciones declaradas: las dos primeras funciones declaradas con sobrecarga de nombres.

Por tanto, además de la tabla de símbolos, necesitamos una tabla de registros que refleje las diferentes funciones con su forma (entradas y salida).

6.2- Especificación de la salida

Los analizadores deberán:

- Generar información de los diferentes tipos de errores del programa de entrada, indicando claramente los tipos de error.
    - Específicamente, a nivel semántico, podemos encontrar errores como: variables no declaradas, asignación de tipos incompatibles, usar un campo de un registro en una expresión inválida, errores de tipos en expresiones o condiciones, llamada a función con tipos incorrectos, etc.
    - El mensaje de error debe ser claro y completo, cuánta más información contenga, mejor (número de línea, columna de comienzo y fin del bloque, etc.)

```
# Fichero entrada - example.vip
int a = 28
type Point:
  float x, y
Point point

# Ejecución
> python main.py example.vip
>> Generating LALR tables
>> ... # Sin conflictos, ni errores

# Ficheros de salida - example.token
INT int
ID a
EQUAL =
INT_VALUE 28
NEW_LINES \n
TYPE type
ID Point
COLON :
NEW_LINES \n
INDENT INDENT
FLOAT float
ID x
COMMA ,
ID y
NEW_LINES \n
DEDENT DEDENT
ID Point
ID point
EOF EOF

# Ficheros de salida - example.symbol
int a
Point point

# Ficheros de salida - example.record
Point float:x, float:y
```

Ilustración 18 - Ejemplo de ejecución y ficheros de salida

- En caso de no tener errores, la salida de la ejecución en la terminal deberá ser vacía (sin ningún conflicto al generar las tablas LALR)
    - Se generará un fichero con el mismo nombre que el de entrada pero de extensión ".token", que incluirá una línea por cada token reconocido donde aparezca el tipo y el valor del token separados por un espacio en blanco.
    - Se generará un fichero con el mismo nombre que el de entrada pero de extensión ".symbol", que incluirá una línea por cada símbolo en la tabla de símbolos, donde aparezca el tipo y el nombre del símbolo separados por un espacio en blanco.
    - Se generará un fichero con el mismo nombre que el de entrada pero de extensión ".record", que incluirá una línea por cada registro en la tabla de registros, donde aparezca el nombre del registro y la lista de campos (o entrada/salida en el caso de las funciones) separados por comas.
- Es importante que en esta entrega se adjunten ficheros con pruebas que demuestren la correcta implementación de cada uno de los objetivos: es muy relevante documentar bien las pruebas realizadas (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.

6.3- Memoria

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 comprimido, 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: detalles de mayor complejidad en la parte léxica, decisiones de diseño, explicación en bloques de la gramática y funciones de las reglas de producción, los controles semánticos realizados, etc.
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).

6.4- Entrega

Cada pareja deberá entregar todo el contenido de su práctica en un único archivo comprimido en formato zip. El nombre del comprimido debe seguir el siguiente formato `AP1_AP2_PL_P3.zip`

- AP1 y AP2 son los primeros apellidos de los integrantes en orden alfabético.
- Dentro del fichero zip, habrá una única carpeta con el mismo nombre que contendrá:
    - La memoria - `AP1_AP2_memoria.pdf`
    - El punto de entrada de ejecución – `main.py`
    - El analizador léxico – `lexer.py`
    - El analizador sintáctico con los controles semánticos – `parser.py`
    - Cualquier otro fichero de código que se haya programado para mejorar la legibilidad del código – `utils.py | model.py | shared.py | etc…`

- La carpeta con los ficheros de prueba que forman parte de la entrega – `/test_files/**.vip`
- De manera opcional, un pequeño fichero con comentarios sobre cómo ejecutar el código y otras consideraciones en relación con el proyecto – `README.md`

6.5- 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.