Introducción al entorno

PROCESADORES DEL LENGUAJE

Fecha 22/4/2024

Curso
2023-2024

Grupo
80

Curso 2023-2024

Titulación
Grado en Ingeniería Informática

Integrantes
Luis Gomez-Manzanilla Nieto – 100472006@alumnos.uc3m.es
Carlos Seguí Cabrera – 100472060@alumnos.uc3m.es

Índice

1. Introducción

En este documento recogemos una breve descripción del trabajo realizado en la práctica con objetivo de proporcionar una explicación extra.

En esta práctica vamos a implementar un compilador capaz de analizar programas del lenguaje AJS (basándonos en el formato AJSON trabajado en la práctica de introducción) . El compilador se compone de un analizador léxico que reconoce la entrada del fichero y genera los tokens que servirán como entrada para el analizador sintáctico.

Dentro del proyecto existen 4 archivos principales. Están el *lexer_ajs.py* y el *parser_ajs.py* que implementan la lógica del compilador. En el fichero *tokens.py* se recoge una clase llamada Tokens que contiene las declaraciones de los tokens y las palabras reservadas, y se importa en los dos archivos anteriores.

Por último, en el fichero *main.py* se ejecuta el compilador y se imprime la salida como se ha mostrado en el enunciado.

El código se ejecuta ejecutando el archivo *main.py* por terminar pasando por argumentos la ruta del archivo de entrada y en qué modo se quiere ejecutar el compilar. Se le puede especificar que solo ejecute el analizador léxico pasándole como último parámetro *"-lex"* o *"-par"* si se quiere ejecutar el analizador sintáctico. En caso de querer ejecutar ambos no es necesario que le pases ningún parámetro extra.

En esta primera entrega parcial, hemos implementado el lexer, y la gramática. Ya que se trata de la primera parte de un proyecto más amplio nos centraremos en estos puntos en el siguiente documento.

2. Reconocer léxico

AJS es un lenguaje de programación imperativo, débilmente tipado y dinámico (trabaja varios tipos y permite definir compuestos basándose en JavaScript). Para reconocer el reconocedor léxico del mismo hemos seguido las siguientes instrucciones. *Para resaltar los cambios con respecto a la entrega anterior marcaremos con * la documentación que ha cambiado o es relevante para esta práctica.*

- Números: Para este token resulta más prudente construir una conjunción de los regex de todos sus tipos, para ello se ha especificado en primera instancia cada uno de estos, su estructura contemplada es la siguiente:

  -Números enteros: **r"\d+"** → {..., -3, -2, -1, 0, +1, +2, +3, ...}.
  -Números decimales(reales): **r"(\d+\.\d+)|(\.\d+)"** → $\{n/10^p: n \in Z, p \in |Z|\}$.
  -Notación científica: **r"((\d+\.\d+)|(\.\d+)|(\d))[eE]-?\d+"** → $\{m \times \mathbf{10}^n: m \in D(1,10), N \in |Z|\}$ (**10** → -e, E).

-*Números binarios*: **r"0(b|B)[01]+"** → Conjunto de cadenas formadas por dígitos {0,1}.
-*Números octales:* **r"0[0-7]+"** → Conjunto de cadenas formadas por los dígitos (0-7).
-*Números hexadecimales:* **r"0(x|X)[0-9a-fA-F]+"** → Conjunto de cadenas formadas por dígitos (0-9) U (A-F).

Para procesar y convertir las diferentes cadenas de texto que representan números en valores numéricos adecuados, para ello seguirá la siguiente estructura de conversión:

-*0b|0B* → int en base 2
-*0x|0x* → int en base 16
-*0(0-7)* → int en base 8
-.|*e|E* → float
-*Resto* → int

***** *Es importante resaltar que se han creado dos token diferenciados para enteros y decimales, para el correcto funcionamiento de los tipos y las operaciones. (AJS maneja de manera interna:*
-([entero/int],[real/float],[carácter/character],[Booleano/boolean])

- Booleanos/ Valor Nulo: Los booleanos utilizarán las palabras reservadas (True y False), reconociendo {*TR|FL|tr|fl*}. Del mismo modo el valor nulo utilizará la palabra reservada (None), reconociendo {NULL|null}.

  Para evitar la ambigüedad, consistencia y legibilidad vamos utilizar palabras reservadas para el reconocedor de estos tokens. Para ello haremos un mapeado a las representaciones de estas en código fuente **(diccionario reserver_map)**.
  El **diccionario reserver_map** contiene (palabras reservadas tanto en mayúsculas como minúsculas [claves] y sus representaciones (False, True, Null) [valores]).

  Ya que denotan un valor en la cadena sin comillas se identificarán estos tipos y se les asignará su valor correspondiente **(“TR”→True, “FL”→ False, “Null”→ None)**.

- * Objetos -> cadenas de caracteres/carácter: Encontramos dos tipos de cadenas de caracteres:
  (Las claves siguen la misma estructura que del formato AJSON.)
  -*Sin comillas(clave)* : Empiezan exclusivamente por {letra (*may./min.)*| *_}* y son formadas por (letras (*may./min.)*| _| *Números),* **(r"[a-zA-Z_][a-zA-Z_0-9]*")**.

  -*Con comillas (clave)* : Están formadas por cadenas de cualquier carácter menos {\n, “} y contenidas en (“). Su valor explícito será el contenido eliminado las comillas dobles (“), **(r"\"[^\"]*\"")**.

(Para los valores las cadenas se han sustituido por los caracteres)
-Carácter:Cualquier carácter de la codificación ASCII-extendido delimitado por comillas simples. **`r"'[\x00-\xFF]?'"->`** el valor del token es aquel entre las comillas simples, excepto si solo tenemos comillas simples ‘’ en cuyo caso su valor será None.

- Operadores de comparación/Delimitadores: Se reconocen los siguientes *operadores y delimitadores* (“==” (igual), “>” (mayor que), “>=” (mayor o igual que), “<” (menor que), “<=” (menor o igual que),“{“ (llave de apertura), “}” (llave de cierre), “:” (dos puntos), “,” (coma)). Como añadido para la mejora de los arrays también reconoce (“[“ (corchete de apertura), “]” (corchete de cierre)). * Además se reconocen *aritméticas y booleanas* Conjunción (&&), disyunción (||) y negación (!) ,operaciones binarias (+, -, *, /) y unarias (+ y -).
- * Comentarios: En un principio los especificamos como tokens, pero observamos que no era necesario para la lectura de la gramática, por ello ya no son tokens. (comentario de línea → **r"//.*"** ; comentario de bloque → **`r'/\*(.|\n)*?\*/'`**)
- * Sentencias: Para la evaluación de expresiones y asignaciones se han creado tokens como asignación y el empleo de palabras reservadas, como if/else o while para los bucles.

Con ello todos los tokens contemplados serán los siguientes:

- **tokens** = ["LLAVE_ABRE","LLAVE_CIERRA","COMA","DOS_PUNTOS","ENTERO","DECIMAL","CADENA_COMILLAS","CADENA_NO_COMILLAS","IGUAL","MAYOR","MENOR","MAYOR_IGUAL","MENOR_IGUAL","CORCHETE_ABRE","CORCHETE_CIERRA","PUNTO_Y_COMA","PUNTO","PARENTESIS_ABRE","PARENTESIS_CIERRA","CONJUNCION","DISYUNCION","NEGACION","SUMA","RESTA","MULTIPLICACION","DIVISION","ASIGNACION","CARACTER"] .
- **reserved** = ["TR", "FL", "LET", "INT", "FLOAT", "CHARACTER", "WHILE", "BOOLEAN", "FUNCTION", "RETURN", "TYPE", "IF", "ELSE", "NULL"].

3. Reconocedor sintáctico

El reconocedor sintáctico que hemos implementado se basa en una gramática recursiva a derechas que aunque sabiendo que el parser se beneficia de las gramáticas recursiva a la izquierda, a la hora de implementarlo nos daba diversos errores.

La gramática que hemos implementado es la siguiente:

Unset

```
    '''programa : statement
                | empty'''
    '''empty : '''
    '''
    statement : content PUNTO_Y_COMA
              | content PUNTO_Y_COMA statement
              | no_semicolon
              | no_semicolon statement
    '''
    '''
    content : define
            | declare
            | assign
    '''
    '''
    no_semicolon : condition
                 | loop
                 | function_def
    '''
    '''ident : CADENA_NO_COMILLAS
             | CADENA_NO_COMILLAS PUNTO ident
             | CADENA_NO_COMILLAS CORCHETE_ABRE CADENA_COMILLAS
CORCHETE_CIERRA
             | CADENA_NO_COMILLAS CORCHETE_ABRE CADENA_COMILLAS
CORCHETE_CIERRA PUNTO ident
             '''
    '''
    declare : LET id
    '''
    '''id : variable
          | variable COMA id
          | variable ASIGNACION valor
          | variable ASIGNACION valor COMA id'''

    '''variable : ident
                | ident DOS_PUNTOS tipo'''
```

```
'''
assign : ident ASIGNACION valor
'''
'''valor : ident
         | ENTERO
         | DECIMAL
         | operacion
         | TR
         | FL
         | NULL
         | ajson_v
         | CARACTER
         | function_call
         | PARENTESIS_ABRE valor PARENTESIS_CIERRA
         | SUMA valor %prec UPLUS
         | RESTA valor %prec UMINUS
'''
'''
define : TYPE CADENA_NO_COMILLAS ASIGNACION ajson
'''
'''ajson : LLAVE_ABRE lista LLAVE_CIERRA'''
'''lista : elemento
         | elemento COMA
         | elemento COMA lista'''
'''elemento : clave DOS_PUNTOS valor_t'''
'''valor_t : tipo
           | ajson'''
'''clave : CADENA_NO_COMILLAS
         | CADENA_COMILLAS'''
'''ajson_v : LLAVE_ABRE lista_v LLAVE_CIERRA'''
'''lista_v : elemento_v
           | elemento_v COMA
           | elemento_v COMA lista_v'''
'''elemento_v : clave_v DOS_PUNTOS valor'''
'''clave_v : CADENA_NO_COMILLAS
           | CADENA_COMILLAS'''
'''operacion : aritmetica
             | booleana
             | comparacion'''
'''aritmetica : valor SUMA valor %prec SUMA
              | valor RESTA valor %prec RESTA
              | valor MULTIPLICACION valor %prec MULTIPLICACION
              | valor DIVISION valor %prec DIVISION'''
'''comparacion : valor MAYOR valor %prec MAYOR
               | valor MENOR valor %prec MENOR
               | valor MAYOR_IGUAL valor %prec MAYOR_IGUAL
```

```
                    | valor MENOR_IGUAL valor %prec MENOR_IGUAL
                    | valor IGUAL valor %prec IGUAL'''
    '''booleana : valor CONJUNCION valor %prec CONJUNCION
                | valor DISYUNCION valor %prec DISYUNCION
                | NEGACION valor %prec NEGACION'''
    '''tipo : INT
            | FLOAT
            | CHARACTER
            | BOOLEAN
            | CADENA_NO_COMILLAS
    '''
    '''
    condition : IF PARENTESIS_ABRE valor PARENTESIS_CIERRA LLAVE_ABRE
statement LLAVE_CIERRA
              | IF PARENTESIS_ABRE valor PARENTESIS_CIERRA LLAVE_ABRE
statement LLAVE_CIERRA ELSE LLAVE_ABRE statement LLAVE_CIERRA
    '''
    '''
    loop : WHILE PARENTESIS_ABRE valor PARENTESIS_CIERRA LLAVE_ABRE
statement LLAVE_CIERRA
    '''
    '''
    function_def : FUNCTION CADENA_NO_COMILLAS PARENTESIS_ABRE id
PARENTESIS_CIERRA DOS_PUNTOS tipo LLAVE_ABRE statement RETURN valor
PUNTO_Y_COMA LLAVE_CIERRA
                 | FUNCTION CADENA_NO_COMILLAS PARENTESIS_ABRE id
PARENTESIS_CIERRA DOS_PUNTOS tipo LLAVE_ABRE RETURN valor PUNTO_Y_COMA
LLAVE_CIERRA
                 | FUNCTION CADENA_NO_COMILLAS PARENTESIS_ABRE
PARENTESIS_CIERRA DOS_PUNTOS tipo LLAVE_ABRE statement RETURN valor
PUNTO_Y_COMA LLAVE_CIERRA
                 | FUNCTION CADENA_NO_COMILLAS PARENTESIS_ABRE
PARENTESIS_CIERRA DOS_PUNTOS tipo LLAVE_ABRE RETURN valor PUNTO_Y_COMA
LLAVE_CIERRA
    '''
    '''
    function_call : CADENA_NO_COMILLAS PARENTESIS_ABRE arg PARENTESIS_CIERRA
                  | CADENA_NO_COMILLAS PARENTESIS_ABRE PARENTESIS_CIERRA
    '''
    '''arg : valor
           | valor COMA arg'''
```

Dicha gramática reconoce los ficheros AJS y no presenta ningún conflicto ni reducción/reducción ni desplazamiento/reducción . La creación de la gramática ha supuesto un proceso iterativo ya que inicialmente la gramática contenía diversos conflictos pero con diversos cambios los íbamos eliminando.

Inicialmente, en lo relativo a la parte de las operaciones, tratamos de crear una gramática que se ahorrarse el tema de comprobar los tipos para el análisis semántico. A continuación, muestro un ejemplo de lo que habíamos hecho:

```
Unset
 '''aritmetica : ex operador1 ex
                 | ex2 operador2 ex2'''
    '''operador1 : SUMA
                 | RESTA'''
    '''operador2 : MULTIPLICACION
                 | DIVISION'''
    '''ex : num
          | CHARACTER
          | CADENA_NO_COMILLAS
          | aritmetica
          | comparacion
          | PARENTESIS_ABRE ex PARENTESIS_CIERRA'''
    '''ex2 : num
           | CADENA_NO_COMILLAS
           | aritmetica
           | PARENTESIS_ABRE ex2 PARENTESIS_CIERRA'''
    '''num : ENTERO
           | DECIMAL'''
```

Esta implementación proporciona mucha combinatoria que resulta en diversos conflictos. Debido a esto, descartamos esta implementación y le dejamos más carga al analizar semántico.

Uno de los problemas que tuvimos fue donde crear la recursividad para dar libertad de poner paréntesis de agrupamiento a los datos y acabamos por determinar que la mejor opción es introducir dicha recursividad en el propio valor de la siguiente manera:

```
Unset
'''valor : ident
            | ENTERO
            | DECIMAL
            | operacion
            | TR
            | FL
            | NULL
            | ajson_v
            | CARACTER
```

```
              | function_call
              | PARENTESIS_ABRE valor PARENTESIS_CIERRA
              | SUMA valor %prec UPLUS
              | RESTA valor %prec UMINUS
'''
```

También quiero destacar como la gramática tiene reglas que parecen duplicadas a la hora de definir y asignar tipos objetos. Esto se debe a que en la declaración, lo que se busca es proporcionar los atributos del objeto y el tipo de estos. En cambio, al asignar, se puede dar cualquier tipo de valor a los atributos en cuanto a la parte sintáctica, posteriormente será necesario comprobar que se respete el tipo asignado en dicha definición.

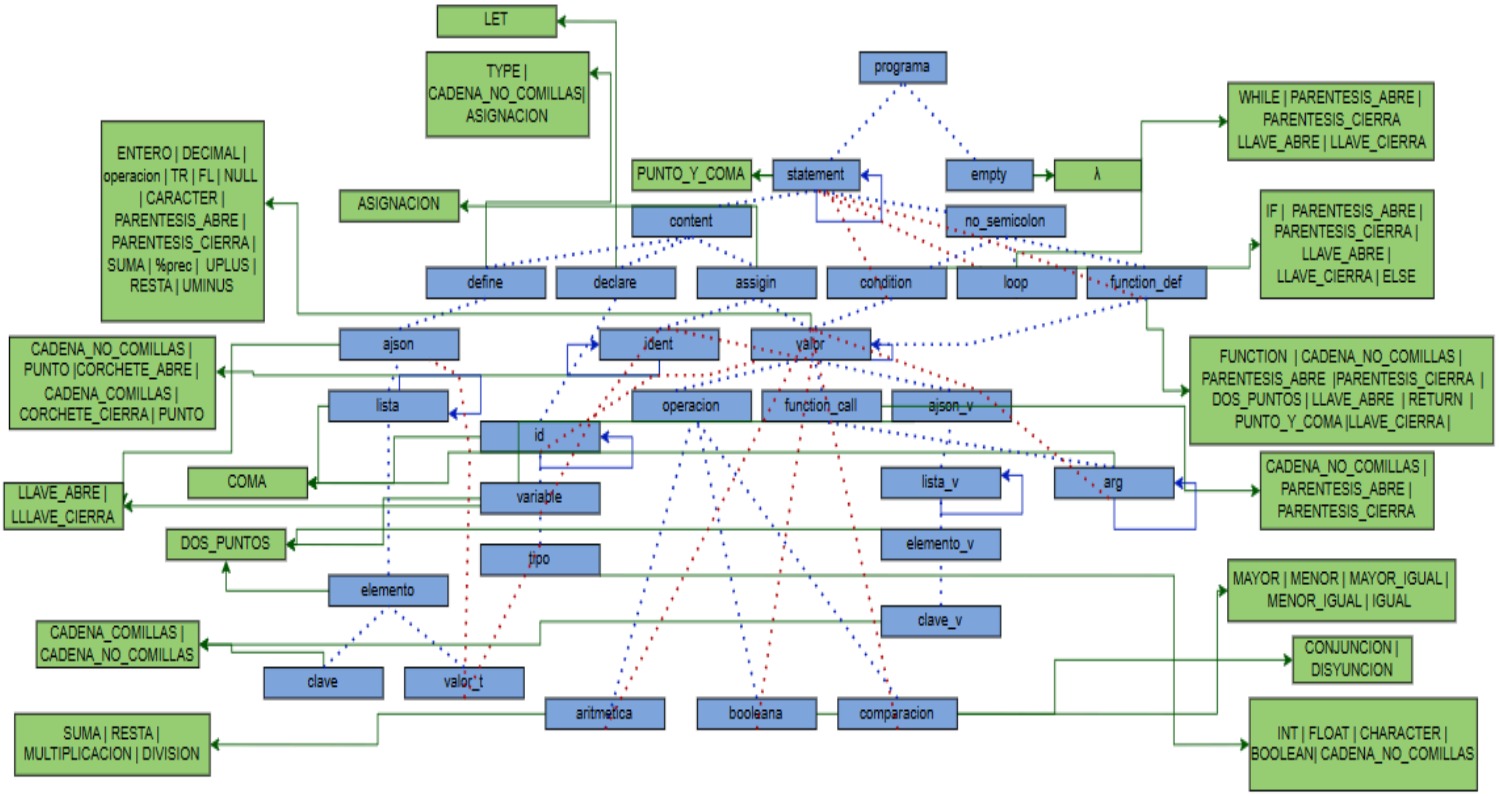
Por otro lado, hemos añadido está estructura de preferencia que soluciona diversos conflictos y determina la asociatividad de operaciones.

```
Unset
precedence = (
        ('left', 'SUMA', 'RESTA'),
        ('left', 'MULTIPLICACION', 'DIVISION'),
        ('right', 'UPLUS', 'UMINUS'),
        ('left', 'CONJUNCION', 'DISYUNCION'),
        ('right', 'NEGACION'),
        ('nonassoc', 'MENOR_IGUAL', 'MENOR', 'MAYOR_IGUAL', 'MAYOR',
'IGUAL'),
    )
```

Con esto, nos aseguramos de realizar primero los cambios de signo, posteriormente las multiplicaciones o divisiones, seguidas de las sumas o restas, seguidas de las comparaciones y terminando con las operaciones booleanas.

Por último, hemos implementado para representar el símbolo de un valor que se pueda hacer recursivo. Ejemplo, “--3” tendrá de valor “3”.

Para comprender la relación de la gramática hemos creado un diagrama de la misma:
(Azul → No terminales (aquellas reglas que creen otras que se encuentran arriba en el árbol → rojas (legibilidad)) ,
Verde → terminales (tokens))



4. Batería de pruebas

Acorde con la comprobación de la gramática y el lexer hemos implementado las siguientes pruebas (**se definen diversos aspectos del lenguaje y se comprueba su correcto funcionamiento**).

Hemos creado una batería de pruebas compuesta por 10 archivos “test{i}.ajs” (siendo i el número del test). Hemos creado dos tipos de pruebas, del test 1 al 3 nos centramos en comprobar el analizador léxico y del 4 al 10 el sintáctico. Queremos destacar que para la futura entrega, estos segundos test también servirán para comprobar la parte semántica.

De los test del léxico, el primero se centra en mostrar cómo trata el compilador los comentarios. El segundo test es mucho más completo y se centra en mostrar como reconoce todo los tokens. En el tercer test nos hemos centrado en mostrar diversos errores léxicos o situaciones a tener en cuenta. Por ejemplo, el “#” es un elemento que no reconoce nuestra gramática o al poner “=>”, en vez de reconocerlo como un mayor igual lo reconoce como una asignación y un mayor. También comprobamos como con el símbolo de

conjunción o disyunción, compuestos por dos “&” o “|”, al poner tres seguidos, reconoce los dos primeros pero no el tercero y lo mismo con el comparador de igualdad. Además podemos ver como las palabras reservadas, para que sean reconocidas como tal, es necesario escribirlas en minúsculas, en caso contrario se reconocen como cadenas sin comillas.

En cuanto al otro conjunto destinado a probar el parser seguimos la misma dinámica, del test 4 al 9 son test que comprueban que funcionan correctamente, y el test número 10 es un test que prueba errores. Entre dichos errores comprobamos que falte una llave de cierre o un corchete de apertura, que falte un punto y coma al final de una sentencia o que haya una sentencia que sea solo un punto y coma, y que una cadena que usa comillas se abra pero no se cierre.

tests (Descripción de la prueba)	Salida
Lexer: test1 (comprobación de los comentarios)	CADENA_NO_COMILLAS Comentario1 PUNTO_Y_COMA ; CADENA_NO_COMILLAS Comentario2 PUNTO_Y_COMA ; CADENA_NO_COMILLAS Comentario3 PUNTO_Y_COMA ; CADENA_NO_COMILLAS Comentario4 PUNTO_Y_COMA ;
Lexer: test2 (comprobación de tokens correctos)	CARACTER None ENTERO 55 DECIMAL 5.5 DOS_PUNTOS : ENTERO 1 MAYOR > ENTERO 3 ENTERO 255 MAYOR_IGUAL >= ENTERO 59 DECIMAL 0.1 IGUAL == DECIMAL 0.1 ENTERO 63 MENOR < ENTERO 62 DECIMAL 14000000000.0 ENTERO 3 DECIMAL 1000.0 DECIMAL 50.0 DECIMAL 5.5 MULTIPLICACION *

	ENTERO 7 CADENA_NO_COMILLAS a ASIGNACION = ENTERO 5 MULTIPLICACION * DECIMAL 5.5 ENTERO 5 MAYOR_IGUAL >= ENTERO 3 LLAVE_ABRE { LLAVE_CIERRA } CORCHETE_ABRE [CORCHETE_CIERRA] CADENA_NO_COMILLAS a PUNTO . CORCHETE_ABRE [CADENA_COMILLAS variable CORCHETE_CIERRA] ASIGNACION = ENTERO 45 CADENA_NO_COMILLAS a ASIGNACION = CARACTER d ELSE else COMA , CADENA_NO_COMILLAS abc DOS_PUNTOS : CADENA_NO_COMILLAS cba ENTERO 44 MENOR_IGUAL <= ENTERO 44 COMA , ENTERO 44 MAYOR_IGUAL >= ENTERO 44 PUNTO_Y_COMA ; PARENTESIS_ABRE (PARENTESIS_CIERRA) CONJUNCION && DISYUNCION || NEGACION ! LET let BOOLEAN boolean FUNCTION function

	CADENA_NO_COMILLAS calculo ASIGNACION = ENTERO 44 RESTA - ENTERO 3 SUMA + ENTERO 33 CADENA_NO_COMILLAS condicion ASIGNACION = TR True CADENA_NO_COMILLAS condicion ASIGNACION = NULL None CADENA_NO_COMILLAS condicion ASIGNACION = FL False WHILE while CADENA_NO_COMILLAS condicion DOS_PUNTOS : IF if ENTERO 44 IGUAL == ENTERO 44 DOS_PUNTOS : RETURN return ELSE else DOS_PUNTOS : TYPE type PARENTESIS_ABRE (ENTERO 33 PARENTESIS_CIERRA)
Lexer: test3 (comprobación de tokens incorrectos)	Illegal character '#' ENTERO 1 ASIGNACION = MAYOR > ENTERO 3 ENTERO 3903 ASIGNACION = NEGACION ! ENTERO 50 CADENA_NO_COMILLAS b111011 ENTERO 1 CADENA_NO_COMILLAS e SUMA + RESTA -

	ENTERO 1 ASIGNACION = MAYOR > DECIMAL 0.1 CADENA_NO_COMILLAS ELSE PUNTO_Y_COMA ; PARENTESIS_ABRE (PARENTESIS_CIERRA) CONJUNCION && Illegal character '&' DISYUNCION || Illegal character '|' CADENA_NO_COMILLAS LET CADENA_NO_COMILLAS BOOLEAN FUNCTION function CADENA_NO_COMILLAS calculo ASIGNACION = ENTERO 44 RESTA - ENTERO 3 SUMA + ENTERO 33 CADENA_NO_COMILLAS condicion ASIGNACION = CADENA_NO_COMILLAS false CADENA_NO_COMILLAS condicion ASIGNACION = NULL None CADENA_NO_COMILLAS condicion ASIGNACION = CADENA_NO_COMILLAS True CADENA_NO_COMILLAS WHILE CADENA_NO_COMILLAS condicion DOS_PUNTOS : Illegal character '"' IF if Illegal character '"' ENTERO 44 IGUAL == ASIGNACION = ENTERO 44 DOS_PUNTOS : CADENA_NO_COMILLAS RETURN CADENA_NO_COMILLAS ELSE DOS_PUNTOS : CADENA_NO_COMILLAS TYPE PARENTESIS_ABRE (ENTERO 33 PARENTESIS_CIERRA)
Parser: test4 (asignación y	Salida correcta

operaciones)	
Parser: test5(if con condiciones)	Salida correcta
Parser: test6(objetos)	Salida correcta
Parser: test7(objetos con nombrado con corchetes)	Salida correcta
Parser: test 8(funciones)	Salida correcta
Parser: test 9(programa más complejo con nested en valores)	Salida correcta
Parser: test 10 (ajs incorrecto)	[ERROR][PARSER] At line: LexToken(PUNTO_Y_COMA,';',8,244) Illegal character '' [ERROR][PARSER] At line: LexToken(CADENA_NO_COMILLAS,'prop2',20,487) [ERROR][PARSER] At line: LexToken(LLAVE_CIERRA,'}',22,521) Illegal character '" [ERROR][PARSER] At line: LexToken(CADENA_NO_COMILLAS,'error',23,547)

5. Conclusiones

Esta primera entrega parcial, es una introducción al análisis y desarrollo de un compilador más complejo, y una gramática de más peso como es aquella que referencia a un lenguaje como es AJS. Aun con su complicaciones resulta una aproximación de gran utilidad para ampliar nuestros conocimientos sobre los compiladores, tras ya conocer los fundamentos a nivel de análisis léxico, sintáctico y semántico.

Esta sin embargo, solo es la primera parte del proyecto, en conjunto nos harán aplicar y perfeccionar todos los conocimientos adquiridos en la materia.