

PRÁCTICA 1: ANALIZADOR DESCENDENTE RECURSIVO



Universidad Carlos III
Grado en ingeniería informática
Curso Procesadores del Lenguaje 2024-25
Práctica 1: drLL
Curso 2024-25

GRUPO: 403

Fecha: 11/03/2025

Alumnos: Mario Ramos Salsón (100495849)

Miguel Yubero Espinosa (100495984)

ÍNDICE

1. Elementos léxicos proporcionados	2
2. Gramática	2
2.2 Paréntesis	3
2.3 Variables	3
2.4 Fotos	4
2.5 Transformaciones aplicadas	4
2.6 Distinción entre niveles de léxico y sintáctico	5
3. Resumen del diseño del parser	5
3.1 ParseExpression().	6
3.2 ParseAxiom() y ParseYourGrammar().	8
4. Diagrama sintáctico de la gramática propuesta	9
5. Batería de pruebas	10

1. Elementos léxicos proporcionados

La función proporcionada *rd_lex()* se encarga de leer la entrada del programa y reconocer diferentes elementos léxicos o tokens que serán utilizados por el parser. Estos elementos léxicos los podemos categorizar en:

- Números enteros: La función es capaz de reconocer cualquier sentencia de dígitos al igual que un número entero.
- Operadores aritméticos: La función reconoce los 4 operadores aritméticos básicos. La suma "+", la resta "-", la multiplicación "*" y la división "/".
- Variables: La función también reconoce los nombres de las variables que el parser puede usar. Estos nombres están limitados a dos formaciones. La primera de ellas es cualquier letra ya sea mayúscula o minúscula, y la segunda es cualquier letra mayúscula o minúscula pero terminada por un dígito.
- Caracteres literales: La función también reconoce cualquier carácter literal que no cumple con ninguna de las reglas mencionadas anteriormente. Estos caracteres se devuelven directamente como tokens.

2. Gramática

Para el desarrollo de la gramática hemos decidido ir de forma progresiva implementando primero las funcionalidades más básicas y poco a poco añadiendo las más complejas.

Queremos destacar que para probar la gramática con el programa JFLAP hemos tenido que hacer algunos cambios a la hora de representar la gramática:

- Sustitución de los paréntesis "()" por "<>" para que JFLAP los pueda leer.
- Sustitución de los números por un terminal genérico "n" para poder realizar las pruebas.
- Sustitución del nombre de las variables por otro terminal genérico "v" para poder realizar las pruebas.

2.1 Números

La primera funcionalidad que hemos añadido ha sido las operaciones básicas entre dos números y la aceptación de números aislados.

```
C/C++  
E → N  
S → OEE  
O → + | - | * | /  
N → n // cualquier número
```

2.2 Paréntesis

La segunda funcionalidad que hemos añadido es la aceptación de expresiones con paréntesis y todas sus diversas concatenaciones.

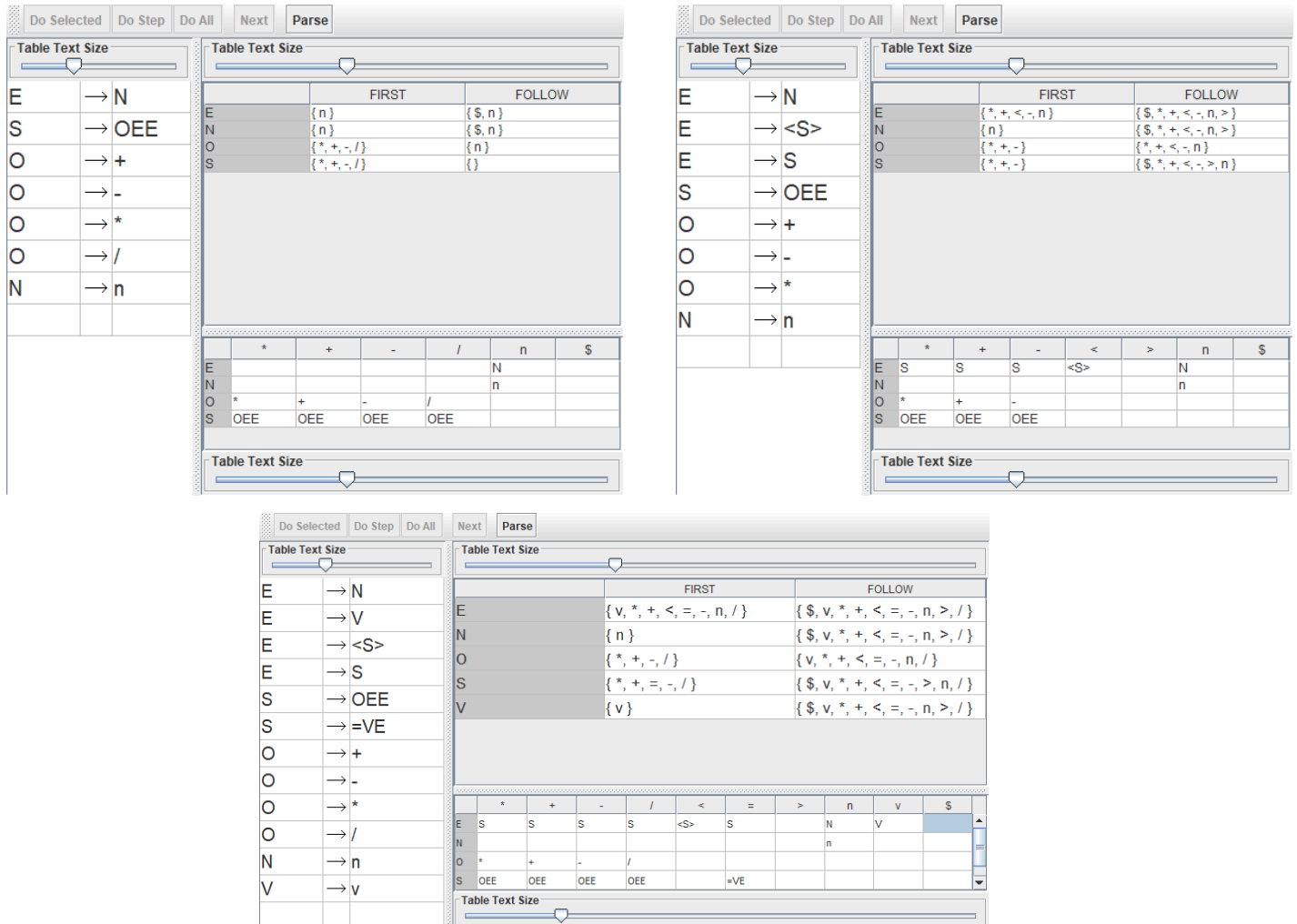
```
C/C++  
E → N | S | <S>  
S → OEE  
O → + | - | * | /  
N → n // cualquier número
```

2.3 Variables

La última funcionalidad que hemos añadido es la aceptación de variables. Tanto su declaración, como su impresión, y su operación con el resto de números y variables.

```
C/C++  
E → N | V | S | <S>  
S → OEE | =VE  
O → + | - | * | /  
N → n // cualquier número  
V → v
```

2.4 Fotos



The three screenshots show the parser interface for different grammars. Each interface includes a 'Table Text Size' slider, a list of grammar rules, and a table of FIRST and FOLLOW sets.

First Screenshot (Left): Grammar rules: $E \rightarrow N$, $S \rightarrow OEE$, $O \rightarrow +$, $O \rightarrow -$, $O \rightarrow *$, $O \rightarrow /$, $N \rightarrow n$. FIRST and FOLLOW sets are shown for each non-terminal.

Second Screenshot (Middle): Grammar rules: $E \rightarrow N$, $E \rightarrow <S>$, $E \rightarrow S$, $S \rightarrow OEE$, $O \rightarrow +$, $O \rightarrow -$, $O \rightarrow *$, $N \rightarrow n$. FIRST and FOLLOW sets are shown for each non-terminal.

Third Screenshot (Bottom): Grammar rules: $E \rightarrow N$, $E \rightarrow V$, $E \rightarrow <S>$, $E \rightarrow S$, $S \rightarrow OEE$, $S \rightarrow =VE$, $O \rightarrow +$, $O \rightarrow -$, $O \rightarrow *$, $O \rightarrow /$, $N \rightarrow n$, $V \rightarrow v$. FIRST and FOLLOW sets are shown for each non-terminal.

En estas tres imágenes se aprecia que cada una de las gramáticas cumple LL(1) y además, se muestran su conjuntos FIRST y FOLLOW.

2.5 Transformaciones aplicadas

En la primera gramática solo implementamos la aceptación de números, y con la producción " $S \rightarrow OEE$ " aceptamos las operaciones más básicas entre dos números. Los paréntesis los añadimos introduciendo la regla " $E \rightarrow <S>$ " y para no obligarnos siempre a poner paréntesis también añadimos " $E \rightarrow S$ ". Por último añadimos las variables. Para la aceptación simplemente del nombre de la variable añadimos la producción " $E \rightarrow V$ ". También introducimos " $S \rightarrow =VE$ " para aceptar la asignación de variables, y por último añadimos " $V \rightarrow v$ " para tratar a las variables como no terminales.

2.6 Distinción entre niveles de léxico y sintáctico

El análisis léxico es la primera fase del procesamiento del código fuente. Se encarga de leer la secuencia de caracteres de entrada y agruparlos en unidades significativas llamadas tokens. En este caso, el analizador léxico es la función `rd_lex()`. Los objetivos del análisis léxico son:

- Identificar y clasificar los diferentes tipos de tokens en base a la gramática dada.
- Filtrar caracteres irrelevantes (espacios en blanco, tabulaciones, retornos de línea).
- Detectar y manejar errores léxicos.
- Almacenar la información de los tokens en una estructura (`s_tokens`).

Mientras que el análisis sintáctico se encarga de organizar los tokens en estructuras que cumplen con las reglas de la gramática. Este nivel verifica que los tokens estén ordenados de manera correcta según la sintaxis del lenguaje definido. Los objetivos del análisis sintáctico son construir la estructura jerárquica de las expresiones, verificar que las secuencias de tokens sean válidas según la gramática y reportar errores sintácticos si se encuentran tokens mal estructurados.

3. Resumen del diseño del parser

Se ha implementado un parser descendente recursivo, donde cada regla de la gramática se traduce en una función recursiva que maneja su sintaxis correspondiente. La función principal encargada de procesar la expresión es *ParseExpresion()*, que corresponde a la producción E de la gramática.

La producción $E \rightarrow N \mid V \mid S \mid (S)$, se corresponde con la función **ParseExpression()**, que lo que hace es evaluar el token actual y aplica un caso u otro dependiendo del token.

3.1 ParseExpression().

- **Caso N (Número):**

Si el token es un número (T_NUMBER), se imprime el valor y se consume el token. Esto corresponde a la producción $N \rightarrow n$.

C/C++

```
if (tokens.token == T_NUMBER) {  
    printf("%d", tokens.number);  
    MatchSymbol(T_NUMBER);  
}
```

- **Caso V (Variable):**

Si el token es una variable ($T_VARIABLE$), se imprime el nombre de la variable y se consume el token. Este bloque implementa la producción $V \rightarrow v$.

C/C++

```
else if (tokens.token==T_VARIABLE) {  
    printf("%s", tokens.variable_name);  
    MatchSymbol(T_VARIABLE);  
}
```

- **Caso O (Operador o Asignación):**

Si el token es un operador ($T_OPERATOR$) o el símbolo de asignación ($=$), se entra en el bloque que procesa expresiones de la forma OEE o $=VE$.

Se traduce la producción $S \rightarrow OEE$ (para operadores) y $S \rightarrow =VE$ (para asignaciones). Se destaca que se comprueba que el operador tenga dos operandos y se realizan llamadas recursivas a `ParseExpression()` para cada uno.

```
C/C++
else if (tokens.token == T_OPERATOR || tokens.token ==
'=' ) {
    char op = tokens.token_val;
    int asignacion = (tokens.token == '=');
    MatchSymbol(tokens.token); //Consumimos el operador

    //Un operador debe tener dos operandos
    printf("(");
    ParseExpression(); //Primer operando

    if (asignacion) {
        printf(" = ");
    } else {
        printf(" %c ", op);
    }

    //Validamos que haya un segundo operando
    if (tokens.token == ')' || tokens.token == '\n') {
        rd_syntax_error(0, tokens.token, "Falta un operando
en la expresión\n");
    }

    ParseExpression(); //Segundo operando
    printf(")");
}
```

- **Caso con paréntesis:**

Si el token es '(', se asume la forma (S). Se consume el paréntesis de apertura, se verifica que no existan paréntesis redundantes y se procesa la expresión interna. Este bloque se corresponde con la producción $E \rightarrow (S)$.

```
C/C++
else if (tokens.token == '(') {
    MatchSymbol('(');
    //Verificamos que después de un '(' no haya otro '('
    if (tokens.token == '(') {
        rd_syntax_error(0, tokens.token, "Paréntesis
redundantes detectados\n");
    }
}
```



```
}  
ParseExpression();  
MatchSymbol(')');  
}
```

Si ninguno de estos casos coincide, se lanza un error de sintaxis.

```
C/C++  
else {  
    rd_syntax_error(0, tokens.token, "Token inesperado en la  
    expresión: %d\n");  
}
```

3.2 ParseAxiom() y ParseYourGrammar().

Esta función representa la entrada completa (axioma) y se encarga de iniciar el proceso de parsing y validar que la expresión finalice correctamente con un salto de línea. En caso contrario, se lanza un error de sintaxis.

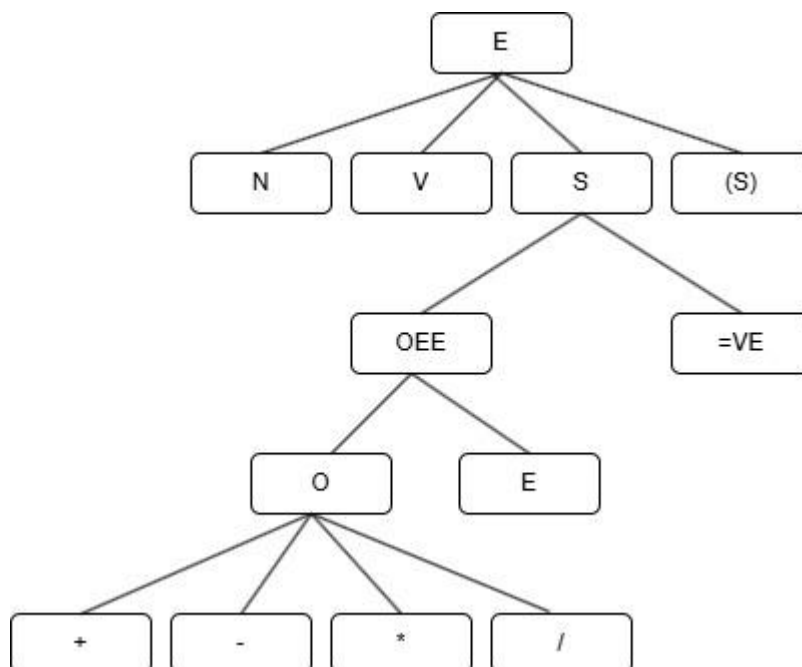
Se llama a *ParseYourGrammar()*, que simplemente invoca *ParseExpression()*, asegurando que la entrada se procese según las producciones definidas.

```
C/C++  
  
void ParseYourGrammar (void)  
{  
    ParseExpression();  
}  
  
void ParseAxiom (void)  
{  
    ParseYourGrammar ();  
}
```

```
printf("\n");

if (tokens.token == '\n') {
    MatchSymbol ('\n');
}
else {
    rd_syntax_error (-1, tokens.token, "-- Token
    inesperado (Expected:%d=None, Read:%d) al final del
    Parsing\n") ;
}
}
```

4. Diagrama sintáctico de la gramática propuesta



5. Batería de pruebas

Notación prefija	Resultado esperado	Resultado obtenido
(+ 2 3)	(2 + 3)	(2 + 3)
(- X Y)	(X + Y)	(X + Y)
(= A 5)	(A = 5)	(A = 5)
(= X (+ 6 7))	(X = (6 + 7))	(X = (6 + 7))
(= E (= F (= G (+ H I))))	(E = (F = (G = (H + I))))	(E = (F = (G = (H + I))))
(* (= N 7) (= O 9))	((N = 7) * (O = 9))	((N = 7) * (O = 9))
(+ (+ (= R 2) (= S 3)) (+ (= T 6) (= U 7)))	(((R = 2) + (S = 3)) + ((T = 6) + (U = 7)))	(((R = 2) + (S = 3)) + ((T = 6) + (U = 7)))
(= V (+ (= W (- (= X 2) 5)) (* (= Y 3) 8)))	(V = ((W = ((X = 2) - 5)) + ((Y = 3) * 8)))	V = ((W = ((X = 2) - 5)) + ((Y = 3) * 8)))
(+ (* 23) (/ 45))	ERROR	ERROR
(+ (+ (+ A B) C) D)	(((A + B) + C) + D)	(((A + B) + C) + D)
(* (= P 1) (= Q (= R (= S 5))))	((P = 1) * (Q = (R = (S = 5))))	((P = 1) * (Q = (R = (S = 5))))
(/ (/ (/ 8 9) 1) 2)	(((8 / 9) / 1) / 2)	(((8 / 9) / 1) / 2)
(= Z (= A (= B (= C 9))))	(Z = (A = (B = (C = 9))))	(Z = (A = (B = (C = 9))))
(+ (= D 1) (+ (= E 2) (+ (= F 3) (+ (= G 4) 5))))	((D = 1) + ((E = 2) + ((F = 3) + ((G = 4) + 5))))	((D = 1) + ((E = 2) + ((F = 3) + ((G = 4) + 5))))
(* (= H 6) (* (= I 7) (* (= J 8) (* (= K 9) 2))))	((H = 6) * ((I = 7) * ((J = 8) * ((K = 9) * 2))))	((H = 6) * ((I = 7) * ((J = 8) * ((K = 9) * 2))))
((* 3 2))	ERROR	ERROR
(= a1 (= b (+ c3 (+ 2 3))))	(a1 = (b = (c3 + (2 + 3))))	(a1 = (b = (c3 + (2 + 3))))