Práctica Parser Descendente Recursivo

Procesadores del lenguaje

Marzo 2023

Grado en Ingeniería Informática

Pablo Hidalgo Delgado. NIA: 100451225
César López Navarro. NIA: 100451326

1. Diseñad una gramática que represente las expresiones aritméticas anteriormente definidas.

Gramática 1

Axioma → Expresión \n

Expresión → (Operador Parámetro Parámetro) |

Número

Parámetro → Expresión

Número → Token_número

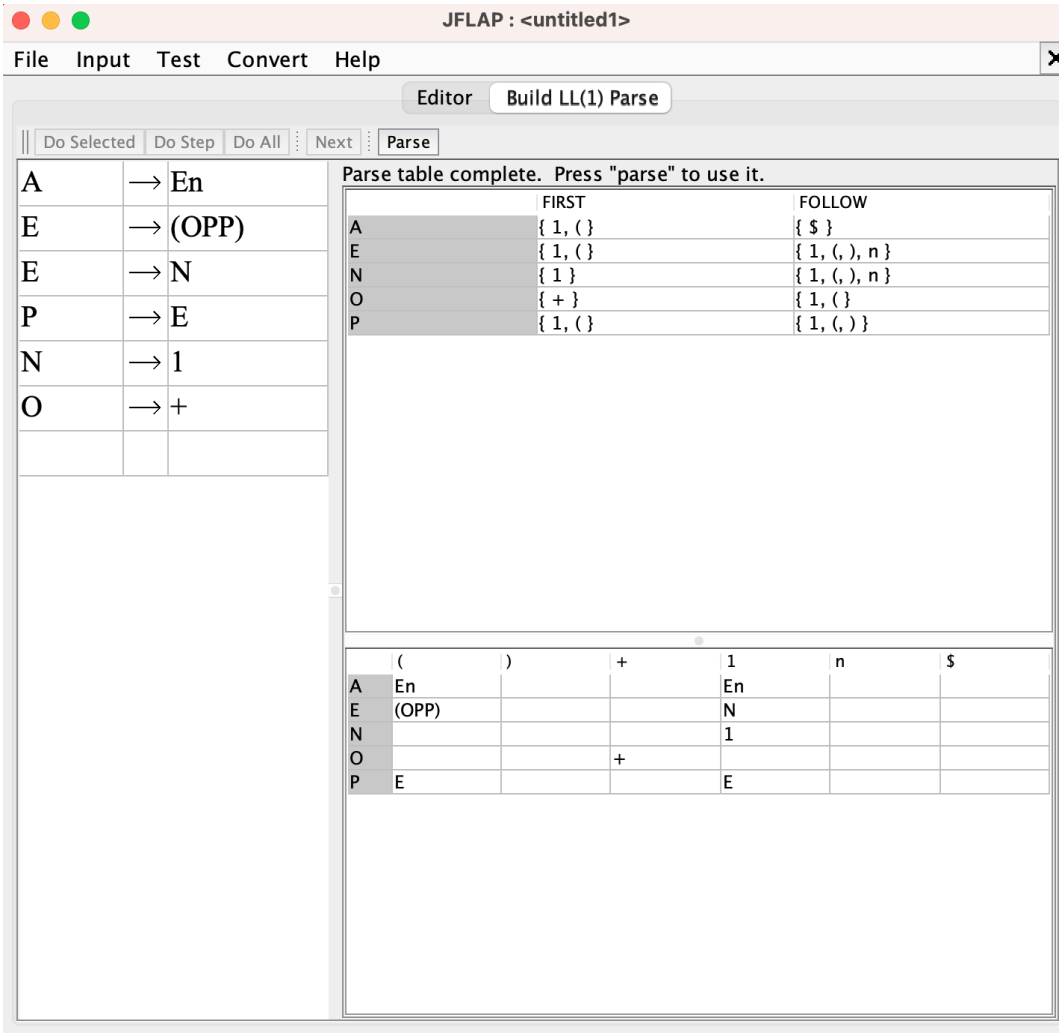
Operador → Token_operador

Esta sería la gramática que representa expresiones aritméticas en notación prefijo.

Cabe mencionar que el símbolo \n representa un salto de línea y no es necesario incluirlo en el código debido al while(1) implementado en el código. Sin embargo, hemos decidido incluirlo en la gramática ya que nos ayudaba a visualizarla con más claridad.

2. Determinad si es necesario transformar la gramática anterior para que cumpla con las condiciones LL(1). Utilizad la herramienta jflap.

No es necesario ya que ya cumple con las condiciones LL(1). Esto lo comprobamos usando JFLAP (no nos da ningún error al seleccionar *Build LL(1) parse table*). Hemos tratado de diseñar la gramática para que cumpla con estas condiciones.



A	⟶	En
E	⟶	(OPP)
E	⟶	N
P	⟶	E
N	⟶	1
O	⟶	+

	FIRST	FOLLOW
A	{ 1, (}	{ $ }
E	{ 1, (}	{ 1, (,), n }
N	{ 1 }	{ 1, (,), n }
O	{ + }	{ 1, (}
P	{ 1, (}	{ 1, (,) }

	(	)	+	1	n	$
A	En			En		
E	(OPP)			N		
N				1		
O			+			
P	E			E		

Imagen1: Conjuntos PRIMERO y SIGUIENTE de la Gramática1

Al no poder representar tokens en JFLAP, utilizamos el terminal ‘1’ para representar cualquier Token_número y el terminal ‘+’ para representar cualquier Token_operador.

3. Desarrollad un Parser Descendente Recursivo conforme a la gramática del punto 2 para procesar y evaluar expresiones en notación prefija.

Definimos los tokens operador y número y llamamos al analizador léxico rd_lex cada vez que queramos leer una entrada. Esta llamada la hacemos únicamente antes de llamar a la función principal ParseAxiom() y al final de MatchSymbol(). Esta última función verifica que el token leído sea el esperado. La utilizamos a la hora implementar las funciones Parse de los no terminales que producen tokens.

La idea del parser descendente recursivo es ir llamando a las funciones Parse*NoTerminal* según estén a la derecha de la regla de producción. Por ejemplo,

con la regla Parámetro → Expresión, dentro de la función ParseParámetro() llamamos a ParseExpresion. Esto es justo lo que hacemos en el código. Además, en cada una de las funciones Parse, creamos una variable val de tipo entero para almacenar el valor y lo devolvemos.

La función ParseExpression() es la que lee los parámetros y el operador, por lo que en este caso guardamos estos valores también en variables locales. En esta función debemos comprobar si el símbolo leído es un número o un paréntesis, símbolos pertenecientes al conjunto PRIMERO de Expresión. Este conjunto lo hemos calculado utilizando JFLAP como se puede ver en la *imagen1*. Además, también en esta función realizamos las operaciones aritméticas correspondientes y devolvemos el resultado.

Hemos de mencionar también que al comienzo del código definimos el prototipo de la función ParseExpression() para poder llamarla antes de ser implementada.

4. Ampliad la gramática para manejar variables simples (un carácter, mayúsculas o minúsculas).
    a. Una Variable estará representada por un *Token* específico, y podrán intervenir en una expresión, al igual que un Numero.
    b. Se puede asignar a una Variable el valor de un Numero o de una Expresion mediante la expresión (! Variable Parametro). En este caso, el Operador ! almacena el valor del último Parametro en la Variable especificada.
    c. La entrada A \n debe generar el valor de la variable A.

La gramática inicial que diseñamos para este apartado fue la siguiente:

Axioma → Expresión \n

Expresión → (Operador Parámetro Parámetro) |

(! Variable Parámetro) |

Número |

Variable

Parámetro → (Operador Parámetro Parámetro) | Número | Variable

Número → Token_número

Operador → Token_operador

Variable → Token_variable

No obstante, al introducirla en JFLAP nos dimos cuenta que no cumplía con las condiciones de una gramática LL(1), pues contiene 2 producciones que empiezan por el mismo símbolo. La transformamos a LL(1) factorizando las producciones que tienen el mismo comienzo. La gramática resultante es la siguiente:

Gramática 2

Axioma → Expresión \n

Expresión → (RestoExpresión |

Número |

Variable

RestoExpresión → Operador Parámetro Parámetro) |

! Variable Parámetro)

Parámetro → (Operador Parámetro Parámetro) |

Variable |

Número

Número → Token_número

Operador → Token_operador

Variable → Token_variable

Cabe destacar que en la regla de producción Parámetro → Variable, la variable debe de estar inicializada. Esto lo controlaremos implementando código de C.

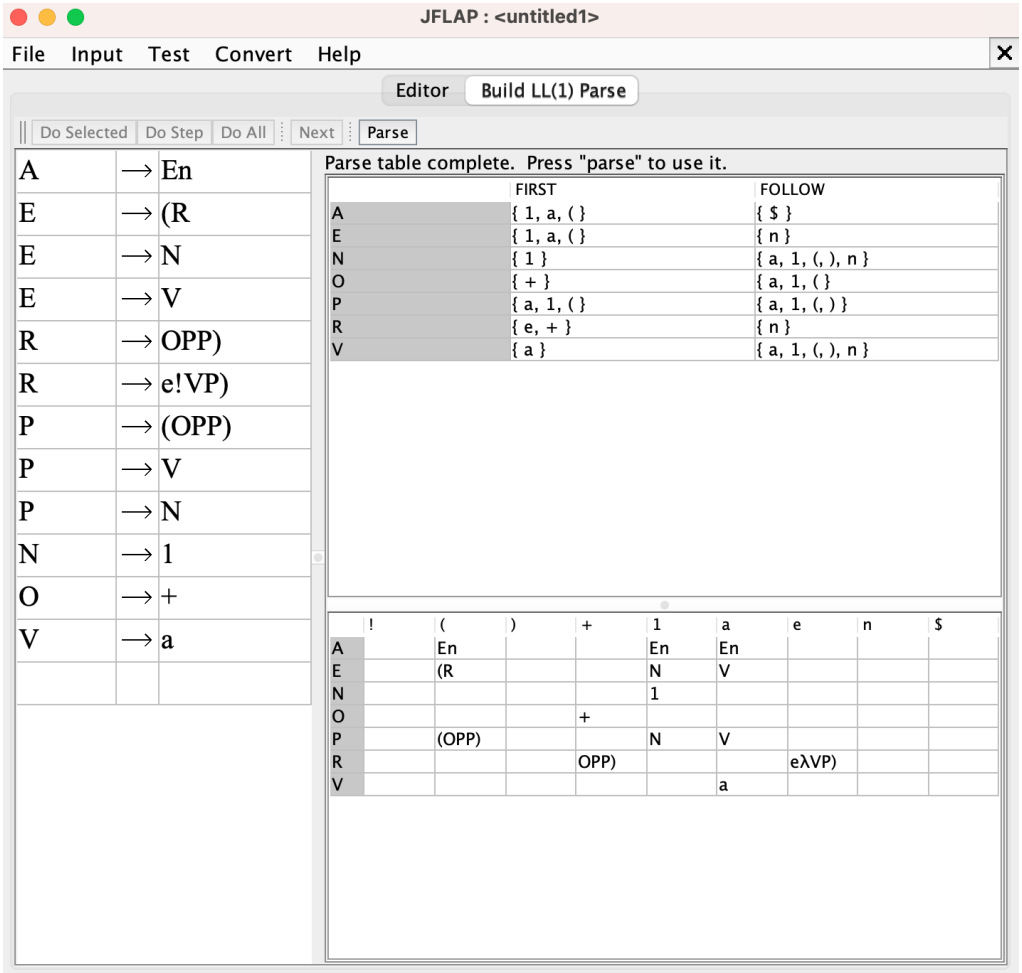


Imagen2: Conjuntos PRIMERO y SIGUIENTE de la Gramática2

5. Ampliad la gramática para permitir dos o más Parámetros en las Expresiones. Por ejemplo, (- 1 1 1 1) debe generar -2 en la salida.

Axioma → Expresión \n

Expresión → (RestoExpresión |

Número |

Variable

RestoExpresión → Operador Parámetro Parámetro MásParámetros) |

! Variable Parámetro)

Parámetro → (Operador Parámetro Parámetro MásParámetros) |

Variable |

Número

MásParámetros → Parámetro MásParámetros | lambda

Número → Token_número

Operador → Token_operador

Variable → Token_variable

Al intentar implementar esta gramática en código C, nos damos cuenta de que, en la producción MásParámetros → Parámetro MásParámetros, necesitaremos sustituir el no terminal Parámetro por sus valores. Por lo tanto, el diseño de nuestra gramática resulta en:

Gramática 3

Axioma → Expresión \n

Expresión → (RestoExpresión |

Número |

Variable

RestoExpresión → Operador Parámetro Parámetro MásParámetros) |

! Variable Parámetro)

Parámetro → (Operador Parámetro Parámetro MásParámetros) |

Variable |

Número

MásParámetros→(Operador Parámetro Parámetro MásParámetros)MásParámetros|

Variable MásParámetros |

Número MásParámetros |

lambda

Número → Token_número

Operador → Token_operador

Variable → Token_variable

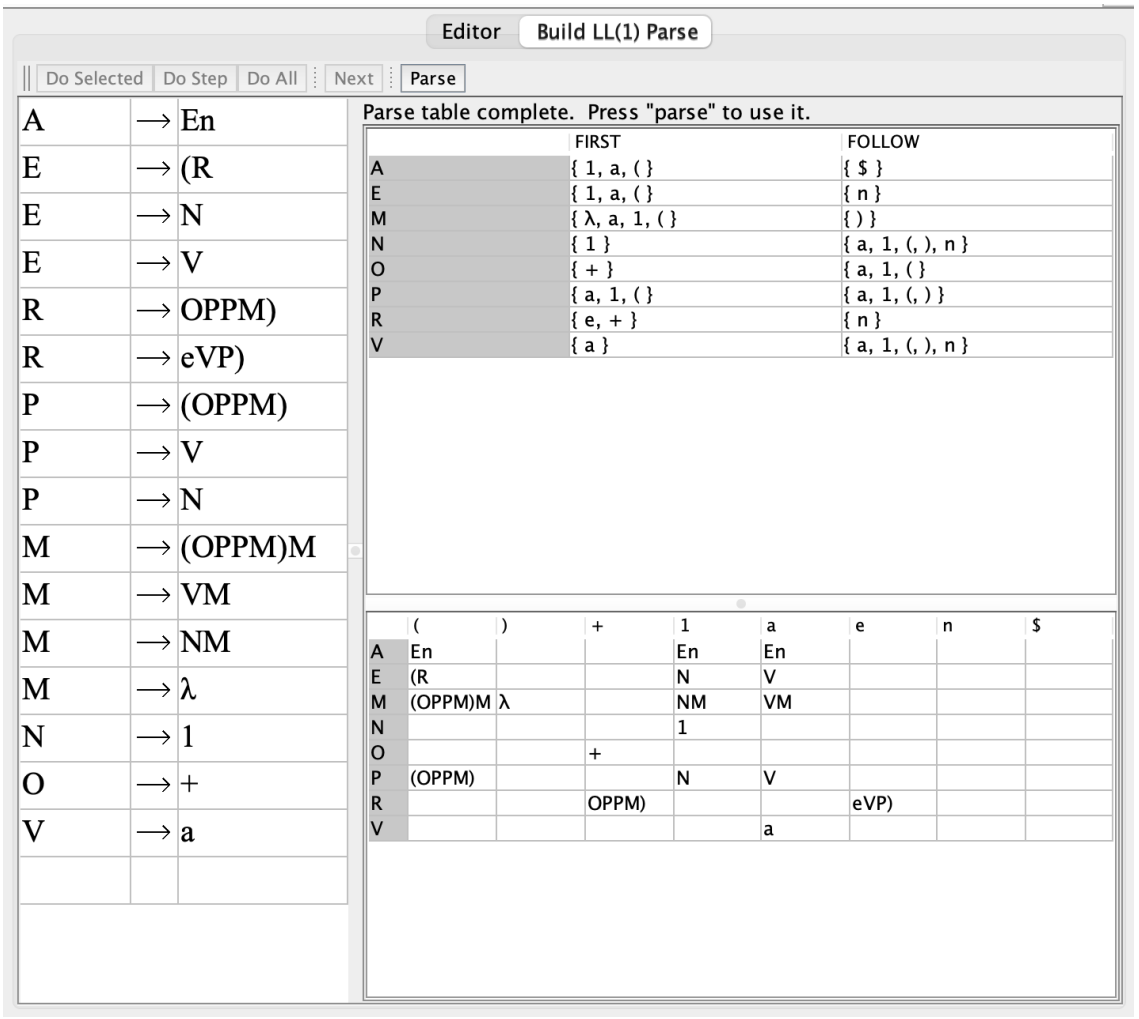


Imagen3: Conjuntos PRIMERO y SIGUIENTE de la Gramática3

6. Repetid los pasos 1-3 para Ías ampliaciones propuestas en 4 y 5.

<u>Ampliaciones del punto 4:</u>

Añadimos un token para guardar variables. También implementamos una array de enteros de longitud 52 donde se guardarán los valores de las variables inicializadas. Cada posición del array se corresponde con un carácter/variable. Del 0 al 26 se almacenarán los valores para los carácteres en mayúscula y del 26 al 52 para los carácteres en minúscula. Inicializamos todos los valores de este array a -1 (ya que la gramática no acepta valores negativos y por tanto no se pueden asignar a variables). Así, podremos saber si una variable ha sido inicializada o no (función *variable_is_initialized()*). Esta función la usamos, como hemos mencionado anteriormente, cada vez que la variable sea un parámetro (en ParseExpression() y en ParseParametro()) y no una declaración de variable.

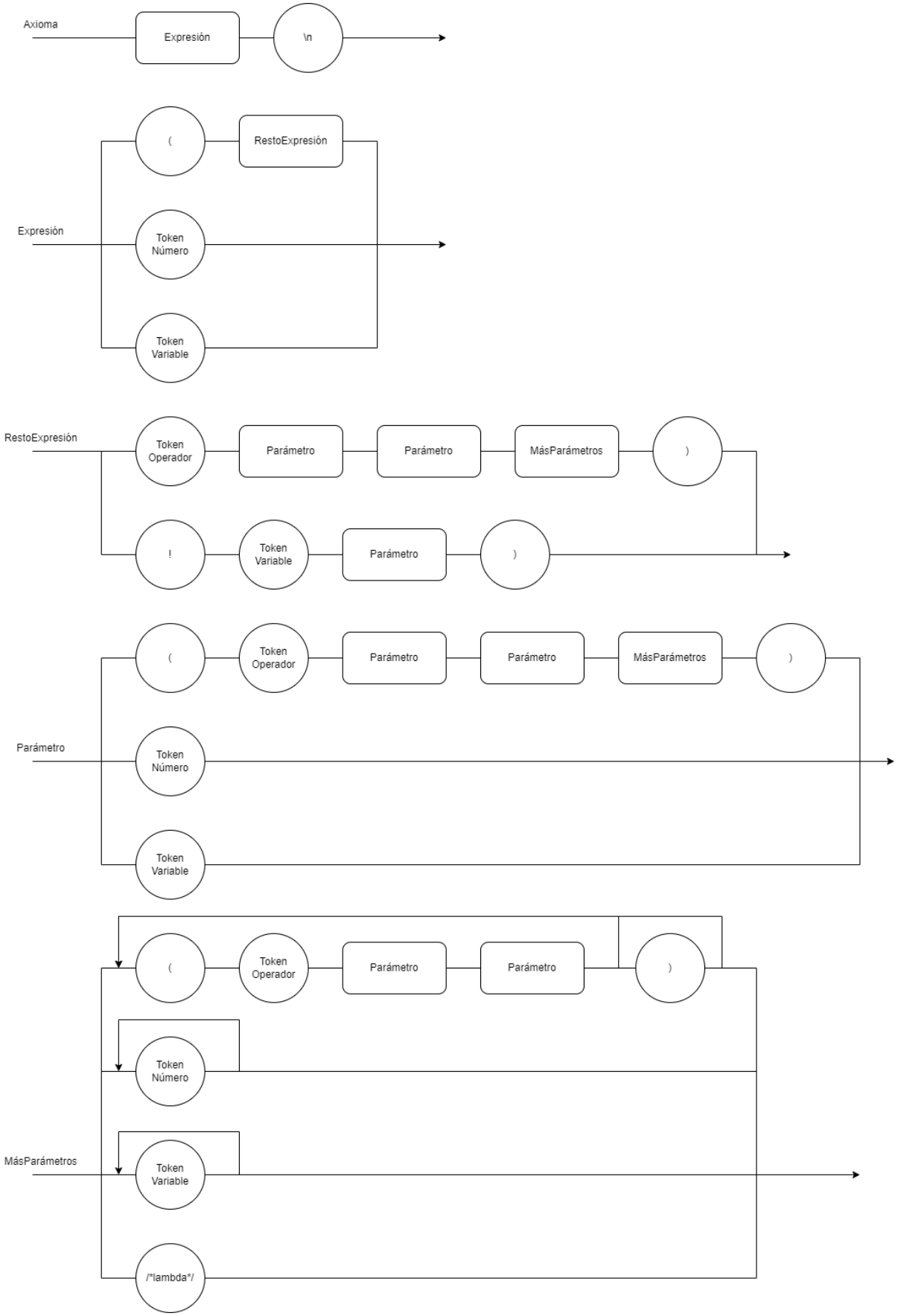
Aunque es cierto que debemos añadir las nuevas producciones introducidas (ParseRestoExpresión() y ParseVariable()), básicamente tanto estas funciones como el resto del código sigue la misma idea que lo explicado anteriormente. Llamamos a las funciones Parse que correspondan, devolvemos los valores correspondientes y chequeamos los conjuntos PRIMERO cuando es necesario.

Ampliaciones del punto 5:

Añadimos las funciones Parse correspondientes a las nuevas producciones (ParseMasParametros()). Esta función será recursiva. Además, aquí chequeamos el conjunto FOLLOW ya que esta producción deriva en lambda. También estructuramos el código (if, else if y else) según los símbolos leídos pertenecientes a su conjunto PRIMERO al igual que hemos hecho hasta el momento. Sin embargo, en esta ocasión, llegados a este punto, llamamos de nuevo a la función de forma recursiva y realizamos la operación. Así, estaremos leyendo y realizando la operación correspondiente con todos los parámetros introducidos.

Con esta idea, debemos añadir que, a la hora de realizar la operación entre los parámetros en ParseRestoExpression() y ParseParametro(), debemos también incluir la operación del parámetro devuelto por ParseMasParámetros() cuando no estemos ante una variable o número (indica que ya no hay más parámetros). Esto lo hacemos guardando el valor devuelto por esta función en param3 y realizando la operación correspondiente con el valor previamente calculado entre los 2 parámetros anteriores.

7. Diagrama sintáctico de la gramática



8. Pruebas

El fichero drLL.txt contiene un conjunto de 82 casos de prueba para probar que nuestro parser descendente recursivo funciona correctamente.

En las primeras 7 líneas comprobamos que el resultado de las distintas operaciones aritméticas con 2 parámetros funciona como esperábamos. Un ejemplo es (línea 4) : (/ 4 4). Se imprime por pantalla → Valor 1.

De las líneas 8 a la 12 se imprimen directamente los números. Por ejemplo (línea 10), al escribir 13454 el resultado que nos da el parser es → Valor 13454.

De las líneas 13 a la 44 tenemos operaciones en las que intervienen dos parámetros pero, en este caso, uno va a ser un número y el otro una operación. También observamos que el valor obtenido cumple con lo esperado. Ejemplo: (- (* 3 4) 4) → Valor obtenido: 8

De las líneas 45 a la 60 tenemos de nuevo operaciones en las que intervienen dos parámetros pero, esta vez, ambos son operaciones con dos parámetros respectivamente. Nuevamente los valores se corresponden. Ejemplo: (* (+ 3 4) (+ 3 4))

→ Valor obtenido: 49

De las líneas 63 a la 74 tenemos casos de pruebas relacionadas con variables. De estas pruebas evaluamos satisfactoriamente que una variable puede ser declarada (tanto caracteres en mayúscula como en minúscula) , el valor de la variable se guarda y por tanto puede ser utilizada en otra expresión, y podemos asignar valores a una variable que estén guardados en otra variable o provengan de una operación. Por ejemplo:

(! C 2) → Valor 2

C → Valor 2

(+ C 3) → Valor 5

(! C (+ 1 1)) → Valor 4

C → Valor 4

(! h 10) → Valor 10

h → Valor 10

(+ C h) → Valor 14

C → Valor 4

h → Valor 10

(! C h) → Valor 10

C → Valor 10

h → Valor 10

Es importante tener en cuenta que, según el diseño de nuestro parser, al definir una variable, se imprime por pantalla el valor adquirido por la variable. Por ejemplo, (línea 62) (! C 2) → Valor 2.

Por último, de las líneas 75 a 82 probamos con éxito los casos en los que existen más de 2 parámetros. Ejemplo:

(* 2 2 2 2 2 2 2 2 2 2) → Valor 1024 ,

(- 1 1 1 1) → Valor -2 ,

(- (* 3 4 2) (+ 3 4 2) 1 8) → Valor 6