Universidad Carlos III
Procesadores del lenguaje 2023-24
Práctica 1: drLL

Curso 2023-24

GRUPO: **85**

Nº Grupo: **93**

Identificador de grupo: **93-gallego**

Members:

Lucas Gallego Bravo (100429005)

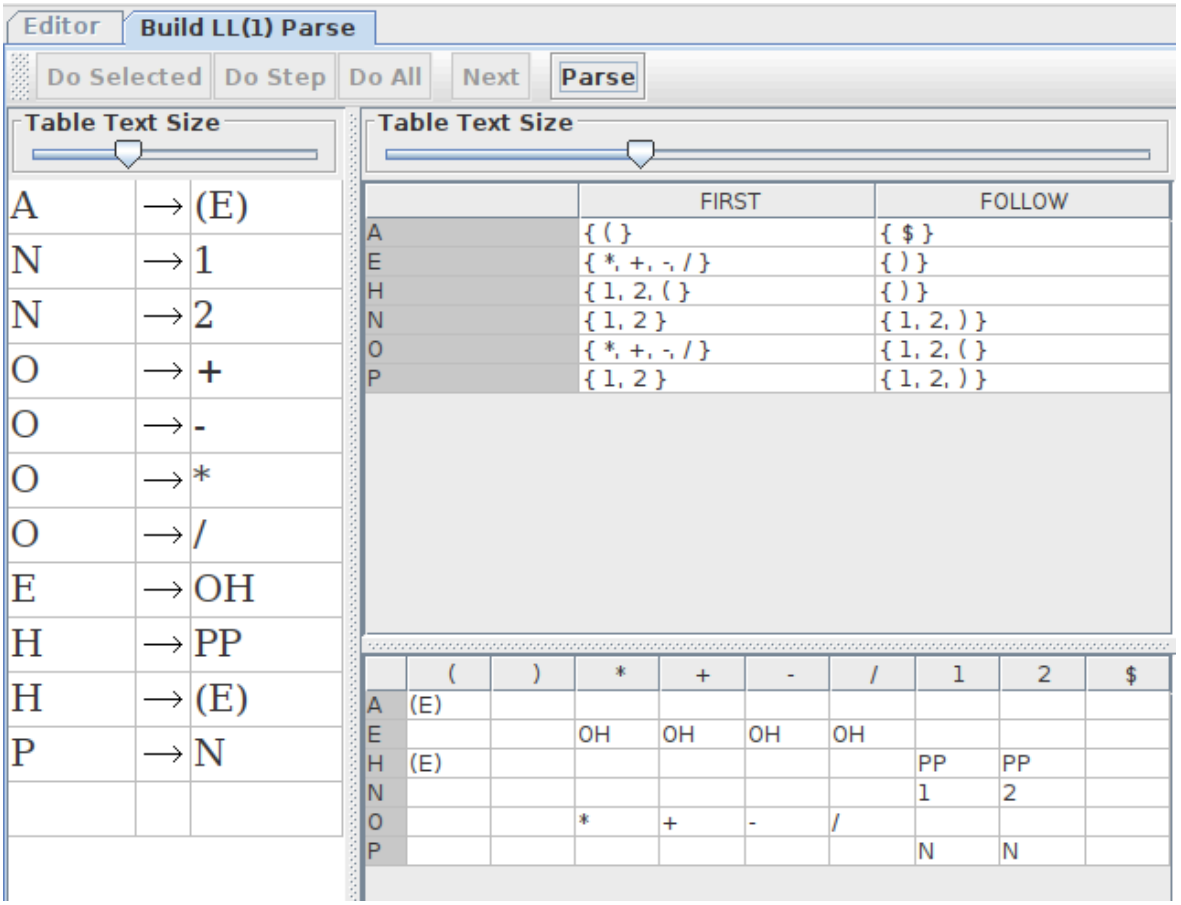
Emails: 100429005@alumnos.uc3m.es

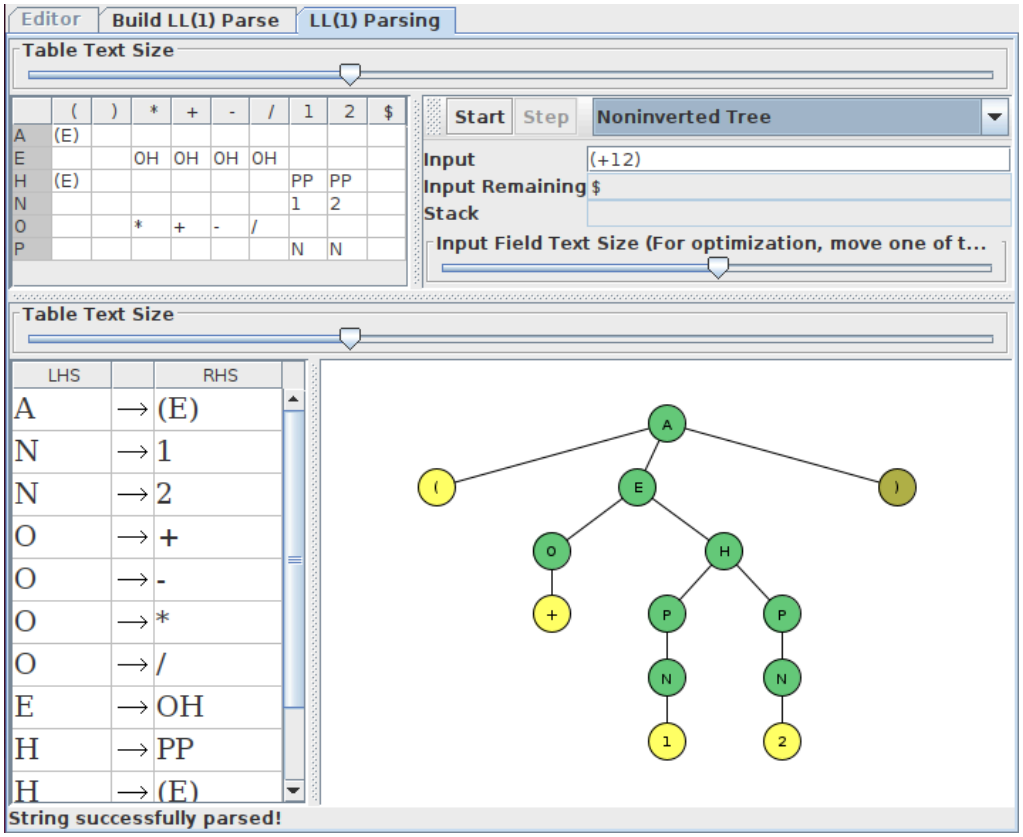
Primera gramática creada para afrontar el problema:

Tal y como se indica en el enunciado de la práctica, lo primero que se ha realizado es la creación de una gramática sencilla a partir de la cual se derivará y evolucionará hasta obtener una gramática final que satisfaga las necesidades especificadas en esta práctica.

A ::= (E) E ::= O H H ::= P P | (E) P ::= N O ::= + | - | * | /	A -> Axioma E -> Expresión O -> Operador H -> Operando P -> Parámetro N -> Número

Resultados JFLAP:





Como se puede observar, esta es una primera aproximación bastante simplificada, ya que solo se tiene en cuenta que el axioma se convierta en una expresión, y ya sabemos por el enunciado que tendremos que ser capaces de traducir tanto expresiones como números, variables e incluso asignación de valores a variables.

Cabe destacar, que en la última captura de pantalla se le han proporcionado algunos valores terminales al nodo **'N'**, esto se ha decidido hacer para que todo funcione de manera correcta.

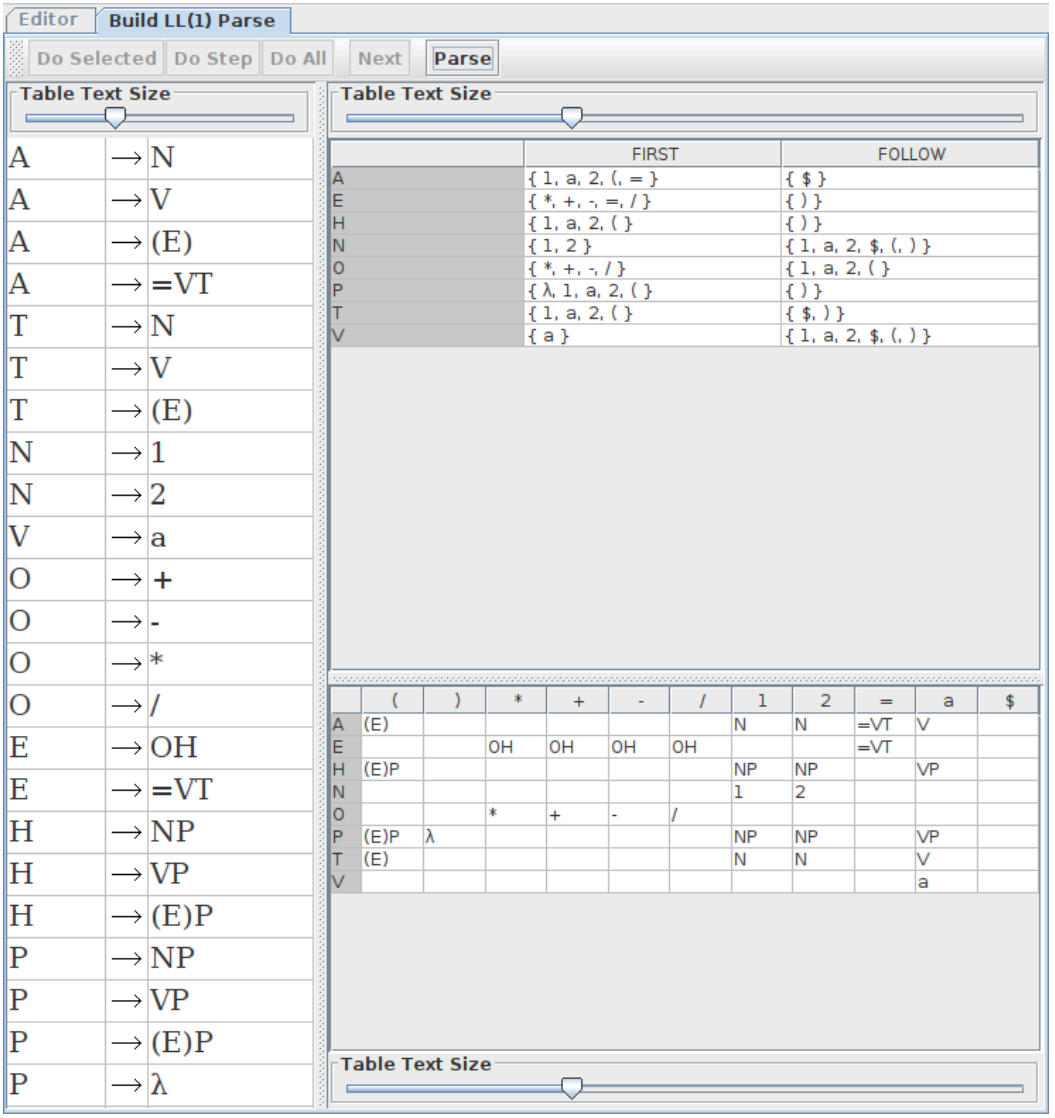
Gramática final:

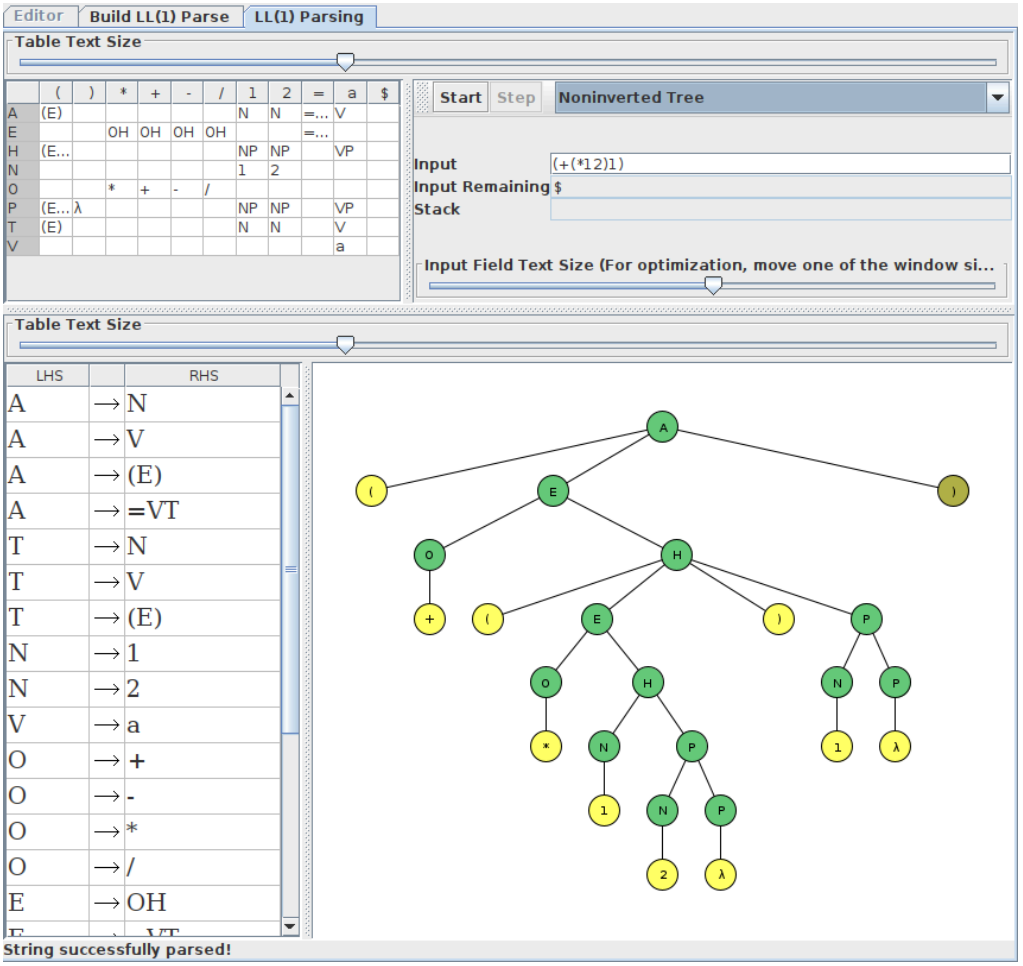
Tras realizar diferentes modificaciones a la primera gramática mencionada anteriormente, para que esta gramática final sea capaz tanto de reconocer el uso de variables como de números, y ser capaz de reconocer notación prefija alternativa para su correcta traducción postfija, se ha conseguido llegar a la siguiente gramática:

A ::= N | (E) | V | ! V T E ::= O H | ! V T H ::= (E) P | N P | V P P::= V P | (E) P | N P | λ T ::= N | (E) | V O::= + | - | * | /	A = Axiom E = Expression H = OpExpression T = Type P = Parameter O = Operator N = Number V = Variable

Para esta gramática se han realizado algunas asumpciones viendo el pdf del laboratorio, una de estas asumpciones es que se pueden declarar variables aun si no están entre paréntesis, es decir, que es posible realizar ***'= A 15'*** sin que se encuentre entre paréntesis.

Resultados JFLAP:





El ejemplo que se ha utilizado para evaluar la gramática en JFLAP es ***'(+ (* 1 2) 1)'***, esto se ha decidido para comprobar utilizando esta herramienta si la gramática es capaz de reconocer la notación alternativa prefija.

Otra cosa que cabe destacar a la hora de utilizar la función **'Parse'** de JFLAP es que no hay que utilizar espacios en blanco en las cadenas que se introduzcan como input ya que no los reconoce y puede llevar a error.

Resumen del diseño del código:

ParseEqual(): Esta función únicamente comprueba que el operador introducido es el usado para declarar una variable **('=').**

ParseNumber(): En esta función se llega a comprobar si el valor es esperado, es decir, es un número con la invocación de **MatchSymbol(T_NUMBER)**. Finalmente mediante el uso de un return, se devolverá el número leído.

ParseVariable(): En esta función se obtiene el número ascii de la variable que se ha introducido, una vez se ha obtenido, ese valor se devuelve mediante un **'return'** para su

posterior impresión. Dentro de esta función se hace uso de ***'rd_lex()'*** para pasar al siguiente elemento.

ParseOperator(): Esta función tiene la función de comprobar que el operador introducido es el operador esperado, de ser así se devuelve el valor de dicho operador. Esta comprobación mencionada anteriormente se realiza con la llamada a función **MatchSymbol (T_OPERATOR)**.

ParseParameter(): Esta función tiene como objetivo ser la continuación de la función ***ParseOpExpression()***. Tal y como indica la gramática, esta función tiene 4 posibles derivaciones, λ, un número, una variable o una expresión a la cual sigue esta función de nuevo. Esto es debido a que una expresión debe ser al menos 2 números o variables, pero tal y como se indica en los apartados opcionales, se pueden tener expresiones de más de 2 parámetros, al mismo tiempo, se pueden encadenar 2 expresiones de las cuales se sume su resultado. Debido a esto, es necesaria una función como ***ParseParameter()*** la cual nos ayude en estas situaciones. En caso de que la función lea un número se llamará a la función ***ParseNumber()***, y tras obtener el valor del número leído, este se imprimirá en pantalla haciendo uso de un **'printf'** tras imprimirse, se llamara de nuevo a la función ***ParseParameter()***. En el caso de que sea una variable se llamará a la función ***ParseVariable()***, y una vez obtenido el valor ascii de la variable introducida se pasará a su impresión en pantalla mediante un **'printf'**, poniendo detrás de la variable leída un **@**, tras imprimirse, se llamara de nuevo a la función ***ParseParameter()***. En el caso de que lea una expresión, se llamará primero a la función ***ParseLParen ()***, para posteriormente llamar a la función ***ParseExpression ()*** lo siguiente es una llamada a ***ParseRParen ()*** finalizando con una llamada a ***ParseParameter()***. Finalmente tenemos el caso en el que la función lee ***'\n'***, en este caso la función simplemente terminará haciendo un return (simulando el funcionamiento de **λ**).

ParseType(): Esta función es una simplificación de la función anteriormente comentada ***ParseParameter()*** y únicamente se hace uso de ella en el momento en el que se quiera declarar el valor de una variable. Debido a esto, esta función únicamente puede tomar el valor de un número, de otra variable o de una expresión. Esto se ha decidido realizar de esta forma para evitar problemas que pudieran conllevar la reutilización de funciones, además de esta manera queda una gramática más limpia. En caso de que se lea un número se llamará a la función ***ParseNumber()***, y tras obtener el valor del número leído, este se imprimirá en pantalla haciendo uso de un **'printf'**. En el caso de que sea una variable se llamará a la función ***ParseVariable()***, y una vez obtenido el valor ascii de la variable introducida se pasará a su impresión en pantalla mediante un **'printf'**, poniendo detrás de la variable leída un **@**. En el caso de que lea una expresión, se llamará primero a la función ***ParseLParen ()***, para posteriormente llamar a la función ***ParseExpression ()*** finalizando con una llamada a ***ParseRParen ()***.

ParseOpExpression(): Esta función hace exactamente lo mismo que ***ParseParameter()***, con la única diferencia siendo que ***ParseOpExpression()*** no deriva en **λ.** De esta forma, si lee un número se llamará a la función ***ParseNumber()***, tras recibir su valor lo imprimirá por pantalla y posteriormente llamará a la función ***ParseParameter()***. Si lee una variable se llamará a la función ***ParseVariable()***, y una vez obtenido el valor ascii de la variable

introducida la imprimirá por pantalla para posteriormente imprimir @ detrás de la variable y posteriormente llamará a la función ***ParseParameter()***. En el caso de que lea una expresión, se llamará primero a la función ***ParseLParen ()***, para posteriormente llamar a la función ***ParseExpression ()*** lo siguiente es una llamada a ***ParseRParen ()*** finalizando con una llamada a ***ParseParameter()***.

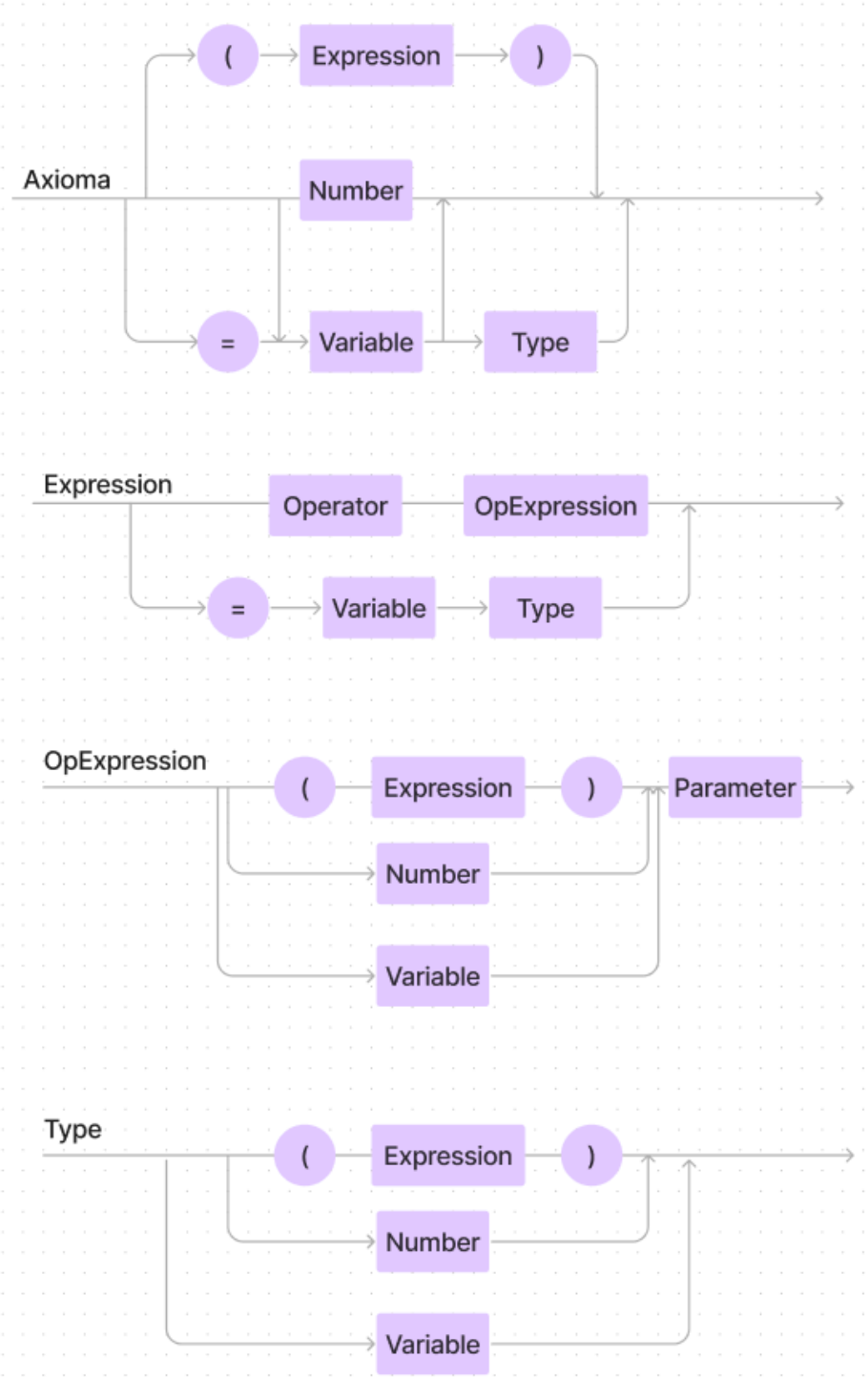
ParseExpression(): Esta es una de las funciones más simples dentro del programa ya que solo tiene dos casos posibles. Primero se comprueba si el **token** actual es un operador y en caso de que lo sea, si **'token_val'** en ese momento es igual a **'='** (operador usado para declarar el valor de una variable), se realizará primero un ***ParseEqual()*** seguido de este se llamara a la función ***ParseVariable()*** para obtener el ascii de la variable que se ha leído, seguido de esto se llamara a la función ***ParseType()***, después de llamar a esta última función se imprimirá por pantalla la variable leída, teniendo esta delante de ella la palabra **'dup'** y detrás de la variable **'!'**, quedando el siguiente print: ***"dup %c ! "***. El otro caso posible que tiene esta función es sí lee cualquier otro operador, en este caso se llamará primero a la función ***ParseOperator()*** para obtener que operador se ha introducido, después de esto se llamara a ***ParseOpExpression()***, una vez se ha llamado a esta última función, se imprimirá en pantalla el operador obtenido al principio.

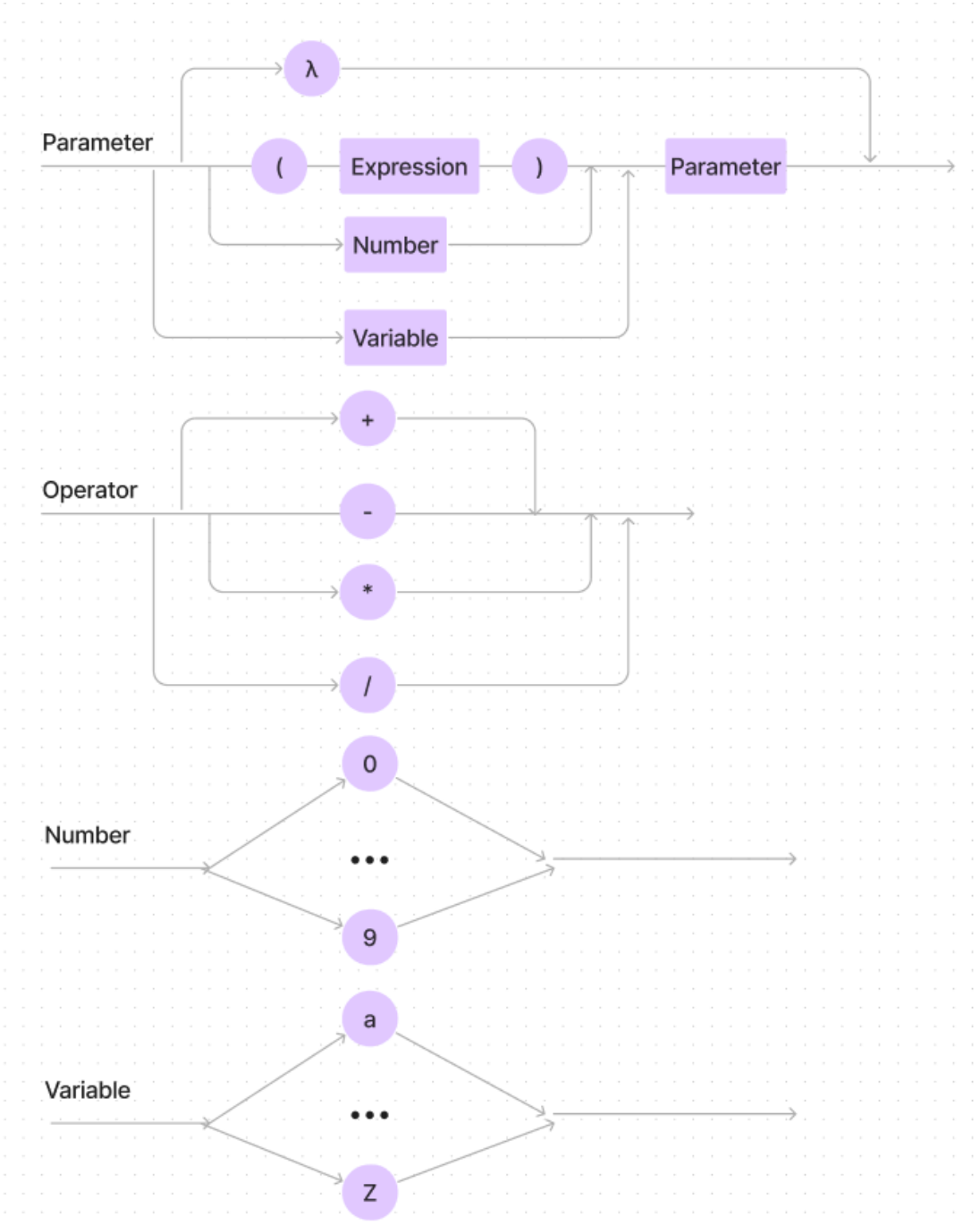
ParseAxiom(): Esta es la función a partir de la cual se desarrolla toda la gramática. Si se ha introducido un único número, se llamará a la función ***ParseNumber()*** y tras obtener su valor, lo imprimirá por pantalla poniendo detrás de dicho valor un punto. Si lee una variable se llamará a la función ***ParseVariable()***, y una vez obtenido el valor ascii de la variable introducida la imprimirá por pantalla para posteriormente imprimir @ y un punto detrás de la variable. En el caso de que lea un operador, se comprobará si es el operador de asignación de variable **'='**, en caso de serlo, se realizará primero un ***ParseEqual()*** seguido de este se llamara a la función ***ParseVariable()*** para obtener el ascii de la variable que se ha leído, seguido de esto se llamara a la función ***ParseType()***, después de llamar a esta última función se imprimirá por pantalla la variable leída, teniendo esta delante de ella la palabra **'dup'** y detrás de la variable **'!'** seguido de un punto, quedando así el siguiente print: ***"dup %c ! ."***. En caso de que no sea un número, una variable o un **'='** deberá ser una expresión, por lo que se realizará un ***ParseLParen ()***, para posteriormente llamar a la función ***ParseExpression ()*** a la que sigue una llamada a ***ParseRParen ()*** finalizando con la impresión de un punto.

main(): Esta es la función que inicializa todo el programa, en ella primero se declara una variable ***'flagMultiple'***, la cual tiene la función de permitir la introducción de más de un comando. Posteriormente se imprime por pantalla todas las variables utilizadas en los test case además de algunas extra, de esta forma al hacer la pipeline de gforth no nos devolverán error las variables. A continuación nos encontramos con un if, el cual comprueba que en el caso en el que **'argc'** sea mayor o igual a 2, se comprobará si en la posición 1 de **'argv'** se encuentra **'-s'**, de ser así solo se podrá introducir un comando a la hora de ejecutar el código, ya que el valor de ***'flagMultiple'*** pasa a 0. Posteriormente se crea un bucle do while, en el cual la condición es la variable ***'flagMultiple'***, dentro de este bucle se llama a la función ***rd_lex()***, posteriormente se comprueba si se ha llegado al final de ***drLL.txt*** (token = -1), de ser así se romperá el bucle. Si no, se llama a ***ParseAxiom()***,

después se imprime un **“\n”**, y se hace un fflush de stdout. Fuera del bucle se imprime **‘bye’**, lo cual es necesario para el pipeline de gforth.

Diagrama sintáctico representando la gramática diseñada:





TEST CASES:

Para ejecutar nuestros test cases es necesario escribir en terminal los siguientes comandos:

gcc drLL.c -o drLL
./drLL <drLL.txt

Si se ha instalado gforth en ubuntu, se puede usar la siguiente pipeline para comprobar los resultados:

./drLL <drLL.txt | gforth

El código se ha probado usando ambas ejecuciones, aquí los resultados:

INPUT	***OUTPUT***	***OUTPUT GFORTH***
123	123 .	123 . 123 ok
= A (* 2 3)	2 3 * dup A ! .	2 3 * dup A ! . 6 ok
A	A @ .	A @ . 6 ok
(+ A 123)	A @ 123 + .	A @ 123 + . 129 ok
(= B (* (+ 123 12) 1))	123 12 + 1 * dup B ! .	123 12 + 1 * dup B ! . 135 ok
(= A (* 2 B))	2 B @ * dup A ! .	2 B @ * dup A ! . 270 ok
(* (+ 1 2) 3)	1 2 + 3 * .	1 2 + 3 * . 9 ok
(+ 1 (* 2 3))	1 2 3 * + .	1 2 3 * + . 7 ok
= A (- (+ 3 1) (/ 6 2))	3 1 + 6 2 / - dup A ! .	3 1 + 6 2 / - dup A ! . 1 ok
(+ A B)	A @ B @ + .	A @ B @ + . 136 ok
= B (* 4 5)	4 5 * dup B ! .	4 5 * dup B ! . 20 ok
= C (/ 15 3)	15 3 / dup C ! .	15 3 / dup C ! . 5 ok
= d (- 22 21)	22 21 - dup d ! .	22 21 - dup d ! . 1 ok
(- (* A d A) (+ B C A) A B C d)	A @ d @ A @ * B @ C @ A @ + A @ B @ C @ d @ - .	A @ d @ A @ * B @ C @ A @ + A @ B @ C @ d @ - . 4 ok
(- (* 8 9) (+ 5 6) 5 7 8)	8 9 * 5 6 + 5 7 8 - .	8 9 * 5 6 + 5 7 8 - . -1 ok
(* (/ 8 4) (- 8 6) (* 3 2) (+ 8 9))	8 4 / 8 6 - 3 2 * 8 9 + * .	8 4 / 8 6 - 3 2 * 8 9 + * . 102 ok
(- (- 5 9) (- 8 9) (- 1 1 1 1) 9)	5 9 - 8 9 - 1 1 1 1 - 9 - .	5 9 - 8 9 - 1 1 1 1 - 9 - . -9 ok
(+ (* 1 2 (/ 3 4 5) 8) (- (* 5 5) 2))	1 2 3 4 5 / 8 * 5 5 * 2 - + .	1 2 3 4 5 / 8 * 5 5 * 2 - + . 23 ok
(+ (/ A d) (- A 5 2) (*d 5 B) (- (+C 7) 5))	A @ d @ / A @ 5 2 - d @ 5 B @ * C @ 7 + 5 - + .	A @ d @ / A @ 5 2 - d @ 5 B @ * C @ 7 + 5 - + . 107 ok
= A B	B @ dup A ! .	B @ dup A ! . 20 ok
(+ (= A 32) (- A 5 2) (*d 5 B) (- (+C 7) 5))	32 dup A ! A @ 5 2 - d @ 5 B @ * C @ 7 + 5 - + .	32 dup A ! A @ 5 2 - d @ 5 B @ * C @ 7 + 5 - + . 107 ok
(= A (+ (= B 2) (= C 3)))	2 dup B ! 3 dup C ! + dup A ! .	2 dup B ! 3 dup C ! + dup A ! . 5 ok
(- 11 1 11)	11 1 11 - .	11 1 11 - . -10 ok
(= A (+ (= B 2) (= C (- 11 1 11))))	2 dup B ! 11 1 11 - dup C ! + dup A ! .	2 dup B ! 11 1 11 - dup C ! + dup A ! . 1 ok