PROYECTO 1: COMPILADOR GOv0

Objetivo: Elaborar un compilador para el lenguaje de programación descrito por la gramática incluida en este documento, que genere código ensamblador para la arquitectura MIPS

Gramática

program→declarations
declarations→ declarations declaration | declaration
declaration→ **var** list-var type | **func id(**list-args**)** type-func body-func
type → **int** | **float**
type-func → type | ε
list-var→ list-var **, id** | **id**
list-args →args | ε
args → args , arg | arg
arg → **id** type
body-func →**{** local-declarations statements **}**
local-declarations →local-declarations local-declaration | local-declaration
local-declaration → **var** list-var type
statements→ statements stmt | stmt
stmt→ **id =** expression | **if** expression **{** statements **}**
    | **if** expression **{** statements **} else {** statements **}**
    | **for** expression **{** statements **}** | stmt-print | stmt-scan
stmt-print →**print(** expression **)** | **print (cadena)**
stmt-scan →**scan(id)**
expression→expression **+** expression | expression **-** expression | expression ***** expression
    | expression **/** expression | expression **%** expression | expression **>** expression
    | expression **<** expression | expression **==** expression | expression **!=** expression
    | **(** expression **)** | **id** | **num** | **id(**list-params**)**
list-params →params | ε
params → params **,** param | param
param → expression

Definición de gramática libre de contexto

GLC(N,Σ, S, P)

Donde:

N: Conjunto de símbolos no terminales (N∩Σ) =Φ
Σ: Conjunto de símbolos terminales
S: Símbolo inicial o axioma S∈N
P: Conjunto de produccioness

Estructura de una producción para un GLC

A→α

Donde:

A: Encabezado, A∈N

α: Cuerpo de producción, es una cadena formada por terminales y no terminales ∈ (N∪Σ)*

→: produce

Actividades

1. Indicar cuál es el símbolo inicial de la gramática:
   Program

2. Escribe el conjunto de símbolos terminales:
   ∑ = {var, func, id, (,), int, float, ‘{‘, ‘}’, if, else, +, -, *, /, %,==, <, >, num, print, scan, cadena, !=, for, ‘,’ ,, =}

3. Escribe el conjunto de símbolos no terminales

   N ={programa, declarations, declaration, type, type-func, list-args, args, arg, list-var, statements, stmt, expression, list-params, params, param, body-func, local-declarations, local-declaration, stmt-print, stmt-scan}

Análisis Léxico

4. Asignar a cada símbolo terminal una categoría léxica

Símbolo	Clase léxica
var, func, int, float, if, else, print, scan, for	Palabras reservadas
id	Identificadores
!=, ==, +, -, *, /. %, =, <, >, (,)	Operadores

num	Números
cadena	Cadenas
{, }, ‘;’	Signos de puntuación
‘ ‘, ‘\t’, ‘\n’, ‘\v’, ‘\r’	Espacios
//, /**/	Comentarios

5. Para cada símbolo terminal generar una o más expresiones regulares que permitan reconocer las cadenas pertenecientes a dicho token

Símbolo	**Expresiones Regulares**	**Token generado**
var	var	VAR
func	func	FUNC
int	int	INT
float	float	FLOAT
if	if	IF
else	else	ELSE
print	print	PRINT
scan	scan	SCAN
for	for	FOR
id	letra→[a-zA-ZñÑáéíóúÁÉÍÓÚäëïöüÄËÏÖÜ_] dígito→[0,1,2,3,4,5,6,7,8,9] id→ letra[letra|dígito)*	ID, lexema
!=	diferente	DESIGUALDAD
==	igual	IGUAL
+	mas	SUMA
-	menos	RESTA
*	multiplicacion	MUTIPLICACION
/	division	DIVISION

%	porcentaje	PORCENTAJE
=	asignacion	ASIGNACION
<	menor que	MENOR QUE
>	mayor que	MAYOR QUE
(	parentesis izquierdo	PARENTESIS IZQUIERDO
)	parentesis derecho	PARENTESIS DERECHO
num	ent →dígito((_)? dígito)* exp_decimal →[eE]([+-])?(dígito)+ real →dígito((_)? dígito)*.dígito((_)? dígito)* (exp_decimal)? real2 → dígito((_)? dígito)* exp_decimal real3 →. dígito((_)? dígito)* (exp_decimal)?	NUM,lexema, tipo
cadena	str → ”(\[abfnrtv\'"]| [^”\n]) “	STR, lexema
\t	\t	TABULADOR
\n	\n	SALTO DE LINEA
\v	\v	ESPACIO
\r	\r	

6. Obtener el autómata finito determinista para el analizador léxico que reconozca dada una de las expresiones regulares.

Edo	**Elementos**	**Transiciones**	**Aceptación**

7. Simplificar la tabla anterior para mostrar el AFD solo con sus estados y transiciones

8. Implementar el AFD en lenguaje C++

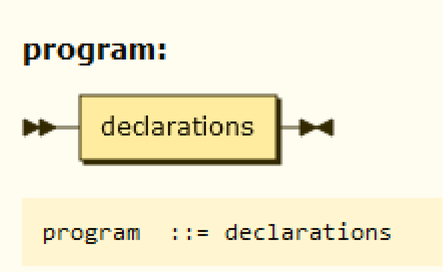
Análisis Sintáctico

9. Representar la gramática en notación EBNF

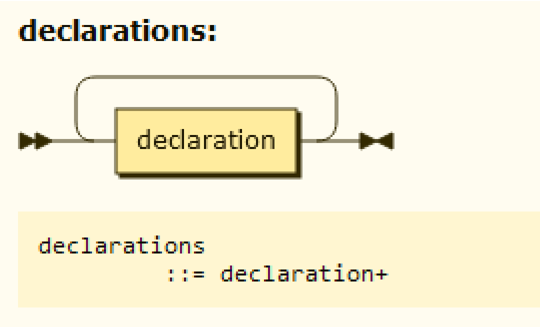
<table>
<tr><th></th><th></th></tr>
<tr><td>program→declarations
declarations→ declarations declaration | declaration
declaration→ **var** list-var type **| func id(**list-args**)** type-func body-func
type → **int** | **float**
type-func → type | ε
list-var→ list-var **, id** | **id**
list-args →args | ε
args → args , arg | arg
arg → **id** type
body-func →**{** local-declarations statements **}**
local-declarations →local-declarations local-declaration | local-declaration
local-declaration → **var** list-var type
statements→ statements stmt | stmt
stmt→ **id** = expression | **if** expression **{** statements **}**
| **if** expression **{** statements **} else { statements }**
| **for** expression **{** statements **}** | stmt-print | stmt-scan
stmt-print →**print(** expression **)** | **print (cadena)**
stmt-scan →**scan(id)**
expression→expression **+** expression | expression **-** expression | expression ***** expression
| expression **/** expression | expression **%** expression | expression **>** expression
| expression **<** expression | expression **==** expression | expression **!=** expression
| **(** expression **)** | **id** | **num** | **id(**list-params**)**
list-params →params | ε
params → params **,** param | param
param → expression</td><td>programa→declarations
declarations → declaration { declaration }
declaration → **var** list-var type | **func id(**list-args**)** type-func body-func
type →**int** | **float**
type-func →[type]
list-var →**id** {**, id**}
list-args →[args]
args →arg { **,** arg }
arg → **id** type
body-func → **{** local-declarations statements **}**
local-declarations →local-declaration {local-declaration}
local-declaration → **var** list-var type
statements →stmt {stmt}
stmt→ **id =** expression | **if** expression **{** statements **}** [**else {** statements **}**]
| **for** expression **{** statements **}** | stmt-print | stmt-scan
stmt-print →**print(** (expression | **cadena**) **)**
stmt-scan →**scan(id)**
expression→(**(** expression **)** | **num** | **id**[**(**list-params**)**]){**+** expression |
- expression | ***** expression | **/** expression | **%** expression
| **>** expression **<** expression | **==** expression | **!=** expression }

list-params →params | ε
params → params **,** param | param
param → expression</td></tr>
</table>

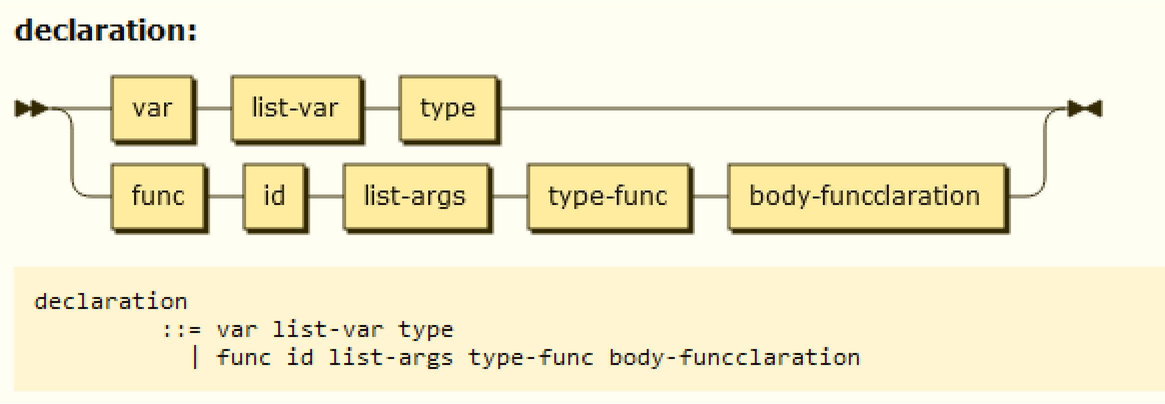
10. Elaborar los diagramas de sintaxis en base a la gramática del punto 9


program:

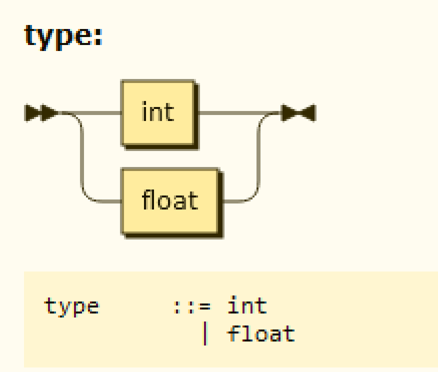
```
program  ::= declarations
```


declarations:

```
declarations
         ::= declaration+
```

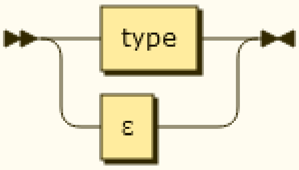

declaration:

```
declaration
         ::= var list-var type
           | func id list-args type-func body-funcclaration
```


type:

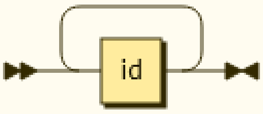
```
type     ::= int
           | float
```

type-func:



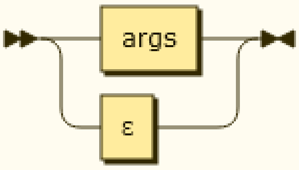
```
type-func
         ::= type
           | ε
```

list-var:



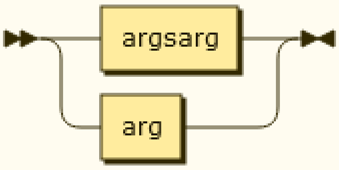
```
list-var ::= id+
```

list-args:



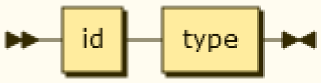
```
list-args
         ::= args
           | ε
```

args:



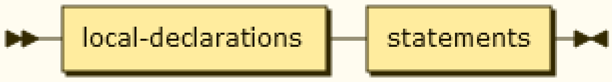
```
args     ::= argsarg
           | arg
```

arg:



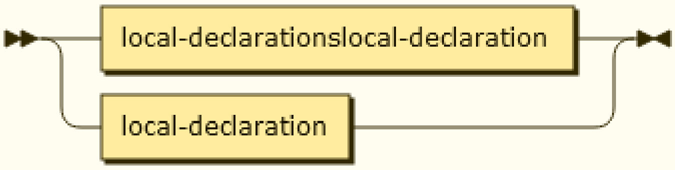
```
arg      ::= id type
```

body-func:



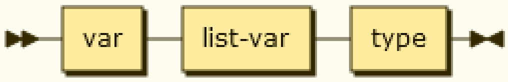
```
body-func
         ::= local-declarations statements
```

local-declarations:



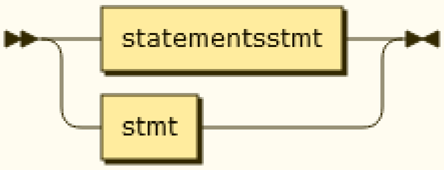
```
local-declarations
         ::= local-declarationslocal-declaration
           | local-declaration
```

local-declaration:



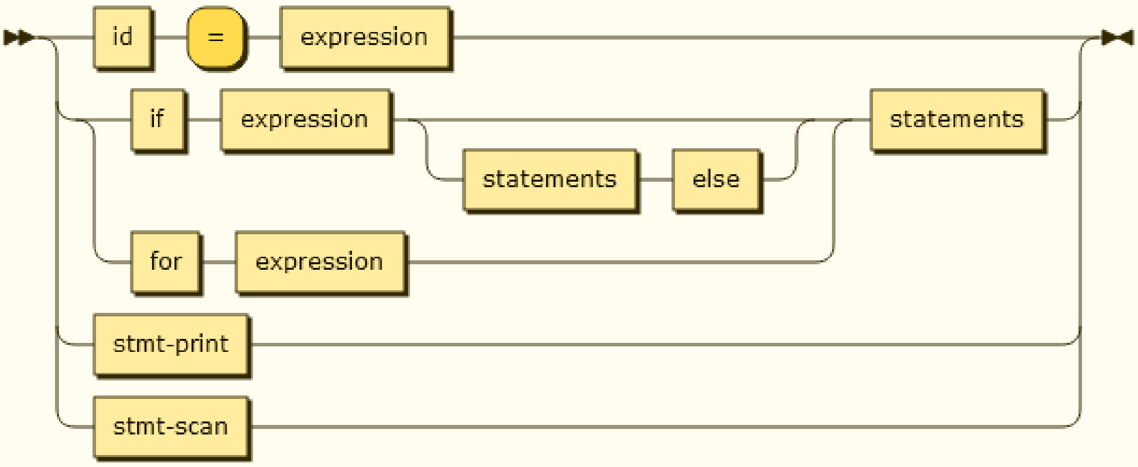
```
local-declaration
         ::= var list-var type
```

statements:



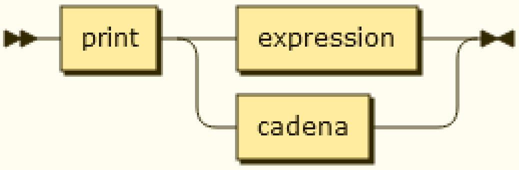
```
statements
         ::= statementsstmt
           | stmt
```

stmt:



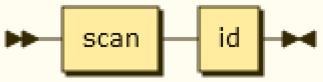
```
stmt     ::= id '=' expression
           | ( if expression ( statements else )? | for expression ) statements
           | stmt-print
           | stmt-scan
```

stmt-print:



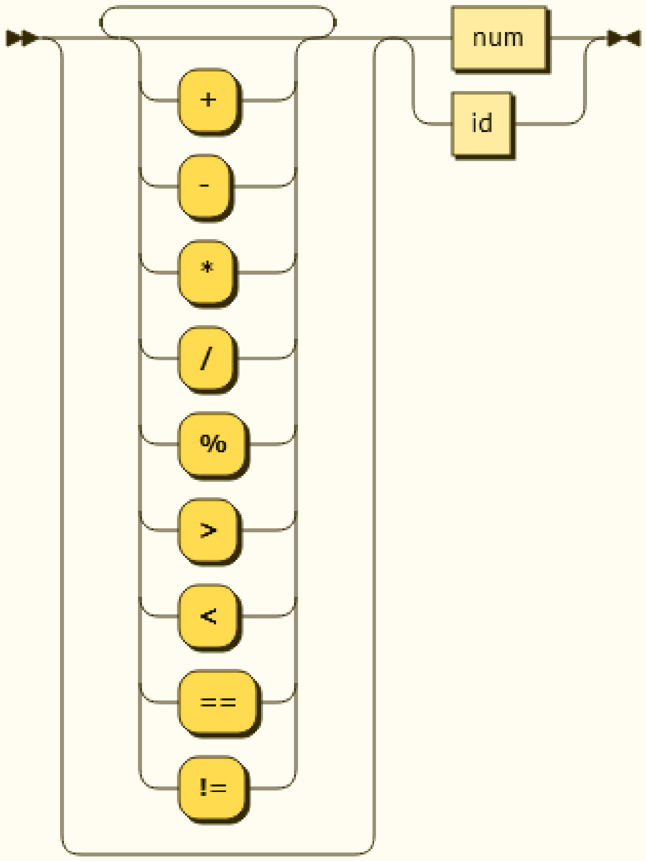
```
stmt-print
         ::= print ( expression | cadena )
```

stmt-scan:



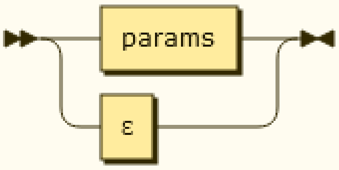
```
stmt-scan
         ::= scan id
```

expression:

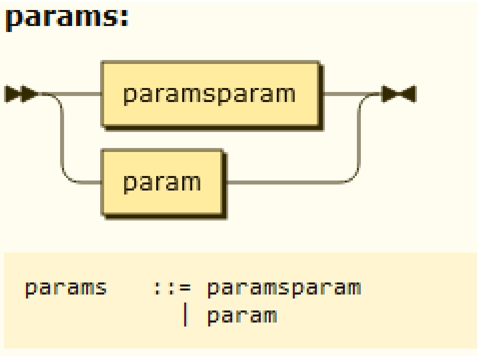


```
expression
         ::= ( '+' | '-' | '*' | '/' | '%' | '>' | '<' | '==' | '!=' )?* ( num | id )
```

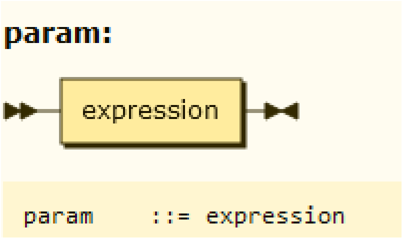
list-params:



```
list-params
         ::= params
           | ε
```



```
params   ::= paramsparam
           | param
```



```
param    ::= expression
```

11. Eliminar la ambigüedad de la gramática en caso de ser necesario

expression→expression **+** expression | expression **-** expression | expression * expression
| expression **/** expression | expression **%** expression | expression **>** expression
| expression **<** expression | expression **==** expression | expression **!=** expression
| **(** expression **)** | **id** | **num** | **id(**list-params**)**

Operador	Nivel	Símbolo	Asociatividad izq	Asociatividad derecha
Igualdad	5	==	SI	
Desigualdad	5	!=	SI	
Menor que	4	<	SI	
Mayor que	4	>	SI	
Resta	3	-	SI	
Suma	3	+	SI	
Multiplicación	2	*	SI	
División	2	/	SI	
Porcentaje	2	%	SI	
Parentesis	1	()		

Análisis Sintáctico Descendente

12. Eliminar la recursividad izquierda

Producción original	**Producciones sin recursividad izquierda**
declarations→ declarations declaration | declaration	declarations → declaration declarations_l declarations_l → declaration declarations_l | ε
list-var→ list-var **, id | id**	list-var → id list-var_l list-var_l → , id list-var_l | ε
args → args , arg | arg	args → arg args_l args_l → arg args_l | ε
local-declarations →local-declarations local-declaration | local-declaration	local-declarations →local-declaration local-declarations_l local-declarations_l→ local-declaration local-declarations_l | ε
statements→ statements stmt | stmt	statements→ stmt staments_l staments_l → stmt staments_l | ε
params → params **,** param | param	params → param params_l params_l → param params-l | ε

Producción original

expression → expression **==** expression_n5 | expression_n5 **!=** expression_n4 | expression_n4

expression_n4 → expression_n4 < expression_n3 | expression_n4 > expression_n3| expression_n3

expression_n3 → expression_n3 + expression_n2 | expression_n3 - expression_n2 | expression_n2

expression_n2 → expression_n2 / expression_n1 | expression_n2 * expression_n1 | expression_n2 % expression_n1 | expression_n1

expression_n1 → | **(** expression **)** | **id** | **num** | **id(**list-params**)**

Producción sin recursividad izquierda

expression → expression_n5 expression_l
expression_l → == expression_n5 expression_l | != expression_n5 expression_l | ε

expression_n4 → expression_n3 expression_n4_l
expression_n4_l → < expression_n3 expression_n4_l | > expression_n3 expression_n4_l | ε

expression_n3 → expression_n2 expression_n3_l
expression_n3_l → + expression_n2 expression_n3_l | - expression_n2 expression_n3_l | ε

expression_n2 → expression_n1 expression_n2_l
expression_n2_l → / expression_n1 expression_n2_l | *expression_n1 expression_n2_l |
%expression_n1 expression_n2_l | ε

Gramática hasta ahora

program→declarations
declarations → declaration declarations_l
declarations_l → declaration declarations_l | ε
declaration→ **var** list-var type | **func id(**list-args**)** type-func body-func
type → **int** | **float**
type-func → type | ε
list-var → id list-var_l
list-var_l → id list-var_l | ε
list-args →args | ε
args → arg args_l
args_l → arg args_l | ε
arg → **id** type
body-func →**{** local-declarations statements **}**
local-declarations →local-declaration local-declarations_l
local-declarations_l→ local-declaration local-declarations_l | ε
local-declaration → **var** list-var type
statements→ stmt staments_l
staments_l → stmt staments_l | ε
stmt→ **id =** expression | **if** expression **{** statements **}**
| **if** expression **{** statements **} else {** statements **}**
| **for** expression **{** statements **}** | stmt-print | stmt-scan
stmt-print →**print(** expression **)** | **print (cadena)**
stmt-scan →**scan(id)**
expression → expression_n5 expression_l
expression_l → == expression_n5 expression_l | != expression_n5 expression_l | ε
expression_n4 → expression_n3 expression_n4_l
expression_n4_l → < expression_n3 expression_n4_l | > expression_n3 expression_n4_l | ε
expression_n3 → expression_n2 expression_n3_l
expression_n3_l → + expression_n2 expression_n3_l | - expression_n2 expression_n3_l | ε
expression_n2 → expression_n1 expression_n2_l
expression_n2_l → / expression_n1 expression_n2_l | *expression_n1 expression_n2_l |
%expression_n1 expression_n2_l | ε

expression_n1 → **(** expression **)** | **id** | **num** | **id(**list-params**)**

list-params →params | ε

params → param params_l

params_l → param params_l | ε

param → expression

13. Eliminar los factores izquierdos

Gramática sin factores izquierdos

program → declarations

declarations → declaration declarations_l

declarations_l → declaration declarations_l | ε

declaration→ **var** list-var type | **func id(**list-args**)** type-func body-func

type → **int** | **float**

type-func → type | ε

list-var → id list-var_l

list-var_l → id list-var_l | ε

list-args →args | ε

args → arg args_l

args_l → arg args_l | ε

arg → **id** type

body-func →**{** local-declarations statements **}**

local-declarations →local-declaration local-declarations_l

local-declarations_l→ local-declaration local-declarations_l | ε

local-declaration → **var** list-var type

statements→ stmt staments_l

staments_l → stmt staments_l | ε

stmt→ **id =** expression | **if** expression **{** statements **}**

| **else {** statements **}**

| **for** expression **{** statements **}** | stmt-print | stmt-scan

stmt-print →**print(** expression **)** | **print (cadena)**

stmt-scan →**scan(id)**

expression → expression_n5 expression_l

expression_l → == expression_n5 expression_l | != expression_n5 expression_l | ε

expression_n4 → expression_n3 expression_n4_l

expression_n4_l → < expression_n3 expression_n4_l | > expression_n3 expression_n4_l | ε

expression_n3 → expression_n2 expression_n3_l

expression_n3_l → + expression_n2 expression_n3_l | - expression_n2 expression_n3_l | ε

expression_n2 → expression_n1 expression_n2_l expression_n2_l → / expression_n1 expression_n2_l | *expression_n1 expression_n2_l | %expression_n1 expression_n2_l | ε

expression_n1 → **(** expression **)** | **id** | **num** | **id(**list-params**)**

list-params →params | ε

params → param params_l

params_l → param params_l | ε

param → expression

14. Calcular anulables, los conjuntos FIRST y FOLLOW para la gramática sin recursividad izquierda y sin factores izquierdos

Símbolo	Anulable	FIRST	FOLLOW
program	No	var, func	$
declarations	No	var, func	$
declaration_l	Si	var, func	$
declaration	No	var, func	var, func,$
type	No	int, float	var,func,id,$
type-func	Si	int,float	var
list-var	No	id	int, float
list-var_l	No	id	int, float
listargs	Si	id	
args	No	id	
args’	Si	id	
arg	No	id	id
body-func	No	var	var,func,$
local-declarations	No	var	id, if, else, for, print(, print, scan(id))
local-declarations’	No	var	id, if, else, for, print(, print, scan(id))
local-declaration	No	var	var
staments	No	id, if, else, for, print(, print, scan(id))	var,fun,staments_l,,$

statements_l	Si	id, if, else, for, print(, print, scan(id))	
stmt	No	id, if, else, for, print(, print, scan(id))	staments_l
stmt-print	No	print(,print)	staments_l
stmt-scan	No	scan(id)	staments_l
expression	No	expression_n5	staments_l, id, if, else, for, print(print, scan(id)), ε, expression_n5
expression_l	Si	==, !=	staments_l, id, if, else, for, print(print, scan(id)), ε, expression_n5
expression_n 4	No	(, id , num , id(listparams))	
expression_n 4_l	Si	<, >	
expression_n 3	No	(, id , num , id(listparams))	<, >
expression_n 3_l	Si	+, -	<, >
expression_n 2	No	(, id , num , id(listparams))	<, >, +, -
expression_n 2_l	Si	/, *, %	<, >, +, -
expression_n 1	No	(, id , num , id(listparams))	<, >, +, -, /, *, %
list-params	No	ε, expression_n5	
params	No	expression_n5	
params_l	No	ε, expression_n5	
param	No	expression_n5	ε, expression_n5

15. Implementar el analizador sintáctico recursivo sin retroceso
16. Construir la tabla de análisis sintáctico LL(1)

1)

	$	**var**	**func**	**id(list-arg)**
program		program → declarations program → S $	program → declarations program → program $	
declarations		declarations → declarations declarations_l	declarations → declarations declarations_l	
declarations_l	declarations_l →	declarations_l → declarations declarations_l	declarations_l → declarations declarations_l	
declaration		declaration → var list-var type	declaration → func id(list-args) type-func body-func	
type				
type-func		type-func → ε		
list-var				
list-var_l				
list-args				
args				
args_l				
arg				
body-func		bodyfunc → local-declarations statements		
local-declarations		local-declarations → local-declaration local-declarations_l		

local-declarations_l		local-declarations_l → local-declaration local-declarations_l		
staments		local-declarations → var list-var type		
stament_l				
stmt				
stmt-print				
stmt-scan				
expression				
expression_l				
expression_n4				
expression_n4_l				
expression_n3				
expression_n3_l				
expression_n2				
expression_n2_l				
expression_n1				
list-params				
params				
params_l				
param				

2)

	int	**float**	**id**	**staments_l**
program				
declarations				

declarations_l				
declaration				
type	type → int	type → float		
type-func	type-func→ type	type-func → type		
list-var			list-var→ id list-var_l	
list-var_l			list-var_l→ id list-var_l	
list-args			list-args → args	
args			args → arg args_l	
args_l			args_l → arg args_l	
arg			arg → id type	
body-func				
local-declarations				
local-declarations_l				
staments			statements → stmt statements_l	
stament_l			statements_l → stmt statements_l	
stmt			stmt → id = expression	
stmt-print				
stmt-scan				
expression				
expression_l			expression_l → ε	expression_l → ε
expression_n4			expression_n4 → expression_n3 expression_n4_l	

expression_n4_l				
expression_n3			expression_n3 → expression_n2 expression_n3_l	
expression_n3_l				
expression_n2			expression_n2 → expression_n1 expression_n2_l	
expression_n2_l				
expression_n1			expression_n1 → id	
list-params				
params				
params_l				
param				

3)

	=	**if**	**else**	**for**
program				
declarations				
declarations_l				
declaration				
type				
type-func				
list-var				
list-var_l				
list-args				

args				
args_l				
arg				
body-func				
local-declarations				
local-declarations_l				
staments		statements → stmt staments_l	statements → stmt staments_l	statements → stmt staments_l
stament_l		statements_l → stmt staments_l	statements_l → stmt staments_l	statements_l → stmt staments_l
stmt		stmt → if expression statements	stmt →else statements	stmt →for expression statements
stmt-print				
stmt-scan				
expression				
expression_l		expression' ::= ε	expression' ::= ε	expression' ::= ε
expression_n4				
expression_n4_l				
expression_n3				
expression_n3_l				
expression_n2				
expression_n2_l				
expression_n1				
list-params				
params				
params_l				
param				

4)

	print(	**)**	**print**	**(**
program				
declarations				
declarations_l				
declaration				
type				
type-func				
list-var				
list-var_l				
list-args				
args				
args_l				
arg				
body-func				
local-declarations				
local-declarations_l				
staments	statements → stmt staments_l		statements → stmt staments_l	
stament_l	statements → stmt staments_l		statements → stmt staments_l	

stmt	stmt → stmt-print		stmt → stmt-print	
stmt-print	stmt-print → print(expression)		stmt-print → print(expression)	
stmt-scan				
expression				
expression_l	expression_l → ε	expression_l → ε	expression_l → ε	
expression_n4				expression_n4 → expression_n3 expression_n4_l
expression_n4_l				
expression_n3				expression_n3 → expression_n2 expression_n3_l
expression_n3_l				
expression_n2				expression_n2 → expression_n1 expression_n2_l
expression_n2_l				
expression_n1				expression_n1 → expression
list-params				
params				
params_l				
param				

5)

	cadena	**scan(id)**	**expression_n5**	**==**
program				

declarations				
declarations_l				
declaration				
type				
type-func				
list-var				
list-var_l				
list-args				
args				
args_l				
arg				
body-func				
local-declarations				
local-declarations_l				
staments		statements → stmt staments_l		
stament_l		statements → stmt staments_l		
stmt		stmt → stmt-scan		
stmt-print				
stmt-scan		stmt-scan → scan(id)		
expression			expression → expression_n5 expression_l	
expression_l		expression_l → ε	expression_l → ε	expression_l → == expression_n5 expression_l

expression_n4				
expression_n4_l				
expression_n3				
expression_n3_l				
expression_n2				
expression_n2_l				
expression_n1				
list-params			list-params → params	
params			params → param params_l	
params_l			params_l → param params_l	
param			param → expression	

6)

	!=	<	>	+
program				
declarations				
declarations_l				
declaration				
type				
type-func				
list-var				
list-var_l				
list-args				

args				
args_l				
arg				
body-func				
local-declarations				
local-declarations_l				
staments				
stament_l				
stmt				
stmt-print				
stmt-scan				
expression				
expression_l	expression_l → != expression_n5 expression_l			
expression_n4				
expression_n4_l		expression_n4_l → < expression_n3 expression_n4_l	expression_n4_l → >expression_n3 expression_n4_l	
expression_n3				
expression_n3_l		expression_n3_l → ε	expression_n3_l → ε	expression_n3_l → + expression_n2 expression_n3_l
expression_n2				
expression_n2_l		expression_n2_l → ε	expression_n2_l → ε	expression_n2_l → ε
expression_n1				
list-params				

params				
params_l				
param				

7)

	-	/	*	%
program				
declarations				
declarations_l				
declaration				
type				
type-func				
list-var				
list-var_l				
list-args				
args				
args_l				
arg				
body-func				
local-declarations				
local-declarations_l				
staments				
stament_l				

stmt				
stmt-print				
stmt-scan				
expression				
expression_l				
expression_n4				
expression_n4_l				
expression_n3				
expression_n3_l	expression_n3_l → - expression_n2 expression_n3_l			
expression_n2				
expression_n2_l	expression_n2_l →ε	expression_n2_l → / expression_n1 expression_n2_l	expression_n2_l → * expression_n1 expression_n2_l	expression_n2_l → % expression_n1 expression_n2_l
expression_n1				
list-params				
params				
params_l				
param				

8)

	num	**id(list-params)**	**ε**
program			
declarations			
declarations_l			

declaration			
type			
type-func			
list-var			
list-var_l			
list-args			
args			
args_l			
arg			
body-func			
local-declarations			
local-declarations_l			
staments			
stament_l			
stmt			
stmt-print			
stmt-scan			
expression			
expression_l			expression_l → ε
expression_n4	expression_n4 → expression_n3 expression_n4_l	expression_n4 → expression_n3 expression_n4_l	
expression_n4_l			
expression_n3	expression_n3 → expression_n2 expression_n3_l	expression_n3 → expression_n2 expression_n3_l	
expression_n3_l			
expression_n2	expression_n2 → expression_n1 expression_n1_l	expression_n2 → expression_n1 expression_n1_l	

expression_n2_l			
expression_n1	expression_n1 → num	expression_n1 → id(list-params)	
list-params			list-params → ε
params			
params_l			params_l → ε
param			

Traducción dirigida por sintaxis

17. Definir las acciones semánticas para el análisis semántico en el análisis sintáctico descendente

Reglas de producción	Reglas semánticas

18. Definir las acciones semánticas para el análisis semántico en el análisis sintáctico ascendente

Reglas de producción	Reglas semánticas

19. Definir las acciones semánticas para la generación de código intermedio en el análisis sintáctico descendente

Reglas de producción	Reglas semánticas

20. Definir las acciones semánticas para la generación de código intermedio en el análisis sintáctico ascendente

Reglas de producción	Reglas semánticas

Generación de código objeto y manejo de memoria

21. Revisar la arquitectura MIPS para generar código

22. Elaborar en MIPS un programa que calcule el factorial de un número utilizando funciones
23. Elaborar en MIPS un programa que calcule el cuadrado de un número utilizando funciones
24. Especificar un generador de código de fuerza bruta