Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Proyecto Final

Equipo:
David G. Palmerin Morales 313223570
Melissa Mendez Servin. 313192687

Contents

1 Gramática

Se dio la siguiente gramática:

- programa → declaraciones funciones
- declaraciones → tipo lista ; | ϵ
- tipo → **int** | **float** | **double** | **char** | **void** | **struct** **{** declaraciones **}**
- lista → lista , **id** arreglo | **id** arreglo
- arreglo → [**numero**] arreglo | ϵ
- funciones → **func** tipo **id**(argumentos){ declaraciones sentencias } funciones | ϵ
- argumentos → lista_argumentos — ϵ
- lista_argumentos → lista_argumentos , tipo **id** parte_arreglo | tipo **id** parte_arreglo
- parte_arreglo → [ ] parte_arreglo | ϵ
- sentencia → sentencia sentencia — **if**(condición) sentencia |**if**(condición) sentencia **else** sentencia | **while**(condición) sentencia | **do** sentencia **while**(condición); | **for**(sentencia ; condición; sentencia) sentencia | parte_izquierda = expresión ;| **return** expresión; |**return**; |**{** sentencia**}** | **switch**(expresión){ casos predeterminado} | **break**; | **print** expresión;
- casos → **case:** **numero** sentencia predeterminado | ϵ
- predeterminado → **default:** sentencia | ϵ
- parte_izquierda → **id** | var_arreglo | **id.id**
- var_arreglo → **id**[expresión] | va_arreglo [expresión]
- expresión → expresión + expresión | expresión − expresión | expresión ∗ expresión | expresión / expresiónn | expresión % expresión | var_arreglo | **cadena** | **numero** | **caracter** | **id**(parámetros)
- parámetros → ϵ | lista_param
- lista_param → lista param , expresión | expresión
- condición → condición || condición | condición **&&** condición | **!** condición | (condición) | expresiónn relacional expresión | true | false
- relacional → < | > | >= | <= | ! = | ==

2 Analizador léxico: Diseño y especificación

Para la construcción de nuestro compilador primero desarrollamos nuentro analizador léxico, para ello, de acuerdo a la gramática dada, se usarán las expresiones regulares especificadas en la siguiente tabla de clases léxicas:

No básicos	**Elementos Básicos**	**Estado**	
Clase Léxica	Descripción	Expresiones Regulares	Acción Léxica
1	Palabras reservadas		
		”int”	Retorna ”int”.
		”float”	Retorna ”float”.
		”double”	Retorna ”double”.
		”char”	Retorna ”char”.
		”void”	Retorna ”void”.
		”struct”	Retorna ”struct”.
		”else”	Retorna ”else”.
		”while”	Retorna ”while”.
		”do”	Retorna ”do”.
		”for”	Retorna ”for”.
		”switch”	Retorna ”switch”.
		”case”	Retorna ”case”.
		”default”	Retorna ”default”.
		”return”	Retorna ”return”.
		”break”	Retorna ”break”.
		”true”	Retorna ”true”.
		”false”	Retorna ”false”.
		”func”	Retorna ”func”.
		”print”	Retorna ”print”.
2	Identificadores	[_a-zA-Z][_a-zA-Z0-9]{0,30}	Retornar un identificador.

Clase Léxica	Descripción	Expresiones Regulares	Acción Léxica
3	Operadores		
		=	Retorna =.
		&&	Retorna &&.
		||	Retorna ||.
		==	Retorna ==.
		! =	Retorna ! =.
		>	Retorna >.
		<	Retorna <.
		>=	Retorna >=.
		<=	Retorna <=.
		+	Retorna +.
		−	Retorna −.
		∗	Retorna ∗.
		%	Retorna %.
4	Signos especiales	[.;,(){}[]]	Retornar un signo especial
5	Constantes Numéricas	(-)?$[0-9]^+$	Retornar constante numércia de tipo int.
		(-)?$[0-9]^*.[0-9]\{1,7\}$	Retornar constante numércia de tipo float.
		(-)?$[0-9]^*.[0-9]\{8,16\}$	Retornar constante numércia de tipo double.
6	Constantes de cadenas o caracteres.	'[^']'	Retornar constante de un caracter.
		\\([(){}] | [^([0-9] | [a-zA-Z] | [\n\t])])	Retornar constante de un símbolo especial.
		"[^"]* "	Retornar constante de una cadena.
7	Espacios en blanco	[\n\t]	Ignorar

Diseño de las expresiones regulares

- Clase 1
  Las palabras reservadas se mantienen tal y como se definen en la gramática propuesta.
- Clase 2

  $$\mathbf{[\_a\text{-}zA\text{-}Z][\_a\text{-}zA\text{-}Z0\text{-}9]\{0,30\}}$$

  Los identificadores los definimos como en el lenguaje C, ya que éstos no pueden iniciar con un número, caracter especial o bien guión alto, en la primera parte solo permitimos la elección de un caracter que sea una letra o bien un guión bajo (para el primer caracter), y en la segunda parte la elección de caracteres que sean nuevamente letras, guión bajo y ahora sí números. También restringimos que el tamaño no sea mayor a 32, puesto que nuestras direcciones para el código intermedio son de tamaño 32 bytes, ayudandonos así en el manejo de direcciones de los identificadores.
- Clase 3
  Los operadores permanecen igual, respetando la gramática propuesta.
- Clase 4
  Los signos especiales del mismo modo se mantienen como fueron definidos en la gramática.
- Clase 5
  Las constantes numéricas que se tienen las siguientes:
  - Tipo int

    $$\mathbf{(\text{-})?}[0-9]^{+}$$

    Para referirnos a los números enteros, donde puede tratarse de un número negativo o positivo, por ello la primera parte, en el que se puede o no elegir el símbolo '-', seguido de un dígito entre 0-9, usando la cerradura positiva de Kleene para asegurarnos de que haya al menos un dígito.
  - Tipo float

    $$\mathbf{(\text{-})?}[0-9]^{*}\mathbf{.}[0-9]\{1,7\}$$

    Es análoga para la parte entera, seguida por un punto, y finalemnte por a lo más 8 dígitos, cumplendo así con el tamaño de un float que es de 4 bytes.

  - Tipo double

    $$\textbf{(-)?}[0-9]^*\textbf{.}[0-9]\{8,16\}$$

    Es ánalogo al float, solo que para la parte que está después del punto debe de tener entre 8 y 16 símbolos asegurandonos así que se trata siempre de un double, el cual tiene un tamaño de 8 bytes.

- Clase 6

  Para las constantes de cadenas y caraternes se eligieron las siguientes expresiones:

  - Caracter

    $$'[^']'$$

    En donde simplente nos preocupamos por que un caracter esté dentro de comillas simples, sin importarnos cual sea éste.

  - Caracter especial

    $$\backslash\backslash(\ [\hat{}(\textbf{[0-9]} \mid \textbf{[a-zA-Z]} \mid \textbf{[\backslash n\backslash t]})] \mid \textbf{[()\{\}]})$$

    Para los simbolos especiales, únicamente nos fijamos no se trate de un número, letra o espacio, y ya que tuvimos un problema al momento de pasarle llaves, decidimos escrbirlas literalmente.

  - Cadena

    $$"[^"]^* \ "$$

    Las cadenas al ser un conjunto de caracteres (sin importar cuales sean estos), decidimos usar la estrella de Kleen para cualquier caracter exceptuando las dobles comillas, pues éstos estaran encerrados por dobles comillas.

- Clase 7

  $$\textbf{[\backslash n\backslash t]}$$

  Para los espacios en blanco, tomamos en cuenta saltos de línea, tabuladores y espacios en blaco, los cuales serán ignorados y nos sirven para separar los tokens.

A continuación, llevamos a cabo el analizador léxico de nuestras expresiones regulares.

Analizador léxico

No básicos	Elementos Básicos	Estado
· float	· float	
· double	· double	
· char	· char	
· void	· void	
· struct	· struct	
· if	· if	
· else	· else	
· while	· while	
· do	· do	
· for	· for	
· switch	· switch	
· case	· case	
· default	· default	
· return	· return	
· break	· break	
· true	· true	
· false	· false	
· func	· func	
· print	· print	
· =	· =	
· &&	· &&	
· ||	· ||	
· ==	· ==	
· ! =	· ! =	
· >	· >	
· <	· <	
· >=	· >=	
· <=	· <=	
· +	· +	
· −	· −	
· ∗	· ∗	
· %	· %	
· !	· !	
· (	· (	
·)	·)	
· {	· {	
· }	· }	
· [	· [	

No básicos	Elementos Básicos	Estado
·] · ; · , · . · [_a-zA-Z][_a-zA-Z0-9]{0, 31} · '[^']' · "[^"]*" · [−+]?[0 − 9]+ · [−+]?[0 − 9]*.[0-9]{1,7} · [−+]?[0 − 9]*.[0-9]{8,16} · \\([^([0-9] | [a-zA-Z] | [\n\t])] | [(){}])	·] · ; · , · . · [_a-zA-Z][_a-zA-Z0-9]{0, 31} · '[^']' · "[^"]*" · [−+]?[0 − 9]+ · [−+]?[0 − 9]*.[0-9]1,7 · [−+]?[0 − 9]*.[0-9]8,16 · \\([^([0-9] | [a-zA-Z] | [\n\t])] | [(){}])	

Goto	Elementos básicos	Estado
goto(q_0,i)	i·nt i·f	q_1
goto(q_1,n)	in·t	q_2
goto(q_2,t)	int·	q_3F
goto(q_1,f)	if·	q_4F
goto(q_0,f)	f·loat f·or f·unc f·alse	q_5
goto(q_5,l)	fl·oat	q_6
goto(q_6,o)	flo·at	q_7
goto(q_7,a)	floa·t	q_8
goto(q_8,o)	float·	q_9F
goto(q_5,o)	fo·r	q_{10}
goto(q_{10},r)	fo·r	$q_{11}F$
goto(q_5,u)	fu·nc	q_{12}
goto(q_{12},n)	fun·c	q_{13}
goto(q_{13},c)	func·	$q_{14}F$
goto(q_5,a)	fa·lse	q_{15}
goto(q_{15},l)	fal·se	q_{16}
goto(q_{16},s)	fals·e	q_{17}
goto(q_{17},e)	false·	$q_{18}F$
goto(q_0,i)	i·nt i·f	q_1
goto(q_1,n)	in·t	q_2
goto(q_2,t)	int·	q_3F

Goto	**Elementos básicos**	**Estado**
goto(q_1,f)	if·	q_4F
goto(q_0,f)	f·loat f·or f·unc f·alse	q_5
goto(q_5,l)	fl·oat	q_6
goto(q_6,o)	flo·at	q_7
goto(q_7,a)	floa·t	q_8
goto(q_8,o)	float·	q_9F
goto(q_5,o)	fo·r	q_{10}
goto(q_{10},r)	fo·r	$q_{11}F$
goto(q_5,u)	fu·nc	q_{12}
goto(q_{12},n)	fun·c	q_{13}
goto(q_{13},c)	func·	$q_{14}F$
goto(q_5,a)	fa·lse	q_{15}
goto(q_{15},l)	fal·se	q_{16}
goto(q_{16},s)	fals·e	q_{17}
goto(q_{17},e)	false·	$q_{18}F$
goto(q_0,d)	d·ouble d·o d·efault	q_{19}
goto(q_{19},o)	do·uble do·	$q_{20}F$
goto(q_{20},u)	dou·ble	q_{21}
goto(q_{21},b)	doub·le	q_{22}
goto(q_{22},l)	doubl·e	q_{23}
goto(q_{24},e)	double·	$q_{24}F$
goto(q_{19},e)	de·fault	q_{25}
goto(q_{25},f)	def·ault	q_{26}
goto(q_{26},a)	defa·ult	q_{27}
goto(q_{27},u)	defau·lt	q_{28}
goto(q_{28},l)	defaul·t	q_{29}
goto(q_{29},t)	default·	$q_{30}F$
goto(q_0,c)	c·har c·ase	q_{31}
goto(q_{31},h)	ch·ar	q_{32}
goto(q_{32},a)	cha·r	q_{33}

Goto	**Elementos básicos**	**Estado**
goto(q_{33},r)	char·	$q_{34}F$
goto(q_{31},a)	ca·se	q_{35}
goto(q_{35},s)	cas·e	q_{36}
goto(q_{36},e)	case·	$q_{37}F$
goto(q_0,v)	v·oid	q_{38}
goto(q_{38},o)	vo·id	q_{39}
goto(q_{39},i)	voi·d	q_{40}
goto(q_{40},d)	void·	$q_{41}F$
goto(q_0,s)	s·truct s·witch	q_{42}
goto(q_{42},t)	st·ruct	q_{43}
goto(q_{43},r)	str·uct	q_{44}
goto(q_{44},u)	stru·ct	q_{45}
goto(q_{45},c)	struc·t	q_{46}
goto(q_{46},t)	struct·	$q_{47}F$
goto(q_{42},w)	sw·itch	q_{48}
goto(q_{48},i)	swi·tch	q_{49}
goto(q_{49},t)	swit·ch	q_{50}
goto(q_{50},c)	switc·h	q_{51}
goto(q_{51},h)	switch·	$q_{52}F$
goto(q_0,e)	e·lse	q_{53}
goto(q_{53},l)	e·lse	q_{54}
goto(q_{54},s)	e·lse	q_{55}
goto(q_{55},e)	e·lse	$q_{56}F$
goto(q_0,w)	w·hile	q_{57}
goto(q_{57},h)	wh·ile	q_{58}
goto(q_{58},i)	whi·le	q_{59}
goto(q_{59},l)	whil·e	q_{60}
goto(q_{60},e)	whilr·e	$q_{61}F$
goto(q_0,r)	r·eturn	q_{62}
goto(q_{62},e)	re·turn	q_{63}
goto(q_{63},t)	ret·urn	q_{64}
goto(q_{64},u)	retu·rn	q_{65}
goto(q_{65},r)	retur·n	q_{66}
goto(q_{66},n)	return·	$q_{67}F$
goto(q_0,b)	b·reak	q_{68}
goto(q_{68},r)	br·eak	q_{69}
goto(q_{69},e)	bre·ak	q_{70}
goto(q_{70},a)	brea·k	q_{71}

Goto	**Elementos básicos**	**Estado**
goto(q_{71},k)	break·	$q_{72}F$
goto(q_0,t)	t·rue	q_{73}
goto(q_{73},r)	tr·ue	q_{74}
goto(q_{74},u)	tru·e	q_{75}
goto(q_{75},e)	true·	$q_{76}F$
goto(q_0,p)	p·rint	q_{77}
goto(q_{77},r)	pr·int	q_{78}
goto(q_{78},i)	pri·nt	q_{79}
goto(q_{79},n)	prin·t	q_{80}
goto(q_{80},t)	print·	$q_{81}F$
goto(q_0,=)	=· =·=	$q_{82}F$
goto(q_{82},=)	==·	$q_{83}F$
goto(q_0,&)	&·&	q_{84}
goto(q_{84},&)	&&·	$q_{85}F$
goto(q_0,|)	|·|	q_{86}
goto(q_{86},|)	||·	$q_{87}F$
goto(q_0,!)	!·= !·	$q_{88}F$
goto(q_{88},=)	!=·	$q_{89}F$
goto(q_0,¿)	¿·= ¿·	$q_{90}F$
goto(q_{90},=)	¿=·	$q_{91}F$
goto(q_0,¡)	¡·= ¡·	$q_{92}F$
goto(q_{92},=)	¡=·	$q_{93}F$
goto(q_0,+)	+·	$q_{94}F$
goto(q_0,-)	-·	$q_{95}F$
goto(q_0,*)	*·	$q_{96}F$
goto(q_0,%)	%·	$q_{97}F$
goto(q_0,{)	{·	$q_{98}F$
goto(q_0,})	}·	$q_{99}F$
goto(q_0,()	(·	$q_{100}F$
goto(q_0,))	)·	$q_{101}F$
goto(q_0,[)	[·	$q_{102}F$
goto(q_0,])	]·	$q_{103}F$
goto(q_0,;)	;·	$q_{104}F$
goto(q_0,„)	,·	$q_{105}F$
goto(q_0,.)	.·	$q_{106}F$

Goto	**Elementos básicos**	**Estado**
goto(q_0,[_a-zA-Z])	[_a-zA-Z] ·[_a-zA-Z0-9]{0, 30} [_a-zA-Z][_a-zA-Z0-9]{0, 30} ·	$q_{107}F$
goto(q_{107},[_a-zA-Z0-9])	[_a-zA-Z] ·[_a-zA-Z0-9]{0, 29} [_a-zA-Z][_a-zA-Z0-9]{0, 30} ·	$q_{108}F$
goto($q_{..}$, [_a-zA-Z0-9])	...	...
goto(q_{137},[_a-zA-Z0-9])	[_a-zA-Z][_a-zA-Z0-9]{0, 0} ·	$q_{138}F$
goto(q_0,')	'·[^']'	q_{139}
goto(q_{139},[^'])	'[^'].'	q_{140}
goto(q_{140},')	'[^']'·	$q_{141}F$
goto(q_0, [-+])	$[-+]? \cdot [0-9]^+$ [-+]? ·[0 − 9]*.[0-9]{1,7} [-+]? ·[0 − 9]*.[0-9]{8,16}	q_{142}
goto(q_{142}, [0-9])	$[-+]?[0-9]^+\cdot$ $[-+]? \cdot [0-9]^+$ [-+]? [0 − 9]*·.[0-9]{1,7} [-+]? ·[0 − 9]*.[0-9]{1,7} [-+]? [0 − 9]*·.[0-9]{8,16} [-+]? ·[0 − 9]*.[0-9]{8,16}	$q_{143}F$
goto(q_{143}, [0-9])	$q_{143}F$	
goto(q_{143}, .)	[-+]? [0 − 9]*.· [0-9]{1,7} [-+]? [0 − 9]*.·[0-9]{8,16}	$q_{144}F$
goto(q_{144}, [0-9])	[-+]? [0 − 9]*.[0-9]{1,7}· [-+]? [0 − 9]*.·[0-9]{1,6} [-+]? [0 − 9]*.·[0-9]{8,15}	$q_{145}F$
goto($q_{..}$, [0-9])	...	...
goto(q_{149}, [0-9])	[-+]? [0 − 9]*.[0-9]{0,0}· [-+]? [0 − 9]*.·[0-9]{1,9}	$q_{150}F$
goto(q_{150}, [0-9])	[-+]? [0 − 9]*.[0-9]{1,8}·	$q_{151}F$
goto($q_{..}$, [0-9])	...	...
goto(q_{157}, [0-9])	[-+]? [0 − 9]*.[0-9]{0,0}·	$q_{158}F$
goto(q_0,\)	\· ([^([0-9] | [a-zA-Z] | [\n\t])] | [(){}])	$q_{159}F$
goto(q_{159},{especial})	\([^([0-9] | [a-zA-Z] | [\n\t])] | [(){}]) ·	$q_{160}F$
goto(q_{159},[(){}])	\([^([0-9] | [a-zA-Z] | [\n\t])] | [(){}]) ·	$q_{161}F$

1

[1] {especial} = [^([0-9] | [a-zA-Z] | [\n\t])] | [(){}])

3 Definición dirigida por sintáxis

programa → decl funciones

decl → tipo

decl → ϵ

tipo → VOID
tipo.code = ”0”

tipo → CHAR
tipo.code = ”1”

tipo → INT
tipo.code = ”2”

tipo → FLOAT
tipo.code = ”3”

tipo → DOUBLE
tipo.code = ”4”

tipo → STRUCT

tipo.code = ”5”

sentencias → sentencias$_1$ sentencia

sentencias.code = sentencias$_1$.code || sentencia.code

sentencias → sentencia

sentencias.code = sentencia.code

sentencia → if (condicion) sentencia$_1$ sentif

sentencias.code = gen(get_first(condicion.trues), ”:”) || condicion.code ||
gen(get_first(condicion.falses), ”:”)
backpatch(condicion.trues,newLabel());
backpatch(condicion.falses,newLabel());

sentencia → while (condicion) sentencia$_1$

sentencia.code = label(label) || gen(label, ”:”) || condicion.code ||
gen(get_first(condicion.trues), ”:”)
backpatch(condicion.trues,newLabel()) sentencia.code += sentencia$_1$.code ||
gen(”goto”,pop_label(lcontrol)) ||
gen(get_first(condicion.falses),”:”)
backpatch(condicion.falses,newLabel())

sentencia → do sentencia$_1$ while (condicion);

sentencia.code = label(label) || gen(label, ”:”) ||
push_label(lcontrol,label)
sentencia.code += sentencia$_1$.code || condicion.code || label(labe2)
backpatch(condicion.trues,newLabel());
backpatch(condicion.falses,label2); sentencia.code += gen(label2, ”:”)

sentencia → for (sentencia$_1$; condicion; sentencia$_2$) sentencia$_3$

sentencia.code = sentencia_1.code || label(label) || gen(label + ":")
push_label(lcontrol, label);
sentencia.code += gen(get_first(condicion.trues), ":") || sentencia_2.code ||
gent(get_first(condicion.trues), ":")
backpatch(condicion.trues, newLabel());
backpatch(condicion.falses, newLabel());

sentencia → parte_izq = expresion ;

sentencia.code = asignar(parete_izq,expresion)

sentencia → return expresion ;

sentencia.code = gen ("ret", expresion.dir)

sentencia → return ;

No genera código intermedio.

sentencia → **{** sentencias$_1$ **}**

sentencia.code = sentencias$_1$.code

sentencia → switch (expresion) **{** casos predeterm **}**

sentencia.code = gen("goto", get_top_label_previous(lcontrol)) ||
gen(get_top_label_previous (lcontrol), ":") || label(switchesback.pop())||
arg.code || gen("iff" + arg.code + "goto" + switchesback.pop()) ||
gen(lcontrol.pop_label(),":")
pop_label(lcontrol)

sentencia → break ;

No genera código intermedio.

sentencia → print expresion ;

i = newIndex();
sentencia.code = create_list(i)

sentif → else sentencia

No se implementó.

var_arreglo → id [expresion]

var_arreglo.arr = ID
base_dir = newExp()
base_dir.type = 2
base_dir.dir = get_dir(ID)
base_type = get_base(get_type(ID));
var_arreglo.type = base_type;
tam_act = newExp();
tam_act.type = 2;
tam_act.dir = get_tam(base_type)
curr_dir = math_function(expresion,tam_act,*)
var_arreglo.dir = math_function(base_dir,curr_dir,+).dir)
var_arreglo.code = base_dir.code || base_type.code || tam_act.code || math_function(expresion,tam_act,*) || math_function(base_dir,curr_dir,+)

var_arreglo → var_arreglo$_1$ [expresion]

---

base_dir = newExp()
base_dir.type = 2
base_dir.dir = var_arreglo$_1$.dir
base_type = get_base(var_arreglo$_1$);
var_arreglo.type = base_type;
tam_act = newExp();
tam_act.type = 2;
tam_act.dir = get_tam(base_type)
curr_dir = math_function(expresion,tam_act,*)
var_arreglo.dir = math_function(base_dir,curr_dir,+).dir)
var_arreglo.code = base_dir.code || base_type.code || tam_act.code || math_function(expresion,tam_act,*) || math_function(base_dir,curr_dir,+)

expresion → expresion$_1$ + expresion$_2$

---

resultado = math_function(expresion$_1$, expresion$_2$,+)
expresion.code = expresion$_1$ || expresion$_2$ || resultado.code

expresion → expresion$_1$ − expresion$_2$

---

resultado = math_function(expresion$_1$, expresion$_2$,−)
expresion.code = expresion$_1$ || expresion$_2$ || resultado.code

expresion → expresion$_1$ ∗ expresion$_2$

---

resultado = math_function(expresion$_1$, expresion$_2$,∗)
expresion.code = expresion$_1$ || expresion$_2$ || resultado.code

expresion → expresion$_1$ / expresion$_2$

---

resultado = math_function(expresion$_1$, expresion$_2$,/)
expresion.code = expresion$_1$ || expresion$_2$ || resultado.code

expresion → expresion$_1$ % expresion$_2$

resultado = math_function(expresion$_1$, expresion$_2$,%)
expresion.code = expresion$_1$ || expresion$_2$ || resultado.code

expresion → var_arreglo

id = newExpr()
id.dir = var_arreglo.arr
resultado = asignar(id,var_arreglo)
expresion.code = id.code || var_arreglo.code || resultado.code

expresion → CAR

car = newExpr()
car.type = 1
car.dir = CAR
expresion.code = car.code

expresion → CADENA

cadena = newExpr()
cadena.type = 6
cadena.dir = CADENA
expresion.code = car.code

expresion → ID (parametros)

id = newExpr()
id.dir = ID.dir;
id.type = get_type(curr_env.symbols,ID) || get_rec(global_funcs, ID)
expresion.code = id.code

condicion → condicion$_1$ || condicion$_2$

condicion$_1$.true = condicion.true
condicion$_1$.false = newLabel()
condicion$_2$.true = condicion.true
condicion$_2$.false = condicion.false
condicion.code = condicion$_1$.code || label(condicion$_1$.false) || condicion$_2$.code

condicion → condicion$_1$ && condicion$_2$

condicion$_1$.true = newLabel()
condicion$_1$.false = condicion.false
condicion$_2$.true = condicion.true
condicion$_2$.false = condicion.false
condicion.code = condicion$_1$.code || label(condicion$_1$.true) || condicion$_2$.code

condicion → ! condicion$_1$

condicion$_1$.falses = condicion$_1$.trues
condicion$_1$.trues = condicion$_1$.falses
condicion.code = condicion$_1$.code

condicion → (condicion$_1$)

condicion.code = condicion$_1$.code

condicion → condicion$_1$ relacional condicion$_2$

condicion.code = condicion$_1$.code || condicion$_2$.code ||
gen(”iff”, condicion$_1$.dir, relacional, condicion$_2$.dir, condicion.true) ||
gen(”goto”,condicion.false)

condicion → TRUE

condicion.code = gen(’goto’, condicion.true);

condicion → FALSE

condicion.code = gen('goto', condicion.false);

relacional.code → MAYOR

relacional.code = GT;

relacional → MENOR

relacional.code = LT;

relacional → MAYOR$_{I}GUAL$

relacional.code = GE;

relacional → MENOR$_{I}GUAL$

relacional.code = LE;

relacional → NE

relacional.code = NE;

relacional → IGUAL

relacional.code = EQ;

lista → lista$_1$ COM ID arreglo

lista.code = lista$_1$.code || arreglo.code || gen(top.push(id.lexema, tamaño, dir, current$_{t}ype$))

lista COM ID arreglo

lista.code = arreglo.code || gen(top.push(id.lexema, tamaño, dir, current$_t ype$))

lista → ID arreglo

lista.code = gen(top.push(id.lexema, tamaño, dir, tipo))

arreglo → LCOR NUMERO RCOR arreglo$_1$

arreglo.dimensiones = dimensiones.append(NUMERO.val);

arreglo → LCOR NUMERO RCOR arreglo$_1$

No genera cófigo intermedio ni acciones.

funciones → FUNC tipo ID LPAR argumentos RPAR LKEY decl sentencias RKEY funciones

funciones.code = argumentos.code || decl.code || sentencias.code || funciones.code || gen(top.push(ID.lexema, tamaño, dir, tipo));
funciones.decls = funs_decls.append(ID.lexema, tipo, dir);

funciones → ϵ

No genera código intermedio.

argumentos → lista_args

argumentos.code = lista_args.code

argumentos → ϵ

No genera código intermedio.

lista_args → lista_args COM tipo ID parte_arr

lista_args.code = lista_args.code || parte_arr.code || gen('param' ID.dir)

lista_args → ID parte_arr

lista_args.code = parte_arr.code || gen('param' ID.dir)

parametros → lista_param

parametros.val = true;

parametros → ϵ

parametros.val = false;

lista_param → lista_param COM expresión

lista_param.code = lista_param.code || expresión.code;

lista_param → expresión

lista_param.code = expresión.code;

4 Esquema de Traducción

programa → decl funciones

---

programa → decl {
my_env = stack_peek(envs);
global_symbols = my_env.symbols;
global_types = my_env.types;
} funciones

decl → tipo

---

decl → tipo {
current_type = tipo;
current_dim = get_tam(global_types,tipo);
}
lista ; decl

decl → ϵ

---

decl → ϵ

tipo → VOID

---

tipo → VOID { tipo = 0; }

tipo → CHAR

---

tipo → CHAR { tipo = 1; }

tipo → INT

---

tipo → INT { tipo = 2; }

tipo → FLOAT

tipo → FLOAT { tipo = 2; }

tipo → DOUBLE

tipo → DOUBLE { tipo = 4; }

tipo → STRUCT

```
tipo → STRUCT {
                    struct_decl = true;
                    add_context(false);
                    } { decl } {
                    tipo = 5;
                    del_context(false);
                    }
```

argumentos → lista_args

argumentos →

argumentos → ϵ

argumentos → ϵ

lista_args → lista_args , tipo ID parte_arr

```
lista_args →
                {
                  curr_env = stack_peek(envs);
                  if (depth_search(curr_env.symbols, ID) == -1 ) {
                      insert_sym(ID, curr_env, tipo);
                      num_params + = 1;
                }}
```

lista_args → tipo id parte_arr

---

lista_args →
```
{
  curr_env = stack_peek(envs);
  if (depth_search(curr_env.symbols, ID) == -1 ) {
     insert_sym(ID, curr_env, tipo);
     num_params + = 1;
}}
```

parte_arr → [] parte_arr

---

parte_arr →

parte_arr → ϵ

---

parte_arr → ϵ

arreglo → [NUMERO] arreglo

---

arreglo → [NUMERO] arreglo
```
{
  if(NUMERO.val == 2) {
     current_dim ∗ = num;
     list_append(dimensiones, NUMERO.val);
}}
```

arreglo → ϵ

---

arreglo → ϵ

lista → lista , ID arreglo

```
lista → lista, ID arreglo
                {
                  curr_env = stack_peek(envs);
                  if (depth_search(curr_env.symbols, ID) == -1
                  && current_type!= 0 && current_type!= 5 ) {
                  symbol = newSymbol();
                  symbo.id = ID;
                  symbol.type = current_type;
                  symbol.dir = dir;
                  dir += current_dim;
                  stack_pop(envs,curr_env);
                  curr_tam = get_tam(global_types,current_type);
                  primero = true;
                  while( list_size(dimensiones)) {
                     temp = list_head(dimensiones,1);
                     tipo = newType();
                     if ( primero ) {
                        tipo.base = symbol.type;
                        primero = false;
                      }
                     else tipo.base = curr_env.types.count -1;
                     tipo.dim = temp;
                     curr_tam *= tipo.dim;
                     tipo.tam = curr_tam;
                    insert_type(curr_env.types,tipo);
                     symbol.type = curr_env.types.count -1;
                   }
                 insert_type(curr_env.symbols,symbol);
                  stack_push(envs,curr_env);
                  if (current_type != 5 && struct_decl)
                     { struct_dim += current_dim; }
               }}
```

lista → ID arreglo

```
lista → ID arreglo
                {
                  curr_env = stack_peek(envs);
                  symbol = newSymbol();
                  if (depth_search(curr_env.symbols, ID) == -1
                  && current_type!= 0 ) {
                  if (current_type == 5) {
                     symtab = curr_env.symbols;
                     symbol.struct_content = symtab;
                     del_context(false);
                   }
                  symbol.id = ID;
                  symbol.type = current_type;
                  symbol.dir = dir;
                  dir += current_dim;
                  stack_pop(envs,curr_env);
                  curr_tam = get_tam(global_types,current_type);
                  primero = true;
                  while( list_size(dimensiones)) {
                     temp = list_head(dimensiones,1);
                     tipo = newType();
                     if ( primero ) {
                        tipo.base = symbol.type;
                        primero = false;
                      }
                     else tipo.base = curr_env.types.count -1;
                     tipo.dim = temp;
                     curr_tam *= tipo.dim;
                     tipo.tam = curr_tam;
                    insert_type(curr_env.types,tipo);
                     symbol.type = curr_env.types.count -1;
                   }
                 insert_type(curr_env.symbols,symbol);
                  stack_push(envs,curr_env);
                  if (current_type != 5 && struct_decl)
                     { struct_dim += current_dim; }
               }}
```

funciones → fun tipo ID (argumentos) **{** decl sentencias **}** funciones

---

```
funciones → fun tipo ID (
            {
              funciones.code = ID + ":";
              curr_env = stack_peek(envs);
              if (depth_search(curr_env.symbols, ID) == -1 ) {
                 add_context(true);
                 return_type = tipo;
                 fun_decl = true;
               }
            } argumentos
            {
              fun_env = stack_peek(envs);
              func_context = fun_env.symbols;
              reg = newFunc();
              reg.id = ID;
              reg.context = func_context;
              reg.counter = 0;
              insert_fun(global_funcs, reg);
              func_tam = dir;
              del_context(false);
              curr_env = stack_peek(envs);
              symbol = newSymbol();
              symbol.id = ID;
              symbol.type = tipo;
              symbol.dir = dir;
              dir += func_tam;
              insert(curr_env.symbols, symbol);
              stack_push(envs, curr_env);
              global_symbols = curr_env.symbols;
              func_decl = false;
             num_params = 0;
              }
            } ) { decl sentencias } funciones
```

funciones → ϵ

---

funciones → ϵ

sentencias → sentencias$_1$ sentencia

---

sentencias → sentencias$_1$ sentencia

sentencias → sentencia

---

sentencias → sentencia

sentencia → if (condicion) sentencia$_1$ sentif

---

sentencias → if (condicion)
{
  sentencia.code = get_first(condicion.trues) + ":";
} sentencia$_1$
{
  sentencia.code += get_first(condicion.falses) + ":";
  label1 = newLabel();
  label2 = newLabel();
  backpatch(condicion.trues, label1);
  backpatch(condicion.falses, label2);
} sentif { // No se terminó de implementar. }

sentencia → while (condicion) $sentencia_1$

```
sentencia → while (
            {
              label = newLabel();
              sentencia.code = label + ":";
              push_label(lcontrol, label);
            } condicion
            {
              sentencia.code += get_first(condicion.trues) + ":";
              push_label(lcontrol, label);
              label1 = newLabel();
              backpatch(condicion.trues, label1);
            } ) sentencia1
              sentencia.code += "goto" + pop_label(lcontrol);
              sentencia.code += get_first(condicion-¿falses) + ":" ;
              label2 = newLabel();
              backpatch(condicion.falses, label2);
            }
```

sentencia → do $sentencia_1$ while (condicion);

```
sentencia → do
            {
              label = newLabel();
              sentencia.code = label + ":";
              push_label(lcontrol, label);
            } sentencia1 while ( condicion );
              label1 = newLabel();
              label2 = newLabel();
              backpatch(condicion.trues, label1);
              backpatch(condicion.falses, label2);
              sentencia.code += label2 + ":";
            }
```

sentencia → for (sentencia$_1$; condicion; sentencia$_2$) sentencia$_3$

```
sentencia → for ( sentencia1 ;
            {
              label = newLabel();
              sentencia.code = label + ":";
              push_label(lcontrol, label);
            } condicion;
            {
              sentencia.code += get_first(condicion.trues) + ":";
            } sentencia2 ) sentencia3
            {
              sentencia.code += get_first(condicion.trues) + ":";
              label1 = newLabel();
              label2 = newLabel();
              backpatch(condicion.trues, label2);
              backpatch(condicion.falses, label1);
            }
```

sentencia → parte_izq = expresion ;

```
sentencia → parte_izq = expresion ;
            {
              call_params = 0;
              compatible = max_type(parte_izq.type, expresion.type);
              if(compatible != -1) {
                  sentencia.code = asignar(parete_izq,expresion);
              }
            }
```

sentencia → return expresion ;

```
sentencia → return expresion ;
            {
              if(return_type ! = 0 && return_type < 5) {
                  sentencia.code = "ret" + expresion.dir;
              }
            }
```

sentencia → return ;

---

sentencia → return ;
{ vertify_type(return_type,0); }

sentencia → **{** sentencias$_1$ **}**

---

sentencia → **{** sentencias **}**
{
sentencia = sentencias$_1$;
label = newLabel();
backpatch(sentencia$_1$, label);
}

sentencia → switch (expresion) **{** casos predeterm **}**

```
sentencia → switch ( expresion )
            {
              push_label(lcontrol,newLabel());
              push_label(lcontrol,newLabel());
              sentencia.code = "goto" + get_top_label_previous(lcontrol));
            } { casos predeterm }
            {
              sentencia.code += get_top_label_previous(lcontrol) + ":";
              while(stack_size(switchesback) > 0) {
                  sw = stack_peek(switchesback);
                  if (sw.link != predeterm ) break;
                  stack_pop(switchesback, sw);
                  arg = expresion.dir + ".caso";
                  sentencia.code += "iff" + arg + "goto" + sw.label;
               }
              sentencias.code += pop_label(lcontrol) + ":";
             pop_label(lcontrol);
            }
```

sentencia → break ;

---

sentencia → break ;

sentencia → print expresion ;

---

sentencia → print expresion ;
{
i = newIndex();
sentencia = create_list(i);
}

sentif → else sentencia

---

sentencias → else sentencia //No se implementó.

casos → case : NUMERO sentencias $casos_1$

---

casos → case : NUMERO
{
index = newLabel();
casos.code = index + ":";
sw = newSwitch();
sw.label = label;
sw.caso = NUMERO.val;
stack_push(switchesback,sw);
} sentencias
{
label = newLabel();
casos.code += "goto" + label;
} $casos_1$
{
casos = $casos_1$
}

casos → ϵ

casos → ϵ { casos = switch_call;}

predeterm → default : sentencia

predeterm → default : sentencia

predeterm → ϵ

predeterm → ϵ

parte_izq → ID

```
parte_izq → ID
            {
              curr_env = stack_peek(envs);
              if (depth_search(curr_env.symbols, ID) != -1 ) {
                  parte_izq.dir = ID;
                  parte_izq.type = get_type(curr_env.symbols, ID);
            }}
```

parte_izq → var_arreglo

```
parte_izq → var_arreglo
            {
              id = newExp();
              id.type = var_arreglo.type;
              parte_izq = asignar(id,var_arreglo);
            }
```

parte_izq → ID_1 . ID_2

---

parte_izq → ID_1 . ID_2
{
curr_env = stack_peek(envs);
if (depth_search(curr_env.symbols, ID) != -1
&& expresion.type == 5) {
struct_content = get_struct_content(curr_env.symbols,ID_1);
if (depth_search(struct_content, ID_2) != -1) {
parte_izq.dir = newTemp();
get_type(struct_content, ID_2);
}}}

var_arreglo → id [expresion]

---

var_arreglo → id [expresion]
{
curr_env = stack_peek(envs);
if (depth_search(curr_env.symbols, ID) != -1
&& expresion.type == 2) {
var_arreglo.arr = ID;
base_dir = newExp();
base_dir.type = 2;
base_dir.dir = get_dir(curr_env.symbols,ID);
curr_type = get_type(curr_env.symbols,ID);
if (curr_type != 5) {
base_type = get_base(curr_env.types,curr_type);
arr_index = expresion;
var_arreglo.type = base_type;
tam_act = newExp();
tam_act.type = 2;
tam_act.dir = get_tam(curr_env.types,base_type);
curr_dir = math_function(arr_index,tam_act,*);
var_arreglo.dir = math_function(base_dir,curr_dir,+).dir);
}}}

var_arreglo → var_arreglo$_1$ [expresion]

---

var_arreglo → var_arreglo$_1$ [expresion]
{
if (expresion.type == 2 && var_arreglo$_1$.type != -1) {
curr_env = stack_peek(envs);
base_dir = newExp();
base_dir.type = 2;
base_dir.dir = var_arreglo$_1$.dir;
curr_type = get_type(curr_env.symbols,ID);
if (curr_type != 5) {
base_type = get_base(curr_env.types,var_arreglo$_1$.type);
arr_index = expresion;
var_arreglo.type = base_type;
tam_act = newExp();
tam_act.type = 2;
tam_act.dir = get_tam(curr_env.types,base_type);
curr_dir = math_function(arr_index,tam_act,*);
var_arreglo.dir = math_function(base_dir,curr_dir,+).dir);
}}}

expresion → expresion$_1$ + expresion$_2$

---

expresion → expresion$_1$ + expresion$_2$
{ expresion = math_function(expresion$_1$, expresion$_2$,+); }

expresion → expresion$_1$ − expresion$_2$

---

expresion → expresion$_1$ − expresion$_2$
{ expresion = math_function(expresion$_1$, expresion$_2$,−); }

expresion → expresion$_1$ ∗ expresion$_2$

---

expresion → expresion$_1$ ∗ expresion$_2$
{ expresion = math_function(expresion$_1$, expresion$_2$,∗); }

expresion → expresion$_1$ / expresion$_2$

---

expresion → expresion$_1$ / expresion$_2$
{ expresion = math_function(expresion$_1$, expresion$_2$,/); }

expresion → expresion$_1$ % expresion$_2$

---

expresion → expresion$_1$ % expresion$_2$
{ expresion = math_function(expresion$_1$, expresion$_2$,%); }

expresion → var_arreglo

---

expresion → var_arreglo
{
id = newExpr();
id.dir = var_arreglo.arr;
expresion = asignar(id,var_arreglo);
}

expresion → CAR

---

expresion → CAR
{
expresion.type = 1;
expresion.dir = CAR;
}

expresion → CADENA

---

expresion → CADENA
{
expresion.type = 6;
expresion.cadena = CADENA;
}

expresion → ID (parametros)

---

expresion → ID { stack_push(func_calls, ID)}

```
                ( parametros ) {
                  curr_env = stack_peek(envs, curr_env)
                  if (depth_search(curr_env.symbols, ID) != -1) {
                        if(is_function(global_funcs, ID))
                         { rec = get_rec(global_funcs, ID) }
                      stack_pop(func_calls, curr_function);
                      expresion.dir = ID.dir;
                      expresion.type = get_type(curr_env.symbols,ID);
               }}
```

condicion → condicion$_1$ || condicion$_2$

---

condicion → condicion$_1$ ||

```
                {
                  condicion =
                  condicion.code = get_first(condicion1.falses) + ":";
               } condicion2 {
                  label = newLabel();
                  backpatch(condicion1.falses, label);
                  condicion.trues = merge(condicion1.trues, condicion2.trues);
                  condicion.falses = condicion2.falses;
               }
```

condicion → condicion$_1$ && condicion$_2$

---

condicion → condicion$_1$ &&
{
condicion =
condicion.code = get_first(condicion$_1$.trues) + ":";
} condicion$_2$ {
label = newLabel();
condicion.falses = merge(condicion$_1$.falses, condicion$_2$.falses);
condicion.trues = condicion$_2$.trues;
backpatch(condicion$_1$.trues, label);
}

condicion → ! condicion$_1$

---

condicion → ! condicion$_1$
{
condicion.falses = condicion$_1$.trues;
condicion.trues = condicion$_1$.falses;
}

condicion → (condicion$_1$)

---

condicion → (condicion$_1$)
{
condicion = condicion$_1$;
}

condicion → condicion$_1$ relacional condicion$_2$

---

condicion → condicion$_1$ relacional condicion$_2$
{
condicion = gen_cond_rel(condicion$_1$, condicion$_2$, relacional);
}

condicion → TRUE

condicion → TRUE
{
dir = newIndex();
condicion.dir = dir; condicion.trues = create_list(dir);
condicion.code = gen_cond_goto(dir);
}

condicion → FALSE

condicion → FALSE
{
dir = newIndex();
condicion.dir = dir; condicion.falses = create_list(dir);
condicion.code = gen_cond_goto(dir);
}

relacional → >

relacional → > { relacional = GT;}

relacional → <

relacional → < { relacional = LT; }

relacional → >=

relacional → >= { relacional = GE; }

relacional → <=

relacional → <= { relacional = LE; }

relacional → ! =

relacional → ! = { relacional = NE; }

relacional → ==

relacional → == { relacional = EQ; }