SEP SEV DGEST DITD

INSTITUTO TECNOLÓGICO SUPERIOR DE TANTOYUCA

INGENIERIA EN SISTEMAS COMPUTACIONALES

ASIGNATURA:
Lenguajes y autómatas II

PROYECTO:
Mini-lenguaje

DOCENTE:
M.C.C. Manuel Hernández Hernández

INTEGRANTES:	**N. Control**
Luis Fernando Juárez Pérez	143S0156
Miguel Eduardo Silvano Martínez	143S0118

GRUPO:
S6/A

Lunes 15 de agosto de 2017

MANUAL DE USUARIO

Índice

Descripción general del mini-lenguaje de programación 4
Tipo de paradigma a implementar 4
Interprete 4
Componentes léxicos 5
Diagrama de estados 6
Gramática libre de contexto en notación BNF 7
Reglas léxicas 8
Reglas semánticas 10
Ventana principal 12
Uso del sistema 12
Creación de una nueva clase 13
Declaración y asignación de variables 13
Instrucciones para imprimir un mensaje 14
Ciclos 14
Condiciones 15

Descripción general del mini-lenguaje de programación

Tipo de paradigma a implementar: *Programación estructurada*

Se opto por este tipo de programación debido a que su tipo de paradigma es familiar para nosotros ya hemos manejado cierto tipo de estructura en los datos, anteriormente, además de ciertas ventajas que posee, como lo son: En la estructura de los programas es clara ya que las condiciones están más ligadas o relacionadas entre si, los programas son más fáciles de entender y pueden ser leídos de forma secuencial, reduce las prueba y depuración en los programas.

Interprete:

Iintérprete o interpretador es un programa informático capaz de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel. Los intérpretes se diferencian de los compiladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción. Estas palabras, analiza el programa fuente y lo ejecuta directamente.

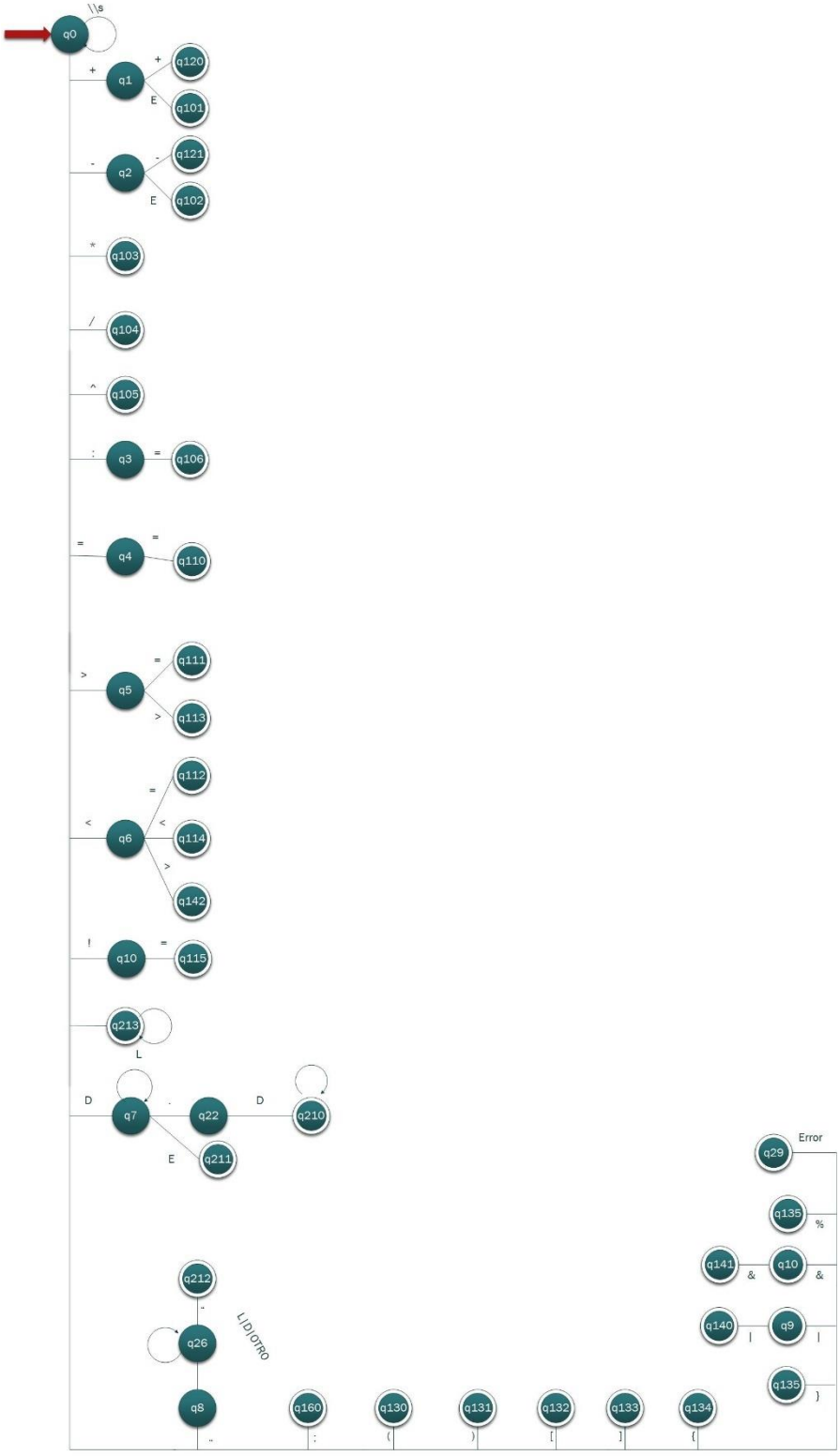
El intérprete facilita la búsqueda de errores debido a que la ejecución del programa puede ser interrumpida en cualquier momento para estudiar el entorno (variables, valores, etc.), además de que puede modificarse sin necesidad de volver a comenzar la ejecución.

permite utilizar funciones y operadores más potentes, como por ejemplo ejecutar código contenido en una variable en forma de cadenas de caracteres.

Componentes léxicos

TOKEN	LEXEMA	ER	Columna1
Programa	Programa	Programa	201
inicio	inicio	Inicio	202
entero	entero	Entero	205
decimales	decimal	Decimal	206
texto	texto	Texto	207
numero_decimal	5.9	D*.D*	210
numero_entero	6	D(D)*	211
cadena	"esto es un texto"	"(L|D)*"	212
variable	temporal	L(L|D)*	213
Expresion_selectiva_si	si(4>>0) { instrucciones}	Si	214
ECS_no	no(instrucciones)	No	215
imprimirMsg	imprimir("Texto")	Imprimir	217
ECR_mientras	mientras(a<<0) {instrucciones}	mientras	220
suma	+	+	101
resta	-	-	102
multiplicacion	*	*	103
division	/	/	104
asignacion	:=	:=	106
igualdad	==	==	110
mayor igual	>=	>=	111
menor igual	<=	<=	112
mayor	>>	>>	113
menor	<<	<<	114
ParentesisApr	(	(	130
ParentesisCierr	)	)	131
CorcheteApr	[	[	132
CorcheteCierr	]	]	133
llaveApr	{	{	134
llaveCierr	}	}	135
modulo	%	%	150
Finalizacion	;	;	160

Diagrama de estados



Gramática libre de contexto en notación BNF

ProgMV	programa Nombreprog { Cuerpo }
Nombreprog	Terminal
Cuerpo	Inicio {instrucciónes }
instrucciones	instrucción Instrucciones | instrucción
instruccion	declaración | Si | mientras | imprimir | asignación |
declaración	tipo_var variable ;
asignacion	Variable := expresión ;
Expresión	valores | valores operador valores
Operador	Ope_aritmetico1 | Ope_aritmetico2
Valores	numero | variable | cadena
Si	si (condicion) { instrucciones} | si (condicion) { instrucciones} no
No	no { instrucciones}
Condición	valores ope_relacional valores
Imprimir	Imprimir (valores) ;
Mientras	mientras (condicion) {instrucciones }
Ope_relacional	<< | >> | == | >= | <= | !=
inc_dec	++ | --
Ope_aritmeticos1	* | / | %
Ope_aritmeticos2	+ | -
Variable	Terminal
Numero	terminal
Cadena	Terminal
tipo_var	entero | decimal | texto
comentario	([Comentario])

Reglas léxicas

ProgMV	programa Nombreprog { Cuerpo }
Nombreprog	Terminal
Cuerpo	Inicio {instrucciónes}
instrucciones	instrucción Instrucciones
instrucciones	instrucción
instruccion	declaración
instrucción	asignación
instrucción	si
instrucción	mientras
instrucción	imprimir
declaración	tipo_dato variable ;
asignacion	Variable := expresión ;
Expresión	valores
Expresión	| valores operador valores
Operadores	Ope_aritmetico1
Operadores	Ope_aritmetico2
Valores	numero
Valores	variable
Valores	cadena
ECS_si	si (condicion) { instrucciones}
ECS_si	si (condicion) { instrucciones} no
ECS_no	no { instrucciones}
conjunto_condiciones	condición
Condición	valores ope_relacional valores
ECR_mientras	mientras (condicion) {instrucciones }
imprimirMsg	imprimir (valores) ;
Ope_relacional	<<
Ope_relacional	>>
Ope_relacional	==
Ope_relacional	>=
Ope_relacional	<=
Ope_unario	++

Ope_unario	--
Ope_aritmetico1	*
Ope_aritmetico1	/
Ope_aritmetico2	+
Ope_aritmetico2	-
Oper_aritmetico3	^
Variable	**terminal**

Numero_entero	**terminal**
Numero_decimal	**terminal**
Cadena	**terminal**
Carácter	**Letra**
tipo_dato	Entero
tipo_dato	Decimal
tipo_dato	texto
tipo_dato	boleano
comentario	**([Comentario])**

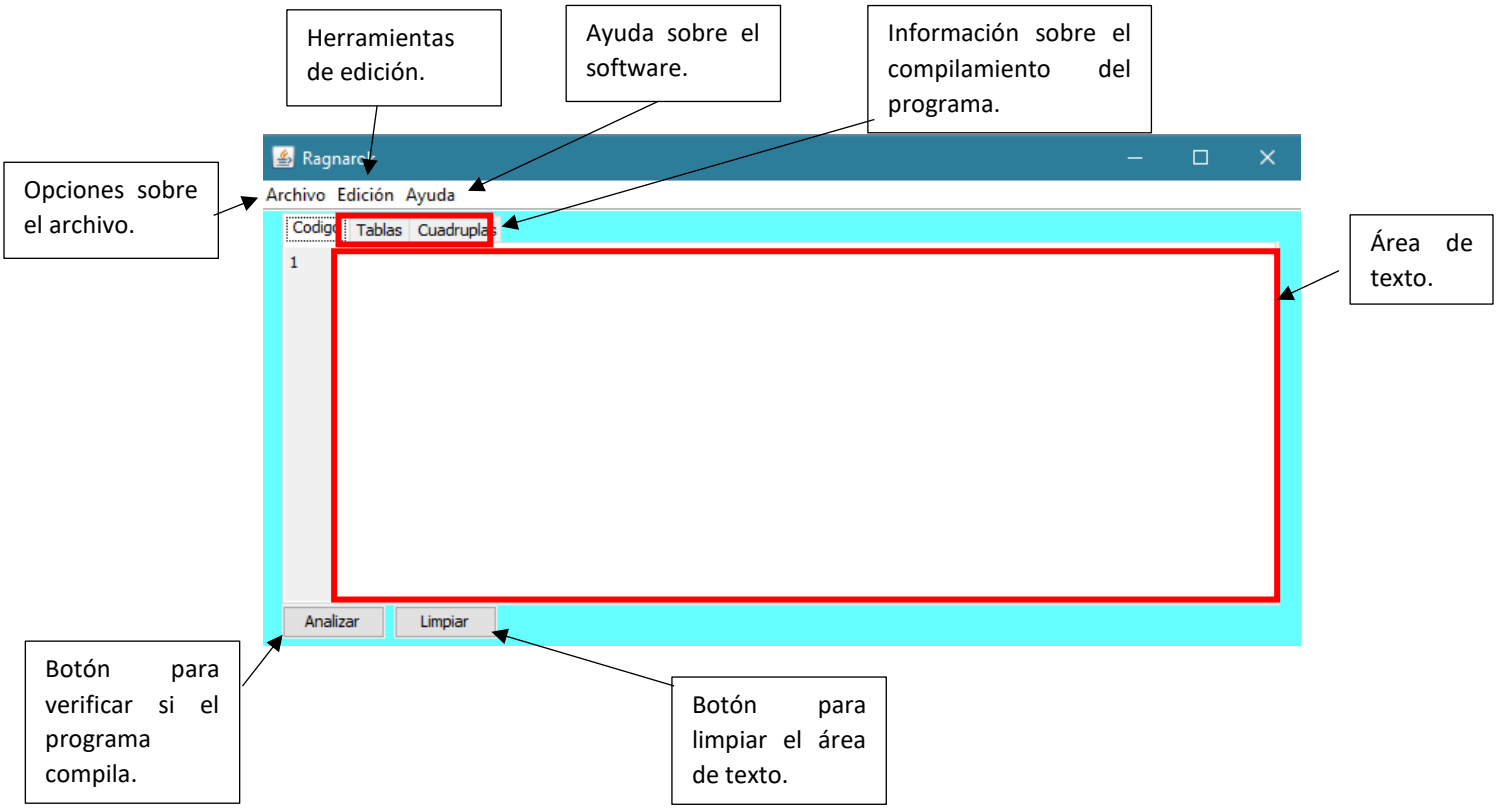
Reglas semánticas

REGLAS SINTÁCTICAS	REGLAS SEMÁNTICAS
declaración→ tipo_dato variable **;**	insertar en TS(variable.name, tipo_var.tipo)
tipo_var→ **Entero**	tipo_dato.tipo=Entero
tipo_var→ **Decimal**	tipo_dato.tipo=Decimal
tipo_var→ **Texto**	tipo_dato.tipo=Texto
variable→ id	id.name=variable.name
asignación→ Variable **:=** expresión **;**	if not type Equals(buscar en TS(variable.name),expresión.tipo) then type-error(sent)
Expresión→ valores	**contenido.tipo**
Expresión→ valores operador valores	**contenido.tipo**
Operador→ Ope_aritmetico1	
Operador→ Ope_aritmetico2	
Valores→ numero	numero.tipo =
Valores→ variable	
Valores→ cadena	
Si→ **si** (condicion) { instrucciones}	**if not type Equals(conjunto_condiciones.tipo, bolean) then type-error(sent)**
ECS_SI→ **si** (condicion) { instrucciones} no	**if not type Equals(conjunto_condiciones.tipo, bolean) then type-error(sent)**
ECS_No→ **no** { instrucciones}	
Condición→ valor operador_relacional valor	

Imprimir→ Imprimir (valores) **;**	
Mientras→ mientras (valores) {instrucciones }	**if type equeals(condición.tipo, bolean)then type-error(sent)**

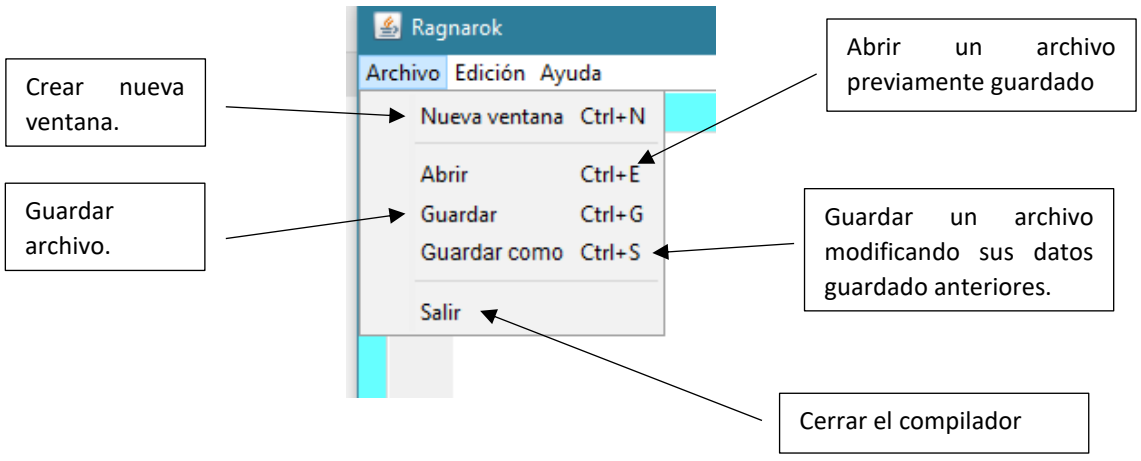
Ventana principal

La ventana principal del sistema muestra un área de texto donde puede ser probado el código característico del lenguaje y su vez las diferentes herramientas y pestañas donde puede obtenerse datos sobre la ejecución del compilador



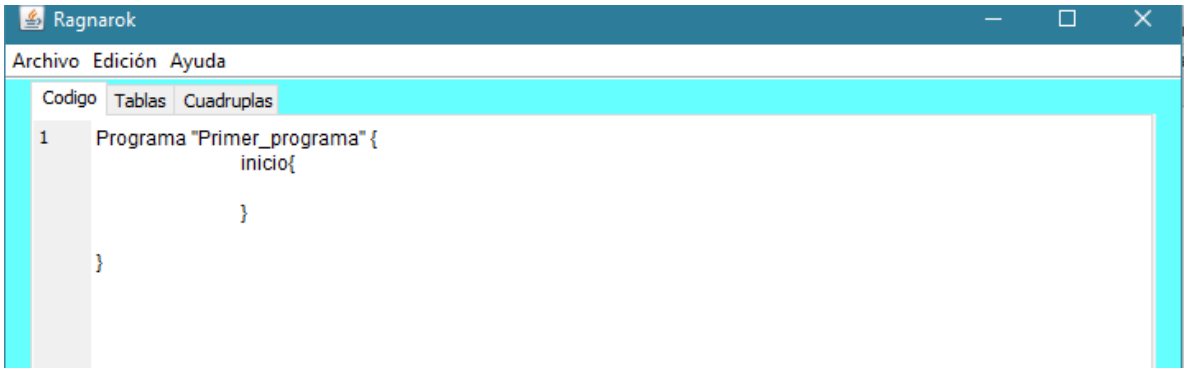
Uso del sistema

Para comenzar a usar el sistema se necesita saber cómo funciona cada opción que se tiene en la ventana principal. Para poder usar una nueva ventana o abrir un archivo es necesario localizar la pestaña de “Archivo”.



Creación de una nueva clase

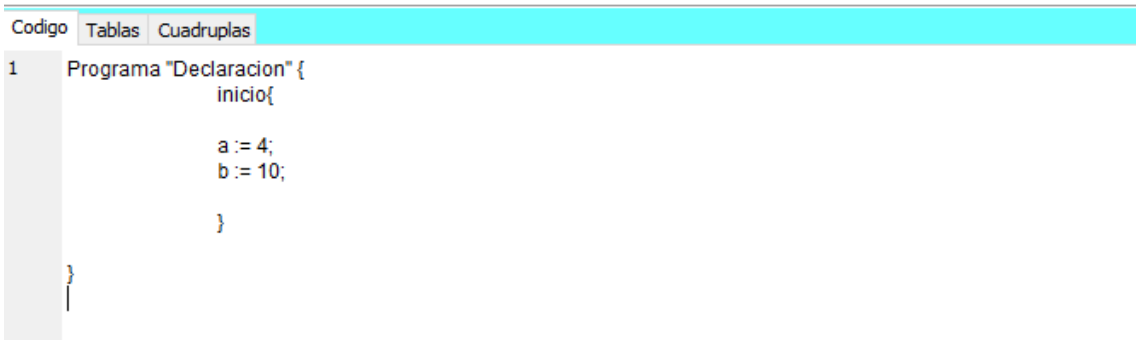
Una vez que está situado en la ventana principal puede comenzar a codificar siguiendo la estructura del lenguaje. Para esto primero se debe “crear” una clase seguido de la palabra reservada “inicio”. Por ejemplo:



Declaración y asignación de variables

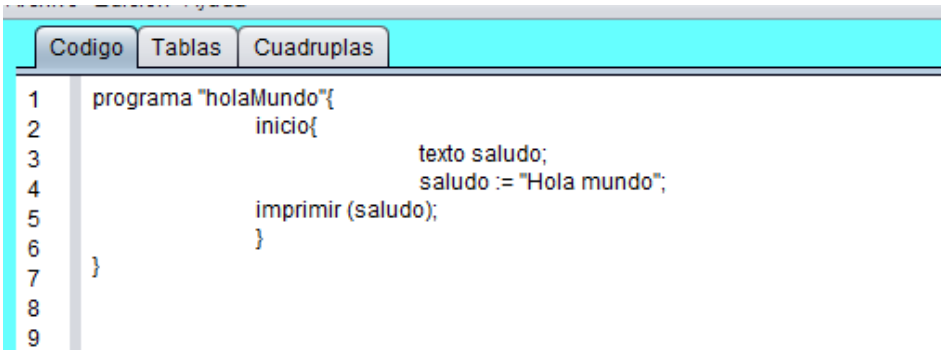
Después de haber creado la “clase” solo es necesario seguir la lista de instrucciones dedicadas propias de este lenguaje.

Un ejemplo sencillo para una declaración de variables se hace creando una “variable” donde esta puede tener o no un valor asignado. Como se muestra a continuación



Instrucciones para imprimir un mensaje

Otro ejemplo sería una asignación con una cadena donde esta podría contener un mensaje el cual podría imprimirse. Haciendo uso de la instrucción “imprimir” y un valor de tipo “Texto” donde esta contendrá una cadena de caracteres para posteriormente imprimir su mensaje.

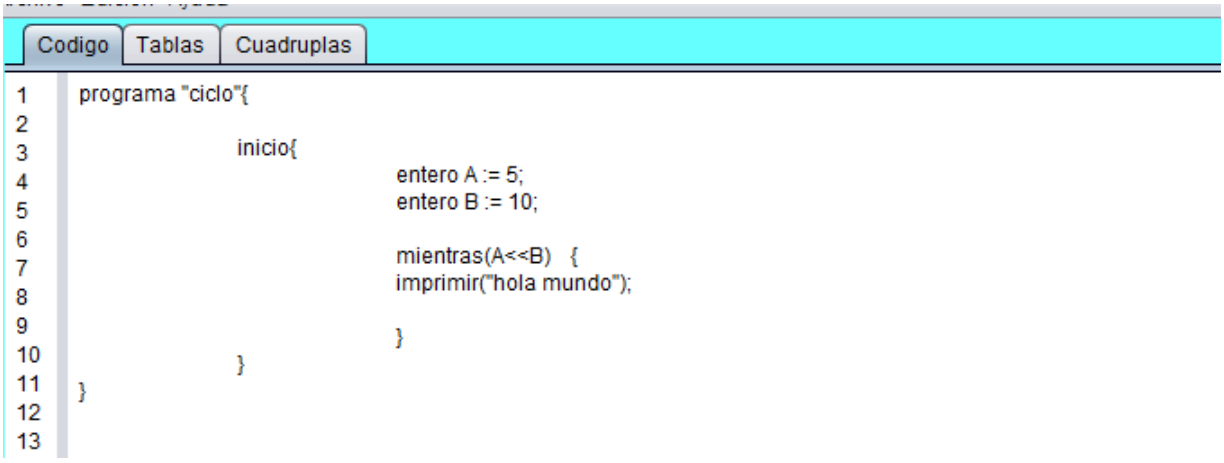


Ciclos

Algo muy usado en programación para repetir una o varias instrucciones varias veces son los ciclos. Ahora para hacer uso de un ciclo se pueden consultar las reglas sintácticas ya explicadas antes.

Para este caso para hacer uso de un ciclo se usará la instrucción “mientras”. Este ciclo es usado para declarar condiciones. Cuando es cumplida esta condición el ciclo termina y el compilador analiza las siguientes instrucciones.

En el caso del ciclo “mientras” se tienen 2 variables con sus respectivos valores. Donde este ciclo analizara si una variable es menor a la otra. Mientras esto se cumpla, deberá imprimir un mensaje.



Condiciones

Pero las condiciones no solo pueden ser usadas en los ciclos. También pueden hacerse fuera. Siendo útiles para hacer comparaciones y poder ejecutar ciertas instrucciones. En la instrucción “Si” podemos de alguna manera “preguntar” si una declaración del programa es cierta y así ejecutar las instrucciones posteriores.

Como es mostrado en el ejemplo; se pregunta si 5 es menor a 10, si esto se cumple el programa deberá imprimir un mensaje declarando que en efecto 5 es menor a 10.

```
1  programa "si numeros"{
2                  inicio{
3                                  si(5<<10){
4                                  imprimir("5 es menor que 10")
5                                  }
6                  }
7  }
8
```

Pero cuando esta expresión no puede llegar a cumplirse debe tenerse una segunda para evitar posteriores errores. Para este caso se usa la instrucción si-no. La cual hace uso de lo mencionado. Si cierta condición no se cumple entonces podrá ser comprobada otra condición que se tenga declarada.

Codigo | Tablas | Cuadruplas

```
1  programa "si"{
2                  inicio{
3                                  entero A;
4                                  A:=0;
5                                  entero B;
6                                  B:=5
7                                  si(A<<B){
8                                                  imprimir("A es menor que B");
9                                  }no{
10                                                 imprimir("A es mayor que B");
11                                 }
12                 }
13
14
15
```