Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer semestre 2020
Catedráticos: Ing. Bayron López, Ing. Edgar Saban, Ing. Erick Navarro e Ing. Luis Espino
Tutores académicos: Luis Lizama, Pavel Vásquez, Rainman Sian, Juan Carlos Maeda

Arit Software

Contenido

1 Competencias

1.1 Competencia general

Que el estudiante realice la fase de análisis de un compilador para un lenguaje de programación de alto nivel utilizando herramientas.

1.2 Competencias específicas

- Que el estudiante utilice una herramienta léxica y dos sintácticas para construir los analizadores ascendentes y descendentes.
- Que el estudiante implemente la traducción orientada por la sintaxis utilizando reglas semánticas haciendo uso de atributos sintetizados y heredados.

2 Descripción

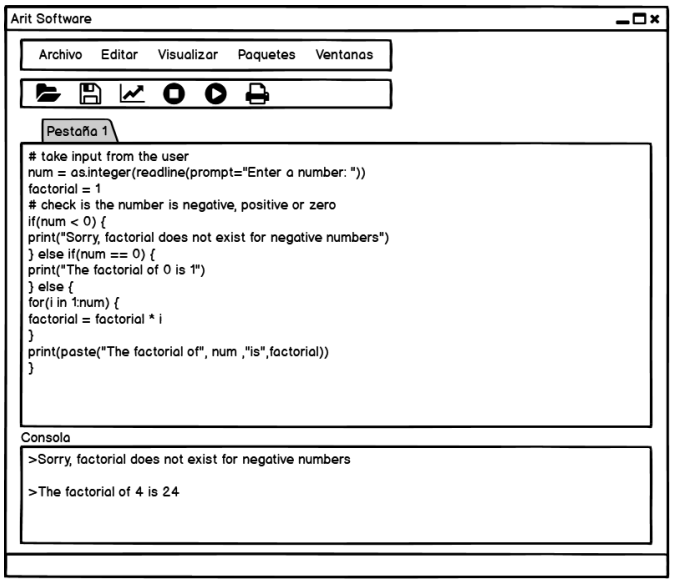
Arit Software es un entorno y lenguaje de programación con un enfoque en el análisis estadístico, el cual está formado por un conjunto de herramientas muy flexibles que pueden ampliarse fácilmente mediante paquetes, librerías o definiendo nuestras propias funciones. Este software también cuenta con un entorno de desarrollo integrado (IDE) que proporciona servicios integrales para facilitarle al programador el desarrollo de aplicaciones.

2.1 Componentes de la aplicación

Se requiere la implementación de un entorno de desarrollo que servirá para la creación de aplicaciones en lenguaje Arit. Este IDE a su vez será el encargado de analizar el lenguaje Arit. La aplicación contará con los siguientes componentes:

2.1.1 AritIDE

AritIDE es un entorno de desarrollo que provee las herramientas para la escritura de programas en lenguaje ARIT. Este IDE nos da la posibilidad de visualizar tanto la salida en consola de la ejecución del archivo fuente como los diversos reportes de la aplicación que se explican a más adelante.



2.1.1.1 Características básicas

Son las funcionalidades que un entorno de desarrollo integrado debe de tener:

- Múltiples pestañas
- Abrir, guardar y guardar como
- Numero de línea y columna del cursor
- Botón para ejecutar archivo
- Reporte de errores
- Reporte de tabla de símbolos
- Reporte de AST
- Mostrar graficas

2.1.1.2 Consola

La consola es un área especial dentro del IDE que permite visualizar las notificaciones, errores, advertencias e impresiones que se produjeron durante el proceso de análisis de un archivo de entrada.

2.2 Flujo del proyecto

A continuación, se describe el flujo general del proyecto, desde su ejecución hasta su salida en consola y presentación de reportes.

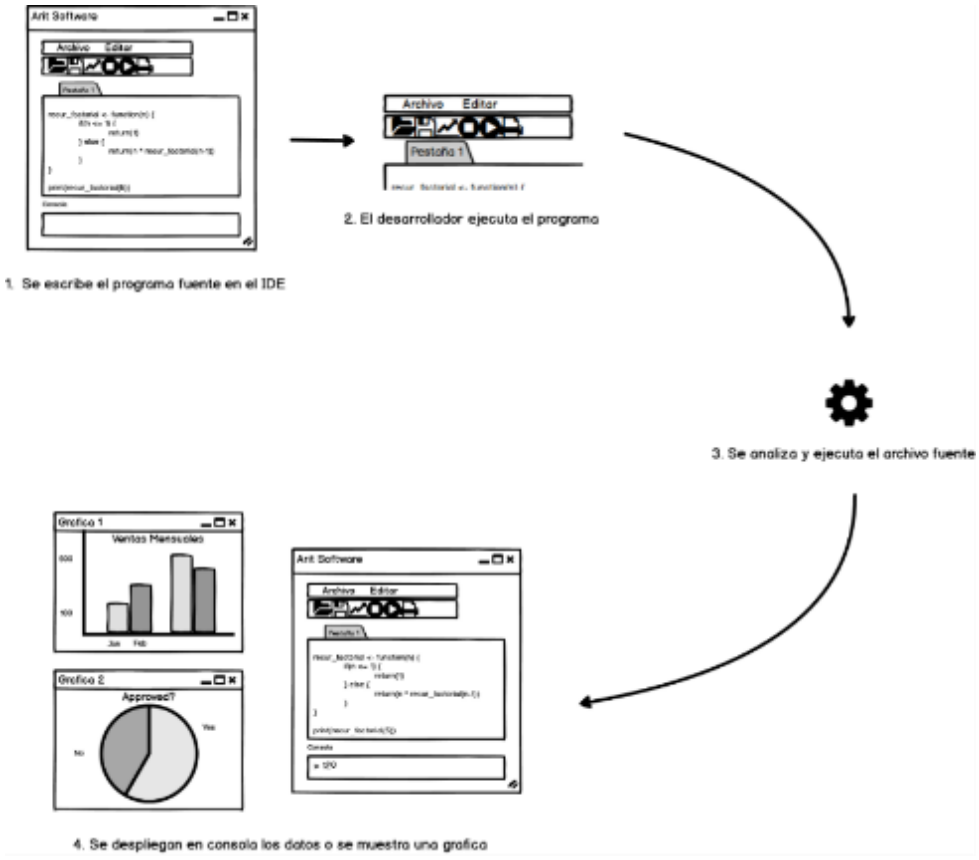
2.2.1 Flujo general de la aplicación

El flujo principal de la aplicación comienza con un archivo en lenguaje Arit y concluye con la presentación de resultados en la consola o como una gráfica según sea el caso, se detalla a continuación:

1. El programador crea su programa en lenguaje Arit a través del entorno de desarrollo.
2. El programador solicita la ejecución de su programa fuente.
3. La aplicación analiza y ejecuta el archivo de entrada.

4. Se despliegan en consola los resultados de la ejecución o se muestran los reportes según sea el caso.

Se explica el flujo anterior con la siguiente imagen:



2.3 Notación utilizada para la definición de los lenguajes

Para una mayor comprensión, se utilizará la siguiente notación.

2.3.1 Sintaxis y ejemplos

Para definir la sintaxis y ejemplos de sentencias se utilizará un rectángulo gris.

2.3.2 Asignación de colores

Dentro del enunciado se utilizará el siguiente formato de código.

Color	Token
Azul	Palabras reservadas
Naranja	Cadenas
Morado	Números
Gris	Comentarios

Negro	Otro

2.3.3 Expresiones

Cuando se haga referencia a una 'expresión', se hará referencia a cualquier sentencia que devuelve un valor. Por ejemplo:

- Una operación aritmética, de comparación o lógica
- Acceso a un variable
- Una llamada a una función

2.3.4 Cuantificadores para expresiones regulares

Cuantificador	Descripción
+	**Indica que el carácter que le precede debe aparecer al menos una vez.**
*****	**Indica que el carácter que le precede puede aparecer cero, una, o más veces.**
?	**Indica que el carácter que le precede puede aparecer cero o una vez.**

3 Generalidades del lenguaje Arit

Arit es un lenguaje derivado de R, por lo tanto, se conforma por un subconjunto de instrucciones de este, pero con la diferencia de que arit va más allá y agrega más funcionalidades a estas para facilitar la experiencia de usuario cuando se está haciendo análisis de datos, una de las características más importantes del lenguaje es que es interpretado además de ser dinámicamente tipado.

3.1 Identificadores

Un identificador será utilizado para dar un nombre a variables y métodos. Un identificador de arit está compuesto básicamente por una combinación de letras, dígitos, punto o guion bajo.

Ejemplos de identificadores válidos

```
IdValido
.id.Valido
id_valido5
```

Ejemplos de identificadores no válidos

```
_idNoValido
.5ID
TRUE
Tot@l
1d
```

Consideraciones

- El identificador debe iniciar con una letra o punto.
- Si inicia con un punto el siguiente carácter **no** debe ser un número.

3.2 Case insensitive

El lenguaje Arit será case insensitive, esto quiere decir que no diferenciará mayúsculas con minúsculas, por ejemplo, el identificador *var* hace referencia al mismo identificador *Var.* El mismo funcionamiento aplica para palabras reservadas.

3.3 Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis sintáctico de la entrada.

Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con el símbolo de "#" y al final con un carácter de finalización de línea.
- Los comentarios con múltiples líneas que empezarán con los símbolos "#*" y terminarán con los símbolos "*#".

```
# Este es un comentario de una línea

#*
 Comentario multilínea
*#
```

3.4 Valor nulo

En el lenguaje Arit se utiliza la palabra reservada **null** para hacer referencia a la nada, esto indicará la ausencia de valor.

3.5 Tipos de dato

Como se mencionó anteriormente, Arit es un lenguaje de programación de tipado dinámico, lo que significa que la comprobación de tipos se realiza durante la ejecución, por lo tanto, las variables pueden cambiar su tipo durante la ejecución.

3.6 Tipos primitivos

Se utilizarán los siguientes tipos de datos primitivos:

Tipo	Definición	Memoria requerida	Rango	Valor por defecto
Integer	**Acepta valores numéricos enteros.**	4 byte (32 bits)	**[-2^{31} , $2^{31}-1$]**	**0**
Numeric	**Acepta valores numéricos de punto flotante.**	8 byte (64 bits)	**[$\pm 1{,}7 \cdot 10^{-308}$, $\pm 1{,}7 \cdot 10^{308}$]**	**0.0**
Boolean	**Acepta valores lógicos de verdadero y falso.**	-	**true, false**	**false**

String	**Acepta cadenas de caracteres.**	**-**	[0, 65535]	**NULL**

Consideraciones

- Cualquier otro tipo de dato que no sea primitivo tomara el valor por defecto de null de ser necesario.
- Por conveniencia y facilidad de desarrollo, el tipo string será tomado como un tipo primitivo.

3.7 Tipos compuestos

Cuando hablamos de tipos compuestos nos vamos a referir a ellos como no primitivos, en estos tipos vamos a encontrar las estructuras básicas del lenguaje arit.

- Vectores
- Listas
- Matrices
- Arreglos

3.8 Secuencias de escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

Secuencia	Descripción
\”	Comilla doble
****	Barra invertida
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación

4 Sintaxis del lenguaje Arit

A continuación, se define la sintaxis para las sentencias del lenguaje arit.

4.1 Bloques de sentencias

Sera un conjunto de sentencias delimitado por llaves “{ }”, cuando se haga referencia a esto querrá decir que se está definiendo un ámbito local con todo y sus posibles instrucciones y que tiene acceso a las variables del ámbito global, además las variables declaradas en dicho ámbito únicamente podrán ser utilizadas en este ámbito o en posibles ámbitos anidados.

```
{
  # SENTENCIAS
}
```

4.2 Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis.

```
3 - (1 + 3) * 32 / 90      # 1.58
```

4.3 Variables

Una variable es un elemento de datos con nombre cuyo valor puede cambiar durante el curso de la ejecución de un programa. Una variable cuenta con un nombre y un valor, los nombres de variables no pueden coincidir con una palabra reservada.

El lenguaje arit maneja sus variables con los tipos de las estructuras, lo que significa que las variables serán estructuras.

Por lo tanto, si declaramos una variable con un tipo primitivo, este se va a convertir en un vector de un elemento, pero su tipo será el de los elementos que este contenga, o si declaramos una variable con una expresión list, el tipo de la variable será list, y así con cada estructura.

Para poder utilizar una variable se tiene que definir previamente, la declaración nos permite crear una variable y asignarle un valor.

Sintaxis

```
<identificador> = <Expresion> [;]?
```

Consideraciones

- Las variables pueden almacenar cualquier tipo de dato.
- Solo se puede declarar una variable por sentencia.
- Si la variable ya existe se debe actualizar su valor por el nuevo.
- El nombre de la variable no puede ser el nombre de una palabra reservada.

Ejemplo

```
# Declarando variables
Var1 = 5; # Esta variable pasa a ser un vector
          # el tipo de la variable es integer

Var1 = c(5, “6”); # Esta variable pasa a ser un vector
                  # el tipo de la variable es string

Var2 = Var1 * 4 – 2 # Esta variable pasa a ser un vector
                    # el tipo de la variable es integer

Var2 = list(1,2,3) # Esta variable pasa a ser una lista
                   # el tipo de la variable es list

Var2 = MATRix(c(1,2,3), 3, 4, TRUE) # Esta variable pasa a ser una matriz
                                    # el tipo de la variable es matrix
```

```
Var3 = c(5, 6) # Esta variable pasa a ser un vector
                    # el tipo de la variable es integer

If = "hola if" # Error, la variable tiene el nombre de una palabra reservada

.5abc = "hola" # Error, el identificador de la variable no es válido.
```

4.4 Operadores Aritméticos

Los operadores Aritméticos toman valores numéricos (ya sean literales o variables) como sus operandos y retornan un valor numérico único. Los operadores aritméticos estándar son adición o suma (+), sustracción o resta (-), multiplicación (*), y división (/), adicionalmente vamos a trabajar la potencia (^) y el módulo (%).

4.4.1 Suma

La operación suma se produce mediante la suma de número o strings concatenados.

Operandos	Tipo resultante	Ejemplos
Integer + Numeric **Numeric + Integer** **Numeric + Numeric**	**Numeric**	**2 + 3.3 = 5.3** **2.3 + 8 = 10.3** **1.2 + 5.4 = 6.6**
Integer + integer	**Integer**	**2 + 3 = 5**
String + integer **Integer + String** **String + numeric** **Numeric + String** **String + boolean** **Boolean + String** **String + String**	**String**	**"hola" + 4 = "hola4"** **4 + "hola" = "4hola"** **"hola" + 4.5 = "hola4.5"** **4.5 + "hola" = "4.5hola"** **"hola" + true = "holatrue"** **True + "hola" = "truehola"** **"hola"+"mundo"="holamundo"**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.4.2 Resta

La resta se produce cuando se sustraen el resultado de los operadores, produciendo su diferencia.

Operandos	Tipo resultante	Ejemplos
Integer - Numeric **Numeric - Integer** **Numeric - Numeric**	**Numeric**	**2 - 3.3 = -1.3** **2.3 - 8 = -5.7** **1.2 - 5.4 = -4.2**
Integer - integer	**Integer**	**2 - 3 = -1**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.4.3 Multiplicación

El operador multiplicación produce el producto de la multiplicación de los operandos.

Operandos	Tipo resultante	Ejemplos
Integer * Numeric **Numeric * Integer** **Numeric * Numeric**	**Numeric**	**2 * 3.3 = 6.6** **2.3 * 8 = 18.4** **1.2 * 5.4 = 6.48**
Integer * integer	**Integer**	**2 * 3 = 6**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.4.4 División

El operador división se produce el cociente de la operación donde el operando izquierdo es el dividendo y el operando derecho es el divisor.

Operandos	Tipo resultante	Ejemplos
Integer / Numeric **Numeric / Integer** **Numeric / Numeric**	**Numeric**	**2 / 3.3 = 0.60** **2.3 / 8 = 0.2875** **1.2 / 5.4 = 0.222**
Integer / integer	**Integer**	**6 / 4 = 1**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.4.5 Potencia

El operador de exponenciación devuelve el resultado de elevar el primer operando al segundo operando de potencia.

Operandos	Tipo resultante	Ejemplos
Integer ^ Numeric **Numeric ^ Integer** **Numeric ^ Numeric** **Integer ^ Integer**	**Numeric**	**2 ^ 3.5 = 9.84** **2.3 ^ 8 = 783.10** **1.2 ^ 5.4 = 2.67** **6^2 = 36.0**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.4.6 Modulo

El operador modulo devuelve el resto que queda cuando un operando se divide por un segundo operando.

Operandos	Tipo resultante	Ejemplos
Integer %% Numeric **Numeric %% Integer** **Numeric %% Numeric**	**Numeric**	**2 %% 3.5 = 2** **2.3 %% 8 = 2.3** **1.0 %% 5.0 = 1**
Integer %% integer	**Integer**	**6 %% 3 = 0**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.4.7 Negación unaria

El operador de negación unaria precede su operando y lo niega (*-1)

Operandos	Tipo resultante	Ejemplos
- Numeric	**Numeric**	**- 3.2 = -3.2**
- Integer	**Integer**	**-(-4) = 4**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.

4.5 Operadores de comparación

Compara sus operandos y devuelve un valor lógico en función de si la comparación es verdadera (true) o falsa (false). Los operadores pueden ser numéricos, Strings o lógicos.

4.5.1 Igualdad y Desigualdad

- El operador de igualdad (==) devuelve true si ambos operandos tienen el mismo valor; en caso contrario, devuelve false.
- El operador no igual a (!=) devuelve true si los operandos no tienen el mismo valor; de lo contrario, devuelve false.

Operandos	Tipo resultante	Ejemplos
Integer [==, !=] Numeric Numeric [==, !=] Integer Numeric [==, !=] Numeric Integer [==, !=] Integer String [==, !=] String Boolean [==, !=] Boolean	**Boolean**	**4 == 4.3 = false** **4.3 == 4 = false** **4.3 == 4.3 = true** **4 == 4 = true** **“hola” == “hola” = true** **True == false = false**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.
- Las comparaciones entre cadenas se hacen lexicográficamente.

4.5.2 Relacionales

- **Mayor que:** Devuelve true si el operando de la izquierda es mayor que el operando de la derecha.
- **Mayor o igual que:** Devuelve true si el operando de la izquierda es mayor o igual que el operando de la derecha.
- **Menor que:** Devuelve true si el operando de la izquierda es menor que el operando de la derecha.
- **Menor o igual que:** Devuelve true si el operando de la izquierda es menor o igual que el operando de la derecha.

Operandos	Tipo resultante	Ejemplos

Integer [>,<,>=,<=] Numeric Numeric [>,<,>=,<=] Integer Numeric [>,<,>=,<=] Numeric Integer [>,<,>=,<=] Integer String [>,<,>=,<=] String	**Boolean**	**4 < 4.3 = true** **4.3 > 4 = true** **4.3 <= 4.3 = true** **4 >= 4 = true** **"hola" > "hola" = false**

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.
- Las comparaciones entre cadenas se hacen lexicográficamente.

4.6 Operadores lógicos

Los operadores lógicos comprueban la veracidad de alguna condición. Al igual que los operadores de comparación, devuelven el tipo de dato Boolean con el valor TRUE, FALSE.

4.6.1 And, or, not

And: TRUE si ambas expresiones booleanas son TRUE.

Or: TRUE si cualquiera de las dos expresiones booleanas es TRUE.

Not: Invierte el valor de cualquier otro operador booleano.

A	B	A & B	A | B	! A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	true	false	true

Consideraciones

- Ambos operadores deben ser booleanos, sino se debe reportar el error.

4.7 Operador ternario

El operador condicional (ternario) es el único operador que tiene tres operandos. Este operador se usa con frecuencia como atajo para la instrucción if.

Si la condición es true, el operador retorna el valor de la expr1; de lo contrario, devuelve el valor de expr2.

Sintaxis

```
Condicion ? Expresion1 : Expresion2
```

Ejemplo

```
3 < 5 ? "hola" : "adios"
A = 3 < 5 ? "hola" : "adios"
```

4.8 Sentencias de control de flujo

Las estructuras de control permiten regular el flujo de la ejecución del programa. Este flujo de ejecución se puede controlar mediante sentencias condicionales que realicen ramificaciones e iteraciones.

4.8.1 Sentencia IF...Else

Ejecuta un bloque de sentencias si una condición especificada es evaluada como verdadera. Si la condición es evaluada como falsa, otro bloque de sentencias puede ser ejecutado.

Sintaxis

```
SI -> If ( condición ) <BLOQUE_SENTENCIAS>
    | If ( condicion ) <BLOQUE_SENTENCIAS> else <BLOQUE_SENTENCIAS>
    | If ( condicion ) <BLOQUE_SENTENCIAS> else SI
```

Ejemplo

```
If( 3 < 4) {
 # Sentencias
} else if (2 < 5){
 # Sentencias
} else {
 # Sentencias
}
If(true){ # Sentencias }

If(false){# Sentencias } else { # Sentencias }

If(false){# Sentencias } else if(true) { # Sentencias }
```

Consideraciones

- Puede venir cualquier cantidad de if de forma anidada.

4.8.2 Sentencia Switch

Evalúa una expresión, comparando el valor de esa expresión con un case, y ejecuta declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.

Si ocurre una coincidencia, el programa ejecuta las declaraciones asociadas correspondientes. Si la expresión coincide con múltiples entradas, la primera será la seleccionada.

Si no se encuentra una cláusula de case coincidente, el programa busca la cláusula default opcional, y si se encuentra, transfiere el control a ese bloque, ejecutando las declaraciones asociadas. Si no se encuentra un default el programa continúa la ejecución en la instrucción siguiente al final del switch. Por convención, el default es la última cláusula.

La declaración break es opcional y está asociada con cada etiqueta de case y asegura que el programa salga del switch una vez que se ejecute la instrucción coincidente y continúe la ejecución en la

instrucción siguiente. Si se omite el break el programa continúa la ejecución en la siguiente instrucción en la declaración de switch.

Sintaxis

```
switch (expresión) {
  case expr1:
    # Declaraciones ejecutadas cuando el resultado de expresión coincide con el expr1
    break[;]?
  case expr2:
    # Declaraciones ejecutadas cuando el resultado de expresión coincide con el expr2
    break [;]?
  ...
  case exprN:
    # Declaraciones ejecutadas cuando el resultado de expresión coincide con exprN
    break [;]?
  default:
    # Declaraciones ejecutadas cuando ninguno de los valores coincide con el valor de la expresión
    break [;]?
}
```

Ejemplo

```
switch (3*54) {
  case 3:
    # Sentencias
    break;
  case 5:
    # Sentencias
    break;
  case 7:
    # Sentencias
    [break;]
  default:
    # Sentencias
    break;
}
```

Consideraciones

- La lista de sentencias asociada a cada caso no debe llevar **{}**
- El caso default es opcional

4.8.3 Sentencia While

Crea un bucle que ejecuta un bloque de sentencias especificadas mientras cierta condición se evalúe como verdadera. Dicha condición es evaluada antes de ejecutar el bloque de sentencias.

Sintaxis

```
While ( condicion ) <BLOQUE_SENTENCIAS>
```

Ejemplo

```
While(false){
 Print("Hola")
}
```

4.8.4 Sentencia Do...While

Crea un bucle que ejecuta un bloque de sentencias especificadas, hasta que la condición de comprobación se evalúa como falsa. La condición se evalúa después de ejecutar las sentencias, dando como resultado que las sentencias especificadas se ejecuten al menos una vez.

Sintaxis

```
Do <BLOQUE_SENTENCIAS> while ( condicion ) [;]?
```

Ejemplo

```
Do{
  Print(2);
}while(true);
```

4.8.5 Sentencia For

Un bucle for en el lenguaje arit itera en base a una estructura, en cada iteración la variable del for cambia su valor por el correspondiente de la iteración actual, finaliza hasta que pasa por todos los elementos de la estructura.

Sintaxis

```
For( variable in <Expresion> ) <BLOQUE_SENTENCIAS>
```

Ejemplo

```
for(i in 3){
   print(i)
}
# Salida
# 3

for(i in c(1,2,3,4,5,6)){
   print(i)
}
# Salida
```

```
#1
#2
#3
#4
#5
#6

B = matrix(c(1,2),2,2,TRUE)
for(i in B){
   print(i)
}
# Salida
# 1
# 1
# 2
# 2
```

4.9 Sentencias de transferencia

Estas sentencias transferirán el control a otras partes del programa y se podrán utilizar en entornos especializados.

4.9.1 Break

Termina el bucle actual, sentencia switch y transfiere el control del programa a la siguiente sentencia a la sentencia de terminación de estos elementos.

Ejemplo

```
While(3 < 4){
  Break;
}
```

4.9.2 Continue

Termina la ejecución de las sentencias de la iteración actual del bucle actual y continua la ejecución del bucle con la próxima iteración.

Ejemplo

```
While(3 < 4){
  continue;
}
```

4.9.3 Return

Finaliza la ejecución de la función y puede especificar un valor para ser devuelto a quien llama a la función.

Ejemplo

```
factorial = (n=3) => {
   Return(n)
}

factorial2 = (n=3) => {
   Return
}
```

4.10 Funciones

En términos generales, una función es un "subprograma" que puede ser llamado por código externo (o interno en caso de recursión) a la función. Al igual que el programa en sí mismo, una función se compone de una secuencia de declaraciones, que conforman el llamado cuerpo de la función. Se pueden pasar valores a una función, y la función puede devolver un valor.

Para devolver un valor especifico, una función debe tener una sentencia return, que especifique el valor a devolver.

4.10.1 Definición de funciones

Sintaxis

```
Identificador = function( <LISTA_PARAMETROS> ) <BLOQUE_SENTENCIAS>
Identificador = (<LISTA_PARAMETROS> ) => <BLOQUE_SENTENCIAS>
```

Ejemplo

```
factorial = function(n){
   if(n <= 0){
      return(result = 1);
   }
   return(result = n*factorial(n-1));
}

factorial = (n) => {
   if(n <= 0){
      return(result = 1);
   }
   return(result = n*factorial(n-1));
}
```

Consideraciones

- No pueden existir funciones con el mismo nombre
- Las funciones y variables si pueden tener el mismo nombre

- Las funciones pueden o no retornar un valor
- El valor de retorno puede ser de cualquier tipo
- Las funciones pueden utilizar cualquiera de las 2 sintaxis aquí planteadas

4.10.2 Parámetros de las funciones

Los parámetros en las funciones son variables que podemos utilizar mientras nos encontremos en el ámbito de la función. Un aspecto importante de los parámetros es poder agregar un valor por defecto. Cuando se haga una llamada se utilizará la pablara reservado default, que nos dará a entender que queremos utilizar el valor por defecto.

Ejemplo

```
# La función ahora posee un parámetro con un valor por defecto de 3, si en la llamada no agregamos el valor del parámetro se tomará el valor de 3.
factorial = (n=3) => {
   if(n <= 0){
      return(result = 1);
   }
   return(result = n*factorial(n-1));
}
```

4.10.3 Llamada a funciones

Los parámetros en la llamada a una función son los argumentos de la función. Los argumentos se pasan a las funciones por valor. Si la función cambia el valor de un argumento, este cambio no se refleja globalmente ni en la llamada de la función.

Además, para indicar que vamos a utilizar un parámetro por defecto, utilizaremos la palabra reservada default.

Sintaxis

```
Identificador(<Lista_Expresiones>)[;]?
```

Ejemplo

```
# Llamando a una función
Factorial(5);

# Creando una función con parámetros por defecto
F1 = (param1=3, param2, param3 = 4) => {
    PRINT(param1+", "+param2+", "+param3);
}

# Llamando a una función que contiene parámetros por defecto
F1(default, 34, 5); # imprime: 3, 34, 5
```

Consideraciones

- Se debe verificar que la cantidad de parámetros coincida con los de la función que se está llamando.

4.10.4 Función print

Esta función nos permitirá imprimir en consola cualquier valor

Sintaxis

```
Print(<Expresion>)[;]?
```

Ejemplo

```
Vector1 = c(4,5,6)
Print(vector1)
# Salida: 4 5 6

Lista1 = list(c(1,5),2,3)
Print(lista1)
# Salida
# [[1]]
# 1 5
# [[2]]
# 2
# [[3]]
# 3

Matriz1 = matrix(c(1,2,3), 3, 4, FALSE)
Print(Matriz1)
# Salida
#      [,1] [,2] [,3] [,4]
# [1,]    1    1    1    1
# [2,]    2    2    2    2
# [3,]    3    3    3    3

Arreglo1 = array(c(4,5,6,7,8,9,10,11,12), c(2,2,2))
# Salida
# , , 1
#
#      [,1] [,2]
# [1,]    4    6
# [2,]    5    7
#
# , , 2
#
#      [,1] [,2]
# [1,]    8   10
```

```
# [2,]   9  11
```

Consideraciones

- El estudiante es libre de imprimir las estructuras con el formato que le parezca mejor, siempre y cuando lo que se imprimió se comprenda de la mejor forma.

4.10.5 Función c

Esta función es de las funciones más importantes del lenguaje, ya que nos ayuda a concatenar elementos de diferentes formas.

Sintaxis

```
C(<Lista_Expresiones>)
```

El valor de retorno de la función c estará dado por los elementos que este contenga, si son distintos tipos de dato se deberán castear los elementos de la siguiente forma

Tipo entrada	Tipo de salida posible	Ejemplo
Boolean	**Integer**	**True -> 1** **False -> 0**
Boolean	**Numeric**	**True -> 1.0** **False -> 0.0**
Boolean	**String**	**True -> “True”** **False -> “False”**
Integer	**Numeric**	**5 -> 5.0**
Integer	**String**	**5 -> “5”**
Numeric	**String**	**4.4 -> “4.4”**
List	**List**	

La prioridad de casteo estará dada de la siguiente forma

Tipo	Prioridad
List	**4**
String	**3**
Numeric	**2**
Integer	**1**

Siendo la más alta 4 y la menor 1

Se debe tomar en cuenta lo siguiente

- Si existe una función c, con distintos tipos de elementos y algún elemento es una lista, el resto de los elementos pasaran a ser parte de esa lista.
- Si existen varias listas, las listas se van a concatenar para formar una nueva lista.

- Si existen varios vectores, se formará un nuevo vector con los elementos de todos los vectores y casteando a los tipos correspondientes.
- Si los elementos tienen distintos tipos de datos se deberá ver cuál es el que tiene más prioridad y aplicar el tipo de casteo correspondiente.

Ejemplo

```
Vector1 = c(1,2,true, "HOLA", 4.5);
# vector1 pasa a ser de tipo string debido a que contiene un elemento de tipo string
# Contenido ["1", "2", "true", "HOLA", "4.5"]

Vector1 = c(Vector1, LIst(5,6));
# vector1 se convierte en una lista
# contenido ["1", "2", "true", "HOLA", "4.5", 5, 6]

Vector1 = c(c(4,5), list(7,8));
# vector1 se convierte en una lista
# contenido [4, 5, 7, 8]

Lista1 = list(1,2,3)
Lista2 = c("hola",4, list(9,8));

Lista3 = c(lista1, lista2);
# Lista3 se convierte en una lista
# contenido [1,2,3, "hola", 4, 9, 8]

Lista4 = c(list(c(1,2,3)),5)
# Lista4 se convierte en una lista
# contenido [(1,2,3), 5]
# Notar que la posición 1 de la lista es un vector
```

4.10.6 Función typeof

Esta función nos devuelve el tipo de alguna variable, el valor de devuelto será de tipo string

Sintaxis

```
Typeof(<expresion>)
```

4.10.7 Funciones generales para estructuras

Estas funciones nos permiten obtener ciertas características sobre las distintas estructuras existentes.

4.10.7.1 Función length

Esta función nos devuelve el tamaño de una estructura.

Sintaxis

```
length(<Expresion>)
```

Ejemplo

```
Arreglo1 = array(c(5,4,3), c(2,2,2));
Var1 = length(arreglo1); # tamaño 8
```

4.10.7.2 Función nCol

Esta función nos devuelve la cantidad de columnas de cierta matriz.

Sintaxis

```
nCol(<Expresion>)
```

Ejemplo

```
Mat1 = matrix(c(5,4,3), 3,5,FALSE)
nCol(Mat1) # devuelve 5
```

Consideraciones

- Si se manda como parámetro algo que no sea una matriz se deberá reportar el error.

4.10.7.3 Función NRow

Esta función nos devuelve la cantidad de filas de cierta matriz.

Sintaxis

```
NRow(<Expresion>)
```

Ejemplo

```
Mat1 = matrix(c(5,4,3), 3,5,FALSE)
nRow(Mat1) # devuelve 3
```

Consideraciones

- Si se manda como parámetro algo que no sea una matriz se deberá reportar el error.

4.10.8 Funciones propias del lenguaje

Arit posee un conjunto de funciones predefinidas que nos permiten dar una experiencia más agradable al usuario.

4.10.8.1 String Length

Recibe como parámetro una cadena y devuelve un entero con la longitud de la cadena.

Sintaxis

```
StringLength(<Expresion>)
```

Ejemplo

```
cant = StringLength(“Cadena”)    # cant toma el valor de 6
```

Consideraciones

- Si se envía un vector string, se debe validar que solo contenga 1 elemento, caso contrario reportar el error.

4.10.8.2 Remove

Recibe como parámetro dos cadenas. La primera indica la cadena que se quiere modificar y la segunda indica que caracteres se quieren remover.

Sintaxis

```
remove(<Expresion>,<Expresion>)
```

Consideraciones

- Si se envían 2 vectores string, se debe validar que cada vector solo contenga 1 elemento, caso contrario reportar el error.

Ejemplo

```
cad = remove (“Cadena”, “Cad”)    # cad toma el valor de “ena”
cad = remove (“Cadena”, “cad”)    # cad toma el valor de “Cadena”
```

4.10.8.3 ToLowerCase

Recibe como parámetro una cadena y devuelve una segunda cadena que contiene la palabra indicada en el parámetro con todas sus letras minúsculas.

Sintaxis

```
toLowerCase (<Expresión>)
```

Ejemplo

```
cad = toLowerCase (“Cadena”)    # cad toma el valor de “cadena”
```

Consideraciones

- Si se envían 1 vector string, se debe validar que solo contenga 1 elemento, caso contrario reportar el error

4.10.8.4 ToUpperCase

Recibe como parámetro una cadena y devuelve una segunda cadena que contiene la palabra indicada en el parámetro con todas sus letras mayúsculas.

Sintaxis

```
toUpperCase (<Expresión>)
```

Ejemplo

```
cad = toUpperCase (“Cadena”)    # cad toma el valor de “CADENA”
```

Consideraciones

- Si se envían 1 vector string, se debe validar que solo contenga 1 elemento, caso contrario reportar el error.

4.10.8.5 Trunk

Recibe como parámetro un valor con punto flotante y devuelve un entero sin los valores decimales.

Sintaxis

```
trunk (<Expresión>)
```

Ejemplo

```
val = trunk (5.8)    #val toma el valor de 5
```

Consideraciones

- Si se envían 1 vector numeric, se debe validar que solo contenga 1 elemento, caso contrario reportar el error.

4.10.8.6 Round

Recibe como parámetro un valor con punto flotante (real) y devuelve un entero redondeando el valor decimal según las siguientes reglas:

- Si el decimal es **mayor o igual** que 0.5, se aproxima al número superior
- Si el decimal es **menor** que 0.5, se aproxima al número inferior

Sintaxis

```
round (<Expresión>)
```

Ejemplo

```
val = round(5.8)    #val tome el valor de 6
val = round(5.5)    #val tome el valor de 6
val = round(5.4)    #val tome el valor de 5
```

Consideraciones

- Si se envían 1 vector numeric, se debe validar que solo contenga 1 elemento, caso contrario reportar el error.

4.11 Mean, Median and Mode

La media, mediana y moda son funciones estadísticas básicas para el procesamiento de datos. Estas funciones se aplicarán como un conjunto de valores agrupados como un vector.

4.11.1 Sintaxis

La sintaxis para la creación de un diagrama de dispersión es la siguiente:

```
Vector = c(12,13,15,16,24,15,17,19,17,15)

Mean1 = mean(Vector)  # No se aplica trim, Mean1 toma el valor de 16.3
Mode1 = mean(Vector)  # No se aplica trim, Mode1 toma el valor de 15
Median1 = mean(Vector)        # No se aplica trim, Median1 toma el valor de 15.5

# Si colocamos el trim en 14, no se toman en cuenta los valores menores a 14
# Solo analizamos los siguientes valores 15,16,24,15,17,19,17,15
Mean2 = mean(Vector,14)       # Se aplica trim, Mean2 toma el valor de 17.25
Mode2 = mean(Vector,14)       # Se aplica trim, Mode2 toma el valor de 15
Median2 = mean(Vector,14)     # Se aplica trim, Median2 toma el valor de 16.5
```

Donde:

- **V:** Es el vector que posee un conjunto de datos.
- **Trim:** Parámetro opcional que indica a partir de que valor se deben tomar en cuenta para la medición, todos los valores menores a trim, se descartaran. Si trim no se indica se toman en cuenta todos los valores.

5 Estructuras de datos

Las estructuras de datos en el lenguaje arit son herramientas que nos permiten almacenar distintos valores, las estructuras básicas que incluye el lenguaje son las siguientes: vectores, listas, matrices y arreglos.

5.1 Vectores

Los vectores son la estructura más básica del lenguaje arit, los tipos de vectores que existen son con base a los tipos primitivos del lenguaje.

5.2 Creación de vectores

Al crear una variable con un valor primitivo este se convertirá en un vector del tipo correspondiente al valor, por lo tanto, para crear un vector de tamaño 1 utilizamos una declaración simple con un valor primitivo, pero si necesitamos un vector de múltiples valores vamos a utilizar la función C (concatenar).

Sintaxis para la creación

```
# Definición de un vector de tamaño 1
<Identificador> = <Expresión> [;]?
```

```
# Definición de un vector de múltiples valores
<Identificador> = c(<LISTA_EXPRESIONES>) [;]?
```

Ejemplo

```
# Creando un vector de tamaño 1
Vector1 = “hola mundo”; # Vector de tipo String

#  Creando un vector de tamaño 1 con la función C
Vector2 = c(“hola mundo”);

# Creando un vector de múltiples valores
Vector3 = c(“Casa”, “Gato”);

# Creando un vector a partir de otro vector
Vector4 = c(Vector1, Vector3); # Contenido del vector vector4 -> [“hola mundo”, “Casa”, “Gato”]

# Creado un vector a partir de otros vectores de tamaño 1
Perro = “Perro”
Gato = “Gato”
Pez = “Pescado”
Animales = c(Perro, Gato, Pez); # Contenido del vector Animales ->[“Perro”, “Gato”, “Pescado”]

# Creando un vector de tamaño 1
Vector1 = NULL # Vector de tipo String
```

Consideraciones

- El tipo del vector estará dado por el tipo de los valores que este contenga.
- Únicamente se pueden crear vectores con tipos primitivos o concatenando otros vectores.
- Si se intenta crear un vector con distintos tipos de dato se debe hacer el casteo correspondiente para que todos los elementos sean del mismo tipo, el casteo es el mismo que se explica en la función c.
- Si se crea un vector con un valor NULL, el tipo de este será string.

5.2.1 Acceso a vectores

Para el acceso a algún elemento del vector debemos utilizar su índice, este debe ser de tipo integer, los índices en arit inician desde 1, al contrario que la mayoría de los lenguajes de programación, esto se debe a la estrecha relación de arit con el análisis estadístico, donde es más común utilizar el 1 que el 0 para manejo de índices.

Sintaxis para acceso

```
# Acceso a un vector
<Identificador> ([ < Expresion > ] ) +
```

El valor de retorno del acceso a un vector, de igual manera será un vector, pero de tamaño 1 y con el tipo correspondiente a su valor.

Ejemplo

```
# Declarando vectores
Perro = “Perro”
Gato = “Gato”
Animales = c(Perro, Gato) # Contenido del vector animales -> [“Perro”, “Gato”]

# Acceso vector
Aux = Animales[1] # Recordar que el índice inicia en 1, contenido del vector Aux -> [“Perro”]

# Creando otro vector
Animales = c(Aux, Animales) # Contenido del vector animales -> [“Perro”, “Perro”, “Gato”]

# Creando otro vector
Animales = c(Animales[3], Animales) # Contenido del vector animales -> [“Gato”, “Perro”, “Perro”, “Gato”]
```

Consideraciones

- El índice de acceso debe ser un valor entre 1 a la cantidad de elementos del vector, si el índice estuviera fuera de rango se deberá indicar error.
- Si se accesa un vector que tiene un valor de null, el tipo de valor que va a devolver será de tipo string.

5.2.2 Modificación de vectores

En ocasiones es necesario modificar los valores existentes de los vectores o incluso agregar nuevos, por lo tanto, es necesario el acceso a los vectores, pero esta vez no para obtener un valor, sino para asignar otro.

Sintaxis para modificar

```
<Identificador>([ <Expresion> ])+ = Expresión [;]?
```

Ejemplo

```
# Creando un vector
Vector1 = "hola”; # Se crea un nuevo vector de tamaño 1

# Agregando un nuevo valor al vector
Vector1[4] = “45” # Contenido del vector Vector1 -> [“hola”, NULL, NULL, “45”]

# Reemplazando un valor existente
Vector1[3] = “cambio” # Contenido del vector Vector1 -> [“hola”, NULL, "cambio”, “45”]

# Agregando un nuevo valor sin índice
Vector1 = c(Vector1, “nuevo”) # Contenido del vector Vector1 -> [“hola”, NULL, "cambio”, “45”,
```

```
“nuevo”]

# Reemplazando un valor existente
Vector1[3] = c(4,“cambio”) # Error, porque estoy tratando de asignar más de un valor a una posición
# del vector

# Reemplazando un valor existente
Vector1[3] = c(“otro valor”) # Contenido del vector Vector1 -> [“hola”, NULL, "otro valor”, “45”,
“nuevo”]
```

Consideraciones

- Se puede agregar o modificar un nuevo elemento en cualquier índice, incluso si este índice es mayor al tamaño actual del vector.
- Si se agrega un nuevo valor en un índice mayor al tamaño del vector, las posiciones entre la última posición del vector y el nuevo índice deben tomar el valor por defecto del tipo de vector que se esté manejando.
- Si agregamos un nuevo valor al vector de un tipo distinto, se debe hacer el casteo correspondiente explicado en la función c, para mantener el mismo tipo de elementos.
- Si se intenta agregar o modificar un valor utilizando acceso por índice con una expresión función c, se debe verificar que la cantidad de parámetros sea 1, caso contrario reportar el error.

5.2.3 Operaciones entre vectores

Los vectores se pueden manipular mediante expresiones aritméticas, lógicas y de comparación. Al utilizar cualquier operación el resultado de esta es un nuevo vector con los resultados de las operaciones realizadas.

Ejemplos

```
# SUMA DE VECTORES
# Creando un nuevo vector
A = c(1, 2, 3, 4, 5)
B = c(5, -2, 1, 2, 8)


# Sumando un numero a un vector
A = A + (5*3-2)      # nuevo contenido del vector A -> [14, 15, 16, 17, 18]

# Suma entre vectores
A = A + A + A      # nuevo contenido del vector A -> [42, 45, 48, 51, 54]


# RESTA DE VECTORES
# Creando un vector
A = c(1, 2, 3, 4, 5)

# Restando un numero a un vector
A = A - 5      # nuevo contenido del vector A -> [-4, -3, -2, -1, 0]
```

```
# Resta entre vectores
A = A - A - A     # nuevo contenido del vector A -> [4, 3, 2, 1, 0]


# Operaciones de comparación entre vectores
# Creando un vector
A = c(1, 2, 3, 4, 5)

# Verificando si cada elemento del vector es menor a 3
C= A < 3      # nuevo contenido del vector C -> [true, true, false, false, false]

# Comparando los elementos de cada vector
C= A < B      # nuevo contenido del vector C -> [true, false, false, false, true]

# Operaciones lógicas entre vectores
# Creando un vector
D = c(true, false, false, false, false)

# Aplicando operación OR
B = A | true      # nuevo contenido del vector B -> [true, true, true, true, true]

# Comparando los elementos de cada vector
A = B & D      # nuevo contenido del vector A -> [true, false, false, false, false]
```

Consideraciones

- Las operaciones permitidas están sujetas a la tabla de operaciones, si se intentan realizar operaciones no listadas se deberá reportar el error.
- Las operaciones entre vectores únicamente son válidas si ambos vectores son del mismo tamaño, sino se debe reportar el error.

5.3 Listas

La lista en el lenguaje arit a diferencia de los vectores pueden almacenar diferentes tipos de datos.

5.3.1 Creación de listas

Para crear una nueva lista será necesario utilizar la función list o también se puede utilizar la función c para crear una nueva lista.

Sintaxis para la creación de una lista

```
# Definición de una lista
<Identificador> = list( <LISTA_EXPRESIONES> ) [;]?

# Definición de un vector de múltiples valores
<Identificador> = c(<LISTA_EXPRESIONES>) [;]?
```

Consideraciones

- Para crear una lista a partir de la función c, será necesario que una expresión de esta lista sea de tipo list, verificar el funcionamiento de la función c para más detalle.

Ejemplo

```
# Creando una lista con un valor primitivo
Lista1 = list("hola mundo");

# Creando otra lista
Lista2 = list("hola mundo", 43, TRUE, 32.3)

# Combinando listas, el tipo de la variable Lista3 es list
Lista3 = c(Lista1, Lista2) # Contenido de Lista3 -> ["hola mundo", "hola mundo", 43, TRUE, 32.3]

# Creando una lista con la función C, el tipo de la variable Lista4 es list
Lista4 = c(List(1,2,3), 4, "hola mundo") # Contenido de Lista4 -> [1,2,3,4, "hola mundo"]
```

5.3.2 Acceso a listas

El acceso a las listas es similar a los vectores, por medio del índice, pero cuenta con una importante diferencia que se explica a continuación.

Sintaxis para el acceso a listas

```
# Tipo de acceso 1 a lista
<Identificador> [ < EXPRESION > ] ([<EXPRESION>] | [[<EXPRESION>]])*
# Este tipo de acceso devuelve el valor en el índice especificado, pero adentro de una nueva lista.

# Tipo de acceso 2 a lista
<Identificador> [ [< EXPRESION >] ] ([<EXPRESION>] | [[<EXPRESION>]])*
# Este tipo de acceso devuelve el valor en el índice especificado.
```

Ejemplo

```
# Creando una lista
Lista1 = list(1,2,3,4)

# Acceso a la lista tipo 1
Lista2 = Lista1[2] # Nos devuelve una nueva lista, pero con un vector de tamaño 1 y un valor de 2
                   # Contenido de list lista2 = [2]
                   # Tipo de lista2 -> list

# Acceso a la lista tipo 2
Num = Lista1[[3]] # Nos devuelve un vector de tamaño 1 y con un valor de 3.
                  # Contenido del vector Num = [3]
                  # Tipo de Num = Vector de integer
```

```
# Creando otra lista
Lista3 = list(1,2, 3, 4, c(5,6))  # Contenido de Lista3 = [1,2,3,4, (5,6)]
                                  # Tipo de Lista3 -> list

# Acceso a la lista tipo 1

Lista4 = Lista3[5] # Nos devuelve una nueva lista, pero con un vector en su interior
                   # Contenido de list Lista4 = [(5,6)]
                   # Tipo de Lista4 -> list

# Acceso a la lista tipo 2
Vector1 = Lista3[[5]] # Nos devuelve un vector
                      # Contenido de Vector1 = [5,6]
                      # Tipo de Vector1 -> Vector de integer

# Imprimiendo un valor dentro de la lista
Print(Lista3[[5]][2]) # imprime 6

# Imprimiendo un valor dentro de la lista
Print(Lista3[5][[1]][1]) # imprime 5
```

Consideraciones

- El índice de acceso debe ser un valor entre 1 a la cantidad de elementos del vector, si el índice estuviera fuera de rango se deberá indicar error.

5.3.3 Modificación de listas

El funcionamiento de estas es similar a los vectores, pero la mayor diferencia se centra en los tipos de acceso ya sea con [] o [[]].

Sintaxis para modificar listas

```
<Identificador>([ <Expresion> ] | [[ <Expresion> ]])+ = Expresión [;]?
```

Ejemplo

```
# Utilizando el acceso de tipo 2 con [[x]]
 # Creando una lista
L1 = list("hola"); # Se crea una nueva lista de tamaño 1

# Agregando un nuevo valor a la lista
L1[[4]] = "45" # Contenido de la lista L1 -> ["hola", NULL, NULL, "45"]

# Reemplazando un valor existente
L1[[3]] = "cambio" # Contenido de la lista L1 -> ["hola", NULL, "cambio", "45"]

# Agregando un nuevo valor sin índice
```

```
L1 = c(L1, “nuevo”) # Contenido de la lista L1 -> [“hola”, NULL, "cambio”, “45”, “nuevo”]

# Utilizando el acceso de tipo 1 con [x] ----------------------------------------------------------------
# Creando una lista
L2 = list("hola”); # Se crea una nueva lista de tamaño 1

# Agregando un nuevo valor a la lista con el acceso de tipo 1
L2[4] = c(“45”, “adios”) # Error

# Agregando un nuevo valor a la lista con el acceso de tipo 1
L2[4] = list(“45”, “adiós”) # Error

# Agregando un nuevo valor a la lista con el acceso de tipo 1
L2[4] = list(c(“45”, ”adiós”)) # Contenido de la lista L2 -> [“hola”, NULL, NULL, (“45”, “adios”)]

# Agregando un nuevo valor a la lista con el acceso de tipo 1
L2[4] = ”45” # Contenido de la lista L2 -> [“hola”, NULL, NULL, “45”)]
```

Consideraciones

- Si se desea modificar o agregar un nuevo elemento utilizando la sintaxis tipo 1 ([]) se debe tomar en cuenta lo siguiente
    - La parte de la expresión puede ser un valor de cualquier tipo primitivo.
    - Si la parte de la expresión fuera una función c, la cantidad de parámetros de esta debe ser solamente 1, si fueran más de uno debe arrojar error.
    - Si la parte de la expresión fuera una función list, la cantidad de parámetros de esta debe ser solamente 1, si fueran más de uno debe arrojar error.
- Si se agrega un nuevo valor en un índice mayor al tamaño de la lista, las posiciones entre la última posición de la lista y el nuevo índice deben tomar el valor de NULL.
- Cuando se obtenga una posición de la lista cuyo valor sea NULL, el tipo de este a nivel de vector será string.

5.4 Matrices

Las matrices en arit nos permiten almacenar solamente datos de tipo primitivo, la diferencia principal entre el vector y la matriz es que esta última organiza sus elementos de forma bidimensional, la primera dimensión indica fila y la segunda indica columna.

5.4.1 Creación de matrices

Cuando creamos una nueva matriz debemos tomar en cuenta que el orden de inserción será por columnas.

Sintaxis de creación

```
# Definición de una matriz de x elementos
Matrix (data, nrow, ncol)
```

En donde:

- **Data:** Puede ser una función c con tipos primitivos o vectores, un vector existente, o un valor de tipo primitivo.
- **nrow:** Es una expresión que indica el número de filas de la matriz.
- **ncol:** Es una expresión que indica el número de columnas de la matriz.

Ejemplo

```
# Definición de una matriz de x elementos
A = matrix(c(1, 2, 3, 4, 5), 5, 4)
# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,]    1    1    1    1
# [2,]    2    2    2    2
# [3,]    3    3    3    3
# [4,]    4    4    4    4
# [5,]    5    5    5    5

# Definición de una matriz de x elementos
B = matrix(c(1, 2, 3, 4), 5, 4)
# Contenido de la matriz B
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    3    4
# [2,]    2    3    4    1
# [3,]    3    4    1    2
# [4,]    4    1    2    3
# [5,]    1    2    3    4

# Definiendo una matriz a partir de un vector
Vec1 = c(1,2,3,4)
A = matrix(vec1, 2, 1)
# Contenido de la matriz A
#      [,1]
# [1,]    1
# [2,]    2

#Definiendo una matriz con 1 elemento
A = matrix(5, 3, 5)
# Contenido de la matriz A
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    5    5    5    5    5
# [2,]    5    5    5    5    5
# [3,]    5    5    5    5    5
```

Consideraciones:

- Si la cantidad de elementos a insertar fuera mayor a la cantidad de posiciones de la matriz, se agregarán tantos elementos como sea posible hasta agotar posiciones de la matriz, sin importar

si faltaron elementos, por ejemplo, si queremos insertar 4 elementos, pero nuestra matriz solo posee capacidad para 2, solamente se agregan los primeros 2 elementos.

- Si la cantidad de elementos a insertar fuera menor a la cantidad de posiciones de la matriz, se van a repetir los elementos en el orden correspondiente hasta agotar las posiciones de la matriz, por ejemplo, si quisiéramos insertar 5 elementos y nuestra matriz tiene una capacidad de 15, se agregarán 3 veces los elementos para completar las posiciones de la matriz.
- La cantidad de elementos a insertar debe múltiplo o submúltiplo de la cantidad de posiciones de la matriz.
- Si se intenta crear una matriz con distintos tipos de dato se debe hacer el casteo correspondiente para que todos los elementos sean del mismo tipo, el casteo es el mismo que se explica en la función c.

5.4.2 Acceso a matrices

Para accesar a la matriz se utilizarán sus índices, para la matriz existirán 4 tipos de acceso:

- Acceso utilizando ambos índices, [x, y] - esto retorna una posición en específico y será un vector de 1 elemento.
- Acceso utilizando solo el índice de la fila, [x,] (notar la coma en la sintaxis) - esto retorna un vector con los elementos de la fila correspondiente.
- Acceso utilizando solo el índice de la columna, [, y] (notar la coma en la sintaxis) - esto retorna un vector con los elementos de la columna correspondiente.
- Acceso a un elemento especifico, [z] - esto retorna un elemento en específico y será un vector de 1 elemento.

Sintaxis de acceso

```
# Tipo 1
[<Expresión>, <Expresión>]

# Tipo 2
[<Expresión>, ]

# Tipo 3
[, <Expresión>]

# Tipo 4
[<Expresión>]
```

Ejemplo

```
# Creando una nueva matriz
A = matrix(c(1, “2”, 3, 4), 3, 4)
# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "2"
# [2,] "2"  "1"  "4"  "3"
# [3,] "3"  "2"  "1"  "4"
```

```
# Acceso tipo1
B = A[2,2]  # Contenido de B -> [“1”]

# Acceso tipo2, devuelve una fila
B = A[2, ]   # Contenido de B -> [“2”, “1”, “4”, “3”]

# Acceso tipo3, devuelve una columna
B = A[, 3]   # Contenido de B -> [“3”, “4”, “1”]

# Acceso tipo4
B = A[5]    # Contenido de B -> [“1”] ya que es acceso por columna
```

Consideraciones

- Los índices de acceso deben ser valores entre 1 a la cantidad de elementos de fila o columna correspondiente, si alguno de los índices estuviera fuera de rango deberá indicar error.
- Si fuera un acceso de tipo 4, se debe verificar que el índice sea un valor de 1 a la cantidad de posiciones de la matriz, si este estuviera fuera de rango deberá indicar error.

5.4.3 Modificación de matrices

La modificación de matrices al igual que el acceso tendrá los mismos tipos de acceso, pero esta vez servirá para modificar su valor.

- Modificación utilizando ambos índices, [x, y] - esto reemplaza una posición en específico con el valor deseado.
- Modificación utilizando solo el índice de la fila, [x,] - esto reemplaza los elementos de la fila con el o los valores deseados.
- Modificación utilizando solo el índice de la columna, [, y] - esto reemplaza los elementos de la columna con el o los valores deseados.
- Modificación utilizando solo un índice, [z] - esto reemplaza una posición en específico con el valor deseado.

Sintaxis para modificar

```
# Tipo 1
[<Expresión>, <Expresión>] = <Expresion>

# Tipo 2
[<Expresión>, ] = <Expresion>

# Tipo 3
[, <Expresión>] = <Expresion>

# Tipo 4
[<Expresión>] = <Expresion>
```

Ejemplo

```
# Creando una nueva matriz
A = matrix(c(1, “2”, 3, 4), 3, 4)
# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "2"
# [2,] "2"  "1"  "4"  "3"
# [3,] "3"  "2"  "1"  "4"

# Modificación tipo 1
A[3,3] = 8

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "2"
# [2,] "2"  "1"  "4"  "3"
# [3,] "3"  "2"  "8"  "4"

# Modificación tipo 2
A[2, ] = 9

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "2"
# [2,] "9"  "9"  "9"  "9"
# [3,] "3"  "2"  "1"  "4"

# Modificación tipo 2
A[2, ] = c(9)

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "2"
# [2,] "9"  "9"  "9"  "9"
# [3,] "3"  "2"  "1"  "4"


# Modificación tipo 3
A[ ,4] = 6

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "6"
# [2,] "2"  "1"  "4"  "6"
# [3,] "3"  "2"  "1"  "6"

# Modificación tipo 3
```

```
A[ ,4] = c(6,3,9,45) # Error

# Modificación tipo 3
A[ ,4] = c(6,7,8)

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "3"  "6"
# [2,] "2"  "1"  "4"  "7"
# [3,] "3"  "2"  "1"  "8"

# Modificación tipo 4
A[7] = 0

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] "1"  "4"  "0"  "2"
# [2,] "2"  "1"  "4"  "3"
# [3,] "3"  "2"  "1"  "4"
```

Consideraciones

- Los índices de acceso deben ser valores entre 1 a la cantidad de elementos de fila o columna correspondiente, si alguno de los índices estuviera fuera de rango deberá indicar error.
- Si fuera una modificación de tipo 4, se debe verificar que el índice sea un valor de 1 a la cantidad de posiciones de la matriz, si este estuviera fuera de rango deberá indicar error.
- Si se va a utilizar una función C como expresión para una modificación de tipo 4, esta debe tener solamente 1 elemento, caso contrario deberá reportar el error.
- Si se va a utilizar una función C como expresión para una modificación de tipo 2, esta puede tener solamente 1 elemento o exactamente la misma cantidad de elementos que de columnas de la matriz, caso contrario deberá reportar el error.
- Si se va a utilizar una función C como expresión para una modificación de tipo 3, esta puede tener solamente 1 elemento o exactamente la misma cantidad de elementos que de filas de la matriz, caso contrario deberá reportar el error.
- Si se va a utilizar la modificación tipo 2 o 3, con un solo valor, se debe reemplazar TODA la fila o columna con este único valor.
- Si al reemplazar un valor y este es de tipo diferente al resto de valores de la matriz, se debe hacer el casteo correspondiente explicado en la función c, para mantener el mismo tipo de elementos.

5.4.4 Operaciones entre matrices

Las matrices se pueden manipular mediante expresiones aritméticas (suma, resta, multiplicación, y división), lógicas y de comparación. Al utilizar cualquier operación el resultado de esta es una nueva matriz con los resultados de las operaciones realizadas.

Ejemplos

```
# Suma de matrices
# Creando una nueva matriz
A = matrix(c(1,2,3,4), 3, 4)

# Sumando matrices
A = A + A + A

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,]    3   12    9    6
# [2,]    6    3   12    9
# [3,]    9    6    3   12

# Restando una expresión a una matriz
A = matrix(c(1,2,3,4), 3, 4)
A = A + A – 3

# Contenido de la matriz A
#     [,1] [,2] [,3] [,4]
# [1,]   -1    5    3    1
# [2,]    1   -1    5    3
# [3,]    3    1   -1    5
# Operaciones de comparación entre matrices
A = matrix(c(1,2,3,4), 3, 4)
A = A + A < 3

# Contenido de la matriz A
#       [,1]  [,2]  [,3]  [,4]
# [1,] TRUE FALSE FALSE FALSE
# [2,] FALSE TRUE FALSE FALSE
# [3,] FALSE FALSE TRUE FALSE

# Operaciones lógicas entre matrices
A = A & A

# Contenido de la matriz A
#      [,1] [,2] [,3] [,4]
# [1,] TRUE FALSE FALSE FALSE
# [2,] FALSE TRUE FALSE FALSE
# [3,] FALSE FALSE TRUE FALSE
```

Consideraciones

- Las operaciones permitidas están sujetas a la tabla de operaciones, si se intentan realizar operaciones no listadas se deberá reportar el error.
- Las operaciones entre matrices únicamente son válidas si ambas matrices tienen las mismas dimensiones, sino se debe reportar el error.

- Si hay una operación entre una matriz y un primitivo numérico, este se debe tratar como un escalar.
- Las operaciones tienen que realizarse celda contra celda como se puede apreciar en los ejemplos.

5.5 Arreglos

Los arreglos en arit son estructuras que pueden almacenar múltiples dimensiones.

5.5.1 Creación de arreglos

La creación de arreglos consta de un conjunto de expresiones y las dimensiones que deseemos.

Sintaxis

```
Array (<Expresion> , <vector>)
```

Ejemplo

```
# Creando un arreglo de 3 dimensiones
result = array(4, c(3,3,2))
# Contenido del arreglo result
#, , 1
#
#    [,1] [,2] [,3]
#[1,]    4    4    4
#[2,]    4    4    4
#[3,]    4    4    4
#
#, , 2
#
#    [,1] [,2] [,3]
#[1,]    4    4    4
#[2,]    4    4    4
#[3,]    4    4    4


# Creando otro arreglo
result = array(c(5, list(7,8,9)), c(2,3,3))
# Contenido del arreglo result
#, , 1
#
#    [,1] [,2] [,3]
#[1,] 5    8    5
#[2,] 7    9    7
#
#, , 2
#
#    [,1] [,2] [,3]
```

```
#[1,] 8   5   8
#[2,] 9   7   9
#
#, , 3
#
#    [,1] [,2] [,3]
#[1,] 5   8   5
#[2,] 7   9   7

# Un arreglo con más dimensiones
result = array(c(5, list(7,8,9,10,11,12)), c(2,3,2,2))
# Contenido del arreglo result
#, , 1, 1
#
#    [,1] [,2] [,3]
#[1,] 5   8   10
#[2,] 7   9   11
#
#, , 2, 1
#
#    [,1] [,2] [,3]
#[1,] 12   7   9
#[2,] 5   8   10
#
#, , 1, 2
#
#    [,1] [,2] [,3]
#[1,] 11   5   8
#[2,] 12   7   9
#
#, , 2, 2
#
#    [,1] [,2] [,3]
#[1,] 10   12   7
#[2,] 11   5   8
```

Consideraciones

- Si se utiliza una función c como expresión, se deben realizar los casteos correspondientes explicados en la función c.
- El arreglo se llena por columnas.
- Los valores se irán a agregando hasta agotar las posiciones del arreglo, por lo tanto, si se agregaron todos los valores y aún restan posiciones en el arreglo se agregarán los valores de nuevo en las posiciones restantes, y si se agotaron las posiciones del arreglo y aún restan valores ya no se seguirán agregando elementos.

- Los elementos del arreglo deben de ser del mismo tipo, por ejemplo, solo de tipo primitivo, o de tipo list, etc.

5.5.2 Acceso a arreglos

El acceso en los arreglos está sujeto a los índices y los valores que retornan los accesos van a depender del tipo de dato del arreglo.

Sintaxis

```
Identificador ( [<Expresion>] ) +
```

Ejemplo

```
# Declarando un arreglo
result = array(c(5, list(7,8,9,10,11,12)), c(2,3,2))

# Accediendo a un arreglo
Variable1 = result[2][3][1] # Tipo de la variable Variable1 = List
                            # Contenido de la variable Variable1 -> list(11)
```

Consideraciones

- Cuando se realice un acceso a un arreglo se debe validar que se esté accediendo a todas sus dimensiones, caso contrario reportar el error.
- Todos los índices de acceso deben estar dentro del rango definido inicialmente caso contrario reportar el error.
- El arreglo puede retornar distintos tipos de datos, ya sean vectores, list, etc...

5.5.3 Modificación de arreglos

Para modificar un arreglo se utilizará la misma sintaxis de acceso que se mencionó anteriormente.

Sintaxis

```
Identificador ([<Expresion>)+ = <Expresion> [;]?
```

Ejemplo

```
# Declarando un arreglo
result = array(c(5, list(7,8,9,10,11,12)), c(2,3,2)) # Arreglo de tipo de list

# Modificando un valor del arreglo
result[1,1,1] = list(9)
```

Consideraciones

- Para modificar un arreglo se debe verificar que los índices de acceso estén dentro del rango establecido, caso contrario reportar el error.
- Cuando se realice un acceso a un arreglo se debe validar que se esté accediendo a todas sus dimensiones, caso contrario reportar el error.

6 Graficas en lenguaje Arit

Arit posee una amplia variedad de funciones que nos permiten realizar una gran cantidad de tablas y graficas. Estas son:

6.1 Graficas de Pie

Un gráfico circular es una representación de valores como cortes de un círculo con diferentes colores. Las secciones se etiquetan y los números correspondientes a cada sección también se representan en el gráfico.

6.1.1 Sintaxis

La sintaxis para la creación de una gráfica de Pie es la siguiente:

```
pie(x, labels, main);
```

Donde:

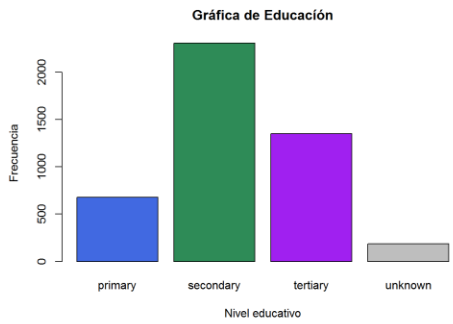
- X: Es un vector que contiene los valores numéricos del grafico
- Labels: Un vector que contiene las descripciones de cada una de las secciones
- Main: Indica el título del grafico

Consideraciones

- La cantidad de valores en el vector “labels” debe ser el mismo que el vector “X”, de lo contrario deberá reportarse un error y agregarle la etiqueta “Desconocido” + n a las secciones faltantes.

6.2 Graficas de Barras

Un gráfico de barras representa datos en barras rectangulares con una longitud de la barra proporcional al valor de la variable.



6.2.1 Sintaxis

La sintaxis para la creación de una gráfica de barras es la siguiente:

```
barplot( H, xlab, ylab, main, names.arg)
```

Donde:

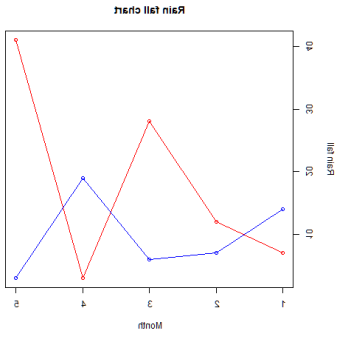
- H: Es el vector o matriz que contiene los valores numéricos para la gráfica.
- Xlab: Es la etiqueta para el eje X.
- Ylab: Es la etiqueta para el eje Y.
- Main: Es el título de la gráfica.
- Names.arg: Es un vector que contiene los nombres para cada una de las barras de la gráfica.

Consideraciones

- La cantidad de valores en el vector “Names.arg” debe ser el mismo que el vector “X”, de lo contrario deberá reportarse un error y agregarle la etiqueta “Desconocido”+ n a las barras faltantes.

6.3 Gráficos de línea

Un gráfico de líneas es un gráfico que conecta una serie de puntos dibujando segmentos de línea entre ellos. Estos puntos están ordenados en uno de sus valores de coordenadas (generalmente la coordenada x). Los gráficos de líneas generalmente se usan para identificar las tendencias en los datos.



6.3.1 Sintaxis

La sintaxis para la creación de una gráfica de Pie es la siguiente:

```
plot( v, type, xlab, ylab, main)
```

Donde:

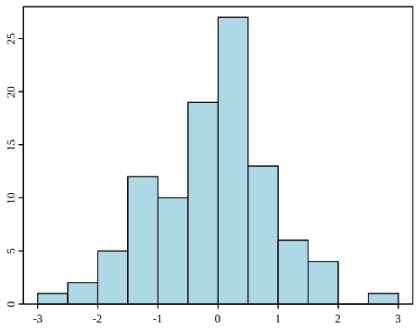
- V: Es el vector o matriz que contiene los valores numéricos para la gráfica.
- Type: Este argumento puede tomar distintos valores.
  - “P”: Grafica únicamente los puntos.
  - “l”: Grafica únicamente las líneas
  - “O”: Grafica ambos, las líneas y los puntos.
- Xlab: Es la etiqueta para el eje X.
- Ylab: Es la etiqueta para el eje Y.
- Main: Es el título de la gráfica.

Consideraciones

- Si el valor de “Type” no corresponde a ninguno de los parámetros indicados, deberá reportarse el error y tomar como defecto el valor “O”.

6.4 Histogramas

Un histograma representa las frecuencias de los valores de una variable agrupada en rangos. El histograma es similar al grafico de barras, pero la diferencia es que agrupa los valores en rangos continuos. Cada barra en el histograma representa la altura del número de valores presentes en ese rango.



6.4.1 Sintaxis

La sintaxis para la creación de un histograma es la siguiente:

```
hist(v, main, xlab, xlim, ylim)
```

Donde:

- V: Es el vector o matriz que contiene los valores numéricos para la gráfica.
- Xlab: Es la etiqueta para el eje X.
- Main: Es el título de la gráfica.

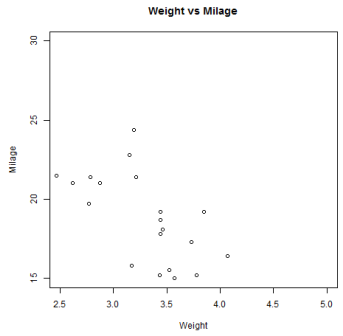
- Xlim: Recibe un vector de dos elementos numéricos que especifica el mínimo y el máximo permitido en el eje X.
- Ylim: Recibe un vector de dos elementos numéricos que especifica el mínimo y el máximo permitido en el eje Y.

Consideraciones

- Si el valor mínimo es mayor que el valor máximo deberá reportarse un error y tomar el valor mínimo como infinito negativo.
- Si el valor máximo es menor que el valor mínimo deberá reportarse un error y tomar el valor máximo como infinito positivo.
- Si alguno de los valores del vector V no cumplen con los criterios de valor mínimo o máximo para la gráfica, el punto se deberá descartar y no aparecerá en la gráfica, además de reportarse como una advertencia.

6.5 Diagrama de dispersión

Los diagramas de dispersión muestran muchos puntos trazados en el plano cartesiano. Cada punto representa los valores de dos variables.



6.5.1 Sintaxis

La sintaxis para la creación de un diagrama de dispersión es la siguiente:

```
plot(mat, y, main, xlab, ylab, xlim, ylim, byrow)
```

Donde:

- **MAT:** Es la matriz de dos dimensiones que contiene el conjunto de valores x,y.
- **Xlab:** Es la etiqueta para el eje X.
- **Ylab:** Es la etiqueta para el eje Y.
- **Main:** Es el título de la gráfica.
- **Xlim:** Recibe un vector de dos elementos numéricos que especifica el mínimo y el máximo permitido en el eje X.
- **Ylim:** Recibe un vector de dos elementos numéricos que especifica el mínimo y el máximo permitido en el eje Y.
- **Byrow:** Es una expresión booleana que indica el orden de la matriz, si fuera verdadero, el eje X corresponde al primer valor de la matriz, caso contrario corresponde al eje Y

Consideraciones

- Si el valor mínimo es mayor que el valor máximo deberá reportarse un error y tomar el valor mínimo como infinito negativo.
- Si el valor máximo es menor que el valor mínimo deberá reportarse un error y tomar el valor máximo como infinito positivo.
- Si alguno de los valores del vector V no cumplen con los criterios de valor mínimo o máximo para la gráfica, el punto se deberá descartar y no aparecerá en la gráfica, además de reportarse como una advertencia.
- La cantidad de valores en el vector "Y" debe ser el mismo que el vector "X", de lo contrario deberá reportarse un error y descartar los puntos que no coincidan.

7 Reportes Generales

Como se indicaba en la descripción del lenguaje, Arit software genera una serie de reportes sobre el proceso de análisis de los archivos de entrada. Estos son:

7.1 Reporte de errores

El intérprete deberá ser capaz de detectar todos los errores que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores en el que, como mínimo, debe mostrarse el tipo de error, su ubicación y una breve descripción de por qué se produjo.

7.1.1 Tipos de errores

Los tipos de errores se deberán de manejar son los siguientes:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos

7.1.2 Contenido de tabla de errores

La tabla de errores debe contener la siguiente información:

- Línea: Número de línea donde se encuentra el error.
- Columna: Número de columna donde se encuentra el error.
- Tipo de error: Identifica el tipo de error encontrado. Este puede ser léxico, sintáctico o semántico.
- Descripción del error: Dar una explicación concisa de por qué se generó el error.

Ejemplo

Tipo	Descripción	Fila	Columna
Léxico	El carácter "α" no pertenece al alfabeto del lenguaje	**33**	**21**
Sintáctico	Se esperaba un ';'	**264**	**15**
Semántico	No existe la variable x	**325**	**19**

7.1.3 Método de recuperación

Se definirán una los métodos por defecto que el estudiante deberá implementar para la recuperación de errores, estos son:

7.1.3.1 Léxicos y sintácticos:

Al descubrir un error, el analizador desecha símbolos de entrada, de uno en uno, hasta que encuentra uno perteneciente a un conjunto designado de componentes léxicos de sincronización. Estos componentes léxicos de sincronización son generalmente delimitadores, como el punto y coma, cuyo papel en el programa fuente está claro. Este método se denomina "Recuperación en modo pánico". Por ejemplo:

```
Variable + 10 = 20;          #Se encuentra un error sintáctico, se descartar símbolos hasta el ";"
Print("Hola mundo");        # Esta sentencia si se ejecuta.
```

7.1.3.2 Semánticos, a nivel de instrucción.

Si se detecta cualquier tipo de error semántico el estudiante deberá descartar la instrucción completa, por ejemplo:

```
a = 10/0  #Error semántico, la división por 0 no esta definida, se descarta la asignación completa
```

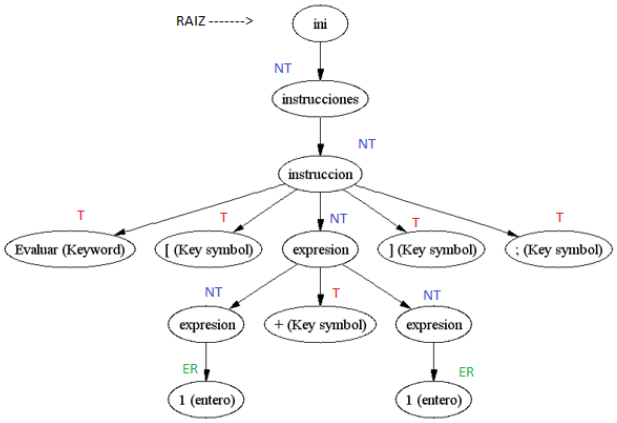
7.2 Reporte de tabla de símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución del archivo. Se deberán de mostrar todas las variables, funciones y procedimientos que fueron declarados, así como su tipo y toda la información que el estudiante considere necesaria.

IDENTIFICADOR	TIPO	DIMENSIÓN	DECLARADA EN	REFERENCIAS
anho	int	0	5	11,23,25
b	real	0	5	10,11,13,23
companhia	int	1	2	9,14,25
format	char	2	4	36,37,38
m	proc	0	6	17,21

7.3 Reporte de AST

Este reporte mostrara el árbol de análisis sintáctico que se produjo al analizar el archivo de entrada. Este debe de representarse como un grafo, se recomienda se utilizar Graphviz. El Estudiante deberá mostrar los nodos que considere necesarios y se realizarán preguntas al momento de la calificación para que explique su funcionamiento.



8 Gramáticas

Se deberá de desarrollar un manual explicando las gramáticas utilizadas, así como las acciones semánticas que permiten la creación del árbol de análisis sintáctico. Este manual debe contener todas las especificaciones de cada uno los lenguajes tales como:

- Expresiones Regulares.
- Precedencia utilizada.
- Cantidad de símbolos terminales.
- Enumeración de los símbolos terminales.
- Cantidad de símbolos no terminales.
- Explicación de cada uno de los símbolos no terminales (cuál fue su uso dentro de la gramática)
- Gramática funcional describiendo cada una de las acciones. (Definición dirigida por la sintaxis o esquema de traducción)

9 Apéndice A: Precedencia y asociatividad de operadores

La precedencia de los operadores indica el orden en que se realizaran las distintas operaciones del lenguaje. Cuando dos operadores tengan la misma precedencia, se utilizará la asociatividad para decidir qué operación realizar primero.

A continuación, se presenta la precedencia de operadores lógicos, aritméticos y de comparación.

NIVEL	OPERADOR	DESCRIPCION	asociatividad
10	[]	Acceso a elemento de arreglo	Izquierda
9	- !	menos unario not	Derecha
8	* / %	multiplicativas	Izquierda
7	+ - +	aditivas concatenación de cadenas	Izquierda

6	< <= > >=	relacionales	no asociativo
5	= !>	Igualdad Diferencia	izquierda
4	&	And	izquierda
3	|	Or	Izquierda
2	? :	ternario	Derecha
1	=	asignación	derecha

10 Entregables y Restricciones

10.1 Entregables

Deberán entregarse todos los archivos necesarios para la ejecución de la aplicación, así como el código fuente, la gramática y la documentación. La calificación del proyecto se realizará con los archivos entregados en la fecha establecida. El usuario es completa y únicamente responsable de verificar el contenido de los entregables. La calificación se realizará sobre archivos ejecutables. Se proporcionarán archivos de entrada al momento de calificación.

- Código Fuente
- Aplicación funcional
- Gramáticas
- Manual técnico
- Manual de usuario

10.2 Restricciones

- El proyecto deberá realizarse como una aplicación de escritorio utilizando Java
- Se debe realizar 2 veces la gramática para el mismo lenguaje, pero utilizando diferentes analizadores, además de agregar las acciones correspondientes a cada gramática para el correcto funcionamiento de estas.
- Se debe realizar una gramática ascendente utilizando flex y cup.
- Se debe realizar una gramática descendente utilizando JavaCC.
- Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ciencias y Sistemas los involucrados.

10.3 Consideraciones

- Durante la calificación se realizarán preguntas sobre el código para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas se reportará la copia.

10.4 Entrega del proyecto

1. La entrega será virtual y se habilitaran dos plataformas, Canvas, que es la habitual del curso y un entregable por medio de DropBox para que el estudiante tenga dos formas de entrega.

2. El periodo de entrega será de 12:00 PM a 20:00 PM para en caso de que exista alguna complicación el estudiante tenga tiempo suficiente de contactar a los auxiliares y solucionar el problema. NO SE RECIBIRÁN PROYECTOS DESPUÉS DE LA FECHA NI HORA DE LA ENTREGA ESTIPULADA.
3. La entrega de cada uno de los proyectos es individual.
4. La entrega del proyecto será mediante un archivo comprimido de extensión zip, el formato [OLC2]P1_<carnet>.zip.
5. Entrega del proyecto:

Domingo 29 de marzo a las 20:00 horas