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

J#

Contenido

1 Competencias

1.1 Competencia general

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

1.2 Competencias específicas

- Que el estudiante utilice un generador de analizadores léxicos y sintácticos para construir un compilador.
- Que el estudiante implemente la traducción orientada por la sintaxis utilizando reglas semánticas haciendo uso de atributos sintetizados.
- Que el estudiante genere una traducción de código de alto nivel a código en 3 direcciones.

2 Descripción

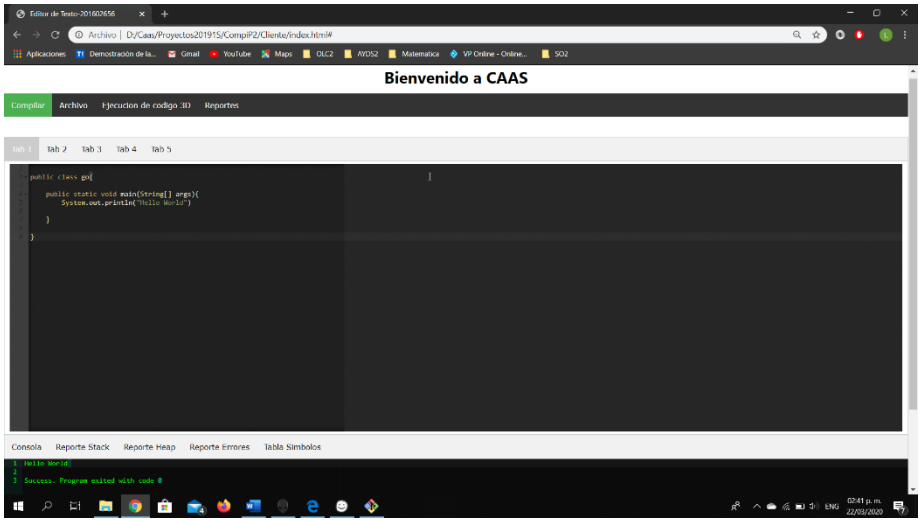
J# es un lenguaje de programación basado en c# y javascript, su mayor característica es el control de ámbitos, lo que nos permite llevar un mejor manejo del consumo de memoria. Este lenguaje está formado por un conjunto de herramientas muy flexibles que pueden ampliarse fácilmente mediante paquetes, 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 J#. Este IDE a su vez será el encargado de analizar el lenguaje J#. La aplicación contará con los siguientes componentes:

2.1.1 Compi Studio

Compi Studio es un entorno de desarrollo que provee las herramientas para la escritura de programas en lenguaje J#. 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 más adelante.



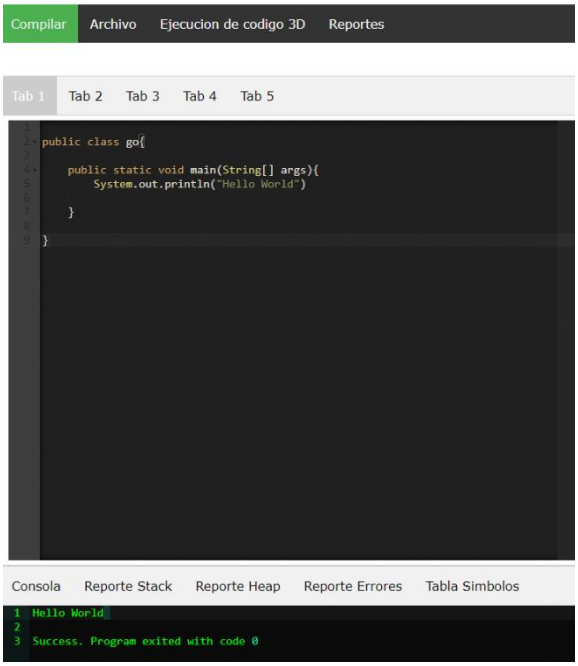
2.1.1.1 Características básicas

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 compilar, ejecutar y optimizar archivo
- Reporte de errores
- Reporte de tabla de símbolos
- Reporte de AST
- Reporte de optimización
- Reporte de Heap
- Reporte de Stack

2.1.1.2 Consola

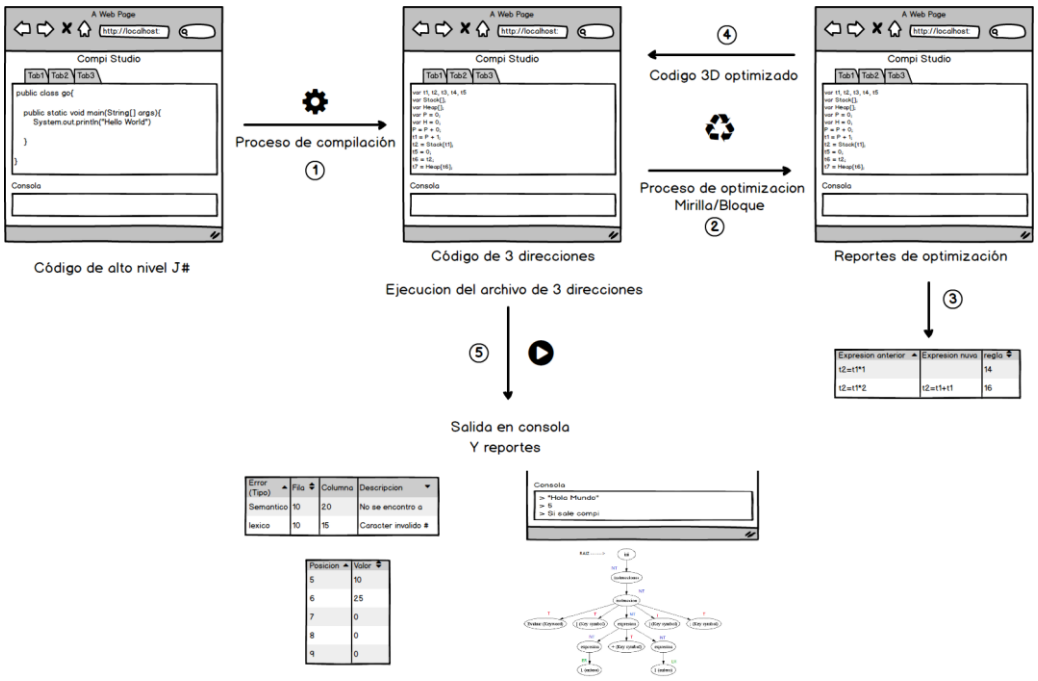
La consola es un área especial dentro del IDE que permite visualizar las notificaciones, errores, y advertencias que se produjeron durante el proceso de análisis de un archivo de entrada.



2.2 Flujo del proyecto

El flujo de la aplicación estará constituido por dos subprocesos, el primero será el proceso de compilación del código de alto nivel J#, y el segundo será la ejecución del código de tres direcciones generado a partir del proceso de compilación.

A partir de todo este proceso de compilación y ejecución se podrá generar diversos reportes en formato HTML, así como impresiones en consola para validar la correcta implementación.



2.2.1 Flujo específico de la aplicación

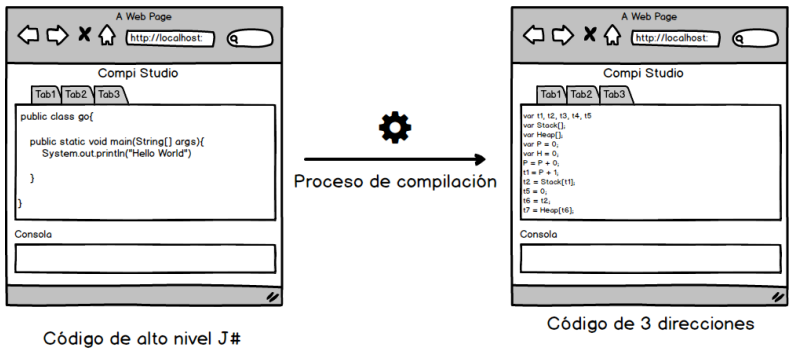
El flujo principal de la aplicación comienza con un archivo en lenguaje J# que se compila desde el IDE. El proceso de compilación retornará un archivo con formato de código de 3 direcciones que deberá ser ejecutado desde un segundo interprete realizado **por los auxiliares**. Un flujo alterno del programa consiste en optimizar la salida en C3D previo a ser ejecutada, ya sea optimizándolo con el método de mirilla o por bloque.

2.2.1.1 Proceso de compilación

Este proceso comenzará con un archivo del lenguaje de alto nivel J#. El proceso es el siguiente:

1. El programador crea su programa en lenguaje J# a través del entorno de desarrollo.
2. El programador solicita la compilación de su programa fuente.
3. La aplicación analiza y compila el archivo de entrada.
4. Como resultado de este proceso se retorna un archivo en formato de C3D. Este archivo puede mostrarse como una nueva pestaña del IDE o como el estudiante considere conveniente.

Se explica el flujo anterior con la siguiente imagen:

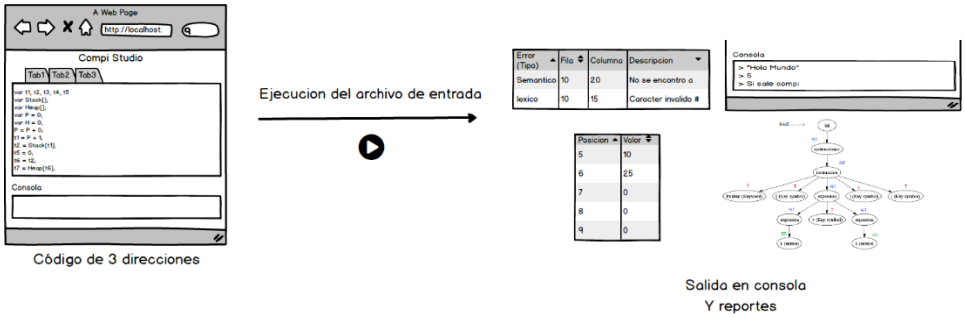


2.2.1.2 Proceso de ejecución

Este proceso comenzará con un archivo en formato de código intermedio. Este archivo se ejecutará y el proceso concluirá cuando se muestren las impresiones en consola.

1. El programador solicita la ejecución del archivo de código intermedio de 3 direcciones
2. La aplicación ejecuta el archivo de entrada.
3. Como resultado de este proceso se mostrarán en consola todas las salidas que el programa contenga, así como los reportes que sean pertinentes.

Se explica el flujo anterior con la siguiente imagen:

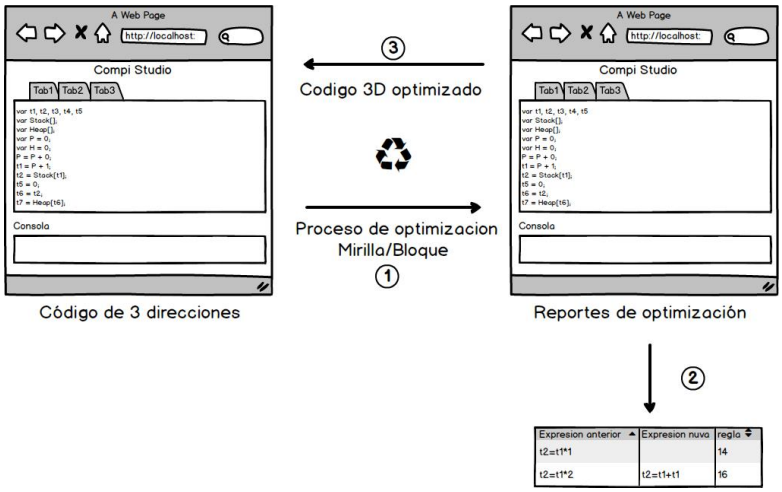


2.2.1.3 Proceso de optimización (Flujo alterno)

Este proceso comenzará con un archivo en formato de código intermedio. Este archivo se ejecutará y como resultado de este proceso se generará un nuevo archivo con formato C3D optimizado, ya sea utilizando optimización por bloque o por mirilla. Este proceso puede repetirse muchas veces.

1. El programador solicita la ejecución del archivo de código intermedio de 3 direcciones
2. La aplicación analiza el archivo de entrada y genera un código intermedio optimizado.
3. Como resultado de este proceso se mostrará un reporte de optimización y se generará un nuevo archivo de 3CD optimizado.

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 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, estructura o arreglo
- Una llamada a una función
- Una condición

Consideraciones

- Si una expresión retorna otra expresión, es válido seguir accediendo sobre ella cuantas veces sea posible.

2.3.3 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 J#

Es un lenguaje de programación basado en c# y Javascript, su mayor característica es el control de ámbitos, lo que nos permite llevar un mejor manejo del consumo de memoria. Una de sus principales ventajas es que posee las mismas funcionalidades que proveería el lenguaje Javascript con la peculiaridad de ser un lenguaje tipado, además una característica sintactica del lenguaje, es que el punto y coma es opcional.

3.1 Identificadores

Un identificador será utilizado para dar un nombre a variables y métodos. Un identificador de J# está compuesto básicamente por una combinación de letras, dígitos o guion bajo. Un identificador válido inicia con una letra o un guion bajo.

```
[a-zA-Z_][_a-zA-Z0-9ñÑ]+
```

```
// Identificador válido
Var_1
Hola

// Identificador inválido
09_abc
```

3.2 Case insensitive

El lenguaje J# 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 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

3.5 Valor nulo

En el lenguaje **J#** se utiliza la palabra reservada **null** para hacer referencia a la nada, esto indicará la ausencia de valor. Las cadenas, estructuras y arreglos tendrán este valor por defecto.

3.6 Palabras reservadas

A continuación, se muestran las palabras reservadas que posee J#

null	import	true	switch	continue	private	define	try
integer	var	false	case	return	void	as	catch
double	const	if	default	print	for	strc	throw
char	global	else	break	public	while	do	

3.7 Tipos de dato

Como se mencionó anteriormente, J# es un lenguaje de programación de tipado estático, lo que significa que las variables no pueden cambiar su tipo durante la ejecución.

3.7.1 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)	**[-2147483648, 2147483647]**	**0**
Double	**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.**	1 byte (8 bits)	**true, false**	**false**
Char	**Acepta un único carácter encerrado en comillas simples ‘a’**	1 byte (8 bits)	[**0, 255**]	**‘\0’**

Consideraciones

- Cualquier otro tipo de dato que no sea primitivo tomara el valor por defecto de null de ser necesario.

3.7.2 Tipos de referencia

Los tipos por referencia surgen de la creación de los siguientes elementos. Su valor está determinado por la referencia al Heap sobre alguno de los siguientes.

- Arreglos
- Estructuras
- String

3.7.2.1 Arreglos

Un arreglo es un conjunto de datos o una estructura de datos homogéneos que se encuentran almacenados en forma consecutiva. En el lenguaje J# únicamente están permitidos los arreglos simples, es decir, de una dimensión.

3.7.2.2 Estructuras

En el lenguaje J#, una estructura es un conjunto de datos almacenados en el heap de forma contigua. Es muy similar a un arreglo con la diferencia que su tamaño no puede cambiar y sus elementos pueden ser de diferente tipo.

3.7.2.3 String (cadenas)

Un String es una secuencia de caracteres ASCII delimitada por un elemento que indica su finalización, los String tienen un valor constante.

4 Estructura general del lenguaje J#

Un archivo escrito en lenguaje J# se compone de imports, definición de variables globales y funciones.

- Los archivos tendrán la extensión "*.j"
- El nombre del archivo podrá estar compuesto por letras, números, guiones (-) y puntos.
- La definición de la sentencia import, definiciones globales y funciones pueden estar en cualquier orden en el archivo, por lo tanto, el estudiante debe hacer las pasadas que considere necesarias para obtener toda la información que necesite utilizar para hacer la compilación y posterior traducción a C3D.

Ejemplo de nombres de archivo

```
Principal.j
funciones-recursivas.v2.j
```

4.1 Import

En el archivo se puede, opcionalmente, **importar funciones** desde otro archivo, esto se hace a través de la sentencia **import.** Y se importa únicamente funciones públicas.

Sintaxis

```
<import> ::=    'import' <nombre-archivo> [, <nombre-archivo>]*
```

Consideraciones

- Si la sentencia `import` existe solo puede venir una vez, y se importan los archivos que se desean separados por coma
- Solo existe un nivel de importación, es decir si un archivo “A.j” hace uso de un archivo “B.j”; y “B.j” a su vez hace uno de un archivo “C.j”, al ejecutar “A”, éste no tendrá acceso a las funciones del archivo “C.j”
- La sentencia `import` no realiza importaciones de variables ni constantes globales
- Se asume que los archivos están alojados en un mismo directorio.
- Lo archivos de entrada se almacenarán en una carpeta dentro del proyecto del estudiante, al utilizar la opción abrir del ide el archivo se debe almacenar en esta carpeta.

4.2 Definiciones globales

El archivo puede contener variables y constantes globales. Estás están definidas en la sección 5.3.

4.3 Definiciones de funciones

El archivo puede contener definición de funciones tanto públicas como privadas, ver sintaxis en sección 5.11

Consideraciones:

- Si se realiza una importación de una función que existe en el archivo actual, se le da precedencia a la función del archivo actual.
- Las funciones pueden ser llamadas desde cualquier parte del archivo, incluso, antes de su definición.

4.3.1 Función principal

Esta función será el inicio de la ejecución del programa. El nombre de la función será “principal”, es de tipo void, tiene visibilidad “private” y no recibe ningún argumento. La palabra “principal” no es reservada.

Ejemplo:

```
// Inicia archivo funciones.j

const PI := 3.1416

// Esta función es pública, lo que significa que puede ser importada
public Double areaCirculo(Double radio) {
  log(“calculando área de un círculo de rario ” + radio);
  return PI * radio ^ 2;
}

// está función es privada, lo que significa que NO puede ser importada
private void log(String msg) {
  print(msg);
}

// Fin archivo funciones.
```

```
// Inicia archivo ejemplo.j
import funciones.j  // importamos las funciones públicas del archivo funciones.j

const radio := 3.12;

private void principal() {
  Double area  = areaCirculo(radio);  // hago uso de la función importada
  log(“área = ” + area);   // ERROR la función log no está definida en el archivo ejemplo.j y no puede ser importada por tener visibilidad privada
}

// fin archivo ejemplo.j
```

5 Sintaxis del lenguaje J#

A continuación, se define la sintaxis para las sentencias del lenguaje J#. La **gramática** que se presenta en **el enunciado** es una **sugerencia y no significa que sea toda la solución para la sintaxis propuesta.** El estudiante puede y debe realizar los cambios a la sintaxis que considere necesario siempre y cuando la gramática siga reconociendo el mismo lenguaje.

5.1 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
}
```

5.2 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
```

5.3 Declaración de Variables y Constantes

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, un valor y un tipo. Los nombres de las variables no pueden coincidir con una palabra reservada.

Por otro lado, las constantes una vez definida su valor no puede cambiar durante la ejecución del programa.

Para poder utilizar una variable o constante se tiene que definir previamente. La declaración nos permite crear una variable y asignarle un valor.

Sintaxis

```
<variable_declaration> ::=   <type> <id_list> = <Expresion>;        // Declaración tipo 1
                             | var <identificador> := <Expresion> ;    // Declaración tipo 2
                             | const <identificador> := <Expresion> ;    // Declaración tipo 3
                             | global <identificador> := <Expresion> ;    // Declaración tipo 4
                             | <type> <id_list> ;                        // Declaración tipo 5

<id_list> ::=  <id_list>  ',' id
               | id

<type> ::= <tipo> <LISTA_COR>
           |<tipo>

<tipo> ::= 'integer'
          | 'double'
          | 'char'
          |'boolean'
          |id
```

Consideraciones

- En la declaración de tipo 1, todas las variables declaradas tomaran el mismo tipo y valor de la expresión. Al no hacer uso de las palabras reservadas 'var' o 'const' se asume que es 'var'. **Se utiliza el símbolo '='**
- En la declaración tipo 2, únicamente se podrá declarar una variable y esta inferirá el tipo según la expresión. La expresión no puede ser nula. **Se utilizará el símbolo ':='**
- En la declaración de tipo 3, se declara una constante, el tipo se infiere a partir de la expresión que no puede ser nula. **Se utilizará el símbolo ':='**
- En la declaración de tipo 4, se declara una variable global. No importa el entorno en el que este se declare, deberá de almacenarse en el entorno global y puede ser utilizada en cualquier entorno local. Se utilizará el símbolo **':='**
- En la declaración tipo 5, similar a tipo 1. Las variables toman el valor inicial por defecto del tipo asociado.

Ejemplo

```
# Declarando variables
Integer Var1 = 5;  // el tipo de la variable es integer

Global Var2 := 10;  // el tipo de la variable es integer

 String String = "hola";                    // Notese que String no es una palabra reservada

char a = 'a' ;

boolean var3,var4,var5,var6 = true;       // Todas las variables son de tipo booleano y toman el valor
```

```
true

global var_boolean := 2<3              // var_boolean tiene el valor de true

num := 10.5 ;                          // num es de tipo double y valor 40.5

Integer base, altura;                  // se definen las variables base y altura
```

5.4 Asignación de variables

Una variable puede cambiar su valor durante el tiempo de ejecución siempre y cuando el valor que se desea asignar sea del mismo tipo que el de la variable. Una variable no puede cambiar su tipo.

Para realizar una asignación se utilizará el operador '=' y es permitido asignar valores dentro de estructuras en cualquier nivel de anidación, dentro de arreglos, variables y cualquier otra expresión. No es válido asignar un valor a una constante después de que fue declarada.

A continuación, se presentan algunos ejemplos:

Sintaxis

```
Variable = 10;
Variable = Variable++;
Variable = Variable--;
Id.elemento[0] = "Hola";
If.funcion().funcion[0].elemento = 10;
Id.valor1.valor2.valor3 = 50.5;
Variable := {1,2,3};
Variable = {1,2,3,4,5,6,7,8,9};
```

Consideraciones

- La asignación puede contener los operadores de aumento y decremento.

5.5 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 (*), división (/), potencia (^^) y el módulo (%).

5.5.1 Suma

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

Operandos			
Operador	**Operador**	**Tipo Resultante**	**Ejemplos**
Integer	*Integer*	*Integer*	2+2=3

Integer	*Double*	*Double*	2+5.5=7.5 5.3+1=6.3
Integer	*char*	*integer*	3+'a'=100 'b'+1=99
Integer	*String*	*String*	1+"hola"="1hola" "hola+100="hola100"
Double	*String*	*String*	10.5+"Hola"="10.5Hola"
Double	*char*	*Double*	1.5+'a'=98.5
Double	*Double*	*Double*	1.5+1.5=3.0
char	*char*	*String*	'a'+'a'="aa";
char	*String*	*String*	'a'+"hola"="ahola"
String	*String*	*String*	"hola"+"hola"="holahola"
String	*boolean*	*String*	"hola"+true="holatrue"

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.
- Para los casos donde los char se tomen como un número, deberá convertirse a su equivalente en ASCII

5.5.2 Resta

La operación resta se produce mediante la sustracción de los siguientes operandos

Operandos			
Operador	**Operador**	**Tipo Resultante**	**Ejemplos**
Integer	*Integer*	*Integer*	2-2=0
Integer	*Double*	*Double*	2-5.5=-3.5 5.3-1=4.3
Integer	*char*	*integer*	3-'a'=-93 'b'-1=96
Double	*char*	*Double*	1.5+'a'=98.5
Double	*Double*	*Double*	1.5-2.5=-1.0
char	*char*	*Integer*	'a'-'b'=-1;

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.
- Para los casos donde los char se tomen como un número, deberá convertirse a su equivalente en ASCII

5.5.3 Multiplicación

La operación multiplicación se produce mediante el producto de los siguientes operandos

Operandos			
Operador	**Operador**	**Tipo Resultante**	**Ejemplos**

Integer	*Integer*	*Integer*	2*2=4
Integer	*Double*	*Double*	2*5.5=11 5.3*1=5.3
Integer	*char*	*integer*	3*'a'=291 'b'*1=98
Double	*char*	*Double*	1.5*'a'=144.5
Double	*Double*	*Double*	1.5-2.5=3.75
char	*char*	*Integer*	'a'*'b'=9506;

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.
- Para los casos donde los char se tomen como un número, deberá convertirse a su equivalente en ASCII

5.5.4 División

La operación division se produce mediante la división de los siguientes operandos

Operandos			
Operador	**Operador**	**Tipo Resultante**	**Ejemplos**
Integer	*Integer*	*Double*	2/2=1
Integer	*Double*	*Double*	3/2.0=1.5 1.5/2=0.75
Integer	*char*	*Double*	194/'a'=2.0 'b'/1=98.0
Double	*char*	*Double*	194.0/'a'=2.0
Double	*Double*	*Double*	10.5/2=5.25
char	*char*	*Double*	'a'/'a'=1;

Consideraciones

- Cualquier otra combinación será invalida y se deberá reportar el error.
- Para los casos donde los char se tomen como un número, deberá convertirse a su equivalente en ASCII

5.5.5 Potencia

La operación potencia se produce mediante la resta de los siguientes operandos

Operandos			
Operador	**Operador**	**Tipo Resultante**	**Ejemplos**
Integer	*Integer*	*Integer*	2 ^^ 2 = 4

Consideraciones

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

5.5.6 Modulo

La operación modulo se produce mediante la resta de los siguientes operandos

Operandos			
Operador	**Operador**	**Tipo Resultante**	**Ejemplos**
Integer	*Integer*	*Integer*	2 % 3 = 2 4 % 2 = 0

Consideraciones

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

5.5.7 Unarios

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

Operandos		Tipo Resultante	Ejemplos
Operador	**Operador**		
Integer		*Integer*	-(-4) = 4
Double		*Double*	-(-3.5)=3.5 -8=-8

Consideraciones

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

5.6 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.

5.6.1 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
Operador	**Operador**		
Integer	*Integer*	*Boolean*	1==1 = true 97.0000 == 'a' = true "a"=="A" = false 100 != 'd' = false True != true = false True == false = false "hola"=="hola" = true
Integer	*Double*		
Integer	*char*		
Double	*char*		
Double	*Double*		
char	*char*		
String	*String*		
boolean	*boolean*		

Consideraciones

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

5.6.2 Igualdad de referencias

- El operador de igualdad (===) devuelve true si los operandos tienen la misma dirección de memoria.

Operandos		Tipo Resultante	Ejemplos
Operador	**Operador**		
String	*String*		Global String c= "hola"

Estructura	*Estructura*	*Boolean*	Global String d= ”hola”
			c == d // true
			c === d // false

Arreglo	*Arreglo*		

Consideraciones

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

5.6.3 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**
Operador	**Operador**		
Integer	*Integer*	*Boolean*	2<3=true 2.5>3=false ‘a’>’b’ = false 97.5>’a’=true 100 < ‘a’ = false
Integer	*Double*		
Integer	*char*		
Double	*char*		
Double	*Double*		
char	*char*		

Consideraciones

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

5.7 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.

5.7.1 And, or, not, xor

And: TRUE si ambas expresiones booleanas son TRUE. Se utiliza el símbolo `&&`

Or: TRUE si cualquiera de las dos expresiones booleanas es TRUE. Se utiliza el símbolo `||`

Not: Invierte el valor de cualquier otro operador booleano. Se utiliza el símbolo `!`

XOR: True si las dos expresiones son diferentes. Se utiliza el símbolo `^`

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

Consideraciones

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

5.8 Operadores de aumento y decremento

Estos operadores nos permitirán aumentar o decrementar los valores de alguna variable

Sintaxis

```
Aumento ::= identificador ++
            | identificador --
```

Ejemplo

```
Integer a = 5;
Integer b = a++;
b = a--;
```

Consideraciones

- Este operador solo será válido para identificadores
- Pueden usarse dentro de otras expresiones

5.9 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.

5.9.1 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 ( <condicion> ) <BLOQUE_SENTENCIAS>
    | If ( <condicion> ) <BLOQUE_SENTENCIAS> else <BLOQUE_SENTENCIAS>
    | If ( <condición> ) <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.

5.9.2 Sentencia Switch

Esta sentencia de selección es de bifurcación múltiple. Evalúa una expresión, comparando el valor de esa expresión con la expresión que se encuentra en un case, y ejecuta declaraciones asociadas a ese case, hasta encontrar un break.

Si no ocurre ninguna coincidencia, el programa ejecuta las sentencias asociadas al bloque default, en caso la hubiera, sino, no ejecutará nada. 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 case una vez que se ejecute la instrucción coincidente y continúe la ejecución en la instrucción siguiente.

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
```

```
}

switch (3*54) {
  case 3:
    print(3);
  case 5:
    Print(2);
  case 7:
    Print(5);
  default:
    Print(8);
}

// Salida
// 3
// 2
// 5
// 8
```

Consideraciones

- La lista de sentencias asociada a cada caso no debe llevar **{}**
- El caso default es opcional
- Tanto la expresión de control, como las expresiones asociadas a cada case, deben ser del mismo tipo y deben ser de tipo primitivo o String.

5.9.3 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(true){
 Print("Hola")
}
```

5.9.4 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);
```

5.9.5 Sentencia For

La sentencia cíclica para permitirá inicializar o establecer una variable como variable de control, el ciclo tendrá una condición que se verificará en cada iteración, luego se deberá definir una operación que actualice la variable de control cada vez que se ejecuta un ciclo para luego verificar si la condición se cumple.

Sintaxis

```
For( [inicio]  ?  ;  [Condicion] ? ; [final] ?)  <BLOQUE_SENTENCIAS>
```

- Donde:
  - Inicio: Puede ser una declaración o asignación de una variable existente
  - Condición: Si la condición se evalúa como verdadera se continuará el for hasta que la condición sea evaluada como falsa.
  - Final: Es una asignación o una expresión de aumento o decremento

Tanto inicio, condición y final son opcionales por lo tanto sino viene ninguna estará en un ciclo infinito por no haber condición de salida.

Ejemplo

```
for( int t=0; t<10; t++){
        Print(t);
}

Int j;
for( ; j<10; j++){ // Valido porque la variable j si existe
        Print(j);
}

for( ; ; ){ // Valido, al no tener condición de salida esto genera un ciclo infinito
  Print("Hola");
  Break;
}

Int a = 0;
for( ; a < 3; ){ // Valido porque el aumento se encuentra adentro del for
  a = a + 1;
  print(a);
```

```
}

Int k = 0;
for( ; ; ){ // Valido, la declaración esta fuera del for, la condición y el aumento adentro
  if(k < 5){
    break;
  }
  K = k +1;
}
```

5.10 Sentencias de transferencia

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

5.10.1 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;
}
```

5.10.2 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 (5 < 4){
  continue;
}
```

5.10.3 Return

Finaliza la ejecución de la función y puede especificar un valor para ser devuelto a quien llama a la función. En el caso de las funciones de tipo void, la sentencia return no va acompañada de ninguna expresión.

Ejemplo

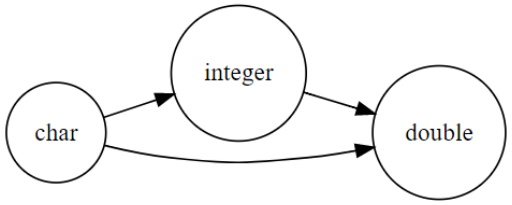
```
If(n==2) {
   Return n;
}
```

5.11 Casteos

Un casteo es un procedimiento para transformar una variable primitiva de referencia de un tipo a otro.

5.11.1 Casteos implícitos

Los casteos implícitos son aquellos que se llevan a cabo automáticamente, sin tener que realizar un procedimiento adicional. En J# los casteos implícitos permitidos son los siguientes:

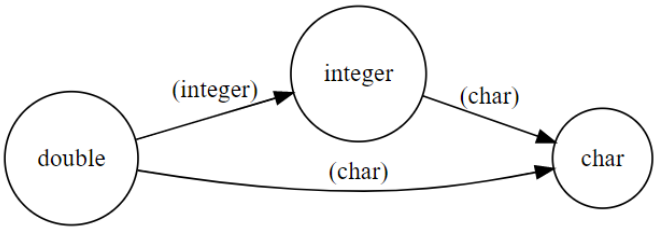


Ejemplo

```
Integer caracter = 'a';                    //caracter toma el valor de 97
double caracter2 = 'a';                    //caracter2 toma el valor de 97.0
double numero = 10                         //caracter2 toma el valor de 10.0
```

5.11.2 Casteos explícitos

El casteo explicito se produce cando se realiza una conversión estrecha, es decir, cuando se coloca un valor grande en un contenedor pequeño.



Sintaxis

```
<casting> = '(' <tipo_primitivo> ')'  <expresión>;
```

Ejemplo

```
Integer var1 = (integer) 10.6;             // 10
Char a = (char) 97.8;                      // a = 'a'
char b = (char) 98                         // b= 'b'
```

Consideraciones:

- Cuando se castea un decimal a entero, se debe aproximar el numero inmediato anterior.
- Cuando se castea un decimal a char, se debe aproximar el numero inmediato anterior.
- Cuando se castea un decimal a char, y el decimal puede considerarse un entero (.00) no se aproxima a ningún valor.

5.12 Funciones

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. Para las funciones que no retornen ningún valor se hará uso de la palabra reservada 'void'

5.12.1 Definición de funciones

Sintaxis

```
[: <VISIBILIDAD>]? (<Type>|`void`)  <ID> (<LISTA_PARAMETROS>) <BLOQUE_SENTENCIAS>
```

Ejemplo

```
public int factorial(Integer n) {
  if (n <= 0){
    return 1;
  } else {
    return n * factorial(n-1);
  }
}

Public integer[] getVal(){
          //retorna un arreglo
}

private void inicializar(Boolean estado) {
 // hace algo
 // no retorna nada
}

private void inicializar(Integer capacidadInicial) {
 // sobrecarga del método inicializar
 // mismo número de parámetros pero diferente tipo
}

Void f1(Integer n) {
  Void  f2(Integer n) {
    Void  f3(Integer n) {
      Print("hola mundo");
    }
    F3(4);
  }
  F2();
}

Void f1(Integer n) {
  Void  f2(Integer n) {
    Void  f3(Integer n) {
```

```
      Print("hola mundo");
      F1(3); // error, no es valido llamar a funciones externas
    }
    F3(4);
  }
  F2(4);
  F3(4);
}
```

Consideraciones

- Sobrecarga de funciones. Pueden existir funciones con el mismo nombre siempre y cuando tenga distinto número de parámetros o distinto tipo.
- Las funciones y variables si pueden tener el mismo nombre
- Para las funciones de tipo 'void' la sentencia 'return' es opcional. En caso contrario se debe validar que exista una sentencia 'return' para cada flujo que pueda tomar la función.
- Las funciones pueden retornar tipos primitivos, estructuras, arreglos y también `null`
- La visibilidad de las funciones puede ser `public` o `private` esto permite importar una función de otro archivo. Sino aparece por defecto es `private`
- Las funciones pueden contener otras funciones dentro de esta.
- Las funciones internas solo pueden llamar a funciones internas, no es válido que una función interna llame a una función fuera de su ámbito.
- Las funciones internas solo existirán en el ámbito de la función, pero de igual manera cumplirán las reglas de sobre carga que se mencionan anteriormente.
- Las funciones internas se deben reportan de igual manera en la tabla de simbolos.

5.12.2 Parámetros de las funciones

Los parámetros en las funciones son variables que podemos utilizar mientras nos encontremos en el ámbito de la función. Los tipos primitivos cómo parámetros se pasan por valor, los tipos estructura, arreglos y cadenas se pasan por referencia.

Ejemplo

```
private void operar(String nombre, Integer edad) {
  edad = edad + 10;
  nombre.toUpperCase();
}

Integer miEdad = 18;
Integer miNombre = "Emily";

operar(miNombre, miEdad);

// al finalizar la ejecución miEdad seguirá en 18 pues se ha pasado por valor
// al finalizar la ejecución miNombre será "EMILY" pues se ha pasado por referencia
```

Es posible pasar estructuras, arreglos y cadenas por valor, para esto hacemos uso del operador **`$`**. Este operador no tiene ningún efecto en los primitivos.

Ejemplo

```
# La función recibe un parámetro de tipo arreglo. En la primera llamada los valores no se modifican y en la segundo si.

Arreglo := {6,7,8,9,10};

Public void modifyArray( integer[] arreglo){
        for(int i =0; i<arreglo.length;i++){
                Arreglo[i]=i;
        }
}

modifyArray($arreglo);

for(int j= 0; j < arreglo.length; j++){
  print(arreglo[j]);  // Imprime 6,7,8,9,10
}


modifyArray(arreglo);

for(int j= 0; j < arreglo.length; j++){
  print(arreglo[j]);  // Imprime 1,2,3,4,5
}
```

5.12.3 Llamada a funciones

Los parámetros en la llamada a una función son los argumentos de la función. Las funciones pueden ser llamadas desde una expresión o cómo una instrucción

Ejemplo

```
// Llamando a una función como instrucción
Factorial(5);  // Se ignora el valor de retorno

// Llamando a una función como expresión
Integer valor = Factorial(5); // El valor de retorno se asigna a valor
```

También es posible llamar a las funciones por el nombre de los parámetros en lugar de usar las posiciones. Para ello se pasa el nombre de los parámetros y su valor, de esta forma el orden no importará.

Ejemplo

```
// Definición de una función
private void operar(Double examen, Double zona, String carnet) {
  // hace algo con los parámetros
}

// Se pasan los parámetros por nombre y no por posición
operar(carnet=”202012345”, zona = calcularZona(), examen = 16.5 );
```

Consideraciones

- Se debe verificar que la cantidad de parámetros y el tipo coincida con los de la función que se está llamando.
- Solo se puede hacer uso de un tipo de llamado a la vez, es decir o todos los parámetros son por posición o todos por nombre.
- Para el llamado por nombre se debe validar que los nombres también coincidan con los definidos por la función.

5.12.4 Función print

Esta función nos permitirá imprimir en consola cualquier valor

Sintaxis

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

Ejemplo

```
Integer[] var1 = {1,2,3};
Print(var1[0]);              // Imprime 2

Const x :=10;

Print(x);
```

Consideraciones

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

5.13 Excepciones

Una excepción es la indicación de que se produjo un error en el programa. Las excepciones, como su nombre lo indica, se producen cuando la ejecución de un método no termina correctamente, sino que termina de manera excepcional como consecuencia de una situación no esperada.

5.13.1 Estructuras de excepción

J# Provee un conjunto limitado de excepciones que deben ser validadas por el compilador en tiempo de ejecución (en 3DC). Estas son:

5.13.1.1 Arithmetic Exception

Ocurre cuando una operación aritmética viola las reglas definidas sobre las operaciones aritmeticas, por ejemplo, una división por 0, o alguno de los elementos listados en sus consideraciones.

5.13.1.2 IndexOutOfBoundException

Esta excepción ocurre cuando intentamos acceder a una posición del arreglo que excede los limites predefinidos.

5.13.1.3 UncaughtException

Ocurre cuando se produce una excepción fuera de un try.

5.13.1.4 NullPointerException

Ocurre cuando se intenta acceder a los elementos de una estructura que no ha sido inicializada, o bien, un arreglo.

5.13.1.5 InvalidCastingException

Ocurre cuando se violan las reglas de los casteos no primitivos. Por ejemplo, el String no tiene el formato correcto para convertirlo a un número o no se puede realizar la conversión entre estructuras.

5.13.1.6 HeapOverflowError

Ocurre cuando el espacio máximo del heap (64 Megabytes) se desborda.

5.13.1.7 StackOverflowError

Ocurre cuando el espacio máximo del stack (64 Megabytes) se desborda.

5.13.2 Manejo de excepciones

J# Posee un conjunto de sentencias para el manejo de excepciones, estas son:

5.13.2.1 Sentencia throw

La sentencia **throw** lanza automáticamente una excepción. El resultado es una transferencia inmediata de control que puede aparecer en cualquier bloque de sentencias mientras que se encuentre dentro del cuerpo de una sentencia try que capte el valor arrojado. Si no tal sentencia, entonces se lanza una UncaughtException y se detiene la ejecución del programa.

Sintaxis

```
<throw_statement> ::= “throw” <expression> “;”
```

Ejemplo

```
var divisor := 0;
integer dividendo = 10;
try {
        if (divisor == 0) {
                throw strc ArithmeticException();
        }
 } catch (ArithmeticException e){
        print (e.message) ;
```

```
}
```

5.13.2.2 Sentencia try catch

La sentencia **throw** lanza automáticamente una excepción. El resultado es una transferencia inmediata de control que puede aparecer en cualquier bloque de sentencias mientras que se encuentre dentro del cuerpo de una sentencia try que capte el valor arrojado. Si no tal sentencia, entonces se lanza una UncaughtException y se detiene la ejecución del programa.

Sintaxis

```
< try_statement> ::=  “try” <BLOQUE_SENTENCIAS> “catch” “(” <declaration>
“)”<BLOQUE_SENTENCIAS>
```

Ejemplo

```
divisor := 0;
integer dividendo = 10;
try {
        if (divisor == 0) {
                throw strc ArithmeticException();
        }
 } catch (ArithmeticException e){
        print (e.message) ;                    // Imprime ”se arrojó una excepción aritmética en
                                               // línea 10 columna 4”
}
```

5.13.2.3 Atributo message

Todas las estructuras de excepción tienen un atributo denominado message. El estudiante es libre de implementar esta funcionalidad como desee conveniente pero el imprimir este atributo deberá dar una descripción de qué tipo de error se produjo y en que fija y columna.

5.14 Estructuras de datos

5.14.1 La estructura String y sus funciones

Un String es un tipo de dato compuesto por una cadena de caracteres. Las variables de este tipo se definirán con la palabra **no reservada** String. Este tipo de dato tiene como valor por defecto ‘null’.

```
String cadena = “Hola”;

String String = null;
```

5.14.1.1 Funciones propias

String posee una serie de funciones por defecto que le permiten darle una funcionalidad adicional en comparación con los tipos de datos primitivo. Estos son:

5.14.1.1.1 toCharArray()

Devuelve un arreglo de caracteres que contiene la misma secuencia de caracteres que un String.

```
Global var1 = “Cadena”;
String [] cadena = var1.toCharArray();
```

5.14.1.1.1 length()

Devuelve un arreglo de caracteres que contiene la misma secuencia de caracteres que un String.

```
Global var1 = “Cadena”;
Integer longitud = var1.length();
```

5.14.1.1.2 toUpperCase()

Devuelve un arreglo de caracteres que contiene la misma secuencia de caracteres que un String.

```
Global var1 = “Cadena”;
String cadena = var1.toUpperCase();
```

5.14.1.1.3 toLowerCase()

Devuelve un arreglo de caracteres que contiene la misma secuencia de caracteres que un String.

```
Global var1 = “Cadena”;
String cadena = var1.toUpperCase().toLowerCase();          //Concatenacion de funciones
```

5.14.1.1.4 charAt()

Devuelve un arreglo de caracteres que contiene la misma secuencia de caracteres que un String.

```
Global var1 = “Cadena”;
char cad = var1.toUpperCase().toLowerCase().charAt(0);          //Concatenacion de funciones
```

Consideraciones:

- **Es permitida la concatenación de llamadas a funciones, acceso a arreglos y atributos de una estructura.**

5.14.2 Arreglos

Los arreglos son estructuras de datos que almacenan de forma contigua valores del mismo tipo. Los arreglos son expresiones como tal y por lo tanto puede ser utilizados como variables, para paso de parámetros, en los retornos de funciones y todos los lugares donde puedan utilizarse expresiones.

En J# hay dos diferentes maneras de definir los arreglos, estos son:

Sintaxis

```
<Array_init > ::=  strc <Tipo> <LISTA_COR_NUM>
                    | <ARRAY_VALUES>

<ARRAY_VALUES> ::= { <LISTA_EXPRESION> }

<LISTA_EXPRESION> ::= <LISTA_EXPRESION> , <ELEMENTO>
```

```
                                <ELEMENTO>

<ELEMENTO> ::= <Expresion>
                        |<ARRAY_VALUES>

<LISTA_COR_NUM> ::= <LISTA_COR_NUM> '[' <Expresión> ']'
                                | '[' <Expresión> ']'
```

Ejemplo

```
Define Estudiante as [ integer edad, String cadena="Hola"];

Estudiante[] est = strc Estudiante[2];

Integer[] var1 = {1,2,3,4};
double[] pra = {1,2,3};

Var Var2 := {"Hola","Luis","Como","Estas"};

Integer[] vale = strc Integer[6];
```

Consideraciones

- Los límites de los arreglos esta definidos desde 0 hasta n-1
- Al declarar un arreglo de estructuras, estas estructuras apuntaran a null hasta que se cree una instancia sobre cada una de sus posiciones.
- Por defecto, los vectores apuntan a null.

5.14.2.1 Acceso a los elementos de un arreglo

Para acceder a los elementos basta con una lista de corchetes en la que cada par de corchetes indique el valor al cual se quiere acceder.

```
Define Estudiante as [ integer edad, String cadena="Hola"];
Estudiante[] est = strc Estudiante[2];
Integer[] var1 = {1,2,3,4,5,6,7};
Integer[] var2 = {1,2};

Var2[1] = var1[3];

Est[0] = strc Estudiante();                    // Crea una nueva instancia de estudiante

Print(var1[0]);                                //imprime '1'

funcion(var1)[2].atributo[2].atributo[9]          //concatenación de accesos
Funcion1(acceso[3].atributo).atributo[3]
```

5.14.2.2 El atributo length

Los arreglos poseen un atributo por defecto, length. Este atributo retorna el tamaño de una dimensión del arreglo. Por ejemplo

```
Define Estudiante as [ integer edad, String cadena= ”Hola”];
Estudiante[] est = strc Estudiante[6];

Print(est.length);                    //Imprime 2
Print(est.length);                    //Imprime 6
```

5.14.2.3 La función linealize

La función linealiza permitirá crear una nueva representación lineal de un arreglo.

```
Var1 := {7,8,9,10,11,12}              //Se infiere un arreglo de integers

Var intvar = var1.linealize()[6]      // Imprime 12
```

Consideraciones:

- Linealize crea un nuevo arreglo, no se debe modificar el arreglo original.

5.14.3 Estructuras

Las estructuras son un conjunto de elementos de diferente tipo agrupados bajo un mismo nombre. Las estructuras pueden contener otras estructuras dentro, así como también arreglos de una o muchas dimensiones.

5.14.3.1 Definición de estructuras

Cuando creamos una estructura, únicamente estamos creando su definición. Esta definición nos permitirá instanciar elementos de una misma estructura, pero con una diferente referencia.

Sintaxis

```
< Struct_Definition > ::=  “define” id “as” “[”  <LISTA_ATRIBUTOS>  “]” ;

<LISTA_ATRIBUTOS> ::= <LISTA_ATRIBUTOS> ‘,’ <ATRIBUTO>
                    | <ATRIBUTO>

<ATRIBUTO> ::= <type> id
             | <type> id = <expresión>
```

Ejemplo

```
Define Estudiante as [ integer edad, String cadena=”Hola”];

Define Libro as [ String [] generos, integer CUI];
```

Consideraciones

- Si alguno de los atributos de la estructura esta igualado a una expresión, todas las instancias de esa estructura deben ser creada con ese valor.
- Si el atributo no está igualado a ninguna expresión, esta tomara el valor por defecto al ser instanciada.
- No puede declararse diferentes estructuras con el mismo nombre, de lo contrario deberá reportarse un error.
- Las estructuras pueden contener otras estructuras dentro.
- Las estructuras pueden contener arreglos dentro y se declararan de la misma forma como en las variables.
- Si la estructura contiene un elemento de su mismo tipo, su atributo interno deberá apuntar a null.
- El valor por defecto de las estructuras es null.

5.14.3.2 Creación de estructuras

Como se menciónó anteriormente, las estructuras deben de ser instanciadas antes de poder utilizarse. El proceso de instanciar una variable no es más que reservar su respectivo espacio en el heap.

Sintaxis

```
<Struct_Declaration> ::= “strc” id <LISTA_COR>
                        | “strc” id ‘(‘ ‘)’


<LISTA_COR> ::= <LISTA_COR> []
              | []
```

Ejemplo

```
Global est := strc Estudiante();

Function_call(strc Estudiante());

Estudiante [] = strc Estudiante[];
```

- Si una estructura se declara un arreglo, la cantidad de dimensiones debe ser la misma que las de la variable donde se desee asignar.

5.14.3.3 Acceso a los atributos de una estructura

Para acceder a los elementos de una estructura usaremos el símbolo ‘.’

Ejemplo

```
Define Libro as [ String [] generos, integer CUI];
Define Estudiante as [ integer edad, Estudiante novia];

Var E1 := strc Estudiante();
E1.novia = strc Estudiante();
```

```
Libro lib = strc Libro ();
print(lib.generos[0]);

function_call(lib.CUI);

print(e1.novia.edad);
```

5.14.3.4 Funciones de una estructura

En el lenguaje J#, las estructuras cuentan con una serie de funciones propias tales como:

5.14.3.4.1 Size

Retorna la cantidad de espacio de memoria que ocupa una estructura en el Heap.

```
Define Estudiante as [ integer edad, Estudiante novia];
Define Libro as [ String [] generos, integer CUI,String nombre];


Var E1 := strc Estudiante();
Global l1 := strc Libro();

Print(e1.size());                    //Imprime el valor de 2
Print(l1.size());                    //Imprime el valor de 3
```

5.14.3.4.2 getReference

Retorna el valor del heap en donde comienza un objeto

```
Define Estudiante as [ integer edad, Estudiante novia];
Define Libro as [ String [] generos, integer CUI,String nombre];


Var E1 := strc Estudiante();         // Almacena el apuntador a 500
Global l1 := strc Libro();

Print(e1.size());                    //Imprime el valor de 500, por ejemplo
Print(l1.size());                    //Imprime el valor de 503
```

5.14.3.4.3 instanceOf

Recibe como parámetro el identificador de una estructura, e indica si la instancia actual es una instancia de dicha estructura

```
Define Estudiante as [ integer edad, Estudiante novia];
Define Libro as [ String [] generos, integer CUI,String nombre];


Var E1 := strc Estudiante();         // Almacena el apuntador a 500
Global l1 := strc Libro();
```

```
If(e1.instanceof(Estudiante)){
        Print("True");                    // Imprimiría true
} else{
        Print("false");
}
```

6 Reportes Generales

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

6.1 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.

6.1.1 Tipos de errores

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

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos

6.1.2 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.
- El reporte de errores debe ser desplegado en la interfaz de la aplicación.

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**

6.1.3 Método de recuperación

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

6.1.3.1 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.
```

6.1.3.2 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 está definida, se descarta la asignación completa
```

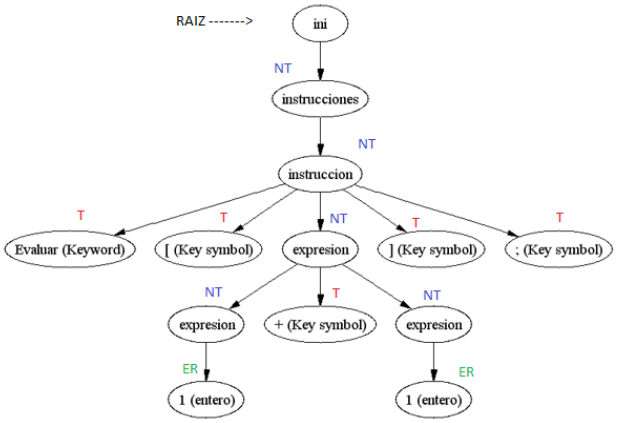
6.2 Reporte de tabla de símbolos

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

6.3 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.



7 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)[1]

8 El formato de código intermedio

El formato estándar de código de tres direcciones es una secuencia de proposiciones que tiene formato general **X** = **Y** *op* **Z**, donde **X**, **Y**, **Z** son nombres, constantes o variables temporales que han sido generadas por el compilador, y *op* representa a operadores aritméticos o relacionales. Este código deberá generarse de acuerdo con el estándar de código intermedio tres direcciones visto en clase.

8.1 Definición léxica

La definición léxica involucra todos los caracteres que son permitidos por el lenguaje. Podemos dividir estos en grupos más concretos.

[1] No es válido presentar la gramática que se utilizó para el generador de analizadores sintácticos. La gramática debe se entregada de forma escrita (no capturas) como la que se realiza en un examen parcial, caso contrario obtendrá una nota de 0 puntos.

8.1.1 finalización de línea

El intérprete de C3D dividirá la entrada en líneas, estas líneas estarán divididas mediante componentes léxicos que determinen la finalización de estas y son los siguientes:

- Carácter salto de línea (LF ASCII)
- Carácter retorno de carro (CR ASCII)
- Carácter retorno de carro (CR ASCII) seguido de carácter salto de línea (LF ASCII)

8.1.2 Espacios en blanco

Los espacios en blancos definirán la división entre los componentes léxicos. Todos los componentes siguientes no deberán ser tomados en cuenta por el analizador sintáctico. Se considerarán espacios en blanco los siguientes elementos:

- Espacio (SP ASCCI)
- Tabulación horizontal (HT ASCII)
- Caracteres de finalización de línea

8.1.3 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 los símbolos “##” 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
#*
        Este es un comentario multilínea
        Hola
        Adios
*#
```

8.1.4 Temporales

Los temporales serán creados por el compilador durante el proceso de generación de código intermedio. Los temporales deberán empezar con la letra “t” seguida de uno o más dígitos.

```
t[0-9]+
```

8.1.5 Etiquetas

Las etiquetas serán creadas por el compilador en el proceso de generación de código intermedio. Las etiquetas deberán empezar con la letra “L” seguida de un número.

```
L[0-9]+
```

8.1.6 Identificadores

Un identificador será utilizado para dar un nombre a métodos.

Un identificador es una secuencia de caracteres alfabéticos **[A-Z a-z]** incluyendo el guion bajo **[_]** o dígitos **[0-9]** que comienzan con un carácter alfabético.

```
[a-zA-Z][_a-zA-Z0-9ñÑ]+
```

Algunos ejemplos serian:

```
Luis_pavel_rainman_10      #Identificador válido
C86                        #Identificador válido

_Juan_Carlos               #Identificador inválido
85_Hola                    #Identificador inválido
```

8.1.7 Palabras reservadas

Son palabras que no podrán ser utilizadas como identificadores ya que representan algo específico y necesario dentro del **C3D**

```
var          stack        heap         goto
begin        end          call         print
if
```

8.1.8 Símbolos terminales

Símbolos			
Suma (+)	Resta (-)	Multiplicación (*)	División (/)
Modulo (%)	Mayor (>)	Menor (<)	Mayor Igual (>=)
Menor igual (<=)	Igual (==)	No igual (<>)	Punto y coma (;)
Coma (,)	Corchete Izq ([)	Corchete Der (])	Paréntesis Izq “(”
Paréntesis Der “)”	Asignación (=)		

8.1.9 Tipos de datos

El código de 3 direcciones contará únicamente de números enteros y decimales. **NO EXISTEN CADENAS, CARACTERES NI BOOLEANOS.**

- Enteros: **[0-9]+**
- Decimales: [0-9]+“.” [0-9]+

Consideraciones

- Si se desea imprimir una expresión y la parte decimal cuenta con 8 dígitos o más, deberá imprimirse en notación científica.
- Si se desea imprimir una expresión y la parte entera cuenta con 8 dígitos o más, deberá imprimirse en notación científica.
- Si tanto la parte entera como la parte decimal tiene 8 dígitos o más, se le dará

prioridad a la parte entera.

8.2 Definición de sintaxis

La definición de sintaxis es el conjunto de todas las reglas gramaticales sobre las que se rige el lenguaje. Puede dividirse en los siguientes grupos. Cada una de las sentencias del lenguaje terminaran con punto y coma. Cabe destacar que el **estudiante** puede realizar las **modificaciones de sintaxis** que considere convenientes **siempre y cuando** la gramática **continúe aceptando el mismo lenguaje**.

8.2.1 Encabezado

Al principio del código de 3D es necesario declarar todas las variables temporales que ser vayan a utilizar, los punteros P y H, así como inicializar el heap y el stack. NO SE PUEDEN DECLARAR NINGUN TIPO DE VARIABLES TEMPORALES EN NINGUNA OTRA SECCION DE ESTE ARCHIVO. Los encabezados de los archivos deberían verse similares a estos.

```
Var t1,t2,t3…,tn;       //Sección de variables temporales
Var P,H;                //Sección de apuntadores
Var Heap[];             //Sección de Heap
Var Stack[];            //Sección de Stack
```

8.2.2 Declaración de variables

Se entiende por variables los punteros y temporales. Las declaraciones de variables **únicamente** pueden realizarse **al principio** del archivo de C3D. Las variables pueden declararse de la siguiente manera.

```
<var_declaration_list> ::=   <var_declaration_list> ‘,’ id
                             | “var” id
```

NOTA: SE PROHÍBE EL USO DE AMBITOS ANIDADOS PARA LA EJECUCIÓN DE CÓDIGO DE 3 DIRECCIONES. LAS LLAMADAS RECURSIVAS DEBEN DE REALIZARSE ALMACENANDO LOS TEMPORALES EN EL STACK. CUALQUIER ESTUDIANTE QUE SE SORPRENDA INTENTANDO HACER TRAMPA TENDRÁ AUTOMÁTICAMENTE NOTA DE 0 PUNTOS.

8.2.3 Operadores aritméticos

Las operaciones permitidas por el lenguaje son las siguientes:

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

8.2.4 Operadores relacionales

Las operaciones permitidas por el lenguaje son las siguientes:

Operador	Operación
<	Menor
<=	Menor igual
>	Mayor
>=	Mayor igual
<>	No igual
==	igual

8.2.5 Operadores lógicos

Los operadores lógicos, !, &&, || y ^ serán implementados utilizando la técnica de backpatch vista en clase.

8.2.6 Asignación

Es el proceso de establecer un valor a una variable previamente creada. La sintaxis es la siguiente:

```
<assigment> ::=  id "=" id op id ";"
                 | id "=" id ";"
                 | id "=" num_literal ";"
                 | id "=" Stack'[' id2 ']' ";"
                 | id "=" Stack'[' num_literal ]' ";"
                 | id "=" Heap[' id2 ']' ";"
                 | id "=" Heap[' num_literal ]' ";"
```

8.2.7 Destino de un salto

Cuando se realiza un salto, ya sea condicional o incondicional, el destino siempre será una etiqueta

```
<destiny_of_jump> ::= label ":"
```

Consideraciones:

- Nótese que no existe la sintaxis para múltiples etiquetas. Esto se debe a que la técnica de backpatch elimina este problema.

8.2.8 Salto incondicional

En este tipo de salto no es necesario evaluar una condición para saltar. Se salta automáticamente.

```
<inconditional_jump> ::= "goto" label ";"
```

8.2.9 Salto condicional

Este salto solo se realizará si la condición a evaluar es verdadera, caso contrario continuara la ejecución de forma top-down.

```
<conditional_jump_if>::= "if" "(" <rel_op> ")" "goto" label ";"
```

8.2.10 Parcheo de retroceso o backpatch

Es **obligatorio** que la generación de código **se implemente de esta manera**, de lo **contrario** todo el formato de código de 3 direcciones **se considerará como incorrecto.**[2]

8.2.11 Declaración de métodos

La sintaxis para la declaración de métodos será la siguiente

```
<method_declaration>::= "proc" id "begin"
                              <3D_statement_block>
                        "end"
```

```
proc ejemplo begin
     t152 = P + 2;
     t153 = Stack[t152];
     if (t153 <= 1) goto L76;
     Stack[P] = 1;
     goto L74;
     goto L75;
     L76:
end
```

Consideraciones:

- En código de 3 direcciones para el paso de parámetros y el retorno de valores se utiliza el stack.

8.2.12 Llamadas a métodos

La sintaxis para la invocación de métodos será la siguiente

```
<method_invocation>::= "call" id ";"
```

8.2.13 Sentencia Print

Esta función nos permitirá imprimir en consola cualquier valor que deseemos. Esta sentencia recibirá como parámetro una bandera que nos indicará el tipo de dato a imprimir.

```
<print_statement>::=  "print" "(" """ "%" char_terminal """ "," id_o_valor
```

[2] Como se detalla en el libro oficial del curso, capitulo 6, sección 6.7, página 410, El backpatch "consiste en pasar listas de saltos como atributos sintetizados. En específico, cuando se genera un salto, el destino de éste se deja temporalmente sin especificar. Cada salto de este tipo se coloca en una lista de saltos cuyas etiquetas deben llenarse cuando se pueda determinar la etiqueta apropiada. Todos los saltos en una lista tienen la misma etiqueta de destino."

```
")" ";"
```

Parámetro	Función
"%c"	Imprime el valor como un carácter
"%i"	Imprime el valor como un entero
"%d"	Imprime el valor como un decimal

```
print("%i",97)    #Imprime 97
print("%c",97)    #Imprime 'a'
t1 = 40.5;
print("%d",t1)    # Imprime 40.5
print("%i",t1)    # Imprime 41
print("%c",t1)    # Imprime ')'
```

Consideraciones:

- 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
- En código de 3 direcciones **no existe** la sentencia **println**

9 Entorno de ejecución

Como ya se expuso en otras secciones de este documento, el proceso de compilación genera el código intermedio y este es el único que se va a ejecutar, siendo indispensable para el flujo de la aplicación. En el código intermedio NO existen cadenas, operaciones complejas, llamadas a métodos con parámetros y muchas otras cosas que sí existen en los lenguajes de alto nivel. Esto debido a que el código intermedio busca ser una representación próxima al código máquina, por lo que todas las sentencias de las que se compone se deben basar en las estructuras que componen el entorno de ejecución. Típicamente se puede decir que los lenguajes de programación cuentan con dos estructuras para realizar la ejecución de sus programas en bajo nivel, la pila (Stack) y el montículo (Heap), en la siguiente sección se describen estas estructuras.

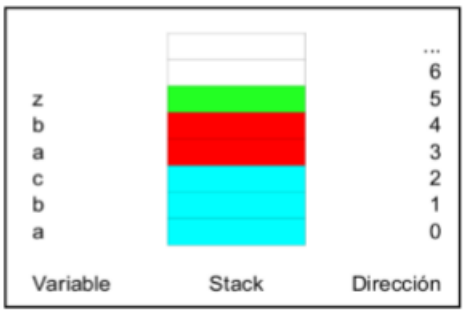
9.1 Estructuras del entorno de ejecución

Las estructuras del entorno de ejecución no son más que arreglos de bytes que emplean ingeniosos mecanismos para emular las sentencias de alto nivel. Este proyecto consistirá de dos estructuras indispensables para el manejo de la ejecución, siendo estas, Stack y Heap, las cuales al ser empleadas en el código de tres direcciones determinará el flujo y la memoria de la ejecución, estas estructuras de control serán representadas por arreglos de números con punto flotante, esto con el objetivo de que se pueda tener un acceso más rápido a los datos sin necesidad de leer por grupos de bytes y convertir esos bytes al dato correspondiente, como normalmente se hace en otros lenguajes, por ejemplo, en Java, un int ocupa 4 bytes, un boolean ocupa 1 solo byte, un double ocupa 8 bytes, un char 2 bytes, etc. **Por facilidad, tomaremos la convención de que todos los tipos de dato tendrán tamaño 1.**

9.1.1 El Stack y su puntero

El stack es la estructura que se utiliza en código intermedio para controlar las variables locales, y la comunicación entre métodos (paso de parámetros y obtención de retornos en llamadas a métodos). Se compone de ámbitos, que son secciones de memoria reservados exclusivamente para cierto grupo de sentencias.

Cada llamada a método o función que se realiza en el código de alto nivel de J# cuenta con un espacio propio en memoria para comunicarse con otros métodos y administrar sus variables locales. Esto se logra modificando el puntero del Stack, que en el proyecto se identifica con la letra P.

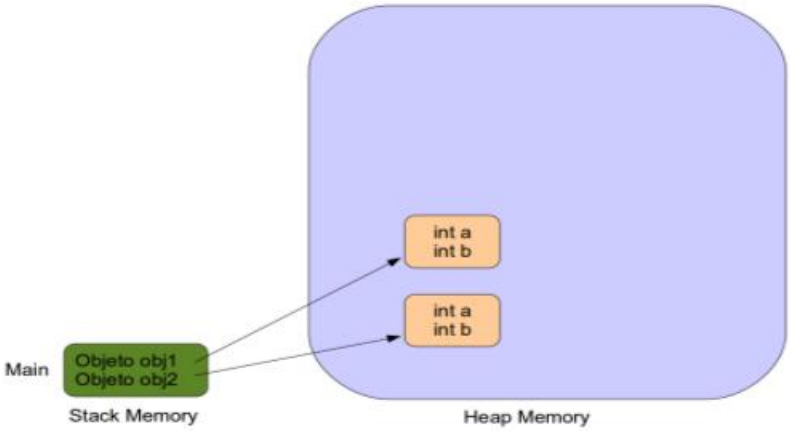


9.1.2 El Heap y su puntero

El **Heap** (o en español, montículo) es la estructura de control del entorno de ejecución encargada de guardar los objetos, arreglos o referencias a variables globales.

El puntero **H**, a diferencia de **P** (que aumenta y disminuye según lo dicta el código intermedio), solo aumenta y aumenta, su función es brindar siempre la próxima posición libre de memoria. A continuación, se muestra un ejemplo de cómo se vería el Heap después de almacenar dos Objetos.

Esta estructura también será la encargada de almacenar las cadenas de texto, guardando únicamente en cada posición el ASCII de cada uno de los caracteres que componen la cadena a guardar.



9.1.3 Acceso a estructuras del entorno de ejecución

Para asignar un valor a una estructura del sistema es necesario colocar el identificador del arreglo (Stack o Heap), la posición donde se desea colocar el valor seguido del valor a asignar con la siguiente sintaxis:

```
Stack[id_o_valor] = id_o_valor;
Heap[id_o_valor] = id_o_valor;
```

Para la obtención de un valor de cualquiera de las estructuras (Stack o Heap) se realizará con la siguiente sintaxis:

```
id = Stack[id_o_valor];
id = Heap[id_o_valor]
```

10 Optimización de código intermedio

J# deberá optimizar el código de tres direcciones como parte de la compilación para su posterior ejecución. Para el proceso de optimización se utilizará el método conocido como mirilla (Peephole) y optimización por bloque.

10.1 Optimización por mirilla

El método de mirilla consiste en utilizar una ventana que se mueve a través del código de 3 direcciones, la cual se le conoce como mirilla, en donde se toman las instrucciones dentro de la mirilla y se sustituyen en una secuencia equivalente que sea de menor longitud y lo más rápido posible que el bloque original. El proceso de mirilla permite que por cada optimización realizada con este método se puedan obtener mejores beneficios.

El método de mirilla consiste en utilizar una ventana que se mueve a través del código de 3 direcciones, la cual se le conoce como mirilla, en donde se toman las instrucciones dentro de la mirilla y se sustituyen en una secuencia equivalente que sea de menor longitud y lo más rápido posible que el bloque original. El proceso de mirilla permite que por cada optimización realizada con este método se puedan obtener mejores beneficios.

Los tipos de transformación para realizar la optimización por mirilla serán los siguientes:

- Eliminación de instrucciones redundantes de carga y almacenamiento.
- Eliminación de código inalcanzable.
- Optimizaciones de Flujo de control.
- Simplificación algebraica y reducción por fuerza.

Cada uno de estos tipos de transformación utilizados para la optimización por mirilla, tendrán asociadas reglas las cuales en total son 18, estas se detallan a continuación:

10.1.1 Eliminación de instrucciones redundantes de carga y almacenamiento

10.1.1.1 Regla 1

Si existe una asignación de valor de la forma a = b y posteriormente existe una asignación de forma b = a, se eliminará la segunda asignación siempre que a no haya cambiado su valor. Se deberá tener la seguridad de que no exista el cambio de valor y no existan etiquetas entre las 2 asignaciones:

Ejemplo	Optimización
t2 = b; b = t2;	t2 = b

10.1.2 Eliminación de código inalcanzable

Consistirá en eliminar las instrucciones que nunca serán utilizadas. Por ejemplo, instrucciones que estén luego de un salto condicional, el cual direcciona el flujo de ejecución a otra parte y nunca llegue a ejecutar las instrucciones posteriores al salto condicional. Las reglas aplicables son las siguientes:

10.1.2.1 Regla 2

Si existe un salto condicional de la forma Lx y exista una etiqueta Lx:, todo código contenido entre el goto Lx y la etiqueta Lx, podrá ser eliminado siempre y cuando no exista una etiqueta en dicho código.

Ejemplo	Optimización
goto L1; <instrucciones> L1:	L1:

10.1.2.2 Regla 3

Si existe un alto condicional de la forma if <cond> goto Lv; goto Lf; inmediatamente después de sus etiquetas Lv: <instrucciones> Lf: se podrá reducir el número de saltos negando la condición, cambiando el salto condicional hacia la etiqueta falsa Lf: y eliminando el salto condicional innecesario a goto Lf y quitando la etiqueta Lv:.

Ejemplo	Optimización
if a == 10 goto L1; goto L2; L1: <instrucciones> L2:	if a != 10 goto L2; <instrucciones> L2:

10.1.2.3 Regla 4

Si se utilizan valores constantes dentro de las condiciones de la forma if <cond> goto Lv; goto Lf; y el resultado de la condición es una constante verdadera, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta falsa Lf.

Ejemplo	Optimización
`if 1 == 1 goto L1;` `goto L2;`	`goto L1;`

10.1.2.4 Regla 5

Si se utilizan valores constantes dentro de las condiciones de la forma if <cond> goto Lv; goto Lf; y el resultado de la condición es una constante falsa, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta verdadera Lv.

Ejemplo	Optimización
`if 1 == 0 goto L1;` `goto L2;`	`goto L2;`

10.1.3 Optimizaciones de flujo de control

Con frecuencia los algoritmos de generación de código intermedio producen saltos hacia saltos, saltos condicionales hacia saltos condicionales o saltos condicionales hacia saltos. Los saltos innecesarios podrán eliminarse, con las siguientes reglas:

10.1.3.1 Regla 6

Si existe un salto incondicional de la forma goto Lx donde existe la etiqueta Lx: y la primera instrucción, luego de la etiqueta, es otro salto, de la forma goto Ly; se podrá realizar la modificación al primer salto para que sea dirigido hacia la etiqueta Ly: , para omitir el salto condicional hacia Lx.

Ejemplo	Optimización

`goto L1;`
`<instrucciones>`
`L1:`
`goto L2;`
	`goto L2;`
	`<instrucciones>`
	`L1:`
	`goto L2;`

10.1.3.2 Regla 7

Si existe un salto incondicional de la forma if <cond> goto Lx; y existe la etiqueta Lx: y la primera instrucciones luego de la etiqueta es otro salto, de la forma goto Ly; se podrá realizar la modificación al primer salto para que sea dirigido hacia la etiqueta Ly: , para omitir el salto condicional hacia Lx.

Ejemplo	Optimización
`if t9 >= t10 goto L1;`	`if t9 >= t10 goto L2;`

<instrucciones> L1: goto L2;	<instrucciones> L1: goto L2;

10.1.4 Simplificación algebraica y por fuerza

Con frecuencia los algoritmos de generación de código intermedio producen saltos hacia saltos, saltos condicionales hacia saltos condicionales o saltos condicionales hacia saltos. Los saltos innecesarios podrán eliminarse, con las siguientes reglas:

10.1.4.1 Regla 8

Eliminación de las instrucciones que tenga la siguiente forma:

Ejemplo	Optimización
x = x + 0;	#Se elimina la instrucción

10.1.4.2 Regla 9

Eliminación de las instrucciones que tenga la siguiente forma:

Ejemplo	Optimización
x = x - 0;	#Se elimina la instrucción

10.1.4.3 Regla 10

Eliminación de las instrucciones que tenga la siguiente forma:

Ejemplo	Optimización
x = x * 1;	#Se elimina la instrucción

10.1.4.4 Regla 11

Eliminación de las instrucciones que tenga la siguiente forma:

Ejemplo	Optimización
x = x / 1;	#Se elimina la instrucción

Para las reglas 12, 13, 14, 15 es aplicable la eliminación de instrucciones con operaciones de variable distinta a la variable de asignación y una constante, sin modificación de la variable involucrada en la operación, por lo que la operación se descartará y se convertirá en una asignación.

10.1.4.5 Regla 12

Ejemplo	Optimización
`x = y + 0;`	`x = y;`

10.1.4.6 Regla 13

Ejemplo	Optimización
`x = y - 0;`	`x = y;`

10.1.4.7 Regla 14

Ejemplo	Optimización
`x = y * 1;`	`x = y;`

10.1.4.8 Regla 15

Ejemplo	Optimización
`x = y / 1;`	`x = y;`

Se deberá realizar la eliminación de reducción por fuerza para sustituir por operaciones de alto costo por expresiones equivalentes de menor costo.

10.1.4.9 Regla 16

Ejemplo	Optimización
`x = y * 2;`	`x = y + y;`

10.1.4.10 Regla 17

Ejemplo	Optimización
`x = y * 0;`	`x = 0;`

10.1.4.11 Regla 18

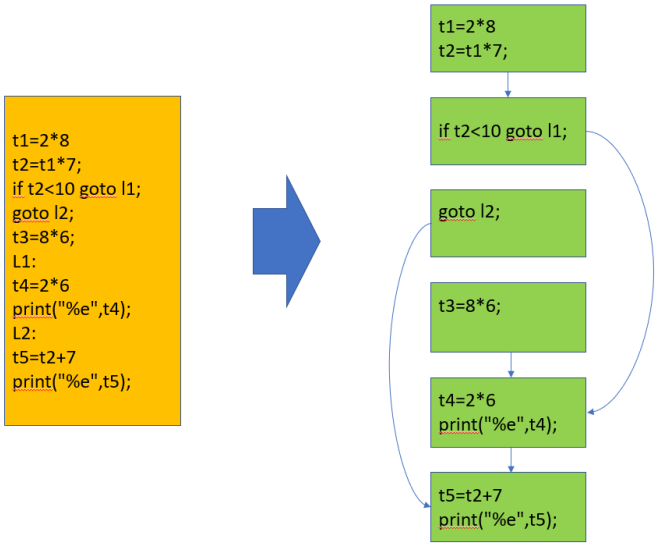
Ejemplo	Optimización
`x = 0 / y;`	`x = 0;`

10.2 Optimización por bloque

Por lo general, los códigos fuente tienen una serie de instrucciones que se ejecutan siempre en orden y están consideradas como los bloques básicos del código. Estos bloques básicos no tienen instrucciones de salto entre ellos, es decir, cuando la primera se ejecuta la instrucción, todas las instrucciones en el mismo bloque básico será ejecutado en su secuencia de aparición sin perder el control de flujo del programa.

10.2.1 Diagrama de bloques básicos

Se solicita desarrollar un diagrama de bloques con todos lo bloques que se formen en el archivo de código de 3 direcciones como se muestra a continuación.



10.2.2 Reglas de optimización

La optimización por bloque permitirá las siguientes reglas.

10.2.2.1 Eliminación de código muerto

Código muerto es aquel código que nunca se utiliza o que la condición que permitiría utilizar este código nunca se cumpliría.

10.2.2.1.1 Regla 19- Código parcialmente muerto

Es aquel código en el que la condición que hace que se ejecute nunca se cumpliría

Ejemplo	Optimización

x = 1 + y;
if 0==1 go to L1
//Código
Goto L2;
L1:
//Código 1
L2:
//Código 2
	x = 1 + y;
	L2:
	//Código 2

10.2.2.1.1 Regla 20- Código muerto

Es aquel código que nunca se ejecuta

Ejemplo	Optimización

x = 1 + y;
if <condicion> go to L1
Goto L2;
z = 1 + w;
L1:
//Código 1
L2:
//Código 2
	if <condicion> go to L1
	Goto L2;
	L1:
	//Código 1
	L2:
	//Código 2

10.2.2.2 Redundancia parcial

Las expresiones redundantes se calculan más de una vez en ruta paralela, sin ningún cambio de operandos.

10.2.2.2.1 Regla 21 - Eliminación de subespresiones comunes

Es aquel código en el que la condición que hace que se ejecute nunca se cumpliría

Ejemplo	Optimización

t1=6/2;
t2=3*t1;
t3=4-2;
t4=t2*t3;
t5=2+5;
t6=t2*t3;
	t1=6/2;
	t2=3*t1;
	t3=4-2;
	t4=t2*t3;
	t5=2+5;
	t6=t4;

t7=t6+t5;	t7=t6+t5;

10.2.2.2.1 Regla 22 – Propagación de la copia

Si a y b tienen el mismo valor, donde se utilice a posteriormente, puede ser reemplazado por b

Ejemplo	Optimización

t1=6/2;
t2=3*t1;
t3=4-2;
t4=t2*t3;
t5=2+5;
t6=t2*t3;
t7=t6+t5;
	t1=6/2;
	t2=3*t1;
	t3=4-2;
	t4=t2*t3;
	t5=2+5;
	t7=t4+t5;

10.2.2.2.1 Regla 23 – variables inutilizadas

Se pueden eliminar todos los temporales no utilizados para no realizar operaciones inservibles.

Ejemplo	Optimización

t1=2+3;
t2=3*t1;
t3=4-2;
t4=t4*t3;
print("%e",t4)
	t3=4-2;
	t4=t4*t3;
	print("%e",t4)

10.2.2.3 Optimización bucle

La mayoría de los programas se ejecutan como un bucle en el sistema. Se hace necesario optimizar los lazos con el fin de ahorrar ciclos de CPU y memoria. Los loops pueden ser optimizados por las siguientes técnicas

10.2.2.3.1 Regla 24 – Código invariante

Un fragmento de código que reside en el bucle y calcula el mismo valor en cada iteración se denomina bucle de código invariante. Este código puede ser trasladado fuera del circuito de ahorro que se calculan sólo una vez, en lugar de en cada iteración.

Ejemplo	Optimización
L1: t1=P+0; t2=Stack[t1]	t1=P+0; t2=Stack[t1] t3=t2-1

`t3=t2-1`
`print("%e",t3);`
`goto L1;`
	`L1:`
	`print("%e",t3);`
	`goto L1;`

11 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
13	[]	Acceso a elemento de arreglo	Izquierda
12	- !	menos unario not	Derecha
11	^^	potencia	Derecha
10	* / %	multiplicativas	Izquierda
9	+ - +	aditivas concatenación de cadenas	Izquierda
8	< <= > >=	relacionales	no asociativo
7	= !>	Igualdad Diferencia	izquierda
6	&	And	izquierda
5	|	Or	Izquierda
4	^	Xor	Izquierda
3	++ --	Incremento Decremento	Izquierda
2	? :	Ternario	Derecha
1	=	asignación	derecha

12 Entregables y Restricciones

12.1 Entregables

Deberán entregarse todos los archivos necesarios para la ejecución de la aplicación, así como el código fuente, las gramáticas. La calificación del proyecto se realizará con los archivos entregados en la fecha establecida. El estudiante es completa y únicamente responsable de verificar el contenido de los entregables. La calificación se realizará sobre archivos ejecutables funcionando en la computadora de su auxiliar correspondiente. **Debe incluir todos los archivos necesarios para la ejecución**. Se proporcionarán archivos de entrada al momento de calificación.

- Código Fuente
- Aplicación funcional
- Gramáticas

12.2 Restricciones

- El proyecto deberá realizarse como una aplicación Web utilizando Javascript.
- Es permitido el uso de cualquier framework
- La herramienta para generar todos los analizadores del proyecto será JISON. La documentación se encuentra en el siguiente enlace http://zaa.ch/jison/docs/
- Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ciencias y Sistemas los involucrados.

12.3 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.
- Si se encuentra que el estudiante únicamente realiza un **intérprete** del lenguaje **J#**, tendrá automáticamente **nota de 0**.
- Se validará que el código de 3 direcciones sea manejado de forma correcta. Cualquier anomalía detectada tendrá automáticamente nota de 0 puntos.
- Se verificará el funcionamiento del código generado ejecutándolo en un intérprete proporcionado por los auxiliares del curso. Si se detecta que el estudiante implementó entornos anidados para la ejecución y generación del código intermedio, automáticamente tendrá nota de 0 puntos.
- Los archivos serán probados por los auxiliares del curso y no contendrán ningún error. Se prohíbe cualquier modificación en los archivos de entrada.

12.4 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]P2_<carnet>.zip.
5. Entrega del proyecto:

Viernes 15 de mayo a las 20:00 horas