



# INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

**Materia:** Lenguajes y Autómatas II

**Maestro:** ISC. Ricardo González González

**Alumnos:**

Isacc Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536.

## ► ACTIVIDAD 7 ◀

**Fecha de entrega:** 29 de Octubre  
de 2024.

EQUIPO NO. 3



## DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 21 / octubre / 2024

### LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ

**SEMESTRE AGOSTO-DICIEMBRE 2024**

**ACTIVIDAD 7** (VALOR 44 PUNTOS)

LEA CUIDADOSAMENTE, Y REALICE LAS SIGUIENTE ACTIVIDADES, CONSIDERANDO LOS CRITERIOS DE CALIDAD PROPUESTOS EN LOS DOCUMENTOS DE LA [GUÍA TUTORIAL](#), Y LA [RÚBRICA DE EVALUACIÓN](#),

EL LECTOR DEBE TOMAR MUY EN CUENTA QUE ESTA ACTIVIDAD ES UN EXAMEN, Y NO UNA SIMPLE TAREA, PUES DEMANDA DEDICACIÓN PARA INVESTIGAR, LEER, ANALIZAR, REDACTAR, ILUSTRAR Y PROPOSER DE MANERA PROFESIONAL LOS TEMAS PROPUESTOS EN LA ESTRUCTURA TEMÁTICA DE ESTA ASIGNATURA.

#### 5. GENERACIÓN DE CÓDIGO INTERMEDIO.

INVESTIGUE, LEA, COMPREnda Y ELABORE UNA **MONOGRAFÍA TÉCNICA** COMPLETAMENTE APEGADA A LO SOLICITADO EN LA [GUÍA TUTORIAL](#) (PUNTO 3, INCISO a ) ACERCA DE LOS SIGUIENTES TEMAS :

- TEMA 5.1 NOTACIÓN PREFIJA, INFIXA, POSTFIJA.
- TEMA 5.2 REPRESENTACIONES DE CÓDIGO INTERMEDIO.  
NOTACIÓN POLACA, CÓDIGO P, TRIPLOS, CUÁDRUPLOS.
- TEMA 5.3 LOS ESQUEMAS DE GENERACIÓN APLICADOS A:  
VARIABLES Y CONSTANTES, EXPRESIONES, INSTRUCCIÓN DE ASIGNACIÓN,  
INSTRUCCIONES DE CONTROL, FUNCIONES Y ESTRUCTURAS

#### CONSIDERACIÓN :

DEBE USTED ENTENDER EL VALOR QUE TIENE ESTA ACTIVIDAD Y QUE LOS TEMAS ANTES REFERIDOS, PARA NADA DEBEN SER ABORDADOS COMO SIMPLES CONCEPTOS REDACTADOS CON LA LIGEREZA, PUES ESTA ACTIVIDAD ESTÁ CONSIDERADA COMO UN EXAMEN.

ANALICE CADA TEMA, SUS CARACTERÍSTICAS, SU IMPORTANCIA, SUS CONCEPTOS, SUS EJEMPLOS, SUS ILUSTRACIONES, Y LOS TIPOS DE EVIDENCIAS QUE USARÁ PARA DEMOSTRAR QUE USTED HA ADQUIRIDO UN VERDADERO CONOCIMIENTO ACERCA DE ÉSTOS.

A MODO DE PRÁCTICAS REALICE ESTE PUNTO Y ELABORE EJERCICIOS NECESARIOS CON LOS CUÁLES USTED DEMUESTRE





- ELABORE UN VIDEO DE AL MENOS 30 MINUTOS EN LOS QUE EXPONGA SUS CONOCIMIENTOS ADQUIRIDOS. DESPUÉS COLOQUE SU MATERIAL EN DRIVE O YOUTUBE E INCLUYA LAS LIGAS EN SU EXAMEN.

**MUY IMPORTANTE:** SI ESTA ACTIVIDAD ES ENTREGADA EN EQUIPO, CADA UNO DE LOS INTEGRANTES DE ÉSTE DEBEN PARTICIPAR EN CADA VIDEO, EXPONIENDO JUNTO A SUS COMPAÑEROS CADA TEMA SOLICITADO.

POR FAVOR NO USE APUNTADORES O MATERIALES DE APOYO TAN SOLO LEER LOS CONCEPTOS. LA IMPORTANCIA Y EL VALOR DE LOS VIDEOS RADICA EN EXPRESAR Y EVALUAR CORRECTAMENTE SU CONOCIMIENTO EN ESTOS TEMAS.

**IMPORTANTE:** SI LO REQUIERE PUEDE CONSULTAR EL [SIGUIENTE DOCUMENTO](#) PARA ORIENTAR SU TRABAJO EN CONOCER QUÉ ES Y CÓMO HACER UNA MONOGRAFÍA CON EL RIGOR ACADÉMICO REQUERIDO.

POR ÚLTIMO, RECUERDE LEER LA GUÍA\_TUTORIAL PARA EL CORRECTO TRATAMIENTO DE ESTE INCISO.

¿ QUÉ SE CALIFICARÁ ?

LA RÚBRICA PARA EVALUAR ESTA ACTIVIDAD ESTARÁ INTEGRADA POR LOS SIGUIENTES CRITERIOS.

- LA OPORTUNIDAD.** SI EL TRABAJO FUE ENTREGADO OPORTUNAMENTE.
- LA COMPRENSIÓN.** SE VALORARÁ EL GRADO DE COMPRENSIÓN DEL TEMAS ANALIZADOS.
- LA CALIDAD.** SI LAS EVIDENCIAS ENVIADAS CORRESPONDEN A LA CALIDAD ESPERADA PARA ESTE NIVEL PROFESIONAL QUE SE CURSA.
- LA CAPACIDAD DE SÍNTESIS.** SI LAS EVIDENCIAS ENTREGADAS TIENEN EL NIVEL DE DETALLE Y PROFUNDIDAD REQUERIDA, O EN BIEN SI SE OMITIERON CONCEPTOS CON EL AFÁN DE SIMPLIFICAR Y ENTREGAR UN MATERIAL ACADÉMICA Y TÉCNICAMENTE POBRE.
- LA CREATIVIDAD.** LA MANERA EN QUE SE EXPRESAN LOS CONCEPTOS Y EL TRATAMIENTO QUE SE DA A LA INFORMACIÓN ANALIZADA PARA QUE ÉSTA SEA COMPRESIBLE EN SU ESENCIA.

**IMPORTANTE :** CUENTA CON EL TIEMPO SUFFICIENTE PARA REALIZAR ESTA ACTIVIDAD Y SUMAR PUNTOS IMPORTANTES A SU CALIFICACIÓN DE ESTA EVALUACIÓN.

**IMPORTANTE :** TODO EL MATERIAL ESCRITO DEBERÁ SER HECHO A MANO.





## **CONSIDERACIONES.**

CADA UNO DE LOS PUNTOS ANTERIORES DEBE SER DESARROLLADO CON LA PROFUNDIDAD ACORDE A UN NIVEL PROFESIONAL, Y APEGÁNDOSE COMPLETAMENTE A LAS DIRECTRICES DE LA GUÍA TUTORIAL.

NO CONCIBA ESTE TRABAJO, COMO UN SIMPLE RESUMEN O EJERCICIO DE TRANSCRIPCIÓN, PUES EL VALOR INDICADO AL INICIO DE ESTA ACTIVIDAD LE DARÁ A USTED UNA BUENA IDEA DE LO QUE SE ESPERA DE ELLA, EN CUANTO A CALIDAD Y EL APRENDIZAJE OBTENIDO, MISMO QUE SERÁ PUESTO A PRUEBA MEDIANTE UN EXAMEN ESCRITO O BIEN ORAL EN CLASE.

SI DECIDIÓ ELABORAR ESTA ACTIVIDAD EN EQUIPO, CADA INTEGRANTE DE ÉSTE DEBERÁ POSEER EL MISMO NIVEL DE CONOCIMIENTO, PUES TAN SOLO REPARTIR TEMAS ENTRE LOS INTEGRANTES DEL EQUIPO, SUPONDRIÁ UN GRAVE ERROR DE INTERPRETACIÓN A LA INTENCIÓN DIDÁCTICA REAL DE ESTA ACTIVIDAD.

POR ÚLTIMO, ESTA ACTIVIDAD SOLO SE PODRÁ DESARROLLAR EN EQUIPO, SI SE REGISTRÓ EN UNO PREVIAMENTE, UTILIZANDO EL FORMATO ENTREGADO EN LA ACTIVIDAD INICIAL. DE LO CONTRARIO DEBERÁ ELABORAR Y ENTREGAR LA ACTIVIDAD DE FORMA INDIVIDUAL.

LA ENTREGA DE DICHO REGISTRO SE HARÁ VÍA CORREO ELECTRÓNICO ENVIANDO ÉSTE AL PROFESOR DESIGNADO, Y POSTERIORMENTE EN CLASE ENTREGANDO LA HOJA EN FÍSICO.

## **OBSERVACIONES:**

- CADA HOJA QUE ENTREGUE DE SU ACTIVIDAD, DEBERÁ ESTAR FIRMADA AL MARGEN DERECHO, INCLUIDA LA PROPIA SOLICITUD DE LA ACTIVIDAD.
- INTEGRE TODO SU TRABAJO EN UN SOLO ARCHIVO DE TIPO .PDF, Y ASIGNE EL NOMBRE QUE A CONTINUACIÓN SE INDICA.

NO OLVIDE ANEXAR LAS HOJAS DE ESTA ACTIVIDAD Y DE SU TRABAJO DESPUÉS DE SU PORTADA.

- UNA VEZ ELABORADA SU ACTIVIDAD, RECUERDE DIGITALIZARLA Y NOMBRARLA EN BASE A LA NOMENCLATURA QUE SE INDICA MÁS ADELANTE EN ESTE DOCUMENTO.
- SI SUS EVIDENCIAS ENVIADAS POR CORREO, NO CUMPLEN CON LA NOMENCLATURA SOLICITADA, NO SERÁN CONSIDERADAS COMO EVIDENCIAS PARA SU EVALUACIÓN.
- POR ÚLTIMO, POR FAVOR GESTIONE APROPIADAMENTE SU TIEMPO, Y SEA PUNTUAL EN SU ENTREGA Y ASÍ EVITAR PROBLEMAS DE NULIDAD POR EXTEMPORANEIDAD.





**LA NOMENCLATURA SOLICITADA PARA ENVIAR SU TRABAJO ES LA SIGUIENTE :**

AAAA-MM-DD\_TNM\_CELAYA\_MATERIA\_DOCUMENTO\_[EQUIPO]\_NOCTROL\_APELLIDOS\_NOMBRE\_SEM.PDF

**( NOTA : \*\*\* TODO DEBE SER ESCRITO USANDO LETRAS MAYÚSCULAS \*\*\* )**

**DONDE :**

TNM_CELAYA	:	INSTITUCIÓN ACADÉMICA
AAAA	:	AÑO
MM	:	MES
DD	:	DÍA
MATERIA	:	LAI <sub>II</sub> MÁS EL GRUPO ( -A , -B, -C )
DOCUMENTO	:	A1-ACTIVIDAD 1, P1-PRACTICA 1, R1-REPORTE 1, T1-TAREA 1, PG1-PROGRAMA, ETC. (CAMBIANDO EL NÚMERO CONSECUТИVO POR EL QUE CORRESPONDA)
[EQUIPO]	:	NÚMERO DEL EQUIPO QUE CORRESPONDA SEGÚN INDICACIÓN DEL PROFESOR. [OPCIONAL]
NOCTROL	:	SU NÚMERO DE CONTROL
APELLIDOS	:	SUS APELLIDOS
NOMBRE	:	SU NOMBRE
SEM	:	EL PERIODO SEMESTRAL EN CURSO: AGO-DIC

**EJEMPLO :**

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2024-10-21\_TNM\_CELAYA\_LAI<sub>II</sub>-A\_A7\_EQUIPO\_99\_9999999\_PEREZ\_PEREZ\_JUAN\_AGO-DIC24.PDF

DONDE EL NOMBRE DEBERÁ CORRESPONDER AL JEFE DE EQUIPO QUE HACE LA ENTREGA DEL TRABAJO.

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2024-10-21\_TNM\_CELAYA\_LAI<sub>II</sub>-A\_A7\_9999999\_PEREZ\_PEREZ\_JUAN\_AGO-DIC24.PDF





**FECHA Y HORA DE ENTREGA:**

LA INDICADA EN LA PLATAFORMA VIRTUAL.

EN CASO DE QUE EL TRABAJO SE HAYA SOLICITADO EN EQUIPO, EL JEFE DEL MISMO SERÁ EL ÚNICO RESPONSABLE DE ENVIAR LA ACTIVIDAD EN LA PLATAFORMA VIRTUAL.

**MUY IMPORTANTE:**

1. DESPUÉS DE LA HORA INDICADA EN LA PLATAFORMA VIRTUAL ( AÚN CUANDO SOLO SEA UN MINUTO O VARIOS ), LA ACTIVIDAD SERÁ CONSIDERADA COMO EXTEMPORÁNEA Y NO CONTARÁ COMO EVIDENCIA PARA SU EVALUACIÓN.

SE LE SUGIERE ENVIAR CON ANTICIPACIÓN SU ACTIVIDAD A FIN DE EVITAR CONFLICTOS POR NO ENTREGAR ÉSTA A TIEMPO.

BAJO NINGÚN PRETEXTO O JUSTIFICACIÓN SE ACEPTARÁN LOS TRABAJOS EXTEMPORÁNEOS, EVITE LA PENA DE RECORDAR A USTED QUE EL VALOR DE LA PUNTUALIDAD ES PARTE IMPORTANTE DE SUS EVIDENCIAS Y ES EL PRIMER PUNTO QUE SE HA DE EVALUAR.

2. NO OLVIDE ANEXAR A SU ARCHIVO .PDF DE EVIDENCIAS UNA PORTADA PROFESIONAL, Y ESTA SOLICITUD DE ACTIVIDADES CON TODAS LAS HOJAS FIRMADAS EN EL MARGEN DERECHO.
3. POR ÚLTIMO, TODA EVIDENCIA GENERADA QUE CONTENGA AL MENOS UNA TRANSCRIPCIÓN DE CUALQUIER FUENTE Y DE CUALQUIER TIPO, ES DECIR CON MATERIAL PLAGIADO SERÁ ANULADA DE FORMA INCONTROVERTIBLE.





# INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

**Materia:** Lenguajes y Autómatas II

**Maestro:** ISC. Ricardo González González

**Alumnos:**

Isacc Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536

## MONOGRAFÍA GENERAL SOBRE

Generación de código Intermedio

**Fecha de entrega:** 29 de Octubre de  
2024.

**EQUIPO NO.3**

# Índice de Temas

1.1.1 Notación prefija, infija, postfija.  
1.1.2 Historia de la notación matemática.  
1.1.3 Conceptos clave en notación.  
1.1.4 Aplicación de la notación en ciencias de la computación.  
1.1.5 Notación infija.  
1.1.6 Notación polaca.  
1.1.7 Notación postfija.  
1.1.8 Tabla comparativa.  
1.1.9 Conversión entre notaciones.  
1.1.10 Implementación en compiladores e intérpretes.  
1.1.11 Conclusión.

## 5.2 Representaciones de código intermedio.

- 5.2.1 Objetivo del código intermedio en el proceso de compilación.
- 5.2.2 Notación polaca: prefija y postfija.
  - 5.2.2.1 Fundamentos matemáticos y lógicos.
  - 5.2.2.2 Algoritmos de evaluación de notación postfija.
- 5.2.3 Código P(P-CODE): estructura y ejecución en máquinas virtuales.
  - 5.2.3.1 Concepto y origen del código P.
  - 5.2.3.2 Arquitectura de la P-MACHINE.
  - 5.2.3.3 Ejemplo de código P.
  - 5.2.3 Impacto del P-CODE en la portabilidad,
- 5.2.4 Triplos: Representación compacta de operaciones.
  - 5.2.4.1 Estructura de los triplos.
  - 5.2.4.2 Ejemplos de triplos.
  - 5.2.4.3 Ventajas de los triplos.
  - 5.2.4.4 Desventajas de los triplos.

5.2.5 Cuádruplos: Representación estructurada para optimización.

5.2.5.1 Estructura de los cuádruplos.

5.2.5.2 Ejemplo de cuádruplos.

5.2.5.3 Uso en la generación de código.

5.2.6 Comparativa entre triplos y cuádruplos.

5.2.7 Conclusión.

5.3 Los esquemas de generación aplicados a: variables y constantes, expresiones, instrucción de asignación, instrucciones de control, funcionales y estructuras.

5.3.1 Esquemas de generación.

5.3.1.1 Variables y constantes.

5.3.1.2 Expresiones.

5.3.1.3 Instrucción de asignación.

5.3.1.4 Instrucciones de control.

5.3.1.5 Funciones.

5.3.1.6 Estructuras.

5.3.2 Conclusion.

Karen Montes  
K Montes

# Notación prefija, infija, postfija

## Introducción

La notación de expresiones matemáticas ha sido un tema central en la informática y las matemáticas computacionales, debido a su importancia en el análisis, la optimización y la ejecución de programas. En el contexto de compiladores e intérpretes, el uso de diferentes notaciones -infija, prefija y postfija- permiten procesar expresiones de manera eficiente, facilitando tanto legibilidad para los humanos como el procesamiento para las máquinas.

Las expresiones aritméticas y lógicas son la base de numerosos algoritmos y aplicaciones en programación. Los compiladores, encargados de traducir el código de alto nivel a un lenguaje comprensible por las máquinas, deben interpretar y evaluar estas expresiones de manera precisa y eficiente. Para ellos, las notaciones prefija (también conocida como polaca), infija y postfija (o polaca inversa) juegan un papel fundamental, cada una con sus ventajas y desventajas.

## Fundamentos de la notación Matemática

Las notaciones matemáticas son representaciones formales que ayudan a expresar relaciones, operaciones y funciones de forma clara y precisa. En el contexto de la informática y la teoría de lenguajes formales, estas notaciones permiten representar expresiones de manera que puedan ser interpretadas y evaluadas por computadoras de forma

eficiente y sin ambigüedades. Para comprender la notación infijas prefijo y postfija es importante revisar algunos conceptos clave en la estructura de las expresiones matemáticas.

## Historia de la Notación Matemática.

La notación matemática ha evolucionado desde tiempos antiguos, buscando siempre mejorar la claridad en la comunicación de ideas y simplificar el cálculo. Originalmente, las expresiones se escribían en forma verbal, hasta que se introdujeron símbolos para operaciones básicas como la suma y la multiplicación. Con el tiempo, surgieron convenciones para la disposición de operadores y operandos que fueron refinándose hasta la estructura que conocemos hoy. Estas convenciones son fundamentales, ya que permiten que una misma expresión sea comprendida sin ambigüedades en contextos de álgebras, cálculo y posteriormente en la programación.

## Conceptos Clave en notación

En la evaluación de expresiones matemáticas en cualquier notación, los siguientes conceptos son esenciales:

### • Operadores y Operandos

- Un operador es un símbolo que representa una operación, como  $+$ ,  $-$ ,  $\times$ ,  $/$ .
- Un operando es el valor o la variable sobre el cual actúa el operador.

$$5 + 3$$

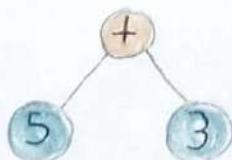
↑      ↑      ↗  
operando operador operando

### • Procedencia y Asociatividad:

- La precedencia de operadores es una regla que define el orden en que se ejecutan las operaciones en una expresión
- La asociatividad de operadores define el orden de ejecución en operaciones con la misma precedencia.

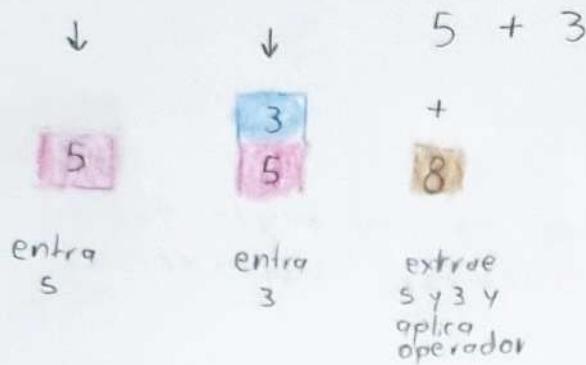
### Aplicación de la notación en Ciencias de la computación

Las reglas de precedencia y asociatividad se adaptan en ciencias de la computación para ser interpretadas y procesadas de manera formal en algoritmos y estructuras de datos. Los compiladores utilizan estas reglas para crear una estructura interna llamada árbol de expresión, donde cada nodo es un operador y sus hijos son operandos. Este árbol permite la evaluación ordenada de operaciones y es especialmente útil cuando se trabaja con expresiones complejas en lenguajes de programación.



Por otra parte, las estructuras de datos como pilas facilitan la evaluación de expresiones posfijas. En esta notación, las expresiones se pueden evaluar de manera secuencial, utilizando una pila para almacenar temporalmente

los operandos. Esto hace que la notación postfix sea preferida en algunos lenguajes y sistemas que se basan en máquinas de pilas, ya que evita la complejidad de los paréntesis y reduce la ambigüedad en la evaluación.



## Notación infija

La notación infija es la forma de representación de expresiones matemáticas que se utiliza más comúnmente en la vida cotidiana y es la más familiar para los humanos. En la notación infija, el operador se coloca entre los operandos, como en la expresión  $A + B$ . Este formato es intuitivo y fácil de leer, especialmente para quienes están acostumbrados a la notación algebraica convencional, donde se respetan las reglas de precedencia y asociatividad de los operadores.

## Definición y estructura

En la notación infija, los operadores (suma, resta, etc.) se sitúan entre los operandos. Esta disposición facilita la comprensión humana, pero presenta ciertas limitaciones para su procesamiento por máquinas, ya que los computadores deben analizar los operadores y su orden para determinar la secuencia correcta de evaluación. Por ejemplo la expresión  $A + B * C$ , la multiplicación se realiza antes que la suma debido a su precedencia superior.

# Ventajas y desventajas de la notación Infija

## • Ventajas:

- Legibilidad y familiaridad: Es la notación que más se asemeja al álgebra clásica, por lo que resulta natural para los personas.
- Claridad en operaciones Simples: En expresiones cortas o básicas, la notación infija es directa y clara

## • Desventajas

- Ambigüedad sin Paréntesis: En expresiones complejas, es necesario el uso de paréntesis para evitar las ambigüedades en el orden de las operaciones.
- Complejidad en Evaluación Computacional: Debido a la necesidad de respetar la procedencia y asociatividad de operadores, los compiladores deben aplicar algoritmos adicionales para interpretar correctamente las expresiones.

## Evaluación de Expresiones en Notación Infija

Para evaluar una expresión en notación infija, los compiladores suelen utilizar un proceso llamado análisis sintáctico, que interpreta y organiza los operadores y operandos según sus reglas de precedencia. Este análisis implica la creación de una estructura jerárquica, comúnmente en forma de árbol de expresión.

- 1.- Análisis de precedencia: El compilador identifica los operadores y les asigna una prioridad según su procedencia.
- 2.- Organización Jerárquica: A partir de estas reglas, construye un árbol en el que los operadores con mayor precedencia

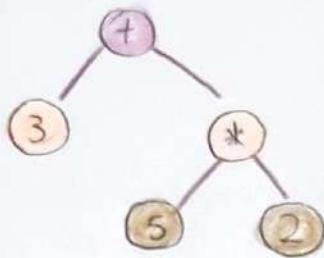
son evaluados primero

3. Evaluación del árbol: El compilador recorre el árbol de expresión en el orden adecuado (a menudo en un recorrido postorden) para evaluar el resultado

Ejemplo

$$3 + 5 * 2$$

- se detecta que la multiplicación tiene mayor precedencia que la suma, por lo que se evalúa primero.



## Notación Polaca (prefija)

La notación prefija, también conocida como notación polaca, coloca el operador antes de los operandos, esta notación, una expresión como  $A + B$  se escribe como  $+ A B$ . Esta estructura evita ambigüedades en el orden de las operaciones y permite prescindir de los paréntesis, lo que facilita la evaluación directa de las expresiones.

## Definición y Estructura

En la notación prefija, el operador se coloca primero, seguido por sus operandos. Esto significa que la estructura de las expresiones mismas determinan el orden en que las operaciones deben ser evaluadas, eliminando la necesidad de parentesis o de preocuparse por la precedencia de operadores. Por ejemplo:

$$(3 + 5) * 2$$

$$* + 3 5 2$$

## Características de la Notación Prefija.

- Orden Natural de la evaluación: La notación prefija está diseñada para evaluarse de izquierdo a derecho, comenzando desde el operador más externo. Esto permite interpretar expresiones de manera directa y sistemática, lo que resulta útil en ciertos contextos de programación.
- Sin paréntesis: A diferencia de la notación infix, no se requieren parentesis para agrupar operaciones, ya que el orden de evaluaciones está dado de forma explícita por la posición del operador respecto a los operandos.

Esta notación fue propuesta originalmente por el matemático polaco Jan Lukasiewicz, de ahí su denominación. Su objetivo era evitar ambigüedades en la notación matemática y proporcionar una representación que pudiera ser evaluada de forma sistemática.

### Ejemplo

Para ilustrar cómo se evalúa una expresión en notación prefija, consideraremos la expresión infix  $(3+4) \times 5$  que se convierte en  $* + 3 4 5$

Para evaluarla:

- 1= Se lee primero el operador  $*$ , lo que indica una multiplicación
- 2= A continuación, se encuentra el operador  $+$ , que se evalúa primero con sus operandos  $3$  y  $4$  dando como resultado  $7$ .
- 3= Finalmente se multiplica el resultado de  $3+4$  (que es  $7$ ) por el operando  $5$ , dando como resultado final de  $35$ .

El orden de evaluación, de izquierdo a derecho, permite que las expresiones se evalúen sin necesidad de convertirlas a otra notación. Esto es especialmente útil en lenguajes de programación funcionales y en estructuras de árboles de expresión utilizados por compiladores.

## Notación Postfija (Polaca Inversa)

La notación postfija, también llamada Polaca inversa, coloca el operador después de los operandos. Esta notación, una expresión como  $A+B$  se representa como  $AB+$ . Esta forma de notación es especialmente útil para la evaluación de expresiones mediante una estructura de datos tipo pilas, ya que elimina la necesidad de paréntesis y simplifica el procesamiento de las expresiones aritméticas.

### Definición y Estructura.

En la notación postfija, los operandos aparecen primero y después los operadores. Por ejemplo la expresión infija  $(3+4) \times 5$  se representa en notación postfija  $3\ 4\ +\ 5\ \times$ . Al colocar el operador al final, esta notación garantiza que cada operación tenga definidos sus operandos de antemano, lo que facilita su evaluación sin requerir reglas de precedencia ni asociatividad.

### Ventajas de la Notación Postfija en Computación.

La notación postfija presenta varias ventajas en el contexto de la programación y la computación:

- **Eliminación de Paréntesis y Ambigüedad:** Al colocar el operador después de sus operandos se evita la necesidad de paréntesis, ya que el orden de evaluación queda claro y no depende de reglas de precedencia.

• Facilidad de evaluación con pilas: La notación Postfija se evalúa eficientemente mediante una estructuras de pila. A medida que se leen los operandos y operadores, se almacenan temporalmente en la pila y se procesan en el orden correcto, lo cual simplifica la implementación en un algoritmo.

Ejemplo

$$(3+4)*5$$

$$3 \ 4 \ + \ 5 \ *$$

Tabla comparativa

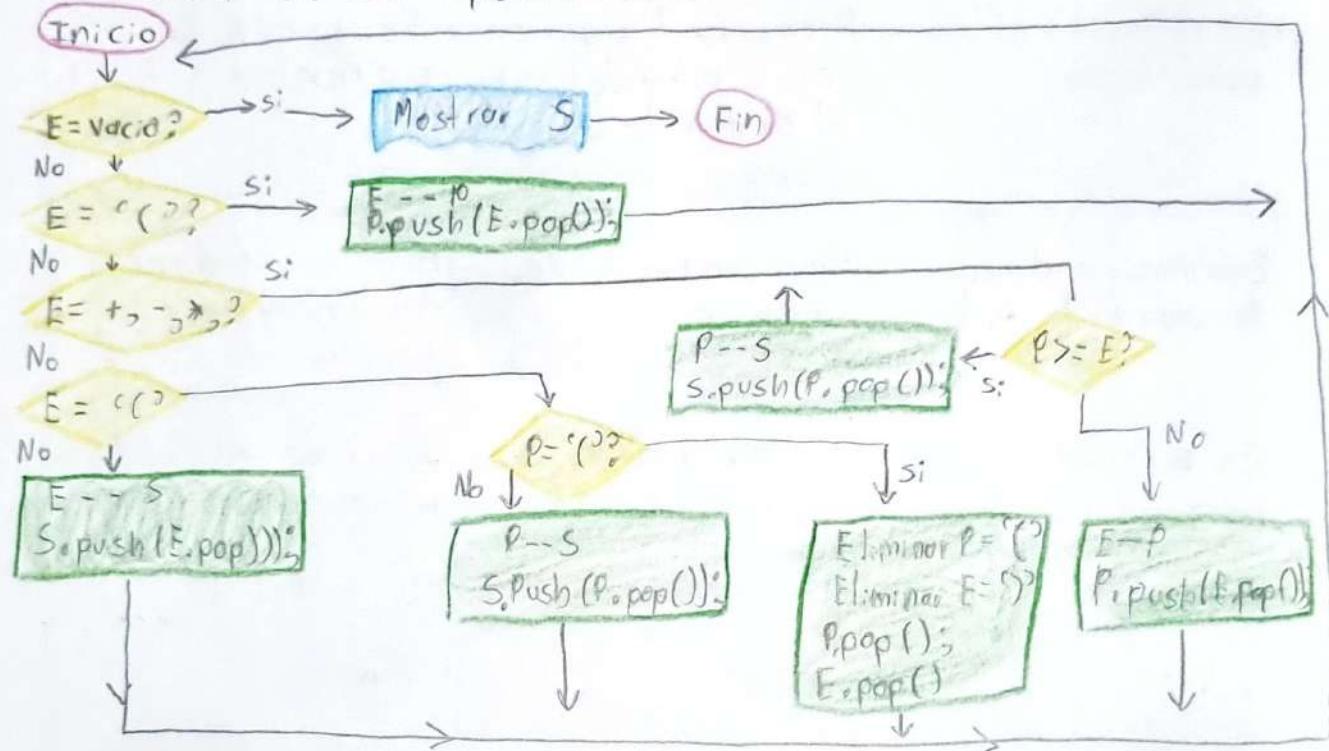
Notación prefija	Notación Infixa	Notación Postfija
El operador está antes de los operandos	El operador está entre los operandos	El operador está después de los operandos.
No necesita el paréntesis	Necesita los paréntesis para definir el orden de operaciones	No necesita parentesis
Evaluación directa de izquierda a derecha	Requiere reglas de precedencia y Parentesis	Evaluación directa de izquierda a derecha
Es más adecuada para computadoras	Más intuitiva para los humanos	Es más adecuada para computadoras
$* + S \ 3 \ 2$	$(S+3)*2$	$S \ 3 \ + \ 2 \ *$
$+ A \ B$	$A + B$	$A \ B \ +$
$- * + A \ B \ C \ D$	$(A+B)*C-D$	$A \ B \ + \ C \ * \ D \ -$

## Conversión entre Notaciones

La conversión entre notaciones infija, prefija y postfixa es un proceso fundamental en la teoría de lenguajes de programación y es ampliamente utilizado en compiladores e interpretes. Estos sistemas necesitan interpretar las expresiones de código de manera que puedan ser evaluadas de forma ordenada y sin ambigüedades. Existen algoritmos específicos para realizar estas conversiones, como el algoritmo de Shunting Yard de Edsger Dijkstra, que es ampliamente utilizado para convertir expresiones infijas en notación postfixa o prefija.

### Conversión de infija a Postfija

Este algoritmo utiliza una pila para manejar la precedencia de operadores y gestionar el orden correcto de evaluación de las operaciones.



E = Pila de entrada    P = Pila temporal    S = pila de salida

## Conversion de Infija a Prefija

Para convertir una expresión infija a una notación prefija, se sigue un procedimiento similar al anterior con algunos ajustes. La notación prefija requiere que los operadores precedan a sus operandos, por lo que se invierte el orden de evaluación.

- 1.- Invertir la expresión Original: la expresión infija se lee de derecho a izquierdo, invirtiendo los parentesis de apertura y cierre.
- 2.- Aplicar Algoritmo similar al shunting Yard: los operadores se gestionan en una pila según sus reglas de precedencia y asociatividad, pero en este caso se añade al resultado antes de los operandos
- 3.- Invertir el resultado final: Una vez terminada la conversión, el resultado se invierte para obtener la notación prefija correcta

## Implementación en compiladores e interpretos

El desarrollo de compiladores e interpretos, la evolución y manipulación de expresiones matemáticas y lógicas es un componente esencial. Para ellos es crucial manejar correctamente las notaciones infija, prefija y postfixa ya que permiten procesar expresiones de manera ordenada y eficiente, optimizando la generación de código y la ejecución del programa.

## Análisis Sintáctico y evaluación de expresiones

Uno de los principales procesos en los compiladores es el análisis sintáctico, donde se descompone el código fuente en una estructura comprensible para la máquina.

Este procesos, los compiladores utilizan notaciones prefija y postfija para transformar y simplificar las expresiones de modo que sean evaluables de forma eficiente. Esto permite que las expresiones se convierten en una representación intermedia que facilita la generación de código optimizado.

• Árboles de expresión: Para manejar las expresiones infixos, los compiladores construyen una estructura llamada árbol de expresión. En este árbol:

- Cada nodo representa un operador.
- Los hijos del nodo representan los operandos los cuales actua el operador.

Estos árboles facilitan la organización y el orden de las operaciones según las reglas de precedencia y asociatividad. Además, el árbol de expresión permite que el compilador elija entre varios ordenes de evaluación, optimizando el rendimiento según sea necesario.

### Ventajas de las Notaciones en la optimización del código

Cada notación tiene ventajas especiales que pueden aprovecharse para optimizar el código:

- Notación Postfija: Al eliminar la necesidad de paréntesis, permite una evaluación más rápida en tiempo de ejecución. La mayoría de los lenguajes máquina utilizan esta notación para reducir la complejidad del análisis Sintáctico.
- Notación Prefija: Útil en lenguajes de programación funcionales y en entornos donde la ejecución secuencial es clave. Este permite una evaluación inmediata de

las operaciones, lo que es beneficioso en el análisis de expresiones y en optimizaciones específicas de los compiladores.

- Notación Infija: Aunque es la más compleja para la evaluación en tiempo de ejecución, es ideal para la legibilidad y se utiliza en la mayoría de los lenguajes de alto nivel. Los compiladores transforman esta notación en prefijo o postfix durante el análisis y la generación de código para facilitar su evaluación.

## Conclusion

La compresión y uso de las notaciones infija, prefijo y postfix no solo enriquecen el campo de la teoría de lenguajes formales, si no que también contribuyen al desarrollo de herramientas eficientes para la programación. Estos conceptos, aunque en apariencia simples, son esenciales para resolver problemas complejos en la evaluación de expresiones, en la optimización de programas y en la ejecución de diversos sistemas. Su relevancia sigue siendo significativa en la informática Moderna, demostrando que los principios matemáticos básicos pueden llevar a soluciones prácticas de alto impacto en el desarrollo de tecnología avanzada.

## TEMA 5.2 REPRESENTACIONES DE CÓDIGO INTERMEDIO

Notación Polaca, Código P, Triplos Y Cuádruos

El código intermedio es una representación abstracta entre el código fuente (escrito en un lenguaje de alto nivel, como C o Python) y el código máquina o binario específico de la arquitectura destino. Esta etapa intermedia no se ejecuta directamente en el hardware, sino que es utilizada para:

- Optimizar el código fuente antes de traducirlo a ensamblador o binario.
- Aislar las dependencias específicas de la arquitectura, facilitando la portabilidad del compilador.
- Simplificar la generación del código final mediante un lenguaje más cercano a las operaciones básicas del procesador.

El uso de código intermedio permite que el compilador se estructure en fases independientes, logrando un diseño más modular y eficiente. De esta forma, un compilador puede generar el mismo código intermedio para múltiples plataformas y luego adoptar las últimas fases.

# Objetivo Del Código Intermedio En El Proceso De Compilación

La transformación del código fuente a código intermedio tiene varios propósitos clave:

## 1- Optimización del programa;

Las optimizaciones pueden ser realizadas en esta representación sin necesidad de preocuparse por los detalles específicos de la máquina destino. Ejemplos de optimizaciones son:

- Eliminación de subexpresiones comunes
- Propagación de constantes
- Reordenamiento de instrucciones para minimizar accesos a memoria.

## 2. Portabilidad:

Al generar código intermedio independiente del hardware, se puede reutilizar el compilador en múltiples plataformas con solo cambiar la fase de backend (la parte del compilador que genera el código final).

## 3. Modularidad:

El uso del lenguaje intermedio permite dividir el compilador en fases independientes, facilitando el mantenimiento y desarrollo del compilador. Por ejemplo, la fase de análisis sintáctico puede producir código intermedio, que luego es optimizado y transformado por otras fases.

## 1. NOTACIÓN POLACA: PREFIJA Y POSTFILA

### 2.1 Fundamentos Matemáticos Y Lógicos

La notación polaca fue introducida por Jan Lukasiewicz en 1920 como una forma de evitar la ambigüedad en la evaluación de expresiones matemáticas. Al no depender de paréntesis para definir la precedencia, es especialmente útil en el ámbito de compiladores donde la simplificación del análisis sintáctico es crucial.

#### Tipos De Notación Polaca:

- Notación Prefija (Polaca Original):

El operador antecede a los operandos, como en  $* + 43 - 21$ .

- Notación Postfija (Polaca Inversa):

El operador se coloca después de los operandos, como en  $43 + 21 - *$ .

Estas notaciones son esenciales en lenguajes basados en pila como forth o PostScript, donde la evaluación es directa y no requiere análisis sintáctico completo ni complejo.

## 2.2 Algoritmo De Evaluación De Notación Postfija.

Para interpretar expresiones en notación polaca inversa (RPN), se utiliza una pila. El proceso consiste en empujar operandos a la pila y extraerlos para operar cuando aparece un operador.

Ejemplo:

$$5 \ 1 \ 2 \ + \ 4 \ * \ + \ 3 \ -$$

1. Inserta 5 en la pila.
2. Inserta 1 y 2.
3. Aparece +: Retira 1 y 2, inserta 3 en la pila
4. Inserta 4.
5. Aparece \*: Multiplica 3 y 4, inserta 12.
6. Aparece +: Suma 12 y 5, inserta 17.
7. Aparece -: Resta 3 de 17, el resultado es 14.

Aplicaciones en Compiladores:

- Evaluación eficiente: Las expresiones RPN no requieren backtracking durante la ejecución.
- Optimización en tiempo de Compilación: Se puede generar código intermedio a partir de la representación polaca.

## 3. CÓDIGO P (P-CODE): ESTRUCTURA Y EJECUCIÓN EN MÁQUINAS VIRTUALES.

### 3.1 Concepto y Origen Del Código P.

El P-code es un lenguaje de bajo nivel diseñado para ser interpretado por una máquina virtual, como la P-machine del lenguaje Pascal. Los compiladores que generan P-code traducen el código fuente a una representación más abstracta que luego puede ser ejecutada en cualquier sistema que tenga una máquina virtual compatible.

### 3.2 Arquitectura de la P-Machine

La P-Machine es una máquina virtual basada en una pila, donde la mayoría de las operaciones se realizan con los valores en la cima de la pila. El código P contiene instrucciones como:

- LOAD: Carga un valor en la pila
- STORE: Almacena el valor superior de la pila en una variable.
- ADD, SUB, MUL, DIV: Operaciones aritméticas básicas
- JMP, JZ: Saltos condicionales e incondicionales.

### 3.3 Ejemplo de Código P.

Expresión:  $(x + y) * z$

Traducción:

```
LOAD X  
LOAD Y  
ADD  
LOAD Z  
MUL
```

### 3.4 Impacto del P-code en la Portabilidad

El código P es una solución eficaz para garantizar que los programas compilados puedan ser ejecutados en diferentes plataformas mediante la implementación de una máquina virtual específica para cada sistema. Sin embargo, este enfoque puede sacrificar rendimiento en comparación con el código compilado a máquina.

## 4. TRIPLOS: REPRESENTACIÓN COMPACTA DE OPERACIONES

### 4.1 Estructura de los Triplos

Un triplo se representa mediante una tupla (Operador, Operando1, Operando2). En esta técnica, las posiciones dentro de la lista de triplos identifican los resultados intermedios, evitando la necesidad de variables temporales explícitas

## 4.2 EJEMPLOS DE TRIPLOS

Expresión:  $(a+b) * (c-d)$

Lista de triplos:

1.  $(+, a, b) \rightarrow \text{triplo 1}$
2.  $(-, c, d) \rightarrow \text{triplo 2}$
3.  $(*, \#1, \#2) \rightarrow \text{triplo 3.}$

La principal característica de los triplos es que no asignan nombres explícitos a variables temporales generadas durante los cálculos intermedios. En lugar de eso, se hace referencia a otras operaciones mediante el número de las instrucciones que las generó. Esta estructura simplifica la representación interna del compilador, reduciendo la cantidad de símbolos a gestionar.

### Ventajas de los Triplos.

1. Eficiencia en el manejo de variables temporales:  
Al referirse a los resultados intermedios por el número de la instrucción en lugar de usar variables temporales explícitas, se simplifica la estructura interna del compilador.

2. Optimización más sencillas:

Las dependencias entre instrucciones son más fáciles de identificar mediante la referencia directa a los índices, lo que facilita la propagación de constantes o la eliminación de código muerto.

3. Reducción de colisiones en tablas de símbolos;  
Dado que no se crean muchas variables temporales explícitas, se evita saturar la tabla de símbolos como nombres intermedios.

## Desventajas de los Triplos

1. Complejidad en la generación de código final  
Durante la fase de generación del código máquina, los triplos pueden hacer más difícil mapear las referencias a registros o variables, ya que cada instrucción depende de su índice.

2. Problemas en arquitecturas con múltiples bloques básicos:

Cuando una instrucción depende de otra ubicada en un bloque básico diferente (una sección del flujo de control), puede volverse complicado seguir las dependencias entre las instrucciones.

3. Menor claridad en el análisis manual:

Los triplos pueden ser menos intuitivos que otras representaciones, como los cuádruplos, donde cada resultado se almacena explícitamente en una variable temporal.

## 5. Cuádruplos: Representación Estructurada para Optimización

### 5.1 Estructura de los Cuádruplos

Cada cuádruplo consta de 4 campos:

1. Operador: la operación a realizar, como  $+, -, *, /$ .

2. Operando 1: Puede ser un valor directo, una variable o un resultado intermedio.

3. Operando 2: También puede ser una variable, un valor directo o el resultado de una operación previa.

4. Resultado: Donde se almacenará el resultado de la operación, suele ser un registro temporal para reutilizarlo posteriormente.

Con esta estructura se puede trabajar con resultados intermedios, referenciándolos de manera explícita y eficaz. Esta representación es fundamental por ofrecer una forma normalizadora de expresar operaciones, facilitando el posterior proceso de traducción y optimización.

### 5.2 Ejemplo de Cuádruplos

Expresión:  $(a+b) * (c-d)$

Conjunto de cuádruplos

$(+, a, b, T1)$

$(-, c, d, T2)$

$(*, T1, T2, T3)$

### 5.3 Uso en la generación de código

El uso de cuádruplos es esencial en las etapas de optimización del compilador, buscando mejorar la eficiencia del código generado. Con estos se pueden implementar técnicas de optimización como:

1. Eliminación de subexpresiones comunes: detectar y reutilizar cálculos repetitivos, evitando duplicar operaciones.
2. Propagación de constantes: reemplazar variables con sus valores constantes si es posible, reduciendo la necesidad de cálculos en tiempo de ejecución.
3. Reordenamiento de instrucciones: organizar las operaciones para minimizar los accesos a memoria o reducir la cantidad de registros usados, optimizando recursos.

### 6. Comparativa entre Triplos y Cuádruplos

Aspecto	Triplos	Cuádruplos
Estructura	Tuplas de tres elementos	Cuatro elementos eópticos
Memoria	Menor uso	Mayor por variables temporales
Facilidad de Optimización	Baja	Alta

## Conclusion

Las representaciones de código intermedio vistas, son herramientas fundamentales en el proceso de compilación, son diseñadas para facilitar la traducción y optimización del código fuente en lenguaje de alto nivel a un código máquina.

Estas representaciones proporcionan una base estructurada y versátil para las etapas de análisis semántico, generación y optimización de código en los compiladores. Cada uno presenta sus ventajas según el objetivo de optimización, generando así código final eficiente y optimizado.



# Tema 5.3 Los esquemas de generación aplicados a: variables y constantes, expresiones, instrucción de asignación, instrucciones de control, funciones y estructuras.

## Introducción:

En el ámbito de la programación, la generación de código intermedio es una etapa crucial en la compilación ya que transforma el código fuente en una representación más abstracta y optimizada antes de producir el código máquina final. Este proceso implica la creación de estructuras y esquemas que representan las diferentes expresiones, instrucciones de asignación, instrucciones de control, funciones y estructuras.

Comprender cómo se generan y manejan estas estructuras en el código intermedio es esencial para poder optimizar el rendimiento de los programas y garantizar su correcto funcionamiento.

En esta monografía técnica se podrá explorar más a detalle las características, importancia y conceptos fundamentales de los esquemas de generación aplicados a cada una de las construcciones, proporcionando una visión integral de su papel en el proceso de compilación y en la eficiencia del código ejecutable.

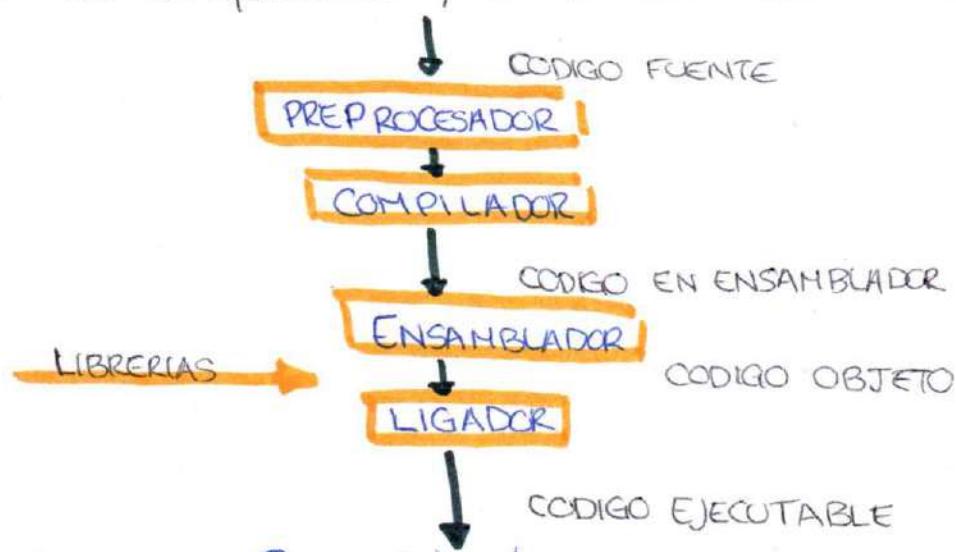


Figura 1. Compilador



# Esquemas de generación:

- ▶ Los esquemas de generación son las estrategias o acciones que se deberán realizar y tomarse en cuenta en el momento de generar código intermedio.
- ▶ Los esquemas de generación dependen de cada lenguaje.

## 1-VARIABLES Y CONSTANTES:

### Definiciones:

- **Variable:** Espacio en memoria que almacena datos y cuyo contenido puede cambiar durante la ejecución del programa.
- **Constante:** Valor numérico o alfanumérico que no cambia durante la ejecución del programa.

### Esquemas de generación:

Al generar código intermedio, es crucial identificar y declarar correctamente las variables y constantes. Las constantes se pueden asignar directamente, mientras que las variables requieren una gestión adecuada de memoria para permitir su modificación durante la ejecución.

### Características:

- ▶ **Variables:** Tienen un tipo de dato (entero, flotante, cadena, etc.) que define qué tipo de información pueden almacenar.
- ▶ **Constante:** Se usan datos que no deben ser modificados, como parámetros o valores inmutables.

### Importancia:

Permiten al programador trabajar con datos de manera flexible y controlada, asegurando que algunos valores clave permanezcan fijos mientras otros pueden ser cambiados según sea necesario durante la ejecución.

- Las declaraciones de variables y constantes deben separarse de tal manera que queden las expresiones una por una de manera simple.



## 1-Expresiones :

### Definición :

Combinación de constantes, variables, operadores y paréntesis que produce un valor.

### Esquemas de generación :

Las expresiones deben evaluarse y simplificarse durante la generación de código intermedio. Es esencial manejar correctamente los operadores y garantizar que las expresiones complejas se descompongan en operaciones más simples para su procesamiento eficiente.

### Características :

Son combinaciones de operadores y operandos, que devuelven un valor. Estas expresiones pueden ser Matemáticas, lógicas o de otro tipo, dependiendo del contexto del lenguaje de programación.

### Importancia :

Permiten realizar cálculos y tomar decisiones basadas en condiciones lógicas, lo cual es esencial para manipular datos y controlar el flujo del programa.

- Para generar expresiones estas deben representarse de manera más simple y más literal para que su conversión sea más rápida.

Por ejemplo:

La traducción de operaciones aritméticas debe especificarse por una, de tal forma que una expresión sea lo más mínimo posible.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas.



## 3-Instrucción de asignación:

### Definición:

Consiste en asignar el resultado de la evaluación de una expresión a una variable.

### Esquemas de generación:

Las instrucciones de asignación deben traducirse de manera que el lado derecho de la asignación se evalúe y el resultado se almacene en la variable correspondiente los tipos de datos y las posibles conversiones entre ellos.

### Características:

Asigna valor a una variable. Generalmente, se utiliza el operador  $=$  para hacer la asignación, por ejemplo  $x = 5$

### Importancia:

Permite almacenar y actualizar valores en variables, lo cual es clave para mantener el estado del programa y realizar cálculos secuenciales.

- Las operaciones de asignación deben quedar expresadas por una expresión sencilla, si está es compleja se debe reducir hasta quedar un operador sencillo.

► Por ejemplo:

$$x = a + b / 5;$$

Debe quedar de la forma.

$$\begin{aligned}y &= b / 5; \\z &= a + y; \\x &= z;\end{aligned}$$

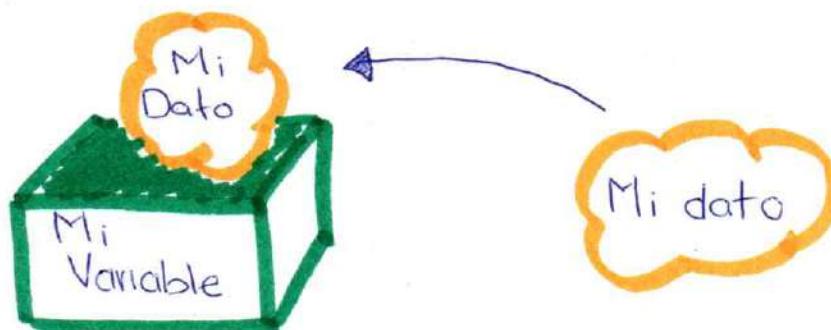


Figura 2. Instrucción de asignación



## 4. Instrucciones de control:

### Definición:

Comandos que alteran el flujo de ejecución del programa basándose en condiciones o iteraciones.

### Esquemas de generación:

Estas instrucciones, como **if**, **while** y **for**, requieren una atención especial para garantizar que las condiciones se evalúen correctamente y que el flujo de control se maneje adecuadamente en el código intermedio.

### Características:

Incluyen estructuras como: **if**, **else**, **switch**, **for**, **while**, entre otras. Controlan el flujo de ejecución del programa basándose en condiciones o repeticiones.

Se clasifican en:

- **Condicionales:** Ejecutan código solo si se cumple una condición (**if**, **else**, **switch**)
- **Repetitivas:** Ejecutan un bloque de código varias veces (**for**, **while**, **do-while**).

### Importancia:

Son esenciales para tomar decisiones y ejecutar acciones repetitivas de manera automática, permitiendo que los programas sean más dinámicos y eficientes.

- El término "estructuras de control", viene del campo de la ciencia computacional. Cuando se presentan implementaciones de Java para las estructuras de control, nos referimos a ellas con la terminología de la especificación del lenguaje de Java, que se refiera a ella como instrucciones.
- Con las estructuras de control se puede:
  - De acuerdo a una condición, ejecutar un grupo u otro de sentencias (**If - Then - Else** y **Select - Case**)
  - Ejecutar un grupo de sentencias **mientras** exista una condición (**Do - while**)
  - Ejecutar un grupo de sentencias **hasta** que exista una condición (**Do - Until**)
  - Ejecutar un grupo de sentencias un número determinado de veces (**For - Next**).



## 5. Funciones:

### Definición:

Conjunto de instrucciones que realizan una tarea específica y pueden ser llamadas desde otras partes del programa.

### Esquemas de generación:

Al generar código para funciones, es esencial manejar la declaración, los parámetros de entrada y el valor de retorno. Además se debe gestionar el ámbito de las variables y garantizar que las funciones se llamen correctamente desde otras partes del código.

### Características:

Son bloques de código reutilizables que realizan una tarea específica. Estos pueden recibir parámetros y devolver valores. En muchos lenguajes se definen con la palabra clave función o def.

### Importancia:

Fomentan la modularidad, facilitando la organización de código y su reutilización en diferentes partes del programa. Esto hace que el programa/código sea más legible, mantenible y eficiente.

- Se diferencian de los procedimientos ya que estos no devuelven un resultado.
- En general las funciones deben tener un nombre único en el ámbito para poder ser llamadas, un tipo de dato de resultado, una lista de parámetros de entrada y su código.
- Estas pueden tener un objetivo particular y este es ejecutado al ser llamado desde otra función o procedimiento.

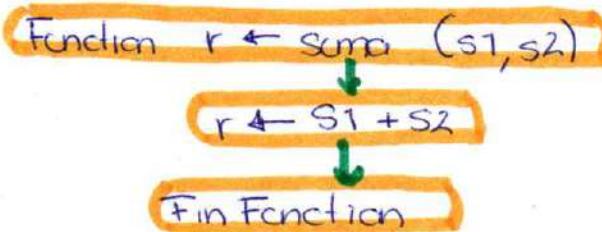


Figura 3. Ejemplo Función



## 6-Estructuras: Definición:

Son formas de organizar y controlar el flujo de ejecución del programa, incluyendo estructuras de selección (if, switch) y estructuras de iteración (for, while).

## Esquemas de generación:

Las estructuras deben implementarse de manera que el flujo del programa sea lógico y eficiente. Es crucial manejar correctamente las condiciones y las iteraciones para evitar errores y asegurar un rendimiento óptimo.

## Características:

Se refiere a la organización de datos de un programa. Pueden incluir estructuras de control (como bucles y condicionales) o estructuras de datos (como arrays, listas, pilas, colas, etc.)

## Importancia:

Ayudan a manejar y organizar grandes cantidades de datos de forma eficiente, permitiendo un acceso y manipulación optimizados según las necesidades del programa/código.

- Estructuras de selección: son aquellas que se utilizan para realizar operaciones basadas en el valor de una expresión.

- Estructura if:

Al escribir uno o varios flujos de acción el código contenido en estos se ejecutará siempre y cuando la evaluación de la expresión en la sentencia if se evalúe como verdadera.

if (expresión - booleana)  
{la expresión se evalúa verdadera?}

- Estructura switch:

La estructura de selección switch tiene una expresión de control y los flujos de código alternos son ejecutados dependiendo del valor constante asociado con esta expresión.

Los tipos de datos permitidos para la expresión de control son sbyte, byte, short, ushort, uint, long, ulong, char, string o un tipo enumeración (enumeration).



## Estructuras de selección (SINTAXIS):

switch (expresión - de - control)  
{ case expresion-contante:  
sentencias;  
break;

case expresion-contante:  
sentencias;  
break;  
}

- Estructuras de Iteración: son aquellas que nos permiten ejecutar un bloque de código repetidamente mientras una condición específica sea verdadera.

### • Estructura For:

La estructura for se utiliza cuando se conoce previamente cuantas veces ha de repetirse un bloque de código. Este bloque de código se repetirá mientras la condición evalúe una expresión booleana verdadera, no será posible evaluar otro tipo de expresión.

for (inicializador; condición; iterador)  
{ acciones; }

### • Estructura While:

La estructura while se utiliza cuando no se conoce previamente cuantas veces ha de repetirse un bloque de código, por lo que puede ejecutarse 0 o más veces. Este bloque se repetirá mientras la condición evalúe una expresión booleana verdadera, no será posible evaluar otro tipo de expresión.

while (condicional)  
{ acciones; }

### • Estructura Do :

La diferencia entre la sentencia while y do es que do se evalúa después de su primer iteración, por lo que al menos siempre se ejecuta una vez.

do  
{ acciones; }

while (condición);



Componente	Ventajas	Desventajas
Variables y Constantes	<ul style="list-style-type: none"><li>Claridad en el código</li><li>Facilita la gestión de memoria</li><li>Mejor optimización por parte del compilador.</li></ul>	<ul style="list-style-type: none"><li>Puede haber problemas de rectificación de memoria si no se gestionan correctamente</li><li>Posibles errores de inicialización.</li></ul>
Instrucción de Asignación	<ul style="list-style-type: none"><li>Facilita la generación de código claro y directo.</li><li>Optimización en asignaciones múltiples.</li></ul>	<ul style="list-style-type: none"><li>Puede generar errores en la gestión de tipos si no se manejan adecuadamente.</li></ul>
Expresiones	<ul style="list-style-type: none"><li>Genera código más eficiente y optimizado.</li><li>Reducción de complejidad en cálculos complejos.</li></ul>	<ul style="list-style-type: none"><li>La generación de expresiones complejas puede resultar en código difícil de depurar.</li></ul>
Instrucciones de control	<ul style="list-style-type: none"><li>Mejora la estructura del programa.</li><li>Optimiza el flujo de control y bucles.</li></ul>	<ul style="list-style-type: none"><li>El manejo incorrecto de condiciones puede producir problemas de lógica o ciclos infinitos.</li></ul>
Funciones	<ul style="list-style-type: none"><li>Facilita la modularización del código.</li><li>Permite la rectificación de código.</li><li>Mejora la legibilidad.</li></ul>	<ul style="list-style-type: none"><li>Sobrecarga de tiempo de ejecución si se abusa de las llamadas a funciones.</li><li>Problemas de recursividad mal controlada.</li></ul>
Estructuras	<ul style="list-style-type: none"><li>Facilita el manejo de datos complejos.</li><li>Mejora la organización y claridad del código.</li></ul>	<ul style="list-style-type: none"><li>Incrementa el consumo de memoria.</li><li>Puede ser costoso en tiempo de ejecución cuando se usan estructuras anidadas.</li></ul>

Tabla 1. Ventajas y Desventajas

Cada componente tiene sus ventajas en cuanto a la claridad y optimización del código, pero también puede presentar desventajas si no se manejan correctamente, como errores en la lógica o problemas de eficiencia.



TABLA 2. EFICIENCIA Y COMPLEJIDAD

Componente	EFICIENCIA	COMPLEJIDAD	FACILIDAD DE MANTENIMIENTO	RIESGO DE ERRORES
Variables y Constantes	Alta eficiencia cuando se optimiza bien la memoria	Baja complejidad. Fácil de manejar en casos simples.	Alta, ya que el uso de constantes y variables bien definidas ayuda a la legibilidad.	Moderado, debido a errores de inicialización o uso incorrecto.
Expresiones	Eficiencia alta si se optimizan las operaciones	Complejidad moderada en expresiones simples, pero alta en complejas.	Media depende del número de operaciones y la claridad en su uso.	Alto riesgo si hace demasiada complejidad o mal manejo de operadores.
Instrucción de asignación	Eficiente, especialmente en compiladores que optimizan asignaciones múltiples.	Baja complejidad en asignaciones simples. Aumenta con asignaciones anidadas o múltiples.	Alta, es fácil seguir el flujo de valores.	Bajo, aunque pueden surgir problemas con la gestión de tipos.
Instrucciones de Control	Eficiente si las condiciones y bucles están bien optimizados.	Complejidad alta en casos de anidación profunda o condiciones complejas.	Moderada, ya que un mal manejo puede generar código confuso.	Alto, específicamente en la lógica de condiciones y bucles infinitos.
Funciones	Moderada, puede haber sobrecarga por las llamadas a funciones.	Baja en funciones simples. Alta en funciones recursivas o con muchos parámetros.	Alta, facilita la modularización y rectificación del código.	Alto, especialmente en funciones recursivas mal controladas.
Estructuras	Moderada a baja, dependiendo del tamaño y complejidad de la estructura	Alta, ya que manejar estructuras complejas puede requerir más esfuerzo.	Media, puede dificultarse si hay muchas estructuras anidadas.	Moderado, sobre todo en la gestión de la memoria y el acceso a datos.



## Conclusion :

La generación de código intermedio es una fase esencial en la compilación que convierte el código fuente en una forma más abstracta y optimizada antes de producir el código máquina final. Este proceso implica la creación de estructuras que representan las diversas construcciones del lenguaje de programación, tales como variables, constantes, expresiones, instrucciones de asignación, instrucciones de control, funciones y estructuras.

La correcta implementación de estos esquemas de generación es fundamental para optimizar el rendimiento de los programas y asegurar su correcto funcionamiento. Al comprender y manejar adecuadamente estas estructuras, se logra una ejecución eficiente y efectiva del código, lo que redundaría en aplicaciones más rápidas y confiables.

## Referencias:

- De código G. (S/F). (11/26) Procesadores de lenguaje. Uji.es. Recuperado el 27 de Octubre de 2024, de <https://www3.uji.es/~vjmenez/AULASVIRTUALES/PL-0910/T5-GENERACION/codigo.apun.pdf>
- EXP2EQ4 LEA2B.pptx. (S/F) SlideShare. Recuperado el 27 de Octubre de 2024, de <http://es.slideshare.net/slideshow/exp2eq4lea2b.pptx/253213261>
- Guest (2021, Enero 14) Pdfcoffee.com. 2.3 esquemas de generación . Pdfcoffee.com. <https://pdfcoffee.com/23-esquemas-de-generacion-2-pdf-free.html>

## Referencias

- Algoritmo, E., Es, Y., & del ferrocarril., M. E. en la N. de I. P. S. U. P. P. la S. en la N. P. I. o. C. Á. de S. A. E. A. F. I. P. E. D. y. N. C. A. "shunting Y. P. su O. se A. al de un P. de C. (s/f). Algoritmo shunting yard. Buap.mx. Recuperado el 29 de octubre de 2024, de <https://www.cs.buap.mx/~andrex/estructuras/AlgoritmoPolacasPosfijo.pdf>
- Compilers: Principles, techniques, and tools. (2007). Addison-Wesley Professional.
- Cosmico, C. (s/f). Código Java. Blogspot.com. Recuperado el 29 de octubre de 2024, de <https://censorcosmico.blogspot.com/2012/09/primeros-pasos-para-conversion-infijo.html>
- KNUTH. (1997). The art of computer programming: Volume one fundamental algorithms wss: Knuth: Art comp prog 3e wss: Volume one fundamental algorithms wss (3a ed.). Addison Wesley.
- Villegas, F. J. D. (2019, septiembre). Notaciones Prefija, Infija y Postfija (Con ejemplos de conversiones). [\(S/f\). Runestone.academy. Recuperado el 29 de octubre de 2024, de <https://runestone.academy/ns/books/published/pythoned/BasicDS/ExpresionesInfijasPrefijasYSufijas.html>](https://www.youtube.com/watch?v=PgZfeEZnwg&ab_channel=FranciscoJavierDiazVillegas)
- Martínez, C. (s.f.). Lenguajes y autómatas 2: Representación de código intermedio. Blogspot. Recuperado de <https://lenguajesyautomatas2.blogspot.com>
- Tecnológico Nacional de México en Zacatecas. (s.f.). Unidad 2: Generación de código intermedio. Recuperado de <https://enlinea.zacatecas.tecnm.mx>
- ITZ. (s.f.). Lenguajes y Autómatas II - Representación de código intermedio. Recuperado de <https://5e344735705b1.site123.me>
- ByteZero. (s.f.). Lenguajes y autómatas 2: Notación y representación de código intermedio. Recuperado de <https://l-byte0.github.io>

## Link a video

<https://youtu.be/x19g7liptUY>