LENGUAJES Y AUTOMATAS II

Tecnológico Nacional de México

Ingeniería en sistemas computacionales

Ing. Jesus Alberto Espinoza Arzola

SALTILLO, COAHUILA AGOSTO 2023

-Árbol -expresiones -operaciones -ramas -análisis -código -datos -nodo

SALTILLO, COAHUILA AGOSTO 2023

LENGUAJES Y AUTOMATAS II

Instituto tecnológico de México campus saltillo

Ingeniería en Sistemas Computacionales

Ing. Jesus Alberto Espinoza Arzola

SALTILLO, COAHUILA AGOSTO 2023

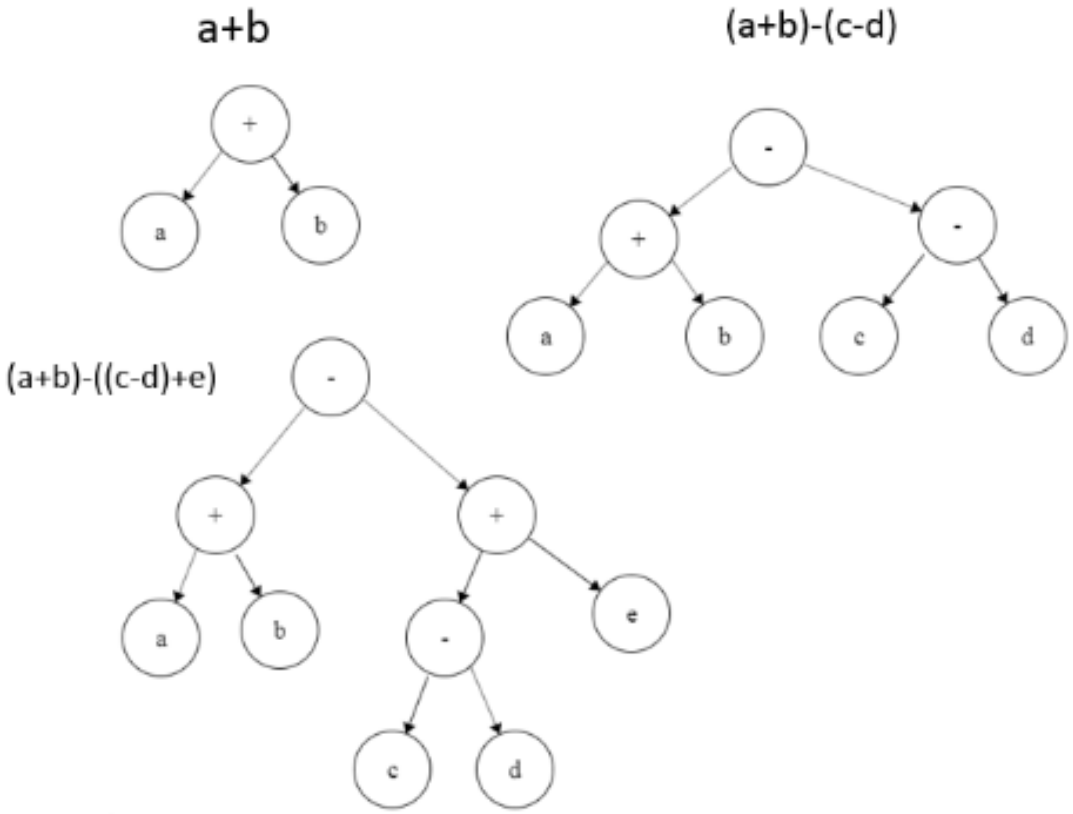
LENGUAJES Y AUTOMATAS II

No.	Temas	Subtemas7
1.	Análisis semántico.	1.1 Árboles de expresiones. 1.2 Acciones semánticas de un analizador sintáctico. 1.3 Comprobaciones de tipos en expresiones. 1.4 Pila semántica en un analizador sintáctico. 1.5 Esquema de traducción. 1.6 Generación de la tabla de símbolo y tabla de direcciones. 1.7 Manejo de errores semánticos.

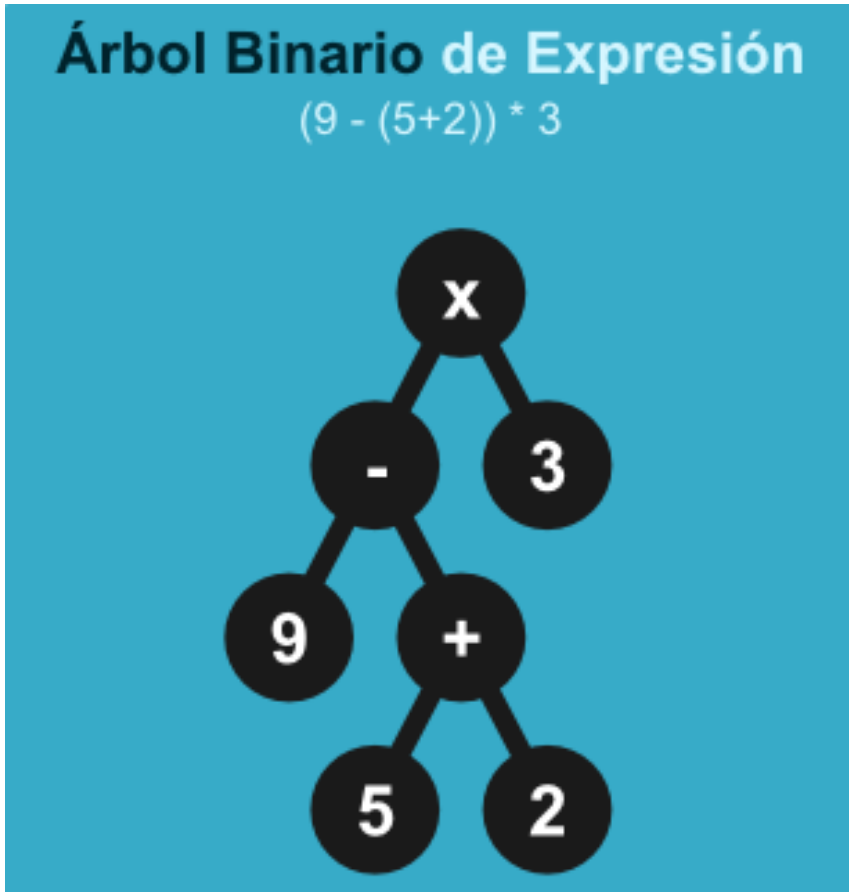
1.1 Árboles de expresiones o *árbol semántico.*

- **Es una estructura jerárquica en la cual se registran las operaciones que realiza el programa fuente.**
- **En cada una de las ramas del árbol se registra el valor o significado que este debe tener y el análisis se encarga de terminar cuál de los valores registrado en la rama es aplicable.**
- **Los árboles de expresiones representan el código de nivel del lenguaje en forma de datos. Los datos se almacenan en una estructura con forma de árbol.**
- **Cada nodo del árbol de expresión representa como tal, una expresión.**

1.1 Árboles de expresiones o *árbol semántico.*



1.1 Árboles de expresiones o *árbol semántico.*



1.2 Acciones semánticas de un analizador sintáctico

Se encargan de que los tipos que intervienen en las expresiones sean compatibles o que los parámetros reales de una función sean coherentes con los parámetros formales.

¿Qué hace un análisis semántico?

El análisis semántico, es un método de procesamiento del lenguaje natural, consiste en examinar el significado de las palabras y frases para comprender el propósito de una oración o párrafo.

¿Cómo funciona un analizador sintáctico?

Su principal función es analizar la secuencia de componentes léxicos de la entrada para verificar que cumplen con las reglas gramaticales especificadas. Esta interacción se aplica bajo un esquema donde el analizador léxico es una subrutina o corutina del analizador sintáctico.

1.3 Comprobaciones de tipos en expresiones

La labor de comprobación de tipos consiste en conferir a las construcciones sintácticas del lenguaje, la semántica de tipificación y en realizar todo tipo de comprobaciones de dicha índole. Por su naturaleza, sin embargo, ésta se encuentra repartida entre la fase de análisis semántico y la generación de código intermedio.

- **Comprobaciones estáticas**

Las comprobaciones estáticas recogen el compendio de todas aquellas tareas de carácter semántico que, por su naturaleza, pueden ser realizadas directamente durante la fase de compilación mediante el uso de artefactos y mecanismos propios de dicha fase. Este tipo de comprobaciones son beneficiosas puesto que confieren seguridad a la ejecución del programa.

1.3 Comprobaciones de tipos en expresiones

1. **Verificar que los tipos y valores asociados a los objetos de un programa se utilizan de acuerdo con la especificación del lenguaje.**

2. **Detectar conversiones implícitas de tipos para efectuarlas o insertar el código apropiado para efectuarlas**

Almacenar información relativa a los tipos de los objetos

1.2 Acciones semánticas de un analizador sintáctico

¿Qué es la tabla de símbolos en lenguajes y autómatas?

La tabla de símbolos (TS) es la estructura utilizada por el compilador para almacenar los atributos asociados a los símbolos que se utilizan en un lenguaje de programación. Los atributos que esta estructura almacena para cada símbolo puede ser: Tipo: entero, real, char, boolean.

Aplicar las reglas de verificación de tipos ¿cómo y cuándo aplicarlas?

- **Equivalencia: determina cuándo dos objetos pueden considerarse del mismo tipo.**
- **Compatibilidad: determina cuándo un objeto de cierto tipo puede ser usado en un cierto contexto.**
- **Inferencia: derivación del tipo de un objeto a partir de sus componentes.**
- **Conversión: permitir y efectuar un cambio de tipo.**
- **Coerción: conversión automática de un tipo a otro.**

1.2 Acciones semánticas de un analizador sintáctico

Ejemplo

```
using System;
using System.Windows.Forms;

namespace Sumade2
{
   public partial class Form1 : Form
   {
      int a, b, r;
      public Form1()
      {
         InitializeComponent();
      }

      private void button1_Click(object sender, EventArgs e)
      {
         a = Convert.ToInt32(textBox1.Text);
         b = Convert.ToInt32(textBox2.Text);

         r = a + b;

         label1.Text = r.ToString();
      }
   }
}
```

Tabla de símbolos

No. De línea	Nombre del identificador	Categoría	Tipo	Longitud	Argumentos
1	using	Directiva	Palabra reservada	---	Sirve para importar espacios de nombres
1	System	Función	Palabra reservada	---	indica que las funciones que estamos usando pertenecen a la estructura básica de C# y de la plataforma
1	;	Fin de línea	---	---	Indica el final de una línea de instrucción

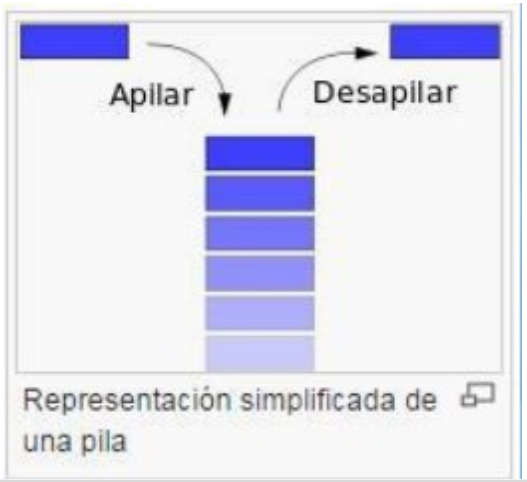
1.4 Pila semántica en un analizador sintáctico.

› **PILA**

› **Estructura de datos que se usa en programación para simplificar ciertas operaciones**

› **Arrays* Listas enlazadas ***

› **Una colección de datos a los que se puede acceder mediante un extremo, que se conoce generalmente como tope.**



Representación simplificada de una pila

Push
Pop
LIFO
TOS

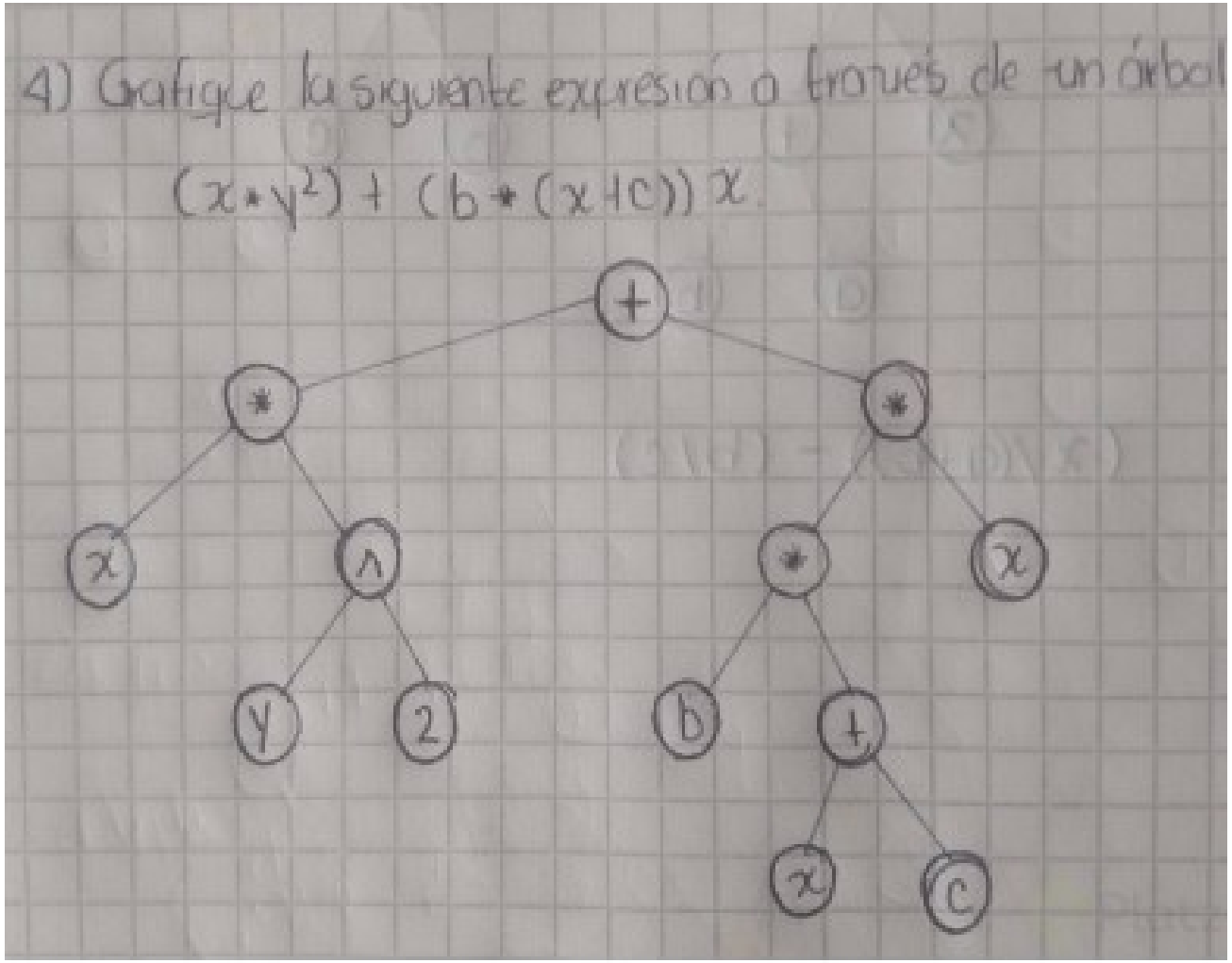
1.4 Pila semántica en un analizador sintáctico.

› El análisis semántico usa como entrada el árbol sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

› ¿Para qué se usa la pila ?

› Para contener la información semántica asociada a los operandos (y operadores) en forma de registros semánticos tomando en cuenta las reglas semánticas.
(Conj. de normas y especificaciones que definen al lenguaje).
Conversiones implícitas

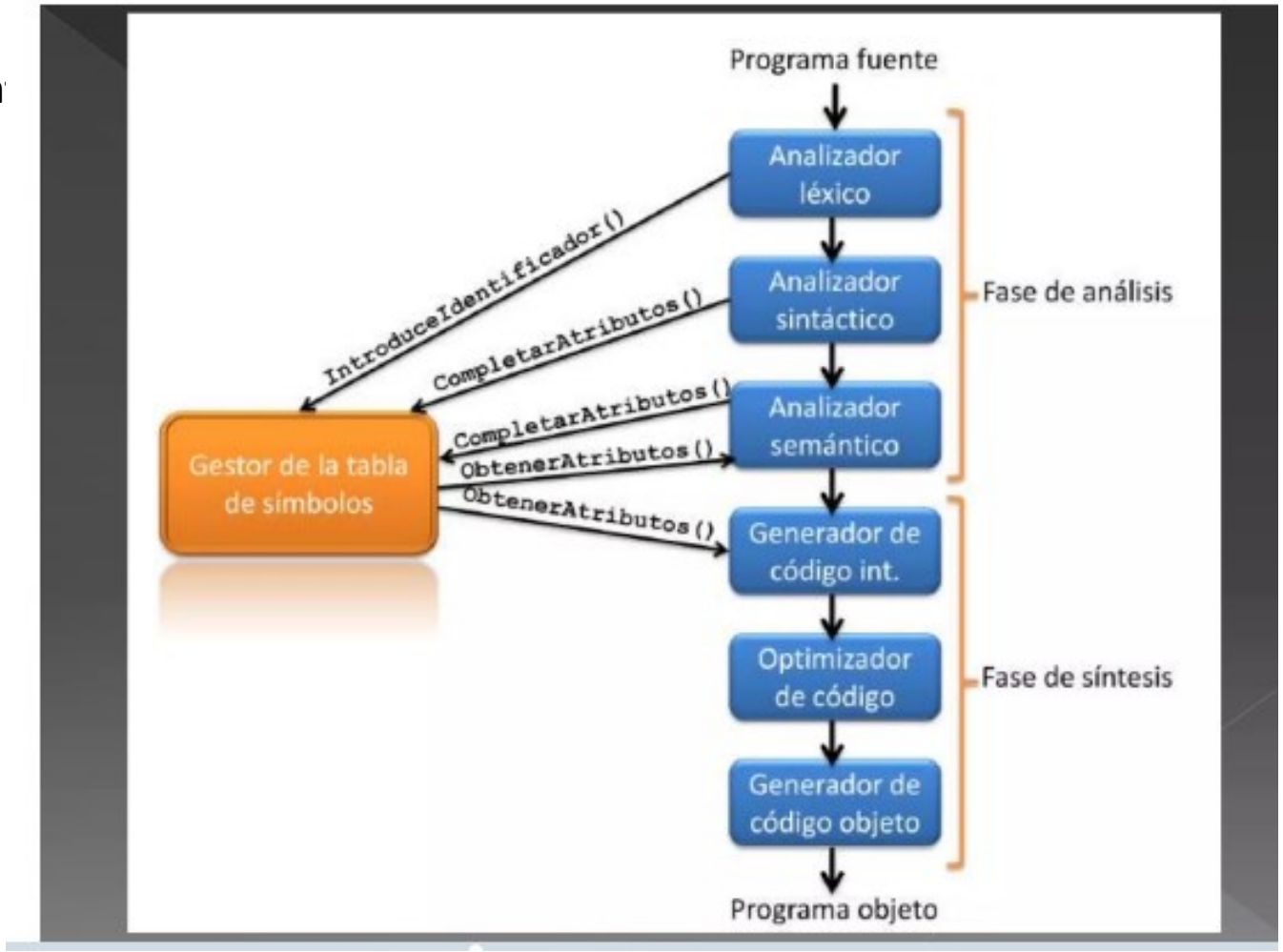
LENGUAJES Y AUTOMATAS II



Sistema de tipos

- Funciones principales:
  - Inferencia de tipos: calcular y mantener la información sobre los tipos de datos.
  - Verificación de tipo: asegurar que las partes de un programa tienen sentido según las reglas de tipo del lenguaje.
- La información de tipos puede ser estática o dinámica:
  - LISP, CAML o Smalltalk utilizan información de tipos dinámica.
  - En ADA, Pascal o C la información de tipos es estática.
  - También puede ser una combinación de ambas formas.
- Cuantas más comprobaciones puedan realizarse en la fase de compilación, menos tendrán que realizarse durante la ejecución
  - Mayor eficiencia del programa objeto.

1.4 Pila semán



LENGUAJES Y AUTOMATAS II

2	Generación de código intermedio.	2.1 Notaciones. 2.1.1 Prefija. 2.1.2 Infija. 2.2.3 Postfija. 2.2 Representaciones de código Intermedio. 2.2.1 Notación Polaca. 2.2.2 Código P. 2.2.3 Triplos. 2.2.4 Cuádruplos. 2.3 Esquema de generación. 2.3.1 Variables y constantes. 2.3.2 Expresiones. 2.3.3 Instrucción de asignación. 2.3.4 Instrucciones de control. 2.3.5 Funciones. 2.3.6 Estructuras.

2.0 Generación de código intermedio.

La mayoría de los compiladores generan código como parte del proceso de análisis sintáctico, esto es debido a que requieren del árbol de sintaxis y si este no va a ser construido físicamente, entonces deberá acompañar al analizador sintáctico al barrer el árbol implícito.

2.0 Generación de código intermedio.

En lugar de generar código ensamblador directamente, los compiladores generan un código intermedio que es más parecido al código ensamblador, las operaciones por ejemplo nunca se hacen con más de dos operandos.

Al no generarse código ensamblador el cual es dependiente de la computadora especifica, sino código intermedio, se puede reutilizar la parte del compilador que genera código intermedio en otro compilador para una computadora con diferente procesador cambiando solamente el generador de código ensamblador al cual llamamos back-end, la desventaja obviamente es la lentitud que esto conlleva.

LENGUAJES Y AUTOMATAS II

La tarea de síntesis suele comenzar generando un código intermedio.
El código intermedio no es el lenguaje de programación de ninguna máquina real, sino que corresponde a una máquina abstracta, que se debe de definir lo más general posible, de forma que sea posible traducir este código intermedio a cualquier máquina real.

El objetivo del código intermedio es reducir el número de programas necesarios para construir traductores, y permitir más fácilmente la transportabilidad de unas máquinas a otras. Supóngase que se tienen n lenguajes, y se desea construir traductores entre ellos. Sería necesario construir n*(n-1) traductores.

Sin embargo, si se construye un lenguaje intermedio, tan sólo son necesarios 2*n traductores.

Así por ejemplo un fabricante de compiladores puede construir un compilador para diferentes máquinas objeto con tan sólo cambiar las dos últimas fases de la tarea de síntesis.

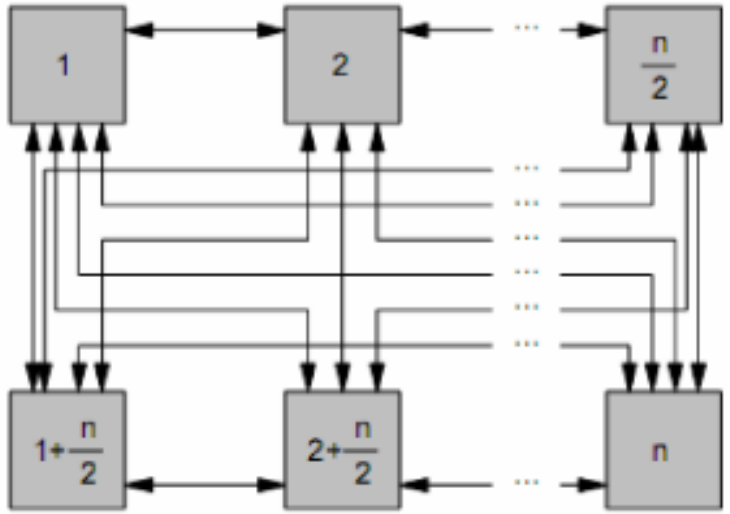


Figura 1. n*(n-1) traductores entre n lenguajes

2.0 Generación de código intermedio.

- Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológico y sintáctico.

- El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

- En compiladores de un solo paso, las llamadas a las rutinas semánticas se realizan directamente desde el analizador sintáctico y son dichas rutinas las que llaman al generador de código.

- El instrumento más utilizado para conseguirlo es la gramática de atributos.

2.1 Notaciones.

- **Las notaciones son una forma especial en la que se pueden expresar una expresión matemática y puedan ser de 3 formas: infija, prefija y posfija.**

Los prefijos, Pre - Pos - In se refieren a la posición relativa del operador con respecto a los dos operandos.

Las notaciones sirven de base para expresar sentencias bien definidas.
El uso más extendido de las notaciones sirve para expresar operaciones aritméticas.
Las expresiones aritméticas se pueden expresar de tres formas distintas: infija, prefija y postfija. La diversidad de notaciones corresponde en que para algunos casos es más sencillo un tipo de notación.

2.1.1 Prefija

La notación prefija, también conocida como notación de prefijo, es una forma de notación para la lógica, la aritmética, y el álgebra. Su característica distintiva es que coloca los operadores a la izquierda de sus operandos. Si la paridad (es el número de argumentos necesarios para que dicho operador o función se pueda calcular.) de los operadores es fija, el resultado es una sintaxis que carece de paréntesis u otros signos de agrupación, y todavía puede ser analizada sin ambigüedad.

2.1.1 Prefija

La expresión o notación prefija nos indica que el operador va antes de los operandos sus características principales son:

Los operadores conservan el mismo orden que la notación infija equivalente.

No requiere de paréntesis para indicar el orden de precedencia de operadores ya que él es una operación.

Se evalúa de izquierda a derecha hasta que encuentra el primer operador seguido inmediatamente de un par de operando.

Se evalúa la expresión binaria y el resultado se cambia como un nuevo operando. Se repite hasta que nos quede un solo resultado.

El orden es operador, primer operando, segundo operando.

EJEMPLO:

Si deseamos representar las expresiones (2+(3*4)) = *x* y ((2+3)*4)= *x* en las tres notaciones mencionadas, el resultado sería:

	(2+(3*4)) = *x*	**((2+3)*4) = *x***
Notación prefija	= + 2 * 3 4 *x*	= * + 2 3 4 *x*
Notación infija	2+3*4 = *x*	(2+3)*4 = *x*

2.1.1 Prefija

Expresión infija
A + B * C + D
(A + B) * (C + D)
A * B + C * D
A + B + C + D

Expresión infija	Expresión prefija
A + B * C + D	+ + A * B C D
(A + B) * (C + D)	* + A B + C D
A * B + C * D	+ * A B * C D
A + B + C + D	+ + + A B C D

2.1.1 Posfija

La notación postfija pone el operador al final de los dos operandos, por lo que la expresión queda:
ab+5-La notación postfija utiliza una estructura del tipo LIFO (ab+5-Last First Out) pila, la cual es la más utilizada para la implementación.
Llamada también polaca inversa, se usa para representar expresiones sin necesidad de paréntesis.

2.1.1 Posfija

Como su nombre lo indica se refiere a que el operador ocupa la posición después de los operandos sus características principales son:

-El orden de los operandos se conserva igual que la expresión infija equivalente no utiliza

paréntesis ya que no es una operación ambigua.

-La operación posfija no es exactamente lo inverso a la operación prefija equivalente:

(A+B)*C AB+C*

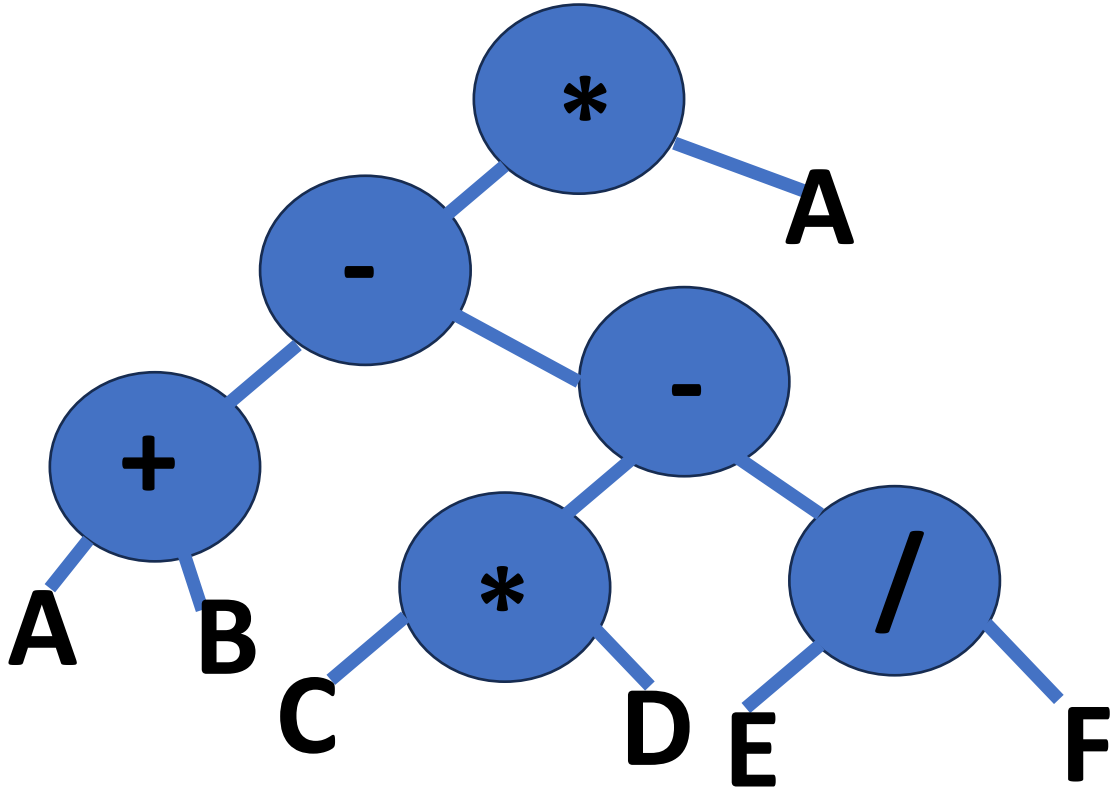
Notación postfija: El orden es primer operando, segundo operando, operador.

EJEMPLO:

Si deseamos representar las expresiones (2+(3*4)) = *x* y ((2+3)*4)= *x* en las tres notaciones mencionadas, el resultado sería:

	(2+(3*4)) = *x*	**((2+3)*4) = *x***
Notación prefija	= + 2 * 3 4 *x*	= * + 2 3 4 *x*
Notación infija	2+3*4 = *x*	(2+3)*4 = *x*
Notación postfija	2 3 4 * + *x* =	2 3 + 4 * *x* =
Notación funcional	igual(suma(2,producto(3,4)),*x*)	igual(producto(suma(2,3),4),*x*)

ACTIVIDAD



PREFIJA

INFIJA

POSFIJA

PREFIJA	***--+AB-*CD/EFA**
INFIJA	**A+B-C*D-E/F*A**
POSFIJA	**AB+CD*EF/--A***

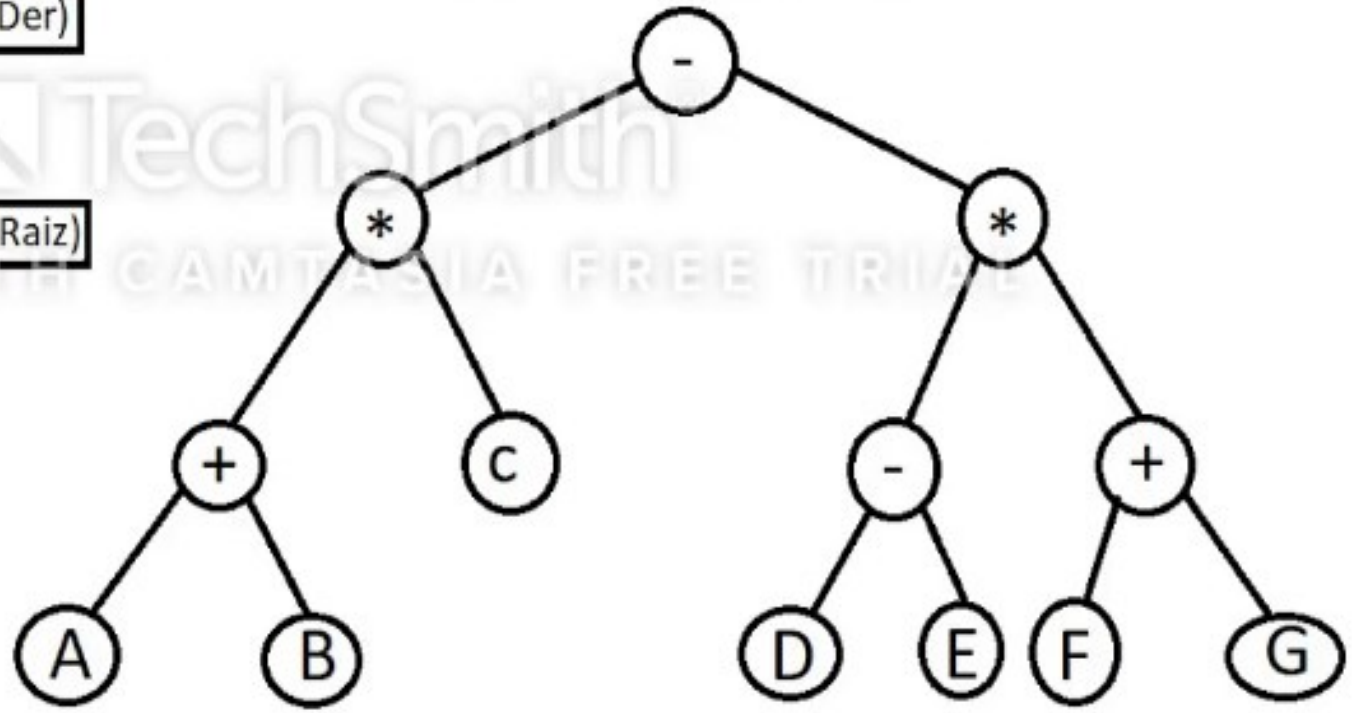
LENGUAJES Y AUTOMATAS II

Infijo	Recorrido inorden (Izq-Raiz-Der)
Prefijo	Recorrido Preorden (Raiz-Izq-Der)
Postfijo	Recorrido Postorden (Izq-Der-Raiz)

Notación infija, prefija y postfija

Dada una expresión:

(A + B) * C - (D - E) * (F + G)



ACTIVIDAD

(5+8) (9-3)*2

ACTIVIDAD

$((2+3)*4) = x$

$(2+(3*4)) = x$

$(7-(9/5)) + (5*(3*4)) = x$

$5*4+((7/2)-3)$