INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA
DE INGENIERÍA Y CIENCIAS SOCIALES
Y ADMINISTRATIVAS

“DESARROLLO DE UN COMPILADOR PARA PSEUDOCÓDIGO EN LENGUAJE ESPAÑOL”

T E S I S

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN INFORMÁTICA

P R E S E N T A N
ISAAC ANDRÉS CANALES MARTÍNEZ
MICHEL RUIZ TEJEIDA

MÉXICO, D.F. **2011**

ÍNDICE

RESUMEN

Cuando se enseña o aprende a utilizar un lenguaje de programación, es un supuesto el hecho de que el desarrollador ya tiene la habilidad de generar algoritmos para dar soluciones a problemas, los cuales deberá seguir el programa desarrollado. No es buena práctica programar mientras se busca el algoritmo que da solución al problema, sino que es recomendable utilizar herramientas de apoyo tales como diagramas de flujo o pseudocódigo para plantear de forma genérica los pasos a seguir, y posteriormente traducir a instrucciones particulares del lenguaje a utilizar.

Un pseudocódigo es una técnica utilizada para plantear un algoritmo, cuya característica principal es la no dependencia de la sintaxis de algún lenguaje de programación específico pero si presenta una estructura parecida a alguno, lo anterior tiene la bondad de que la transformación a un lenguaje cualquiera varía muy poco del pseudocódigo original.

El objetivo de éste trabajo es mostrar el desarrollo de una herramienta capaz de procesar pseudocódigo, con reglas de sintáxis y palabras reservadas, para generar la transformación de dicho pseudocódigo a un lenguaje de alto nivel, específicamente Java, el cual se puede compilar y ejecutar.

La herramienta desarrollada está orientada a ser usada por personas que están aprendiendo a programar, lo anterior se debe a que facilita la generación de código Java a partir de pseudocódigo en español, además disminuye en gran medida que los aprendices de programación se tengan que preocupar por tipos de datos, palabras en inglés, puntos y coma, y algunos otros detalles que el compilador de un lenguaje de alto nivel exige para la correcta compilación del código fuente, esto genera que la preocupación principal sea la generación del algoritmo.

INTRODUCCIÓN

Sin lugar a dudas la programación es una de las habilidades que todos los estudiantes de Informática y carreras afines deben dominar, pero antes de comenzar a aprender algún lenguaje en específico, es muy útil el uso de técnicas que permitan a los estudiantes comenzar a obtener la lógica de programación.

Algunas herramientas utilizadas para la enseñanza de la programación son los diagramas de flujo y el pseudocódigo, siendo éste último seleccionado como base para desarrollar una herramienta que simplifique aún más la tarea de traducir el pseudocódigo a un lenguaje de programación de alto nivel.

La traducción de un lenguaje a otro, específicamente hablando, del pseudocódigo en español a un lenguaje de alto nivel, requiere de análisis de diferentes tipos, para asegurar que las reglas definidas en el pseudocódigo se cumplan y se puedan transformar, dicho análisis y transformación son posibles si se utiliza un compilador. Entonces, la herramienta que aquí se describe es un compilador (como lo son aquellos de lenguajes de alto nivel como C y Java) que recibe como entrada un archivo que contiene el pseudocódigo, y genera como salida un archivo Java, el cual es compilable y ejecutable por la Java Virtual Machine.

Éste trabajo, en primer lugar muestra todos aquellos conceptos que se deben tener presentes al hablar de compiladores. Posteriormente brinda un panorama general de la estructura y funcionamiento de los mismos sin detallar las técnicas y herramientas para su construcción. Así mismo, se establecen convenciones en tipografía e imágenes utilizadas a lo largo del documento.

Las técnicas, procedimientos y fundamentaciones matemáticas de los conceptos generales, se encuentran en el capítulo denominado “Marco Teórico”. Éste capítulo está seccionado de acuerdo a las fases que sigue un compilador, con la intención de lograr una secuencia clara entre los conceptos del capítulo I y el desarrollo que se muestra en el capítulo V.

El capítulo III llamado “Pseudocódigo”, muestra la definición de reglas y palabras reservadas que presenta el pseudocódigo en español que es procesable por el compilador, así como también se especifica la manera de declarar variables, estructuras repetitivas y selectivas, bloques de código, subrutinas y operadores.

La fundamentación matemática permite expresar de forma genérica el comportamiento que tendrá el compilador, siguiendo como base aquellos conceptos que fueron detallados en los capítulos I y II. El capítulo “Fundamentación Matemática” demuestra cómo funcionan los autómatas para el análisis léxico y la gramática libre de contexto usada para el análisis sintáctico.

Finalmente, se expone el desarrollo computacional del compilador en el capítulo V. Como se mencionó previamente, éste capítulo está estructurado de acuerdo a las etapas que sigue un proceso de compilación; se menciona el uso de herramientas que facilitan el desarrollo de analizadores léxicos y sintácticos y se muestran también corridas del compilador terminado.

CAPÍTULO I INTRODUCCIÓN AL ESTUDIO DE COMPILADORES

1.1 ¿QUÉ ES UN COMPILADOR?

Uno de los principales mecanismos de interacción entre una computadora y un usuario viene dado por el envío y recepción de mensajes textuales, el usuario ingresa ciertos comandos por medio del teclado y observa los resultados en una pantalla. El proceso interno de la computadora lo realiza el traductor.

“Un traductor se define como un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino produciendo, si cabe, mensajes de error.” (Galvez Rojas & Mora Mata, 2005).

Los traductores engloban tanto a los compiladores como a los intérpretes del cual podemos definir:

Un compilador, “es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, realiza una traducción de un lenguaje de alto nivel a código máquina, (también se entiende por compilador aquel programa que proporciona un fichero objeto en lugar del ejecutable final)” (Galvez Rojas & Mora Mata, 2005)

Un intérprete, “es como un compilador, solo que la salida es una ejecución. El programa de entrada se reconoce y ejecuta a la vez. No se produce un resultado físico (código máquina) sino lógico (una ejecución).” (Galvez Rojas & Mora Mata, 2005)

1.2 ANTECEDENTES HISTÓRICOS DEL COMPILADOR

Durante la década de los años 40 se crea la primer computadora digital, una computadora es una herramienta electrónica que permite almacenar, organizar y procesar información; sin embargo para su funcionamiento debe ser programada, es decir, el usuario debe indicar las órdenes necesarias para realizar alguna tarea específica.

En un principio las órdenes no eran más que un alfabeto numérico mejor conocido como código binario o lenguaje máquina, las operaciones que se realizaban eran de muy bajo nivel: mover datos de una ubicación a otra, sumar el contenido de registros, comparar datos, entre otras cosas, éste tipo de programación era lenta, tediosa y propensa a errores, sin mencionar el problema de mantener un sistema o peor aún, modificarlo. A razón de este hecho se implementaron claves más fáciles de recordar que los códigos numéricos, traduciéndolas manualmente a lenguaje máquina.

Estas claves constituían el llamado lenguaje ensamblador, el cual se generaliza al automatizarse los procesos de traducción. Aunque estas palabras permitían reducir el número de líneas al realizar un programa, todavía resultaba complicado poder estructurar una lógica de programación.

En la búsqueda de un método que permitiera facilitar el desarrollo de aplicaciones un grupo de trabajo de IBM (International Business Machines) comenzó el desarrollo de una técnica que sustituyera los comandos del lenguaje ensamblador por expresiones con una lógica más natural al habla o la escritura; aunque en un principio se rechazó la idea de crear programas a partir de un código en el que no se apreciaban las sentencias directas que la máquina ejecutaría, con el uso del lenguaje y la comprobación del funcionamiento, se convirtió en el primer lenguaje de alto nivel, mejor conocido como FORTRAN y con esto la aparición del primer compilador.

El desarrollo del compilador utilizado por el lenguaje FORTRAN tardó poco más de una década en su implementación, uno de los principales problemas que presentaron durante ese tiempo fue la arquitectura de la máquina en que se utilizaría.

A finales de la década de los años 50, se ofreció una solución al problema de que un compilador fuera utilizable por varias máquinas, ésta consistía en dividir el compilador en dos etapas, una inicial (análisis) y otra final (de síntesis) que se denominaron “front-end” y “back-end”. En la etapa inicial se analiza el programa fuente y la etapa final es la encargada de generar el código utilizado por la máquina objeto. La unión de estas dos etapas se realiza por medio de un lenguaje intermedio universal, este lenguaje, ampliamente discutido pero nunca implementado, se denominó UNCOL (Universal Computer Oriented Language) (Steel, 1961).

Esta idea origina la división del compilador en etapas o fases:

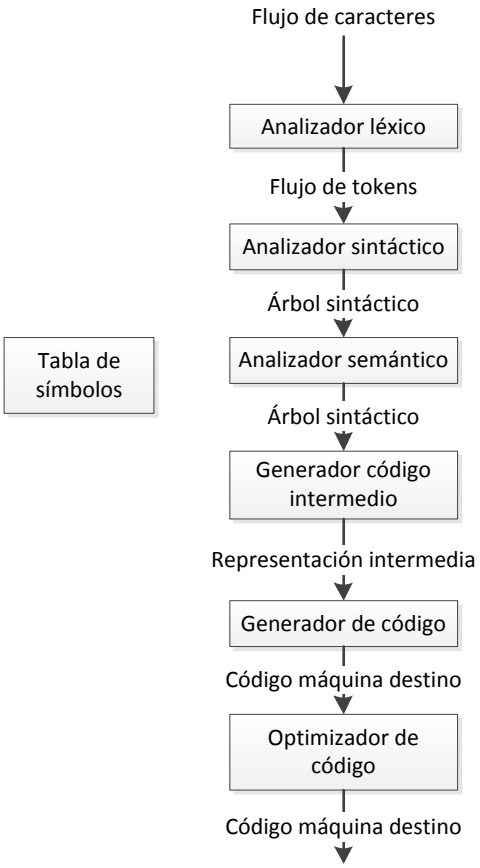


Ilustración 1 Fases de un compilador

1.3 ESTRUCTURA DE UN COMPILADOR

El objetivo principal de esta investigación es la creación de un compilador, para cubrir dicho propósito es necesario conocer como está compuesto.

1.3.1 ANÁLISIS LÉXICO

Es la primera fase del compilador que se encarga de leer el código fuente y procesarlo, también es conocido como escaneo por su nombre en inglés (scanner). Durante este proceso se realizan operaciones básicas de manejo de cadenas de acuerdo a ciertas reglas del lenguaje, estas reglas las conocemos en teoría computacional como expresiones regulares.

1.3.2 ANÁLISIS SINTÁCTICO

La siguiente fase de compilación es el analizador sintáctico (parser), en donde se analiza la estructura gramatical del lenguaje fuente, estas reglas son representadas por las gramáticas libres del contexto y su escaneo con los árboles sintácticos.

1.3.3 ANÁLISIS SEMÁNTICO

En la tercera etapa aún se analiza el código fuente para verificar las reglas semánticas, estas reglas están representadas por la correspondencia de los tipos de datos que se manejen en el lenguaje.

1.3.4 GENERADOR DE CÓDIGO INTERMEDIO

Una vez analizado el código fuente y sin haber encontrado algún error durante las fases anteriores, el compilador genera un código intermedio para prepararlo al código destino.

1.3.5 GENERADOR DE CÓDIGO

La quinta etapa del compilador también conocida como generador de código o back-end, es el proceso encargado de traducir el lenguaje intermedio a un código máquina, el cual dependerá de la arquitectura de la misma.

1.3.6 OPTIMIZADOR DE CÓDIGO

El último proceso de un compilador es opcional, ya que la optimización de código depende de la arquitectura de la máquina, si ésta acepta paralelismo y que tipo de paralelismo, además de que no es posible saber a ciencia cierta si un código optimizado es mejor que el código ingresado por el programador.

1.4 NOTACIONES Y CONVENCIONES PARTICULARES

Las siguientes notaciones y convenciones son utilizadas a lo largo de éste trabajo escrito:

- ε elemento vacío
- ∅ conjunto vacío
- { } conjunto de elementos
- | operador OR
- · operador AND

Gramáticas

α→β β representa una producción de α

Expresiones regulares

- * cerradura de Kleene
- | operador OR
- · operador AND
- + cerradura positiva

Autómatas

-  estado no terminal
-  estado final o terminal
-  transición de estado por medio del símbolo ‘a’

Pseudocódigo:

- `Ancho fijo` .- utilizado para denotar que es código del compilador
- *`Ancho fijo itálica`* .- denota código que es sustituible por alguna palabra reservada o expresión
- `<contenido>` .- denota que el contenido entre los signos mayor que y menor que, son opcionales

Representaciones numéricas

- Sufijo - primero se anota el operador y después el operando
- Infijo - por cada operador debe haber un operando a la izquierda y uno a la derecha

CAPÍTULO II MARCO TEÓRICO

Al desarrollar un compilador, se debe realizar una minuciosa investigación con el propósito de conocer a detalle las partes que lo integran, un compilador está estructurado por un analizador léxico, un analizador sintáctico, un analizador semántico, un generador de código intermedio, un generador de código, un optimizador de código y una tabla de símbolos que almacena información del programa fuente, la cual se utiliza en todas las fases del compilador.

2.1 ANALIZADOR LÉXICO

La primera fase de un compilador es leer el lenguaje de entrada, en este punto el analizador léxico se encarga de agrupar los caracteres en lexemas y producir una salida en secuencia de tokens para cada lexema en el programa fuente. Durante este proceso de reconocimiento de lexemas el analizador léxico envía el flujo de tokens al analizador sintáctico e interactúa con la tabla de símbolos, de tal forma que por cada identificador que encuentre, debe introducirse ese lexema en la tabla de símbolos.

En la Ilustración 2, se representan la relación que tienen los analizadores léxico y sintáctico con la tabla de símbolos, el analizador sintáctico solicita al léxico el siguiente token por medio de la función representada como *obtenerToken,* el léxico lee los caracteres del programa fuente hasta que puede formar un lexema y producir el token para devolverlo al analizador sintáctico.

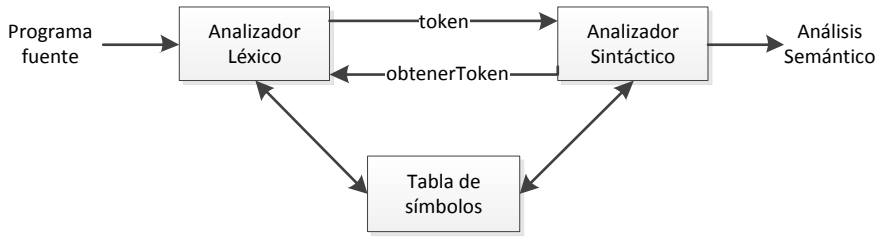


Ilustración 2 Interacciones del analizador léxico y sintáctico

Como el analizador léxico es el encargado de leer el programa fuente, además de formar los lexemas se ocupa de eliminar los cometarios y espacios en blanco (caracteres de espacio, tabuladores, retorno de línea o cualquier caracter que separe los tokens).

2.1.1 TOKENS Y LEXEMAS

- *Token*: “es un par que consiste en un nombre de token y un valor de atributo opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico” (Aho, Lam, Sethi, & Ullman, 2008).
- *Lexema*: “es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.” (Aho, Lam, Sethi, & Ullman, 2008)

Para cada lexema, el analizador léxico produce como salida un token de la forma:

⟨nombreToken,valorAributo⟩

En el token, el primer componente *nombreToken* es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el siguiente componente *valorAtributo* apunta a una entrada en la tabla de símbolos. La entrada a la tabla de símbolos se necesita más tarde en el analizador semántico y generador de código; supongamos que el analizador léxico encuentra dos tokens de la forma **identificador**, para los conjuntos de caracteres **suma** y **resta**, para esta etapa de reconocimiento bastará con dicha información; pero el generador de código necesita saber la posición y uso de éstos identificadores en el programa fuente.

Para entender la forma en que el analizador léxico realiza esta identificación de lexemas se propone la siguiente instrucción:

```
velocidad = distancia / tiempo
```

Los caracteres en esta instrucción se pueden agrupar en los siguientes lexemas:

1. **velocidad** es un lexema que se representa por la forma ⟨id|1⟩, en donde **id** representa de forma abstracta la palabra identificador y **1** la posición en que se encuentra el lexema en la tabla de símbolos.
2. El símbolo de asignación **=** se representa por el token ⟨=⟩, en éste caso no es necesario agregar un *valorAtributo* o una representación abstracta para el *nombreToken*, debido a que dentro de la especificación del lenguaje se identifica como un operador[1].

[1] Tanto operadores como palabras reservadas del lenguaje se cargan en la tabla de símbolos, desde el inicio del proceso de compilación.
[2] La tabla de símbolos es volátil, ya que su información solamente estará presente en el proceso de traducir el

3. **distancia** es un lexema al que se le asigna la forma ⟨id|2⟩
4. Para el símbolo / se asigna ⟨/⟩
5. **tiempo** con la forma ⟨id|3⟩

El código generado por el analizador léxico se representa como:

⟨id|1⟩⟨=⟩⟨id|2⟩⟨/⟩⟨id|3⟩

2.1.2 TABLA DE SÍMBOLOS

Las tablas de símbolos "son estructuras de datos que utilizan los compiladores para guardar información acerca de las construcciones de un programa fuente" (Aho, Lam, Sethi, & Ullman, 2008)

Como se ha mencionado con anterioridad, la tabla de símbolos almacena de forma volátil[2] los lexemas encontrados en el programa fuente y la información adicional necesaria para la generación de código, como su cadena de caracteres (lexema), su tipo, su posición en el espacio de almacenamiento, y cualquier otra información relevante. Por lo general una tabla de símbolos debe soportar varias declaraciones del mismo identificador dentro del programa.

Recordando nuestro ejemplo de la sección 2.1.1, en la Tabla 1 se muestra la forma en que se almacenan los lexemas dentro de la tabla de símbolos.

	Unidad léxica	Lexema	Valor	Tipo
	=	Operador		
	/	Operador		
1	Velocidad	Identificador		
2	Distancia	Identificador		
3	Tiempo	Identificador		

Tabla 1. Tabla de símbolos representativa del ejemplo de la sección 2.1.1

[2] La tabla de símbolos es volátil, ya que su información solamente estará presente en el proceso de traducir el lenguaje entrada a código máquina.

Para que el analizador léxico sea capaz de asignar un lexema (o flujo de caracteres) a un token, es necesario especificar cada unidad léxica en un patrón determinado. La herramienta utilizada para realizar dicha especificación, es una expresión regular.

2.1.3 EXPRESIONES REGULARES

La especificación de un lenguaje de programación incluye a menudo un conjunto de reglas que define el léxico. Estas normas se denominan expresiones regulares y definen el conjunto de posibles secuencias de caracteres que se utilizan para formar fichas o lexemas.

Cada expresión regular denota un lenguaje. Para definir “las expresiones regulares (ER) sobre un vocabulario V, se utilizan las siguientes reglas:

1. ε es una ER que denota {ε}
2. Para cada a perteneciente a V, a es la ER que denota {a}
3. Si p y q son dos ER que denotan los lenguajes P y Q respectivamente, entonces:
    a. (p)|(q) es una ER que denota P∪Q
    b. (p)·(q) es una ER que denota P·Q
    c. (p)* es una ER que denota P*” (Valverde Andreu, 1989)

Por ejemplo, el grupo formado por las cadenas Handel, Händel y Haendel se describe mediante el patrón "H(a|ä|ae)ndel". Específicamente, las expresiones regulares se construyen utilizando los operadores unión, concatenación y cerradura de Kleene.

Alternación

Una barra vertical separa las alternativas. Por ejemplo, "marrón|castaño".

Cuantificación

Un cuantificador tras un caracter especifica la frecuencia con la que éste puede ocurrir. Los cuantificadores más comunes son +, ? y *:

+ El signo más indica que el caracter al que sigue debe aparecer al menos una vez. Por ejemplo, "ho+la" describe el conjunto infinito hola, hoola, hooola, hoooola,

etcétera.

? El signo de interrogación indica que el caracter al que sigue puede aparecer como mucho una vez. Por ejemplo, "ob?scuro" casa con oscuro y obscuro.

* La cerradura de Kleene, indica que el caracter al que sigue puede aparecer cero, una, o más veces. Por ejemplo, "0*42" casa con 42, 042, 0042, 00042, etcétera.

Agrupación

Los paréntesis pueden usarse para definir el ámbito y precedencia de los demás operadores. Por ejemplo, "(p|m)adre" es lo mismo que "padre|madre", y "(des)?amor" casa con amor y con desamor. Los constructores pueden combinarse libremente dentro de la misma expresión, por lo que "H(ae?|ä)ndel" equivale a "H(a|ae|ä)ndel".

2.1.4 DEFINICIONES REGULARES

Para facilitar la creación de todas las expresiones que conformarán un analizador léxico, es conveniente poner nombres a ciertas expresiones regulares y utilizarlos en otras expresiones. Lo anterior se denomina definición regular.

En la creación de un lenguaje de programación se tiene que identificar las cadenas de letras, dígitos y símbolos de operación, un ejemplo que representa esta definición es:

```
letra          →    A | B | … | Z | a | b | … | z | _

digito         →    0 | 1 | … | 9

identificador  →    letra ( letra | digito )*
```

2.1.5 AUTÓMATAS FINITOS

Una forma de generar un analizador léxico a partir de las definiciones regulares que modelan el lenguaje es con el uso de autómatas finitos, en esencia, éstos consisten en grafos como los diagramas de transición de estados, con las siguientes características:

- Los autómatas finitos son reconocedores, es decir, solo pueden representar cierto o falso en relación con cada posible cadena de entrada, esto se representa por la transición que lleva de un estado a otro.
- Los autómatas finitos pueden ser de dos tipos:
    - Los autómatas finitos no deterministas (AFN) no tiene restricciones en cuanto al número de transiciones de estados.
    - Los autómatas finitos deterministas (AFD) tienen para cada estado, y para cada símbolo del alfabeto de entrada, exactamente una línea con ese símbolo que sale de ese estado.

Tanto los autómatas finitos deterministas como no deterministas son capaces de reconocer los mismos lenguajes, mejor conocidos como lenguajes regulares.

2.1.6 AUTÓMATAS FINITOS NO DETERMINISTAS (AFN)

"Un autómata finito no determinista (AFN) consiste en:

1. Un conjunto finito de estados S
2. Un conjunto de símbolos de entrada ∑, el *alfabeto de entrada*. Suponemos que ε, que representa la cadena vacía, nunca será miembro de ∑
3. Una *función de transición* que proporciona, para cada estado y para cada símbolo en∑=∪ {ε}, un conjunto de estados siguientes.
4. Un estado s_0 de S, que se distingue como el *estado inicial*.
5. Un conjunto de estados F, un subconjunto de S, que se distingue como los estados aceptantes (o estados finales)." (Aho, Lam, Sethi, & Ullman, 2008)

Se denominan *no deterministas* debido a que el autómata puede estar en varios estados a la vez o en ninguno si se trata del conjunto vacío de estados.

Un ejemplo AFN (Autómata Finito No Determinista) es el siguiente:

$$M = (S, \Sigma, s_0, F, \Delta), donde$$
$$S = \{q0, q1, q2\}$$
$$\Sigma = \{a, b\}$$
$$s_0 = q0$$
$$F = \{q0\}$$

$\Delta = se\ define\ mediante\ la\ tabla\ siguiente$

Δ	A	b
q_0	$\{q_1\}$	$\varnothing$
q_1	$\varnothing$	$\{q_0, q_2\}$
q_2	$\{q_0\}$	$\varnothing$

Tabla 2. Tabla de transición de un AFN

La Ilustración 3 muestra la representación gráfica del autómata.

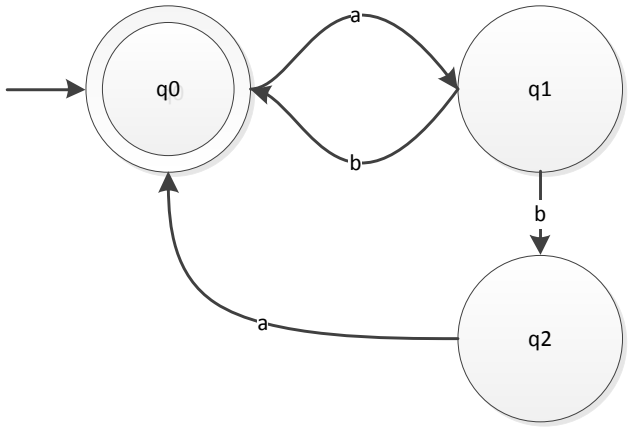


Ilustración 3 Autómata finito no determinista

2.1.7 AUTÓMATAS FINITOS DETERMINISTAS (AFD)

"Formalmente, un autómata finito determinista M es una colección de cinco elementos:

1. Un alfabeto de entrada Σ
2. Una colección finita de estados Q
3. Un estado inicial s
4. Una colección F de estados finales o de aceptación
5. Una función δ: $Q \times \Sigma \rightarrow Q$ que determina el único estado siguiente para el par (q_i, σ) correspondiente al estado actual y la entrada " (Kelly, 1995)

Lo anterior muestra que un autómata es un modelo matemático, y su función es el simular una máquina, que de acuerdo a su elemento alfabeto y conjunto de transiciones, es capaz de decidir si una cadena es reconocida o no por un lenguaje.

Un ejemplo sencillo de AFD (Autómata Finito Determinista) es el siguiente:

M = (Q, Σ, s, F, δ), donde

Q = {q_0, q_1}

Σ = {a, b}

s = q_0

F = {q_0}

δ = se define mediante la tabla siguiente

δ	A	b
q_0	q_0	q_1
q_1	q_1	q_0

Tabla 3. Tabla de transición de un AFD

Los autómatas pueden ser representados en un dibujo utilizando grafos dirigidos como el siguiente:

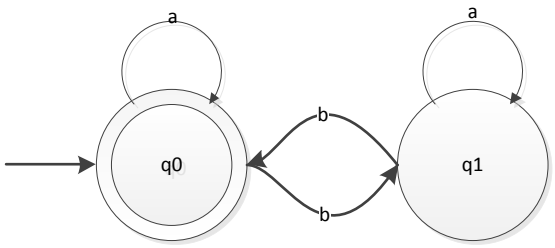


Ilustración 4 Autómata finito determinista

En el dibujo anterior se puede apreciar que aparecen los elementos del alfabeto, uno sobre cada flecha que representa cada una de las funciones de transición, el conjunto de estados está representado uno por cada círculo, el conjunto de estados finales por uno o varios círculos con línea doble y finalmente el estado inicial como un círculo al que apunta una flecha de entrada.

Nótese que en el autómata anterior cada uno de los estados tiene una flecha para cada uno de los elementos del alfabeto, ésta característica indica que el autómata finito es determinista, o sea, que para cada uno de los estados existe una regla de transición con cada uno de los elementos del alfabeto.

2.2 ANALIZADOR SINTÁCTICO

Todos los lenguajes de programación tienen reglas precisas, las cuales definen una estructura sintáctica de los programas bien formados; un programa por ejemplo, puede estar formado por procedimientos, éstos a su vez por instrucciones y una instrucción por una secuencia de palabras que tienen una expresión regular declarada.

La segunda fase del compilador es el análisis sintáctico o *parsing.* El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador léxico para crear una representación en forma de árbol que describa la estructura gramatical del flujo de tokens.

Como se muestra en la Ilustración 5, el analizador sintáctico recibe los tokens del analizador léxico y verifica la estructura de tal forma que concuerde con la definición del lenguaje fuente, durante este proceso de reconocimientos el parser genera cualquier error sintáctico en forma inteligible y esperamos que se recupere con el propósito de seguir procesando el resto del programa. Para los programas bien formados se espera que no se genere ningún error y se pueda construir el árbol de análisis sintáctico que será procesado en las siguientes fases del compilador.

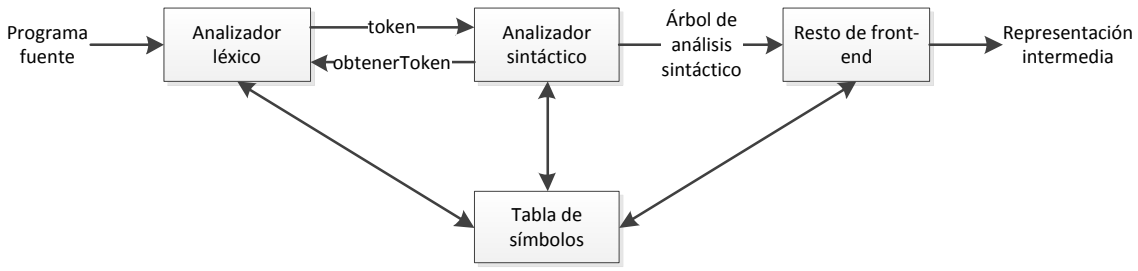


Ilustración 5 Función del analizador sintáctico

Existen tres tipos de analizadores sintácticos para las gramáticas:

- **Universales:** Son aquellos analizadores capaces de analizar cualquier tipo de gramática, (ejemplos: Cocke-Younger-Kasami y Earley).

- **Descendentes:** Se caracterizan por construir el árbol de análisis sintáctico de la parte superior (raíz) a la parte inferior (hojas).
- **Ascendentes:** Se caracterizan por construir los árboles empezando por las hojas y avanzan hasta la raíz.

2.2.1 JERARQUÍA DE LAS GRAMÁTICAS

Chomsky clasificó las gramáticas en 4 tipos, dependiendo de la forma que tienen sus reglas de producción:

Tipo 0: "Son las gramáticas más generales, tal y como se han definido formalmente y sin ninguna restricción. Una gramática de tipo 0 genera un lenguaje denominado de tipo 0" (Valverde Andreu, 1989). Ejemplo:

aAB →a

bB →aBC

Tipo 1: "También denominadas gramáticas **sensibles al contexto.** Son aquellas cuyas reglas de producción cumplen la restricción:

|β|>=|α|

Es decir, la longitud de la parte derecha de la regla es mayor o igual a la de la parte izquierda. Además la cadena α debe tener al menos un no terminal. Las gramáticas de tipo 1 generan lenguajes de tipo 1." (Valverde Andreu, 1989) Ejemplos:

0B120 →012D12D

A12 →01DB2

AF →AB12

S →0AB

Tipo 2: "También llamadas **libres de contexto.** Se caracterizan por la siguiente restricción en sus reglas de producción: cada producción debe ser de la forma A→n, donde A pertenece a VN, y n pertenece a V*. Este es el tipo de programación utilizado por los lenguajes de programación" (Valverde Andreu, 1989). Ejemplo:

Sea la gramática G=(VN,VT,P,S) donde VN=(S,A,B);VT=(0,1) y P son las siguientes producciones:

S→0B

A→0S

B→1S

S→1A

A→1AA

B→ABB

A→0

B→1

Tipo 3: "También denominadas gramáticas lineales a la derecha. Se caracterizan por que sus producciones son del tipo A→xB o bien A→x, donde A y B pertenecen a VN y ⟨x⟩pertenece a VT." (Valverde Andreu, 1989) Ejemplo. Sea

G=((⟨digito⟩),(0,1,…,9),(⟨digito⟩→0|1|…|9),⟨digito⟩)

Donde ⟨dígito⟩ es considerado como un símbolo elemental no terminal. El lenguaje L(G) generado por G es, evidentemente, el conjunto de los diez dígitos decimales. Obsérvese pues que L(G) es un conjunto finito.

2.2.2 GRAMÁTICAS LIBRES DE CONTEXTO

Una gramática formal es un objeto o modelo matemático que permite especificar un lenguaje o lengua, es decir, es el conjunto de reglas capaces de generar todas las posibilidades combinatorias de ese lenguaje, ya sea éste un lenguaje formal o un lenguaje natural.

Una gramática libre de contexto tiene cuatro componentes:

1. Un conjunto de símbolos *terminales*, a los que algunas veces se les conoce como “tokens”. Los terminales son los símbolos elementales del lenguaje definido por la gramática.
2. Un conjunto de *no terminales*, a las que algunas veces se les conoce como “variables sintácticas”. Cada no terminal representa un conjunto de cadenas o terminales.
3. Un conjunto de *producciones*, en donde cada producción consiste en un no terminal, llamado *encabezado o lado izquierdo* de la producción, una flecha y una secuencia de terminales y no terminales, llamado *cuerpo o lado derecho* de la producción. La intención intuitiva de una producción es especificar una de las formas escritas de una instrucción; si el no terminal del encabezado representa a una instrucción, entonces el cuerpo representa una forma escrita de la instrucción.
4. Una designación de uno de los no terminales como el símbolo inicial.

La gramática que representaría nuestro ejemplo empezado en el tema 2.1.1 está dado por:

```
⟨instruccion⟩→⟨expresion⟩/⟨identificador⟩

expresion→identificador=identificador

identificador→letraletradigito*

letra→abc|…|z

digito→012345678|9
```

2.2.3 ÁRBOLES

“Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación (de paternidad) que impone una estructura jerárquica sobre los nodos.” (Aho, Ullman, & Hopcroft, 1988)

Un nodo como un elemento de una lista se puede representar por cualquier tipo. Regularmente se representa como un nodo por medio de una letra, una cadena de caracteres o dígitos. Formalmente, un árbol se puede definir de manera recursiva como:

1. Un solo nodo es, por sí mismo, un árbol. Ese nodo es también la raíz de dicho árbol.

2. Supóngase que n es un nodo y que $A_1, A_2, \ldots, A_k$ son árboles con raíces $n_1, n_2, \ldots, n_k$ respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos $n_1, n_2, \ldots, n_k$. En dicho árbol, n es la raíz y $A_1, A_2, \ldots, A_k$ son los subárboles de la raíz. Los nodos $n_1, n_2, \ldots, n_k$ reciben el nombre de hijos del nodo n.

2.2.4 ORDEN PREVIO, ORDEN POSTERIOR Y ORDEN SIMÉTRICO

Existen tres formas de poder ordenar los nodos en los árboles, los métodos de ordenamiento se llaman orden previo (preorder), orden simétrico (inorder) y orden posterior (postorder), y se definen a continuación:

- "Si un árbol A es nulo, entonces la lista vacía es el listado de los nodos de A en los órdenes previo, simétrico y posterior.

- Si A contiene un solo nodo, entonces ese nodo constituye el listado de los nodos de A en los órdenes previo, simétrico y posterior.

1. El listado en orden previo (o recorrido en orden previo) de los nodos de A está formado por la raíz de A, seguida de los nodos de A_1, en orden previo, luego por los nodos de A_2, en orden previo y así sucesivamente hasta los nodos de A_k, en orden previo.

2. El listado en orden simétrico de los nodos de A está constituido por los nodos de A_1, en orden simétrico, seguidos de n y luego por los nodos de $A_2, \ldots A_k$ con cada grupo de nodos en orden simétrico.

3. El listado en orden posterior de los nodos de A tiene los nodos de A_1 en orden posterior, luego los nodos de A_2 en orden posterior y así sucesivamente hasta los nodos de A_k en orden posterior y por último la raíz n." (Aho, Ullman, & Hopcroft, 1988)

2.2.5 ÁRBOLES SINTÁCTICOS

Un árbol de análisis sintáctico muestra, en forma de gráfica, la manera en que el símbolo inicial de una gramática deriva a una cadena en el lenguaje. Retomando nuestro ejemplo formulado en la sección 2.1.1 TOKENS Y LEXEMAS, en donde se propuso la siguiente instrucción:

```
Velocidad = distancia / tiempo
```

El árbol de análisis sintáctico, Ilustración 6, se forma a partir del token ⟨/⟩, del cual se produce un hijo izquierdo con el nodo ⟨id|2⟩ y un hijo derecho para ⟨id|3⟩. El nodo ⟨id|2⟩ representa el identificador de **distancia**, mientras que el ⟨id|3⟩ al del **tiempo**. El nodo etiquetado como ⟨/⟩ hace explícito que primero debemos realizar la operación de división para el valor de distancia y tiempo. Finalmente el nodo etiquetado como ⟨=⟩ o nodo raíz indica que se tiene que asignar el valor de dicha división (nodo derecho) al token ⟨id|1⟩ como nodo izquierdo que representa la **velocidad**.

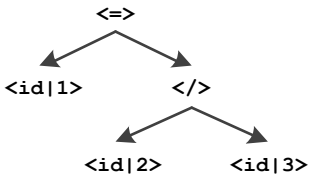


Ilustración 6 Árbol de análisis sintáctico

Las fases siguientes del compilador utilizan la estructura gramatical para ayudar a analizar el programa fuente y generar el programa destino.

De manera formal, dada una gramática libre de contexto, "un árbol de análisis sintáctico de acuerdo con la gramática es un árbol con las siguientes propiedades:

1. La raíz se etiqueta con el símbolo inicial
2. Cada hoja se etiqueta con un terminal, o con ε.
3. Cada nodo interior se etiqueta con un no terminal.
4. Si A es el no terminal que etiqueta a cierto nodo interior, y $X_1, X_2, \ldots, X_n$ son las etiquetas de los hijos de ese nodo de izquierda a derecha, entonces debe haber una producción $A \rightarrow X_1\ X_2 \ldots . X_n$. Aquí, cada una de las etiquetas $X_1, X_2, \ldots, X_n$ representa a un símbolo que puede ser o no un

terminal. Como un caso especial, si A→ε es una producción entonces un nodo etiquetado como A puede tener un solo hijo, etiquetado como ε" (Aho, Lam, Sethi, & Ullman, 2008)

2.2.6 ANÁLISIS SINTÁCTICO DESCENDENTE

El análisis sintáctico descendente puede verse como el problema de construir un árbol de análisis sintácticos para la cadena de entrada, partiendo desde la raíz y creando los nodos del árbol de análisis sintáctico en preorden. De manera equivalente, podemos construir el análisis sintáctico descendente como la búsqueda de una derivación por la izquierda para una cadena de entrada.

En cada caso de un análisis sintáctico descendente, el problema clave es el de determinar la producción que debe aplicarse para un no terminal, es decir, cuando se selecciona una producción X, el resto del proceso consiste en relacionar los símbolos terminales en el cuerpo de la producción con la cadena de entrada.

En base a nuestro ejemplo de la instrucción:

```
Velocidad = distancia / tiempo
```

La construcción del árbol sintáctico descendente está representada en la Ilustración 7.

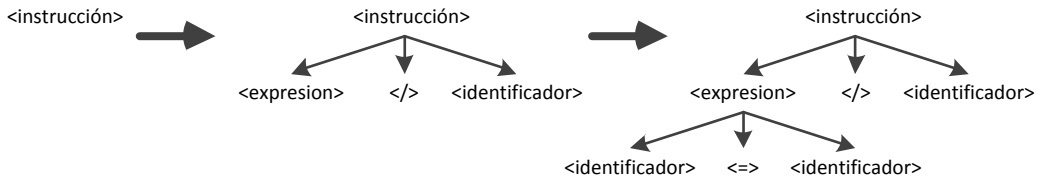


Ilustración 7 Análisis sintáctico descendente para ID =ID / ID

2.2.7 ANÁLISIS SINTÁCTICO ASCENDENTE

Un análisis sintáctico ascendente corresponde a la construcción de un árbol de análisis sintáctico para una cadena de entrada que empieza en las hojas y avanza hacia la raíz.

La representación en forma de árbol de análisis sintáctico ascendente para la instrucción del tema anterior se presenta en la Ilustración 8

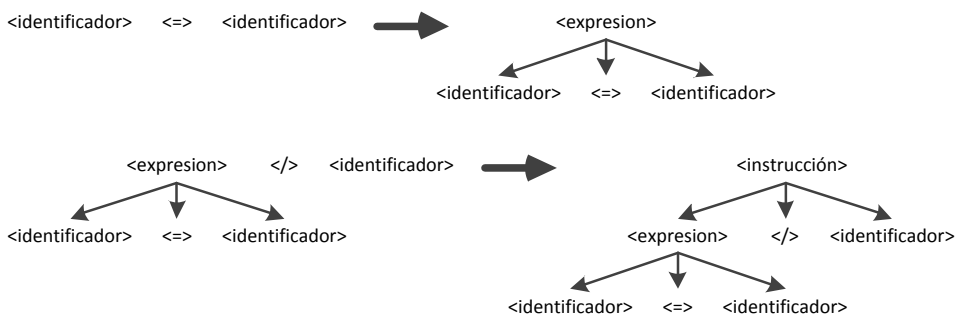


Ilustración 8 Análisis sintáctico ascendente para ID = ID / ID

2.3 ANÁLISIS SEMÁNTICO

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores léxico 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.

En compiladores de dos o más pasos, el análisis semántico se realiza independientemente de la generación de código, pasándose información a través de un archivo intermedio, que normalmente contiene información sobre el árbol sintáctico en forma lineal (para facilitar su manejo y hacer posible su almacenamiento en memoria auxiliar).

En cualquier caso, las rutinas semánticas suelen hacer uso de una pila (la pila semántica) que contiene la información semántica asociada a los operandos (y a veces a los operadores) en forma de registros semánticos.

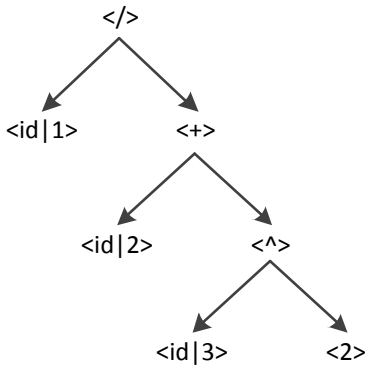
2.3.1 PROPAGACIÓN DE ATRIBUTOS

Considerando la siguiente expresión:

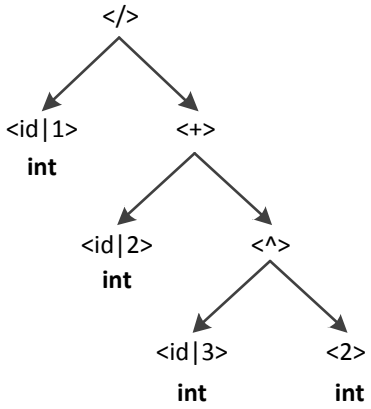
```
int a,b,c:

a/b+c^2
```

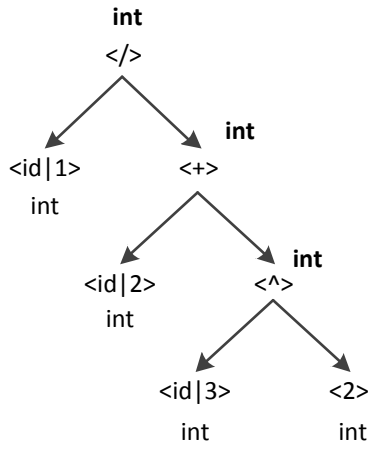
El árbol sintáctico generado es:



De la instrucción declarativa, la tabla de símbolos y el analizador léxico obtenemos los atributos de los operandos:



Propagando los atributos obtenemos:



Como se puede observar los tipos de datos de los operadores correspondieron con los tipos de dato de los operando confirmando que la instrucción escrita es correcta.

2.3.2 GRAMÁTICA DE ATRIBUTOS

Es una extensión de la notación de Backus que consiste en introducir en las reglas sintácticas ciertos símbolos adicionales no sintácticos (símbolos de acción) que, en definitiva, se reducen a llamadas implícitas o explícitas a rutinas semánticas.

Por ejemplo: sea la gramática simplificada que analiza las instrucciones de asignación:

```
sentencia→id#Pid=expresion#RAs

expresion→expresion+T#RS|T

T→id#Pid|c#Pc
```

Se observará que hemos hecho uso de cuatro símbolos de acción:

- `#Pid:` PUSH a la pila semántica el registro asociado al identificador.
- `#Pc:` PUSH a la pila semántica el registro asociado a la constante.
- `#RS:` Realizar suma: POP los dos registros superiores de la pila; comprobar que es posible sumarlos; realizar la suma (mediante una llamada al generador de código) o generar representación intermedia (si es una compilación en dos o más pasos); PUSH registro semántico del resultado en la pila semántica.

- `#RAs:` Realizar asignación: POP los dos registros superiores de la pila; comprobar que es posible realizar la asignación; realizarla (mediante una llamada al generador de código) o generar representación intermedia (si es una compilación en dos o más pasos).

2.4 GENERACIÓN DE CÓDIGO INTERMEDIO

Generalmente un compilador cuenta con un front-end encargado de realizar las fases de compilación antes vistas y la generación de un código intermedio, a partir del cual un back-end genera el código destino.

Existen dos representaciones intermedias principales:

- Notación sufija
- Cuádruplas

Los operadores diádicos (o binarios) pueden especificarse mediante tres notaciones principales:

- Prefija: el operador diádico es analizado antes que sus operandos.
- Infija: el operador diádico es analizado entre sus operandos.
- Sufija: el operador diádico es analizado después que sus operandos.

En los lenguajes de programación clásicos, los operadores diádicos se representan usualmente en notación infija. La notación prefija permite al operador influir sobre la manera en que se procesan sus operandos, pero a cambio suele exigir mucha más memoria. La sufija no permite esa influencia, pero es óptima en proceso de memoria y permite eliminar el procesado de los paréntesis.

Los operadores monádicos sólo pueden presentarse en notación prefija o sufija.

Además, un árbol sintáctico puede representarse en forma de tuplas de *n* elementos, de la forma `(operador, operando-1, ..., operando-k, nombre)`. Las tuplas pueden tener longitud variable o fija (con operandos nulos). Las más típicas son las cuádruplas, aunque éstas pueden representarse también en forma de tripletes.

2.4.1 NOTACIÓN SUFIJA

Llamada también postfija o polaca inversa, se usa para representar expresiones sin necesidad de paréntesis.

Ejemplos:

Infija	Sufija
a*b	ab*
a*(b+c/d)	abcd/+*
a*b+c*d	ab*cd*+

Tabla 4. Ejemplo de la notación sufija

Los identificadores aparecen en el mismo orden. Los operadores en el de evaluación (de izquierda a derecha).

Problema: operadores monádicos (unarios). O bien se transforman en diádicos (binarios) o se cambia el símbolo.

Ejemplo: -a se convierte en 0-a o en @a

```
a*(-b+c/d)  →  ab@cd/+*
```

Existen dos problemas principales:

- Construir la notación sufija a partir de la infija.
- Analizar la notación sufija en el segundo paso de la compilación.

2.4.2 RUTINA SEMÁNTICA PARA TRANSFORMAR DE INFIJO A SUFIJO

Se utiliza una pila donde se genera la salida, inicialmente vacía. Las acciones semánticas asociadas a las reglas son:

```
E ::= E + T          Push +

E ::= E - T          Push -

E ::= T

T ::= T * F          Push *

T ::= T / F          Push /

T ::= F

F ::= i              Push i

F ::= (E)

F ::= - F            Push @
```

2.4.3 ANÁLISIS DE LA NOTACIÓN SUFIJA

La gramática completa que permite analizar la notación sufija es:

⟨operador⟩→id|cte|⟨operador⟩⟨operador⟩⟨diadico⟩|⟨operador⟩⟨monodico⟩

⟨diadico⟩→+|-|*|/

⟨monodico⟩→@|…

Algoritmo de evaluación de una expresión en notación sufija que utiliza una pila:

- Si el próximo símbolo es un identificador, se pasa a la pila. Corresponde a la aplicación de la regla

⟨operando⟩ → id

- Si el próximo símbolo es una constante, se pasa a la pila. Corresponde a la aplicación de la regla

⟨operando⟩→cte

- Si el próximo símbolo es un operador diádico, se aplica el operador a los dos operandos situados en lo alto de la pila y se sustituyen éstos por el resultado de la operación. Corresponde a la aplicación de la regla

⟨operador⟩→⟨operador⟩⟨operador⟩⟨diadico⟩

- Si el próximo símbolo es un operador monádico, se aplica el operador al operando situado en lo alto de la pila y se sustituye éste por el resultado de la operación. Corresponde a la aplicación de la regla.

⟨operador⟩→⟨operador⟩⟨monodico⟩

2.4.4 EXTENSIÓN DE LA NOTACIÓN SUFIJA A OTROS OPERADORES

- La asignación, teniendo en cuenta que podemos no querer el valor resultante. Además, no interesa tener en la pila el valor del identificador izquierdo, sino su dirección.

```
a:=b*c+d      abc*d+:=
```

- La transferencia (GOTO).

```
GOTO L        L TR
```

- La instrucción condicional

```
if p then inst1 else inst2
```

se convierte en

```
p L1 TRZ inst1 L2 TR inst2
```

```
L1:   L2:
```

- Subíndices:

```
a[exp1; exp2; ...; expn]
```

se convierte en

```
a exp1 exp2 ...expn SUBIN-n
```

2.4.5 CUÁDRUPLAS

Una operación diádica se puede representar mediante la cuádrupla

```
(<Operador>,<Operando1>,<Operando2>,<Resultado>)
```

Ejemplo:

```
(*,A,B,T)
```

Una expresión se puede representar mediante un conjunto de cuádruplas. Ejemplo: la expresión `a*b+c*d` equivale a:

```
(*,a,b,t1)

(*,c,d,t2)

(+,t1,t2,t3)
```

Ejemplo: la expresión `c:=a[i;b[j]]` equivale a:

```
(*,i,d1,t1)

(+,t1,b[j],t2)
```

```
(:=,a[t2],,c)
```

2.4.6 TRIPLETES

No se pone el resultado, se sustituye por referencias a tripletes. Por ejemplo: la expresión `a*b+c*d` equivale a:

```
(1) (*,a,b)

(2) (*,c,d)

(3) (+,(1),(2))
```

mientras que $a * b + 1$ equivale a:

```
(1) (*,a,b)

(2) (*,(1),1)
```

Tripletes indirectos: se numeran arbitrariamente los tripletes y se da el orden de ejecución. Ejemplo, sean las instrucciones:

```
a:= b*c

b:= b*c
```

Equivalen a los tripletes

```
(1) (*,b,c)

(2) (:=,(1),a)

(3) (:=,(1),b)
```

y el orden de ejecución es `(1),(2),(1),(3)`. Esta forma es útil para preparar la optimización de código. Si hay que alterar el orden de las operaciones o eliminar alguna, es más fácil hacerlo ahí.

2.5 GENERACIÓN DE CÓDIGO

El código generado por un compilador puede ser de uno de los tipos siguientes:

- Código simbólico
- Código relocalizable. Es el sistema más flexible y el más común en compiladores comerciales.
- Código absoluto.

Usaremos como ejemplo la traducción a código simbólico (más legible).

- Existen dos formas de realizar la generación de código:
- En un solo paso: integrada con el análisis semántico.

En dos o más pasos: el analizador semántico genera un código intermedio (cuádruplas, notación sufija), a partir del cual se realiza la generación del código definitivo como un paso independiente.

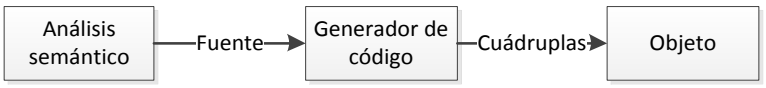


Ilustración 9 Analizador semántico

La primera posibilidad es más rápida, pero más compleja.

CAPÍTULO III PSEUDOCÓDIGO

Lo más importante en la realización del compilador es la definición del lenguaje, ya que la correcta ejecución de un programa dependerá de las estructuras secuenciales, de control, repetitivas, los tipos de datos creados, incluso la notación que se utilizará para los comentarios. En este apartado se detallará la estructura del lenguaje definido para el compilador.

Los lenguajes de programación actuales de alto nivel permiten declarar métodos (subrutinas) y variables o atributos, prácticamente en cualquier parte del programa principal, siguiendo las reglas sintácticas definidas de cada uno de éstos lenguajes, por ejemplo en Java se pueden declarar métodos y atributos sin importar el orden en el que estén situados con respecto a otras subrutinas (por ejemplo el public static void main).

A diferencia de los lenguajes actuales, el compilador a construir presenta rigidez y poca flexibilidad en cuanto al orden de subrutinas, declaración de variables y declaración del programa principal; lo anterior se debe al hecho de que se busca que la estructura del lenguaje se apegue a un pseudocódigo siguiendo las bases de la programación estructurada.

3.1 ESTRUCTURA GENERAL DEL PSEUDOCÓDIGO

La estructura de todo programa realizado en nuestro lenguaje se muestra en la Ilustración 10.

El programa inicia con la declaración de un nombre para el programa, éste nombre corresponde a lo que se le conoce como un identificador, éste elemento será explicado con mayor detalle en el capítulo de construcción del compilador en la fase de análisis léxico, por el momento se especifica que dicho nombre sólo puede comenzar con letras y puede contener letras o números.

Lo que se busca con éste compilador es tener código que se parezca lo más posible al pseudocódigo, por lo cual es necesario indicar dónde comienza y termina el programa, la palabra reservada PROGRAMA indica que el código a compilar comienza ahí y las palabras INICIO y FIN denotan el comienzo y término de la rutina principal; es importante notar que las palabras reservadas están en mayúsculas, ya que existen homónimos a éstas en minúsculas.

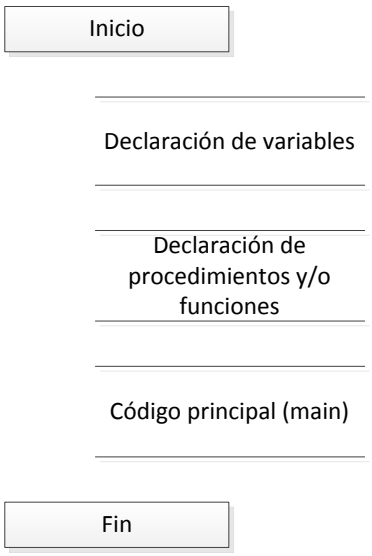


Ilustración 10 Estructura general del lenguaje

Entre las palabras PROGRAMA e INICIO es donde se declararán variables globales y las subrutinas, dichas declaraciones deben seguir ese preciso orden, es decir, no se pueden declarar primero subrutinas y a continuación variables.

Un ejemplo valido de un programa con lo definido anteriormente sería

```
ejemploPrograma PROGRAMA

// Declaración de variables locales

INICIO

//Cuerpo de rutina

FIN
```

3.2 TIPOS DE DATOS

La declaración de variables dentro de un programa permite crear código que presente diferentes resultados de acuerdo a los datos de entrada del usuario, y cada variable tiene un tipo de dato asociado, lo anterior genera la existencia de restricciones en los datos, como por ejemplo qué valores pueden tomar y qué operaciones se pueden realizar.

Los tipos de dato soportados están basados en aquellos de Java, principalmente en su rango y en tamaño que utilizan de la memoria RAM, debido a que el desarrollo (explicado más adelante) está hecho sobre Java.

La lista de tipos primitivos de dato se muestra a continuación:

- **entero.** Corresponde al tipo int de Java, con valor mínimo de -2,147,483,648 y máximo de 2,147,483,647 inclusive.
- **flot.** Corresponde al tipo float de Java, el cual a su vez corresponde al estándar IEEE 754 floating point.
- **texto.** Corresponde a un objeto de la clase String en Java en caso de ser más de un caracter, y al tipo char de Java en caso de ser un solo caracter.
- **bool.** Corresponde al tipo boolean de java, el cual denota valores de cierto y falso.

3.3 OPERADORES

Los operadores son símbolos que realizan operaciones específicas sobre uno o dos operandos, y finalmente devuelven un resultado. En el pseudocódigo se mantiene la precedencia de operadores, lo cual significa que los operadores con mayor precedencia se ejecutan antes que aquellos de menor precedencia siempre y cuando no haya paréntesis. A continuación se muestran los operadores que se utilizan en el pseudocódigo

Operadores	Precedencia
Unarios	`expr++ expr-- !`
Multiplicativos	`* / .mod.`
Aditivos	`+ - .`
Relacionales	`< > <= >=`
Igualdad	`== !=`
Y lógico	`.y.`
O lógico	`.o.`
Asignación	`= += -= *= /= .mod.=`

Tabla 5. Operadores del pseudocódigo

3.4 ESTRUCTURAS DE DATOS

Un arreglo es un contenedor que maneja un número de valores de un solo tipo de dato, el tamaño o longitud del arreglo se indica cuando se crea el mismo. Cada ítem del arreglo se llama *elemento* y es posible acceder a cada elemento mediante un *índice*. El primer índice de un arreglo es 0 y el último es igual a la longitud del arreglo menos uno.

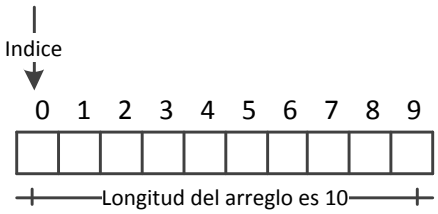


Ilustración 11 Representación gráfica del arreglo

Para la creación de un arreglo en el pseudocódigo se utiliza la siguiente forma:

```
[]tipoDato identificador[númeroEntero] < = {exp1,exp2,exp3…} >
```

donde;

- tipoDato, representa la palabra reservada para algún tipo de dato primitivo
- Identificador, el nombre que tendrá el arreglo
- númeroEntero, el tamaño que tendrá el arreglo
- `= {exp1, exp2, exp3…}`, indica los valores con los cuales serán inicializados cada uno de los elementos del arreglo

ejemplo

```
//Declaración de un nuevo arreglo

   []entero muestra[30]

//Declaración de un arreglo inializando sus elementos

   []entero muestra = {1,2,3}
```

```
//Acceso a un elemento del arreglo

   muestra[1] = 5
```

3.5 COMENTARIOS

Un comentario es texto que permite añadir caracteres que no son tomados en cuenta por el compilador del pseudocódigo, es necesario contar con la posibilidad de poner comentarios ya que permiten colocar explicaciones del código, poner notas que se consideren importantes, etc. Los comentarios se clasifican dependiendo de la cantidad de líneas que contenga.

- Una línea .- se utiliza la secuencia de caracteres `//`

```
//Comentario en una línea
```

- Varias líneas .- se utiliza `/*` para iniciar el comentario y `*/` para terminarlo

```
/*
 * Es posible escribir un comentario
 * en varias líneas.
 */
```

3.6 BLOQUES DE CÓDIGO

La ejecución de sentencias siempre se da de forma secuencial ejecutando instrucción por instrucción. Un bloque de código es aquel conjunto de sentencias que se encuentran delimitadas por las palabras reservadas `inicio` y `fin`.

La utilización de estos delimitadores de bloques son de gran importancia dentro de las estructuras de control (las cuales se detallan más adelante), debido a que indican y delimitan el conjunto de líneas de código que pertenecen a cada uno de los bloques de ejecución de las sentencias de control.

La omisión de las palabras reservadas `inicio` y `fin` resulta en la incorrecta obtención de los resultados del pseudocódigo; al igual que en los lenguajes de programación de alto nivel (Java, C, C++, Pascal, etc.) es necesario indicar que sentencias pertenecen a un bloque de ejecución y cuáles no.

Para ejemplificar lo anterior, se utilizará una sentencia `si` (la explicación de ésta y otras sentencias de control se encuentra en el siguiente apartado). La ejecución del código

```
si (x > -1) entonces
    sentencia 1
    sentencia 2
    sentencia 3
```

genera un resultado diferente a

```
si (x > -1) entonces
    inicio
        sentencia 1
        sentencia 2
        sentencia 3
    fin
```

ya que el primer ejemplo, indica que si la valuación de `(x > −1)` es cierta, se ejecutará solamente `sentencia 1`, mientras que en el segundo ejemplo se ejecutaría el bloque delimitado por `inicio` y `fin`, lo cual es `sentencia 1, sentencia 2, sentencia 3.`

Cuando el bloque de código a delimitar consiste de una sentencia, las palabras reservadas son opcionales, con lo cual la ejecución de

```
si (x > -1) entonces
```

```
    sentencia 1
```

es igual a la ejecución de

```
si (x > -1) entonces
    inicio
        sentencia 1
    fin
```

3.7 ESTRUCTURAS DE CONTROL

Las líneas de código o sentencias que se encuentran en un archivo, se ejecutan de arriba hacia abajo de acuerdo a su orden de aparición, sin embargo las *estructuras de control* rompen o cambian el flujo de ejecución utilizando bucles y tomando decisiones, dando al programa (o pseudocódigo en éste caso) la habilidad de ejecutar condicionalmente bloques de código.

Sentencias selectivas

La sentencia `si-entonces` indica al programa ejecutar una cierta porción de código sólo si una expresión en particular devuelve `cierto`.

```
si (x > -1) entonces
    numeroPositivo = cierto
```

En el bloque anterior la variable booleana `numeroPositivo` es verdadera si la variable a evaluar, `x` en este caso, es mayor a -1. Es importante destacar que todos los bloques de ejecución deben comenzar con `inicio` y terminar con `fin` en minúsculas, de acuerdo a lo explicado en el apartado Bloques de Código, cuando exista más de una sentencia, y siendo éstas palabras opcionales cuando haya una sola instrucción. El ejemplo anterior produce el mismo resultado si se escribiese así:

```
si (x > -1) entonces
   inicio
          numeroPositivo = cierto
   falso
```

La sentencia `si-entonces-sino` proporciona un camino alterno cuando la evaluación de la sentencia `si-entonces` devuelve como resultado `falso`.

```
si (x > -1) entonces
   numeroPositivo = cierto
sino
   numeroPositivo = falso
```

En el bloque anterior la variable `numeroPositivo` toma el valor cierto cuando `x` es mayor a -1, pero en caso contrario omite el bloque de código que se encuentra después de `entonces`, pasando a ejecutar el bloque que se encuentra después de la palabra reservada `sino`.

Además de las dos sentencias anteriores, existe una sentencia que permite *n* número de posibilidades de ejecución, dicha sentencia es `en_caso` y funciona con el tipo de dato entero.

El siguiente fragmento de código permite almacenar el nombre de un mes del año en la variable nombreMes.

```
texto nombreMes
entero num = 3
en_caso (num)
inicio
   opc 1: inicio
```

```
                nombreMes = “Enero”
            fin
    opc 2: inicio
                nombreMes = “Febrero”
            fin
    opc 3: inicio
                nombreMes = “Marzo”
            fin
    opc 4: inicio
                nombreMes = “Abril”
            fin
    otro:  inicio
                nombreMes = “Otro”
            fin
fin
```

Es importante notar que cada opción en la sentencia `en_caso` debe contar con un bloque delimitado entre `inicio` y `fin`. Otra característica importante es que se puede utilizar el mismo código de una opción para que se ejecute en varias opciones.

```
entero num = 2, numDias = 0, año = 2000
en_caso (num)
inicio
```

```
   opc 1:

   opc 3:

   opc 5:

   opc 7:

opc 8:

   opc 10:

   opc 12: inicio

                numDias = 31

           fin

   opc 4:

   opc 6:

opc 9:

   opc 11: inicio

                numDias = 30

           fin

   opc 2: inicio

                si ( ((año .mod. 4 == 0) .y. !(año .mod. 100 == 0)) .o.
(año .mod. 400 == 0) ) entonces

numDias = 29

sino

   numDias = 28
```

```
            fin
      otro:  inicio
                  numDias = 0
            fin
   fin
```

Sentencias repetitivas

La sentencia `mientras` ejecuta un bloque de código mientras una condición específica devuelva `cierto` al ser evaluada.

```
entero i = 1
mientras (i < 10) hacer
   i++
```

Al igual que la sentencia `si-entonces`, los delimitadores `inicio` y `fin` son necesarios solamente cuando el bloque de código contiene más de una instrucción.

Existe una estructura llamada `repetir-hasta` la cual es muy parecida a `mientras`, la diferencia radica en que la estructura `repetir-hasta` evalúa la expresión al final del ciclo y `mientras` la evalúa al principio, lo anterior causa que al utilizar la estructura `repetir-hasta`, ejecuta el bloque de código al menos una vez. Se utiliza el ejemplo anterior implementando la estructura `repetir-hasta`:

```
entero i = 1
repetir
   i++
hasta(i > 9)
```

La última de las instrucciones de control, es la sentencia `para`, la cual provee de un mecanismo que permite iterar entre un rango de valores. La forma genérica de un ciclo `para` se muestra a continuación:

```
para(inicialización; termino; incremento)
   <inicio>
         sentencia(s)
   <fin>
```

Cuando se usa un ciclo para, es importante tener siempre en mente:

- *inicialización* es donde se inicia el ciclo, se ejecuta solamente una vez
- *termino* es la expresión que indica cuando parará el ciclo, la expresión deberá ser `falso` para detener el ciclo
- *incremento* es una expresión que se invoca siempre después de cada iteración y es el lugar en donde se debe incrementar o decrementar un valor

El siguiente ejemplo incrementa una variable numérica:

```
entero miNumero = 0

para(entero i = 1; i <= 11; i++)
   miNumero = i
```

Es importante hacer notar que se declara una variable en la expresión de inicialización, ésta variable está disponible y es accesible dentro de todo el código en donde se ejecuta el ciclo `para`, así como también puede ser usada en las expresiones de termino e incremento. Si la variable que controla el ciclo `para` no es usada fuera del bucle, es muy recomendable declarar la variable en la expresión de inicialización.

3.8 SUBRUTINAS

La secuencia de ejecución del código de un programa se puede ver interrumpida por sentencias de control, las cuales alteran el orden en que el código es ejecutado; otra forma de hacer “brincos” o no seguir la ejecución lineal de una serie de líneas de código, es utilizando subrutinas.

Una subrutina es una porción o bloque de código dentro de un programa más grande, cuya función es ejecutar una tarea específica; una subrutina se codifica en una forma tal que pueda ser *llamada* o iniciada muchas veces y/o de diferentes partes dentro del programa (incluso ser llamada desde otras subrutinas), y después volver al punto de ejecución de donde fue invocada para continuar con las siguientes líneas de código.

El pseudocódigo aquí descrito soporta lo que a veces se puede llamar *procedimientos* y *funciones* (ésta terminología la utiliza Pascal por ejemplo). La diferencia entre una *función* y *procedimiento* es que en la *función* devuelve un resultado al término de su ejecución, mientras que un *procedimiento* no devuelve resultado alguno.

La sintaxis para la declaración de un procedimiento se muestra a continuación

```
procedimiento nombreSubrutina (<parámetros>)
inicio
          <cuerpo de la subrutina>
   fin
```

Y la sintaxis para la declaración de una función es

```
procedimiento tipo nombreSubrutina (<parámetros>)
inicio
          <cuerpo de la subrutina>
          regresar <expresion>
   fin
```

En ambos casos corresponde *nombreSubrutina* al nombre que tendrá la subrutina, y es importante destacar que es necesario tener las palabras `inicio` y `fin` para delimitar el cuerpo de la subrutina. En el caso de las funciones, *tipo* corresponde a alguno de los tipos soportados por el pseudocódigo (explicados en éste capítulo en el tema Tipos de Dato).

Ejemplos correctos de funciones y procedimientos se muestran en seguida

```
procedimiento entero hacerSuma (entero x, entero y)
   inicio
         regresar x + y
   fin
procedimiento suma ()
   inicio
         entero a, b, res
         res = hacerSuma(a,b)
   fin
```

3.9 OPERACIONES DE ENTRADA/SALIDA DE DATOS

La capacidad que se brinda al usuario de poder ejecutar un programa con distintos datos de entrada es de gran importancia, inclusive desde la perspectiva del usuario, es toda la interacción que éste tiene con los programas que se ejecutan, ya que dependiendo de los datos de entrada que se tengan, se obtendrán resultados diferentes.

El pseudocódigo tiene palabras reservadas para la lectura de datos del usuario y para la escritura en pantalla.

Para recibir datos de entrada del usuario se utiliza la palabra reservada `ENTRADA`, cuya sintaxis es:

```
ENTRADA var1 <, var2, var3, ..., varn>
```

Cuando se utiliza `ENTRADA`, es necesario indicar las variables que tomarán los valores de entrada, cuando se define una lista, el valor que toma la primera variable en la lista corresponde al primer dato que el usuario indicó, y la última variable toma el valor del último dato indicado por el usuario.

Si se ejecuta la instrucción

```
ENTRADA x, y, z
```

Y en la pantalla de solicitud el usuario introduce

```
$  1

$  2

$  3
```

El valor que tomarían las variables es

```
x = 1

y = 2

z = 3
```

Para mostrar datos al usuario, se utiliza la palabra reservada `IMPRIME`, cuya sintaxis es como sigue

```
IMPRIME exp1 <, exp2, exp3, ..., expn>
```

Cuando se utiliza `IMPRIME`, se muestran en la salida estándar, todas las expresiones que fueron indicadas en la lista se parada por comas.

Existe una constante definida de tipo texto `NL`, la cual tiene como función la inserción de nueva línea en la salida estándar, lo cual es equivalente a *linefeed* en Unix (\n) y retorno de carro + nueva línea en Windows (\r\n).

Ejemplos de uso de IMPRIME se muestra a continuación

```
res = 4 + 2

IMPRIME “El resultado es:”, res
```

la salida generada sería

```
$  El resultado es: 6
```

Otro ejemplo del uso de IMPRIME:

```
IMPRIME “Ejemplo de IMPRIME”, NL, NL, NL, “¡Hola Mundo!”
```

la salida generada sería

```
$  Ejemplo de IMPRIME

$

$

$  ¡Hola Mundo!
```

CAPÍTULO IV DESARROLLO MATEMÁTICO

La definición del pseudocódigo se basó en la experiencia de aprender a programar y la de utilizar un lenguaje de alto nivel. Sin embargo para poder desarrollar un compilador con éste pseudocódigo es necesario aplicar los fundamentos matemáticos de la computación para crear las reglas lógicas que utilizará la computadora al analizarlo.

4.1 ALFABETO

La creación de un lenguaje comienza por definir los símbolos que lo representan, este conjunto de símbolos se denomina alfabeto, a continuación se muestra como está compuesto el alfabeto del pseudocódigo:

- Todas las letras del alfabeto español desde la **a** hasta la **z**, tanto en minúsculas como en mayúsculas **A** hasta **Z**.
- El conjunto de números enteros, es decir, **0** a **9** y sus combinaciones.
- Los símbolos permitidos son: **+ - * / = . ! > < ( ) ;**

El alfabeto se representa por sigma de la siguiente forma:

$$\Sigma = \{a,\ldots,z,A,\ldots,Z,0,\ldots,9,+,-,*,/,=,.,!,>,<,(,),;\}$$

4.2 EXPRESIONES REGULARES

Para determinar que palabras o cadenas pertenecen al lenguaje es necesario formar las expresiones regulares que determinen dichas reglas.

$$Identificador = \langle letra \rangle\ (\langle letra \rangle | \langle digito \rangle)^*$$

$$númeroEntero = \langle digito \rangle^+$$

$$flotante = \langle digito \rangle^+ \ | \ .\langle digito \rangle^+ \ | \ \langle digito \rangle^+.\langle digito \rangle^*$$

$$operadores = .mod. | \ .y. | .o.$$

```
digito = 0|1|2|3|4|5|6|7|8|9

letra = a|b|c|d|e|f|g|h|i|j|k|l|m|n|ñ|o|p|q|r|s|t|u|v|w|x|y|z

símboles = +|-|*|/|=|.|!|>|<|(|)|{|}|;
```

4.3 AUTÓMATAS

Una forma de generar un analizador léxico a partir de las definiciones regulares que modelan el lenguaje es haciendo uso de autómatas finitos, de acuerdo a las expresiones regulares definidas en el tema anterior, a continuación se presentan los autómatas finitos correspondientes.

Identificador = (Q, Σ, s, F, δ), donde:

Q = {q_0, q_1}

Σ = {digito, letra}

s = q_0

F = {q_1}

δ = se define mediante la tabla siguiente

Δ	Letra	Digito
q_0	{q_1}	∅
q_1	{q_1}	{q_1}

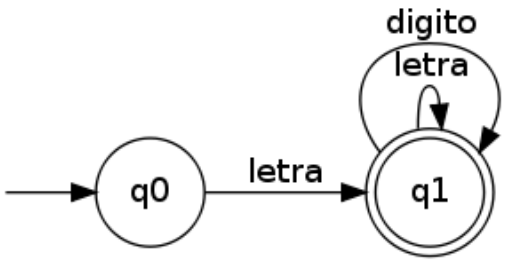


Ilustración 12 Autómata finito para un identificador

Operador = (Q, Σ, s, F, δ), donde:

Q = {q0, q1, q2, q3}

Σ = {.,y,o}

s = q0

F = {q3}

δ = se define mediante la tabla siguiente

Δ	.	y	O
q_0	$\{q_1\}$	∅	∅
q_1	∅	$\{q_2\}$	$\{q_2\}$
q_2	$\{q_3\}$	∅	∅
q_3	∅	∅	∅

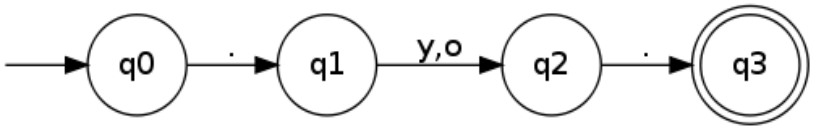


Ilustración 13 Autómata finito para un operador

Entero = (Q, Σ, s, F, δ), donde:

Q = {q_0, q_1}

Σ = {digito}

s = q_0

F = {q_1}

δ = se define mediante la tabla siguiente

Δ	Digito
q_0	$\{q_1\}$
q_1	$\{q_1\}$

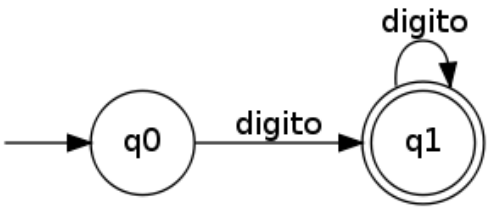


Ilustración 14 Autómata finito para un número entero

Operador = (Q, Σ, s, F, δ), donde:

Q = {q_0, q_1, q_2, q_3}

Σ = {.,digito}

s = q_0

F = {q_1}

δ = se define mediante la tabla siguiente

Δ	.	digito
q_0	{q_2}	{q_1}
q_1	{q_3}	{q_1}
q_2	∅	{q_1}
q_3	∅	{q_1}

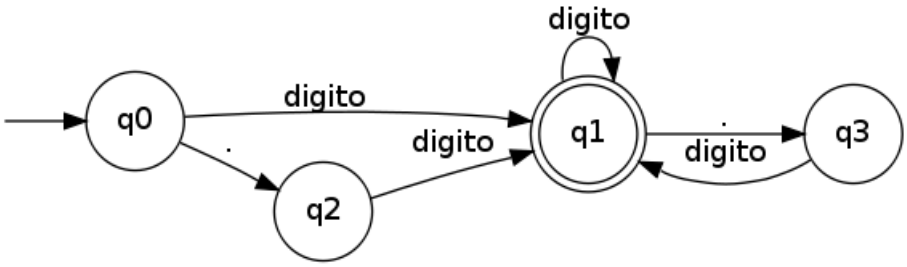


Ilustración 15 Autómata finito para un número real o flotante

Hasta este punto se puede decir que tenemos diseñado nuestro analizador léxico basado en los fundamentos matemáticos, ya que el uso de los autómatas permite definir si una cadena es parte del lenguaje o no.

4.4 GRAMÁTICA

De acuerdo a las fases del compilador, una vez que se reconoce si una cadena pertenece al lenguaje definido, posteriormente se tiene que analizar la línea de código que representa desde el primer símbolo encontrado hasta el retorno de línea. En esta fase conocida como analizador sintáctico se verifica si la línea completa de código está definida por una regla gramatical, estas reglas gramaticales para el pseudocódigo se presentan a continuación (Se utilizó la forma de Backus-Naur).

Ya que toda gramática G debe estar formada de la forma `(N,T,S,P)` en donde N es el conjunto de los elementos no terminales, T el conjunto de los terminales, S un elemento no terminal que pertenece a N el cual será el nodo inicial y P como una serie de reglas gramaticales, listaremos:

El conjunto de elementos no terminales `(N)`:

<programa>

<cr>

<eol>

<literal>

<tipo>

<tipo-basico>

<tipo-arreglo>

<decs-global-opc>

<decs-global>

<dec-global>

<decs-basico>

<dec-basico>

<decs-arrelgo>

<dec-arrelgo>

<tam-arrelgo>

<exp-arreglo>

<elems-arreglo>

<elem-arreglo>

<decs-sub-opc>

<decs-sub>

<dec-sub>

<dec-tipo-sub>

<lista-param-opc>

<lista-param>

<param>

<bloque>

<bloque-sen-opc>

<bloque-sen>

<sen>

<sen-dec-var-local>

<dec-var-local>

<sen-if>

<sen-subsen-final>

<sen-<exp>>

<sen-exp>

<sen-if-then>

<sen-if-then-else>

<sen-if-then-else-if>

<sen-switch>

<bloque-switch>

<sen-bloques-switch>

<sen-bloque-switch>

<etiqs-switch>

<etiq-switch>

<sen-while>

<sen-while-if>

<sen-do>

<sen-for>

<sen-for-if>

<ini-for>

<sen-lista-exp>

<sen-return>

<primario>

<lista-args-opc>

<lista-args>

<inv-metodo>

<inv-entrada>

<var-entrada>

<inv-imprime>

<vars-imprime>

<var-imprime>

<acceso-arreglo>

<exp-postfix>

<exp-post-inc>

<exp-post-dec>

<exp-unaria>

<exp-unaria-otra>

<exp-mult>

<exp-aditiva>

<exp-relacional>

<exp-igual>

<exp-and>

<exp-or>

<exp-asig>

<asig>

<lado-izq>

<op-asig>

<exp>

<exp-cons>

El conjunto de terminales (T):

INICIOPROGRAMA

FINPROGRAMA

PRINCIPAL

ENTRADA

IMPRIME

PROCEDIMIENTO

REGRESAR

INICIO

FIN

ENTERO

TEXTO

FLOTANTE

BOOL

SI

ENTONCES

SINO

ENCASO

OPC

OTRO

MIENTRAS

HACER

REPETIR

HASTA

PARA

String NL

CIERTO

FALSO

PARENI

PAREND

LLAVEI

LLAVED

CORCHI

CORCHD

COMA

DOSPUNTOS

PUNTOCOMA

IGU

MASMAS

MENMEN

MAS

MEN

MUL

DIV

MOD

MASIGU

MENIGU

MULIGU

DIVIGU

MODIGU

MAYQ

MENQ

NO

IGUIGU

MAYQIGU

MENQIGU

NOIGU

Y

O

PUNTO

IDENTIFICADOR

LITENTERO

LITFLOTANTE

LITTEXTO

FINLINEA

ILEGAL

El símbolo inicial (S):

<programa>

El conjunto de reglas o producciones (P):

<programa> ::= IDENTIFICADOR INICIOPROGRAMA <cr> <decs-global-opc> <decs-sub-opc> PRINCIPAL <cr> <bloque> FINPROGRAMA <eol>

<cr> ::= FINLINEA | <cr> FINLINEA

<eol> ::= | <cr>

<literal> ::= LITENTERO | LITFLOTANTE | CIERTO | FALSO | LITTEXTO

<tipo> ::= <tipo-basico> | <tipo-arreglo>

<tipo-basico> ::= ENTERO | FLOTANTE | BOOL | TEXTO

<tipo-arreglo> ::= CORCHI CORCHD <tipo-basico> | CORCHI CORCHD <tipo-arreglo>

<decs-global-opc> ::= | <decs-global>

<decs-global> ::= <dec-global> | <decs-global> <dec-global>

<dec-global> ::= <tipo-basico> <decs-basico> <cr> | <tipo-arreglo> <decs-arrelgo> <cr>

<decs-basico> ::= <dec-basico> | <decs-basico> COMA <dec-basico>

<dec-basico> ::= IDENTIFICADOR | IDENTIFICADOR IGU <exp>

<decs-arrelgo> ::= <dec-arrelgo> | <decs-arrelgo> COMA <dec-arrelgo>

<dec-arrelgo> ::= IDENTIFICADOR <tam-arrelgo> | IDENTIFICADOR IGU <exp-arreglo>

<tam-arrelgo> ::= CORCHI <exp> CORCHD | <tam-arrelgo> CORCHI <exp> CORCHD

<exp-arreglo> ::= LLAVEI <elems-arreglo> LLAVED

<elems-arreglo> ::= <elem-arreglo> | <elems-arreglo> COMA <elem-arreglo>

<elem-arreglo> ::= <exp> | <exp-arreglo>

<decs-sub-opc> ::= | <decs-sub>

<decs-sub> ::= <dec-sub> | <decs-sub> <dec-sub>

<dec-sub> ::= PROCEDIMIENTO <dec-tipo-sub>

<dec-tipo-sub> ::= IDENTIFICADOR PARENI <lista-param-opc> PAREND <cr> <bloque> | <tipo> IDENTIFICADOR PARENI <lista-param-opc> PAREND <cr> INICIO <cr> <bloque-sen-opc> <sen-return> FIN <cr>

<lista-param-opc> ::= | <lista-param>

<lista-param> ::= <param> | <lista-param> COMA <param>

<param> ::= <tipo> IDENTIFICADOR

<bloque> ::= INICIO <cr> <bloque-sen-opc> FIN <cr>

<bloque-sen-opc> ::= | <bloque-sen>

<bloque-sen> ::= <sen> | <bloque-sen> <sen>

<sen> ::= <sen-dec-var-local> | <sen-subsen-final> | <sen-if-then> | <sen-if-then-else> | <sen-while> | <sen-for> | ε

<sen-dec-var-local> ::= dec_var_localVar <cr>

<dec-var-local> ::= <tipo-basico> <decs-basico> | <tipo-arreglo> <decs-arrelgo>

<sen-if> ::= <sen-subsen-final> | <sen-if-then-else-if> | <sen-while-if> | <sen-for-if>

<sen-subsen-final> ::= <bloque> | <sen-<exp>> | <sen-switch> | <sen-do>

<sen-<exp>> ::= <sen-exp> <cr> | <inv-entrada> <cr> | <inv-imprime> <cr>

<sen-exp> ::= <asig> | <exp-post-inc> | <exp-post-dec> | <inv-metodo>

<sen-if-then> ::= SI PARENI <exp> PAREND ENTONCES <cr> <sen>

<sen-if-then-else> ::= SI PARENI <exp> PAREND ENTONCES <cr> <sen-if> SINO <cr> <sen>

<sen-if-then-else-if> ::= SI PARENI <exp> PAREND ENTONCES <cr> <sen-if> SINO <cr> <sen-if>

<sen-switch> ::= ENCASO PARENI <exp> PAREND <cr> <bloque-switch>

<bloque-switch> ::= INICIO <cr> <sen-bloques-switch> FIN <cr> | INICIO <cr> FIN <cr>

<sen-bloques-switch> ::= <sen-bloque-switch> | <sen-bloques-switch> <sen-bloque-switch>

<sen-bloque-switch> ::= <etiqs-switch> <bloque>

<etiqs-switch> ::= <etiq-switch> | <etiqs-switch> <etiq-switch>

<etiq-switch> ::= OPC <exp-cons> DOSPUNTOS | OTRO DOSPUNTOS

<sen-while> ::= MIENTRAS PARENI <exp> PAREND HACER <cr> <sen>

<sen-while-if> ::= MIENTRAS PARENI <exp> PAREND HACER <cr> <sen-if>

<sen-do> ::= REPETIR <cr> <bloque-sen-opc> HASTA PARENI <exp> PAREND <cr>

<sen-for> ::= PARA PARENI <ini-for> PUNTOCOMA <exp> PUNTOCOMA <sen-lista-exp> PAREND <cr> <sen>

<sen-for-if> ::= PARA PARENI <ini-for> PUNTOCOMA <exp> PUNTOCOMA <sen-lista-exp> PAREND <cr> <sen-if>

<ini-for> ::= <sen-lista-exp> | <dec-var-local>

<sen-lista-exp> ::= <sen-exp> | <sen-lista-exp> COMA <sen-exp>

<sen-return> ::= REGRESAR <exp> <cr>

<primario> ::= <literal> | PARENI <exp> PAREND | <inv-metodo> | <acceso-arreglo>

<lista-args-opc> ::= | <lista-args>

<lista-args> ::= <exp> | <lista-args> COMA <exp>

<inv-metodo> ::= IDENTIFICADOR PARENI <lista-args-opc> PAREND

<inv-entrada> ::= ENTRADA <var-entrada>

<var-entrada> ::= IDENTIFICADOR | <var-entrada> COMA IDENTIFICADOR

<inv-imprime> ::= IMPRIME <vars-imprime>

<vars-imprime> ::= <var-imprime> | <vars-imprime> COMA <var-imprime>

<var-imprime> ::= <exp> | NLTxt

<acceso-arreglo> ::= IDENTIFICADOR <tam-arrelgo>

<exp-postfix> ::= <primario> | IDENTIFICADOR | <exp-post-inc> | <exp-post-dec>

<exp-post-inc> ::= <exp-postfix> MASMAS

<exp-post-dec> ::= <exp-postfix> MENMEN

<exp-unaria> ::= MAS <exp-unaria> | MEN <exp-unaria> | <exp-unaria-otra>

<exp-unaria-otra> ::= <exp-postfix> | NO <exp-unaria>

<exp-mult> ::= <exp-unaria> | <exp-mult> MUL <exp-unaria> | <exp-mult> DIV <exp-unaria> | <exp-mult> MOD <exp-unaria>

<exp-aditiva> ::= <exp-mult> | <exp-aditiva> MAS <exp-mult> | <exp-aditiva> MEN <exp-mult> | <exp-aditiva> PUNTO <exp-mult>

<exp-relacional> ::= <exp-aditiva> | <exp-relacional> MENQ <exp-aditiva> | <exp-relacional> MAYQ <exp-aditiva> | <exp-relacional> MENQIGU <exp-aditiva> | <exp-relacional> MAYQIGU <exp-aditiva>

<exp-igual> ::= <exp-relacional> | <exp-igual> IGUIGU <exp-relacional> | <exp-igual> NOIGU <exp-relacional>

<exp-and> ::= <exp-igual> | <exp-and> Y <exp-igual>

<exp-or> ::= <exp-and> | <exp-or> O <exp-and>

<exp-asig> ::= <exp-or>

<asig> ::= <lado-izq> <op-asig> <exp-asig>

<lado-izq> ::= IDENTIFICADOR | <acceso-arreglo>

<op-asig> ::= IGU | MULIGU | DIVIGU | MODIGU | MASIGU | MENIGU

<exp> ::= <exp-asig> | ε

<exp-cons> ::= LITENTERO | LITTEXTO

CAPÍTULO V DESARROLLO DEL COMPILADOR

Anteriormente se ha explicado la fundamentación teórica que sirve como base para el desarrollo de un compilador, desde autómatas y lenguajes hasta la parte matemática de cada una de las fases del proceso de compilación.

En éste capítulo se mostrará el proceso de desarrollo, la forma en que pasamos los autómatas y las expresiones regulares a un sistema que pueda interpretarlos.

La estructura general del compilador no varía en comparación con la teoría, ya que el flujo de información que representará el programara será el mismo, mostrado en la Ilustración 16, sin embargo a diferencia de la teoría existe un cambio en el proceso que lleva a cabo el compilador en referencia a la creación de la tabla de símbolos, lo cual se explicará más adelante.

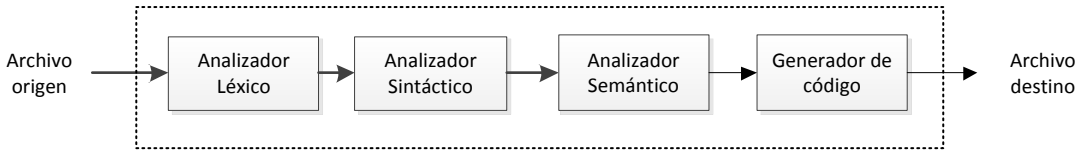


Ilustración 16 Diagrama general del compilador

Como se muestra en la Ilustración 17 y como todo en compilador, el proceso inicia en el momento en que ingresa el conjunto de símbolos ordenados (archivo origen), para este conjunto es necesario verificar que cumpla las reglas léxicas de tal forma que se puedan agrupar en palabras, para recordar, una palabra es un conjunto de caracteres que inicia y termina con épsilon y los caracteres intermedios pasan la validación de las reglas léxicas o autómatas; a partir de éste proceso y hasta que el compilador termine su flujo se verifica que no se produzcan errores en caso de presentarse el flujo se interrumpe.

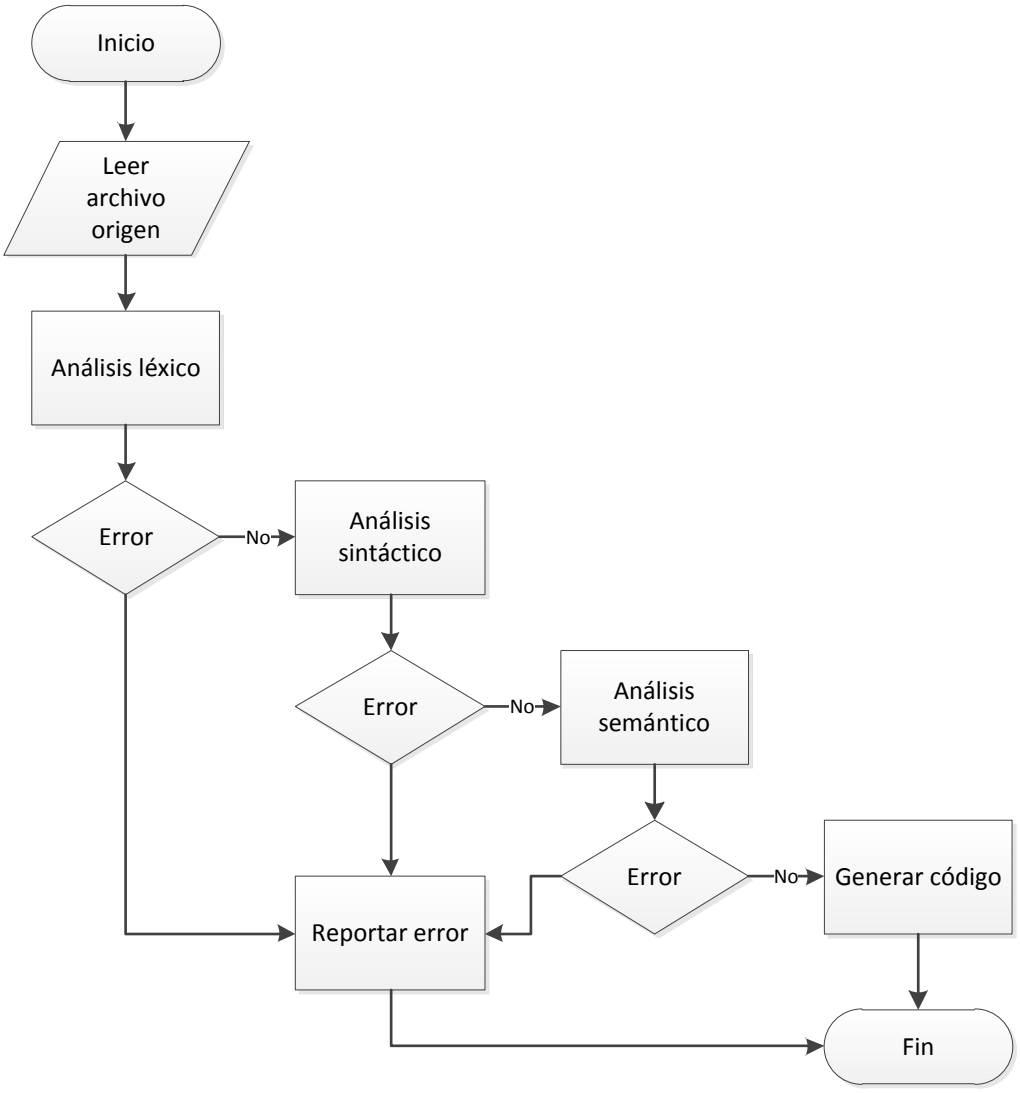


Ilustración 17 Diagrama de flujo del compilador.

Cuando los símbolos son reconocidos por el compilador como palabras es necesario verificar que estas palabras formen una oración, es decir, que estén ordenadas de forma lógica cumpliendo las reglas gramaticales y finalmente, el último proceso de verificación es el análisis semántico, en donde se verifican los tipos de datos.

5.1 ANALIZADOR LÉXICO

Como se ha mencionado a lo largo de éste trabajo, el análisis léxico es el proceso que sirve para identificar que los símbolos de entrada pertenecen al alfabeto de un lenguaje dado; lo anterior aplicado al entorno del compilador, se refiere a que ésta etapa debe analizar que el programa a

compilar contenga caracteres del lenguaje español, así como números y símbolos permitidos por el lenguaje definido.

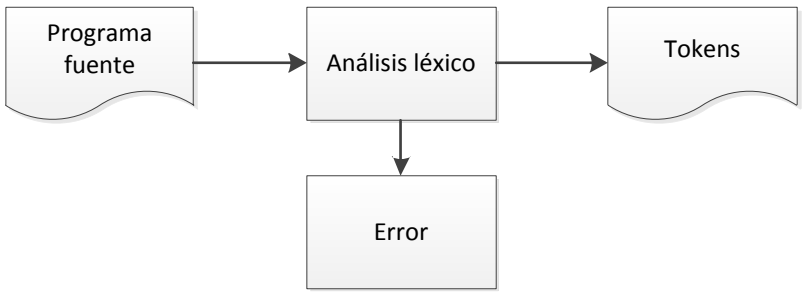


Ilustración 18 Diagrama general del analizador léxico

Es posible realizar el análisis léxico de diferentes formas, como las que se muestran a continuación.

El primer paso que se debe cubrir es verificar que los símbolos en el archivo origen pertenezcan al lenguaje, para computar lo anterior se pueden presentar los siguientes algoritmos:

Algoritmo simple para identificar que los símbolos pertenezcan al lenguaje:

- Crear un arreglo que contenga todos los símbolos permitidos por el lenguaje.
- Leer el archivo de origen.
- Interpretar el archivo de tal forma que se pueda leer caracter por caracter.
- Comparar el caracter con el arreglo para verificar que éste caracter es válido, es decir, que el caracter leído sea igual a alguno de los caracteres del arreglo.
- En caso de leer el símbolo de fin de archivo se indicará que no existen caracteres incorrectos, de lo contrario el flujo se interrumpe y se indica el caracter.

Otra forma en la que se puede programar para resolver el mismo problema es la construcción de un algoritmo que simule el recorrido de un autómata, a grandes rasgos se puede implementar de la siguiente forma:

- Crear un arreglo unidimensional para representar el conjunto de estados finales
- Crear un arreglo unidimensional para representar el alfabeto
- Crear un arreglo bidimensional para representar la tabla de transiciones
- Crear una variable para representar el estado inicial

- Crear un arreglo unidimensional para representar el conjunto de estados
- Leer el archivo de origen.
- Interpretar el archivo para leer caracter por caracter.
- Seleccionar el primer caracter y buscarlo en el arreglo para identificar la transición.
- Si llega a un estado válido se lee el siguiente caracter hasta llegar al estado final

La implementación del analizador léxico no termina en tener exclusivamente autómatas que acepten cadenas, también es necesario entregar tokens que servirán para la etapa posterior de análisis.

Actualmente existen muchas herramientas que permiten generar analizadores léxicos de forma rápida, sencilla y óptima. Para el compilador aquí desarrollado se utilizará la herramienta JFlex.

Con lo anterior solamente falta seleccionar la forma en que se desarrollará el analizador léxico y que será decisivo para el resto de las fases, es posible crear una lista con puntos a favor y en contra, así como dar más importancia a factores como tiempo de desarrollo, dificultad, entre otras; pero el aspecto que permitió tomar la decisión fue el mantenimiento que se le puede ofrecer al compilador, una ventaja de crear el analizador con una herramienta es que si se decide modificar el lenguaje solamente es necesario modificar el archivo de configuración que está escrito mediante reglas matemáticas.

5.1.1 EL CREADOR DE ANALIZADORES LÉXICOS JFLEX

JFlex es un generador de analizadores léxicos hecho en Java. Ésta sección no pretende ser un manual en la utilización de JFlex, sino solamente mostrar su uso para la realización del compilador, en caso de requerir documentación para el uso de ésta herramienta, se recomienda revisar la documentación que se encuentra en la página oficial de JFlex (GPL General Public License, 2009)

De forma general, para utilizar JFlex y generar un analizador léxico, se necesita crear un archivo que JFlex pueda compilar, dicho archivo tiene una estructura como se muestra a continuación:

```
Código de usuario %%

Opciones y declaraciones %%

Reglas léxicas
```

en donde:

Código de usuario. Es la sección que permite incluir código Java el cual no será modificado y aparecerá en el archivo generado tal y como fue escrito.

Opciones y declaraciones. Sección que permite utilizar una serie de directivas especificadas por JFlex para configurar el analizador léxico generado de acuerdo a las necesidades del usuario. En ésta sección se hacen también las especificaciones de las expresiones regulares que describen las palabras que el lenguaje acepta.

Reglas léxicas. Sección que permite indicar por medio de código Java las acciones a ejecutar cuando alguna cadena de la serie de símbolos analizados corresponde a alguna expresión regular. Esta sección es la que permite implementar la generación de tokens que se utilizará en la siguiente fase de análisis.

5.1.2 USO DE JFLEX EN EL DESARROLLO

En ésta parte se detalla la implementación de la aplicación matemática del capítulo anterior a los criterios de JFlex. Es importante mencionar que no se muestra completamente el archivo que usa JFlex para generar el analizador léxico, el archivo en su totalidad se encuentra en el apartado de Anexos.

De acuerdo a la especificación de JFlex, en la primera sección del archivo, se utiliza para especificar el código que el usuario considere necesario; para el Compilador, se usó para indicar el paquete Java al que corresponde así como importar las clases necesarias.

```
package com.upiicsa.lenguaje.core;

import com.upiicsa.lenguaje.util.ManejadorErrores;

import com.upiicsa.lenguaje.util.Token;
```

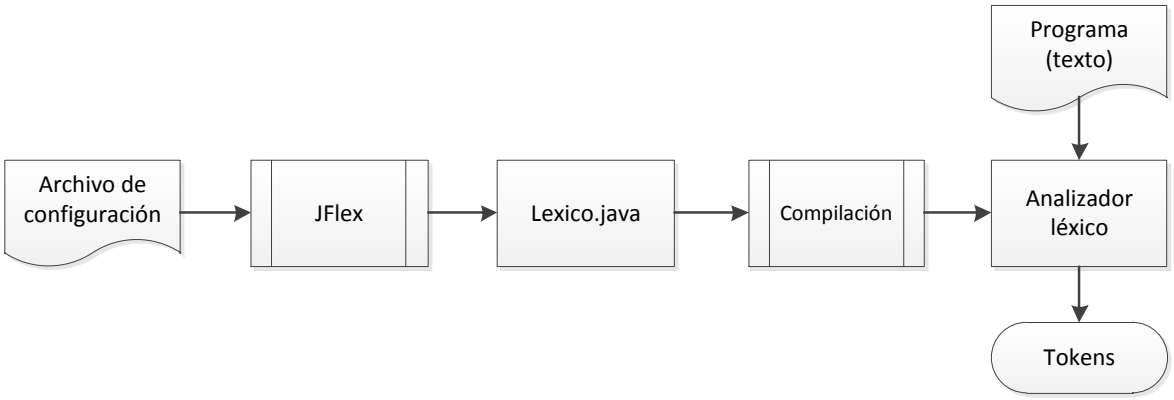


Ilustración 19 Diagrama de operación de JFlex

En la segunda sección se utilizaron directivas para la correcta ejecución de tareas con el Analizador Sintáctico (el cual se explica en la siguiente sección), así como funciones para la creación de tokens, de igual forma, se especificaron las expresiones regulares que se mencionaron en el Capítulo anterior.

El archivo JFlex completo se encuentra como anexo al final de éste trabajo escrito, pero a continuación se muestran algunos ejemplos de la utilización de las expresiones regulares dentro del archivo JFlex.

```
//funcion para el manejo de Tokens

private Token simbolo(int tipo) {

        return new Token(tipo, yyline + 1, yycolumn + 1);

    }

//Ejemplos de especificación de expresiones regulares

TerminadorLinea             =   \r | \n | \r\n

ComentarioMultilinea        =   "/*" [^*] ~"*/" | "/*" "*"+ "/"

Flot1                       =   [0-9]+ \. [0-9]*

Identificador               =   [:jletter:][:jletterdigit:]*
```

Particularmente la última expresión regular contiene `[:jletter:]` y `[:jletterdigit:]`, los cuales son expresiones propias de JFlex, que contienen el conjunto de letras y letras además de los digitos del 0 al 9 respectivamente. Se puede profundizar más en dichas expresiones dentro de la documentación oficial de JFlex en la página oficial del proyecto.

En la última sección se indica que debe hacer el analizador cuando reconoce un token, todo el tratamiento de éstas acciones se llevan a cabo por medio de funciones, las cuales devuelven un objeto del tipo token, se ponen algunos ejemplos:

```
/* Identificadores */

  {Identificador} {return simbolo(Simbolos.IDENTIFICADOR, yytext());}

/* Numeros */

  {NumeroEntero}  {return simbolo(Simbolos.LITENTERO, yytext());}

  {NumeroFlotante}{return simbolo(Simbolos.LITFLOTANTE, yytext());}

/* Fin del archivo */

  <<EOF>>         {return simbolo(Simbolos.EOF, "<EOF>");}
```

La utilización de JFlex permite que la generación del Analizador Léxico sea de manera rápida y sencilla. Para la obtención del archivo Java a partir del archivo de especificaciones de JFlex, se debe ejecutar el comando que se indica en la documentación de JFlex.

5.2 ANALIZADOR SINTÁCTICO

La segunda etapa del análisis de un compilador, como ya se ha mencionado, es el análisis sintáctico, el cual permite identificar que las palabras del lenguaje (tokens, hablando del compilador) siguen correctamente una regla de producción al momento que el analizador léxico devuelve tokens.

Para implementar un Analizador Sintáctico, se utilizan gramáticas libres de contexto, las cuales se denotan por medio de producciones cual es el orden que debe seguir cierta secuencia de tokens,

para lo anterior se utilizan los árboles de derivación como un método de ayuda para verificar que una gramática acepta la secuencia de tokens que se analiza.

Al igual que el Analizador Léxico, para el desarrollo del Analizador Sintáctico se usa una herramienta que permite crear analizadores sintácticos a través de un archivo que contiene especificaciones propias y generan como resultado un archivo Java el cual está listo para ser compilado y utilizado.

5.2.1 EL CREADOR DE ANALIZADORES SINTÁCTICOS CUP

Cup es un generador de analizadores sintácticos hecho en Java. Ésta sección no pretende ser un manual en la utilización de Cup, en caso de requerir documentación para el uso de ésta herramienta, se recomienda revisar la documentación que se encuentra en la página oficial de Cup (Scott Hudson, 2003).

La utilización de Cup es muy parecida al uso de JFlex, se necesita un archivo con extensión CUP, el cual siga las especificaciones que se encuentran establecidas en la documentación del proyecto CUP. El archivo fuente de Cup se divide en secciones, las cuales se listan a continuación:

- Especificaciones de Paquete e “Import”
- Código de usuario
- Listas de símbolos
- Precedencia
- Gramática

en donde:

Especificaciones de Paquete e “Import”. Sección que permite indicar el paquete al que pertenece el analizador sintáctico a crear, así como la inclusión de librerías que se necesiten

Código de usuario. Sección que permite incluir código Java el cual no será modificado y aparecerá en el archivo generado tal y como fue escrito.

Listas de símbolos. Aquí se indican todos aquellos terminales y no terminales de la gramática

Precedencia. En caso de ser necesario, en ésta sección se puede indicar la precedencia que tienen algunos terminales sobre otros, esto para eliminar ambigüedades de la gramática

Gramática. La definición de la gramática.

5.2.2 USO DE CUP EN EL DESARROLLO

En ésta parte se detalla la implementación de la aplicación matemática del capítulo anterior a los criterios de CUP. Es importante mencionar que no se muestra completamente el archivo que usa CUP para generar el analizador sintáctico, el archivo en su totalidad se encuentra en la parte de Anexos.

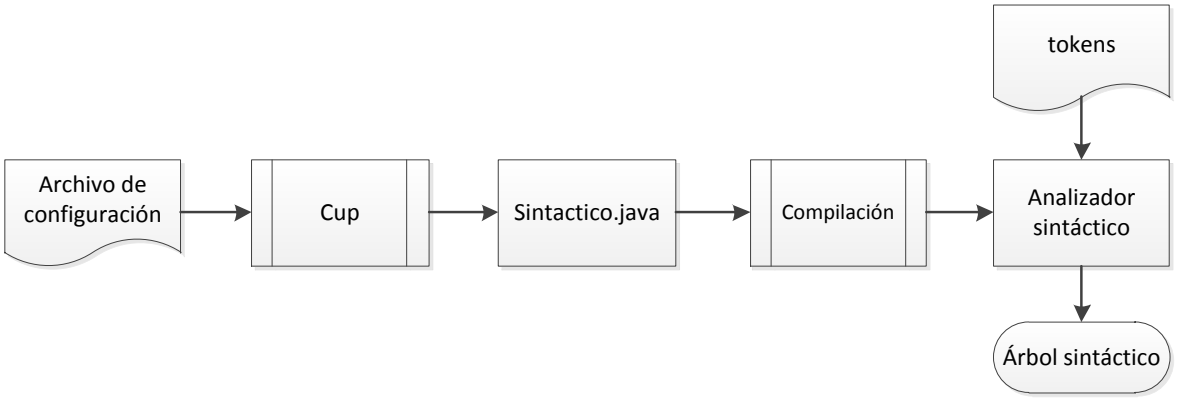


Ilustración 20 Diagrama de operación de Cup

De acuerdo a la especificación de CUP, en la primera sección del archivo, se utiliza para especificar el paquete al cual corresponderá el archivo generado, así como importar las clases necesarias.

```
package com.upiicsa.lenguaje.core;

import com.upiicsa.lenguaje.util.ManejadorErrores;

import com.upiicsa.lenguaje.util.Token;

import java_cup.runtime.Symbol;
```

En la segunda sección se implementaron funciones, que siguiendo la documentación oficial del proyecto CUP, sirven para personalizar el comportamiento del analizador al momento de detectarse un error, así como también indicar las acciones a ejecutar cuando se haga una recuperación de algún error.

```
@Override

    public void report_error(String message, Object info) {

        Token error = (Token) info;

        StringBuilder msjError = new StringBuilder(TipoError.ERROR_SINTAXIS.getMensaje());

        msjError.append(", encontrado cerca de: ");

        msjError.append(error.value.toString());

        manejadorErrores.notificarErrorGrave(msjError.toString(), error.linea);

    }

    @Override

    public void report_fatal_error(String message, Object info) {

        done_parsing();

        report_error(message, info);

    }

    @Override

    public void syntax_error(Symbol cur_token) {

        report_error(null, cur_token);

    }
```

JFlex ofrece de forma nativa soporte para interactuar con CUP, una de las directivas del archivo de especificaciones de JFlex hace posible ésta interacción. Los terminales que se definen en el archivo CUP son los que utiliza JFlex para hacer el reconocimiento léxico, y de acuerdo a la implementación del compilador, con estos terminales es posible pasar tokens al analizador sintáctico.

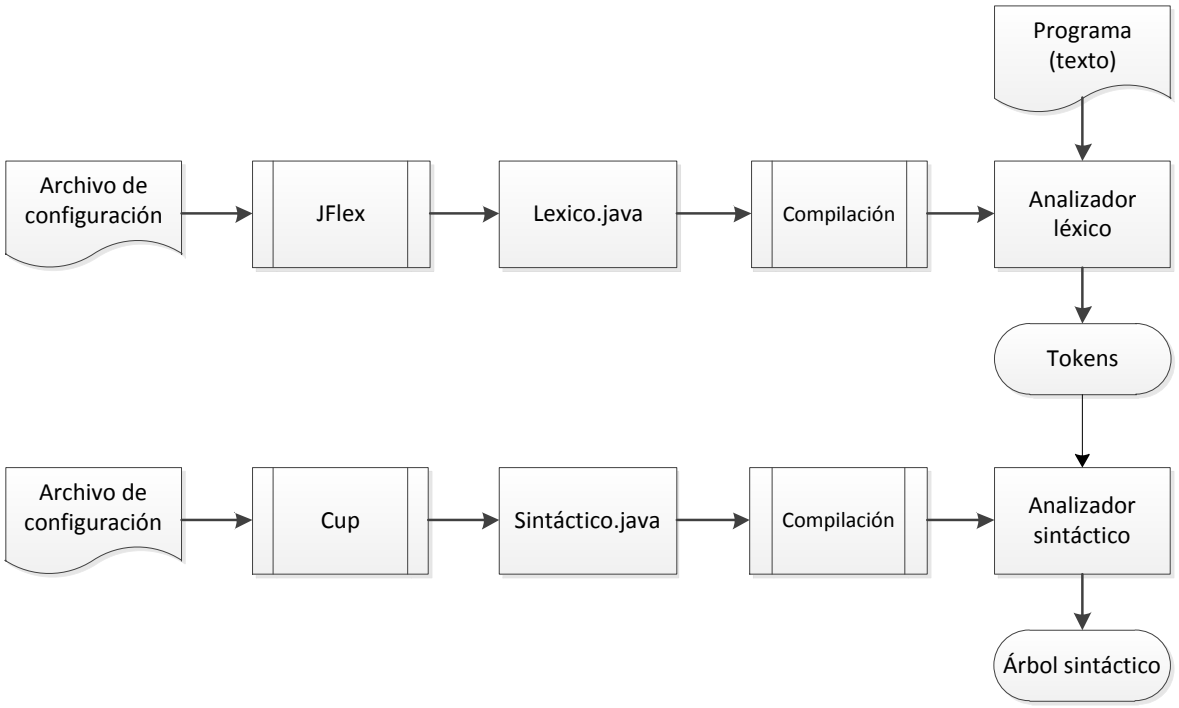


Ilustración 21 Diagrama de integración de JFlex y Cup

La forma en la que se especifican los terminales y no terminales se encuentra documentada en el sitio oficial del proyecto CUP, como ejemplo se ponen las siguientes líneas:

```
/* Palabras Reservadas - Programa */

    terminal INICIOPROGRAMA;

    terminal FINPROGRAMA;

    terminal PRINCIPAL;

/* Palabras Reservadas - Subrutinas */
```

```
    terminal PROCEDIMIENTO;

    terminal REGRESAR;

/* Palabras Reservadas - Bloques de Instrucciones */

    terminal INICIO;

    terminal FIN;

/* Palabras Reservadas - Tipos de Datos */

    terminal ENTERO;

    terminal TEXTO;

    terminal FLOTANTE;

    terminal BOOL;

/* Declaracion de Metodo */

    non terminal declaraciones_subrutina_opc;

    non terminal declaraciones_subrutina;

    non terminal declaracion_subrutina;

    non terminal declaracion_tipo_subrutina;

    non terminal declarador_subrutina;

    non terminal op_lista_paramatros;

    non terminal lista_paramatros;

    non terminal parametro;

/* Declaracion Principal */

    non terminal declaracion_principal_programa;
```

La siguiente parte del archivo, la cual corresponde a ***Precedencia***, no fue utilizada en la realización del compilador, lo anterior debido a que la gramática empleada no presenta ambigüedades.

La última parte del archivo de especificación de CUP, es la gramática como tal. El archivo es muy extenso, pero para demostrar cómo se indican las producciones, se ilustra con un ejemplo:

```
/* Estado Inicial */

  start with programa;

/* Inicio de la Gramatica */

  programa    ::=    IDENTIFICADOR    INICIOPROGRAMA    fin_instruccion_nl
estructura_programa FINPROGRAMA fin_instruccion | error ;

/* Declaracion del Programa */

  estructura_programa           ::=           declaraciones_var_global_opc
declaraciones_subrutina_opc declaracion_principal_programa;

/* Declaracion de Variables Globales */

        declaraciones_var_global_opc ::= |     declaraciones_var_global ;
```

Para indicar cuál es el no terminal que da inicio a la gramática se utiliza

```
start with programa;
```

Lo que indica que el no terminal inicial es programa. Las producciones siguen una Notación de Backus-Naur, lo que la hace sencilla y simple de entender.

5.3 ANALIZADOR SEMÁNTICO

El análisis semántico se encarga de validar aspectos que el análisis sintáctico es incapaz de verificar, por ejemplo la comprobación de tipos y comprobación del alcance o “scope” de las variables utilizadas a lo largo del programa.

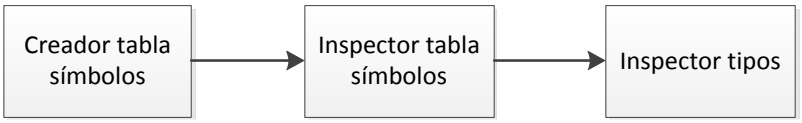


Ilustración 22 Diagrama de flujo del analizador semántico

El desarrollo del Analizador Semántico del pseudocódigo definido previamente, está dividido en 3 componentes los cuales recorren el árbol sintáctico generado durante la etapa de análisis sintáctico. Los componentes son el Creador de tabla de símbolos, el Inspector de tabla de símbolos y el Inspector de tipos.

El Analizador Semántico recorre el árbol sintáctico una vez por cada componente, lo que significa que para completar el análisis semántico, el árbol se recorre tres veces.

5.3.1 CREADOR DE TABLA DE SÍMBOLOS

El Creador de tabla de símbolos es el primer componente del Analizador Semántico, y su tarea principal es generar las entradas de las variables y subrutinas declaradas en el programa dentro de la tabla de símbolos; además verifica si el identificador de una variable ya ha sido declarado previamente en el programa, lo anterior con la finalidad de evitar declaraciones múltiples del mismo identificador y para asegurar que no se usan identificadores que no han sido declarados.

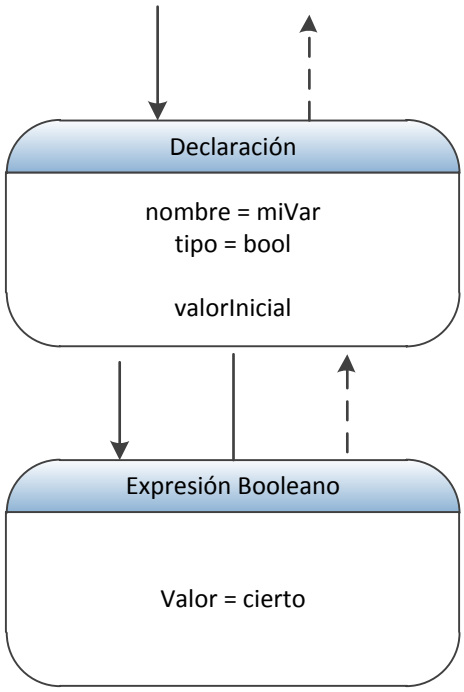


Tabla de símbolos

Nombre	Tipo	Valor	Parámetros
miVar	bool	cierto	Ø

Ilustración 23 Recorrido del árbol para una variable y su entrada

Cada una de las entradas de la tabla de símbolos contiene la siguiente información:

- Nombre o identificador de la variable o subrutina
- El tipo de dato de la variable o subrutina; el tipo de la subrutina es el tipo de dato que regresa después de su ejecución, si la subrutina no regresa algún tipo de dato, la entrada en la tabla de símbolos toma un tipo de dato “void” (solamente se utiliza éste tipo de dato dentro de la tabla de símbolos, no es un tipo que se pueda usar dentro del programa en la declaración de variables).
- El valor con el que fue inicializada la variable. Cuando se agrega una subrutina, ésta propiedad toma un valor nulo.

- Lista de parámetros que tiene una subrutina; si la subrutina no contiene parámetros, el valor es un objeto que contiene una lista con cero elementos. Cuando se agrega una variable, ésta propiedad toma valor nulo.

El recorrido del árbol para una variable y la creación de su entrada en la tabla de símbolos se muestran en la Ilustración 23.

Al recorrer éste árbol de ejemplo, se encuentra que hay un nodo que indica la declaración de una variable, la cual se añade a la tabla de símbolos del alcance o “scope” que corresponda, suponiendo que fue una declaración de variable global, se agrega la entrada a la tabla de símbolos global.

El recorrido del árbol para una subrutina y la creación de su entrada en la tabla de símbolos se muestran en la Ilustración 24.

Al recorrer este árbol de ejemplo, se encuentra que hay un nodo que indica la declaración de una subrutina, la cual se añade a la tabla de símbolos de alcance o “scope” Global, ya que las subrutinas pueden ser llamadas en cualquier parte del programa, el nodo “*Parámetros*” y “*Bloque*” tienen una propia sabla de símbolos debido a que cualquier variable declarada dentro de la subrutina solamente puede ser accedida y utilizada dentro de la subrutina.

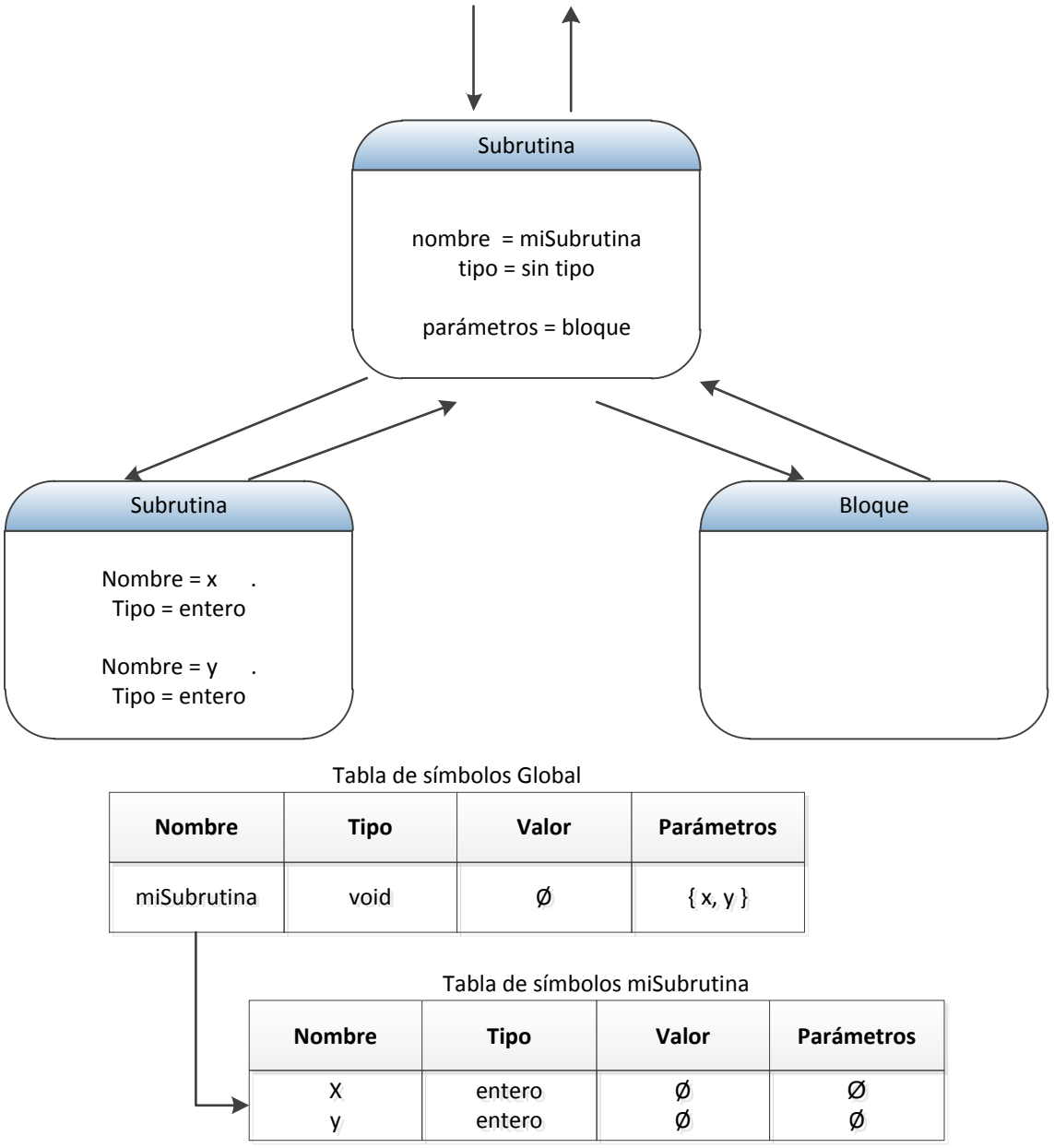


Ilustración 24 Recorrido del árbol para una subrutina y su entrada

Cuando existe un identificador (nombre de variable o subrutina) que ya ha sido utilizado para declarar otra variable o subrutina, significa que existe una entrada en la tabla de símbolos con dicho identificador, cuando se presenta esto, Ilustración 25, el Creador de tabla de símbolos informa acerca de éste evento.

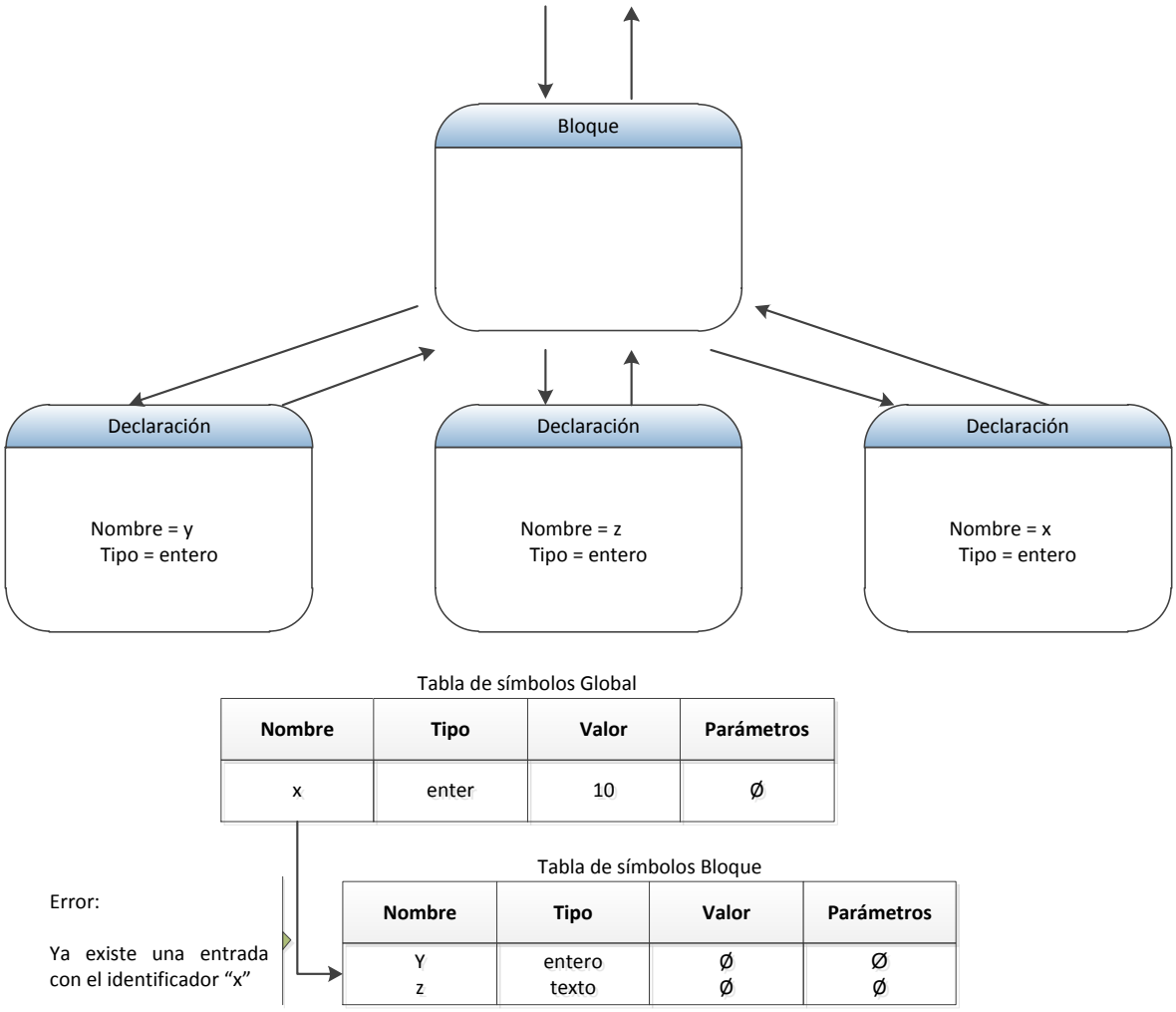


Ilustración 25 Representación de una entrada duplicada

Para que el Creador de tabla de símbolos informe de la existencia de una variable previamente declarada con el mismo identificador, no busca solamente en la tabla del alcance o "scope" donde agrega las entradas, sino busca también en todas aquellas tablas que son "padres" de la tabla de símbolos de su alcance, en la Ilustración 25, ya existe una variable x que fue declarada en las variables globales (debido a que se encuentra en la tabla de símbolos global), por lo cual no es posible declarar otra variable con el identificador x.

Cuando se llega a un nodo del árbol sintáctico que tiene una expresión la cual involucra a una variable, se revisa que exista una entrada con ése identificador, en caso de no existir dicha entrada Ilustración 26, el Creador de tabla de símbolos informa acerca de éste evento.

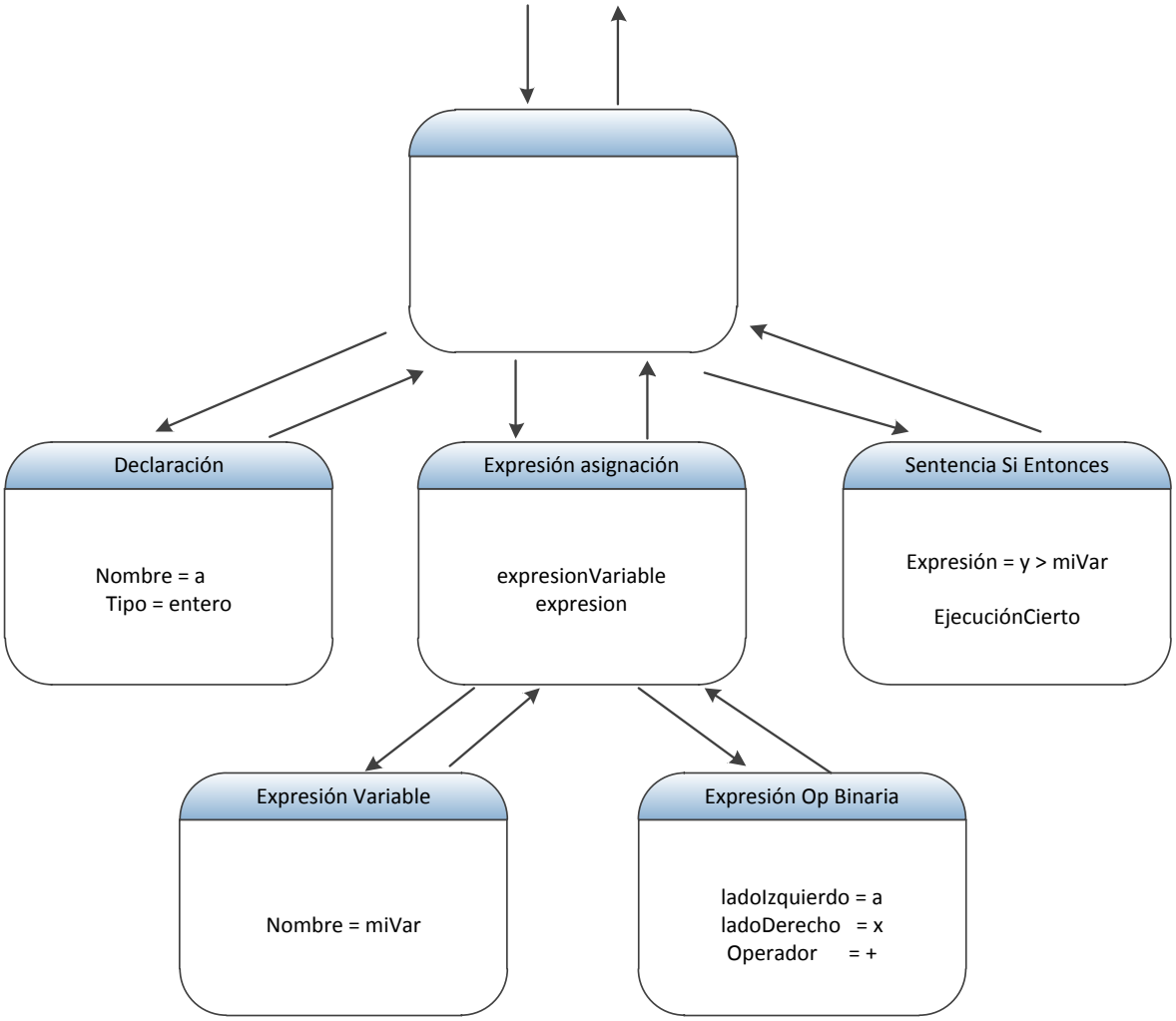


Tabla de símbolos Global

Nombre	Tipo	Valor	Parámetros
miVar	entero	3	Ø
x	entero	5	Ø

Tabla de símbolos Bloque

Nombre	Tipo	Valor	Parámetros
a	entero	Ø	Ø

Error:

No existe una entrada con el identificador "y"

Ilustración 26 Representación de un identificador no asignado

El nodo "*Declaración*" agrega la entrada de la variable "a" a la tabla de símbolos correspondiente a su alcance o "scope". El nodo "*Expresión Asignación*" utiliza las variables "a" y "x" para realizar una suma y la variable `miVar` para asignar el resultado de la suma a dicha variable. El nodo "*Sentencia*

Si-Entonces” contiene una condicional que utiliza las variables “y” y “`miVar`”, dado que no existe una entrada de la variable “y” en la tabla de símbolos del alcance actual ni en las tablas padres, se notifica el error de que la variable utilizada no ha sido declarada.

5.3.2 INSPECTOR DE TABLA DE SÍMBOLOS

El Inspector de tabla de símbolos es el segundo componente del Analizador Semántico, y su tarea es verificar que las subrutinas utilizadas dentro del programa ya han sido declaradas.

El Creador de tabla de símbolos revisa solamente que las variables ya han sido declaradas (además de las tareas explicadas en el apartado anterior) y el componente aquí descrito, el Inspector de tabla de símbolos, se encarga de hacer la misma revisión pero para subrutinas, la verificación en fases separadas existe por el hecho de que el Creador de tabla de símbolos va agregando las entradas mientras recorre el árbol y esto permite que las variables usadas a lo largo del programa sean verificadas con las entradas que existen en la tabla de símbolos hasta el punto que se ha recorrido, asegurando así la comprobación del alcance o “scope” de las variables, y una subrutina puede ser llamada en algún punto del programa sin que necesariamente ya se haya declarado dicha subrutina, lo que se traduce en que aún no existe la entrada de la subrutina en la stabla de símbolos.

La presencia de una situación como la anterior se soluciona en una revisión hecha en dos fases, en la primera fase se crean en la tabla de símbolos las entradas de todas las subrutinas, dicha tarea la realiza el Creador de tabla de símbolos, y en la segunda fase, cuando existe una llamada a subrutina dentro del programa, se busca que exista la entrada correspondiente en la tabla de símbolos, tarea que realiza el Inspector de tabla de símbolos, si no existe la entrada, el Inspector de tabla de símbolos informa acerca de éste evento, en caso contrario se pasa al tercer componente del Analizador Semántico.

Para ejemplificar la explicación anterior se propone el siguiente fragmento de código.

```
procedimiento A()
    inicio
        ...
        variable = B()
```

```
        ...
    fin


procedimiento entero B()
    inicio
        ...
        código subrutina
        ...
    fin
```

Cuando se recorre el árbol por primera vez, es decir, por medio del Creador de tabla de símbolos, se agrega una entrada a la tabla de símbolos para la subrutina A, y cuando se recorre el bloque de ésta subrutina, se encuentra una llamada a la subrutina B, la cual no ha generado una entrada en la tabla de símbolos, dado éste escenario, el Creador de tabla de símbolos no revisa que existan entradas para llamadas a subrutinas sólo se encarga de revisar que las variables a lo largo del programa ya posean su correspondiente entrada en la tabla de símbolos; una vez que se termina el primer recorrido y comienza el segundo, es decir, el recorrido del Inspector de tabla de símbolos, se hace la comprobación de que las subrutinas tengan su entrada en la tabla de símbolos y así las llamadas a subrutinas pueden ser comprobadas.

5.3.3 INSPECTOR DE TIPOS

El Inspector de Tipos es el tercer componente del Analizador Semántico, y su tarea es verificar que todas las expresiones y sentencias a lo largo del programa no contienen errores en el manejo de los tipos de datos

El Inspector de Tipos realiza el tercer recorrido del árbol sintáctico, una vez que se ha llegado a éste punto, se tiene la certeza de que todas las variables y subrutinas utilizadas en el programa tienen una entrada correspondiente en la tabla de símbolos, lo que solamente deja lugar a revisar que los valores asignados a variables, el valor de retorno de subrutinas y las operaciones,

involucren expresiones que correspondan al tipo de dato correcto, por ejemplo, verificar que la condición en una sentencia repetitiva sea del tipo booleana, que cuando haya una suma sea entre tipos numéricos, etc.

El Inspector de Tipos informa cuando existe un error de tipos de datos en algún punto del programa, en caso contrario se termina el análisis semántico y se pasa a la generación de código.

5.4 GENERADOR DE CÓDIGO

El Generador de Código es el componente del Compilador que traduce el código fuente del Pseudocódigo en español al código destino, en éste caso a código Java. Al igual que los componentes del Analizador Semántico, el Generador de Código realiza un recorrido en el árbol sintáctico para traducir cada uno de los nodos.

El código Java que se genera al finalizar el recorrido del Generador de Código, se guarda en un archivo con extensión *java* el cual está listo para ser procesado por el compilador de Java y posteriormente ser utilizado por la Java Virtual Machine y ejecutar el archivo con extensión *class* que se genera. El nombre del archivo Java generado toma el nombre que se indicó en el pseudocódigo en español.

5.5 EJECUCIÓN

El compilador está desarrollado en Java y está contenido en un archivo JAR, dentro del cual se encuentran todas las clases que hacen posible el proceso de compilación. Para ejecutar el compilador de pseudocódigo en español, es necesario que el JAR del compilador y el JAR que contiene las clases de Cup estén en el classpath de Java, para dar solución a la condición anterior, el archivo MANIFEST del JAR del compilador indica que el JAR de JFlex (que incluye las clases de Cup) debe de estar en el mismo directorio que el JAR del compilador.

Hay dos opciones de ejecución del compilador, la primera es a través de la línea de comando y la segunda es utilizando la interfaz gráfica que provee el JAR del compilador.

Para ejecutar el compilador a través de la línea de comando, se debe introducir lo siguiente:

```
java –jar JAR_del_Compilador [opciones] archivo_1 <archivo_2 … archivo_n>
```

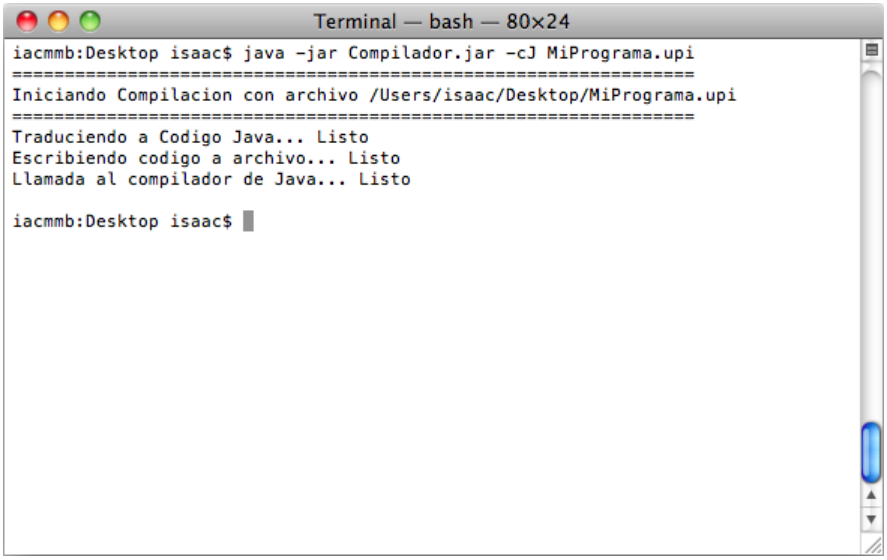
en donde:

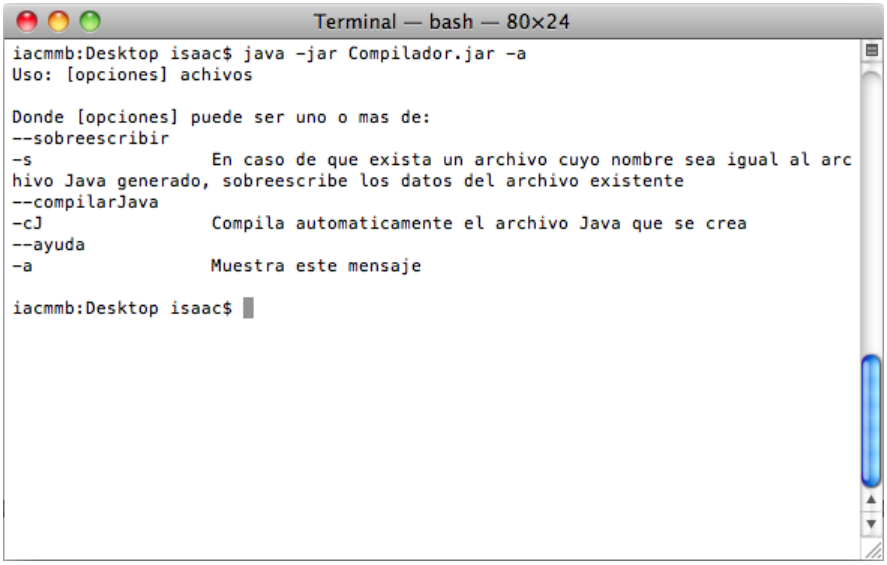
- `java -jar`.- representa el comando que ejecuta la Java Virtual Machine para la ejecucion de programas java a partir de un archivo JAR.
- `JAR_del_Compilador`.- es el nombre del JAR del compilador
- `[opciones]`.- es una lista opcional de parametros para la ejecución del compilador, se mencionan y detallan más adelante
- `archivo_1 <archivo_2 … archivo_n>`.- el nombre del archivo o archivos que serán procesados por el compilador de pseudocódigo

Los parametros que se pueden utilizar son los siguientes:

- `--sobreescribir ó -s`.- en caso de que exista un archivo Java con el mismo nombre al que será generado, sobreescribir el contenido del archivo anterior con el contenido del archivo que se generará después de compilar el pseudocódigo.
- `--compilarJava ó -cJ`.- indica que se debe llamar al compilador de java (comando javac) después de generar el archivo java para que se genere el archivo class.
- `--ayuda ó -a`.- muestra ayuda en la pantalla

A continuación se muestran pantallas de ejecución en el modo de línea de comando, la primer pantalla muestra la compilación de un archivo y la segunda muestra la ayuda cuando se utiliza el comando `-a`.





Para ejecutar el compilador en su modo gráfico puede utilizarse la línea de comandos o simplemente dando doble clic sobre el archivo JAR del compilador, para ejecutar desde la línea de comando, se debe escribir el siguiente comando:

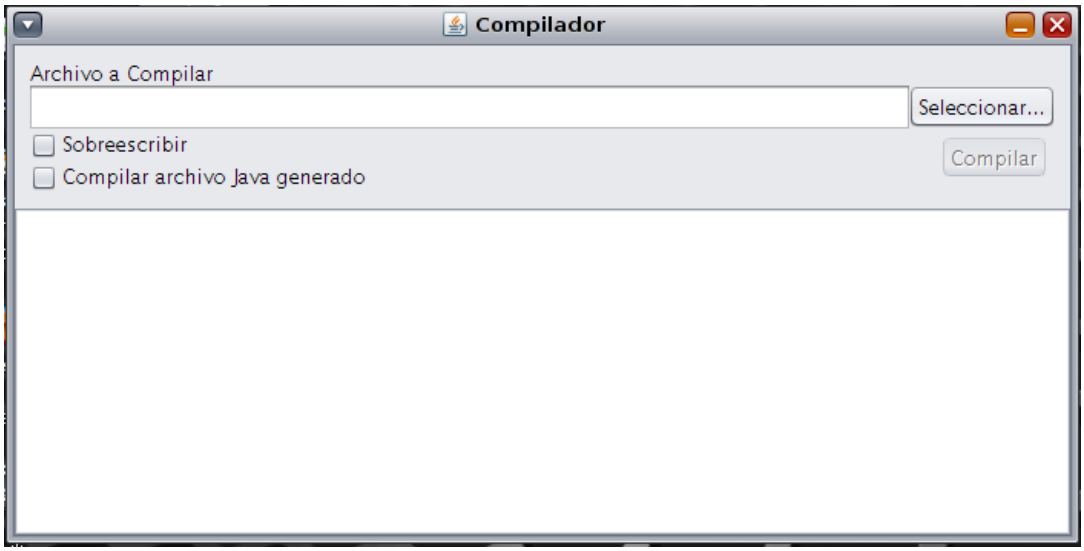
```
java –jar JAR_del_Compilador
```

en donde:

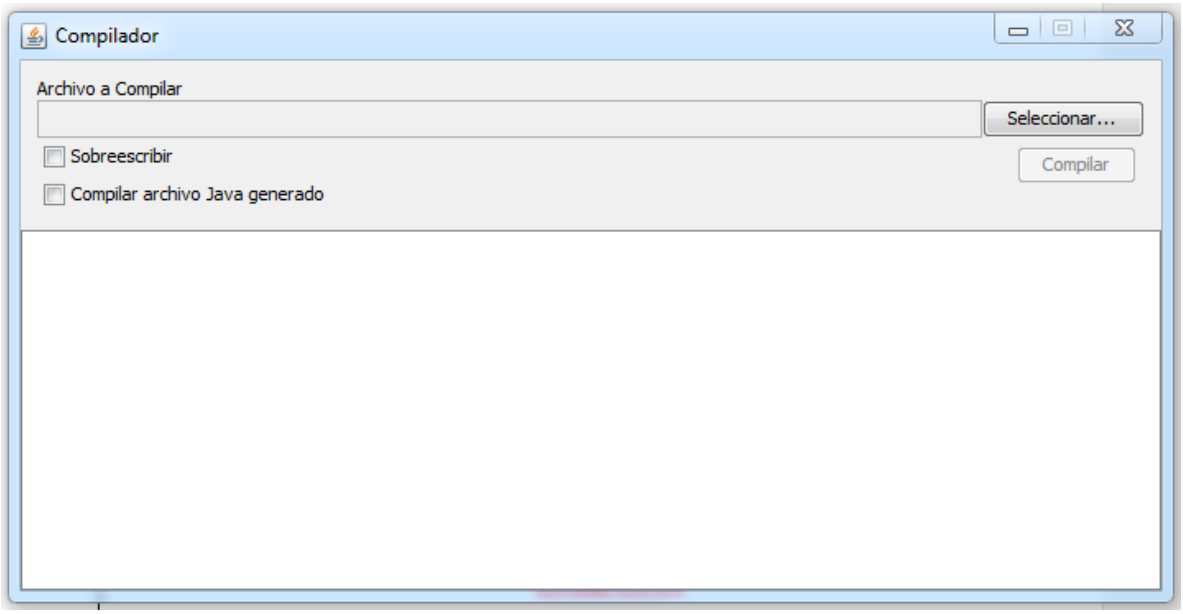
- `java -jar`.- representa el comando que ejecuta la Java Virtual Machine para la ejecucion de programas java a partir de un archivo JAR.
- `JAR_del_Compilador`.- es el nombre del JAR del compilador

Sin importar el metodo de acceso a la interfaz gráfica, se presentará la siguiente ventana:

En Mac OS X



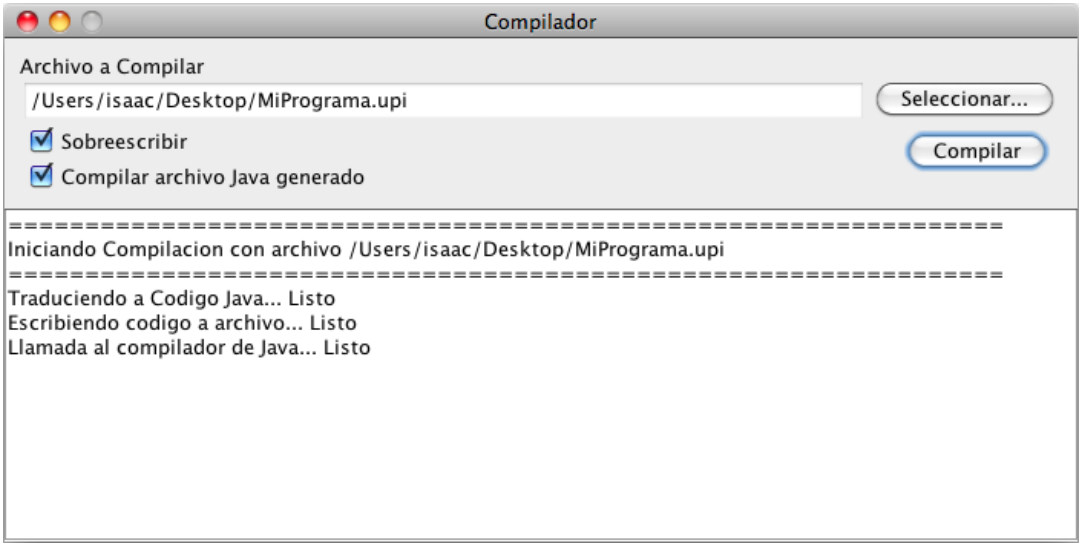
En Oracle Solaris Express 11



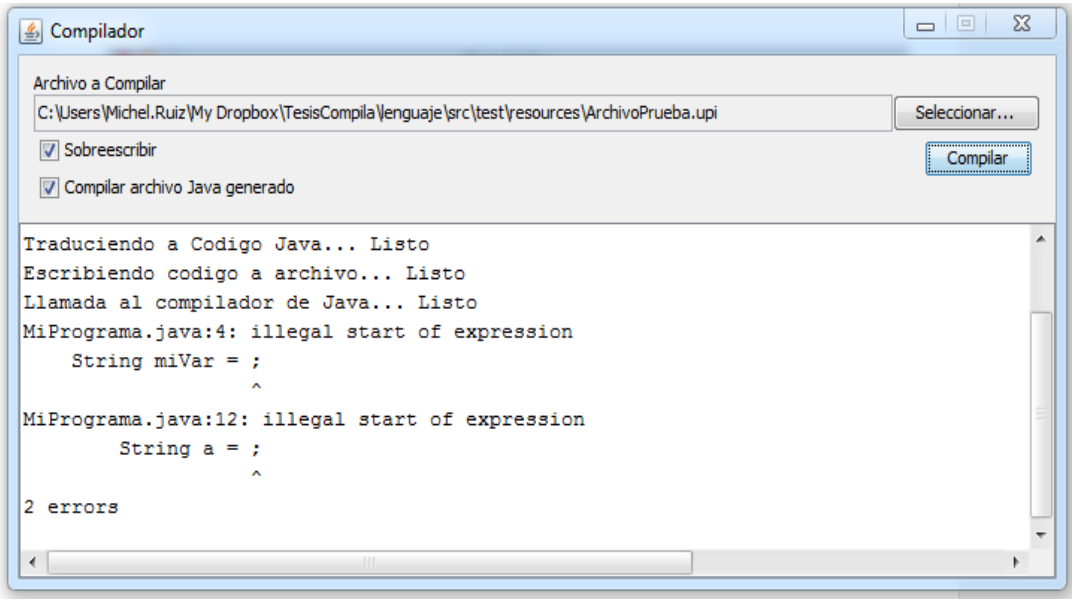
En Windows 7

- El boton “Seleccionar” permite indicar el archivo que contiene el código origen. La diferencia que se presenta con el modo de línea de comando es que solamente se puede compilar un archivo origen a la vez en el modo grafico, mientras que en el modo de línea de comando se pueden indicar multiples archivos
- Al activar la casilla “Sobreescribir”, se indica al compilador que en caso de que exista un archivo Java con el mismo nombre al que será generado, se sobreescribira el contenido del archivo anterior con el contenido del archivo que se generará después de compilar el pseudocódigo. Es equivalente a la opción `--sobreescribir ó -s` en el modo línea de comando.
- Al activar la casilla “Compilar archivo Java generado”, se indica al compilador que se debe llamar al compilador de java (comando javac) después de generar el archivo java para que se genere el archivo class. Es equivalente a la opción `--compilarJava ó -cJ` en el modo línea de comando.

A continuación se muestra una pantalla de ejecución en el modo gráfico.



En caso de presentarse un error en el código, se muestra de la siguiente forma:



CONCLUSIONES

Los objetivos generales indicados en el anteproyecto fueron alcanzados satisfactoriamente. Se ha creado un pseudocódigo cuyas palabras reservadas están definidas en lenguaje español, basado en la programación estructurada, que cumple con las reglas léxicas, sintácticas y semánticas para ser interpretado por un compilador.

Posteriormente a la creación del lenguaje, se definió matemáticamente el diseño del compilador, se identificó el alfabeto, se crearon las expresiones regulares que dieron creación al autómata del compilador, para concluir con la gramática libre del contexto que permitió generar las reglas sintácticas.

Cuando llegó el momento de dar inicio a la codificación del compilador, nos ayudamos de herramientas libres que facilitan la generación del analizador léxico, sintáctico y semántico; aunque tenemos la capacidad para crear éstas fases en un lenguaje de programación teníamos la duda sobre el lenguaje, ya que en caso de ser modificado tendríamos un gran retraso al realizar cambios significativos en alguna fase de compilación. Sin embargo las herramientas no proporcionaron todo, ya que elementos importantes como la tabla de símbolos, el manejador de errores y la generación de código fueron desarrollados en su totalidad por nosotros.

Todos estos pequeños logros dieron origen al compilador, una herramienta diseñada para el estudiante que puede servir de guía al profesor, su desarrollo se inició y concluyó en Java, un lenguaje de programación de alto nivel que si bien sabemos no es tan rápdio como C++ nos permitió su uso en cualquier sistema operativo que soporte la máquina virtual; gracias a esto se realizaron pruebas en diferentes sistemas operativos y se comprobó su correcto funcionamiento.

La definición del lenguaje presentó un reto interesante, primeramente la definición de las palabras reservadas y estructuras, las cuales debían estar en el lenguaje español y además que fueran sencillas de expresar, porque se pretendía que el pseudocódigo fuera simple pero claro a la vez.

El diseño de la comunicación e interacción de los componentes del compilador fue otro punto en el que se debía tener cuidado durante el diseño y desarrollo; la premisa siempre fue la creación de una herramienta que fuera rápida en la realización de sus fases y que la convivencia entre sus módulos (los diferentes analizadores, tabla de simbolos, manejador de errores, entorno gráfico y de línea de comandos) no se diera de una manera forzada, sino que la salida de los módulos se acoplaran perfectamente a los requerimeitnos del siguiente. La sencillez en el diseño y

construcción de los módulos durante la codificación del compilador, nos hizo saber que lo expuesto anteriormente se logró de forma satisfactoria.

Indudablemente es bueno contar con alguna herramienta que facilite el uso de algún lenguaje de programación, un IDE (Integrated Development Environment) por ejemplo. A pesar de que el lenguaje está en español y tiene una sintaxis sencilla, un IDE haría más fácil su uso, lo anterior por medio de distinciones de las palabras reservadas, señalamiento de errores mientras se codifica e identaciones automáticas. La realización de un IDE que soporte el pseudocódigo presenta un reto interesante, que es muy parecido a la realización de un compilador y cuya aplicación fortalecería la herramienta aquí presentada.

En la actualidad los lenguajes orientados a objetos cuentan con una popularidad muy grande entre los programadores y son usados ampliamente para el desarrollo de aplicaciones enterprise. El pseudocódigo descrito en éste trabajo tiene como base la programación estructurada, la cual es útil para aprender a programar, pero indudablemente será necesario enfrentarse algún día con la orientación a objetos, motivo para justificar la creación de un leguaje basado en el que se expone aquí, cuya característica sea que sigue el paradigma de orientación a objetos. De igual forma a lo mencionado previamente, la integración de éste lenguaje orientado a objetos con un IDE sería lo más deseable y ofrecería una herramienta bastante robusta para el ámbito educativo.

El resultado del arduo trabajo realizado en esta tesis nos permitió corroborar la frase de Galvez Rojas donde expresa: *“La construcción de un compilador es una de las tareas más gratas con las que un informático puede encontrase a lo largo de su carrera profesional…”*.

BIBLIOGRAFÍA

Aho, A. V., Ullman, J. D., & Hopcroft, J. E. (1988). *Estructura de Datos y Algoritmos.* (A. Vargas, & J. Lozano, Trads.) Ciudad de México, Distrito Federal, México: Addison-Wesley.

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2008). *Compiladores. Principios, técnicas y herramientas* (Segunda Edición ed.). (L. Cruz Castillo, Ed., & A. V. Romero Elizondo, Trad.) Naucalpan de Juárez, Estado de México, México: Pearson Educación.

Castillo, H. (1 de Marzo de 2003). *Compiladores, Visión histórica del desarrollo de los compiladores.* Recuperado el 22 de Noviembre de 2009, de Benemérita Universidad Autónoma de Puebla: Facultad de Ciencias de la Computación: www.cs.buap.mx/~hilda/notascompila.doc

Galvez Rojas, S., & Mora Mata, M. A. (2005). *Compiladores: Traductores y compiladores con Lex/Yacc, Jflex/Cup y JavaCC.* Malaga, España: Universidad de Malaga.

GPL General Public License. (2009 йил 31-Enero). *Home*. Retrieved 2010 йил 20-Marzo from JFlex - The Fast Scanner Generator for Java: http://jflex.de/

Kelly, D. (1995). *Teoría de Autómatas y Lenguajes Formales* (Vol. I). (M. L. Diez Platas, Trad.) Madrid, España: Prentice Hall.

Oracle Corporation . (24 de Mayo de 2010). *Java SE at a Glance*. Recuperado el 24 de Mayo de 2010, de Oracle: http://java.sun.com/javase/

Scott Hudson, F. F. (28 de Julio de 2003). *CUP Parser Generator for Java*. Recuperado el 24 de Mayo de 2010, de CUP LALR Parser Generator for Java: http://www.cs.princeton.edu/~appel/modern/java/CUP/

Steel, T. B. (9 de May de 1961). A first version of UNCOL. New York, NY, United States of America.

Universidad Tecnológica Nacional. (1 de Agosto de 2008). *Historia de los compiladores.* Recuperado el 22 de Noviembre de 2009, de Facultad Regional Tucuman: Departamento de Sistemas: http://www.frt.utn.edu.ar/sistemas/sintaxis/page28.html

Valverde Andreu, J. (1989). *Compiladores e intérpretes: un enfoque pragmático* (Segunda ed.). Madrid, España: Ediciones Díaz de Santos.

ANEXOS

ARCHIVO DE ESPECIFICACIONES PARA JFLEX

```
package com.upiicsa.lenguaje.core;

import com.upiicsa.lenguaje.util.TipoError;

%% /* ------------------------------------------------------------------------ */


%class AnalizadorLexico
%unicode
%public
%cup
%line

%{
    private ManejadorErrores manejadorErrores;

    public void setManejadorErrores(ManejadorErrores manejadorErrores) {
        this.manejadorErrores = manejadorErrores;
    }

    private Token simbolo(int tipo) {
        return new Token(tipo, yyline + 1);
    }

    private Token simbolo(int tipo, Object value) {
        return new Token(tipo, yyline + 1, value);
    }
%}

%init{
%init}

    /* Caracteres */
        TerminadorLinea           =   \r | \n | \r\n
        Caracter                  =   [^\r\n]
        Espacio                   =   [ \t\f]

    /* Comentario */
        Comentario                =   {ComentarioMultilinea} | {ComentarioLinea}
        ComentarioMultilinea      =   "/*" [^*] ~"*/" | "/*" "*"+ "/"
        ComentarioLinea           =   "//" {Caracter}*

    /* Identificador */
        Identificador             =   [:jletter:][:jletterdigit:]*
```

```
    /* Literales Numericas Enteras */
        NumeroEntero              =   0 | [1-9][0-9]*

    /* Literales Numericas Flotantes */
        Flot1                     =   [0-9]+ \. [0-9]*
        Flot2                     =   \. [0-9]+
        Flot3                     =   [0-9]+
        Exponente                 =   [eE] [+-]? [0-9]+
        NumeroFlotante            =   ({Flot1}|{Flot2}|{Flot3}) {Exponente}?

    /* Literal Cadena De Texto */
        Texto                     =   "\"" ( [^\n\r\"\\] | ("\\" (
["n","t","b","r","f","\\","'","\""] )))* "\""


%% /* ----------------------------------------------------------------------
--------------- */


    <YYINITIAL>{

        /* Palabras Reservadas - Programa */
            "INICIO"        { return simbolo(Simbolos.INICIOPROGRAMA, yytext());
}
            "FIN"          { return simbolo(Simbolos.FINPROGRAMA, yytext()); }
            "PRINCIPAL"    { return simbolo(Simbolos.PRINCIPAL, yytext()); }
            "ENTRADA"      { return simbolo(Simbolos.ENTRADA, yytext()); }
            "IMPRIME"      { return simbolo(Simbolos.IMPRIME, yytext()); }

        /* Palabras Reservadas - Subrutinas */
            "procedimiento"{ return simbolo(Simbolos.PROCEDIMIENTO, yytext()); }
            "regresar"     { return simbolo(Simbolos.REGRESAR, yytext()); }

        /* Palabras Reservadas - Bloques de Instrucciones */
            "inicio"       { return simbolo(Simbolos.INICIO, yytext()); }
            "fin"          { return simbolo(Simbolos.FIN, yytext()); }

        /* Palabras Reservadas - Tipos de Datos */
            "entero"       { return simbolo(Simbolos.ENTERO, yytext()); }
            "texto"        { return simbolo(Simbolos.TEXTO, yytext()); }
            "flot"         { return simbolo(Simbolos.FLOTANTE, yytext()); }
            "bool"         { return simbolo(Simbolos.BOOL, yytext()); }

        /* Palabras Reservadas - Sentencias Selectivas */
            "si"           { return simbolo(Simbolos.SI, yytext()); }
            "entonces"     { return simbolo(Simbolos.ENTONCES, yytext()); }
            "sino"         { return simbolo(Simbolos.SINO, yytext()); }
            "en_caso"      { return simbolo(Simbolos.ENCASO, yytext()); }
```

```
    "opc"          { return simbolo(Simbolos.OPC, yytext()); }
    "otro"         { return simbolo(Simbolos.OTRO, yytext()); }

/* Palabras Reservadas - Sentencias Repetitivas */
    "mientras"     { return simbolo(Simbolos.MIENTRAS, yytext()); }
    "hacer"        { return simbolo(Simbolos.HACER, yytext()); }
    "repetir"      { return simbolo(Simbolos.REPETIR, yytext()); }
    "hasta"        { return simbolo(Simbolos.HASTA, yytext()); }
    "para"         { return simbolo(Simbolos.PARA, yytext()); }

/* Palabras Reservadas - Literal Cadena para Nueva Linea */
    "NL"           { return simbolo(Simbolos.NL, yytext()); }

/* Palabras Reservadas - Literales Booleanas */
    "cierto"       { return simbolo(Simbolos.CIERTO, yytext()); }
    "falso"        { return simbolo(Simbolos.FALSO, yytext()); }

/* Simbolos */
    "("            { return simbolo(Simbolos.PARENI, yytext()); }
    ")"            { return simbolo(Simbolos.PAREND, yytext()); }
    "{"            { return simbolo(Simbolos.LLAVEI, yytext()); }
    "}"            { return simbolo(Simbolos.LLAVED, yytext()); }
    "["            { return simbolo(Simbolos.CORCHI, yytext()); }
    "]"            { return simbolo(Simbolos.CORCHD, yytext()); }
    ","            { return simbolo(Simbolos.COMA, yytext()); }
    ":"            { return simbolo(Simbolos.DOSPUNTOS, yytext()); }
    ";"            { return simbolo(Simbolos.PUNTOCOMA, yytext()); }

/* Operadores Aritmeticos */
    "="            { return simbolo(Simbolos.IGU, yytext()); }
    "++"           { return simbolo(Simbolos.MASMAS, yytext()); }
    "--"           { return simbolo(Simbolos.MENMEN, yytext()); }
    "+"            { return simbolo(Simbolos.MAS, yytext()); }
    "-"            { return simbolo(Simbolos.MEN, yytext()); }
    "*"            { return simbolo(Simbolos.MUL, yytext()); }
    "/"            { return simbolo(Simbolos.DIV, yytext()); }
    ".mod."        { return simbolo(Simbolos.MOD, yytext()); }
    "+="           { return simbolo(Simbolos.MASIGU, yytext()); }
    "-="           { return simbolo(Simbolos.MENIGU, yytext()); }
    "*="           { return simbolo(Simbolos.MULIGU, yytext()); }
    "/="           { return simbolo(Simbolos.DIVIGU, yytext()); }
    ".mod.="       { return simbolo(Simbolos.MODIGU, yytext()); }

/* Operadores Relacionales */
    ">"            { return simbolo(Simbolos.MAYQ, yytext()); }
    "<"            { return simbolo(Simbolos.MENQ, yytext()); }
    "!"            { return simbolo(Simbolos.NO, yytext()); }
    "=="           { return simbolo(Simbolos.IGUIGU, yytext()); }
```

```
            ">="            { return simbolo(Simbolos.MAYQIGU, yytext()); }
            "<="            { return simbolo(Simbolos.MENQIGU, yytext()); }
            "!="            { return simbolo(Simbolos.NOIGU, yytext()); }
            ".y."           { return simbolo(Simbolos.Y, yytext()); }
            ".o."           { return simbolo(Simbolos.O, yytext()); }

        /* Operadores Cadenas */
            "."             { return simbolo(Simbolos.PUNTO, yytext()); }

        /* Identificadores */
            {Identificador}{ return simbolo(Simbolos.IDENTIFICADOR, yytext()); }

        /* Numeros */
            {NumeroEntero} { return simbolo(Simbolos.LITENTERO, new
Integer(yytext())); }
            {NumeroFlotante} { return simbolo(Simbolos.LITFLOTANTE, new
Float(yytext())); }

        /* Texto */
            {Texto}         { return simbolo(Simbolos.LITTEXTO, yytext()); }

        /* Fin de Linea */
            {TerminadorLinea} { return simbolo(Simbolos.FINLINEA, "<FIN DE
LINEA>"); }

        /* Comentarios */
            {Comentario}    { /* No se procesa */ }

        /* Espacios */
            {Espacio}       { /* No se procesa */ }

    }

        /* Error */
            .|\n           { Token error = simbolo(Simbolos.ILEGAL, yytext());
                             StringBuilder msjError = new
StringBuilder(TipoError.CARACTER_ILEGAL.getMensaje());
                             msjError.append(": ");
                             msjError.append(error.value.toString());
                             manejadorErrores.notificarError(msjError.toString(),
error.linea); }

        /* Fin del archivo */
            <<EOF>>         { return simbolo(Simbolos.EOF, "<EOF>"); }
```

ARCHIVO DE ESPECIFICACIONES PARA CUP

```
package com.upiicsa.lenguaje.core;


import com.upiicsa.lenguaje.core.ast.*;

import com.upiicsa.lenguaje.core.tipos.TablaTipos;
import com.upiicsa.lenguaje.core.tipos.Tipo;
import com.upiicsa.lenguaje.core.tipos.TipoArreglo;

import com.upiicsa.lenguaje.util.TipoError;

import java_cup.runtime.Symbol;

import java.util.List;
import java.util.LinkedList;


parser code  {:
    private ManejadorErrores manejadorErrores;

    public void setManejadorErrores(ManejadorErrores manejadorErrores) {
        this.manejadorErrores = manejadorErrores;
    }

    @Override
    public void report_error(String message, Object info) {
        Token error = (Token) info;
        StringBuilder msjError = new
StringBuilder(TipoError.ERROR_SINTAXIS.getMensaje());
        msjError.append(", encontrado cerca de: ");
        msjError.append(error.value.toString());
        manejadorErrores.notificarErrorGrave(msjError.toString(), error.linea);
    }

    @Override
    public void report_fatal_error(String message, Object info) {
        done_parsing();
        report_error(message, info);
    }

    @Override
    public void syntax_error(Symbol cur_token) {
        report_error(null, cur_token);
    }
```

```
    @Override
    public void unrecovered_syntax_error(Symbol cur_token) {
        report_fatal_error(null, cur_token);
    }
:};


/* ---------------------------------------------------------------------- */
/* Terminales
*/
/* ---------------------------------------------------------------------- */

    /* Palabras Reservadas - Programa */
        terminal INICIOPROGRAMA;
        terminal FINPROGRAMA;
        terminal PRINCIPAL;
        terminal ENTRADA;
        terminal IMPRIME;

    /* Palabras Reservadas - Subrutinas */
        terminal PROCEDIMIENTO;
        terminal REGRESAR;

    /* Palabras Reservadas - Bloques de Instrucciones */
        terminal INICIO;
        terminal FIN;

    /* Palabras Reservadas - Tipos de Datos */
        terminal ENTERO;
        terminal TEXTO;
        terminal FLOTANTE;
        terminal BOOL;

    /* Palabras Reservadas - Sentencias Selectivas */
        terminal SI;
        terminal ENTONCES;
        terminal SINO;
        terminal ENCASO;
        terminal OPC;
        terminal OTRO;

    /* Palabras Reservadas - Sentencias Repetitivas */
        terminal MIENTRAS;
        terminal HACER;
        terminal REPETIR;
        terminal HASTA;
        terminal PARA;
```

```
/* Palabras Reservadas - Literal Cadena para Nueva Linea */
    terminal String NL;

/* Palabras Reservadas - Literales Booleanas */
    terminal CIERTO;
    terminal FALSO;

/* Simbolos */
    terminal PARENI;
    terminal PAREND;
    terminal LLAVEI;
    terminal LLAVED;
    terminal CORCHI;
    terminal CORCHD;
    terminal COMA;
    terminal DOSPUNTOS;
    terminal PUNTOCOMA;

/* Operadores Aritmeticos */
    terminal IGU;
    terminal MASMAS;
    terminal MENMEN;
    terminal MAS;
    terminal MEN;
    terminal MUL;
    terminal DIV;
    terminal MOD;
    terminal MASIGU;
    terminal MENIGU;
    terminal MULIGU;
    terminal DIVIGU;
    terminal MODIGU;

/* Operadores Relacionales */
    terminal MAYQ;
    terminal MENQ;
    terminal NO;
    terminal IGUIGU;
    terminal MAYQIGU;
    terminal MENQIGU;
    terminal NOIGU;
    terminal Y;
    terminal O;

/* Operadores Cadenas */
    terminal PUNTO;

/* Identificadores */
```

```
        terminal String IDENTIFICADOR;

    /* Numeros */
        terminal Integer LITENTERO;
        terminal Float LITFLOTANTE;

    /* Texto */
        terminal String LITTEXTO;

    /* Fin de Linea */
        terminal FINLINEA;

    /* Error */
        terminal ILEGAL;


/* ------------------------------------------------------------------------ */
/* No Terminales
*/
/* ------------------------------------------------------------------------ */

    /* Inicio de la Gramatica */
        non terminal Programa programa;

    /* Fin de Instruccion */
        non terminal fin_instruccion_nl;
        non terminal fin_instruccion;

    /* Literales */
        non terminal Expresion literal;

    /* Tipos, Valores y Variables */
        non terminal Tipo tipo;
        non terminal Tipo tipos_basicos;
        non terminal Tipo tipo_arreglo;

    /* Declaracion de Variables Globales */
        non terminal List<Declaracion> declaraciones_var_global_opc;
        non terminal List<Declaracion> declaraciones_var_global;
        non terminal List<Declaracion> declaracion_var_global;
        non terminal SentenciaDeclaraciones declaraciones_var_basicos;
        non terminal Declaracion declaracion_var_basicos;
        non terminal SentenciaDeclaraciones declaraciones_var_arreglo;
        non terminal DeclaracionArreglo declaracion_var_arreglo;

    /* Arreglos */
        non terminal List<Expresion> dimensiones_arreglo;
        non terminal ExpresionArreglo exp_arreglo;
```

```
    non terminal ExpresionArreglo elementos_arreglo;
    non terminal Expresion elemento_arreglo;

/* Declaracion de Metodo */
    non terminal List<Subrutina> declaraciones_subrutina_opc;
    non terminal List<Subrutina> declaraciones_subrutina;
    non terminal Subrutina declaracion_subrutina;
    non terminal Subrutina declaracion_tipo_subrutina;
    non terminal List<Declaracion> lista_paramatros_opc;
    non terminal List<Declaracion> lista_paramatros;
    non terminal Declaracion parametro;

/* Bloques y Sentencias */
    non terminal Bloque bloque;
    non terminal Bloque bloque_sentencias_opc;
    non terminal Bloque bloque_sentencias;
    non terminal Sentencia sentencia;
    non terminal SentenciaDeclaraciones sen_dec_var_local;
    non terminal SentenciaDeclaraciones dec_var_local;
    non terminal Sentencia sen_no_short_if;
    non terminal Sentencia sen_sin_subsen_final;
    non terminal Sentencia sen_expresion;
    non terminal Sentencia sen_exp;
    non terminal SentenciaIf sen_if_then;
    non terminal SentenciaIf sen_if_then_else;
    non terminal SentenciaIf sen_if_then_else_no_short_if;
    non terminal SentenciaSwitch sen_switch;
    non terminal List<SentenciaCase> bloque_switch;
    non terminal List<SentenciaCase> sen_grupos_bloque_switch;
    non terminal SentenciaCase sen_grupo_bloque_switch;
    non terminal List<Expresion> etiquetas_switch;
    non terminal Expresion etiqueta_switch;
    non terminal SentenciaWhile sen_while;
    non terminal SentenciaWhile sen_while_no_short_if;
    non terminal SentenciaDo sen_do;
    non terminal SentenciaFor sen_for;
    non terminal SentenciaFor sen_for_no_short_if;
    non terminal List<Sentencia> ini_for;
    non terminal List<Sentencia> sen_lista_expresiones;
    non terminal SentenciaReturn sen_return;

/* Expresiones */
    non terminal Expresion primario;
    non terminal List<Expresion> lista_argumentos_opc;
    non terminal List<Expresion> lista_argumentos;
    non terminal ExpresionLlamadaSubrutina invocacion_metodo;
    non terminal SentenciaLectura invocacion_entrada;
    non terminal List<ExpresionVariable> variables_entrada;
```

```
        non terminal SentenciaEscritura invocacion_imprime;
        non terminal List<Expresion> variables_imprime;
        non terminal Expresion variable_imprime;
        non terminal ExpresionVariableArreglo acceso_arreglo;
        non terminal Expresion expresion_postfix;
        non terminal Expresion exp_postincremento;
        non terminal Expresion exp_postdecremento;
        non terminal Expresion expresion_unaria;
        non terminal Expresion expresion_unaria_no_mas_menos;
        non terminal Expresion expresion_multiplicativa;
        non terminal Expresion expresion_aditiva;
        non terminal Expresion expresion_relacional;
        non terminal Expresion expresion_igualdad;
        non terminal Expresion expresion_condicional_and;
        non terminal Expresion expresion_condicional_or;
        non terminal Expresion expresion_asignacion;
        non terminal SentenciaAsignacion asignacion;
        non terminal ExpresionVariable lado_izquierdo;
        non terminal Operador operador_asignacion;
        non terminal Expresion expresion;
        non terminal Expresion expresion_constante;



/* ---------------------------------------------------------------------- */
/* Gramatica
*/
/* ---------------------------------------------------------------------- */

start with programa;

programa                        ::=     IDENTIFICADOR:id INICIOPROGRAMA
fin_instruccion_nl declaraciones_var_global_opc:lstVarGlob
declaraciones_subrutina_opc:lstSub PRINCIPAL fin_instruccion_nl bloque:senPrin
FINPROGRAMA fin_instruccion
                                        {: RESULT = new Programa(id, lstVarGlob,
lstSub, senPrin); :}
                                |       error
                                ;

fin_instruccion_nl              ::=     FINLINEA
                                |       fin_instruccion_nl FINLINEA
                                ;

fin_instruccion                 ::=
                                |       fin_instruccion_nl
                                ;
```

```
literal                         ::=     LITENTERO:litEnt
                                        {: RESULT = new
ExpresionEntero(litEnt.intValue());
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                |       LITFLOTANTE:litFlot
                                        {: RESULT = new
ExpresionFlotante(litFlot.floatValue());
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                |       CIERTO
                                        {: RESULT = new ExpresionBooleano(true);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                |       FALSO
                                        {: RESULT = new
ExpresionBooleano(false);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                |       LITTEXTO:litTxt
                                        {: RESULT = new ExpresionTexto(litTxt);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                ;


tipo                            ::=     tipos_basicos:tipo
                                        {: RESULT = tipo; :}
                                |       tipo_arreglo:tipo
                                        {: RESULT = tipo; :}
                                ;

tipos_basicos                   ::=     ENTERO
                                        {: RESULT = TablaTipos.tipoEntero; :}
                                |       FLOTANTE
                                        {: RESULT = TablaTipos.tipoFlotante; :}
                                |       BOOL
                                        {: RESULT = TablaTipos.tipoBooleano; :}
                                |       TEXTO
                                        {: RESULT = TablaTipos.tipoTexto; :}
                                ;

tipo_arreglo                    ::=     CORCHI CORCHD tipos_basicos:tipo
                                        {: RESULT =
TablaTipos.tipoArreglo(tipo); :}
                                |       CORCHI CORCHD tipo_arreglo:tipo
                                        {: RESULT =
TablaTipos.tipoArreglo(tipo); :}
```

```
                                ;


declaraciones_var_global_opc    ::=
                                        {: RESULT = new
LinkedList<Declaracion>(); :}
                                |       declaraciones_var_global:lstVarGlob
                                        {: RESULT = lstVarGlob; :}
                                ;

declaraciones_var_global        ::=     declaracion_var_global:varGlob
                                        {: RESULT = varGlob; :}
                                |       declaraciones_var_global:lstVarGlob
declaracion_var_global:varGlob
                                        {: for (Declaracion d : varGlob)
                                               lstVarGlob.add(d);
                                           RESULT = lstVarGlob; :}
                                ;

declaracion_var_global          ::=     tipos_basicos:tipo
declaraciones_var_basicos:lstDec fin_instruccion_nl
                                        {: for (Declaracion d :
lstDec.declaraciones) {
                                               d.tipo = tipo;
                                               d.revisarValorInicial();
                                           }
                                           RESULT = lstDec.declaraciones; :}
                                |       tipo_arreglo:tipo
declaraciones_var_arreglo:lstDec fin_instruccion_nl
                                        {: for (Declaracion d :
lstDec.declaraciones)
                                               d.tipo = tipo;
                                           RESULT = lstDec.declaraciones; :}
                                ;

declaraciones_var_basicos       ::=     declaracion_var_basicos:dec
                                        {: RESULT = new
SentenciaDeclaraciones(dec); :}
                                |       declaraciones_var_basicos:lstDec COMA
declaracion_var_basicos:dec
                                        {: lstDec.agregarDeclaracion(dec);
                                           RESULT = lstDec; :}
                                ;

declaracion_var_basicos         ::=     IDENTIFICADOR:id
                                        {: RESULT = new Declaracion(id);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
```

```
                                |       IDENTIFICADOR:id IGU expresion:exp
                                        {: RESULT = new Declaracion(id, exp);
                                           RESULT.linea = exp.linea; :}
                                ;

declaraciones_var_arreglo       ::=     declaracion_var_arreglo:dec
                                        {: RESULT = new
SentenciaDeclaraciones(dec); :}
                                |       declaraciones_var_arreglo:lstDec COMA
declaracion_var_arreglo:dec
                                        {: lstDec.agregarDeclaracion(dec);
                                           RESULT = lstDec; :}
                                ;

declaracion_var_arreglo         ::=     IDENTIFICADOR:id dimensiones_arreglo:dim
                                        {: RESULT = new DeclaracionArreglo(id,
dim.toArray(new Expresion[dim.size()]));
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.elementAt(CUP$AnalizadorSintactico$top-1)).linea;
:}
                                |       IDENTIFICADOR:id IGU exp_arreglo:exp
                                        {: RESULT = new DeclaracionArreglo(id,
exp);
                                           RESULT.linea = exp.linea; :}
                                ;


dimensiones_arreglo             ::=     CORCHI expresion:exp CORCHD
                                        {: RESULT = new LinkedList<Expresion>();
                                           RESULT.add(exp); :}
                                |       dimensiones_arreglo:lstDim CORCHI
expresion:exp CORCHD
                                        {: lstDim.add(exp);
                                           RESULT = lstDim; :}
                                ;

exp_arreglo                     ::=     LLAVEI elementos_arreglo:elem LLAVED
                                        {: RESULT = elem; :}
                                ;

elementos_arreglo               ::=     elemento_arreglo:exp
                                        {: RESULT = new ExpresionArreglo(exp);
                                           RESULT.linea = exp.linea; :}
                                |       elementos_arreglo:elem COMA
elemento_arreglo:exp
                                        {: elem.agregarExpresion(exp);
                                           RESULT = elem; :}
                                ;
```

```
elemento_arreglo                ::=     expresion:exp
                                        {: RESULT = exp; :}
                                |       exp_arreglo:exp
                                        {: RESULT = exp; :}
                                ;


declaraciones_subrutina_opc     ::=
                                        {: RESULT = new LinkedList<Subrutina>();
:}
                                |       declaraciones_subrutina:lstSub
                                        {: RESULT = lstSub; :}
                                ;

declaraciones_subrutina         ::=     declaracion_subrutina:sub
                                        {: RESULT = new LinkedList<Subrutina>();
                                           RESULT.add(sub); :}
                                |       declaraciones_subrutina:lstSub
declaracion_subrutina:sub
                                        {: lstSub.add(sub);
                                           RESULT = lstSub; :}
                                ;

declaracion_subrutina           ::=     PROCEDIMIENTO
declaracion_tipo_subrutina:decSub
                                        {: RESULT = decSub; :}
                                ;

declaracion_tipo_subrutina      ::=     IDENTIFICADOR:id PARENI
lista_paramatros_opc:lstParam PAREND fin_instruccion_nl bloque:bl
                                        {: RESULT = new Subrutina(id, lstParam,
bl, TablaTipos.tipoVoid);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.elementAt(CUP$AnalizadorSintactico$top-5)).linea;
:}
                                |       tipo:t IDENTIFICADOR:id PARENI
lista_paramatros_opc:lstParam PAREND fin_instruccion_nl INICIO
fin_instruccion_nl bloque_sentencias_opc:bl sen_return:sen FIN
fin_instruccion_nl
                                        {: bl.agregarSentencia(sen);
                                           RESULT = new Subrutina(id, lstParam,
bl, t);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.elementAt(CUP$AnalizadorSintactico$top-
10)).linea; :}
                                ;
```

```
lista_paramatros_opc            ::=
                                        {: RESULT = new
LinkedList<Declaracion>(); :}
                                |       lista_paramatros:lstParam
                                        {: RESULT = lstParam; :}
                                ;

lista_paramatros                ::=     parametro:param
                                        {: RESULT = new
LinkedList<Declaracion>();
                                           RESULT.add(param); :}
                                |       lista_paramatros:lstParam COMA
parametro:param
                                        {: lstParam.add(param);
                                           RESULT = lstParam; :}
                                ;

parametro                       ::=     tipo:t IDENTIFICADOR:id
                                        {: if (t instanceof TipoArreglo)
                                               RESULT = new
DeclaracionArreglo(id);
                                           else
                                               RESULT = new Declaracion(id);
                                           RESULT.tipo = t;
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                ;



bloque                          ::=     INICIO fin_instruccion_nl
bloque_sentencias_opc:blqSent FIN fin_instruccion_nl
                                        {: RESULT = blqSent; :}
                                ;

bloque_sentencias_opc           ::=
                                        {: RESULT = new Bloque(); :}
                                |       bloque_sentencias:blqSent
                                        {: RESULT = blqSent; :}
                                ;

bloque_sentencias               ::=     sentencia:sent
                                        {: RESULT = new Bloque(sent); :}
                                |       bloque_sentencias:blqSent sentencia:sent
                                        {: blqSent.agregarSentencia(sent);
                                           RESULT = blqSent; :}
                                ;
```

```
sentencia                       ::=     sen_dec_var_local:lstDecVar
                                        {: RESULT = lstDecVar; :}
                                |       sen_sin_subsen_final:sent
                                        {: RESULT = sent; :}
                                |       sen_if_then:sent
                                        {: RESULT = sent; :}
                                |       sen_if_then_else:sent
                                        {: RESULT = sent; :}
                                |       sen_while:sent
                                        {: RESULT = sent; :}
                                |       sen_for:sent
                                        {: RESULT = sent; :}
                                |       error
                                ;

sen_dec_var_local               ::=     dec_var_local:lstDecVar
fin_instruccion_nl
                                        {: RESULT = lstDecVar; :}
                                ;

dec_var_local                   ::=     tipos_basicos:tipo
declaraciones_var_basicos:lstDec
                                        {: for (Declaracion d :
lstDec.declaraciones) {
                                               d.tipo = tipo;
                                               d.revisarValorInicial();
                                           }
                                           RESULT = lstDec; :}
                                |       tipo_arreglo:tipo
declaraciones_var_arreglo:lstDec
                                        {: for (Declaracion d :
lstDec.declaraciones)
                                               d.tipo = tipo;
                                           RESULT = lstDec; :}
                                ;

sen_no_short_if                 ::=     sen_sin_subsen_final:sent
                                        {: RESULT = sent; :}
                                |       sen_if_then_else_no_short_if:sent
                                        {: RESULT = sent; :}
                                |       sen_while_no_short_if:sent
                                        {: RESULT = sent; :}
                                |       sen_for_no_short_if:sent
                                        {: RESULT = sent; :}
                                ;

sen_sin_subsen_final            ::=     bloque:bl
                                        {: RESULT = bl; :}
```

```
                                |       sen_expresion:sent
                                        {: RESULT = sent; :}
                                |       sen_switch:sent
                                        {: RESULT = sent; :}
                                |       sen_do:sent
                                        {: RESULT = sent; :}
                                ;

sen_expresion                   ::=     sen_exp:sent fin_instruccion_nl
                                        {: RESULT = sent; :}
                                |       invocacion_entrada:sent
fin_instruccion_nl
                                        {: RESULT = sent; :}
                                |       invocacion_imprime:sent
fin_instruccion_nl
                                        {: RESULT = sent; :}
                                ;

sen_exp                         ::=     asignacion:sent
                                        {: RESULT = sent; :}
                                |       exp_postincremento:exp
                                        {: RESULT = new SentenciaExpresion(exp);
:}
                                |       exp_postdecremento:exp
                                        {: RESULT = new SentenciaExpresion(exp);
:}
                                |       invocacion_metodo:exp
                                        {: RESULT = new SentenciaExpresion(exp);
:}
                                ;

sen_if_then                     ::=     SI PARENI expresion:exp PAREND ENTONCES
fin_instruccion_nl sentencia:sentIf
                                        {: RESULT = new SentenciaIf(exp,
sentIf); :}
                                ;

sen_if_then_else                ::=     SI PARENI expresion:exp PAREND ENTONCES
fin_instruccion_nl sen_no_short_if:sentIf SINO fin_instruccion_nl
sentencia:sentElse
                                        {: RESULT = new SentenciaIf(exp, sentIf,
sentElse); :}
                                ;

sen_if_then_else_no_short_if    ::=     SI PARENI expresion:exp PAREND ENTONCES
fin_instruccion_nl sen_no_short_if:sentIf SINO fin_instruccion_nl
sen_no_short_if:sentElse
```

```
                                        {: RESULT = new SentenciaIf(exp, sentIf,
sentElse); :}
                                ;

sen_switch                      ::=     ENCASO PARENI expresion:exp PAREND
fin_instruccion_nl bloque_switch:lstCase
                                        {: RESULT = new SentenciaSwitch(exp,
lstCase); :}
                                ;


bloque_switch                   ::=     INICIO fin_instruccion_nl
sen_grupos_bloque_switch:lstCase FIN fin_instruccion_nl
                                        {: RESULT = lstCase; :}
                                |       INICIO fin_instruccion_nl FIN
fin_instruccion_nl
                                        {: RESULT = new
LinkedList<SentenciaCase>(); :}
                                ;

sen_grupos_bloque_switch        ::=     sen_grupo_bloque_switch:senCase
                                        {: RESULT = new
LinkedList<SentenciaCase>();
                                           RESULT.add(senCase); :}
                                |       sen_grupos_bloque_switch:lstCase
sen_grupo_bloque_switch:senCase
                                        {: lstCase.add(senCase);
                                           RESULT = lstCase; :}
                                ;

sen_grupo_bloque_switch         ::=     etiquetas_switch:lstEtiq bloque:bl
                                        {: RESULT = new SentenciaCase(lstEtiq,
bl); :}
                                ;

etiquetas_switch                ::=     etiqueta_switch:etiq
                                        {: RESULT = new LinkedList<Expresion>();
                                           RESULT.add(etiq); :}
                                |       etiquetas_switch:lstEtiq
etiqueta_switch:etiq
                                        {: lstEtiq.add(etiq);
                                           RESULT = lstEtiq; :}
                                ;

etiqueta_switch                 ::=     OPC expresion_constante:exp DOSPUNTOS
                                        {: RESULT = exp; :}
                                |       OTRO DOSPUNTOS
                                        {: RESULT = new ExpresionDefault();
```

```
                                        RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                ;

sen_while                       ::=     MIENTRAS PARENI expresion:exp PAREND
HACER fin_instruccion_nl sentencia:sent
                                        {: RESULT = new SentenciaWhile(exp,
sent); :}
                                ;

sen_while_no_short_if           ::=     MIENTRAS PARENI expresion:exp PAREND
HACER fin_instruccion_nl sen_no_short_if:sent
                                        {: RESULT = new SentenciaWhile(exp,
sent); :}
                                ;

sen_do                          ::=     REPETIR fin_instruccion_nl
bloque_sentencias_opc:sent HASTA PARENI expresion:exp PAREND fin_instruccion_nl
                                        {: RESULT = new SentenciaDo(exp, sent);
:}
                                ;

sen_for                         ::=     PARA PARENI ini_for:sentIni PUNTOCOMA
expresion:expParo PUNTOCOMA sen_lista_expresiones:sentAct PAREND
fin_instruccion_nl sentencia:sent
                                        {: RESULT = new SentenciaFor(sentIni,
expParo, sentAct, sent); :}
                                ;

sen_for_no_short_if             ::=     PARA PARENI ini_for:sentIni PUNTOCOMA
expresion:expParo PUNTOCOMA sen_lista_expresiones:sentAct PAREND
fin_instruccion_nl sen_no_short_if:sent
                                        {: RESULT = new SentenciaFor(sentIni,
expParo, sentAct, sent); :}
                                ;

ini_for                         ::=     sen_lista_expresiones:lstSent
                                        {: RESULT = lstSent; :}
                                |       dec_var_local:sent
                                        {: RESULT = new LinkedList<Sentencia>();
                                           RESULT.add(sent); :}
                                ;

sen_lista_expresiones           ::=     sen_exp:sent
                                        {: RESULT = new LinkedList<Sentencia>();
                                           RESULT.add(sent); :}
                                |       sen_lista_expresiones:lstSent COMA
sen_exp:sent
```

```
                                        {: lstSent.add(sent);
                                           RESULT = lstSent; :}
                                    ;

sen_return                          ::=     REGRESAR expresion:exp
fin_instruccion_nl
                                        {: RESULT = new SentenciaReturn(exp); :}
                                    ;


primario                            ::=     literal:lit
                                        {: RESULT = lit; :}
                                    |       PARENI expresion:exp PAREND
                                        {: RESULT = exp; :}
                                    |       invocacion_metodo:exp
                                        {: RESULT = exp; :}
                                    |       acceso_arreglo:exp
                                        {: RESULT = exp; :}
                                    ;

lista_argumentos_opc                ::=
                                        {: RESULT = new LinkedList<Expresion>();
:}
                                    |       lista_argumentos:lstArg
                                        {: RESULT = lstArg; :}
                                    ;

lista_argumentos                    ::=     expresion:exp
                                        {: RESULT = new LinkedList<Expresion>();
                                           RESULT.add(exp); :}
                                    |       lista_argumentos:lstArg COMA
expresion:exp
                                        {: lstArg.add(exp);
                                           RESULT = lstArg; :}
                                    ;

invocacion_metodo                   ::=     IDENTIFICADOR:id PARENI
lista_argumentos_opc:lstArg PAREND
                                        {: RESULT = new
ExpresionLlamadaSubrutina(id, lstArg);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                    ;

invocacion_entrada                  ::=     ENTRADA variables_entrada:lstVar
                                        {: RESULT = new
SentenciaLectura(lstVar); :}
                                    ;
```

```
variables_entrada               ::=     IDENTIFICADOR:id
                                        {: RESULT = new
LinkedList<ExpresionVariable>();
                                           ExpresionVariable exp = new
ExpresionVariable(id);
                                           exp.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea;
                                           RESULT.add(exp); :}
                                |       variables_entrada:lstVar COMA
IDENTIFICADOR:id
                                        {: ExpresionVariable exp = new
ExpresionVariable(id);
                                           exp.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea;
                                           lstVar.add(exp);
                                           RESULT = lstVar; :}
                                ;

invocacion_imprime              ::=     IMPRIME variables_imprime:lstExp
                                        {: RESULT = new
SentenciaEscritura(lstExp); :}
                                ;

variables_imprime               ::=     variable_imprime:exp
                                        {: RESULT = new LinkedList<Expresion>();
                                           RESULT.add(exp); :}
                                |       variables_imprime:lstExp COMA
variable_imprime:exp
                                        {: lstExp.add(exp);
                                           RESULT = lstExp; :}
                                ;

variable_imprime                ::=     expresion:exp
                                        {: RESULT = exp; :}
                                |       NL:litTxt
                                        {: RESULT = new ExpresionTexto(litTxt);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                ;

acceso_arreglo                  ::=     IDENTIFICADOR:id dimensiones_arreglo:dim
                                        {: RESULT = new
ExpresionVariableArreglo(id, dim);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.elementAt(CUP$AnalizadorSintactico$top-1)).linea;
:}
                                ;
```

```
expresion_postfix               ::=     primario:pri
                                        {: RESULT = pri; :}
                                |      IDENTIFICADOR:id
                                        {: RESULT = new ExpresionVariable(id);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                |       exp_postincremento:exp
                                        {: RESULT = exp; :}
                                |       exp_postdecremento:exp
                                        {: RESULT = exp; :}
                                ;

exp_postincremento              ::=     expresion_postfix:exp MASMAS
                                        {: RESULT = new ExpresionOpUnaria(exp,
Operador.MASMAS);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                ;

exp_postdecremento              ::=     expresion_postfix:exp MENMEN
                                        {: RESULT = new ExpresionOpUnaria(exp,
Operador.MENMEN);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                ;

expresion_unaria                ::=          MAS expresion_unaria:exp
                                        {: RESULT = new ExpresionOpUnaria(exp,
Operador.MAS);
                                           RESULT.linea = exp.linea; :}
                                |      MEN expresion_unaria:exp
                                        {: RESULT = new ExpresionOpUnaria(exp,
Operador.MEN);
                                           RESULT.linea = exp.linea; :}
                                |      expresion_unaria_no_mas_menos:exp
                                        {: RESULT = exp; :}
                                ;



expresion_unaria_no_mas_menos   ::=     expresion_postfix:exp
                                        {: RESULT = exp; :}
                                |      NO expresion_unaria:exp
                                        {: RESULT = new ExpresionOpUnaria(exp,
Operador.NO);
                                           RESULT.linea = exp.linea; :}
                                ;
```

```
expresion_multiplicativa        ::=     expresion_unaria:exp
                                        {: RESULT = exp; :}
                                |       expresion_multiplicativa:exp1 MUL
expresion_unaria:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MUL);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_multiplicativa:exp1 DIV
expresion_unaria:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.DIV);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_multiplicativa:exp1 MOD
expresion_unaria:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MOD);
                                           RESULT.linea = exp2.linea; :}
                                ;

expresion_aditiva               ::=     expresion_multiplicativa:exp
                                        {: RESULT = exp; :}
                                |       expresion_aditiva:exp1 MAS
expresion_multiplicativa:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MAS);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_aditiva:exp1 MEN
expresion_multiplicativa:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MEN);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_aditiva:exp1 PUNTO
expresion_multiplicativa:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.PUNTO);
                                           RESULT.linea = exp2.linea; :}
                                ;

expresion_relacional            ::=     expresion_aditiva:exp
                                        {: RESULT = exp; :}
                                |       expresion_relacional:exp1 MENQ
expresion_aditiva:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MENQ);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_relacional:exp1 MAYQ
expresion_aditiva:exp2
```

```
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MAYQ);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_relacional:exp1 MENQIGU
expresion_aditiva:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MENQIGU);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_relacional:exp1 MAYQIGU
expresion_aditiva:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.MAYQIGU);
                                           RESULT.linea = exp2.linea; :}
                                ;

expresion_igualdad              ::=     expresion_relacional:exp
                                        {: RESULT = exp; :}
                                |       expresion_igualdad:exp1 IGUIGU
expresion_relacional:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.IGUIGU);
                                           RESULT.linea = exp2.linea; :}
                                |       expresion_igualdad:exp1 NOIGU
expresion_relacional:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.NOIGU);
                                           RESULT.linea = exp2.linea; :}
                                ;

expresion_condicional_and       ::=     expresion_igualdad:exp
                                        {: RESULT = exp; :}
                                |       expresion_condicional_and:exp1 Y
expresion_igualdad:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.Y);
                                           RESULT.linea = exp2.linea; :}
                                ;

expresion_condicional_or        ::=     expresion_condicional_and:exp
                                        {: RESULT = exp; :}
                                |       expresion_condicional_or:exp1 O
expresion_condicional_and:exp2
                                        {: RESULT = new ExpresionOpBinaria(exp1,
exp2, Operador.O);
                                           RESULT.linea = exp2.linea; :}
                                ;

expresion_asignacion            ::=     expresion_condicional_or:exp
```

```
                                        {: RESULT = exp; :}
                                ;

asignacion                      ::=    lado_izquierdo:ladIzq
operador_asignacion:op expresion_asignacion:exp
                                        {: if (op.equals(Operador.IGU)) {
                                               RESULT = new
SentenciaAsignacion(ladIzq, exp);
                                               RESULT.linea = exp.linea;
                                           } else {
                                               ExpresionOpBinaria primExp = new
ExpresionOpBinaria(null, null, null);
                                               switch (op) {
                                                   case MULIGU: primExp = new
ExpresionOpBinaria(ladIzq, exp, Operador.MUL); break;
                                                   case DIVIGU: primExp = new
ExpresionOpBinaria(ladIzq, exp, Operador.DIV); break;
                                                   case MODIGU: primExp = new
ExpresionOpBinaria(ladIzq, exp, Operador.MOD); break;
                                                   case MASIGU: primExp = new
ExpresionOpBinaria(ladIzq, exp, Operador.MAS); break;
                                                   case MENIGU: primExp = new
ExpresionOpBinaria(ladIzq, exp, Operador.MEN); break;
                                               }
                                               primExp.linea = exp.linea;
                                               RESULT = new
SentenciaAsignacion(ladIzq, primExp);
                                               RESULT.linea = exp.linea;
                                           }
                                        :}
                                ;

lado_izquierdo                  ::=     IDENTIFICADOR:id
                                        {: RESULT = new ExpresionVariable(id);
                                           RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                |      acceso_arreglo:exp
                                        {: RESULT = exp; :}
                                ;

operador_asignacion             ::=     IGU
                                        {: RESULT = Operador.IGU; :}
                                |      MULIGU
                                        {: RESULT = Operador.MULIGU; :}
                                |      DIVIGU
                                        {: RESULT = Operador.DIVIGU; :}
                                |      MODIGU
                                        {: RESULT = Operador.MODIGU; :}
```

```
                                    |       MASIGU
                                            {: RESULT = Operador.MASIGU; :}
                                    |       MENIGU
                                            {: RESULT = Operador.MENIGU; :}
                                    ;

expresion                           ::=     expresion_asignacion:exp
                                            {: RESULT = exp; :}
                                    |       error
                                    ;

expresion_constante                 ::=     LITENTERO:litEnt
                                            {: RESULT = new
ExpresionEntero(litEnt.intValue());
                                               RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                    |       LITTEXTO:litTxt
                                            {: RESULT = new ExpresionTexto(litTxt);
                                               RESULT.linea = ((Token)
CUP$AnalizadorSintactico$stack.peek()).linea; :}
                                    ;
```