



UNIREMINGTON®
CORPORACIÓN UNIVERSITARIA REMINGTON
RES. 2661 MEN JUNIO 21 DE 1996

COMPILADORES
INGENIERÍA DE SISTEMAS
FACULTAD DE CIENCIAS BÁSICAS E INGENIERÍA

Vicerrectoría de Educación a Distancia y Virtual
2015

CRÉDITOS



El módulo de estudio de la asignatura Compiladores es propiedad de la Corporación Universitaria Remington. Las imágenes fueron tomadas de diferentes fuentes que se relacionan en los derechos de autor y las citas en la bibliografía. El contenido del módulo está protegido por las leyes de derechos de autor que rigen al país.

Este material tiene fines educativos y no puede usarse con propósitos económicos o comerciales.

AUTOR

Luis Fernando Zapata Álvarez

Ingeniero de Sistemas, Especialización en Gerencia Informática, Pedagogía para profesionales, Investigación Holística, Docente Corporación Universitaria Remington Docente del departamento de Antioquia, Investigador de la CUR y el proyecto del cual soy investigador principal "TIC en los procesos de evaluación"
Luis.zapata@remington.edu.co O feingeniosa@yahoo.es

Nota: el autor certificó (de manera verbal o escrita) No haber incurrido en fraude científico, plagio o vicios de autoría; en caso contrario eximió de toda responsabilidad a la Corporación Universitaria Remington, y se declaró como el único responsable.

RESPONSABLES

Jorge Mauricio Sepúlveda Castaño

Decano de la Facultad de Ciencias Básicas e Ingeniería
jsepulveda@uniremington.edu.co

Eduardo Alfredo Castillo Builes

Vicerrector de Educación a Distancia y Virtual
ecastillo@uniremington.edu.co

Francisco Javier Alvarez Gómez

Coordinador CUR-Virtual
falvarez@uniremington.edu.co

GRUPO DE APOYO

Personal de la Unidad de CUR-VIRTUAL

EDICIÓN Y MONTAJE

Primera versión. Febrero de 2011.

Impresión Agosto de 2015.

Derechos Reservados



Esta obra es publicada bajo la licencia Creative Commons. Reconocimiento-No Comercial-Compartir Igual 2.5 Colombia.



TABLA DE CONTENIDO

1.	INTRODUCCIÓN	7
2.	MAPA DEL MÓDULO.....	8
3.	DEFINICION DE UN COMPILADOR Y CONCEPTOS GENERALES	9
3.1.	Definición del compilador	10
3.2.	Fases de un Compilador	11
3.3.	Ejercicios por temas	16
3.4.	Actividad.....	17
4.	ANÁLISIS LÉXICO.....	18
4.1.	Definición de la primera parte del análisis de un compilador	19
4.1.1.	Funciones del analizador léxico:	19
4.1.2.	Expresiones Regulares.....	27
4.1.3.	Ejemplos de Expresiones Regulares	28
4.2.	Ejercicios por temas	33
4.3.	Actividad.....	37
4.3.1.	Primera práctica de compiladores	37
5.	ANÁLISIS SINTÁCTICO	38
5.1.	Teoría de Gramáticas	38
5.1.2.	Teoría de reconocedores del lenguaje(reconocimiento descendente)	45
5.1.3.	Desapile, Retenga.....	47
5.2.	Ejercicios por temas	55
5.2.1.	Ejercicios definición de gramáticas	55
5.3.	Actividad.....	57
5.3.1.	Segunda Práctica de Compiladores (15%).....	57
6.	FASE SÍNTESIS DEL COMPILADOR	59
7.	PISTAS DE APRENDIZAJE	70
8.	GLOSARIO	71
9.	FUENTES.....	72



1. INTRODUCCIÓN

El estudio de los compiladores presenta una visión del fundamento de su construcción y su utilización en la teoría de sistemas, aunque siempre este cobijada por el desarrollo de los lenguajes de programación pues dependen de su aparición y utilización. Un compilador es creado al lado del lenguaje de programación con el objeto primordial de realizar la revisión sobre el código del programador para determinar si este está siendo bien usado (sin errores de sintaxis) y posteriormente encadenar las líneas de código de alto nivel con líneas de código de máquina; que permitan generar un código ejecutable para la computadora.

Se presenta inicialmente un conjunto de conceptos que permiten a los estudiantes adquirir una cultura que facilite la comprensión de sus contenidos y asociarlos el desarrollo posterior del curso. Principalmente se trata de la definición de un compilador dividido en sus dos fases principales y su diferenciación con un simple traductor, también se muestra la importancia de entender el funcionamiento del lenguaje de bajo nivel para poder realizar el equivalente de instrucciones de alto y la obtención del lenguaje de máquina (lenguaje ejecutable).

El curso continua con el estudio de las fases del compilador divididas por etapas, siendo la primera el análisis léxico que se encarga de revisar las líneas del lenguaje por caracteres para obtener los componentes importantes del lenguaje (tokens) y almacenarlos en una tabla denominada de símbolos que va a ser usada posteriormente por la siguiente etapa de análisis. Es en el análisis sintáctico donde se realiza la revisión de la línea solicitando los tokens a la tabla de símbolos y devolviendo un mensaje de línea completa o con errores de acuerdo a lo que se dé. El análisis semántico verifica la utilización de los tipos de acuerdo a su definición de almacenamiento para el programa y es capaz de mostrar algunas inconsistencias en el uso de los tipos determinados.

Después del análisis viene la Síntesis que permite generar códigos de lenguaje binario o de ese tipo de representación que pueden generar programas ejecutables desde un código de alto nivel.

2. MAPA DEL MÓDULO

COMPILADORES

Este módulo está diseñado para estudiantes de ingeniería de sistemas o de profesionalización de ingeniería de sistemas y se refiere a un conocimiento tan necesario y aplicable en el área que tiene que ver con el desarrollo de programas usando lenguajes de alto nivel y la forma como se debe relacionar con códigos que están al nivel de la máquina.

OBJETIVO GENERAL

Desarrollar destrezas en el manejo de la estructura general de construcción de un compilador involucrando y relacionando todas sus fases, desde el análisis hasta la síntesis, usando ese conocimiento en el posterior desarrollo de nuevas herramientas de compilación o en el mejoramiento y optimización de las ya construidas.

OBJETIVOS ESPECÍFICOS

- ◆ Comprender el concepto de compilación y su uso dentro de los sistemas diferenciando los lenguajes de alto nivel con el lenguaje natural y con conocimiento de cómo funcionan los diferentes compiladores actuales (como fue su diseño lógico y físico).
- ◆ Aplicar la teoría de autómatas para el reconocimiento de patrones de los lenguajes, aplicando el conocimiento teórico en la solución de problemas reales partiendo de las prácticas propuestas en clase (realizar toda la fase de análisis en la creación de un compilador).
- ◆ Construir un pequeño reconocedor de las instrucciones de un lenguaje tanto a nivel teórico como práctico, utilizando la teoría de gramáticas de los lenguajes de programación.
- ◆ Relacionar la escritura de instrucciones de alto nivel con su equivalente en bajo nivel y las estructuras básicas de programación de alto nivel con su código correspondiente en bajo nivel.

UNIDAD 1

Introducción a los compiladores, Definición de un compilador y conceptos generales.

UNIDAD 2

Análisis léxico, Definición de Primera parte del análisis de un compilador.

UNIDAD 3

Análisis sintáctico, revisión de líneas del lenguaje para determinar si están bien escritas

UNIDAD 4

Fase de síntesis del compilador, con el objetivo de enlazar el código fuente con el lenguaje ensamblador.

3. DEFINICION DE UN COMPILADOR Y CONCEPTOS GENERALES

Objetivo General

- ◆ Comprender el concepto de compilación y su uso dentro de los sistemas diferenciando los lenguajes de alto nivel con el lenguaje natural y con conocimiento de cómo funcionan los diferentes compiladores actuales (como fue su diseño lógico y físico).

Objetivo específico

- ◆ Aprender a diferenciar los lenguajes de alto nivel con el lenguaje natural

Prueba Inicial

Seleccione la respuesta correcta según su conocimiento previo:

1. Un compilador tiene como objeto:
 - a) Traducir las reglas de un lenguaje dado a otro
 - b) Revisar y corregir errores de escritura de un lenguaje
 - c) Realizar las fases completas de análisis y síntesis a un lenguaje dado, para convertir un código para que sea entendido por la máquina.
 - d) Ninguna de las anteriores.
2. La diferencia entre el compilador y el traductor es:
 - a) El compilador es de propósito más general que el traductor
 - b) El traductor es para lenguajes convencionales
 - c) El compilador detecta errores de sintaxis
 - d) Ninguna de las anteriores
3. La diferencia entre los lenguajes de alto y bajo nivel radica en que:
 - a) El lenguaje de alto nivel maneja más palabras reservadas del lenguaje.
 - b) El lenguaje de bajo nivel solo lo pueden usar los programadores expertos
 - c) El lenguaje de alto nivel fue diseñado por programadores

- d) El lenguaje de bajo nivel utiliza instrucciones que se comunican directamente con los registros de la máquina.
- 4.Cuál es la diferencia entre las fases de análisis y las de síntesis en la construcción de un compilador:
 - a) El análisis se preocupa por la optimización de memoria y generación de código.
 - b) El análisis es la fase previa del cumplidor que revisa las componentes individuales y sintácticas del lenguaje para poder pasar a la construcción del código intermedio.
 - c) No hay análisis sin una síntesis previa dentro del diseño del compilador.
 - d) Ninguna de las anteriores.
- 5. El código producido por el compilador en lenguaje:
 - a) Ensamblador
 - b) Lenguaje Fortran
 - c) Lenguaje de máquina
 - d) Las respuestas a y c son acertadas.

3.1. Definición del compilador

Que es un compilador: Consiste en mirar la solución de un lenguaje de alto nivel a un lenguaje de bajo nivel. El proceso de traducir el programa escrito en lenguaje de alto nivel a un formato ejecutable por un computador se conoce como compilación. O sea, que un compilador es un programa que lee en un lenguaje y traduce a otro lenguaje.

Hay compiladores que generan otro tipo de salida, para ser usada con fines diferentes al lenguaje de máquina. Por ejemplo como lo hace un intérprete que ejecuta las operaciones especificadas por un programa.

Los compiladores presentan diferencias claves respecto a los traductores pues su propósito no es tan particular como el de convertir un lenguaje escrito con unas normas en otro lenguaje que responde a normas distintas; a esta tarea se suma la revisión lexicográfica, sintáctica, semántica y al final la escritura optimizada de un código que puede ser perfectamente entendido por la computadora (código en lenguaje de maquina o ensamblador).

3.2. Fases de un Compilador

Por razones de diseño, la construcción del compilador se divide en varios pasos o fases. Se divide en dos procesos genéricos:

Análisis: El cual hace referencia a la escritura correcta del programa fuente, escrito por un programador.

Síntesis: Una vez se ha ejecutado el análisis de manera exitosa se procede a agrupar las componentes, que conforman el programa fuente, para construir frases con sentido con el fin de generar una salida. Que puede ser en lenguaje de máquina o algún otro lenguaje destino que se considere conveniente.

El proceso de análisis comprende varias fases que son:

Análisis léxico: Es la fase encargada de la lectura y exploración misma del programa fuente, Esto quiere decir que el analizador léxico lee el programa fuente, y lo fracciona en componentes que tienen sentido para el lenguaje de programación que se está considerando.

Análisis sintáctico: A esta fase le corresponde evaluar que el programa fuente escrito realmente cumpla con las especificaciones del lenguaje definido para el compilador. Para ello normalmente el programa fuente debe reflejar una estructura especial. Esta debe responder a una serie de reglas, que pueden ser recursivas o no, las cuales se denominan con el nombre de gramáticas. (Es una de las fases más importantes de la compilación.)

Análisis Semántico: Esta fase se dedica a determinar si todos los componentes del programa están siendo usados de manera válida, para el contexto en el cual aparecen. Es decir, se deben los componentes colindantes a cada componente siendo analizado, antes de determinar que las operaciones ejecutadas por el mismo estén dentro de las operaciones permitidas por el lenguaje, para dicho tipo de situaciones.

Una vez el programa fuente ha sido sometido a un análisis completo y se puede tener en cuenta de que esta correctamente escrito. Solo queda faltando generar algún tipo de salida para que el ciclo de compilación quede completo. Las fases restantes hacen una síntesis del programa fuente para generar la salida. Estas fases son:

Generación de código Intermedio: La mayoría de los compiladores modernos intentan optimizar, hasta donde sea posible, el código que generan. Para lograr esto los compiladores analizan

internamente y tratan de generar secuencias de instrucciones internamente equivalentes a las del programa fuente, o reemplazan instrucciones para hacer un uso más eficiente en la memoria. Su objeto general es generar un código intermedio del programa fuente para que sea usado posteriormente por el optimizador de código.

Optimización de código: El objeto de esta fase es el de mejorar el código fuente escrito para que sea más rápido de ejecutar, o use de manera más eficiente los recursos de la máquina. Este proceso se apoya en la generación de código intermedio que fue realizada en la fase anterior. Por lo general es mucho más complicado optimizar el código basándose en el programa fuente tal como fue escrito por el programador.

Generación de código: El proceso de generación de código es el que constituye la salida, es decir, genera el código de máquina que corresponde al programa fuente. Hay que recordar la diferencia con los intérpretes y su salida exclusiva de código.

◆ Ejemplo de cómo se compila una línea genérica de un lenguaje:

Tomando una sencilla instrucción del lenguaje pascal se explicará cuál es la forma en que el compilador se comporta en cada fase y etapa respectiva. Sea la instrucción:

Suma:= ultimo + penúltimo

Inicialmente el analizador léxico leerá la línea carácter a carácter usando como referencia los espacios en blanco, que indican donde comienza y donde termina cada componen. El analizador léxico será invocado de manera sucesiva por el analizador sintáctico y a cada llamada, en su orden devolviendo lo siguiente:

El identificador: Suma

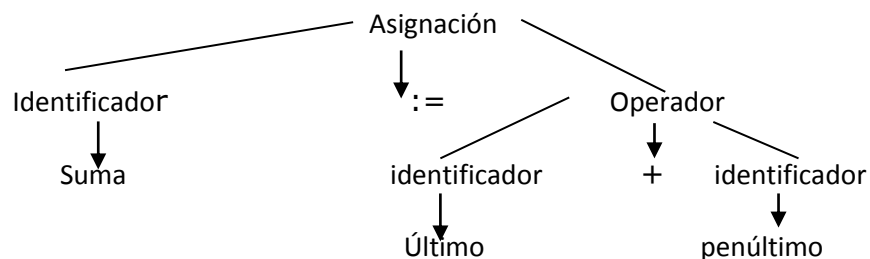
El símbolo de asignación: =

El identificador: último

El operador de suma: +

El identificador: penúltimo

Luego comienza el análisis sintáctico para verificar que la línea escrita en el lenguaje esta correcta. Para lo anterior representa en una estructura jerárquica denominada árbol gramatical así:



Las reglas gramaticales usadas por la línea que estamos discutiendo podrán ser las siguientes:

1. $\langle \text{asignación} \rangle \rightarrow \text{identificador} := \langle \text{expresión} \rangle$
2. $\langle \text{expresión} \rangle \rightarrow \text{identificador}$
3. $\langle \text{expresión} \rangle \rightarrow \langle \text{expresión} \rangle + \langle \text{expresión} \rangle$

La última etapa del análisis se ocupa de la semántica de construcción de las instrucciones del lenguaje; revisando el uso de cada uno de los tipos definidos para determinar si están correctamente usados de acuerdo a su definición.

Posteriormente al análisis continúa la etapa de síntesis que se realiza en su orden de la siguiente manera:

Generación de código intermedio: Consiste en construir un código con variables temporales donde se use el recurso de memoria para representar cada instrucción de forma temporal. En el problema que llevamos como ejemplo el código intermedio sería:

```
Temp1=id2+id3  
Id1=Temp1
```

Después el código intermedio es pasado por el optimizador de código y obtenemos lo siguiente:

```
Id1=Temp1
```

La etapa final en el proceso de síntesis del compilador es convertir el código optimizado a un código entendible por la máquina (lenguaje de máquina o ensamblador). En nuestro ejemplo queda de la siguiente forma:

```
Mov Id3, R1    Mueva el contenido de Id3 a R1  
Add Id2, R1     Sume Id2 con lo que tiene en R1  
Mov R1, Id1     Lleve el contenido de la suma a Id1
```

- ◆ Un segundo ejemplo sobre la forma en que se trabaja un compilador. Sea la instrucción del lenguaje C:

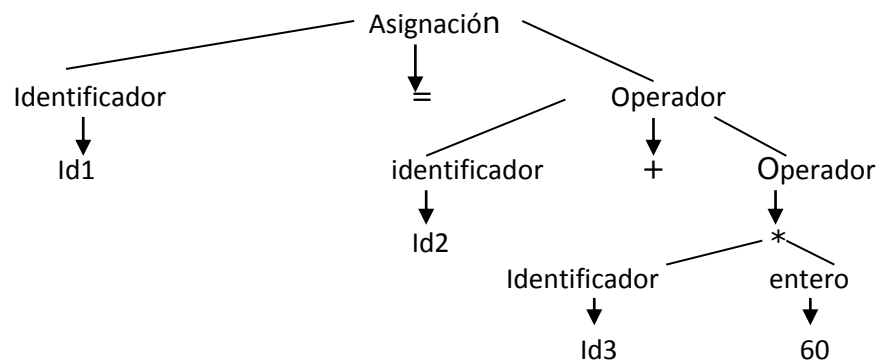
$\text{Posición} = \text{Inicial} + \text{Velocidad} * 60$

Etapas de análisis:

Analizador léxico:

El identificador: Posición
El símbolo de asignación: =
El identificador: Inicial
El operador de suma: +
El identificador: Velocidad
El operador de suma: *
El identificador constante: 60

Analizador Sintáctico: (Representación mediante un árbol sintáctico)



Etapas de Síntesis:

Generación de código intermedio

Temp1=enteroreal (60)
Temp2=Id3*temp1
Temp3=id2+temp2
Id1=temp3

Optimización de código:

Temp1=id3*60
Id1=id2+temp1

Generación de código:

Mov Id3, r2
Mult #60, r2
Mov r2, r1
Add Id2, r1
Mov r1, Id1

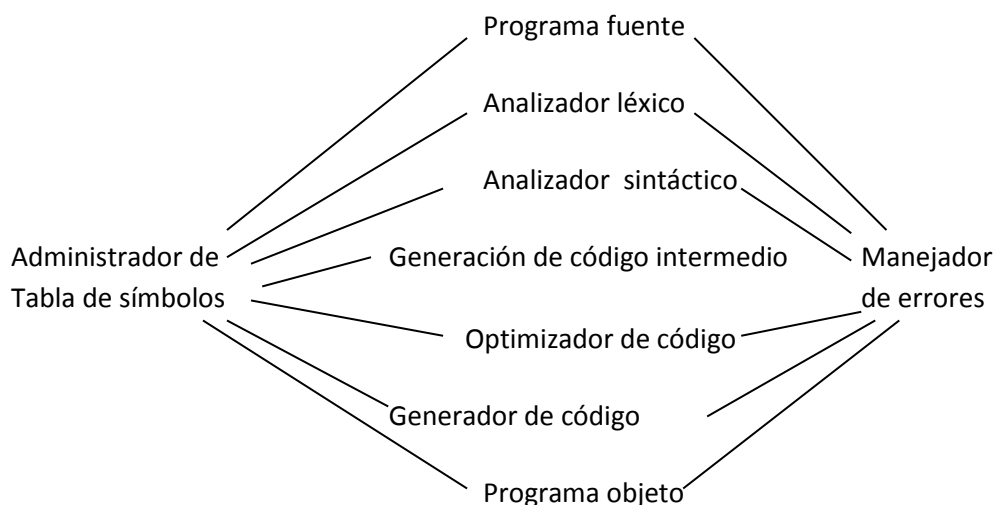
1. Conceptos Adicionales sobre Compiladores:

Para llevar a cabo todo el proceso anterior es necesario de:

Generar y administrar una tabla de símbolos: Esta es una estructura de datos, que puede ser una lista doblemente enlazada, o una tabla en la memoria, que se utiliza para almacenar la información concerniente a las variables definidas por el programa fuente. El objetivo de la tabla de símbolos es facilitar la consulta de la información referente a las variables, sin necesidad de hacer retrocesos en el programa fuente. Por tal razón, existen actividades propias de inserción y búsqueda en la tabla de símbolos que ameritan que estén localizadas en su propia área del compilador.

Detección e información de errores: En cada fase de la compilación se pueden detectar errores. Uno de los objetos de la compilación es tratar de detectar el mayor número posible de errores, antes que se detenga la compilación, para esos efectos se debe informar del error y luego tratar de manipularlo de alguna forma, para que el compilador pueda continuar con el proceso de compilación. Se conoce la poca utilidad de los compiladores que detienen el proceso de compilación ante la presencia del primer error encontrado en el programa fuente.

2. Gráficamente:



3.3. Ejercicios por temas

◆ **Preguntas del tema 1:**

- ◆ De acuerdo a la definición de compilador diga tres características que lo hacen importante para la formación de un ingeniero.
- ◆ Describa tres características que diferencian a un compilador de un traductor normal

◆ **Ejercicio tema 2:**

- ◆ ¿Diga cómo se relacionan el análisis y la síntesis de un compilador?
- ◆ “El analizador léxico trabaja para el sintáctico”. Explique la anterior frase y describa como se realiza el proceso.

◆ **Ejercicio tema 3:**

Mostrar cómo se aplican las fases del compilador a las siguientes instrucciones:

- ◆ $Y = V_0 * t + g * t^2$
- ◆ `If(var1 >= var2)`
- ◆ $w = f * d / t$
- ◆ `while(a >= 1500)`
- ◆ $c = a * (b - d) / f$

Prueba Final

¿Cuál es la diferencia entre un compilador y un traductor?

¿Qué diferencias hay entre el análisis léxico, el sintáctico y el semántico?

¿Con que objeto se realiza análisis léxico en el análisis de un compilador?

¿Qué criterio fundamental usa el compilador al optimizar el código intermedio? Explique

Realizar las dos fases completas del reconocedor para la siguiente instrucción de un lenguaje dado:

- ◆ `while(a >= 1500)`
- ◆ $c = a * (b - d) / f$



3.4. Actividad

Lectura de la introducción a los compiladores y sus conceptos básicos en los libros guía o en alguna dirección de internet con el tema. Discusión en el aula o a nivel virtual, para entender la terminología sobre compiladores.

4. ANÁLISIS LÉXICO

Objetivo General

- ◆ Aplicar la teoría de autómatas para el reconocimiento de patrones de los lenguajes, aplicando el conocimiento teórico en la solución de problemas reales partiendo de las prácticas propuestas en clase (realizar toda la fase de análisis en la creación de un compilador).

Objetivos específicos

- ◆ Aplicar el conocimiento teórico en la solución de problemas reales a partir de las prácticas propuesta en clase.

Prueba Inicial

Seleccione la respuesta correcta según su conocimiento previo:

- ◆ Una componente del lenguaje es:
 - a. Una línea
 - b. Una instrucción
 - c. Un identificador
 - d. Un bloque de instrucción
- ◆ Para recorrer por caracteres una instrucción debo:
 - a) Capturar el tamaño de la cadena
 - b) Usar un ciclo que recorra con una variable hasta el fin de la cadena
 - c) Recorrer hasta encontrar caracteres especiales
 - d) Usar variables temporales de almacenamiento
- ◆ Las variables se diferencian de las constantes en:
 - a. El carácter de inicio
 - b. La cantidad de caracteres
 - c. Las constantes pueden ingresar con una letra al inicio
 - d. Las variables son alfanuméricas

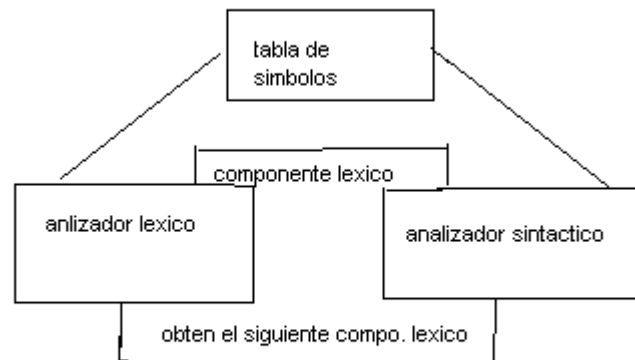
4.1. Definición de la primera parte del análisis de un compilador

El analizador léxico una forma sencilla de crear un analizador léxico consiste en la creación de un diagrama que ilustre la estructura de los componentes léxicos del lenguaje fuente y después hacer la traducción a mano del diagrama a un programa para encontrar las componentes léxicos.

4.1.1. Funciones del analizador léxico:

Permite la obtención de componentes léxicas del lenguaje denominadas tokens, que posteriormente van a ser usadas por el analizador sintáctico en la revisión completa de las líneas del lenguaje (el analizador sintáctico pide las componentes al léxico, quien las saca de la tabla de símbolos y se las entrega al analizador sintáctico).

Gráficamente:



4.1.1.1 Funciones secundarias de interfaz de usuario:

- ◆ Eliminar del programa fuente comentarios y espacios en blanco (blancos, tab y nueva línea).
- ◆ Relaciona los mensajes de error con el programa fuente.
- ◆ En algunos compiladores el analizador de léxico se encarga de hacer una copia del programa fuente en el que están marcados los mensajes de error
- ◆ Si hay procesamiento de macros entonces estas se pueden aplicar en análisis léxico.

- ◆ Razones para dividir el analizador léxico del sintáctico:
- ◆ Un diseño sencillo: Permite la simplificación de una u otra de dicha fases. La eliminación de comentarios y espacios en blanco son hechos dentro del analizador léxico, lo cual libera al sintáctico de tales funciones
- ◆ La eficiencia del compilador por el uso de una lógica de revisión a partir de la teoría de autómatas que permite separar con rapidez y precisión los tokens que contiene cada línea analizada
- ◆ Se mejora la transportabilidad del compilador: Puede darse una generalidad entre los símbolos analizados entre distintos lenguajes de compilación.
- ◆ Componentes Léxicos, patrones y lexemas:
- ◆ Patrón: Representa un conjunto de símbolos con sentido para el lenguaje; más preciso, es una expresión regular.
- ◆ Lexema: Grupo general de caracteres que tienen concordancia con una componente del lenguaje al que se le realiza el análisis del compilador.

TABLA EN LENGUAJE DE ALTO NIVEL

Componentes léxico lexemas Descripción informal del patrón

Const Const Const

IfIfIf

Relación<, <=,=, <>, >, >=< o <= o = o <>

Idpi, cunta, d2 Letra seguida de letras y dígitos

Num3.1416, 0, 6,02E23 constante numérica

En la mayoría de los lenguajes de programación se consideran como componentes léxicos: Palabras clave, operadores, identificadores, constantes, cadenas literales y signos de puntuación como: paréntesis, coma y punto y coma.

◆ Teoría de Autómatas finitos (máquinas de estado finito MEF):

Es un modelo matemático que puede ser simulado e implementado como programa de computador.

Se utiliza porque:

- ◆ Resuelve la mayoría de los problemas en el análisis léxico.
- ◆ Consumen una cantidad fija de memoria.
- ◆ Son muy eficientes.
- ◆ Existe una teoría matemática que da sustento y permite modificarlos.

Una MEF consta de:

- ◆ Un conjunto finito de símbolos de entrada.
- ◆ Un conjunto finito de estados
- ◆ Uno o más estados definidos como estado inicial
- ◆ Uno o más estados definidos como estados de aceptación
- ◆ Un conjunto de transiciones.

Una transición es una función que determina el nuevo estado en que quedará la MEF, con base en el estado actual y el símbolo de entrada.

Ejemplo 1:

Construir una MEF que reconozca secuencias de unos y ceros tal que el número de unos sea impar.

Son válidas: 01101 - 11001 – 1

No es válida: 00010

Símbolos de entrada= {0,1}

Estados = {S0: Numero de unos par
S1: Número de unos impar}

Estado inicial = {So}
Estado de aceptación = {S1}

Se puede construir un diagrama de estados para representar las transiciones de acuerdo al símbolo de entrada en cada variación. Los estados entre círculos, las transiciones con flechas dirigidas y los estados de aceptación con doble círculo.

Tabla de transiciones:

Es una matriz de M filas y N columnas, siendo M el número de estados y n el número de símbolos de entrada.

A la primera fila siempre le corresponde el estado inicial; gráficamente:

SE/Estado	0	1
So	So	S1
S1	S1	So

A

Si son varios los estados iniciales se señalan con flechas(->)

Ejemplo2:

Construir una MEF que reconozca secuencias de unos y ceros tal que el número de unos sea impar y el número de ceros sea par.

Símbolos de entrada = {0, 1}

Estados: {CPUP, CPUI, CIUP, CIUI}

CPUP: ceros pares unos pares
CPUI: ceros pares unos impares
CIUP: ceros impares unos pares
CIUI: ceros impares unos impares

Estado inicial: CPUP

Estado de aceptación es: CPUI

El Gráfico de Transiciones representa la MEF con un gráfico de círculos y flechas que equivale a la matriz representada a continuación:

Tabla de transiciones:

SE/Estados	0	1	Aceptación=1
CPUP	CIUP	CPUI	0
CPUI	CIUI	CPUP	1
CIUP	CPUP	CIUI	0
CIUI	CPUI	CIUP	0

Sentencia nula: Es la ausencia de símbolos de entrada (ϵ)

1. Construir una MEF que reconozcan secuencias de letras a y b de tal forma que el número de b sea par.

Estados = {x = número de b par, x1 = número de b impar}

Estado inicial = {número de b par}

Estado de aceptación = {número de b par}}

Tabla de transiciones:

2. Construir una MEF que reconozca lista de variables en el lenguaje de programación FORTRAN.

Símbolos de entrada = {variable, constante, ',', '(', ')'}.

Estados = {0 = Estado inicial

- 1 = Entró la variable
- 2 = Entró coma fuera de paréntesis
- 3 = Entró Paréntesis izquierdo
- 4 = Entró constante
- 5 = Entró paréntesis derecho
- 6 = Entró coma dentro de paréntesis
- 7 = Entró variable dentro de paréntesis
- Err = Ocurrió un error}

Por ejemplo: a, b, v (10), mat (5, n, 3), h

Estados: 01 21213452 1 346764521

Estado de aceptación = {1: entró variable fuera de paréntesis

5: Entró paréntesis derecho}

La tabla de transiciones de estado es como sigue:

	Var	Cte	,	()	
0	1	err	err	err	err	0
1	Err	err	2	3	err	1
2	1	err	err	err	err	0
3	7	4	err	err	err	0
4	Err	err	6	err	5	0
5	Err	err	2	err	err	1
6	7	4	err	err	err	0
7	Err	err	6	err	5	0
Err	Err	err	err	err	err	0

Nota: Se detectaron estados equivalentes pos inspección: Los estados 0,2; 3,6; 4,7.

Estados equivalentes

Son estados a partir de los cuales se reconoce el mismo conjunto de secuencias.

Cuando se detecta que una MEF tiene estados equivalentes se puede simplificar la máquina.

La MEF para la lista de variables en FORTRAN se simplificaría de la siguiente manera:

	Var	cte	,	()	
02	1	err	err	err	err	0
1	Err	err	02	36	err	1
36	47	47	err	err	err	0
47	Err	err	36	err	5	0
5	Err	err	02	err	err	1
Err	Err	err	err	err	err	0

Hay situaciones en las cuales no se detectan estados equivalentes por simple inspección como en el caso anterior y la MEF tiene estados equivalentes. Para que dos o más estados sean equivalentes deben cumplir las condiciones:

- ◆ Condición de Compatibilidad: los dos o más estados deben ser de aceptación o de rechazo.
- ◆ Condición de Propagación: Para un mismo símbolo de entrada los dos o más estados deben hacer transición hacia estados equivalentes.

Método de particiones:

Sirve para identificar estados equivalentes en una MEF y consiste en aplicar las dos condiciones anteriores, con el objeto de simplificar la MEF.

Ejemplo:

	A	b	
1	6	3	0
2	7	3	0
3	1	5	0
4	4	6	0
5	7	3	1
6	4	1	1
7	4	2	1

Po = {1, 2, 3, 4} No Hay equivalentes por simple inspección

P1 = {5, 6, 7}

Po/a
1 a 6
2 a 7
3 a 1
4 a 4

1, 2 no es equivalente con 3, 4

Po = {1, 2}

Po/b

1 b 3

2 b 3

1 y 2 son equivalentes



P1/a

5 a 7

6 a 4

7 a 4

$P2 = \{6, 7\}$ Son de la misma partición (con el símbolo a van ambos al estado 4)

Estados que quedan en la misma partición son equivalentes. 1 y 2; 6 y 7 son equivalentes.

MEF simplificada:

	A	B	
12	67	3	0
3	12	5	0
4	4	67	0
5	67	3	1
67	4	12	1

Las MEF construidas en los ejemplos iniciales se denominan determinísticas ya que cada transición es única y tiene un solo estado inicial.

La MEF no determinísticas son aquellas que pueden tener más de un estado inicial y/o alguna transición puede ser hacia dos o más estados.

Ejemplo:

	A	B	
0	1	2	0
1	3,0	1	1
2	2,3	0	0
3	1	2	1

Es una MEF no determinística

Abbaba es una hilera que se acepta:

2

0 b 2 a

0 a 1 b 1 b 1 a3*Aceptación

2

3 b 2 a

3*Aceptación

Sí para una hilera de entrada hay solo un camino que lleva a un estado de aceptación la hilera se acepta.

4.1.2. Expresiones Regulares

Una expresión regular representa con un conjunto de símbolos la formación de componentes léxico de un lenguaje formal.

De una expresión regular podemos obtener conjuntos finitos o infinitos de cadenas que corresponden a ella es decir, conjuntos regulares.

Expresiones regulares en teoría de lenguajes formales:

Las expresiones regulares están formadas por constantes y operadores y denotan conjuntos de palabras llamados conjuntos regulares. Dado un alfabeto finito Σ , se definen las siguientes constantes:

1. (conjunto vacío) \emptyset que denota el conjunto \emptyset
2. (Palabra vacía) ϵ que denota el conjunto $\{\epsilon\}$

Del alfabeto) a elemento de Σ que denota el conjunto $\{ "a" \}$

Y las siguientes operaciones:

1. (unión) $r | s$ que denota la unión de R y S , donde R y S son respectivamente los conjuntos denotados por las expresiones r y s
2. (Concatenación) rs que denota el conjunto $\{ \alpha\beta \mid \alpha \text{ en } R \text{ y } \beta \text{ en } S \}$, donde R y S representan respectivamente los conjuntos denotados por las expresiones r y s . Por ejemplo, la expresión $"(ab|c)(d|ef)"$ denota el conjunto $\{ "ab", "c" \} \{ "d", "ef." \} = \{ "abd", "abef", "cd", "cef" \}$.
3. (clausura de Kleene) r^* que denota el más pequeño conjunto que extiende a R , contiene ϵ y está cerrado por concatenación de palabras, donde R es el conjunto denotado por la expresión r . r^* es también el conjunto de todas las palabras que pueden construirse por concatenación de cero o más ocurrencias de R . Por ejemplo, $"(ab|c)^*"$ contiene las palabras ϵ , $"ab"$, $"c"$, $"abab"$, $"abc"$, $"cab"$, $"cc"$, $"ababab"$, etcétera. Para reducir al mínimo el número de paréntesis necesarios para escribir una expresión regular, se asume que la clausura de Kleene es el operador de mayor

prioridad, seguido de concatenación y luego la unión de conjuntos. Los paréntesis solo se incluyen para eliminar ambigüedades.

(Ing. Miguel Ángel Durán Jacobo, Introducción a los Lenguajes formales, Documento en pdf, página12, <http://www.itchetumal.edu.mx/paginasvar/Maestros/mduran/Archivos/Unidad%206%20Introduccion%20a%20los%20lenguajes%20formales.pdf>)

4.1.3. Ejemplos de Expresiones Regulares

- ◆ dígito $d=0|1|2|3|4|5|6|7|8|9$
- ◆ entero_sin_signo $=d^+$
- ◆ entero $=(+|-|)d^+$
- ◆ real $=d^+.d^+(|e|+|-|)d^+$
- ◆ letra $l=a|...|z|A|...|Z$
- ◆ identificador $=l(l|d)^*$
- ◆ string $="(V-")^*$

4.1.3.1 Máquinas de pila (MP)

Es un autómata finito que soluciona algunos problemas del análisis léxico y es usado primordialmente para problemas de reconocimiento en el analizador sintáctico.

Consta de:

- ◆ Un conjunto finito de símbolos de entrada incluyendo fin de secuencia.
- ◆ Un conjunto finito de símbolos en la pila incluyendo pila vacía
- ◆ Un conjunto finito de estados.
- ◆ Un estado designado como estado inicial
- ◆ Una configuración inicial de la pila.
- ◆ Un conjunto de transiciones. Cada transición consta de tres operaciones:
 - ◆ Operación de pila: Apilar, Desapilar o ninguna.
 - ◆ Operación de estado: Cambiar de estado o permanecer en el estado actual.
 - ◆ Operación de entrada: Leer el siguiente símbolo: Avance
 - ◆ No leer el siguiente símbolo: Retenga

Nota: La transición en blanco es de **RECHAZO**

Ejemplo 1:

Construir una máquina de pila que reconozca que los paréntesis estén bien apareados en una expresión aritmética:

Cadenas posibles: $(() ()) \neg$

Símbolos de entrada = $\{ (,), \neg \}$

Símbolos de la pila = {pila vacía, (}

Estados = {So}

Estado inicial = So

Configuración inicial de la pila = Pila Vacía

	()	\neg
(Apile(() So avance	Dasapile(() So Avance	
Pila vacía	Apile(() So avance		A

Ejemplo 2: Construir una máquina de pila que reconozca secuencias de la forma:

$0^n 1^n$ con $n > 0$.

Cadenas posibles

01 \neg

0011 \neg

00001111 \neg

Símbolos de entrada = $\{0, 1, \neg\}$

Símbolos en la pila = {pila vacía, 0}

Estados = So: Se aceptan ceros hasta el primer uno

S1: Se aceptan solo unos

Estado inicial = So

Configuración inicial de la pila = Pila Vacía

	0	1	¬
0	Apile (0) So Avance	S1 retenga	
Pila vacía	Apile (0) So Avance		

So

	0	1	¬
0		Desapile (0) S1 Avance	
Pila vacía			A

S1

Ejemplo 3:

Construya una MP que reconozca secuencias de la forma: $W^2 W'$ con $W = (1+0)^*$ y

R: cadena en reversa.

10111211101¬

Símbolos de entrada= {0, 1, 2, ¬}

Símbolos en la pila= {0,1, pila vacía}

Estados= {So, S1}

Estado inicial= So

Configuración inicial de la pila= pila vacía

	0	1	2	¬
1	Apile (0) So Avance	Apile (1) So Avance	S1 Avance	
0	Apile (0) So Avance	Apile (1) So Avance	S1 Avance	
Pila vacía	Apile (0) So Avance	Apile (1) So Avance	S1 Avance	

So

	0	1	2	\neg
1		desapile (1) So Avance		
0	desapile (0) So Avance			
Pila vacía				A

S1

Operaciones adicionales con la máquina de pila

1. Operación Replace:

Forma general. Replace (α)

Siendo α una hilera cualquiera de símbolos.

La operación reemplaza el tope de la pila por la cantidad de símbolos que tiene la hilera α .

2. Operación Out

Forma general: Out (α)

Imprime la hilera α

Uso de la operación replace

Construir una máquina de pila de un solo estado que reconozca secuencias de la forma:

$0^n 1^n$ con $n > 0$.

Símbolos de entrada= {0,1, \neg }

Símbolos en la pila= {pv, *, 0}

So= Se reciben ceros y unos después de * en la pila

Configuración inicial de la pila = {pv*}

	0	1	¬
0	R	Desapile (1) So Avance	R
*	Replace(0,*) So Avance	Desapile (*) So retenga	R
Pv	R	R	A

Uso de la operación Out

Construir una máquina de pila que acepte como entrada cualquier secuencia de ceros o de unos y produzca una salida de la forma: 0m1n, con m# de ceros y n# de unos.

Imprima ceros y apile unos.

Símbolos de entrada={0,1, ¬}

Símbolos en la pila={pv,1}

So: Acepta cualquier secuencia de ceros o de unos.

Configuración inicial de la pila={pv}

	0	1	¬
1	Out(0) So Avance	Apile(1) So Avance	Desapile out(1) retenga
Pv	Out(0) So avance	Apile(1) So avance	Acepte

4.2. Ejercicios por temas

Conteste las siguientes preguntas:

- ◆ ¿Por qué el analizador léxico elimina espacios blancos y comentarios cuando revisa el código fuente en alto nivel?
- ◆ ¿Qué hace diferente a una componente léxico agrupada en la tabla de símbolos?
- ◆ ¿Qué funciones de los lenguajes de programación deben ser utilizadas por el analizador léxico?
- ◆ ¿Porque en vez de usar el analizador léxico se utiliza el semántico de una vez?

TALLER DE ANALISIS LEXICO

MAQUINAS DE ESTADO FINITO (MEF)

- ◆ Construir una MEF que reconozca las palabras claves while, for para el manejo de ciclos en el lenguaje C
- ◆ Construir una MEF que reconozca las constantes π y $e(3.1416; 2.7172)$
- ◆ Construir una MEF que reconozca las constantes positivas sin punto decimal
- ◆ Construir una MEF que reconozca las de a, b, c en las cuales el número de a sea par el número de b sea impar y el número de c sea par. Suponga que el estado inicial es de paridad para el número de símbolos
- ◆ Construir una MEF que reconozca las secuencias de 1 y 0 tales que se intercale la secuencia partiendo de uno de los símbolos y terminando en el mismo.
- ◆ Construya una MEF para el problema anterior teniendo en cuenta que se comienza en un símbolo y se debe terminar la intercalación con el otro.
- ◆ Construir una MEF que reconozca secuencias que comiencen por la letra os y terminen con la letra a.
- ◆ Construir una MEF que tome como entrada una línea del lenguaje de definición de variables en C y reconozca el tipo y las variables definidas (suponga que se pueden definir vectores y matrices)
- ◆ Construir una MEF que reconozca las variables en una línea de definición de variables en el lenguaje C.
- ◆ Construir una MEF que reconozca las constantes decimales sin signo, con punto decimal y exponente.
- ◆ Construir una MEF que reconozca las señales marinas de auxilio recibidas por un guardacostas que comienzan por 0 o 1 y poseen secuencias intercaladas de a dos ceros y dos unos terminando finalmente en 1.

- ◆ Construir una MEF que reconozca secuencias de unos y de ceros de tal manera que el número de unos sea par y el número de ceros sea impar o se de imparidad en la cantidad para ambos.
- ◆ Construir una MEF que reconozca secuencias que comiencen en cualquier letra y terminen en aba.
- ◆ Construir una MEF que reconozca secuencias de a y b tales que el número de b sea impar.
- ◆ Construir una MEF que reconozca trayectorias para subir desde el primer piso hasta el quinto usando las escaleras o el ascensor y sabiendo que por él ascensor subo de a dos pisos y por las escaleras de a piso.
- ◆ Construir una MEF que permita reconocer secuencias de unos y de ceros que empiecen por 1 tengan posteriormente cantidad par de ceros.
- ◆ Construir una MEF que lea una línea de VB con estructuras condicionales y reconozca las variables los operadores, las constantes y el tipo.
- ◆ Construir una MEF que reconozca secuencias combinadas de vocales y consonantes que permitan solamente la intercalación de símbolos (vocal-consonante o consonante vocal). La secuencia puede terminar en cualquier letra.

TALLER DE EXPRESIONES REGULARES

Construir una expresión regular para los siguientes conjuntos:

- ◆ Conjunto de todas las variables con solo letras y un dígito al final
- ◆ Secuencias de caracteres que comiencen con un dígito y terminen con una letra teniendo una o más letras o números entre dígito y letra en cualquier orden y cantidad
- ◆ Todos los alfanuméricos que empiecen por vocal terminen en cualquier combinación de dígitos.
- ◆ Números decimales de cualquier cantidad de cifras
- ◆ Secuencias de caracteres que comiencen con un dígito y terminen con una letra teniendo una o más letras o números entre dígito y letra en cualquier orden y cantidad
- ◆Cuál es el conjunto determinado por la expresión regular dada:
- ◆ $(\text{Vocal}) | (0 | 1)^*(\text{consonante})$
- ◆ Si $L(r) = \{a, b, c\}$ y $L(s) = \{d, e\}$ que es: $L(s) | L(r)L(s) | L(r)$
- ◆ $(\text{dígito})^* | \text{letra dígito}$
- ◆ $(\text{vocal} | \text{consonante}) \text{ dígito}$

TALLER DE MAQUINAS DE PILA

- ◆ Construir una máquina de pila que reconozca secuencias de la forma:
- ◆ $0^m 1^n 0^m 2 1^m 0^n 1^m$. Con $m > 0; n > 0$
- ◆ Construir una máquina de pila que reconozca secuencias de la forma:

- ◆ $W20^m 1^m 2W^r$, con $m>0$. W es cualquier secuencia de unos y de ceros.
- ◆ Construir una máquina de pila que reconozca expresiones aritméticas bien apareadas con llaves y paréntesis.
- ◆ Construir una máquina de pila que imprima $0^m 1^m$; para cualquier secuencia de 1 ó 0.
- ◆ Construir una MP de un solo estado que reconozca secuencias de 1 y 0 de la forma: $0^m 1^n 0^n 1^m$.
- ◆ Construir una máquina de pila que permita que permita reconocer secuencias de la forma: $a^n b^m z a^m b^n$.
- ◆ Construir una MP de un solo estado que reconozca secuencias de 1 ó 0 que sean intercaladas y tengan la misma cantidad de unos que de ceros.

Prueba Final

Selecciona la respuesta correcta para los siguientes enunciados:

TALLER GENERAL DE ANALISIS LEXICO

1. Una expresión regular es importante porque:
 - a) Permite manejar operadores para símbolos de un lenguaje
 - b) Representa un alfabeto por medio de una expresión concreta y entendible.
 - c) Usa la cerradura de Kleene
 - d) Puede representar el alfabeto sin símbolos
2. Porque es importante la precedencia de operadores en las expresiones regulares:
 - a) Se puede hacer primero la concatenación y después las otras
 - b) Todas tienen la misma precedencia
 - c) Se pueden omitir los paréntesis en expresiones que no los necesitan
 - d) La unión es de igual precedencia que la concatenación
3. El autómata de pila se diferencia de la MEF en que aspectos básicos:
 - a) En la definición de los símbolos de entrada
 - b) En la definición de los estados
 - c) En el uso de la estructura pila
 - d) Todos los anteriores
4. Pila vacía y fin de secuencia en el último estado quiere decir:

- a) Que puedo aceptar en esta condición en el primer estado de una máquina de dos estados
 - b) Que no puedo aceptar en una máquina de dos estados en el primer estado
 - c) Que no puedo aceptar en una máquina de dos estados en su último estado
 - d) Ninguna de las anteriores.
5. Las expresiones regulares son importantes porque?:
- a. Permiten solucionar problemas del análisis léxico
 - b. Permiten solucionar problemas del análisis sintáctico
 - c. Permiten representar un lenguaje con símbolos
 - d. Permiten representar un alfabeto de forma simbólica y con operaciones
6. El autómata de pila que tipo de problemas resuelve:
- a. Cualquier tipo de secuencias
 - b. Secuencias solamente direccionadas
 - c. Secuencias direccionadas y apareamiento
 - d. identifica tokens variables y constantes
7. La expresión regular $D \mid L^*D$ es equivalente a:
- a. $(D \mid L)^*D$
 - b. $D \mid (L^*D)$
 - c. $(D \mid L^*) D$
 - d. $D \mid D^*L$
8. El autómata de pila define símbolos en la pila para:
- a. Permitir manejar la entrada de símbolos en la cadena
 - b. Permitir que se de transición entre los estados
 - c. Permitir la estructura de datos permita generar una condición de aceptación.
 - d. Permitir que la estructura pueda comenzar sin ningún símbolo inicialmente
9. La lógica de un autómata finito sirve para:
- a) Revisar hasta el carácter final de la línea
 - b) Separar por caracteres la línea
 - c) Encontrar errores de escritura en la línea
 - d) Establecer una lógica de revisión por caracteres que separe componentes
10. Un árbol sintáctico representa:
- a. la estructura gramatical de las líneas del lenguaje

- b. La ubicación de componentes léxicas
- c. La estructura de las componentes sintácticas
- d. árbol de componentes del lenguaje

4.3. Actividad

4.3.1. Primera práctica de compiladores

Elaborar un programa **en el lenguaje que escojas** que reciba una instrucción del lenguaje “C” con una expresión aritmética (con los operadores $+$, $-$, $*$, $/$) y los signos de agrupación $\{$, $\{$, $($, $)$ y usando la lógica de los autómatas finitos que se desarrollará en clase permita realizar análisis carácter por carácter en una línea dada identificando y mostrando una lista de variables encontradas, las constantes y los operadores. El programa deberá también identificar si los signos de agrupación están bien apareados en la expresión original y deberá sacar un mensaje de acuerdo a si hay o no un buen apareamiento. Tenga en cuenta que la expresión aritmética solo tiene una llave, pero puede tener cualquier cantidad de paréntesis. Suponga para este análisis léxico la línea de entrada no va a tener faltantes de variables, constantes y operadores.

Se debe entregar lo siguiente:

El archivo fuente y ejecutable funcionando

El análisis con los autómatas finitos,

El diseño (manual corto de usuario): Explicación de la interfaz.

La explicación del código (corto manual técnico): Explicación de los bloques de instrucciones con los cuales se resolvió el problema.

5. ANÁLISIS SINTÁCTICO

Objetivo General

- ◆ Construir un pequeño reconocedor de las instrucciones de un lenguaje tanto a nivel teórico como práctico, utilizando la teoría de gramáticas de los lenguajes de programación.

Objetivos específicos

- ◆ Conocer y usar la teoría de gramáticas de los lenguajes de programación.
- ◆ Aplicar el conocimiento teórico en la solución de problemas reales partiendo de las prácticas propuestas en clase

Prueba Inicial

De acuerdo a tu conocimiento actual contesta las siguientes preguntas:

- ◆ ¿Cómo crees que están hechos los lenguajes de programación de alto nivel?
- ◆ ¿Conociendo la gramática del lenguaje español con cuantas normas está construida? Un lenguaje tendrá la misma cantidad de normas de construcción
- ◆ ¿Cómo se construyeron entonces los lenguajes de programación que conoces?
- ◆ ¿Si no existiera el análisis en la compilación se podría encontrar los errores cometidos por el programador al escribir el código?

5.1. Teoría de Gramáticas

Son conjuntos de normas que definen la construcción de un lenguaje.

Cada norma se dice que es una producción. Toda producción consta de un lado izquierdo y un lado derecho y el separador, el cual se lee como “se define como”.

El lado izquierdo siempre será un no terminal (<nt>) los cuales siempre van entre ángulos <> , los no terminales definen la estructura del lenguaje.

El lado derecho es cualquier hilera de terminales (T) y N.T. pudiendo ser €.

El objetivo de toda gramática es generar hilera final de T, lo cual se consigue mediante un proceso llamado derivación.

Derivar consiste en reemplazar un N.T por el lado derecho de alguna producción que defina ese N.T.

Todo proceso de derivación debe comenzar con el símbolo inicial de la gramática, el cual es el N.T. del lado izquierdo de la primera producción.

Ejemplo:

1. $\langle S \rangle \rightarrow \langle S \rangle + \langle P \rangle$
2. $\langle S \rangle \rightarrow \langle P \rangle$
3. $\langle P \rangle \rightarrow I$

Derivando la anterior gramática

$\langle S \rangle$
#2
 $\langle P \rangle$
#3
I

Otra forma de derivar

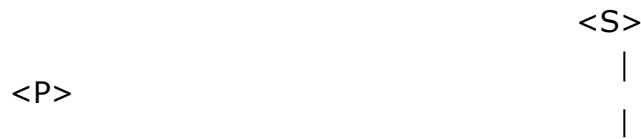
$\langle S \rangle$
#1
 $\langle S \rangle + \langle P \rangle$

#2 #3
 $\langle P \rangle + I$
#3
I + I

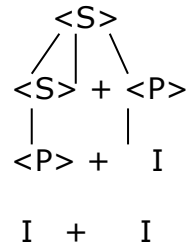
Como la primera producción es de aplicación recursiva la gramática genera sumas consecutivas a partir de un solo termino (sumas infinitas).

Todo proceso de derivación responde a un árbol de derivación. En el caso del ejemplo anterior se representa de la siguiente manera:

Primer árbol de derivación:



I
Segundo árbol de derivación:



Construcción de gramáticas con base en una MEF

- ◆ Los estados de la MEF serán los NT de la gramática
- ◆ Los símbolos de entrada a la MEF serán los terminales de la gramática
- ◆ Cada transición será una producción
- ◆ Cada estado de aceptación implica una producción cuyo lado derecho será ϵ

Forma especial: Son gramáticas cuyas Gramáticas de la producción son de la forma:

$$\begin{array}{l}
 <NT> \xrightarrow{t} <NT> \\
 <NT> \xrightarrow{\epsilon}
 \end{array}$$

Gramáticas Lineales por la derecha: son gramáticas cuyas producciones son de la forma:

$$\begin{array}{l}
 <NT> \xrightarrow{\quad} w <NT> \\
 <NT> \xrightarrow{w}
 \end{array}$$

Siendo w una hilera cualquiera de terminales t que puede ser ϵ

Para poder construir un reconocedor del lenguaje debemos trabajar con gramáticas lineales por la derecha convertidas a una forma especial.

Consideración de lenguajes generados por gramáticas

Ejemplos:

- ◆ Considere la gramática G con la gramatical simple:

1. $\langle E \rangle \rightarrow (\langle E \rangle)$
2. $\langle E \rangle \rightarrow I$
- ◆ Considere la gramática G con la gramatical simple:

1. $\langle E \rangle \rightarrow \langle E \rangle + I$
2. $\langle E \rangle \rightarrow I$
- ◆ La gramática siguiente:

1. $\langle \text{sentencia} \rangle \rightarrow \langle \text{sent_if} \rangle$
2. $\langle \text{sentencia} \rangle \rightarrow \text{otro}$
3. $\langle \text{sent_if} \rangle \rightarrow \text{if}(\langle \text{exp} \rangle) \langle \text{sentencia} \rangle$
4. $\langle \text{sent_if} \rangle \rightarrow \text{if}(\langle \text{exp} \rangle) \langle \text{sentencia} \rangle \text{else} \langle \text{sentencia} \rangle$
5. $\langle \text{expre} \rangle \rightarrow 0 \text{ o } 1$

Sean las gramáticas:

- | | |
|--|--|
| 1. $\langle A \rangle \rightarrow \langle A \rangle a$ | 1. $\langle A \rangle \rightarrow a \langle A \rangle$ |
| 2. $\langle A \rangle \rightarrow a$ | 2. $\langle A \rangle \rightarrow a$ |

Recursiva por la izq.

Recursiva por la derecha

Gramáticas Libres de contexto:

Una gramática libre de contexto es una especificación para la estructura sintáctica de un lenguaje de programación. La gramática involucra reglas de recursividad que determinan la orientación del árbol que sale de la representación.

Como construir gramáticas a partir de lenguajes:

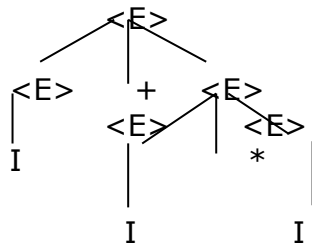
1. Construir una gramática que genere expresiones de la forma: $0^n 1^n$ con $n > 0$
 1. $\langle A \rangle \rightarrow 0 \langle A \rangle 1$
 2. $\langle A \rangle \rightarrow 01$
2. Construir una gramática que genere: $(0+1)^* = W$
 1. $\langle A \rangle \rightarrow 1 \langle A \rangle$
 2. $\langle A \rangle \rightarrow 0 \langle A \rangle$
 3. $\langle A \rangle \rightarrow \epsilon$
3. Construir una gramática que genere sumas y restas recursivas.
4. Construir una gramática que genere secuencias: $0^n 1^m 0^m 1^n$ con $m, n > 0$.

Gramática para generar expresiones aritméticas

1. $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$
2. $\langle E \rangle \rightarrow \langle E \rangle - \langle E \rangle$
3. $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$
4. $\langle E \rangle \rightarrow \langle E \rangle / \langle E \rangle$
5. $\langle E \rangle \rightarrow \langle E \rangle ^ \langle E \rangle$
6. $\langle E \rangle \rightarrow (\langle E \rangle)$
7. $\langle E \rangle \rightarrow I$

NOTA: El identificador I es un token (variable o constante)

Si se realizan derivaciones para obtener la cadena $a + b * c$ así:



Pero se puede derivar aplicando primero el producto y el árbol de derivación queda abriendo a la izquierda y generando un árbol de derivación distinto para la misma expresión extraída del proceso.

Estas operaciones son ambiguas, lo cual puede traer errores, por esto se utiliza la prioridad de operadores además se debe tener en cuenta la asociatividad. Se debe tener en cuenta esta última característica por la izquierda o por la derecha dependiendo cuando lo necesite.

Operadores de menor prioridad deberían ser primeros en las producciones que los de mayor. Una gramática para expresiones aritméticas reescrita con estas características sería:

1. $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$
2. $\langle E \rangle \rightarrow \langle E \rangle - \langle T \rangle$
3. $\langle E \rangle \rightarrow \langle T \rangle$
4. $\langle T \rangle \rightarrow \langle T \rangle * \langle F \rangle$
5. $\langle T \rangle \rightarrow \langle T \rangle / \langle F \rangle$
6. $\langle T \rangle \rightarrow \langle F \rangle$
7. $\langle F \rangle \rightarrow \langle F \rangle ^ \langle P \rangle$
8. $\langle F \rangle \rightarrow \langle P \rangle$

9. $\langle P \rangle \rightarrow \langle E \rangle$

10. $\langle P \rangle \rightarrow I$

En la potenciación se realiza la asociatividad por la derecha:

$\langle F \rangle \rightarrow \langle P \rangle \wedge \langle F \rangle$

Otra gramática para expresiones aritméticas:

1. $\langle E \rangle \rightarrow \langle T \rangle \langle LT \rangle$

2. $\langle LT \rangle \rightarrow + \langle T \rangle \langle LT \rangle$

3. $\langle LT \rangle \rightarrow - \langle T \rangle \langle LT \rangle$

4. $\langle LT \rangle \rightarrow \epsilon$

5. $\langle T \rangle \rightarrow \langle F \rangle \langle LF \rangle$

6. $\langle LF \rangle \rightarrow * \langle F \rangle \langle LF \rangle$

7. $\langle LF \rangle \rightarrow / \langle F \rangle \langle LF \rangle$

8. $\langle LF \rangle \rightarrow \epsilon$

9. $\langle F \rangle \rightarrow \langle P \rangle \langle LP \rangle$

10. $\langle LP \rangle \rightarrow \wedge \langle P \rangle \langle LP \rangle$


11. $\langle LP \rangle \rightarrow \epsilon$

12. $\langle P \rangle \rightarrow \langle E \rangle$

13. $\langle P \rangle \rightarrow I$

5.1.1.1 Procesamiento dirigido por la sintaxis

Gramáticas de traducción:

$a+b*abc*+$ 

Lea(a), imprima(a), lea(+),lea(b),imprima(b),lea(*), lea(c), imprima(c),imprima(*), imprima(+).

Entre llaves las acciones que hay que tomar (imprimir {}). Lo cual implica:

A {a}+b {b}*c{c}{*}{+}

Secuencia de actividades: Es una secuencia de símbolos de entrada y símbolos de acción.

Si a una secuencia de actividades se le suprimen los símbolos de entrada queda la traducción.

Ejemplo 1: Gramática para expresiones en infijo y las traduce a posfijo

1. $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle \{ + \}$
2. $\langle E \rangle \rightarrow \langle T \rangle$
3. $\langle T \rangle \rightarrow \langle T \rangle * \langle F \rangle \{ * \}$
4. $\langle T \rangle \rightarrow \langle F \rangle$
5. $\langle F \rangle \rightarrow (\langle E \rangle)$
6. $\langle F \rangle \rightarrow [\{]$

Ejemplo 2: Construir una gramática de traducción que genere cualquier secuencia de unos y de ceros que produzca una salida: $1m0^n$ siendo m el número de unos y n el número de ceros.

Primero Generamos la gramática que genere cualquier secuencia de unos y ceros:

1. $\langle A \rangle \rightarrow 1 \langle A \rangle$
2. $\langle A \rangle \rightarrow 0 \langle A \rangle$
3. $\langle A \rangle \rightarrow \epsilon$

La gramática de traducción seria:

1. $\langle A \rangle \rightarrow 1 \{ 1 \} \langle A \rangle$
2. $\langle A \rangle \rightarrow 0 \langle A \rangle \{ 0 \}$
3. $\langle A \rangle \rightarrow \epsilon$

Gramáticas con atributos:

Atributo se refiere a la parte del valor de un terminal o un no terminal

1. $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$
2. $p = q + r$
3. $\langle E \rangle \rightarrow \langle T \rangle$
4. $p = q$
5. $\langle T \rangle \rightarrow \langle T \rangle * \langle F \rangle$
6. $p = q * r$
7. $\langle T \rangle \rightarrow \langle F \rangle$
8. $p = q$
9. $\langle F \rangle \rightarrow (\langle E \rangle)$
10. $p = q$
11. $\langle F \rangle \rightarrow [\{]$
12. $p = q$

Atributos Sintéticos: Atributos que se mueven de abajo hacia arriba en el árbol de derivación

Atributos heredados: Atributos que se mueven de arriba hacia abajo o de izquierda a derecha en el mismo nivel.

5.1.2. Teoría de reconocedores del lenguaje(reconocimiento descendente)

Todo el reconocimiento descendente como el ascendente, funcionan como máquinas de pila de un estado.

Definición de la máquina de pila de un estado:

Símbolos de entrada= {los terminales + \rightarrow }

Símbolos en la pila= {pila vacía, n.t, los terminales que estén en una posición distinta a la primera en el lado derecho de alguna producción}

Configuración inicial de la pila={pila vacía<S>, siendo <S> el símbolo inicial de la gramática}

Transiciones: Sea: A: Símbolo en el tope de la pila

X: símbolo de entrada

1. Existe una producción de la forma $A \rightarrow X\alpha$
2. La transición será: replace (α^r) y avance
3. A es un T implica que la transición correspondiente al cruce de x con A, siendo $A=X$, será: DESAPILE, AVANCE.
4. A pila vacía y x es \rightarrow la transición será. ACEPTA
5. En cualquier otro caso: RECHACE

Ejemplo 1: Construir un reconocedor descendente para la siguiente gramática:

1. $\langle S \rangle \rightarrow I$
2. $\langle S \rangle \rightarrow \langle S \rangle \langle R \rangle$
3. $\langle R \rangle \rightarrow \langle S \rangle \langle R \rangle$
4. $\langle R \rangle \rightarrow)$

Nota: Esta gramática posee una particularidad que el primer símbolo del lado derecho en todas las producciones es un terminal (se conoce como gramática S)

SE= {I, (,)}

SP= {<S>, <R>, ∇ }

CIP= { ∇ <S>}

Estado único de reconocimiento.

	I	(,)	¬
<R>			#3	#4	
<S>	#1	#2			
▽					A

- #1: DESAPILE , AVANCE
- #2: REPLACE(<R><S>), AVANCE
- #3: REPLACE(<R><S>), AVANCE
- #4: DESAPILE, AVANCE

Si derivamos secuencias de la gramática y probamos con el reconocedor deben ser reconocidas como de aceptación. Si hay secuencias que no responden a las normas de la gramática el analizador sintáctico las debe rechazar en el análisis.

Características:

Gramáticas S: Son gramáticas en las cuales el lado derecho comienza por un terminal y producciones cuyo símbolo del lado izquierdo sea el mismo, los terminales con los que comienzan sus lados derechos son diferentes.

Conjuntos

Primeros de un N.T.: Es el conjunto de los terminales con los cuales puede comenzar una hilera derivada a partir de ese N.T.

Primeros de una producción: Es el conjunto de terminales con los cuales puede comenzar una hilera al aplicar esa producción.

Siguientes de un símbolo X: (x es T. o N.T.) es el conjunto de T que pueden aparecer después de X en algún proceso de derivación.

Normas para el cálculo de siguientes:

La marca de fin de secuencia (¬) siempre pertenecerá al conjunto de siguientes del símbolo inicial de la gramática.

Cuando busco los siguientes de un símbolo que es último en el lado derecho, entonces el conjunto de siguientes es el mismo que el de su lado izquierdo.

Selección de una producción: Es el conjunto de T. que permite identificar cuando se aplicó la producción, cuando se está haciendo un proceso de reconocimiento.

Normas para cálculo de conjuntos de selección:

Selección de una producción que comienza por un terminal es el mismo terminal.

Selección de una producción que comienza por ϵ (secuencia nula) es el conjunto de siguientes del símbolo del lado izquierdo

Selección de una producción que comience por un N.T. es el conjunto de primeros de ese N.T.

Gramáticas Q: Son gramáticas en las cuales el lado derecho de cada producción comienza con un T. o es la secuencia nula y producciones cuyo símbolo del lado izquierdo sea el mismo tienen conjuntos de selección disjuntos.

M.P. para reconocimiento descendente: La variación en la teoría del reconocedor para las transiciones de la gramática Q es que las producciones con ϵ la transición será:

5.1.3. Desapile, Retenga

Ejemplo de reconocedor para gramática Q. Sea la siguiente gramática:

1. $\langle S \rangle \rightarrow a \langle A \rangle \langle S \rangle$
2. $\langle S \rangle \rightarrow b$
3. $\langle A \rangle \rightarrow c \langle A \rangle \langle S \rangle$
4. $\langle R \rangle \rightarrow \epsilon$

Conjuntos de Selección de las producciones de la gramática:

Selección (1) = {a}

Selección (2) = {b}

Selección (3) = {c}

Selección (4) = {a, b}

Definición del reconocedor:

Símbolos de entrada = {a, b, c, -}

Símbolos en la pila= {<S>, <A>, }

Configuración inicial de la pila= { <S>}

	a	B	C	⌈
<S>	#1	#2		
<A>	#4	#4	#3	
▽				A

#1: REPLACE (<S><A>), AVANCE

#2: DESAPILE, AVANCE

#3: REPLACE (<S><A>), AVANCE

#4: DESAPILE, RETENGA

GRAMATICAS LL (1): Son gramáticas en las cuales el lado derecho de cada producción puede comenzar con un terminal, un N.T. , o ser la secuencia nula, y producciones cuyo símbolo del lado izquierdo sea el mismo tiene conjuntos de selección disjuntos.

M.P. para reconocimiento descendente: La variación en la teoría del reconocedor para las transiciones de la gramática LL1 es que las producciones que comienzan con un N.T.; la hilera α es todo el lado derecho y la transición será:

5.1.3.1 REPLACE (α^r), RETENGA

1. Construya un reconocedor descendente del lenguaje para la siguiente gramática:

1. $\langle IF \rangle \rightarrow \langle S \rangle (\langle EXP \rangle)$
2. $\langle S \rangle \rightarrow \text{if} \mid \text{while}$
3. $\langle EXP \rangle \rightarrow \langle EXP1 \rangle \langle R \rangle$
4. $\langle R \rangle \rightarrow \langle \langle EXP1 \rangle$
5. $\langle R \rangle \rightarrow \rangle \langle EXP1 \rangle$
6. $\langle R \rangle \rightarrow \epsilon$
7. $\langle EXP1 \rangle \rightarrow \langle SUM \rangle \langle SUMAR \rangle$
8. $\langle SUMAR \rangle \rightarrow + \langle SUM \rangle \langle SUMAR \rangle$
9. $\langle SUMAR \rangle \rightarrow - \langle SUM \rangle \langle SUMAR \rangle$
10. $\langle SUMAR \rangle \rightarrow \epsilon$
11. $\langle SUM \rangle \rightarrow \langle MUL \rangle \langle FACT \rangle$
12. $\langle FACT \rangle \rightarrow * \langle MUL \rangle \langle FACT \rangle$
13. $\langle FACT \rangle \rightarrow / \langle MUL \rangle \langle FACT \rangle$
14. $\langle FACT \rangle \rightarrow \epsilon$
15. $\langle MUL \rangle \rightarrow \langle IDEN \rangle$
16. $\langle IDEN \rangle \rightarrow (\langle EXP1 \rangle)$

17. <IDEN> → I

Conjuntos de selección:

Sel(1)=primeros(<S>)= {if | while}

Sel(2)= {if | while}

Sel(3)=primeros(<EXP1>)= {(, I}

Sel(4)={<}

Sel(5)={>}

Sel(6)=siguientes(<R>)= { } }

Sel(7)=primeros(<SUM>)= {(, I}

Sel(8)={+}

Sel(9)={-

Sel(10)=siguientes(<SUMAR>)= {<, >, }

Sel(11)=primeros(<MUL>)= {(, I}

Sel(12)={*}

Sel(13)={/}

Sel(14)={<, >, +, -}

Sel(15)= {(, I}

Sel(16)={}

Sel(17)={I}

Definición del reconocedor:

Simbolos de Entrada={if, while, (,), +, <, >, *, /, -}

Simbolos en la pila={ ∇

, <IF>, <S>, <EXP>, <EXP1>, <R>,

<SUM>, <SUMAR>, <MUL>, <FACT>, <IDEN>, (,)}

Configuración inicial de la pila={ , <IF> } ∇

Reconocedor descendente para la gramática:

	if	While	<	>	()	+	-	*	/	I	∇
)						Desapile, avance						
(Desapile, avance							
<IDEN>					#16						#17	
<FACT>			#14	#14		#14	#14	#14	#12	#13		
<MUL>					#15						#15	
<SUMAR>			#10	#10		#10	#8	#9				

<SUM>					#11						#11	
<R>			#4	#5		#6						
<EXP1>					#7						#7	
<EXP>					#3						#3	
<S>	#2	#2										
<IF>	#1	#1										
▽												A

Definición de operaciones para las transiciones en el reconocedor descendente:

- #1: replace()<EXP>(<S>), RETENGA
- #2: DESAPILE, AVANCE
- #3: REPLACE(<R><EXP1>),RETENGA
- #4: REPLACE(<EXP1>), AVANCE
- #5: REPLACE(<EXP1>), AVANCE
- #6: DESAPILE, RETENGA
- #7: REPLACE(<SUMAR><SUM>),RETENGA
- #8: REPLACE(<SUMAR><SUM>),AVANCE
- #9: REPLACE(<SUMAR><SUM>),AVANCE
- #10: DESAPILE, RETENGA
- #11: REPLACE(<FACT><MUL>),RETENGA
- #12: REPLACE(<FACT><MUL>),AVANCE
- #13: REPLACE(<FACT><MUL>),AVANCE
- #14: DESAPILE, RETENGA
- #15: REPLACE(<IDEN>), RETENGA
- #16: REPLACE()<EXP1>), AVANCE
- #17: DESAPILE, AVANCE

Definición del reconocedor descendente del Lenguaje:

Procesamiento de errores en reconocimiento descendente

1. <S>→|
2. <S>→(<S><R>
3. <R>→,<S><R>
4. <R>→)
5. SE={|,(, , ,), -}
6. SP={▽, <S>, <R>, }
7. CIP={ <▽>}
8. Estado único de reconocimiento.

	I	(,)	¬
<R>	Rd	Re	#3	#4	Rf
<S>	#1	#2	Ra	Rb	Rc
▽	Rg	Rh	Ri	Rj	A

#1: DESAPILE, AVANCE

#2: REPLACE (<R><S>), AVANCE

#3: REPLACE (<R><S>), AVANCE

#4: DESAPILE, AVANCE

Ra, Rb: Escriba("se esperaba una expresión S y legó", símbolo);

Solución: Avance

Apile(<R>), retenga

Rc: write ("Se esperaba expresión S"); exit.

Rd,Re: escriba("falta coma"); avance; apile(<S>), retenga

Rf: write("Expresión incompleta"); exit

Rg, Rh: write("más símbolos después de una expresión S"); Avance; apile(<S>), retenga

Ri, Rj: write("massímbolos de una expresión S"); Avance, apile(<R>), retenga

◆ Reconocimiento ascendente

1. $\langle S \rangle \rightarrow (\langle A \rangle \langle S \rangle)$
2. $\langle S \rangle \rightarrow (b)$
3. $\langle S \rangle \rightarrow (\langle S \rangle a \langle A \rangle)$
4. $\langle S \rangle \rightarrow (b)$

Existen dos técnicas: shift-Identify; shift, reduce

Manejador: Es el último símbolo del lado derecho de una producción.

Cuando se desapila todo un lado derecho de una producción, apila el lado izquierdo de esa de esa producción.

Al final en la pila debe quedar pila vacía y el símbolo inicial de la gramática.

Símbolos de entrada = $\{(, a, b, \neg)\}$

Símbolos en la pila= $\{\langle S \rangle, \langle A \rangle, (, a, b, \neg)\}$ ▽

Configuración inicial: ▽

	(A	b)	¬
<S>					IDO
<A>		SHIFT			R
(R
A					R
B					R
)	IDA	ID1	ID1	ID1	ID1
▽	SHIFT				R

ID0: IF USEP= <S>then ▽

Acepte

Else

Rechace

Símbolos de entrada: T + ¬

Símbolos en la pila: T + NT + ▽

Configuración inicial: ▽

Transiciones:

Si en el tope de la pila hay un manejador implica proceso de identificación. (ID1)

En la columna correspondiente a ¬: si el símbolo en el tope de la pila es el símbolo inicial de la gramática implica proceso de identificación, en las demás filas rechazo (ID0).

Las demás transiciones serán shift (apile y avance).

ID1: if usep= (<A><S>) then

Reduce (1)

Else

If usep= (b) then

Reduce (2)

Else

If usep=(<S>a<A>) then

Reduce (3)

Else

If usep=(a) then

Reduce (4)

Else

Rechace

Reduce(1): DESAPILE(4), APILE(<S>), RETENGA

Reduce(2): DESAPILE(3), APILE(<S>), RETENGA

Reduce(3): DESAPILE(5), APILE(<A>), RETENGA

Reduce (1): DESAPILE (3), APILE (<A>), RETENGA

Para el shiftidentify no hay producciones con la secuencia nula.

Con el shift reduce evito el proceso de identificación.

SHIFT REDUCE

Hay que definir símbolos en la pila para definir qué lado derecho de cual producción se está construyendo

Símbolo gramatical	Símbolo en la pila	Hilera que representa
<S>	<S>1 <S>3 <S>0	(<A><S> (<S>
<A>	<A>1 <A>3	(<A> (<S>a<A>
(((
B	b	(b
A	A3 A4	(<S>a (a
))1)2)3)4	(<A><S>) (b) (<S>a<A>) (a)

Tabla de apilamiento:

	<S>	<A>	(A	B)
	<S>0		(
<S>0						
<S>1)1
<S>3				A3		
<A>1	<S>1		(
<A>3)3
(<S>3	<A>1	(A4	B	
B)2
A3		<A>3	(
A4)4
)1						
)2						
)3						
)4						

En la pila construimos el lado derecho de las producciones.

Cajones en blanco son rechace

La máquina de pila queda como se muestra a continuación:

	(A	b)	¬
					R
<S>V					A
<S>1				t	R
<S>3				f	R
<A>1			i		R
<A>3		H			R
(S				R
B					R
A3					R
A4					R
)1		Reduce(1)			
)2		Reduce(2)			
)3		Reduce(3)			
)4		Reduce(4)			

Shift: Apile, avance según la tabla

((b) a(a)) (b))

En la pila se construye el lado derecho de las producciones

Sea A el tope de la pila y x el símbolo de entrada:

Principio de reducción

Se utiliza para determinar cuáles transiciones serán proceso de identificación. Se hará identificación cuando se presente una de las dos siguientes situaciones:

1. Existe una producción de la forma $\langle L \rangle \rightarrow \text{alfa } A$ y x pertenece a los siguientes $\langle L \rangle$.
2. A es el símbolo inicial de la gramática y x es El principio de reducción se enuncia:

A es reducido por X

PRINCIPIO DE SHIFT: Se utiliza para determinar cuáles transiciones serán shift (apile, avance). Se hará shift cuando se presente una de las dos siguientes situaciones:

1. Existe en el lado derecho de una producción una hilera de la forma αAB y x pertenece a primeros (B).
2. A es y x pertenece a primeros ($\langle S \rangle$) viendo $\langle S \rangle$ el símbolo inicial de la gramática.
El principio de shift se enuncia:
 A está debajo de X

5.2. Ejercicios por temas

5.2.1. Ejercicios definición de gramáticas

- ◆ Construir una gramática que genere sumas y restas recursivas.
- ◆ Construir una gramática que genere secuencias: $0^n 1^m 0^m 1^n$ con $m, n > 0$.
- ◆ Derivar la siguiente gramática Para obtener una cadena derivada de ella.

1. $\langle S \rangle \rightarrow \langle A \rangle b \langle B \rangle$
2. $\langle S \rangle \rightarrow d$
3. $\langle A \rangle \rightarrow \langle C \rangle \langle A \rangle b$
4. $\langle A \rangle \rightarrow \langle B \rangle$
5. $\langle B \rangle \rightarrow c \langle S \rangle d$
6. $\langle B \rangle \rightarrow \epsilon$
7. $\langle C \rangle \rightarrow a$
8. $\langle C \rangle \rightarrow ed$

◆ Ejercicio sobre gramáticas

Construir una gramática de traducción para las siguientes gramáticas:

◆ Gramática 1

1. $\langle S \rangle \rightarrow 1 \langle A \rangle 0$
2. $\langle A \rangle \rightarrow 1 \langle A \rangle 0$
3. $\langle S \rangle \rightarrow \epsilon$

◆ Gramática 2

En una gramática que genere W que muestre primero los ceros y después los unos.



◆ **Ejercicios sobre reconocedores:**

- ◆ Construir reconocedores descendentes para las siguientes gramáticas y realizar la prueba respectiva al reconocedor:

1. Construir un reconocedor para cada una de las siguientes gramáticas:

A.

1. $\langle S \rangle \rightarrow \langle A \rangle \ b \ \langle B \rangle$
2. $\langle S \rangle \rightarrow d$
3. $\langle A \rangle \rightarrow \langle C \rangle \langle A \rangle b$
4. $\langle A \rangle \rightarrow \langle B \rangle$
5. $\langle B \rangle \rightarrow c \ \langle S \rangle \ d$
6. $\langle B \rangle \rightarrow f$
7. $\langle C \rangle \rightarrow a$
8. $\langle C \rangle \rightarrow ed$

b.

1. $\langle S \rangle \rightarrow \langle A \rangle \ b \ \langle B \rangle$
2. $\langle S \rangle \rightarrow d$
3. $\langle A \rangle \rightarrow \langle C \rangle \langle A \rangle b$
4. $\langle A \rangle \rightarrow \langle B \rangle$
5. $\langle B \rangle \rightarrow c \ \langle S \rangle \ d$
6. $\langle B \rangle \rightarrow \epsilon$
7. $\langle C \rangle \rightarrow a$
8. $\langle C \rangle \rightarrow ed$

C.

1. $\langle S \rangle \rightarrow \langle A \rangle$
2. $\langle S \rangle \rightarrow a$
3. $\langle A \rangle \rightarrow \epsilon$
4. $\langle A \rangle \rightarrow b \ \langle S \rangle c \ \langle A \rangle$

D.

1. $\langle S \rangle \rightarrow a \ \langle B \rangle \langle S \rangle$
2. $\langle S \rangle \rightarrow b$
3. $\langle B \rangle \rightarrow c \ \langle S \rangle$
4. $\langle B \rangle \rightarrow \epsilon$
5. $\langle B \rangle \rightarrow d \ \langle B \rangle a$

E.

1. $\langle S \rangle \rightarrow b \langle A \rangle \langle S \rangle \langle B \rangle$
2. $\langle S \rangle \rightarrow d \langle A \rangle$
3. $\langle A \rangle \rightarrow d \langle S \rangle ca$
4. $\langle A \rangle \rightarrow e$
5. $\langle B \rangle \rightarrow c \langle A \rangle a$
6. $\langle B \rangle \rightarrow a$

2. Realizar un proceso corto de derivación y verificar cada uno de los reconocedores que acaban de construir.

Prueba Final

- ◆ Escribir gramáticas para cada uno de los siguientes enunciados:
 1. Condicional de un lenguaje
 2. Instrucción de asignación
 3. Instrucción con ciclos
 4. Instrucción con operadores lógicos
 5. Instrucción con operadores relacionales
- ◆ Derivar cada una de las gramáticas para obtener al menos una hilera del lenguaje que representa.
- ◆ Construir los reconocedores descendentes para las gramáticas y probarlos con las derivaciones anteriores.
- ◆ Explicar el proceso del reconocedor ascendente y compararlo con el reconocedor descendente del lenguaje.
- ◆ ¿Cuál es la importancia de la teoría de gramáticas y el reconocimiento como parte del análisis sintáctico en el compilador?

5.3. Actividad

5.3.1. Segunda Práctica de Compiladores (15%)

Elaborar un programa para un reconocedor ascendente para la siguiente gramática que involucra los operadores relacionales ($>$, $<$, $==$), los operadores aritméticos ($+$ y $*$), los signos de agrupación (los paréntesis) y las palabras claves:

1. $\langle EX \rangle \rightarrow \langle PCLAVE \rangle \langle S \rangle$
2. $\langle S \rangle \rightarrow \langle REL \rangle \langle SIGNO \rangle$
3. $\langle SIGNO \rangle \rightarrow = \langle REL \rangle \langle SIGNO \rangle$
4. $\langle SIGNO \rangle \rightarrow \langle REL \rangle \langle SIGNO \rangle$
5. $\langle SIGNO \rangle \rightarrow \langle \langle REL \rangle \langle SIGNO \rangle$
6. $\langle SIGNO \rangle \rightarrow \epsilon$
7. $\langle REL \rangle \rightarrow \langle SUMA \rangle \langle SUMP \rangle$
8. $\langle SUMP \rangle \rightarrow + \langle SUMA \rangle \langle SUMP \rangle$
9. $\langle SUMP \rangle \equiv \epsilon$
10. $\langle SUMA \rangle \rightarrow \langle PROD \rangle \langle MULT \rangle$
11. $\langle MULT \rangle \rightarrow * \langle PROD \rangle \langle MULT \rangle$
12. $\langle MULT \rangle \rightarrow \epsilon$
13. $\langle PROD \rangle \rightarrow (\langle REL \rangle)$
14. $\langle PROD \rangle \rightarrow I$
15. $\langle PCLAVE \rangle \rightarrow IF-WHILE$

El programa deberá realizarse teniendo en cuenta las siguientes revisiones:

2. Primera revisión: Planteamiento de Solución analítica derivación de la gramática (que incluya mínimo 8 terminales) y construcción del reconocedor descendente (40%)
3. Segunda revisión: Entrega del practica programada con manuales respectivos

Los programas deben entregarse con manuales respectivos de usuario y técnico y con la solución analítica. Parejas de Estudiantes que no tengan los primeros seguimientos en las fechas dadas se les califica sobre 3 el trabajo.

6. FASE SÍNTESIS DEL COMPILADOR

Objetivo General:

- ◆ Relacionar la escritura de instrucciones de alto nivel con su equivalente en bajo nivel y las estructuras básicas de programación de alto nivel con su código correspondiente en bajo nivel.

Objetivos específicos:

- ◆ Relacionar las estructuras básicas de programación de alto nivel con su código correspondiente en bajo nivel.

Prueba Inicial

- ◆ ¿Qué son instrucciones representadas en Memoria RAM?
- ◆ ¿Qué es un registro físico de información?
- ◆ ¿Cómo es en general el lenguaje de máquina?
- ◆ ¿El código de bajo nivel como se relaciona con el código de alto nivel?
- ◆ ¿Conoces sobre instrucciones de bajo nivel?

La síntesis del compilador enmarca los procesos que tienen que ver con conversión del código fuente al lenguaje natural o lenguaje de máquina, para realizar este proceso se deben relacionar toda la estructura del hardware con las instrucciones de programación y los recursos de procesamiento. A continuación se explica el proceso de forma resumida:

Generación de Código intermedio

- ◆ Mantener la semántica del programa.
- ◆ El programa traducido debe tener alta calidad
- ◆ Uso efectivo de los recursos
- ◆ Correr eficientemente = “buen código” (generar la traducción óptima es un problema indecidible)
- ◆ Producir código correcto

Representación Intermedia

Tabla de Símbolos

Programa traducido al lenguaje destino

- ◆ Tres tareas del Generador de Código:
- ◆ Selección de las instrucciones



- ◆ Localización de Registros y asignación
- ◆ Ordenar las instrucciones

Depende de:

- ◆ Representación Intermedia
- ◆ Lenguaje Destino
- ◆ Run-time system

Conjuntos de instrucciones y arquitecturas de Máquina usuales:

- ◆ RISC (ReducedInstruction Set)
- ◆ CISC (ComplexInstruction Set)
- ◆ Basada en Pila (Stackbased)

Explicación de las arquitecturas:

RISC (ReducedInstruction Set):

- ◆ Muchos registros
- ◆ Instrucciones de 3 direcciones
- ◆ Modos de direccionamiento simples
- ◆ Conjunto de instrucciones simple

CISC (ComplexInstruction Set):

- ◆ Pocos registros
- ◆ Instrucciones de 2 direcciones
- ◆ Muchos modos de direccionamiento
- ◆ Instrucciones de longitud variable

Basado en Pila (Stackbased):

- ◆ Las operaciones se realizan sobre operandos que están en el tope de la pila
- ◆ Requiere muchas operaciones de intercambio y copia (swap, copy)
- ◆ Ejemplo: JVM (Java Virtual Machine)

3.4.5 Optimización de código

Localización y asignación de Registros

- ◆ Localización: que variables se asignan a registros
- ◆ Asignación: establecer la relación variable-registro
- ◆ La asignación óptima es un problema NP-Completo.

Orden de Evaluación

- ◆ El orden puede afectar la eficiencia del código
- ◆ En algún orden se puede requerir menos registros que en otro
- ◆ Encontrar un orden óptimo es un problema completo

Referencia tomada de:

(Pontificia U. Javeriana Cali - Ingeniería de Sistemas y Computación – Compiladores – Prof. Ma. Constanza Pabón

http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:compi:comp_sesion26.pdf

3.4.6 Generación de código de bajo nivel

Conversión de código en macro ensamblaje arquitectura IA-32

Formato de Instrucción: La arquitectura IA-32 codifica sus instrucciones máquina con un formato de longitud variable. Toda instrucción tiene una longitud entre 1 y 16 bytes. Las instrucciones comienzan por un prefijo de hasta cuatro bytes, seguido de uno o dos bytes que codifican la operación, un byte de codificación de acceso a operandos, un byte denominado escala-base-índice, un desplazamiento de hasta cuatro bytes, y finalmente un operando inmediato de hasta cuatro bytes. Excepto los bytes que codifican la operación, el resto de componentes son todos opcionales, es decir, su presencia depende del tipo de operación.

Los prefijos son bytes que modifican la ejecución normal de una instrucción de acuerdo a unas propiedades predefinidas. El procesador agrupa estos prefijos en cuatro categorías y se pueden incluir hasta un máximo de uno por categoría. Por ejemplo, el prefijo LOCK hace que mientras se ejecuta la instrucción el procesador tiene acceso en exclusiva a cualquier dispositivo que sea compartido. Este prefijo se utiliza en sistemas en los que se comparte memoria entre múltiples procesadores.

Las instrucciones del lenguaje máquina de la arquitectura IA-32 pueden tener uno de los tres siguientes formatos:

- ◆ Operación. Las instrucciones con este formato no precisan ningún operando, suelen ser fijos y por tanto se incluyen de forma implícita. Por ejemplo, la instrucción RET retorna de una llamada a una subrutina.
- ◆ Operación Operando. Estas instrucciones incluyen únicamente un operando. Algunas de ellas pueden referirse de manera implícita a operandos auxiliares. Un ejemplo de este formato es la instrucción INC %eax que incrementa en uno el valor de su único operando.

- ◆ Operación Operando1, Operando2. Un ejemplo de este tipo de instrucciones es ADD \$0x10, %eax que toma la constante 0x10 y el contenido del registro %eax, realiza la suma y deposita el resultado en este mismo registro. Como regla general, cuando una operación requiere tres operandos, dos fuentes y un destino (por ejemplo, una suma), el segundo operando desempeña siempre las funciones de fuente y destino y por tanto se pierde su valor inicial.

Ejemplo:

```
Push (%ecx)
Push 4(%ecx)
Push $msg
callprintf
add $12, %esp
Pop %edx
Pop %ecx
Pop %eax
Ret
```

Descripción de la instrucción de suma de enteros en la arquitectura IA-32

ADD—Add

Opcode	Instruction	Description
04 ib	ADD AL,imm8	Addimm8to AL
05 iw	ADD AX,imm16	Addimm16to AX
05 id	ADD EAX,imm32	Addimm32to EAX
80 /0 ib	ADD r/m8,imm8	Add imm8 to r/m8
81 /0 ADD iw	r/m16,imm16	Add imm16 to r/m16
81 /0 id	ADD r/m32,imm32	Add imm32 to r/m32
83 /0 ib	ADD r/m16,imm8	Add sign-extended imm8 to r/m16
83 /0 ib	ADD r/m32,imm8	Add sign-extended imm8 to r/m32
00 /r	ADD r/m8,r8	Addr8tor/m8
01 /r	ADD r/m16,r16	Addr16tor/m16
01 /r	ADD r/m32,r32	Addr32tor/m32

02 /r	ADD r8,r/m8	Addr/m8tor8
03 /r	ADD r16,r/m16	Addr/m16tor16
03 /r	ADD r32,r/m32	Addr/m32tor32

Instrucciones aritméticas:

Instrucción	Comentario
ADDL \$3, contador	Suma la constante 3 al número de 32 bits almacenado a partir de la posición contador. El tamaño viene determinado por el sufijo, que en este caso es imprescindible.
SUB %eax, contador	Deposita en memoria el número de 32 bits resultante de la operación contador-%eax.
NEGL contador	Cambia de signo el número de 32 bits almacenado en memoria a partir de la posición contador.
Instrucción	Comentario
MULB \$3	Multiplica el número natural 3 representado en 8 bits por el registro implícito %al y deposita el resultado en %eax. El tamaño de los operandos lo determina el sufijo B.
IMUL %eax	Multiplica el número entero almacenado en %eax por sí mismo (operando implícito). El resultado se almacena en el registro de 64 bits %edx: %eax.
MUL contador, %edi	Multiplica el número natural de 32 bits almacenado a partir de la posición de memoria representada por contador por el registro %edi en donde se almacenan los 32 bits de menos peso del resultado.
IMUL \$123, contador, %ecx	Multiplica el número de 32 bits almacenado en memoria a partir de la posición contador por la constante \$123 y almacena los 32 bits menos significativos del resultado en %ecx.

El programa ensamblador:

En general, a los programas encargados de traducir de un lenguaje de programación a otro se les denomina “compiladores” y todos ellos trabajan de forma similar. Dado un conjunto de ficheros escritos en un lenguaje, producen como resultado otro fichero que contiene la traducción a un segundo lenguaje. En el caso del ensamblador, la traducción es de lenguaje ensamblador a lenguaje máquina.

En adelante se utilizarán los términos “compilador” y “ensamblador” de forma indistinta y siempre en referencia al programa que traduce de lenguaje ensamblador a lenguaje máquina.

Así como el lenguaje máquina de un procesador es único e inmutable (a no ser que se rediseñe el procesador), pueden coexistir múltiples lenguajes ensamblador que representen el mismo lenguaje máquina. La representación de las instrucciones mediante cadenas alfanuméricas es un convenio utilizado para facilitar su escritura, por lo que pueden existir múltiples convenios de este tipo siempre y cuando se disponga del ensamblador los que traduzca al lenguaje máquina del procesador.

El desarrollo de programas en ensamblador requiere un conocimiento en detalle de la arquitectura del procesador y una meticulosidad extrema a la hora de decidir qué instrucciones y datos utilizar. Al trabajar con el lenguaje máquina del procesador, la comprobación de errores de ejecución es prácticamente inexistente. Si se ejecuta una instrucción con operandos incorrectos, el procesador los interpretará tal y como estipula su lenguaje máquina, con lo que es posible que la ejecución del programa produzca resultados inesperados.

◆ Traducción de construcciones de alto nivel a ensamblador

Las construcciones que ofrecen los lenguajes de alto nivel como Java para escribir programas distan mucho de la funcionalidad ofrecida por el lenguaje máquina del procesador. Por ejemplo, en Java se permite ejecutar una porción de código de forma iterativa mediante las construcciones `for` o `while` hasta que una condición se deje de cumplir. El compilador es el encargado de producir el código ensamblador tal que su ejecución sea equivalente a la especificada en el lenguaje Java.

A continuación se muestra cómo la funcionalidad ofrecida por el procesador es suficiente para traducir estas construcciones a secuencias de instrucciones ensamblador con idéntico significado.

Traducción de un `if/then/else`

Estructura de un `if/then/else`

If (expresión booleana) {

Bloque A

} Else {

Bloque B

}

Lo más importante para traducir un bloque a ensamblador es saber su significado o semántica. La semántica del bloque *if/then/else* es que se evalúa la expresión booleana y si el resultado es verdadero se ejecuta el bloque A de código y se ignora el bloque B, y si es falsa, se ignora el bloque A y se ejecuta el bloque B.

El elemento clave para traducir esta construcción a ensamblador es la instrucción de salto condicional. Este tipo de instrucciones permiten saltar a un destino si una condición es cierta o seguir la secuencia de ejecución en caso de que sea falsa. Lo único que se necesita es traducir la expresión booleana de alto nivel a una condición que pueda ser comprobada por una de las instrucciones de salto condicional ofrecida por el procesador. Supóngase que la expresión es falsa si el resultado de la evaluación es cero y cierta en caso contrario. Además, tras ejecutar las instrucciones de evaluación, el resultado se almacena en `%eax`.

Traducción de un *if/then/else* a ensamblador

```
# Evaluar la expresión booleana
# Resultado en %eax
cmp $0, %eax
jebloque b
# Traducción del bloque A
# Fin del bloque A
jmpfinifthenelse
Bloque b: # Traducción del bloque B
# Fin del bloque B
finifthenelse:
# Resto del programa
```

Tras la evaluación de la condición, el resultado previamente almacenado en `%eax` se compara, y si es igual a cero se ejecuta el salto que evita la ejecución del bloque A. En el caso de un *if/then/else* sin el bloque B, el salto sería a la etiqueta `finifthenelse`.

Ejemplo:

Código de alto nivel	Código ensamblador
<code>if ((x <= 3) && (i == (j + 1))) {</code>	<code>cmpl \$3, x # Comparar si x <= 3</code>
Bloque A	<code>jgbloqueB # Si falso ir a bloque B</code>
<code>} else {</code>	<code>mov j, %eax # Obtener j + 1</code>
Bloque B	<code>inc %eax</code>
<code>}</code>	<code>cmp %eax, i # Comparar i == (j + 1)</code>
	<code>jnebloqueB # Si falso ir a bloque B</code>



Código de alto nivel	Código ensamblador
	# Traducción del bloque A jmpfinifthenelse # Evitar bloque B bloqueB: # Traducción del bloque B finifthenelse: # Final de traducción

Estructura de un switch

switch (expresión) {

casevalor A:

Bloque A

Break; // Opcional

Casevalor B:

Bloque B

Break; // Opcional

Default:

Bloque por defecto // Opcional

La semántica de esta construcción establece que primero se evalúa la condición y a continuación se compara el resultado con los valores de cada bloque precedido por la palabra clave case. Esta comparación se realiza en el mismo orden en el que se definen en el código y si alguna de estas comparaciones es cierta, se pasa a ejecutar el código restante en el bloque (incluyendo el resto de casos). Si ninguna de las comparaciones es cierta se ejecuta (si está presente) el caso con etiqueta default. La palabra clave break se puede utilizar para transferir el control a la instrucción que sigue al bloque switch

La estructura del código ensamblador para implementar esta construcción debe comenzar por el cálculo del valor de la expresión. A continuación se compara con los valores de los casos siguiendo el orden en el que aparecen en el código. Si una comparación tiene éxito, se ejecuta el bloque de código que le sigue. Si se encuentra la orden break se debe saltar al final del bloque. En el caso de que ninguna comparación tenga éxito, se debe pasar a ejecutar el bloque default. Supóngase que la evaluación de la expresión es un valor que se almacena en el registro %eax.

Traducción de un switch a ensamblador

Evaluar la expresión

Resultado en %eax

mp \$valorA, %eax # Caso A

Je bloquea



```
cmp $valorB, %eax    # Caso B
```

```
je bloque b
```

```
Jmp default
```

```
Bloquea:
```

```
# Traducción del bloque A
```

```
jmpfinswitch # Si bloque A tiene break
```

```
Bloque b:
```

```
jmpfinswitch # Si bloque B tiene brea
```

```
Default:
```

```
# Caso por defecto
```

```
finswitch:
```

```
# Resto del programa
```

Traducción de un switch a ensamblador

Código de alto nivel	Código ensamblador
switch (x + i + 3 + j) {	mov x, %eax # Evaluar la expresión
case 12:	add i, %eax
Bloque A	add \$3, %eax
break;	add j, %eax
case 14:	cmp \$12, %eax # Caso 12
case 16:	je bloquea
Bloque B	cmp \$14, %eax # Caso 14
case 18:	je bloque b
Bloque C	cmp \$16, %eax # Caso 16
break;	je bloque b
default:	cmp \$18, %eax # Caso 18
Bloque D	je bloquec
}	jmp default
	bloquea:
	# Traducción del bloque A#
	jmpfinswitch # Bloque A tiene break
	bloque b:
	# Traducción del bloque
	bloquec:



Código de alto nivel	Código ensamblador
	# Traducción del bloque C jmpfinswitch # Bloque C tiene break default: # Bloque D finswitch: # Resto del programa

Estructura de un bucle while

While (expresión booleana) {
Código interno}

En este bloque es importante tener en cuenta que la expresión booleana se evalúa al menos una vez y se continúa evaluando hasta que sea falsa. Supóngase que la evaluación de la expresión es cero en caso de ser falsa y diferente de cero si es cierta y el valor resultante se almacena en %eax.

Traducción de un bucle while a ensamblador

eval: # Evaluar la expresión booleana
Resultado en %eax
cmp \$0, %eax
jefinwhile
Traducción del código interno

Jmpeval
finwhile:
Resto del programa

Tras evaluar la condición se ejecuta una instrucción que salta al final del bloque si es falsa. En caso de ser cierta se ejecuta el bloque de código y tras él un salto incondicional a la primera instrucción con la que comenzó la evaluación de la condición. El destino de este salto no puede ser la instrucción de comparación porque es muy posible que las variables que intervienen en la condición hayan sido modificadas por lo que la evaluación se debe hacer a partir de estos valores.



Ejemplo Traducción de un bucle while a ensamblador

Código de alto nivel	Código ensamblador
<pre>while ((x == i) (y < x)) { Código interno }</pre>	<pre>eval: # Comienza evaluación mov x, %eax cmp i, %eax # Comparar si x == i je codigointerno # Si cierto ejecutar código cmp y, %eax jlefinwhile # Si falso ir al final codigointerno: # Código interno jmpeval # Evaluar de nuevo finwhile: # Final de traducción</pre>

Ultimo capitulo extraído de:

(<http://ocw.uc3m.es/ingenieria-telematica/arquitectura-de-ordenadores/lecturas/html/sub.html>,
Capítulos 4, 5, 6, 7,8)

7. PISTAS DE APRENDIZAJE

Tener en cuenta: El estudio de los principales operadores y registros de bajo nivel para la conversión de código.

Tener en cuenta: La relación entre el analizador léxico y el sintáctico a la hora de realizar la práctica de revisión de líneas.

Tener en cuenta: La definición de estados en la máquina de estado finito para dar una solución conveniente a los problemas del analizador léxico.

Tener en cuenta: Que en las matrices de la máquina de pila se deben colocar claramente al final de la fila en los estados cuales de esto son de aceptación.

Tenga presente: que en una máquina de pila solo se acepta en el último estado.

Tenga presente: que los reconocedores de pila siempre son de un solo estado y comienzan en la pila con el símbolo inicial de la gramática

Traiga a la memoria: que en el cálculo de siguientes de un símbolo, se analiza su aparición en el lado derecho de todas las producciones.

Traiga a la memoria: que para construir reconocedores descendentes se deben hallar los conjuntos de selección de las producciones de la gramática.

Tener en cuenta: que para convertir código debe saber la relación entre las estructuras de programación de alto nivel y las de bajo nivel.

Traiga a la memoria: que una expresión regular es una representación simbólica de operadores y operaciones que representa a un lenguaje regular.

8. GLOSARIO

Ascendente: Reconocimiento de gramáticas que se hace de derecha a izquierda en la producción.

Autómatas: Teoría aplicada en la solución de problemas del analizador léxico

Concatenación: Operación de las expresiones regulares que pega el primer conjunto al segundo

Descendente: Reconocimiento de gramáticas que se hace de izquierda a derecha en una producción.

Ensamblador: Lenguaje que se encuentra al nivel de la máquina en el compilador y tiene su propio código de instrucción.

Gramáticas: Conjunto de normas del lenguaje que permiten su generación.

Identify: Principio de identificación de símbolos en un reconocedor ascendente.

Lexemas: Componente generada por el analizador léxico que hace parte de la revisión por caracteres.

Producción: Es una norma escrita para una gramática que combina terminales y no terminales con el símbolo se define.

Replace: Operación sobre la pila que permite reemplazar el tope por una cadena dada

Reduce: principio de reducción usado por el reconocimiento Ascendente

Selección: Conjuntos usados por la teoría de reconocimiento para saber dónde se aplica una producción.

Semántico: Parte del análisis del compilador que determina si las componentes están siendo bien usadas o no dentro del código de alto nivel.

Sintáctico: Parte del análisis del compilador que se encarga de revisar si las líneas están bien escritas

Token: Son las partes de los grandes grupos de componentes principales de los lenguajes de programación (ejemplo: variables, constantes y palabras claves).

Palote: Operación de una expresión regular que representa la unión.

9. FUENTES

Fuentes Bibliográficas

- ◆ Lemote Karen A.(1996) Fundamentos de CompiladoresCECSA
- ◆ Aho, A. Sethi, Ullman J (1990) Compiladores principios, técnicas y herramientas , Addison-wesley
- ◆ Grune Dick y otros (2000), Modern CompilerDesingWiley
- ◆ Rosenkrantzstearns Lewis Compiler Desing Theory
- ◆ Allen Holub ompilerDesig in Compiler
- ◆ G. Sánchez Dueñas Compiladores e interpretes
- ◆ J.A. Valverde Andrew, Un enfoque pragmático

Fuentes Digitales o Electrónicas

- ◆ http://www.todoprogramacion.com.ar/archives/2005/04/interpretes_y_c.html
- ◆ <http://www.slideshare.net/josehaar4920/compiladores-presentation>
- ◆ <http://es.wikipedia.org/wiki/Compilador>
- ◆ <http://www.infor.uva.es/~mluisa/talf/docs/aula/A7.pdf>
- ◆ <http://faustol.wordpress.com/2007/04/10/fases-de-un-compilador/>
- ◆ <http://www.slideshare.net/FARIDROJAS/compilador-presentation>
- ◆ http://www.angelfire.com/linux/eotto/comp_clase1.pdf
- ◆ <http://nereida.deioc.ull.es/~pl/perlexamples/node71.html>
- ◆ <http://www.hardwarecult.com/foro/index.php?topic=2070.msg27800>
- ◆ <http://www.slideshare.net/leopoldocapa/anlisis-lxico-presentation>



- ◆ http://www.angelfire.com/linux/eotto/comp_clase2.pdf
- ◆ http://148.202.148.5/cursos/cc209/teoriacomp/MODULO_2/Teoria_2_2.htm
- ◆ <http://search.conduit.com/Results.aspx?q=expresiones+regulares+compiladores&ctid=CT1854633&octid=CT2431232&SearchSource=1>
- ◆ <http://www.scribd.com/doc/30646172/Expresiones-Regulares>
- ◆ <http://www.lpsi.eui.upm.es/webtalf/Analisis%20sintactico.pdf>
- ◆ <http://ocw.uc3m.es/ingenieria-telematica/arquitectura-de-ordenadores/lecturas/html/isasubset.html>
- ◆ http://www.dea.icae.upcomillas.es/daniel/asignaturas/EstComp_2_IINF/TranspEstComp_2_IINF/Cap4ModDir.pdf
- ◆ <http://www.scribd.com/doc/30646172/Expresiones-Regulares>