

## Puntos de la Programación de

S

ones simples

de símbolos

octos

# Unidad

## # 2

bs 3 fases básicas  
nodo, ejecuta  
lente lo utili-

duce directamente  
código generado  
almacenar en un fichero  
tanto al código máquina  
que almacena la estructura  
códigos para el programa fuente.

Salida: Fich.obj;

nodo



perm. la compilación separada, de  
los programadores pueden desarrollar  
de las partes de un programa más  
lo que es más importante, poder compilar los  
separadamente y realizar una depuración en paralelo.

# Elementos de la Programación de Sistemas

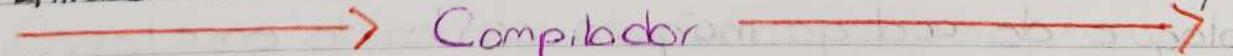
- 2.1 Cargadores
- 2.2 Ensambladores
- 2.3 Macroprocesadores
- 2.4 Sistemas operativos
- 2.5 Traductores de expresiones simples
- 2.6 Incorporación a la tabla de símbolos
- 2.7 Máquinas de pilas abstractas

## Cargadores

Compilación, enlace y carga. Estos son las 3 fases básicas que hay que seguir para que un código sea ejecutado. La interpretación de un texto escrito mediante la utilización de un lenguaje de alto nivel.

Por regla general, el compilador no produce directamente un fichero ejecutable, sino que el código generado se estructura en módulos que se almacenan en un fichero objeto. Poseen información relativa tanto al código máquina como a una tabla de símbolos que almacena la estructura de las variables y tipos utilizados por el programa fuente.

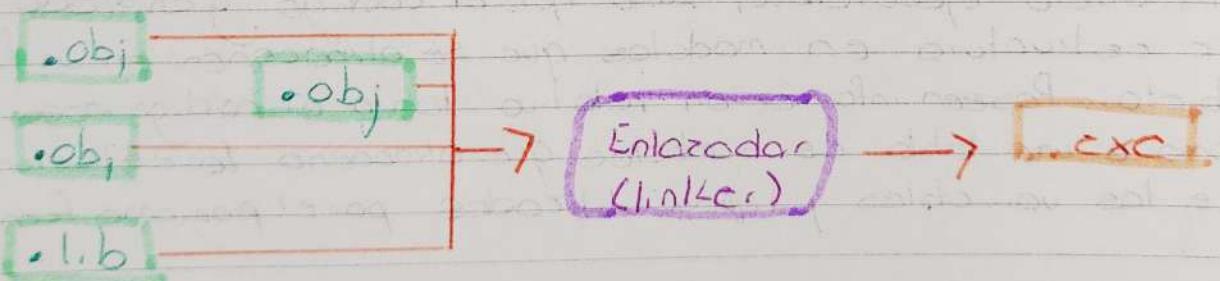
Entrada: Fich. mod



El motivo es para permitir la compilación separada, de manera que varios programadores pueden desarrollar simultáneamente las partes de un programa más grande y, lo que es más importante, pueden compilarlos independientemente y realizar su depuración en paralelo.

Como se ha comentado, un fichero objeto posee una estructura de módulos también llamados registros. Estos tienen longitudes diferentes dependiendo de su tipo y complejidad. Algunos tipos de estos registros almacenan código máquina, otros poseen información sobre las variables globales, y otros incluyen información sobre los objetos externos (p. ej. variables que se supone que están declaradas en otros ficheros).

Durante la fase de enlace, el enlazador o linker, resuelve las referencias cruzadas, (así se llama a la utilización de objetos externos), que pueden estar declarados en otros ficheros objeto, o en bibliotecas (ficheros con extensión "lib" o "dll"), engloba en un único bloque los distintos registros que almacenan código máquina, estructura el bloque de memoria destinado a almacenar las variables en tiempo de ejecución, genera el ejecutable final incorporando algunas rutinas adicionales procedentes de bibliotecas, como por ejemplo las que implementan funciones matemáticas o de E/S básicas.



Así, el bloque de código máquina contenida en el fichero ejecutable es un código reubicable, es decir, un código que en su momento se podía ejecutar en diferentes posiciones de memoria, según la situación de lo mismo en el momento de la ejecución. Según el modelo de estructuración de la memoria del microprocesador, este código se estructura de diferentes formas.

Cuando el enlazador construye el fichero ejecutable asume que cada segmento va a ser colocado en la dirección 0 de la memoria. Como el programa va a esto, dividido en segmentos, las direcciones que hacen referencia a las instrucciones dentro de cada segmento (instrucciones de cambio de control de flujo, de acceso a datos, etc), no se tratan como absolutas, sino que son direcciones relativas a partir de la dirección base en la que se colocaba cada segmento en el momento de la ejecución.

El cargador carga el fichero .exe, cabea sus diferentes segmentos en memoria (donde el sistema operativo le digo que hay memoria libre) y asigna los registros base a sus posiciones correctas, de manera que las direcciones relativas funcionen correctamente.

Seg. Código
Seg. Datos
Seg. Pila
Otros. // / / /

→ Cargador

Dir. base
→ Datos
Dir. base
→ Código
Dir. base
Otros
Dir. base
→ Pila

Fich. exc

Memoria  
ppal.

Cada vez que una instrucción máquina hace referencia a una dirección de memoria (partiendo de la dirección 0), el microprocesador se encarga automáticamente de sumar a dicha dirección la dirección base de inicio de su segmento.

González Morales Víctor

## Ensambladores y macroensambladores

Antes que los compiladores, en los albores de la informática, los programas se escribían directamente en código máquina, y el primer paso hacia los lenguajes de alto nivel lo constituyeron los ensambladores.

En lenguaje ensamblador se establece una relación biunívoca entre cada instrucción y una palabra mnemotécnica, de manera que el usuario escribe los programas haciendo uso de las mnemotécnicas y el ensamblador se encarga de traducirlo a código máquina puro. De esta manera, los ensambladores suelen producir directamente código ejecutable en lugar de producir ficheros objeto.

04/sep/24

Un ensamblador es un compilador sencillito, en el que el lenguaje fuente tiene una estructura tan sencilla que permite a la traducción de cada sentencia fuente a una única instrucción en código máquina. Al lenguaje que admite este compilador también se le llama lenguaje ensamblado. Finalmente, existe una correspondencia uno a uno entre las instrucciones máquina. Ejemplo:

Instrucción ensamblador: LD HL, #0100

Código máquina generado: 65h. 00h. 01h

Por otro lado, existen ensambladores avanzados que permiten definir macroinstrucciones que se pueden traducir a varias instrucciones máquina. A estos programas se les llaman macroensambladores, y suponen el siguiente paso hacia los lenguajes de alto nivel. Desde un punto de vista formal, un macroensamblador puede entenderse como un ensamblador con un preprocesador propio.

## Macroprocessadores

Una macro instrucción, en ocasiones abreviada como macro, es una notación utilizada en la programación, representando un grupo de sentencias usadas comúnmente en el código fuente, permite al programador escribir versiones más cortas de los programas aunque el código objeto tiende a ser mayor que cuando se utilizan funciones.

Las funciones de un macroprocessador invocan la sustitución de un grupo de caracteres o líneas por otros, registran todas las declaraciones de macros y registra el programa fuente para detectar todas las macrollamadas. En cada lugar donde encuentre una macrollamada, el macroprocessador hace la sustitución por las instrucciones correspondientes. A esta acción, en la que el macroprocessador reemplaza cada macro instrucción con el grupo de sentencias correspondiente, se le llama expansión del macro. A excepción de algunos casos, el macro procesador no realiza análisis alguno del texto que maneja.

Para utilizar una macro, primero hay que declararla. En la declaración se establece el nombre que se le dará a la macro y el conjunto de instrucciones que representará. El programador escribirá el nombre de la macro en cada uno de los lugares donde se requiera la aplicación de las instrucciones por ello representadas. La declaración se realiza una sola vez, pero su utilización o invocación a la macro (macrollamado) puede hacerse cuantas veces sea necesario.

Es tan común el empleo de macroinstrucciones se les considera como una extensión de los lenguajes. De manera similar se considera al procesador de macroinstrucciones o macroprocessador como una extensión del ensamblador

o compilador utilizado. En ocasiones es conveniente agrupar macros, de acuerdo a las tareas que realizan, y almacenarlas en archivos que se constituyen en bibliotecas de macros. De esta manera, cuando se requiera la utilización de alguna macro en particular, se incluye en el programa frente el archivo de la biblioteca, de macros correspondiente.

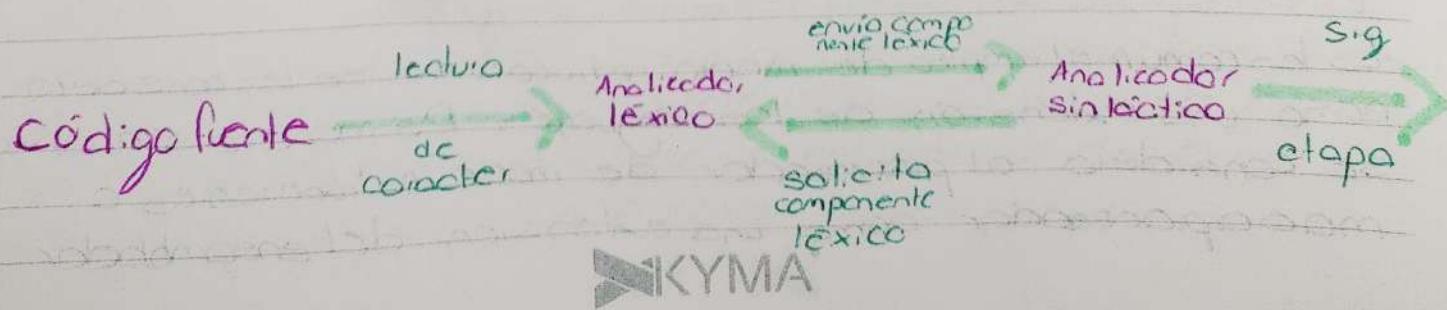
## Análisis Léxico

### Funciones del analizador léxico:

- Agrupa caracteres según categorías establecidas por la espec. func. del lenguaje fuente.
- Recoger texto en caracteres ilegales o agrupados según un criterio no especificado

### Secuencia de actividades de un analizador léxico (scanner)

- 1) Construir tokens válidos a partir de la lectura carácter a carácter del fichero de entrada
- 2) Pasar tokens válidos al analizador sintáctico (parser.)
- 3) Gestión del manejo del archivo de entrada
- 4) Ignorar caracteres de espacio, comentarios y en general caracteres innecesarios para el análisis.
- 5) Aviso, de errores encontrados en esta fase
- 6) Lleva, lo cuenta del número de linea para incluirlo en el mensaje de error.
- 7) Realizar funciones de preprocesos



1.3.902.10

05.sep.24

González Morales Víctor

Los caracteres leídos por el scanner se van guardando en un buffer; al encontrar un carácter que nos sirve para construir un token, se detiene, y envía los caracteres acumulados al parser, y espera una nueva petición de lectura.

Supongamos: int x;  
main () {  
}

La secuencia de operaciones del scanner serán:

Entrada Buffer Acción

i	x	leer otro carácter
n	n	leer otro carácter
t	int	leer otro carácter
blanco	int	enviar token y limpiar buffer
x	x	leer otro carácter
;	x	enviar token y limpiar buffer
j	j	enviar token y limpiar buffer
m		
m		
a		
;		
n		
(		
(		
)		
)		
blanco		
{		
}		
}		
EOF		

Gonzalez Morales Victor

09. sep. 24

## Ejemplo de scanning en un código

```
int x;  
main () {  
}
```

Entrada	Buffer	Acción
i	i	Leer otro carácter
n	in	Leer otro carácter
+	int	Leer otro carácter
espacio blanco	int	Enviar token y limpiar buffer
x	x	Leer otro carácter
:	x	Enviar token y limpiar buffer
m	:	Leer otro carácter
m	;	Enviar token y limpiar buffer
a	;	Leer otro carácter
;	ma	Leer otro carácter
n	mai	Leer otro carácter
(	main	Enviar token y limpiar buffer
(	main	Leer otro carácter
)	(	Enviar token y limpiar buffer
)	)	Leer otro carácter
espacio en blanco	)	Enviar token y limpiar buffer
	{	Leer otro carácter
}	{	Enviar token y limpiar buffer
}	}	Leer otro carácter
Fin de fichero		Enviar token y limpiar buffer

## Términos Usados

**Patrón:** Representación lógica de una serie de cadenas con características comunes. Por ej.: identificadores de una variable en Java, cualquier combinación de letras y números (incluyendo el subrayado) que no comiencen con un número. El patrón sería la definición formal de esto. Para describir formalmente esta definición, se utilizan las expresiones regulares.

Los expresiones regulares se utilizan también en teoría de automáticos, y son una manera de ver las entradas válidas para un automata. Por ej., en cuanto a los identificadores de variables en Java, serían:

Letra ::= (a-zA-Z)

Dígito ::= (0-9)

Subrayado ::= (-)

Identificador

(Subrayado | Letra)(Subrayado | Letra | Dígito)\*

**Lexema:** Cada una de las cadenas que encajen en la definición de un patrón. Por ejemplo, los secuencias "variable 1", "x", "y12" encajarían en el patrón de identificadores de una variable en Java. Es decir, el patrón es la definición formal y el lexema es cada una de las secuencias que pueden caer en esa definición.

**TOKEN:** Nombre del patrón definido. Se utilizan en los procesos de análisis siguientes en representación de todos los lexemas encontrados. Donde encaje en la gramática un token, encajará o combinará de los lexemas que representa. Luego, se podrá ver qué lexemas en concreto son los representados por un token.

## González Moroles Viator

El token es, por ejemplo, la palabra fruta y las lexemas son las frutas en concreto, manzana, plátano, etc. De esta manera, cuando se menciona la palabra fruta, se entendería que se hace referencia a cualquier fruta.

**Atributo:** Cada tipo de token tiene una serie de características propias o su tipo que serán necesarios más adelante en los siguientes etapas del análisis. Cada una de estas características se denominan atributos del token. El analizador léxico no solo pasa el lexema al sintáctico, sino también el token. Es decir, se le pasa una pareja (token, lexema).

## Especificación del analizador léxico

10. Sep. 24

Para entender cómo funciona un analizador léxico, lo explicamos como una máquina de estados (llamado DT). Es similar a un automata finito determinista (AFD), pero con algunas diferencias:

- El AFD solo indica si una secuencia de caracteres es válida o no. En cambio, el DT lee la secuencia completa para identificar una palabra (token), la devuelve, y luego sigue leyendo.
- Si el DT encuentra una secuencia no válida, lo tira como un error. En el AFD, estos errores podrían manejar con estados especiales.
- Los estados finales del DT deben ser estados de aceptación.
- Si el DT encuentra un carácter que no pertenezca a una secuencia válida, va a un estado especial y reinicia desde el siguiente carácter.

Un analizador léxico debe distinguir entre identificadores y palabras reservadas, ya que ambas pueden parecer iguales. Por ej., en Java, "int" sigue siendo patrón de un identificador, pero en realidad es una palabra reservada. Para saber si un lexema es un identificador o una palabra reservada, hay 2 pasos:

- 1) Crear una tabla con todas las palabras reservadas y construirla para verificar cada identificador.
- 2) Programar las palabras reservadas directamente en la máquina de estados (DT), aunque esto sería complicado.

Lo más común es usar una tabla de consulta. Cada vez que se encuentra un posible identificador, se verifica si está en la tabla de palabras reservadas. Si está, es una palabra reservada, si no, es un identificador. Esta tabla debe ser rápida, sin importar cuántas palabras reservadas haya, más a menos palabras reservadas.

La construcción del automata que reconoce un lenguaje es un paso previo a la implementación del algoritmo de reconocimiento. A partir del automata es sencilla la implementación. Un 1er paso es representarlo en una tabla de transiciones.

Por ejemplo: Veamos cómo generar un analizador léxico para un lenguaje sencillo que reconoce números enteros sin signo, la suma, incremento y el producto. Son útiles los sig lexemas: "31", "32", "+", "++", "\*".

Para empezar, deben definirse las patrones o ER. Para este ejemplo:

Gonzalez Moroles, Victor

10. sep. 24

Digito ::= ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9")

Entero ::= Digito +

Suma ::= "+"

Producto ::= "\*"

Incremento ::= "+"

El simbolo + significa que debe haber al menos una de los nro. o más. Si se hubiera utilizado el simbolo \*, significaría 0 o más. El simbolo / significa O lógico. Los caracteres van encerrados entre comillas dobles. Ya definidos los valores, se crean los automatos que los reconocen. Los estados de aceptación están marcados con un círculo doble, el nombre del token, en mayúsculas. Un asterisco es un estado de aceptación indica que el apuntador que scíela la lectura del siguiente carácter debe retroceder una unidad (si hubiere más asteriscos, retrocederá tantas veces como asteriscos).

González Morales Victor

Fecha  
10 - Sep

Producto:

$$\textcircled{0} \xrightarrow{*} \textcircled{1}$$

Suma

$$\textcircled{0} \xrightarrow{+} \textcircled{1} \xrightarrow{0+10} \textcircled{2} *$$

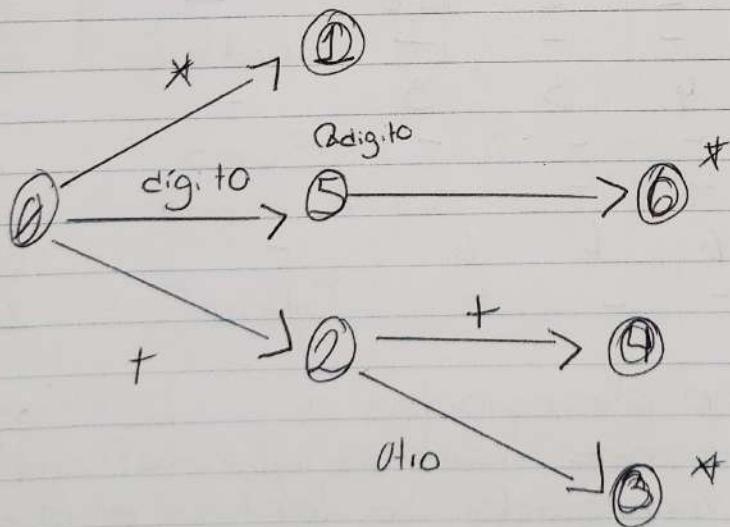
Incremento

$$\textcircled{0} \xrightarrow{+} \textcircled{1} \xrightarrow{+} \textcircled{2}$$

Entero

$$\textcircled{0} \xrightarrow{\text{digito}} \textcircled{1} \xrightarrow{\text{digito}} \textcircled{0} *$$

Autómata Completo



Al llegar a un estado de aceptación, se puso el token al analizado sintático y espera una nueva petición del sintático para comenzar otra vez en el estado 0. Ya creó el automata y comprobar que funcione, adecuadamente, se crea la tabla de transiciones, forma más cercana a la implementación de automata. La tabla de transición tiene tantas filas como estados el automata. En cuanto a las columnas, tiene una para numerar el estado, otras tantas como entradas, es posible unirlas si tienen origen y fin en el mismo estado, otras para el token y otra para los retrocesos en el lexema.

Para el ejemplo, la tabla podría ser:

Estado	Dígito	+	*	Otro	TOKEN	Retroceso
0	5	2	1	Error	-	-
1	-	-	-	-	Producto	0
2	3	4	3	3	-	-
3	-	-	-	-	Suma	1
4	-	-	-	-	Incremento	0
5	5	6	6	6	-	6
6	-	-	-	-	Entero	1

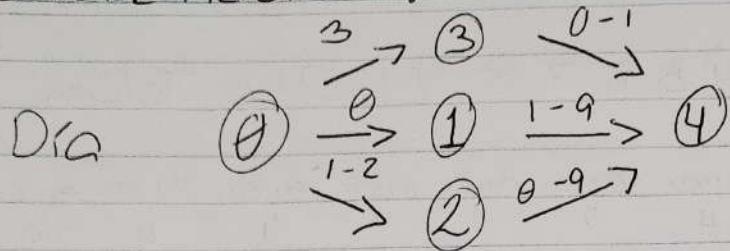
Diseñar una máquina de estados a partir de las expresiones regulares que definen la gramática para un reconocedor de fechas en formato dd/mm/aaaa. Elabore el diagrama de transiciones. Nota: los valores para año están en rango 1000 - 2999.

12-Sep

Día:  $(\emptyset [1-9] | [1-12] [\emptyset-9] | 3 [\emptyset-1])$   
 Mes:  $(\emptyset [1-9] | 1 [\emptyset-2])$   
 Año:  $(1 [\emptyset-9] \{33} | 2 [\emptyset-9] \{33}) = ([1-2] [\emptyset-9] \{33})$

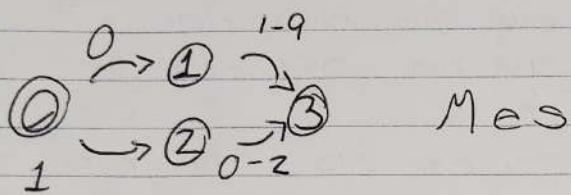
Gonzalez Morales Victor

Fecha 12 - Sep

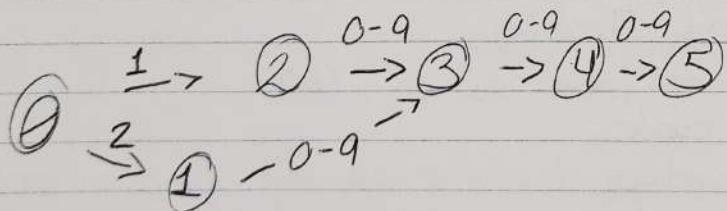
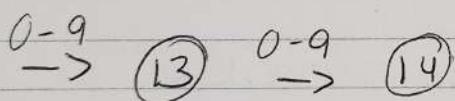
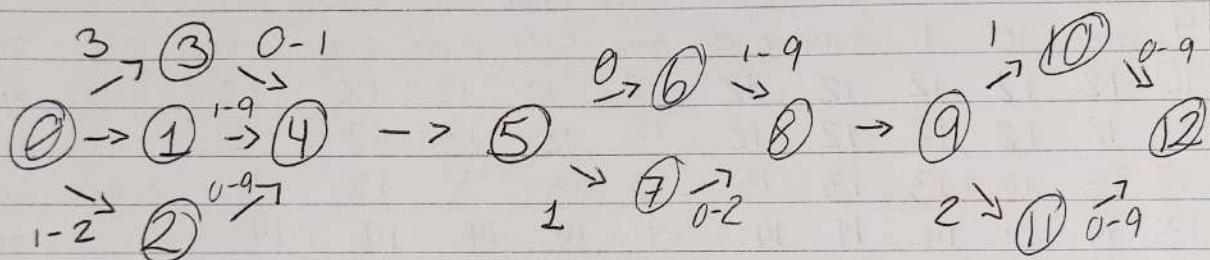


$$\text{Mes} := (\emptyset [1-9] \mid 1[\emptyset-2])$$

17 - Sep



Mes



$$\text{Ano} := (1[\emptyset-9] \{3\} \mid 2[\emptyset-9] \{3\}) -$$

$$(\{1\} \{2\} [\emptyset-9] \{3\})$$

Gonzalez Morales Victor

Fecha  
18-Sep

Estado 0 1 2 3 4 5 6 7 8 9 / Otro Token

0	1	2	2	3	error								
1	error	4	4	4	4	4	4	4	4	4	error	error	error
2	4	4	4	4	4	4	4	4	4	4	error	error	error
3	4	4	error										
4	error	5	error	error									
5	6	7	clor	error	error	clor	error	error	error	error	clor	error	error
6	error	8	8	8	8	8	8	8	8	8	error	error	error
7	8	8	8	clor	error	error	error	error	error	clor	error	error	error
8	error	error	clor	error	error	error	error	error	error	clor	9	error	error
9	error	10	11	clor	error								
10	12	12	12	12	12	12	12	12	12	12	clor	error	error
11	12	12	12	12	12	12	12	12	12	12	error	error	error
12	13	13	13	13	13	13	13	13	13	13	error	error	error
13	14	14	14	14	14	14	14	14	14	14	error	error	error
14	-	-	-	-	-	-	-	-	-	-	-	-	Fecha

## Incorporación de una tabla de símbolos

Estructura de datos: Tabla

Análisis léxico: Inserta lexema y token

Interfaz de la tabla:

inserta(s, t): devuelve índice de la nueva entrada para la cadena "s" y el token "t"

busca(s): devuelve el índice de la entrada para la cadena "s" ó -1 si no lo encontró.

Tanto el scanner como el parser crean en el formato de las entradas en la tabla de símbolos.

Para palabras reservadas se puede prellenarse la tabla de símbolos en llamadas como: inserta ("div", div); inserta ("mod", mod);

Hector

Gonzalez Mordes Victor

18-sep

## Implementación de una tabla de símbolos:

Matriz symbolTable

	lexptr	lexcom p	atributos
0			
1	.	div	
2	:	mod	
3	*	id	

\* La entrada Ø se deja vacía porque es el valor devuelto por busca(s) cuando no encuentra la cadena

div	v	FDC	mod	FDC	identificadora	FDC
-----	---	-----	-----	-----	----------------	-----

Matriz Lexemor

## Tabla de Símbolos y su implementación

19 · sep · 24

Las tablas de símbolos generalmente necesitan admitir múltiples declaraciones del mismo identificador dentro de un programa. El alcance de una declaración es la parte de un programa a la que se aplica la declaración. Implementaremos los alcances estableciendo una tabla de símbolos separada para cada alcance. Un bloque de programa con declaraciones tendrá su propia tabla de símbolos en una entidad para cada declaración en el bloque. Este enfoque también funciona para otras constituciones que establecen alcances; por ejemplo, una clase tendrá su propia tabla con una entidad para cada campo y método.

Los entidades de las tablas de símbolos se crean y se utilizan durante la fase de análisis por

el analizador léxico, sintático y semántico. Hacemos que el analizador sintático cree las entradas. Con su conocimiento de la estructura sintática de un programa, un analizador sintático a menudo está en una mejor posición que el analizador léxico para distinguir entre diferentes declaraciones de un identificador.

En algunos casos, un analizador léxico puede crear una entrada en la tabla de símbolos de pronto como ve las características que componen un lexema. Más a menudo, el analizador léxico solo pue de devolver al analizador sintético un token, como 'id' junto con un puntero al lexema. Sin embargo, sólo el analizador sintético puede decidir si usar una entrada de tabla de símbolos creada previamente o crea, o no, una nueva para el identificador.

Las implementaciones de tabla de símbolos para bloques pueden aprovechar la regla de anidamiento más cercano. El anidamiento garantiza que la cadena de tablas de símbolos aplicables forme una pila. En la parte superior de la pila coloca la tabla para el bloque actual. Debajo de ella en la pila están las tablas para los bloques envolventes. Por lo tanto, los tablas de símbolos se pueden asignar y desasignar de manera similar a una pila.

Algunas compiladoras mantienen no solo tabla hash de entradas accesibles, de entradas que no están ocultas por una declaración en un bloque anidado. Tal tabla hash admite búsquedas esencialmente de tiempo constante, a expensas de insertar y eliminar entradas al entrar y salir del bloque!

Al salir de un bloque B, el compilador, debe deshacer cualquier cambio en la tabla hash debido a las declaraciones en el bloque B. Puede hacerlo utilizando una pila auxiliar para realizar un seguimiento de los cambios en la tabla hash mientras se procesa el bloque B.

Un programa consta de bloques con declaraciones genéricas y "sentencias" que consisten en identificadores individuales. Cada una de estas sentencias representa un uso del identificador. Aquí hay un ejemplo de programa en este lenguaje:

```
{
    int x; char y;
}
bool y;
x;
y;
{
    x, y;
}
```

La tarea a realizar es imprimir un programa revisado, en el que se hayan eliminado las declaraciones y cada "sentencia" tenga su identificador seguido de dos puntos y su tipo, el objetivo es producir:

```
{
    {
        x:int; y:bool;
    }
    x:int; y:char;
```

Los primeros x e y son del bloque interno de entrada. Dada que este uso de x se refiere a la declaración de x en el bloque externo, va seguido de int, el tipo de esa declaración. El uso de y en el bloque interno se refiere a la declaración de y en ese

Gonzalez Morales Victor

misma bloque y, por lo tanto, tiene tipo booleano. También vemos los usos de  $x$  e  $y$  en el bloque externo, con sus tipos según lo dada por las declaraciones del bloque externo: entero y carácter, respectivamente.

## Máquina de pila abstracta

23. Sep

Modelo teórico de una arquitectura de computadora que utiliza una pila como su estructura de datos principal para gestionar la memoria y las operaciones.

Es un concepto utilizado en el diseño de compiladores e intérpretes, especialmente para lenguajes de programación de alto nivel, ya que simplifica la ejecución de programas y el manejo de las variables y funciones.

Características clave de una máquina de pila abstracta:

Uso de una pila: En lugar de usar registros como en una CPU real, una máquina de pila realiza todas las operaciones en una pila, donde los valores más recientes se colocan en la parte superior. Las instrucciones típicas manipulan esta pila, empilando o sacando valores de ella.

Instrucciones simples: Las instrucciones de una máquina de pila abstracta suelen ser muy básicas. Por ej., para sumar dos números, se utiliza una instrucción que saca (pop) los 2 valores superiores de la pila, los suma, y luego empuja (push) el resultado de vuelta en la pila.

1. Manipulación de la pila:

PUSH : Inserta un valor en la cima de la pila.

POP : Elimina el valor de la cima de la pila.

DUP : Duplica el valor de la cima de la pila.

SWAP : Intercambia los 2 valores superiores de la pila.

## 2) Operaciones aritméticas:

ADD : Suma los dos valores superiores de la pila y coloca el resultado.

SUB : Resta el segundo valor de la cima del primero y coloca el resultado.

MUL : Multiplica los 2 valores superiores de la pila y coloca el resultado.

DIV : Divide el 2do valor de la cima del 1ero y coloca el resultado.

Otros: Operaciones como módulo, negación, etc.

## 3) Comparaciones

EQ : Compara si los 2 valores superiores son iguales.

NE : Compara si los 2 valores superiores son diferentes.

LT : Compara si el 2do valor es menor que el 1ero.

GT : Compara si el 2do valor es mayor que el 1ero.

LE : Compara si el 2do valor es menor o igual que el 1ero.

GE : Compara si el 2do valor es mayor o igual que el 1ero.

## 4) Control de Flujo

JUMP : Salta a una etiqueta determinada.

JZ : Salta a una etiqueta si el valor de la cima de la pila es cero.

JNZ : Salta a una etiqueta si el valor de la cima de la pila no es cero.

CALL : Llama a una subrutina.

RETURN : Retorno de una subrutina.

23-Sep

## 5) Acceso a memoria:

**LOAD:** Carga a un valor de una dirección de memoria en la pila.

**STORE:** Almacena el valor de la cima de la pila en una dirección de memoria.

Ejecución en el manejo de funciones:

Las máquinas de pila son muy útiles para implementar la llamada de funciones y la gestión de variables locales. Con la llamada a una función puede crear un nuevo "marco de pila" (stack frame) para almacenar los valores de las variables locales y los resultados intermedios.

Fundamental para el análisis: Una máquina de pila abstracta simplifica el análisis y la optimización del código en el proceso de compilación. Al estar orientada a la pila, las dependencias entre operaciones son explícitas, lo que facilita la evaluación de expresiones y la gestión de las variables.

Ejemplo 1: Si tuviéramos una expresión como  $3+5$ , una máquina de pila podría ejecutar algo como:

Push 3 (empujar 3 a la pila)

Push 5 (empujar 5 a la pila)

Add (sumar 3 y 5 de la pila, sumarlos y empujar el resultado, 8, de vuelta a la pila)

Ejemplo 2: Factorial en el lenguaje funcional usando una máquina de pila abstracta.

$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{otherwise} \end{cases}$

González Morales Víctor

23-Sep

La evaluación de esta función en una máquina de pila abstracta podría ser la siguiente:

- 1) Se empuja el valor de ' $n$ ' a la pila.
- 2) Se compara ' $n$ ' con  $0$ .
- 3) Si ' $n$ ' es  $0$  se empuja  $1$  a la pila.
- 4) Si ' $n$ ' es distinto de  $0$  se realiza una llamada recursiva a factorial ( $n - 1$ ). El resultado de esta llamada se multiplica por ' $n$ ' y se empuja el resultado a la pila.

Esta función podría traducirse a un código de pila de la sig. manera:

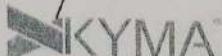
factorial ( $n$ ):

1. PUSH  $n$
2. PUSH  $0$
3. EQ // Compara si  $n$  es igual a  $0$
4. JZ etiquete 1 // Si es igual, salta a etiqueta 1
5. PUSH  $n$
6. PUSH  $1$
7. SUB //  $n - 1$
8. CALL factorial // Llama a la función factorial recursivamente
9. MUL // Multiplica  $n$  por el resultado de factorial ( $n - 1$ )
10. GOTO etiqueta 2
11. Etiqueta 1:
12. PUSH  $1$
13. Etiqueta 2
14. RETURN.

Implementación de MPA.

Java:

```
import java.util.ArrayList;  
import java.util.EmptyStackException;
```



Gonzalez Morales Victor

23-Sep

```
class Pila {  
    private ArrayList<Object> items;  
  
    public Pila() {  
        this.items = new ArrayList<>();  
    }  
    public void push(Object item) {  
        items.add(item);  
    }  
    public Object pop() {  
        if (!isEmpty()) {  
            return items.remove(items.size() - 1);  
        } else {  
            throw new EmptyStackException();  
        }  
    }  
    public boolean isEmpty() {  
        return items.isEmpty();  
    }  
};
```

```
#include <csstd.h>  
#include <csdlb.h>  
#include <csdbool.h>  
  
#define MAX 100  
  
typedef struct Pila {  
    int items[MAX];  
    int top;  
} Pila;
```

```
void init(Pila *pila) {  
    pila->top = -1;  
}
```

González Morales Víctor

23-Sep

```
void push (Pila *pila, int item) {
    if (pila->top < MAX - 1) {
        pila->items [++(pila->top)] = item;
    } else {
        printf ("Error: La pila está llena.\n");
    }
}

int pop(Pila *pila) {
    if (!is_empty (pila)) {
        return pila->items [(pila->top)--];
    } else {
        printf ("Error: La pila está vacía.\n");
        exit (EXIT_FAILURE);
    }
}

bool is_empty (Pila *pila) {
    return pila->top == -1;
}

int main() {
    Pila pila;
    init (&pila);

    push (&pila, 10);
    push (&pila, 20);

    printf ("Elemento extraído: %d\n", pop (&pila));
    printf ("Elemento extraído: %d\n", pop (&pila));

    return 0;
}
```

Ventajas: Simplicidad en la generación de código intermedio.

Facilita la implementación de lenguajes que permiten revisión o un gran manejo de variables locales.

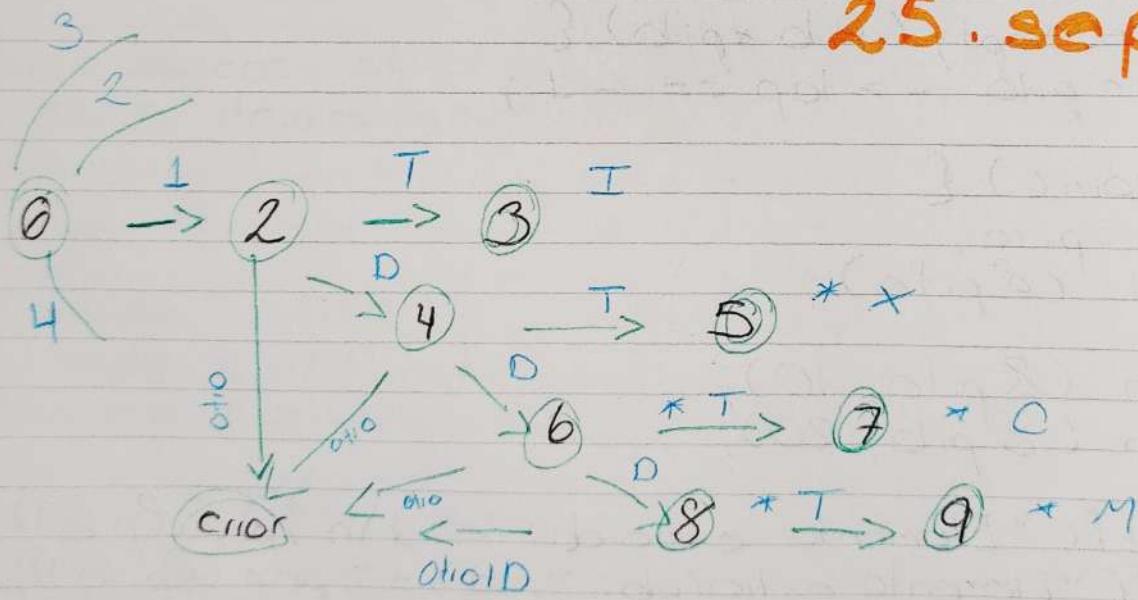
Gonzalez Morales Victor

23-sep

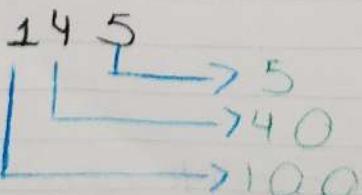
Aplicación: Este concepto se usa comúnmente en intérpretes y máquinas virtuales, como lo Java Virtual Machine y la P-Code machine del lenguaje Pascal. En estos casos, el código fuente es traducido a un conjunto de instrucciones para una máquina de pila abstracta que luego es ejecutado o interpretado.

En resumen, una máquina de pila abstracta es una representación teórica que simplifica la manera en que un compilador o intérprete ejecuta código, utilizando una pila para gestionar operaciones y resultados, lo que facilita el análisis, la optimización y la ejecución de programas.

25.sep



{ 1, 2, 3 }

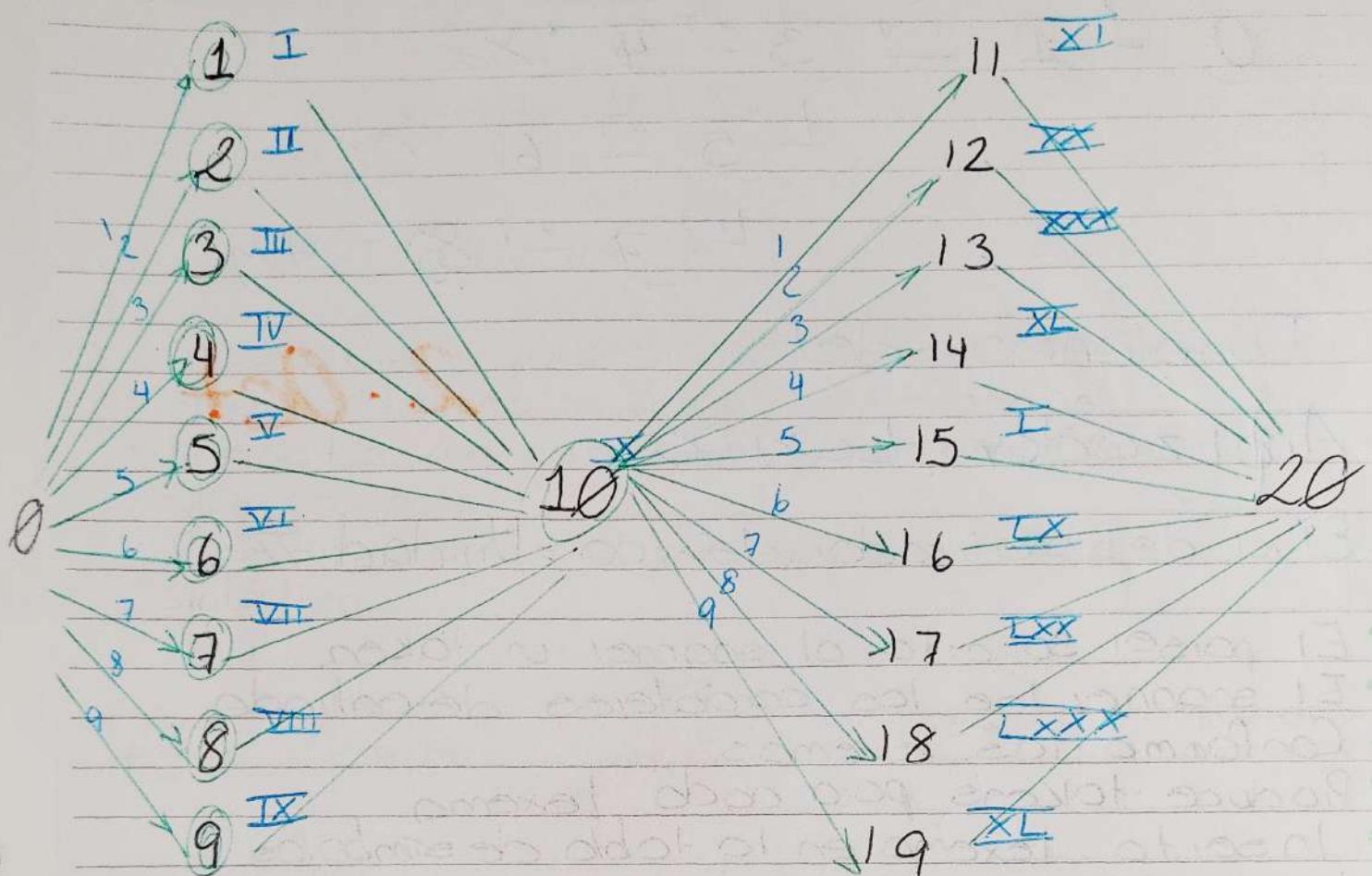


→ 1000  
2000  
3000

1 4  
1 1 1 1 ↑  
1 1 1 ↓  
CD X L V  
CXLV

Gonzalez Mardes Victor

26-SEP



$$2 \text{ I} \\ 1 - 3 = 4 * X$$

$$\cancel{2} \cancel{3} T \\ 1 \cancel{4} 0$$

$$L^{1-4} 5 - 6 * C$$

$$L^{-4} 7 = 8 * M$$

$$0^3 - 1^4 - 3 * - 4 * \cancel{XXX}$$

$$L^{-4} 5 * - 6 * CCC$$

$$L^{-4} 7 * - 8 * MMM$$

Gonzalez Morales

victor

0 -<sup>2</sup> 1  $\underline{-4}$  2 II  
0 -<sup>2</sup> 1  $\underline{-4}$  3  $\cancel{+T}$  4  $\cancel{+XX}$   
L<sup>-4</sup> 5  $\cancel{+T}$  6  $\cancel{+CC}$   
L<sup>-9</sup> 7  $\cancel{+T}$  8  $\cancel{+MM}$