Apuntes de Compiladores

Mauricio Elian Delgadillo Garcia

Índice general

Capítulo 1

Conceptos Introductorios

Como se sabe, la computadora solo entiende 1's y 0's (Lenguaje Maquina)

(IMAGEN)

En el principio, el programador debía escribir su código (Programa) en Binario.

1.1. 1er Paso El Assembler (ASM)

Lo que pasa en el Assembler es que dado una especie de alias dandole una instrucción.

En vez de escribir : 101011010
Se escribiría : MOV EAX, EAX

La conversión del ASM a lenguaje Maquina (Binaria) es 1 a 1, osea equivalente.

1 Instrucción ASM = 1 Instrucción (del lenguaje) maquina

mientras un lenguaje esta mas cercano al lenguaje maquina, se dice **Lenguaje de Bajo Nivel**

(IMAGEN)

El primer lenguaje de programación de Alto Nivel fue FORTRAN (**Form**ula **Tran**slation) el cual se tardo 18 años en completarse (El compilador)

(IMAGEN)

Un compilador de un lenguaje de Alto Nivel traduce una instrucción del lenguaje a N instrucciones maquina.

(IMAGEN)

Las 3 instrucciones básicas: IF - THEN, IF - THEN - ELSE, WHILE (y la Asignación x:=3;)

Capítulo 2

Lenguaje Intermedio

/* Lenguaje Intermedio = Intermediate Language = IL */
Recordemos que:

(IMAGEN)

El IL es un lenguaje de bajo nivel, muy cercano al Assembler, pero no es entendido por ninguna computadora del mundo.

2.1. Porque usar I.L.?

Un lenguaje de alto nivel, es muy complicado para llevarlo al lenguaje maquina. Por este motivo, el diseñador del compilador se inventa un lenguaje que le permite, mas fácilmente hacer la traducción:

(IMAGEN)

Otra razón de la asistencia del I.L., es la de poder generar archivos ejecutables en distintas **plataformas de Hw y Sw** (Arquitectura propia de la computadora, su lenguaje maquina y sistemas operativos)

/* Archivo Ejecutable = que es capaz de correr solo (Stand-alone)

El archivo ejecutable (Aplicación o App) contiene código maquina e.g. Windows el .exe

(IMAGEN)

*/

Ademas, que es la moda Actual, al I.L. se le crea un interprete que ejecute el lenguaje intermedio generado. A este Interprete se lo llama Virtual Machina o V.M.

//V.M. = Computadora Imaginaria

Cual es el problema de los V.M.?

La V.M., obviamente es una APP (un ejecutable) escrito para la plataforma donde va a correr (e.g. Windows, Linux, MacOS)

Por ejemplo JAVA:

(IMAGEN)

En Windows instalo la V.M. de Java, llamado Java Virtual Machine (J.V.M.), que viene en el archivo Java.exe

(IMAGEN)

Los IDE's no compilan nada ni corren ningún programa, Para Linux, necesitamos el ejecutable Java

(IMAGEN)

2.2. El Codigo de 3 direcciones

Es el I.L. mas utilizado en la compilación. En realidad, todos los I.L.'s se basan en el.

Se llama de 3 direcciones, porque a lo sumo cada una de sus instrucciones ocupa 3 direcciones de memoria. Por direcciones de memoria entendemos a variables, procedimientos etiquetas

Ejemplo:

(IMAGEN)

2.2.1. Operaciones Arimeticas y Logicas

Toda operación aritmética o lógica, almacena su resultado en una variable:

$$\text{Var1} = \text{Var2 } \textbf{oper.} \text{ Var3}$$

- Var1,Var2,Var3 = Variables
- oper. = operaciones = $\{*, +, -, /, mod, and, or\}$

Ejemplo:

$$x = z - y$$

Las operaciones (cálculos) se hacen entre variables.

Por ejemplo:

$x = z + \underline{2} \rightarrow 2$ es una constante no puede ejecutar esta operación, debe ser una variable.

$x = -y$ //operación minus

También se permite la asignación simple:

$p = 25$ //asignación a una ctte.
$x = p$ //asignación a una variable.

En el caso lógico

$\{\sim, \vee, \wedge\}$ se puede generar cualquier conectivo entre ellos.

p = not q
q = z and x
w = p or y

No existe tipo de datos, son solo números, no existe el **boolean** en si.

Como el C3 solo maneja números, hace un tratamiento lógico, usando el criterio del lenguaje C.

Cuando lee	Cuando escribe
Si es $! = 0 \rightarrow true$	1 = true
Si es $== 0 \rightarrow false$	0 = false

Por ejemplo:

x = 20
y = 0
z = x or y

Cuanto vale z?

x = 20 != 0 = true
y = 0 == false
z = true or false = true = 1
Respuesta: z = 1

2.2.2. Usando operaciones relacionales (OPREL)

// $<, >, \leqslant, \geqslant, =, \neq$

$$x = (y \geqslant z) \rightarrow \text{x = valor (0 o 1) de que } y \text{ es mayor o igual que } z$$

Por ejemplo:

$$t_1 = 1$$
$$t_2 = 5$$
$$z = (t_1 = t_2) \text{ //z = 0 (false)}$$
$$y = (t_1 \geqslant t_2) \text{ //y = 1 (true)}$$

2.2.3. Incremento - Decrementos

$$\text{x} = 2$$

INC x //Equivalente a x++ $t_1 = \text{x } //t_1 = 3$

DEC x //Equivalente a x - - $t_1 = \text{x } //t_1 = 1$

2.2.4. Saltos (Jump's or GOTO's)

Saltos Incondicionales

GOTO Etiqueta

Las etiquetas pueden ser :

$$E_1: \quad E_2: \quad E_3: ...$$

Las etiquetas no pueden tomar cualquier nombre

Saltos Condicionales

Un salto condicional se hace usando el valor de verdad de una variable. Entonces, Se tienen dos posibles caminos:

- if(var = 1) $\Rightarrow$ GOTO etiqueta
  //Si var es true (osea distinto de 0) ira a la etiqueta
  //Si var es false (osea igual a 0) esta linea no se ejecuta, osea seguiria bajando

- if(var = 0) $\Rightarrow$ GOTO etiqueta
  //Si var es true (osea distinto de 0) esta linea no se ejecuta, osea seguiria bajando
  //Si var es false (osea igual a 0) ira a la etiqueta

Ejemplo:
Leer un numero N (Read(N)) y mostrar N asteriscos en la pantalla

```
Introduzca N: 5
*****
```

Solución:

En Delphi:

```
writes("Introduzca N:")
READ(N)
for i=1 to N Do{
   writes(*)
}
```

En C3:

```
     writes("Introduzca N:")
     READ(N)
     i = 1
E1:  t1 = (i ⩽ N)
     if(t1 = 0) ⇒ Goto E2 //Va a E2 si t1 es False
     writes("x")
     inc i
     Goto E1
E2:
```

2.2.5. Operaciones del System (S.O.)

- Para la Salida

```
–Writes("Mensaje")
//1 solo mensaje
//(String ctte.)

–Write(var)
//1 sola variable

–NL
//New Line
//(para que el cursor
//baje a la siguiente
//linea)
```

- Para la entrada

```
–Read(var)
//1 sola variable
```

Ejemplos:

Codigo-3	Resultado
β = paso en blanco x = 4 Writes("El valor de x es β") Write(x)	El valor de x es 4
β = paso en blanco x = 4 Writes("El valor de x es β") NL Write(x)	El valor de x es 4

- Leer un numero N y calcular F = N!

Codigo-3	Resultado
Read(N) F = 1 i = 1 E1: t1 = (i⩽N) t2 = (t1 = 0) ⇒ Goto E2 F = F * 1 INC i Goto E1 E2: Writes("El Factorial de β") Write(N) Writes("β es β") Write(F)	El Factorial de 5 es 120

2.3. Procedimientos en el Codigo-3

Como todo programa, el código de tres direcciones (Codigo-3, C3) se divide en procedimientos, teniendo al `$MAIN` como el principal.

Es decir, un programa C-3 tiene como mínimo un procedimiento: `$MAIN`

2.3.1. Las instrucciones CALL y RET

Instrucción	Equivalente (Java)
CALL Proc	//Proc();
RET	// Return;

2.3.2. Las variables temporales ("Auxiliares" del compiladores)

Las variables temporales son "variables" creadas por el compilador para realizar calculos intermedios durante la traducción. Las variables temporales, también se crean en la pila de la aplicación y son de uso local.

Programador	Codigo - 3

`int x1,x2,y;`
`x1 = (y*3)+x2-1;`
	`t1 = 3`
	`x1 = y * t1`
	`t2 = x1 + x2`
	`t3 = 1`
	`x1 = t2 - t3`

El programador definió x1,x2,y	El compilador crea sus propias variables temporales (t1,t2,...)

Las etiquetas son globales

Es decir, las etiquetas son únicas.

Proc Lectura
E1: x = 10
⋮
Proc Calculo
E1: t1 = 5

`Error!`, si ambas pertenecen al mismo programa, se debe poner distinta enumeración

Ejemplo: El triangulo, Leer N (Validar que sea >0) y mostrar en consola un triangulo de N lineas de asteriscos, e.g. N = 4

```
*
* *
* * *
* * * *
```

Solución

```
Proc Lectura
     t1 = 0
E1:  Writes("Cantidad de Lineas?β")
     Read(N)
     t2 = (N ⩽ t1)
     if(t2=1) ⇒ Goto E1
     RET
```

```
Proc Linea
     i=1
E2:  t1=(i ⩽ k)
     if(t2=0) ⇒ Goto E3
     Writes("*")
     INC i
     Goto E2
E3:  NL
     RET
```

```
$MAIN
        CALL Lectura
        k = 1
 E4:    t1=(k ⩽ N)
        if(t1=0) ⇒ Goto E5
        CALL Linea
        INC k
        Goto E4
 E5:    RET
```

2.4. Esquema de Traducción

Consiste en estrategias que el compilador debe seguir para realizar la traducción de las construcciones de programación (if, while, for, ...)

2.4.1. Las expresiones (booleanas y aritméticas)

/* expr = expresiones aritméticas
exprBoole = expresiones booleanas */

Las expresiones tradicionales por el compilador usando un algoritmo ideado por cada diseñador. En muchos casos, se aplica de la optimizacion.

Por este motivo, dejamos la traducción de las expresiones al libre albedrio.

Si anotamos → `C3-Expr(ti)`
//ti = temporal i-esima

Es el codigo-3 de la expr, cuyo resultado esta en el temporal ti

/* Usamos ti y no t2, t3 o t4... porque no sabemos cual es el numero de la temporal final */

(∘) Por ejemplo

Se tiene la Expr1

$$y * 3 + 4$$

convertir a C3

Solución

```
t1 = 3
t2 = y * t1
t3 = 4                →   C3 - Expr1(t4) //El resultado esta en t4
[t4] = t2 + t3
```

(∘) Convertir a C3 la siguiente exprBoole1

$$(x \leqslant 0)\ and\ z$$

Solución

```
t1 = 0
t2 = x ⩽ t1          →   C3 - exprBoole1(t3) //El resultado esta en t3
[t3] = t2 and z
```

2.4.2. La Asignación

En un lenguaje de Alto nivel

$$x = expr;$$

Escribir un esquema de traducción C3 para este construcción.

Solución

```
C3 - Expr(ti)
   x = ti
```

Por ejemplo, usando el esquema de traducción anterior, Convertir a C3.

$$z = 2 * y - x$$

Solución

```
t1 = 2
t2 = t1 * y        →   C3 - Expr(t3)
[t3] = t2 - x
   z = t3          →      var = t3
```

2.4.3. Traducción para el If y While

Java		C-3

if(exprBoole){
Sentencia;
}
	→	C3–ExprBoole(ti)
		if(ti = 0) ⇒ Goto Ef
		C3 − Sentencia
		Ef:

(∘) Ejemplos:

Java		C-3

`if(x<z){`
`x = z + x;`
`}`
	→	`t1 = (x < z)`
		`if(t1 = 0) ⇒ Goto E1`
		`t2 = z + x`
		`x = t2`
		`E1:`

`if(x>0){`
`if(z == 5){`
`z++;`
`}`
`}`
	→	`t0 = 0`
		`t1 = (x > t0)`
		`if(t1 = 0) ⇒ Goto E1`
		`t2 = 5`
		`t3 = (z == 5) ⇒ Goto E2`
		`INC z`
		`E2:`
		`E1:`

(∘) Escriba el esquema de traducción para el if-else

Java		C-3

`if(exprBoole){`
`Sentencia1;`
`}`
`else{`
`Sentencia2;`
`}`
	→	`C3–ExprBoole(ti)`
		`if(ti = 0) ⇒ Goto Ef1`
		`C3–Sentencia1`
		`Goto Ef2`
		`Ef1:`
		`C3–Sentencia2`
		`E2:`

(∘) Para el while

Java		C-3

`while(ExprBoole){`
`Sentencia;`
`}`
	→	`Ef0:C3–ExprBoole(ti)`
		`if(ti=0) ⇒ Goto Ef`
		`Sentencia`
		`Goto Ef0`
		`Ef:`

2.5. Representación del Codigo-3

Cada instrucción del C-3 se represanta con un:

`Block` (Registros, Record, Struct, Class con todos los campos public)

Nota.- Los campos de este `Block` son de tipos primitivos (enteros)

La cantidad de campos que se utiliza bautiza la representación (de nombre)

- Terceto = 3 campos → Menos memoria, algoritmos mas complicados
- Cuadrupla = 4 campos → Cant. media de memoria, algoritmos termino medio (la asignada para la V.M.)
- Quintupla = 5 campos → Mucha memoria, algoritmos muy simples (Deprecated)

```
Class Cuadrupla{
  public int Op_code; //I.L. de Java
  public int Dir1, Dir2, Dir3; //Direcciones
}
```

Graficamente:

Cuadrupla = | Op_code | Dir1 | Dir2 | Dir3 |

//Op_code = **Op**eration - **Code**

Obviamente, no todas las instrucciones usan las tres direcciones. En estos casos, simplemente el campo `Dir` no utilizado se ignora (I don't care) y se denota con un underscore ("_")

Ejemplo:

x = 15 → | OpAsignacion | Dir x | 15 | _ |

El que diseña el compilador, crea el I.L., Una vez definidas las operaciones del C-3, usualmente define constantes para el "Opcode".

En Delphi	En Java	El programador hace	

Const
opMAS = 1;
opMENOS = 2;
opAND = 3;
	Public static final opMAS = 1;
	Public static final opMENOS = 2;
	(etc)
		OpAND	

		↓	El compilador lo sustituye por su valor
		3	

La inserción de los campos en la cuadrupla se leen de Izquierda a Derecha, tal como lo muestra textualmente la instrucción

Ejemplo:

x = y and z	OpAND	x	y	z	Operación
z = −x	OpMINUS	z	x	_	Operación
E1:	Etiqueta	1	_	_	Etiqueta
Goto E2:	OpGoto	2	_	_	Jump

Al decir `Op` es que va a efectuar algo la maquina, no ponerlo a las etiquetas!

El `IF` que se utilizamos en C-3 no es un If de verdad, solo se usa para representación. En el salto se recomienda tener dos Opcodes (`IF var = x` → `Goto E`) `OpIF0` y `OpIF1`

Ejemplo:

`IF z = 0 ⇒ Goto E5`	OpIFO	z	5	_
`IF z = 1 ⇒ Goto E9`	OpIF1	z	9	_

5 y 9 Son las Etiquetas E5 y E9 respectivamente.

Entonces un programa en C-3 es una Collection (Vector, Lista, Archivo,...) de **Cuadruplas**.

Por Ejemplo:

```
    x = 27                    0
    y= 1                      1
    p = x and y               2
    if(p = 0) ⇒ Goto E1       3
        writes("Hola")        4
E1:                           5
```

OpAsignNum	x	27	_
OpAsignNum	y	1	_
OpAnd	p	x	y
OpIF0	p	1	_
OpWrites	Dir. de “Hola”	_	_
Etiqueta	1	_	_

2.6. Tabla de Simbolos

/* Tabla de Simbolos = T.S. = Symbol Table = S.T. */

Tradicionalmente, se dice tabla de símbolos a un conjunto de tablas, no a una sola. Pero, para evitar esa ambigüedad, actualmente se llama: **“Administrador de Tablas de Símbolos”**. Para V.M. (Virtual Machine) que se quiere implementar, utilizaremos dos Tablas de Símbolos.

`TSID` = Tabla de Identificadores
`TSS` = Tablas de String’s ctte /*Println(“Hola”); Hola → String ctte*/

2.6.1. La T.S.I.D.

La **TSID** utiliza tuplas cuyos atributos son:

NombreID	ValorI	ValorD	CantTMP

CantTMP = Cantidad de Temporales

En esta tabla se almacenan las variables y los procedimientos.

1) Si la tupla es una variable:

   - ValorF contiene el tipo (es siempre negativo, Integer = -2, Boolean = -3)
   - ValorI mantiene el valor actual de la variable
   - CantTMP = No se usa

2) Si la tupla es un Procedimiento:
   ValorI y ValorF almacenan, respectivamente el inicio y el fin de su C3

3) Para el campo ValorF se puede saber si la tupla es un Procedimiento o Variable:

   - ValorF $< -1 \Rightarrow$ Variable
   - ValorF $\geqslant -1 \Rightarrow$ Procedimiento

Ejemplo:

TSID

	Nombre	**ValorI**	**ValorF**	**CantTMP**
0	"x"	?	-2	_
1	"Lectura"	0	2	2
2	"y"	?	-3	_
3	"$Main"	3	6	1

int x;
boolean y;

Codigo-3 (Vector de Cuadruplas)

0	_			
1	_			
2	Ret			
3	Call	Lectura		
4	_			
5	_			
6	Ret			

Cuadrupla C3 [ ]

2.6.2. La T.S.S.

// T.S.S. = Tabla de String Ctte

Es simplemente una tabla de un solo atributo y almacena los string's cttes del programa

T.S.S.

0	"Hola Mundo"
1	"*"
2	"El factorial es "
3	"bye"

2.6.3. Codificaciones de las direcciones

Cuadrupla = | OpCode | Dir1 | Dir2 | Dir3 |

Si se refiere a un temporal (`t1,t2,...`) o una etiqueta, simplemente se le anota su numero.

`t2 = t4 + t7` $\Longrightarrow$

OpMas	2	4	7

// t2 t4 t7

`Goto E9` $\Longrightarrow$

OpGoto	9	_	_

// E9

`if (t4 = 0)` $\Rightarrow$ `Goto E5` $\Longrightarrow$

OpIf0	4	5	_

// t4 E5

Si se refiere a un identificador (ID) (Variables o procedimientos definidos por el programador), se usa el indice de la tabla negativo

T.S.I.D.

	Nombre			
0	"x"			
1	"Base"			
2	"Fact"			

`x = t4 + Base`	$\Longrightarrow$	OpMas	0	4	-1
`Call Fact`	$\Longrightarrow$	OpCall	-2	_	_
`t2 = Base - Base`	$\Longrightarrow$	OpMenos	2	-1	-1

Porque `0,-1,-2` ? son los indices en el **T.S.I.D** de dichas variables/procedimientos, si es positivo es un temporal o una etiqueta. **No olvides que las OpCodes son una función, no una Cadena!**

De la misma forma se hace con las String Ctte. (Operación: write o writes). Por ejemplo, Usando la **T.S.S.** del principio:

`writes("Bye")`	$\Longrightarrow$	OpWrites	-3	_	_
`write(Base)`	$\Longrightarrow$	OpWrite	-1	_	_
`writes("*")`	$\Longrightarrow$	OpWrites	-1	_	_

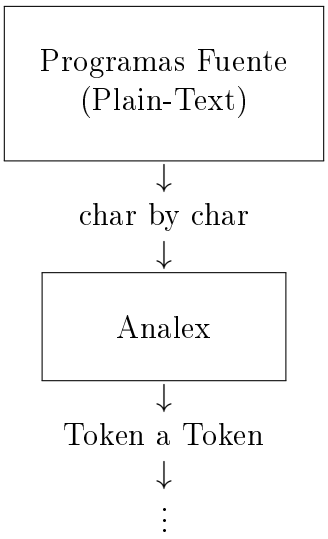
Capítulo 3

Analizador Léxico

/* Tambien se lo conoce como scanner, lexer o Analex */

El analizador es el encargado de:

- "Descompone" el programa fuente en piezas léxicas, llamado Tokens
- "Elimina" los comentarios del programa fuente
- "Instala" (insert) los string cttes. a la TSS
- Si el lenguaje es no tipado (no tiene tipos) instala los ID's a la TSID. (Si el lenguaje es tipado, la instalación de ID's lo hace Parser)



/* Plain-Text = Texto (archivo) plano = Que se puede leer en el block de Notas */

Los Caracteres Char: En un principio cada computadora definía los chars a su antojo:

Computadora 1	Computadora 2
A = 0	A = 2
B = 1	B = 3
C = 2	C = 4
D = 3	D = 5

La Computadora 1 envía el text "DDC" a la Computadora 2

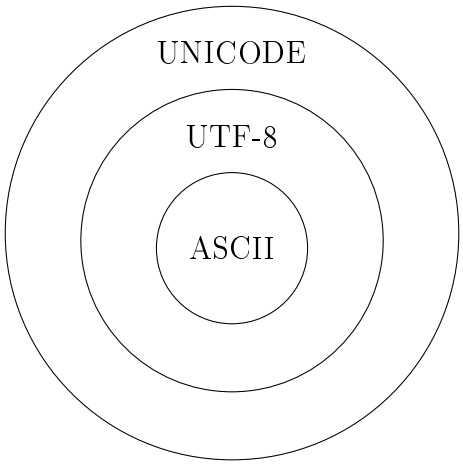
$$\text{Comp. 1} \xrightarrow{\text{DDC (3,3,2)}} \text{Comp. 2} \rightsquigarrow \text{recibe "BBA"}$$

Para evitar esto, crearon:

- Un código estándar de 7 bits llamado A.S.C.I.I. (Codigo de 0 a 127)

$$\begin{aligned} 0 - 31 &= \text{Char no imprimibles} \\ 62 &= \text{"Space"} \\ &\vdots \\ 65 &= \text{A} \\ 66 &= \text{B'} \\ &\vdots \end{aligned}$$

- UTF-8 los char van desde 0 al 255
- Unicode (0 - 66535)

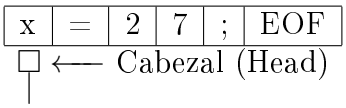


ASCII = 7 bits, UTF = 1 byte, UNICODE = 2 bytes

3.1. Manejo del Programa Fuente

Para la manipular el programa se utiliza un ADT (Maquina) llamado "**Cinta de Caracteres**" o multilineas o serializador

Para el Analex, la cinta se muestra asi:



Cada celda de la cinta aloja un char (carácter) y siempre la cinta termina en un char especial EOF (End Of File)

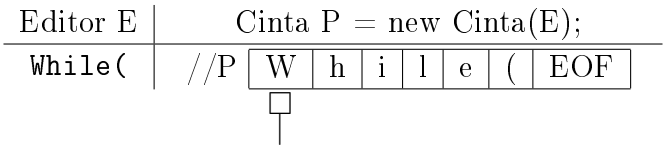
Por ejemplo una cinta vacia:

El Cabezal o Head permite leer el caracter o char que esta "mirando". La cinta puede trabajar con un editor, un archivo, un puerto de conexión en la red, etc.

Métodos de la Class Cinta

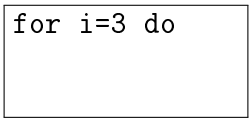
- `Public Cinta (Editor E)`
- `Public Cinta (String FileName)`
- `Public Cinta (Net algo)`

Por ejemplo:

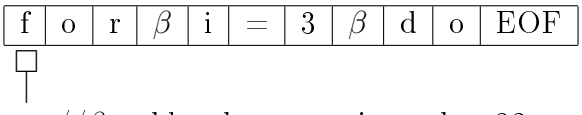


- `Public void init()` //Rewind
  //Mueve el cabezal a la primera celda
- `Public void Avanzar()` //Forward
  // Mueve el cabezal a la sgte. celda de la derecha
- `Public char cc()` //Current-Char (Caracter del cabezal)
  //Devuelve el char que esta viendo el cabezal

Por ejemplo: Usando la Cinta: `Alfa.txt`

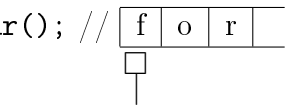



```
Cinta Q = new Cinta("Alta.txt");
```

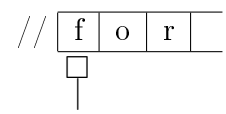


//β = blanck = espacio = char 32

- `char x = Q.cc();` //x = "f"
- `Q.avanzar();` //



- `x = Q.cc();` //x = "o"
- `Q.init();` //



El EOF es una ctte. y se la puede usar:

```
if(Q.cc() == EOF)
```

(○) Dado una cinta P, contar la cantidad de chars que hay en P (`int Contar(Cinta P)`, no contar al EOF). Ejemplo: usando la cinta Q anterior, `Contar(Q) = 10`

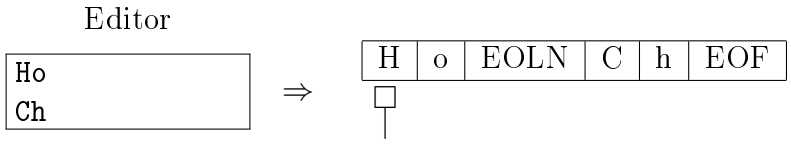
Solución:

```
int Contar(Cinta P){
  int i = 0;
  P.init();
  while(P.cc() != EOF){
    i++;
    P.avanzar();
  }
  return i;
}
```

3.1.1. El fin de linea (EOLN)

//EOLN = End Of Line

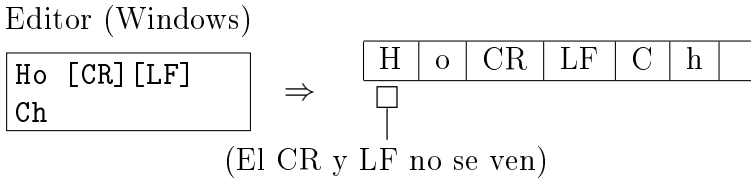
El EOLN hace una ilusión óptica en un entorno



Representación en Windows

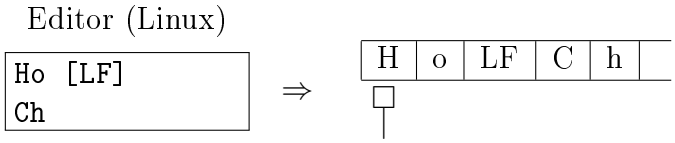
Microsoft Utiliza 2 Char (caracteres) para representar el EOLN

- Carry-Return = 13 char (Enter)
- Line-Feed = LF = 10 char (Vuelve al comienzo después de bajar)



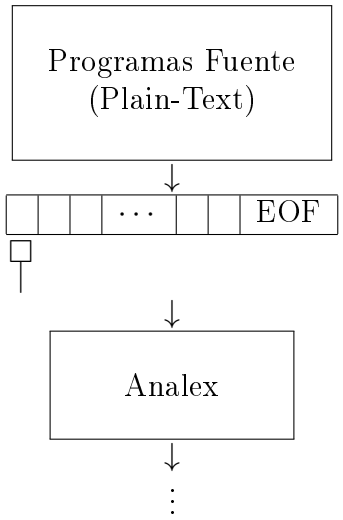
Representación en Linux

Solo utiliza el LF



Para la cinta no tiene problema en trabajar en ambos formatos. Sin importar el programa fuente, fuera hecho en Linux o Windows, la cinta representa al salto de linea con un solo char (de caracteres): EOLN

Con la cinta, la Arquitectura del compilador, empieza así:



3.2. Especificación de los Componentes Lexicos

//Componentes Léxicos = Tokens

Matematicamente, un token es un lenguaje regular que no contiene a la cadena vacía (λ)

Un alfabeto define los char's (letras o símbolos) que se van a utilizar. Un lenguaje es un conjunto de cadenas que se forma con los char's de Σ

Por Ejemplo:

$\Sigma = \{a, b, c\}$ y $L = \{\lambda, \text{“}ab\text{”}, \text{“}ccb\text{”}\}$

Un lenguaje es regular si puede ser especificado por una expresión regular (REGEX).

3.2.1. Definiciones Regulares

$Sol = a(a|e|i|o|u)^*b$ ó

$$algo = (a|e|i|o|u)$$
$$sol = a(algo)^*b$$

Permite especificar REGEX por partes, a traves de producciones. Una definición Regular no necesita especificar el alfabeto Σ

Operadores

- $|$ = lease “o” (Unión)
- . = concatenación
- * = 0 o mas (Estrellas de Kleene)
- $+$ = 1 o mas (Estrella de Kleene)
- $\rightarrow$ = lease “es” o “puede ser”

Ejemplos:

($\circ$) Especificar el Token NUM (Números enteros sin signo)

$$NUM = \{\text{“}0\text{”}, ..., \text{“}9\text{”}, \text{“}3564\text{”}, \text{“}800\text{”}, etc\}$$

Solución
`Digitos` $\rightarrow$ `0|1|2|3|4|5|6|7|8|9`
`NUM` $\rightarrow$ `Digitos`$^+$

($\circ$) Especificar el Token NumP (para los números enteros pares)

$$NUM = \{\text{“}0\text{”}, ..., \text{“}22\text{”}, \text{“}154\text{”}, \text{“}430\text{”}, etc\}$$

Solución
`Par` $\rightarrow$ `0|2|4|6|8`
`Digitos` $\rightarrow$ `0|1|2|3|4|5|6|7|8|9`
`NUM` $\rightarrow$ `Digitos`* `Par`

Recuerda: Los números (Enteros o Float) en compilación se anotan sin signo, -34 o +34 saldría ERROR

(○) Especificar el Token ID, el cual esta compuesto por solamente letras

$$ID = \{x, y, Area, Base, Altura, ...\}$$

`Letra` → `a|b|c|...|z`
`ID` → `Letra`$^+$

/* Cuando se anotan las letras en una definición regular (DEFREG), no es necesario escribir mayúsculas y minúsculas. Solo basta 1 de estas cosas.

UpperCase = Mayúsculas
LowerCase = Minúsculas

El compilador, según la política del lenguaje, hará una diferencia en los Case

- Si el lenguaje es Case-Sensitive, distinguirá las mayúsculas de las minúsculas. Por ejemplo: Java, C, C++, C# (`int a,A;` se refiere a distintos casos)
- Si el lenguaje no es Case-Sensitive, el compilador lleva todo a mayúsculas y trabaja internamente así. Por ejemplo: Pascal (`Var a,A: Integer;` se refiere al mismo ID)

*/

(○) Escribir una DEFREG para el Token IDX, el cual esta formado con solo letras, pero tienen dos formas:

Empieza con: A *algo* Z
Empieza con: Z *algo* A

Solución
`Letra` → `a|b|c|...|z`
`IDX` → `a Letra`* `z | z Letra`* `a`

3.2.2. Lexemas (Lexem's)

Simplemente, se llama lexemas a los Strings que pertenecen a un Token:

NUM = {... 315, 860, 1001, ...}
↑ ↖ ↑ ↗
Token Lexemas

Del Ejercicio de DEFREG, `ZBAA` es un lexema del token `IDX`

3.2.3. Números Reales

En compilación un número entero no es un número real (flotante)

$$2{,}0 \neq 2$$

Todo numero real si o si debe tener el **punto decimal (.)**, si no lo tiene se considera entero.

Números Reales Formales

Los números reales formales son aquellos que usan las siguientes 3 partes:

`Parte entera` `Punto(.)` `Parte decimal`

Ejemplo: 3.17 (V) .24 (X) → 0.24(V)

Números Reales Informales

Se puede obviar la `Parte entera` o la `Parte decimal` (pero no ambas)

Ejemplo: 35. (V) .24 (V)

(∘) Mediante una Def. regular, especificar. Los NUMR Formales (NUMR = Números reales):

Solución
`Digito` → `0|1|2|...|9`
`NUMR` → `Digito`$^+$ `.` `Digito`$^+$

Nota: Dado que las producciones `Digitos` → y `Letra` →, son muy utilizadas, no es necesario definirlas todo el tiempo

3.3. Diagrama de Transiciones (dt)

Puesto que los Tokens son Lenguajes Regulares, por un teorema, es posible reconocer utilizando un AFD (Autómata Finito Deterministico). Pero... Un compilador asume que el alfabeto (Σ) es infinito.

Para solucionar esto usamos un AFD que use definiciones regulares, llamado **diagrama de transiciones o dt.**

Las definiciones regulares permitidas en un dt solo agrupan **char's**. Es decir, no usan la Estrella de Kleene ni la concatenación.

3.3.1. El ANALEX como un jugador de sopa de letras

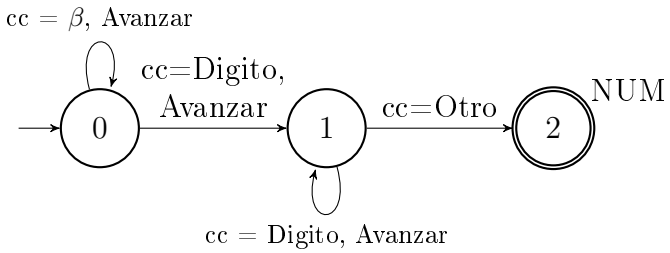
El ANALEX busca en la cinta (SCAN) los lexemas que pertenecen a un Token, por ejemplo:

3	8	7	β	A	B	β	$<$	$=$	EOF
NUM				ID		Op. Relacional			

(∘) En una cinta se encuentra un NUM dibuja un dt para reconocerlo. (NUM = números enteros sin signo)

	β	β	β	3	8	7	β	

Respuesta:



//cc = Caracter del Cabezal

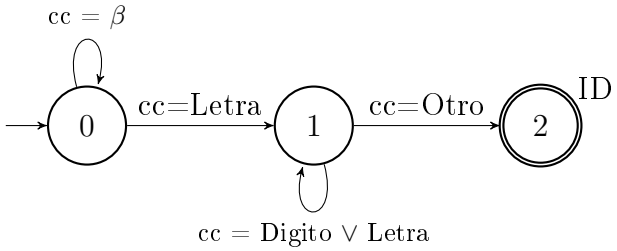
Cuando pasas de (1) a (2) gracias a "Otro" se dice: **lo que esta detrás de mi es un Número.**

(∘) En una cinta hay un ID que empieza con letra y luego le sigue una combinación de letras y dígitos.

	β	β	B	1	A	2	C	β	

Dibujar un dt para reconocer este Token.

Respuesta:



Cuando pasas de (1) a (2) gracias a "Otro" se dice: **lo que esta detrás de mi es un ID.**

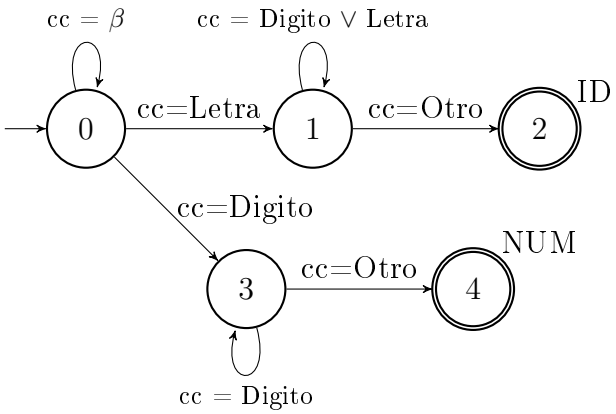
(∘) En una cinta hay un NUM o un ID, hacer un dt para reconocerlo.

	β	P	X	1	2	7	β	

ó

	β	β	3	1	5	β	β	

Solución:



Simplificación: Es posible eliminar "cc=" de las aristas

3.3.2. El Espacio

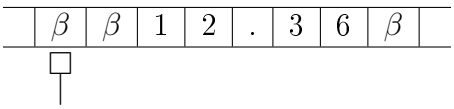
El espacio se usa para separar las palabras en el programa fuente. Según la Norma ASCII. El compilador toma como espacio a **tres caracteres.**

- β = barra espaciadora = ASCII 32
- TAB = tabulador = ASCII 9
- EOLN = fin de linea = LF = ASCII 10

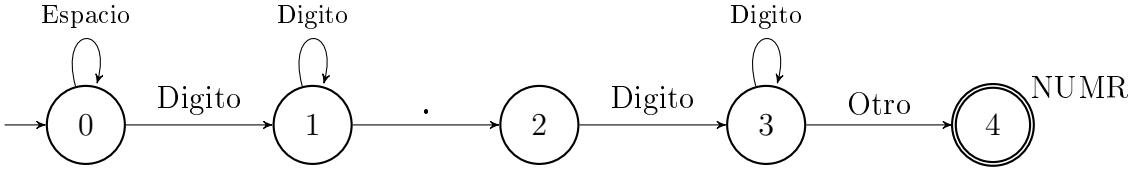
Entonces en vez de usar "β" usamos la producción

$$\texttt{Espacio} \rightarrow \beta|\texttt{TAB}|\texttt{EOLN}$$

(∘) Dibujar un dt que reconozca NUMR formal que esta en la Cinta



Solución:



3.3.3. Los Tokens ERROR y FIN

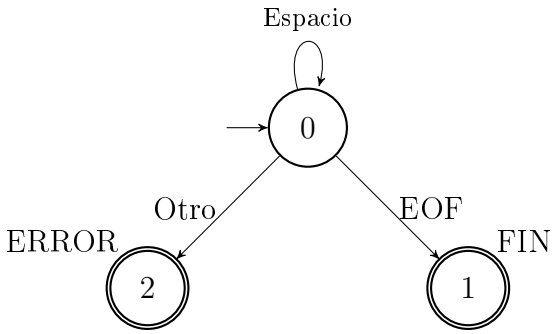
El Token ERROR

El dt, devuelve el Token ERROR cuando no es posible reconocer un Token

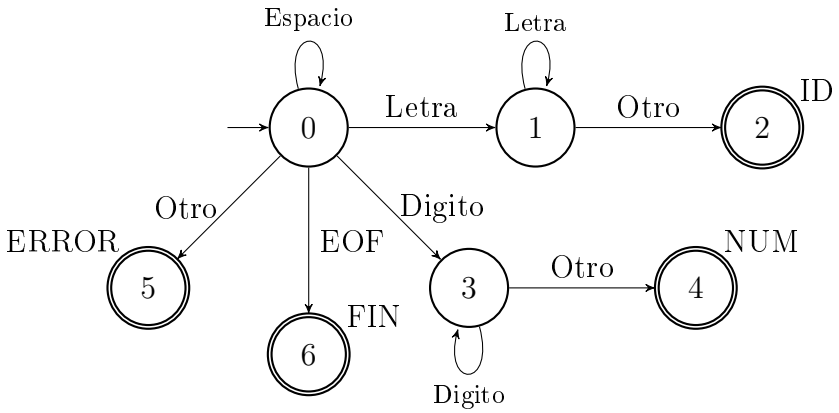
El Token Fin

Este Token es devuelto por el dt, cuando el **cc** alcanza al EOF.

La estructura general seria:



(∘) Dibujar un dt que reconozca NUM (enteros sin signo) e ID (solo letras)



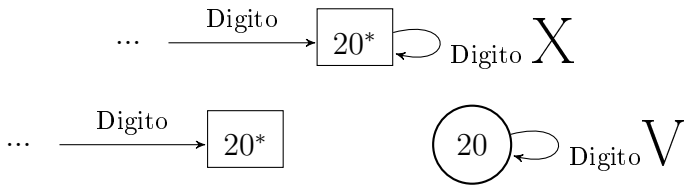
Recuerda:

- Todas las transiciones implican un `Avanzar`, excepto "Otro" y "EOF"
- `Espacio =` β`|TAB|EOLN`
- Uno puedo llamar al dt cuantas veces quiera
- Al llegar a un estado aceptado se dice: "Lo que esta detrás de mi es un ID, NUM, NUMR..."

3.4. "Reglas" de los dt

1) Existe 1 y solo 1 estado inicial (tradicionalmente el estado inicial es el 0)

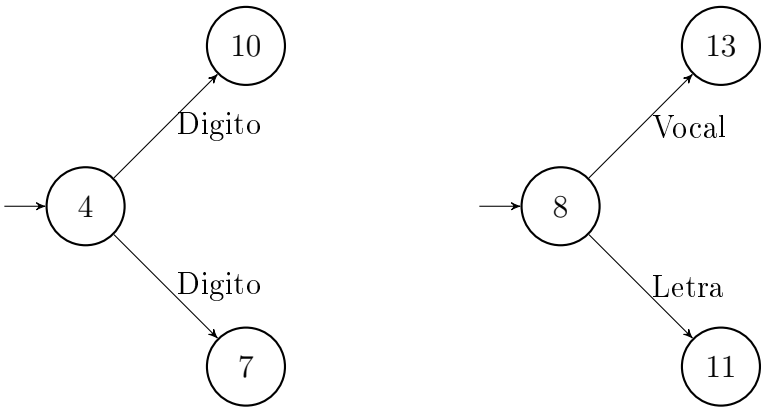
2) Todos los estados son diferentes.
   Error Común: Cuando el dt tiene muchos estados y aristas, el diseñador utiliza conectores (los conectores no son estados)



3) Si un estado tiene aristas salientes entonces tiene una arista rotulada con “Otro".
Errores Comunes:

- Usar “Otro” como una única arista
- Tener dos “Otro” saliendo de un estado
- Usar “Otro” como ciclo de un estado

4) Un dt no puede presentar un No-Determinismo (ambigüedad en las transiciones)
Errores Comunes:



No puede haber dos aristas que compartan el mismo conjunto (o subconjunto) de caracteres (Digito = Digito, Vocal $\subseteq$ Letra)

Para solucionarlo se debe especificar en las definiciones regulares las diferencias que tienen entre si:

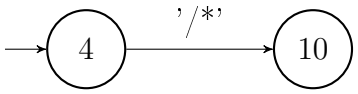
`Vocal = a|e|i|o|u`

`Letra = b|c|d|f|...|z` //Letras sin vocales

5) El dt debe tener al menos un estado final. Si no tiene un estado final, nunca finalizara.

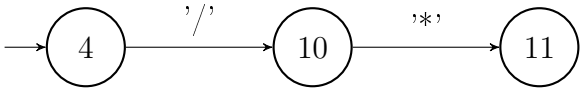
Errores Comunes:

- Definir mal una definición regular. **Recuerda:** el dt analiza char a char el cual le otorgan las definiciones regulares (`a|b|c|...|z`) estas no deben agruparse en mas de dos (`ab|cd|dd|..|za`) puesto que cc (Cabezal de la Cinta) no lo reconocería
- Poner mas de un char en la arista



Error: Hay dos char's

La solución seria crear un nuevo estado que separe ambos caracteres:



(∘) Dibujar un dt que reconozca Tokens usando solamente letras

	C	A	β	A	E	Z	β	B	

`IDV` → `termina en Vocal (CA,...)`
`ID` → `no termina en Vocal (AEZ, B,...)`

Solución:

`Const = B|C|D|...|Z` //no tiene vocales
`Vocal = A|E|I|O|U`