

ducción a la compilación

mos de traducción:

Traducir un programa fuente en un idioma
escribible en otro idioma y producir resul-
tos ejecutables del nuevo programa.

TRAN

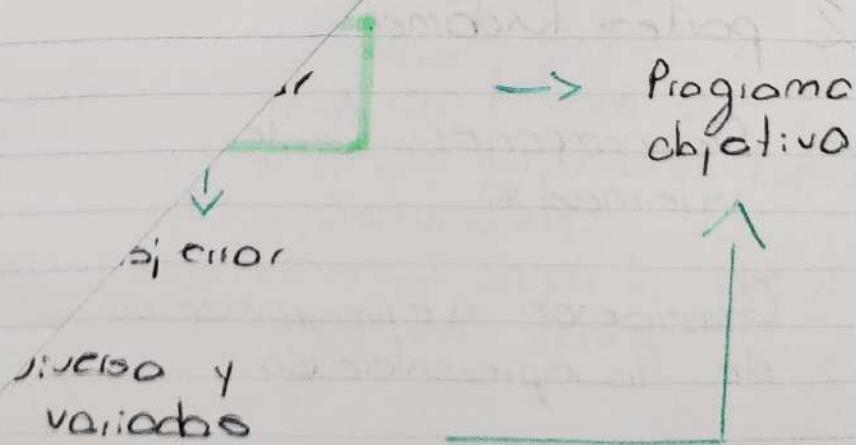
programa fuente y producir los resul-
tos ejecutables de ese programa.

Unidad

1

(M)

programadores traducen
(fuente) a otro (destino).



Introducción a la compilación

Mecanismos de traducción:

• **Compilación:** Traducir un programa fuente en un idioma o un programa ejecutable en otro idioma y producir resultados mientras se ejecuta el nuevo programa.

Ej: C, C++, FORTRAN

• **Interpretación:** Leer un programa fuente y producir los resultados mientras se entiende ese programa.

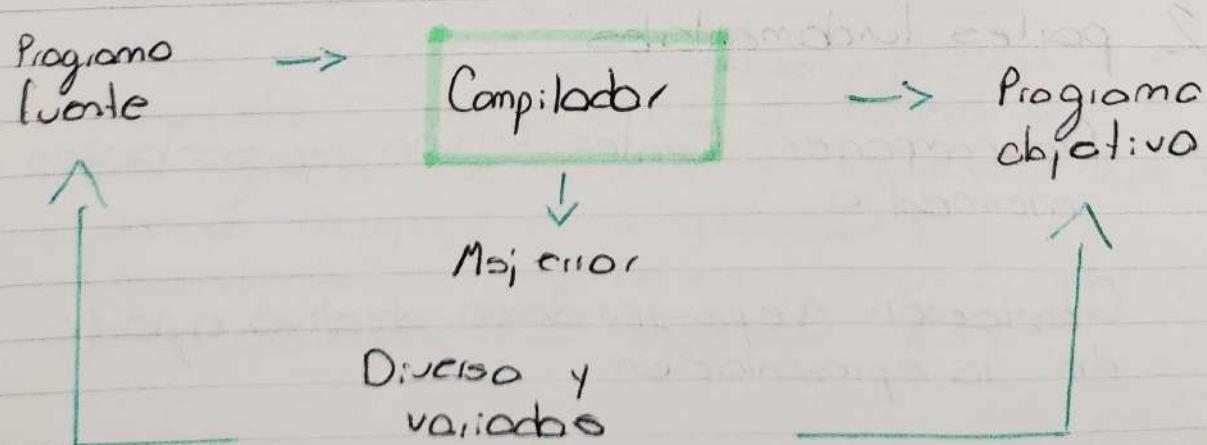
Ej: BASIC, LISP

• **Caso Study: JAVA**

1ro: Traducir a bytecode de java

2do: ejecutar por interpretación (JVM)

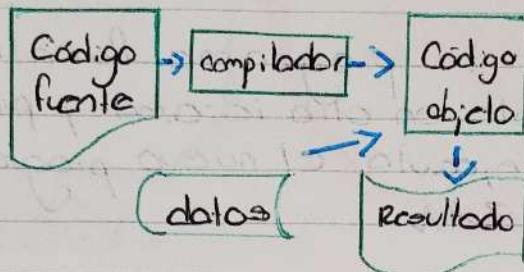
Propósito del compilador: Los compiladores traducen un programa escrito en un lenguaje (fuente) a otro (destino).



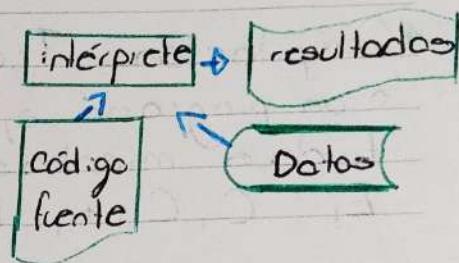
Gonzalez Morales Victor

Comparación de compilador / interprete.

COMPILADOR



INTERPRETE



Visión General

Ejecución rápido del programa; explora las características de la arquitectura

Fácil de depurar; flexible para modificar; independiente de la maq.

Ventajas

Pre-procesamiento del programa; complejidad

Sobrecarga en ejecución; Sobrecarga de espacio.

Deseventajas

EL MODELO

Los 2 partes fundamentales

Análisis: Descomponer fuentes en una representación intermedia

Síntesis: Generación de programación objetiva a partir de la representación

Centrarse en el análisis

notas importantes:

Hoy: hay muchas herramientas de software que usan el modelo de análisis-síntesis:

Editoras dirigidas a estructura/sintaxis: Sintaxis:
Forzar la introducción del código "sintácticamente correcto".

Impresoras estéticas: Versión estandarizada para la estructura del programa (es decir, espacios en blanco, sangría)

Verificadores estéticos: Una compilación "rápida" para detectar errores sencillos.

Interpretes: Ejecución "en tiempo real" del código
"línea a línea"

La compilación no se limita a los aplicaciones de lenguaje de programación

Formatadores de texto

LATEX y TROFF son lenguajes cuyos comandos dan formato al texto

Compiladores de silicio

Texto gráfico: Toman entradas y generan el diseño

Procesadores de consultas de base de datos:

Los lenguajes de consulta de bases de datos también son un lenguaje de programación

La entrada se compila en un conjunto de operación para acceder a la base de datos

Gonzalez Morales Victor

22-08-24

La tarea de análisis para la compilación (del programa Rente)

Tres fases:

Análisis léxico / léxico.

Escaneo de I-A-D para identificar tokens: Secuencia de caracteres que tienen un significado significativo.

Análisis jerárquico: Agrupación de tokens en una estructura significativa.

Análisis semántico: Comprobación para garantizar la corrección de los componentes.

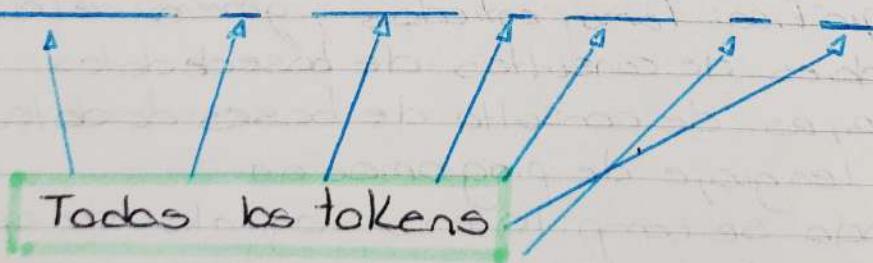
FASE 1 - Análisis Léxico

23-08-24

Análisis más fácil - Identificar tokens que son los bloques de construcción básicos

ej:

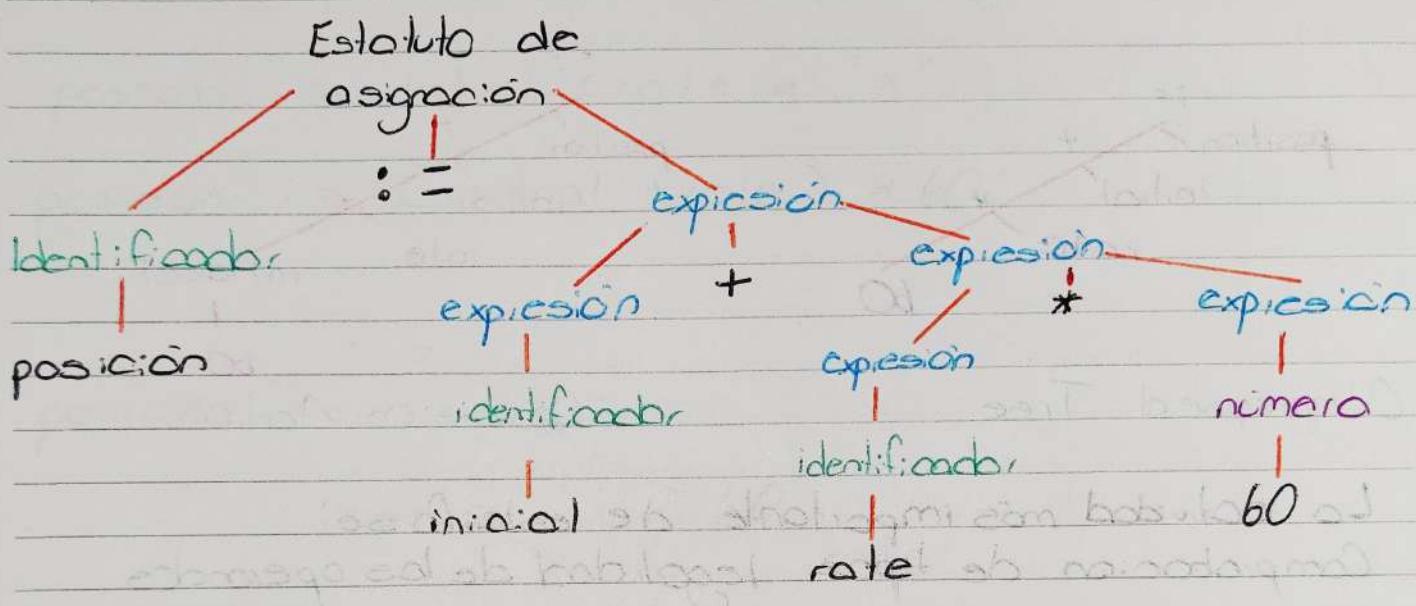
posición := inicial + rate * 60 ;



No se escanean espacios en blanco, saltos de linea, etc.

Fase 2: Análisis jerárquico alias Parsing o Análisis de Sintáctico

Para el ejemplo anterior, tendriamos Árbol sintáctico:



Los nodos de árbol se construyen usando una gramática para el lenguaje

¿Qué es una gramática?

Es un conjunto de reglas que gobiernan las interdependencias y la estructura entre los tokens.

estatuto

es estatuto de asignación, o while, o estatuto if, o...

asignación

es identificador, `:` = expresión;

expresión

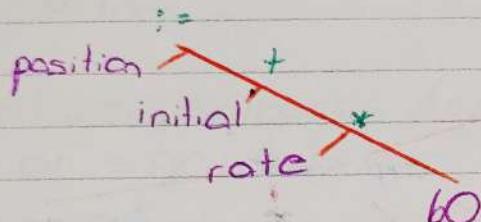
es (`expresión`), o `expresión + expresión`, o `expresión * expresión`, o `número`, o `identificador`, o...

González Morales Víctor

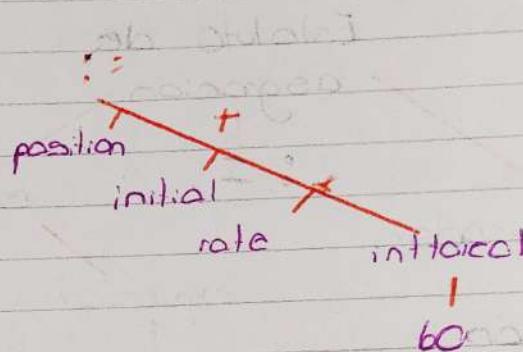
Fase 3: Análisis Semántico

Encontrar errores semánticos más complicados y operar la generación de código.

El árbol de análisis se complementa con acciones semánticas.



Compressed Tree



Concisión Action

La actividad más importante de esta fase es:
Comprobación de tipos, legalidad de los operandos

C) Por qué hemos dividido el análisis
de este modo?

Análisis léxico: escaneo lo entiendo, sus acciones lineales
no son recursivas.
Identificar solo "palabros" individuales que son los tokens del lenguaje

La recursividad es necesaria para identificar la estructura
de una expresión, como se indica en el árbol sintáctico
Verificar que los palabras estén correctamente
ensambladas en oraciones

Gonzalez Morales Victor
expresión

22-Ago

número

-20.8E06

initial \star position $+ 'x' \star rate \star :=$

position := initial + rate * 'x'

position := (initial + rate) * 60;

position :=
position \star expression * 60
expression
C expression)
expression

28 - Ago

Fases / actividades

de apoyo para el análisis.

Creación / mantenimiento de bloques de símbolos

- contiene información (almacenamiento, tipo, ámbito, argumentos) de cada token 'significativo' normalmente identificables
- estructura de datos creado / inicializado durante el proceso análisis léxico
- utilizado / actualizado durante el análisis y la síntesis posteriores

Gonzalez Morales Victor

28/10/8

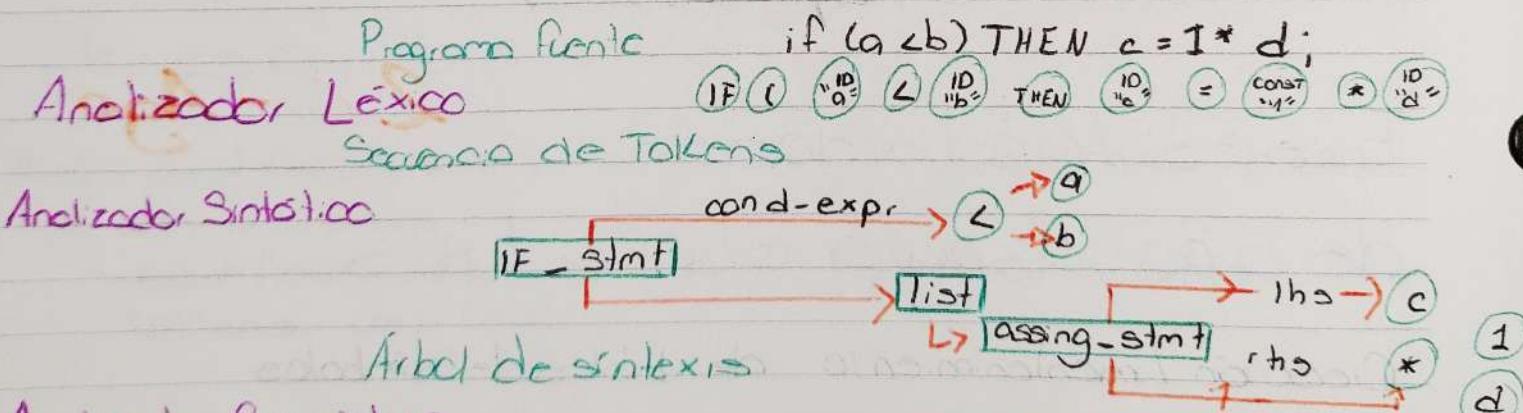
Tabla de Símbolos

Name	Type	Def/Used	other
avg	real.d	D	...
if	Keyword		...
num	int.d	D	...
sum	real.d	D	...
then	Keyword		...

Controlador de errores

- Detecion de d.ferentes errores que corresponden a todos los tipos
- CQué tipo de errores se encuentran durante la fase de análisis?
- CQué sucede cuando se encuentra un error?

Fases de un compilador moderno



Código de 3 direcciones

GE, a, b, L1
MULT, 1, d, c
L1:

Optimizador de Código

Código de 3 direcciones Optimizada

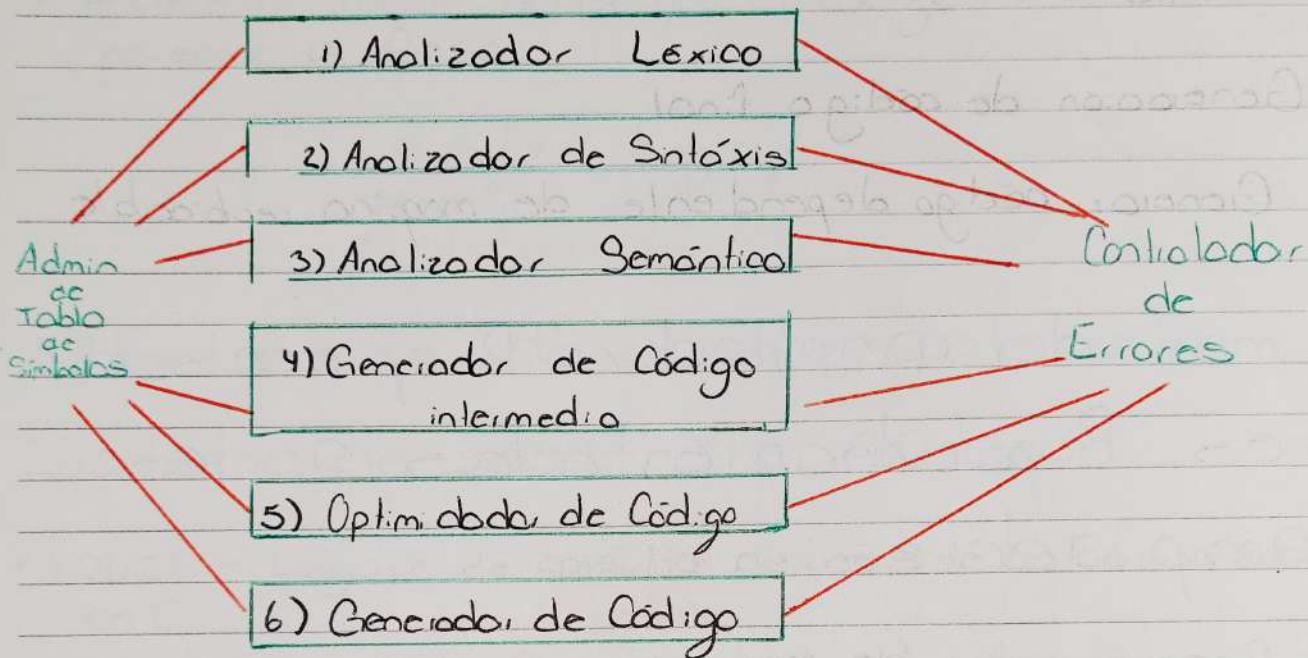
GE, a, b, L1
MOV, a, c,
L1

Generación de código

Código ensamblador

bad: R1, a
cmpl R1, b
jge L1
load: R1, a
store R1, c L

Las muchas fases de un compilador



Programa Fuente

1, 2, 3: Análisis - nuesta enfoque

La tarea de síntesis

29-08-24

para la compilación

* Generación de cod. go intermedio

- Versión del cod. go independiente de la arquitectura para una máquina abstracta
- Fácil de producir y hacer la generación de cod. go final independiente de la máquina.

Gonzalez Maroles Victor

* Optimización de código

28/10/8

- Encuentra formas más eficientes de ejecutar código
- Reemplaza código con instrucciones más óptimas
- 2 enfoques: Lenguaje de alto nivel y optimización de "mínimo"

* Generación de código final

- Genera código dependiente de máquina recibible

Primos del compilador: Proprietas cod. -
ores. Proporcionan entradas a los
compiladores.

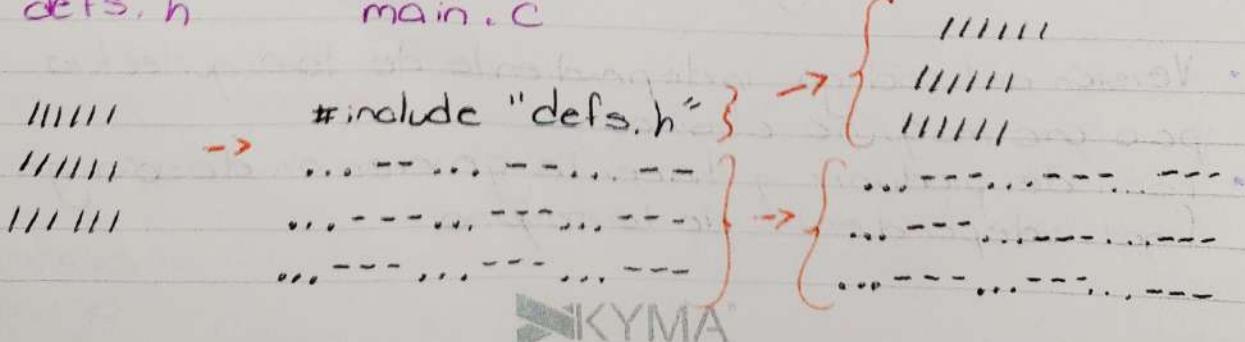
1. Procesamiento de macros:

```
# define en C: Sustituye el texto antes de compilar  
# define X 3  
# define Y A*B+C  
# define Z getchar()
```

2. Inclusión de archivos

#include en C - Trae otro archivo antes de compilar.

defs.h main.c



3) Preprocesadores racionales

- Aumentan lenguajes "antiguos" con construcciones modernas
- Agregan macros para if-then, while, etc.
- `#define` Puede hacer que el código C sea más parecido a Pascal

```
# define begin {  
# define end }  
# define then
```

4) Extensión de idiomas para un Sistema de bases de datos

- EQUEL - Lenguaje de consulta de base de datos incrustado en C.


```
## Retrieve (DN - Department.Dnum) where  
## Department.Dname = 'research'
```
- se procesa en:

`ingres -system ("Retr... research", -, -);`

- Una llamada a procedimiento en un lenguaje de programación.

Ensambladores

- Código de ensamblaje: Se utilizan nombres para las instrucciones y también se utilizan nombres para las direcciones de memoria

`MOV a, R1`

`ADD #2, R1`

`MOV R1, b`

Gonzalez Morelos Víctor

- Ensamble de pasos: 29108

- Primer paso: todos los identificadores se asig-
nan a direcciones de memoria (desplazamiento)
por ejemplo, sustituye 0 por a, y 4 por b
- Segundo paso: Producir código máquina reubicable.

001 01 00 00000000 * > 0001 01 00 00001111
0011 01 10 00000010 0010 01 10 00000010
0010 01 00 0000100 * 0010 01 00 00010011

* bit de reubicación

Ensambladores

Todavía no conocemos la dirección, por lo que el código fuente debe leerse dos veces.

Código fuente (ensamblado:)	Primer Paso de		Segundo Paso de	
	Dirección relativa	Instrucción mnemónica	Dirección relativa	instrucción memoria
John Start 0				
Using *,15				
Load 1, Five	0	L1,-(0,15)	0	L1,16(0,15)
Add 1, Four	4	A1,-(0,15)	4	A1,12(0,15)
Sto 1, Temp	8	ST1,-(0,15)	8	ST,20(0,15)
Four DC F'4'	12	4	12	4
Five DC F'5'	16	5	16	5
Temp DS 1F	20	-	20	-
End				

Cargadores y Editores de Enlaces

- Cargador: toma código máquina reubicable, obtiene las direcciones y coloca las instrucciones a literales en la memoria.
- Editor de enlaces: toma muchas programas (reubicables) de código máquina (con referencias cruzadas) y produce un solo archivo.
 - Necesita realizar un seguimiento de la correspondencia entre los nombres de las variables y las direcciones correspondientes en cada fragmento de código.

La agrupación de fases

Front End : Análisis + Generación de Código intermedio vs

Back end : Generación de Código + Optimización

Número de pasadas:

Una pasada: requiere lectura / escritura archivos intermedios
Menos pasadas: más eficiente

Sin embargo: menos pasadas requieren una administración de memoria más sofisticada y una interacción de fase del compilador.

Ejemplo...

Gonzalez Morales Victor

29/08

Sistema de procesamiento de lenguaje

Programa fuente

1 Pre-procesador

2 Compilador

3 Ensamblador

4 Código máquina
reducible

5 Cargadores y
editores de enlace

Ejecutable

Biblioteca
archivos
objetos
reducibles

Herramientas de construcción de compiladores

Generadores de analizadores: Producen analizadores
de sintaxis

Generadores de escáner: Producen analizadores
léxicos

Motores de traducción dirigidos por sintaxis:
Generan código intermedio

Generadores automáticos de código: Generan
código real

Motores de flujo de datos: Apoyan optimización

Herramientas

- Existen herramientas para ayudar en el desarrollo de algunas etapas del compilador
- Lex(Flex) - Generador de analizadores léxicos
- Yacc(Bison) - Generador de analizadores sintáticos

González Morales Víctor

29/08

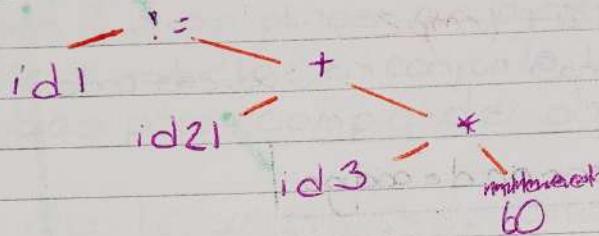
Revisión de todo el proceso

posición := inicial + rate * 60

Análizador Léxico

id1 := id2 + id3 * 60

Análizador Sintáctico



Análisis semántico

Tabla dc
Símbols
posición...
inicial...
rate...

int[60]
|
60

E
r
r
o
r
e
s

← Generador de Código
intermedio →

Gonzalez Molles Victor

Revisión de todo el proceso

29/08

Tabla de símbolos

posicion ...

inicial ...

rate ...

E

r

r

o

r

c

s

Generador de código intermedio

temp1 := intToReal(60)

temp2 := id3 * temp1

temp3 := id2 + temp2

id1 := temp3

address code

Optimización de código

temp1 := id3 * 60.0

id1 := id2 + temp1

Generador de código final

MOVE id3, R2

MULF H 60.0, R2

MOVE id2, R1

ADDF R1, R2

MOVF R1, id1

Morfosintaxis

02-Sep

Es la parte de la gramática que estudia la formación de las palabras y las relaciones entre ellas para formar oraciones.

Morfosintaxis es una palabra compuesta por morfología y sintaxis. Morfología viene del griego morphé, que quiere decir "forma" y logos, que significa "estudio" ó "tratado". Sintaxis, griego, significa "orden" o "disposición".

S. bien son 2 disciplinas que pueden estudiarse por separado, nos ayuda a estudiar en conjunto la forma y función de las palabras para comprender ampliamente el lenguaje.

Partes de la morfosintaxis

La morfología nos cuenta como se forman las palabras de un idioma, las categoriza según el tipo de palabras y su función (en español se ven varias categorías de palabras: sustantivos, adjetivos, verbos, adverbios, preposiciones, pronombres, artículos, conjunciones y determinantes).

- Los **sustantivos** son palabras que nombran a personas, animales o cosas, realidades físicas, y mentales (emociones, cualidades, relaciones, etc.). Pueden ser propias, comunes. Peña, Asunción, mesa, perro, gato, paz, nación, miedo, valentía, reja, María... Tienen marcas de género y número para concordar con otras palabras.
- Los **adjetivos** son palabras que describen los sustantivos: rico, corta, mexicana, medoso, veloz, etc. y tienen marcas de género y número que deben concordar con el sustantivo.

Gonzalez Marales Victor 021 Sep

Los **verbos** son palabras que expresan acciones y tienen marcas de número, persona, tiempo, modo y aspecto: camina, corre, cantó, tuviese, pensaremos, etc.

Los **adverbios** son palabras invariables (que no tienen marcas de género o número) que designan diversas circunstancias: de lugar (aquí, allí), de tiempo (hoy, ayer, mañana), de modo (así), de cantidad (más, menos), de afirmación (si, seguramente), de negación (no, jamás), de duda (tal vez, quizás, puede que), contrarios.

Los **artículos** son palabras sin significado propio que determinan la presencia de un sustantivo, y concuerdan con él en género y número; el, la, los, la, los, un, uno, unos, una.

Los **determinantes**, también llamados adjetivos determinantes o determinativas, concuerdan en número y género, y la presión o determinan: este, nuestro, nuestra, primera.

Los **pronombres** son las palabras que sustituyen ocasionalmente al sustantivo, y se reconocen por el contexto: ello, el, tú, vosotros, vosotras, estos, aquéllos, me, se, mí.

Los **preposiciones** son palabras invariables que unen o relacionan otras palabras con otras: a, ante, con, contra, de, desde, en, por, entre, bajo, para, sin, sobre, ticas, durante, aunque, pero, mas, etc.

Los **conjugaciones** son también palabras invariables que conectan o unen palabras, sintagmas o oraciones: y, o, ni, e, que.

02/1 Sep.

González Morales Víctor

¿Qué es la sintaxis?

Se encarga del ordenamiento de las palabras en una oración, y de cómo juntas pueden formar sintagmas. Son, a su vez, tienen un sentido concreto y una función específica de la oración.

En español hay una orden por defecto (las palabras no se juntan u ordenan arbitrariamente), es SVP (Sujeto, verbo, predicado) : Pedro (sujeto), come (verbo), pescado (predicado). El sujeto, o sintagma nominal, es el conjunto de palabras que indica quién ejecuta la acción. Tiene un núcleo, generalmente un sustantivo o pronombre.

El verbo es la palabra que indica la acción dentro de la oración, y debe tener concordancia con el sujeto. El predicado es el sintagma verbal, y es todo lo que se dice del sujeto. El núcleo siempre es el verbo.

Con la sintaxis se aprende a reconocer en las oraciones el sujeto y el predicado, los verbos y los complementos gramaticales.

Ejemplos de análisis morfosintáctico.

Ejemplo #1: Los niños juegan con sus primas.

De esta acción podemos decir que es bimembre, tiene sujeto (los niños), predicado (juegan con sus primas). El núcleo del sujeto es niño (nombre común, femenino, plural) y el núcleo del predicado es juegan (verbo intensivo, presente de indicativo, 3ra persona del plural).

También podemos advertir que en esta oración hay 2 sintagmas nominales: los niños y con sus primos.

El sustantivo del primer sintagma nominal, niños, está precedido por un determinante, el artículo Los. Y el segundo sintagma nominal está precedido por un adjetivo "sus". Ambas determinantes concuerdan en número y género con las respectivas sustantivas que acompañan (niños y primos) sustantivas femeninas, plurales.

Además, el sintagma verbal (juegan con sus primos), está unido al primer sintagma nominal por una preposición con. Esta convierte a sus primos en un complemento y complemento o constituyente de compañía, que responde a: ¿Quién? Es una oración simple, predicativa, cativa, intransitiva, declarativa, o afirmativa.

Ejemplo #2: Comí una hamburguesa

(Yo) es el sujeto elíptico o tácito, es decir, no aparece expresamente. Comí una hamburguesa, sintagma verbal. Núcleo del sintagma: Comí (verbo transitivo, pasado simple de ind. cativo, 1ra persona del singular).

El sintagma verbal también tiene un sintagma nominal, una hamburguesa. Esto precedido por un artículo determinante, no (concordada en número y género con el sustantivo, femenino, singular, hamburguesa).

A su vez, una hamburguesa constituye un complemento directo porque dice qué fue lo que comió. El verbo comer es transitivo, y necesita ser completado por un constituyente sintáctico, que explique qué comió: 1 hamburguesa. Sobre el complemento directo recoge la acción del verbo. Es una oración simple, predicativa, transitiva, declarativa.

02/sep

Ejemplo #3: Aún es temprano

Advertimos que no hay sujeto, pues este es impersonal, lo que significa que todo la oración es un predicado, es decir, un sintagma verbal. Esto conformado por 2 sintagmas adverbiales. aún + temprano y ambos son adverbios de tiempo. Sintácticamente, cumplen la función de complementos circunstanciales de tiempo.

"Es" es el verbo (verbo ser, presente de indicativo, 3era persona singular) es el núcleo del sintagma verbal. Es una oración simple, impersonal, declarativa, afirmativa.

Ejemplo #4: Mi hijo se parece su padre.

Oración bimembre, con sujeto y predicado. Mi hijo es el sujeto, y es el sintagma nominal del que hijo es el núcleo. Se parece a su padre es el predicado o sintagma verbal, cuyo núcleo es se parece (acompañando de la forma impersonal se, en presente, de indicativo, 3era persona del singular, que concuerda con mi hijo).

Ejemplo #5: El barca jugó mañana contra el Real Madrid.

Oración bimembre: Sujeto, el barca, pred.gado, jugó mañana contra el real madrid. Núcleo del sintagma nominal, barca; Núcleo del sintagma verbal: jugó (verbo intencionativo, futuro simple del indicativo, 3era persona singular, que concuerda con barco).

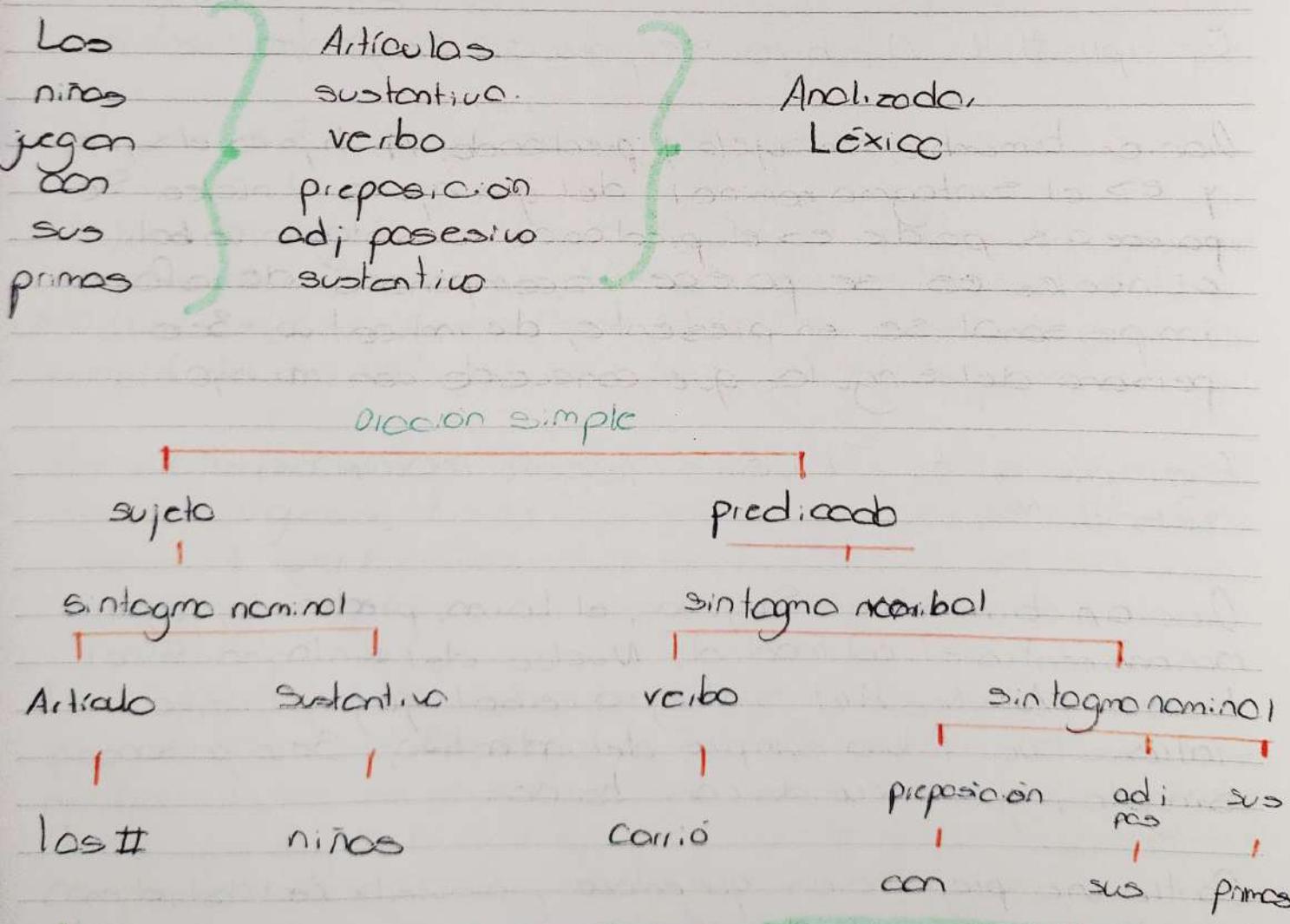
Contra una preposición, que enlaza y convierte Real Madrid en un sintagma personal de término y complemento circunstancial de modo (oposición).

Gonzalez Moroles Victor

Mañana, es un adverbio de tiempo, indica un momento futuro, y por eso el verbo se conjuga en futuro.

El Pcol Mod., d es otro sintagma nominal dentro del sintagma verbal, precedido por el determinante El. El núcleo es Pcol Mod., d, y concuerdan ambas en género y número. Completa la función sintáctica de *se*, el término del sintagma preposicional, es decir, el objetivo dentro de la acción: *me eq* jugó contra otra, que es Pcol Mod., d.

Es una acción simple, predictiva, intencional, declarativa, afirmativa.



3 - Sep

Gonzalez Morales Victor

Ejercicios

Vídeo: análisis morfológico de oración

1 - Realiza el análisis morfológico de las sig. oraciones:

- La niña llevaba un vestido rosa

La: artículo, femenino, singular

niña: sustantivo, común, femenino, singular

llevaba: verbo, pasado, imperfecto, indicativo, 3ra persona singular

un: masculino, determinante, indefinido

vestido: sustantivo, masculino, singular, común

rosa: adjetivo, singular, masculino.

- Mi perro está enfermo desde ayer.

Mi: Adjetivo posesivo, singular

perro: sustantivo, singular, masculino

está: verbo presente, indicativo, 3 persona, singular

enfermo: adjetivo, masculino, singular

desde: preposición

ayer: adverbio de tiempo.

Oración simple

sujeto

predicado

sintagma nominal

sintagma verbal

artículo sustantivo

núcleo verbal

complemento (O. D.)

La

niña

llevaba

determinante
sust. ad;
indefinido

un vestido rosa

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.

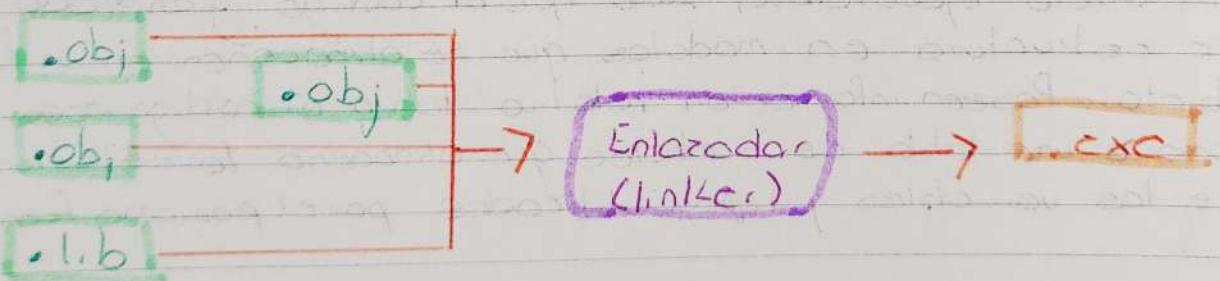
Entrada: Fich. mod



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).

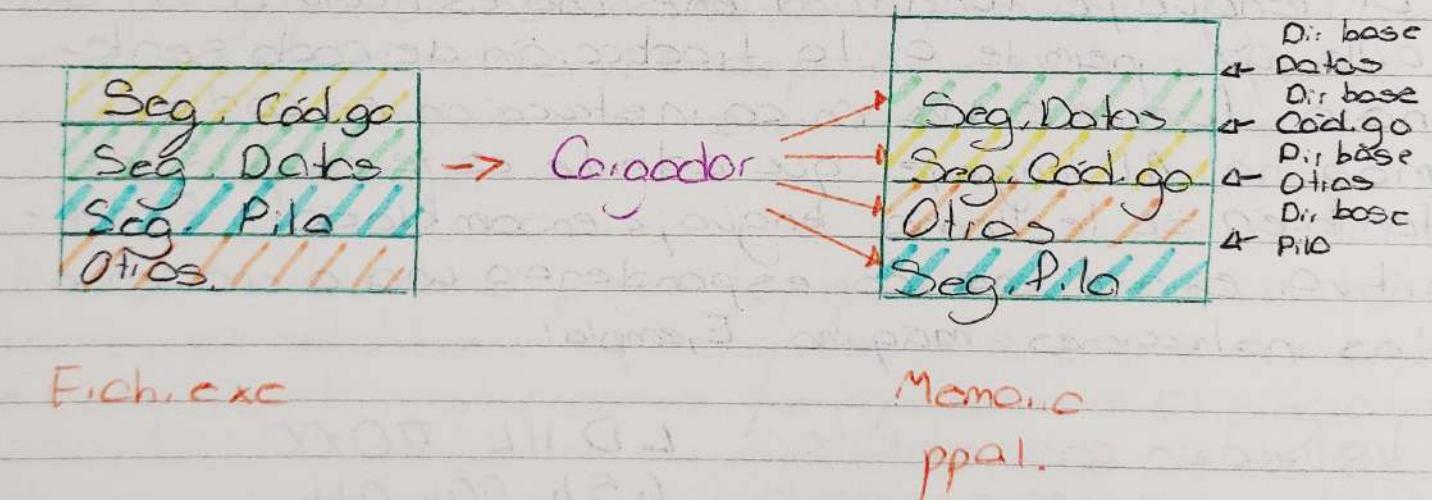
Durante la fase de enlace, el enlazador o linker, resuelve las referencias cruzadas, (así se llama a la utilización de objetos externos), que pueden estar declarados en otros ficheros objeto, o en bibliotecas (ficheros con extensión "lib" o "dll"), engloba en un único bloque los distintos registros que 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 contenido 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.



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.

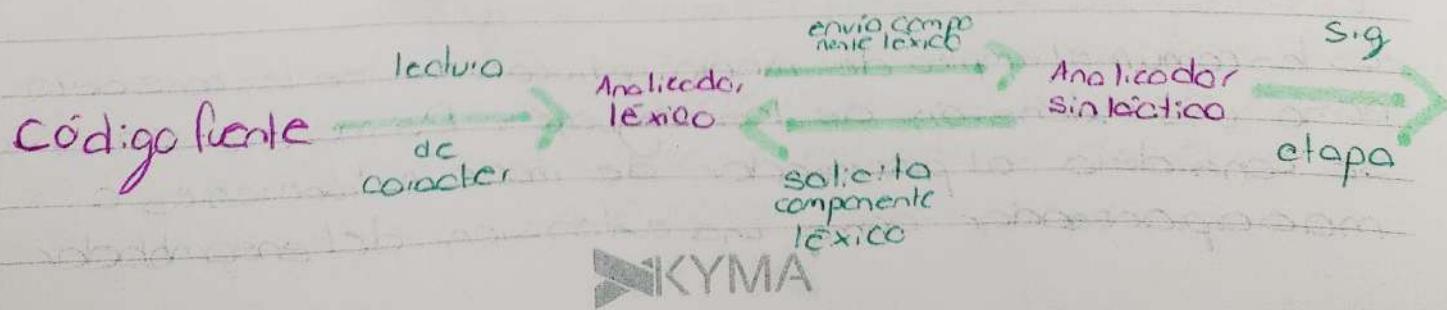
Análisis Léxico

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



13.902.10

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 encajar 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

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

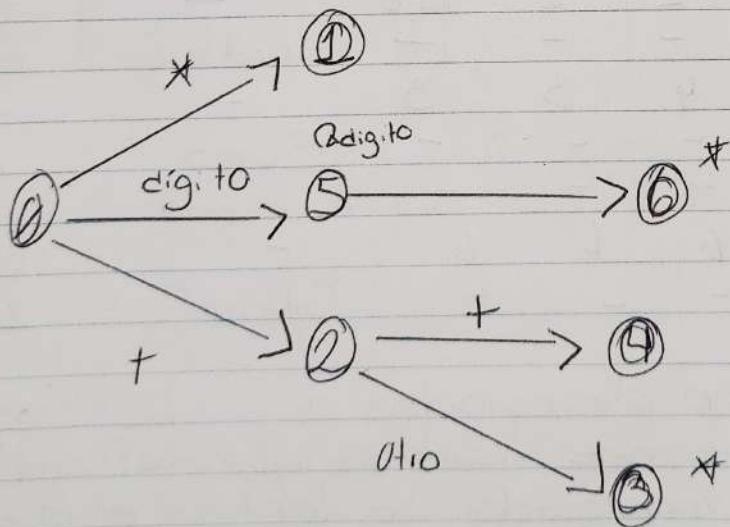
Incremento

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

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

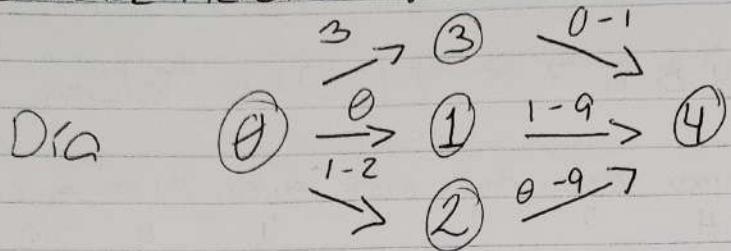
Diseñar una máquina de estados a partir de las expresiones regulares que definen la gramática para un reconocedor de fechas en formato dd/mm/aaaa. Elabore el diagrama de transiciones. Nota: los valores para año están en rango 1000 - 2999.

12-Sep

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

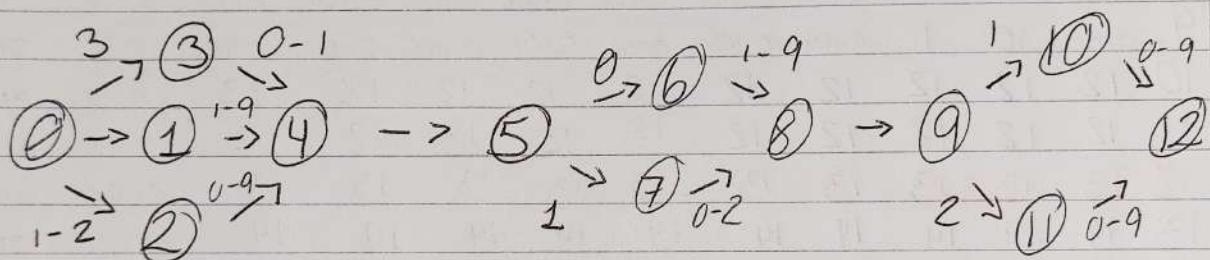
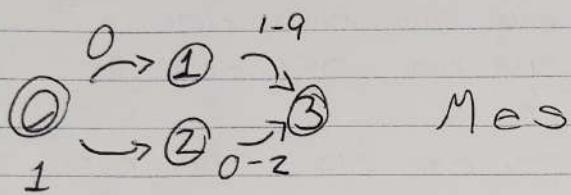
Gonzalez Morales Victor

Fecha 12 - Sep

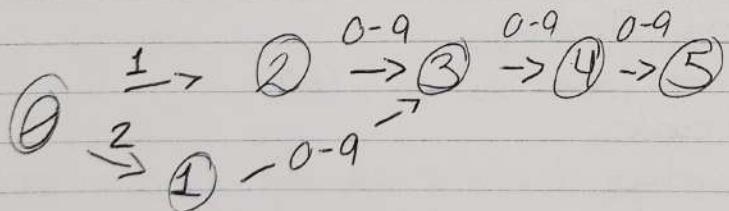


$$\text{Mes} := (\emptyset [1-9] \mid 1[\emptyset-2])$$

17 - Sep



$$\xrightarrow{0-9} 13 \xrightarrow{0-9} 14$$



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

$$(112[\emptyset-9] \{3\})$$

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"

bucsa(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	dclentia	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 tan 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.

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

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:

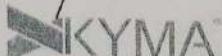
factorial (n):

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.

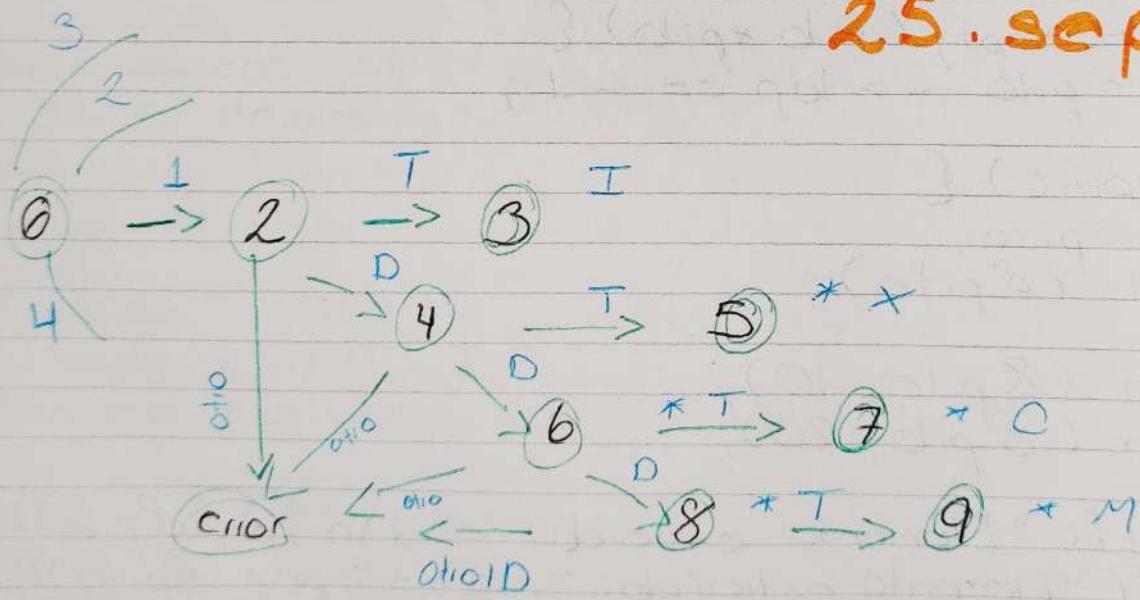
Gonzalez Morales Victor

23-sep

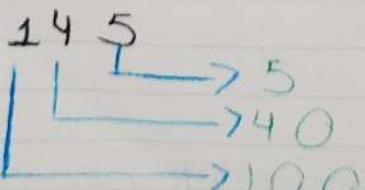
Aplicación: Este concepto se usa comúnmente en intérpretes y máquinas virtuales, como lo Java Virtual Machine y la P-Code machine del lenguaje Pascal. En estos casos, el código fuente es traducido a un conjunto de instrucciones para una máquina de pila abstracta que luego es ejecutado o interpretado.

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 }

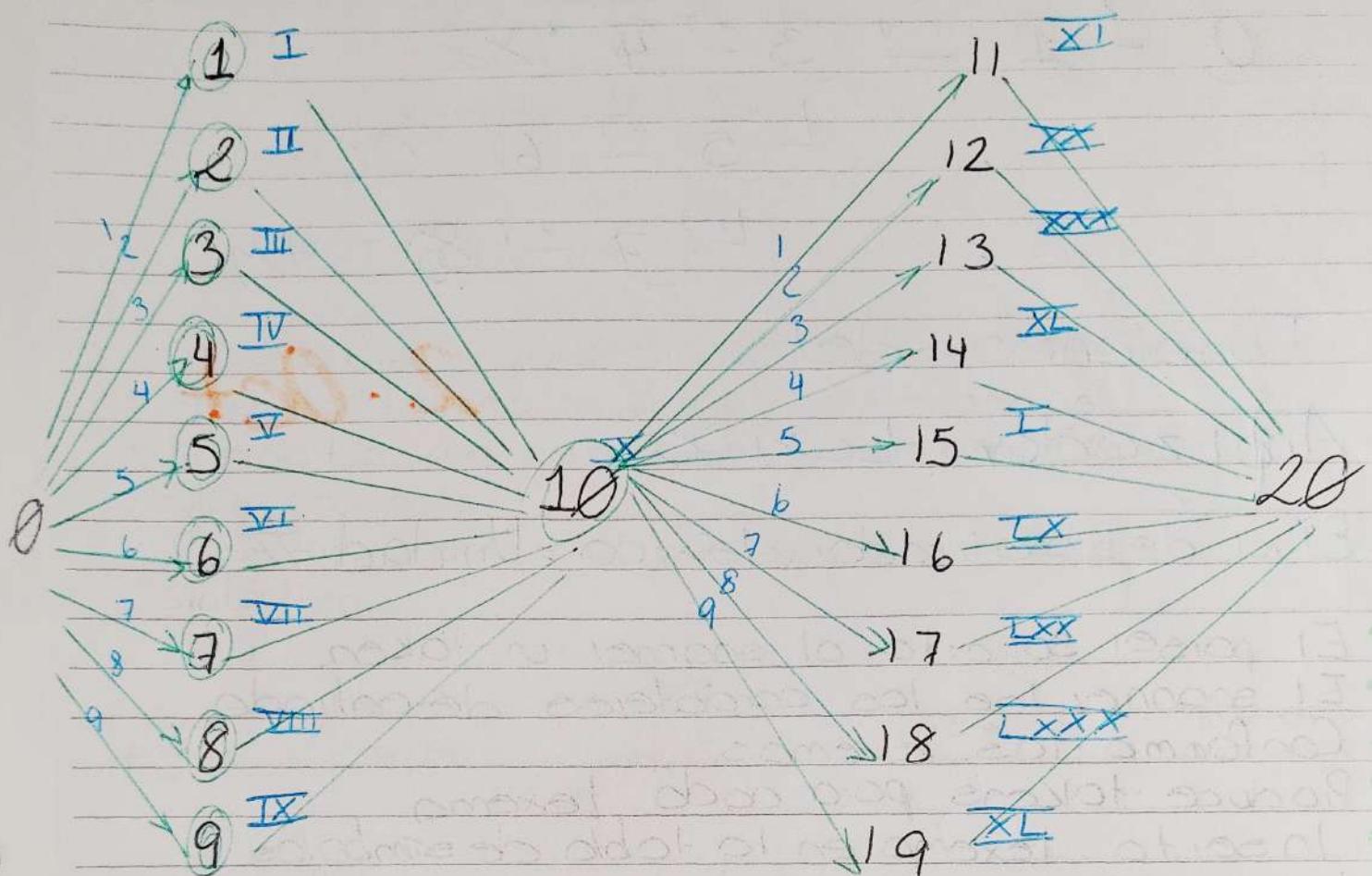


→ 1000
2000
3000

1 4
1 1 1 1 ↑
1 1 1 ↓ v
CD X L V
CXLV

Gonzalez Mardes Victor

26-SEP



$\overset{T}{\curvearrowleft} \quad 2 \text{ I}$

$1 - 3 \equiv 4 * X$

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

$L^{1-4} \quad 7 - 8 * M$

~~2x23T~~

~~1x40~~

$0 - 1 \overset{3}{-} 3 * - 4 * XXX$

$L^{1-4} \quad 5 * - 6 * CCC$

$L^{1-4} \quad 7 * - 8 * MMM$

Gonzalez Morales

victor

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

Morales Victor Eduardo

Sesiones del Analizador Léxico

Fecha

03-10-24

analizador léxico.

• Pasa al scanner un token.
• Los caracteres de entrada
• mas.

• A cada lexema
• esa tabla en símbols.
• tabla para identificar el
• para análisis sintáctico.

Unidad

parser → to semantic analysis

#3

• Usos de espacioado.
• Es de nuevo línea.
• Cero de línea en fuente,
• errar en fuente.
• También expanden las macros.

Analizador Léxico

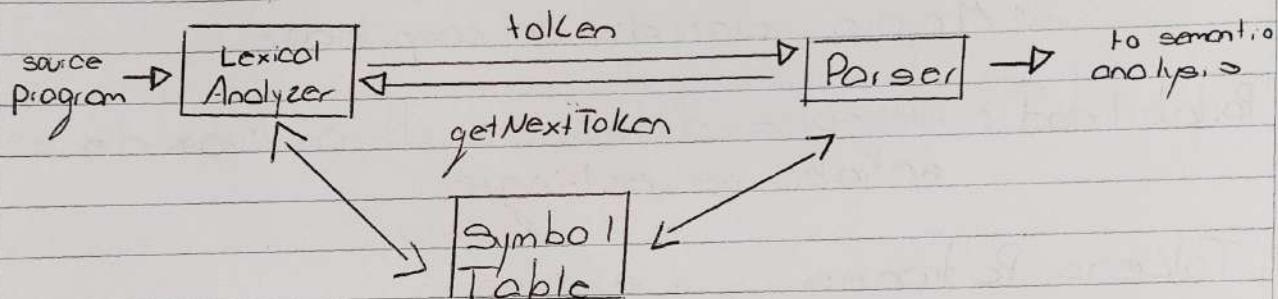
Léxico puede dividirse en 2 procesos:

- Elimina comentarios
- Compacta espacios en blanco consecutivos.

El rol del analizador léxico.

U3

- El parser solicita al scanner un token
- El scanner lee los caracteres de entrada
- Conforma los lexemas.
- Produce tokens para cada lexema
- Inserta lexema con esa tabla en símbols.
- Interactúa con esa tabla para identificar el token correcto.
- Devuelve tokens al parser para análisis sintáctico.



Otras tareas del analizador léxico:

- Eliminar comentarios y caracteres de espacioado.
- Mantener conteo de caracteres de nuevo línea.
- Asociar errores con número de línea en fuente.
- Insertar mensajes de error en fuente.
- Algunos compiladores también expanden las macros.

Procesos del analizador léxico

El analizador léxico puede dividirse en 2 procesos:

- 1) Escanea:
 - Elimina comentarios
 - Compacta espacios en blanco consecutivos

- 2) Análisis Léxico :
- Proceso más complejo
 - Genera tokens del resultado del escaneo

Razones para Separar Análisis Léxico y Sintático

- Simplicidad :
- Facilita diseño de compiladores
 - Parser más simple sin espacio y comentarios

- Eficiencia :
- Técnicas especializadas mejoran análisis léxico.
 - Mejora velocidad del compilador

- Portabilidad :
- Peculiaridades de dispositivos de entrada se restringen

Tokens, Patrones y Lexemas

Token : Por nombre -valor, representa un díl exico, ej: palabra clave.

Patrón : Describe la forma del lexema; Para palabra clave, es la secuencia de caracteres.

Lexema : Secuencia de caracteres que coincide con el patrón, identificado como una instancia de un token.

Token

if	caracteres i, f	if
else	caracteres e, l, s, e	else
comparación	< or > or <= or >= or = = or !=	<=, !=
id	letras seguidas de ^{1 o más} digitos	p., score, D2
número	cualquier constante numérica	3.14159, 0, 6.023
literal	cualquier cosa entre " " sin los " " corredorped	

```
printf("Total = %. d\n", score);
```

Tanto printf como score son lexemas que coinciden con el patrón de token id y Total = %. d \n" es un ejemplo que coincide con literal.

Atributos con Tokens

- Varios lexemas pueden coincidir con un patrón
- El analizador léxico, brinda información adicional. Ejemplo: Token "número" puede ser 0 a 1
- Se devuelve nombre del token y su valor atributo
- El nombre afecta el miosis; el atributo lo traducen.
- Tokens tienen un atributo, que puede contener varios datos.
- Caso "id", atributo apuntará a de tabla correspondiente.
- Ciertos tokens operadores, puntuación, palabras clave no requieren atributos.
- Un token "número" tendrá un atributo comprobatorio.
- Realmente, el compilador almacena la constante como cadena.
- El valor atributo para "número" es un puntero a esa cadena.

$$E = M * C ** 2$$

se escribirá como una secuencia de pares.

<id, puntero a entrada en la tabla de símbolos para E>

<assign_op>

<id, puntero a entrada en la tabla de símbolos M>

<mult_op>

<id, puntero a otro entrada en la tabla para símbolos C>

<exp_op>

<número, valor entero 2>

Errores léxicos

- Difícil para analizador léxico detectar errores de código.
- El ; " ; " puede ser un final de ";" o un identificador no declarado.
- El analizador léxico devuelve token "id" al parsear.
- El parser o fases posteriores manejan el error.
- Si no encuentra token válido, procede a "modo pónico", que es eliminar caracteres hasta encontrar token válido.
- Puede confundir al parsear, pero es adecuado en entornos interactivos.

Otros estrategias de recuperación de errores

- Eliminar un carácter del input constante.
- Insertar un carácter faltante.
- Reemplazar un carácter por otro.
- Transponer 2 caracteres adyacentes.
- Transformar el prefijo para obtener lexema válido.

Gonzalez Morales Victor Eduardo

Fecha
03/10/20

- Mayoría de errores léxicos son de un solo carácter.
- Correciones más complejas implicativas debido a alto costo.

Divide los sig. programas

03/10/24

C++:

```
float limiteSquare (x) float x; {  
    // Retorna x cuadrado, pero no más de 100 *!  
    return (x <= -10.0 || x > 10.0)? 100: x * x;  
}
```

Java:

```
public static double CalcularAreaCirculo(double  
radio) { double area = Math.PI *  
Math.pow(radio, 2);  
return area;
```

}

Python:

x = 5

if x > 9:

```
print ("! Hola Mundo! ")
```

¿Qué lexemas deben tener palabras léxicas
adequadas? ¿Cuáles debieran ser esas palabras?

TOKEN	Descripción informal	Lexemas múltiples	Lexemas en el programa	Palabras léxicas adecuadas
; f	caracteres ;, f	;	f	NA
else	caracteres e, l, s, o	else	NA	NA
comparación	<=, >=, ==, !=, <, >	<, !=	NA	NA

Gonzalez Moroles Victor Eduardo

Fecha
03/10ct

```
int num1 = 5;
int num2 = 10;
```

```
public static int sumar (int a, int b) {
    int suma = a + b;
    System.out.println ("La suma es: " + suma);
}
```

TOKEN	Descrip informal	Lexemas muestra	Lexemas en el programa	valores lógicos asociados
public	caracteres p,u,b,l,i,c	public	public	NA
static	caracteres s,t,a,t,i,c	static	static	NA
assign- op	=, + =, - =, *=, /=, % =	=, / =	=	=
add- op	*	+	+	NA
id	Letras seguidas de letra y digitos	num1, num2, num3,	num 1	puntero a entrada en la tabla de simbolos para num1
número	Cualquier constante numérica	3.14159, 8	5	5
literal	Cualquier cosa entre los ""	una cadena de palabras	La suma es	puntero a entrada en la tabla de simbolos para la suma es.
curly left	{	{	{	NA
puntuación	, ; , :	:	:	;

Gonzalez Morales Victor Eduardo

Fecha
03/Oct

En el caso del token assign-op, en el ejemplo, se da como descripción informal el conjunto de operadores de asignación definidos en los diversos lenguajes de programación existentes, es por eso que después es necesario decir cuál fue el que se leyó del código, en la columna de valores léxicos asociados, lo mismo sucede para el token "puntuación"

En el caso del token add-op como nadamas es un símbolo puede ser add-op, pues no se necesita agregarle atributo, por eso, dice NA en la columna de los valores léxicos asociados, lo mismo sucede con el token cuty left.

Python $x = 5$

; f $x > 9$

print ("Hola mundo")

07/10ct

TOKEN	DESC Informal	Lexemas muestra	Lexemas programa	Valores
id		X	X	
assign-op	=, +, ==, +=, /=, *=	=, /s	=	=
num	cualquier constante		5	5
; f	cualquier carácter	; f	; f	NA
puntuación	<, >, == !=, <, >	! =	>	>
num	cualquier constante	5, 9	9	9
Puntuación	;	;	;	;

TOKEN	DESCRIPCION	LEXEMA	LEXEMAS PROGRAMA	VALORES
Print	carácteres	print	print	N/A
par 129	print, + (((N/A
LITERAL	cualquier cosa ente " " o "	"coredump"	Hola mundo	Puntero a entrada en tabla de símbolos para hola mundo.
PARA)))	N/A

09 Oct +

Manejo de Buffers de Entrada

Buffer de entrada.

Se necesita mirar más allá del próximo lexema.

Ej: No se sabe si un identificador termina hasta ver un carácter no alfanumérico.

Operadores en C pueden ser de uno o dos caracteres.

Se usa un esquema de dos buffers para manejar el look ahead.

Los buffers permiten anticipar varios caracteres con seguridad.

La mejora "sentinelas" reduce el tiempo en checar el fin de los buffers.

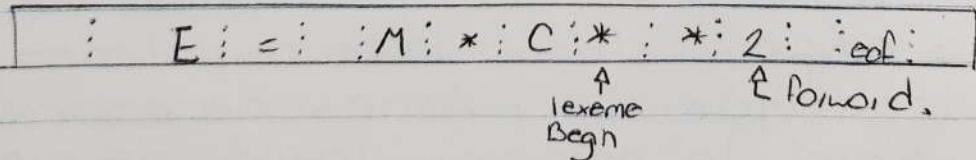
El buffer acelera la lectura del programa fuente.

Técnica de Pares de Buffers

- Justificación: Procesar eficientemente grandes volúmenes de caracteres.
- Dos buffers altanos se reaigan para evitar sobre cargo.
- Cada buffer tiene tamaño N , 4096 bytes.
- Un comando de lectura cargo N caracteres en lugar de uno por uno.
- Si quedan menos de N caracteres, eof marca el final del archivo.
- El puntero lexema Begin marca el inicio del lexema.
- El puntero forward avanza hasta encontrar una coincidencia de patrón.

Estrategia de Punteros en pares de buffers

- Al encontrar un lexema, forward se mueve a lo derecho del último.
- Lexema Begin se ajusta al carácter después del lexema.
- El puntero forward puede regresar si se pasa del lexema.



Estrategias de Punteros

- Se aprueba si forward alcanza el final de un buffer.
- Si es así, se recarga el otro buffer y el forward se mueve al inicio.
- Mientras no se sobrepase el tamaño N , el lexema no se sobrescribe.

Gonzalez Morales Victor, Eduardo

Fecha: 09/10ct

- Garantiza la detección del lexema antes de perderlo en el búfer.

Sentinelas en Búferes

- Se debe verificar si forward sale de búfer.
- Si sale, se recarga el otro búfer.
- Se realizan 2 pruebas por cada carácter leído.
- El uso de sentinelas simplifica esta verificación.
- Es un carácter especial, como eof.
- eof marca el fin del búfer o del archivo completo.
- El algoritmo solo prueba el carácter actual, salvo al final de un búfer.

```
| : |E| = | : |M| *eof| C| *| *| 2 |eof| ;eof|  
↑ ↑  
lexeme forward  
Begin.
```

Especificación de los Componentes Léxicos

11/10ct

Especificación de tokens:

- expresión - regular, especifican patrones de lexemas.
- Efectivos aunque no cubren todos los patrones posibles.
- Definen los tipos de patrones necesarios.
- Se estudiará la notación formal de expresiones regulares.
- Se analizará su uso en generadores de analizadores léxicos.
- Se explicará la construcción de expresiones regulares en automatos.
- Automatos reconocen tokens específicos.

Cadenas y Alfabetos

- alfabeto conjunto finito de símbolos
- Ejemplos comunes: letras, dígitos, puntuación.
- $\{0, 1\}$ alfabeto binario, ASCII, Unicode.
- cadena secuencia finita de símbolos de un alfabeto.
- Longitud de una cadena se denota $|s|$.
- Cadena vacía, denotada ϵ , tiene longitud cero.

Lenguajes y Concatenación

- lenguajes conjunto contable de cadenas sobre un alfabeto.
- Lenguajes abstractos incluyen \emptyset (conjunto vacío) y $\{\epsilon\}$.
- Ejemplos: programas C bien formados, frases gramaticales en español.
- concatenación de 2 cadenas x y y es xy .
- Ej: si $x = \text{"pasu"}$ y $y = \text{"mecho"}$; entonces $xy = \text{"pasumecho"}$.
- La cadena vacía es la identidad en la concatenación: $\epsilon s = s \epsilon = s$.
- "exponentiación" de cadenas: $s^2 = ss, s^3 = sss$, etc.

Términos para partes de cadenas

- Prefijo: Elimina símbolos del final de la cadena (ej: ban, banana, ϵ).
- Sufijo: Elimina símbolos del inicio (ej: nana, banana, ϵ).
- Subcadena: Elimina cualquier prefijo y sufijo (ej: nan, banana, ϵ).

- Prefijo / Sufijo / Subcadena propia: Igual que el prefijo / sufijo / subcadena pero no es ni la cadena completa.
- Subsecuencia: Formada al eliminar cero o más símbolos no consecutivos (ej: ban de banana)

Operaciones con Lenguajes:

- Unión: Combina los elementos de 2 lenguajes.
- Concatenación: Une una cadena de un lenguaje con una del otro.
- Clasura Kleene (L^*): Concatena el lenguaje L cero o más veces.
- L^0 : Es $\{\epsilon\}$, la concatenación de L cero veces.
- Clasura Positiva (L^+): Concatena L al menos una vez.
- L^+ no incluye $\{\epsilon\}$ a menos que L lo contenga.

Operación

Unión de L y M

Concatenación de L y M

Cerradura de Kleene de L

Cerradura positiva de L

Definición y notación

$L \cup M = \{s | s \text{ está en } L \text{ o } s \text{ está en } M\}$

$LM = \{st | s \text{ está en } L \text{ y } t \text{ en } M\}$

$L^* = \bigcup_{i=0}^{\infty} L^i$

$L^+ = \bigcup_{i=1}^{\infty} L^i$

Expresiones Regulares

- Describen lenguajes usando operadores: Unión, concatenación y clausura.

- Ej: Identificadores en un lenguaje.

- 'letter' representa letras o guioncitos bajos,
- 'digit' representa dígitos

- Un identificador válido se escribe como:
 - 'letter - (letter | digit)*'
- ' '|=union, '*' = cero o más ocurrencias
- Se constituyen recursivamente usando subexpresiones
- Cada expresión denota un lenguaje.

Reglas básicas de expresiones regulares

Das reglas fundamentales.

- 1) 'ε' es ER, $L(\epsilon) = \{\epsilon\}$, el lenguaje con solo la cadena vacía.
- 2) Si 'a' es un símbolo en Σ, entonces 'a' es uno ER y $L(a) = \{a\}$, un lenguaje con una cadena de longitud 1.

Construcción por inducción de expresiones regulares

- Se construyen expresiones más grandes a partir de otras más pequeñas.

- 1) '(r)' | '(s)' denota $L(r) \cup L(s)$
- 2) '(r)(s)' denota $L(r)L(s)$
- 3) '(r)*' denota $L(r)^*$.
- 4) '(r)' denota $L(r)$ (agregar paréntesis no cambia el lenguaje).

Simplificación de expresiones regulares.

- Expresiones regulares a menudo contienen paréntesis innecesarios.
- Podemos eliminarlos si seguimos las convenciones:

- 1) El operador '*' tiene la mayor precedencia y es asociativo a la izquierda.
- 2) La concatenación tiene la 2da precedencia más alta y es asociativo a la izquierda.
- 3) '**' tiene la menor precedencia y es asociativo a la izquierda.

• Ej: '(a)1((b)* (c))' se puede simplificar a 'a1b*c'.

Conjuntos regulares y leyes algebraicas.

- Un lenguaje definido por una expresión regular se llama en conjunto regular.
- Si 2 expresiones regulares denotan el mismo conjunto regular, son equivalentes.
- Ej: '(a1b)= (b1a)'.
- Las leyes algebraicas afirman que expresiones diferentes pueden ser equivalentes.
- Estas leyes se aplican a expresiones regulares 'r', 's', y 't' arbitrarias.

LEY

DESCRIPCIÓN

$$r1s = s1r$$

1 es commutativo

$$r1(s1t) = (r1s)1t$$

1 es asociativo

$$r(s+t) = (r s) +$$

La concatenación es asociativa

$$r(s1t) = r s1 t; (s1t)r = s1t r$$

11 se distribuye a través de 1

$$\epsilon_r = r \epsilon = r$$

ϵ es la identidad para la concatenación

$$r * = (r1\epsilon)^*$$

ϵ es garantizada en un cierre.
* es idempotente.

$$r** = r^*$$

Definiciones regulares

- Para simplificar asignar nombres a ER.
- Definición regular secuencia de la forma:
 - $d_1 \rightarrow r_1$
 - $d_2 \rightarrow r_2$
- Cada ' d_i ' es un símbolo nuevo, cada ' r_i ' es una ER.
- Los ' r_i ' utilizan el alfabeto Σ y las definiciones previas.
- Se evitan definiciones recursivas.
- Se reemplazan los ' d_i ' gradualmente para obtener expresiones regulares completas.

Extensiones de Expresiones Regulares.

1) Una o más instancias (+):

- Representación clausura positiva.
- $(r)^+$ denota $L(r)^+$.

2) Cero o uno instancia (?) :

- $r^?$ es equivalente a $r^* \cup \epsilon$.

3) Clases de caracteres:

- $a \mid b \mid c$ puede escribirse como $[a, bc]$.
- Rango lógico, como $[a - z]$, representa $a \mid b \mid \dots \mid z$.

Especificación de las Componentes Léxicos
ejemplos

Operaciones con Lenguajes - Ejemplo

Sea: L el conjunto de letras $\{A; B; \dots; z; a; b; \dots; z\}$

D el conjunto de dígitos $\{0; 1 \dots; 9\}$.

Podemos pensar en L y D de 2 maneras, esencialmente equivalentes. Una forma es que L y D son los alfabetos de letas y mayus, minus y de dígitos. La 2da forma es que L y D son lenguajes, cuyas cadenas tienen todas longitud uno. Aquí hay algunas otras lenguajes que se pueden construir a partir de los idénticos L y D , usando los operadores vistos.

$$1) LUD$$

$$2) LD$$

$$3) L^4$$

$$4) L^*$$

$$5) L(LUD)^*$$

$$6) D^+$$

1) LUD es el conjunto de datos de letras y dígitos, es hablando, lenguaje con 62 cadenas de longitud 1, cada una de las cuales es una letra o un dígito.

2) LD es el conjunto de 520 cadenas de longitud dos, cada una de las cuales consta de una letra seguida de un dígito.

- 3) L^4 es el conjunto de todas las cadenas de 4 letras.
- 4) L^* es el conjunto de todas las cadenas de letras, incluida la cadena vacía.
- 5) $L(LUD)^*$ es el conjunto de todas las cadenas de letras y dígitos que comienzan con una letra.
- 6) D^* es el conjunto de todas las cadenas de uno a más dígitos.

Ej: Sea $\Sigma = \{a; b\}$

1. $a|b$
2. $(a|b)(a|b)$
3. a^*
4. $(a|b)^*$
5. $a|a^*b$

1. $a|b$ denota el lenguaje $\{a, b\}$.
2. $(a|b)(a|b)$ denota $\{aa, ab, ba, bb\}$, el lenguaje de todas las cadenas de longitud dos sobre el alfabeto Σ . Otra expresión regular para el mismo lenguaje es $aabb|baabbb$.
3. a^* denota el lenguaje que consta de todas las cadenas de cero o más a, es decir, $\{\epsilon, a, aa, aaa, \dots\}$.
4. $(a|b)^*$ denota el conjunto de todas las cadenas de cero o más instancias de a o b, es decir, todas las cadenas de a y b: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$.
5. $a|a^*b$ denota el lenguaje $\{a; b, ab, aab, \dots\}$, es decir, la cadena a y todos los cadenas que constan de cero o más a y terminan en b.

Gonzalez Morales Victor Eduardo

Fecha

11 Oct

Definiciones Regulares - Ejemplo.

Ej # 1.- Los identificadores de Java son cadenas de letras, dígitos y guiones bajos. Estructura una definición regular para el lenguaje de los identificadores de Java.

letra $\rightarrow A \mid B \mid \dots \mid z \mid a \mid b \mid \dots \mid z \mid -$
dígito $\rightarrow 0 \mid 1 \mid \dots \mid 9$
id $\rightarrow \text{letra} (\text{letra} \mid \text{dígito})^*$

Ej # 2 - Los números sin signo (entero o punto flotante) son cadenas como 5280, 0.01234, 6, 336E4, o 1.89E-4. Genera una definición regular.

dígito $\rightarrow 0 \mid 1 \mid \dots \mid 9$
dígitos $\rightarrow \text{dígito dígito}^*$
fracción Opcional $\rightarrow . \text{dígitos} \mid \epsilon$
exponente Opcional $\rightarrow (E \mid + \mid -) \text{dígitos} \mid \epsilon$
número $\rightarrow \text{dígitos fracción Opcional exponente Opcional}$

Extensiones de Expresiones Regulares - Ejemplo

Ej : Usando extensiones de EB, reescribe la definición regular para los identificadores de Java:

letra $\rightarrow [A - Z \mid a - z \mid _]$
dígito $\rightarrow [0 - 9]$
id $\rightarrow \text{letra} (\text{letra} \mid \text{dígito})^*$

González Morales Víctor Eduardo

Fecha
11 Oct

Ej: Usando extensiones de EER, reescribo la definición regular para los números enteros sin signo:

dígito $\rightarrow [0-9]$

dígitos \rightarrow dígito⁺
número \rightarrow dígitos (dígitos)? (E [+ -])? dígitos)?

Reconocimiento de

15 Oct

ToLens

- Expresamos patrones con expresiones regulares.
- Objetivo: construir código que identifica lexemas en la cadena de entrada.
- ToLens a reconocer: 'if', 'then', 'else', 'relop', 'id', 'numero'.
- Comparaciones: como en Pascal o SQL: '=' es igual, '<' es diferente.
- Los patrones de 'id' y 'numero' son EER's.
- Palabras clave serán reservadas, no identificables.

stmt \rightarrow if expr then stmt

| if expr then stmt else stmt

| ε

expr \rightarrow term relop term

| term

term \rightarrow id

| number

González Molinos Víctor, Eduardo

Fecha
15/Oct

digit → [0-9]
digits → digit +
number → digits(.digits)? (E [+ -]?, digits)?
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

Eliminar espacios en Blanco

- Responsabilidad del analizador léxico, eliminar blancos: ws...
- Blancos: Espacios, tabulaciones y nuevas líneas.
- El token ws no se devuelve al analizador.
- Análisis léxico se reinicia tras espacio blanco.
- Sólo el siguiente token es devuelto al analizador.

ws → (blank | tab | newline) *

Objetivo del Analizador Léxico.

- Asignar tokens a cada lexema o familia de lexemas
- Devolver el nombre del token y valor de atributos a parser.
- Para operadores relacionales, se usan constantes simbólicas: 'LT', 'LE', etc.
- Constantes necesarias para indicar instrucciones encontrada de relop.
- La tabla resume esta información para cada token reconocido.

LEXEMES	TOKEN NAME	ATTRIBUTE	VALUE
Any ws	-	-	-
if	;f	-	-
then	then	-	-
else	else	-	-
Any id	:d	Pointer to table entry	
Any number	number	Pointer to table entry	
<	relOp	LT	
<=	relOp	LE	
=	relOp	EQ	
>	relOp	NE	
>=	relOp	GT	
>=	relOp	GE	

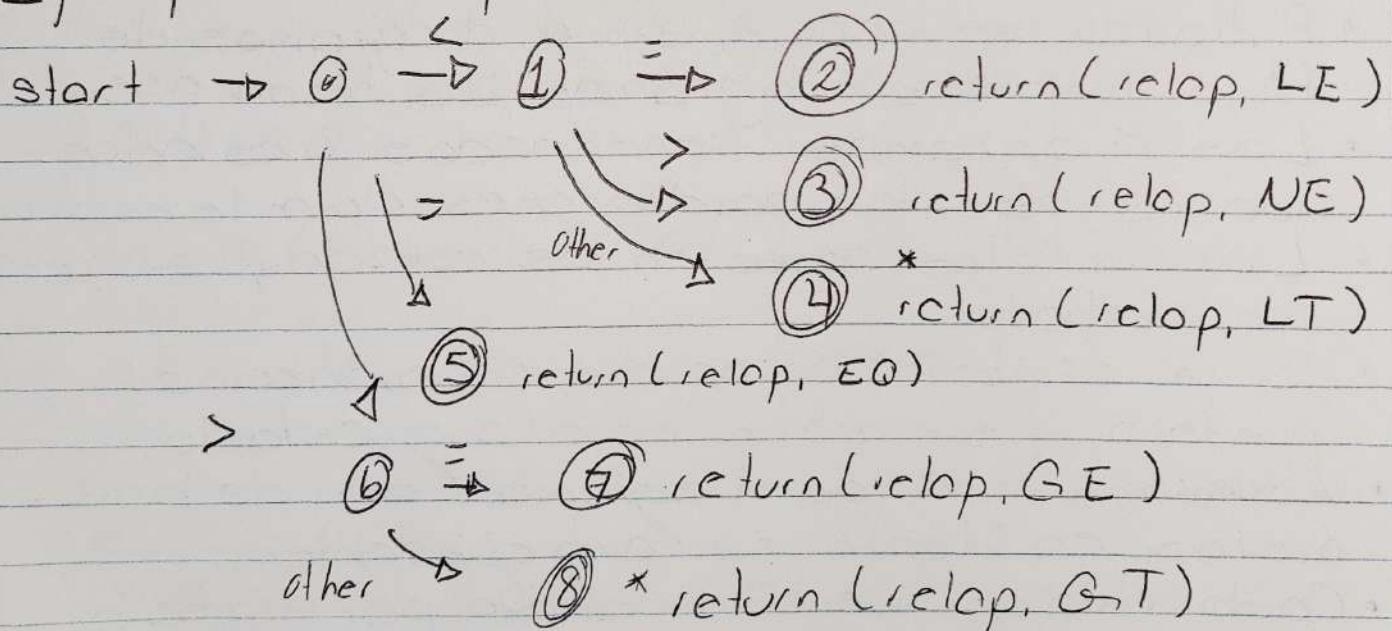
Diagramas de Transición - Concepto.

- Páginas se convierten en diagramas de flujo llamadas diagramas de transición
- Los diagramas tienen nodos o estados, que representan condiciones durante escaneo.
- Los aristas conectan los estados, etiquetados con símbolos
- Si un símbolo coincide, se avanza el puntero y se entra en el sig estado.
- Asumimos que los diagramas son deterministas, es decir, sin ambigüedades.
- Cada estado tiene un arista por símbolo o conjunto de símbolos
- Al procesar un símbolo, el puntero forward avanza y el estado cambia.
- Se simplifica la búsqueda de lexemas durante el análisis léxico:
 - Asegurando una única transición posible por símbolo (determinismo).
 - La condición de determinismo se lleva a cabo en etapas avanzadas del diseño léxico.

Convenciones de los Diagramas de Transición.

- Estados de aceptación indican que se encontró un lexema
- Se representan con un doble círculo y pueden reformar un token
- Si se debe retroceder el puntero forward, se coloca un *.
- Estados iniciales comienzan el proceso, indicados con un orilla "Inicio"
- El diagrama siempre empieza en el estado inicial antes de tener símbolos.
- En este ejemplo, nunca se retrocede más de una posición.

Ejemplo Relop



Reconocimiento de Palabras Clave e Identificadores

- Problema: Los palabras clave como : fo then parecen identificadores.

- Las palabras clave están reservadas, no son identificadores
- El mismo diagrama reconoce palabras clave e identificadores


```

start → ⑨ → ⑩ → ⑪ *return getToken(),
          letter dig. → other
      
```

Manejo de Palabras Reservadas - Método 1

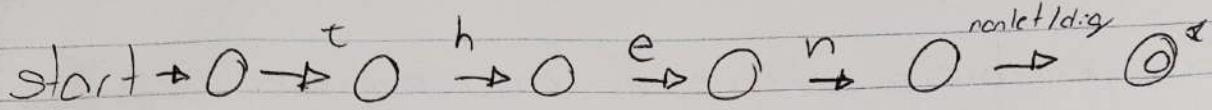
- Instalar palabras reservadas en la tabla de símbolos.
- Un campo en la tabla indica que no son identificadores.
- Al encontrar un identificador, instalo la colas en la tabla.
- Si el identificador no está en la tabla, su token es id.
- getToken revisa la tabla y devuelve el nombre del Token.
- Tokens pueden ser id o palabras clave prioritario
- Método usado en el ejemplo.

Método 2

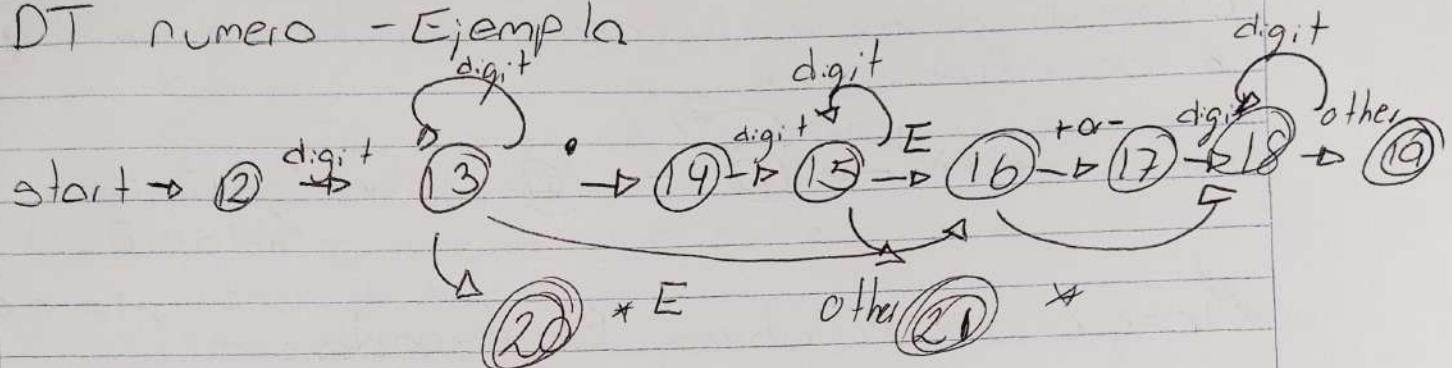
- Crear diagramas de transición separados para cada palabra clave.
- Ejemplo: diagrama de la palabra clave then.
- Verifica que el identificador haya finalizado.
- Evita confundir lexemas como thenvalue con then.
- Priorizar palabras reservadas sobre id.
- No se usó este enfoque en nuestro ejemplo.

Gonzalez Maroles Victor Eduardo

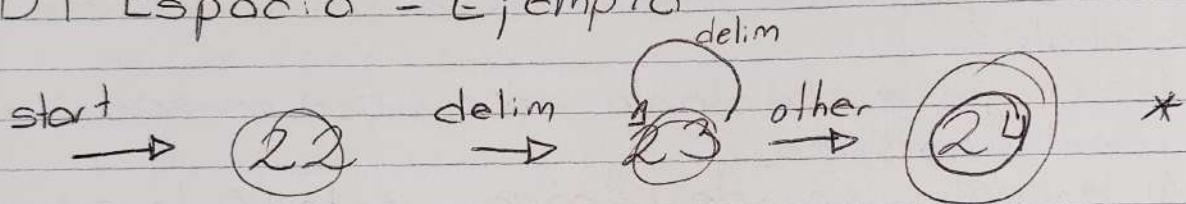
Fecha 17 Oct +



DT numero - Ejemplo



DT Espacio - Ejemplo



Arquitectura de un
Analizador Léxico

17 Oct

- DT's producen analizador léxico
- Cada estado tiene un fragmento de código.
- La variable static almacena el estado actual.
- Un switch se usa para determinar las acciones por estado.
- El código cambia el siguiente carácter de entrada.
- Cambio de estado según el carácter leído.
- Implementa bifurcación o instrucciones múltiples.

getRelop()

{ Token getRelop()

```
TOKEN retToken = new(IRELOP);
while (1) {
    switch(state) {
        case 0: c = nextChar();
            if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else fail();
            break;
        case 1: ...
```

```
        case 8: retract();
            retToken.attribute = GT;
            return (retToken);
```

3 Simulación de Diagramas de Transición

- Organizar diagramas por token
- Usar funciones 'fail()' para reiniciar puntero.
- Palabras clave como reservados.
- Ejecutar diagramas "en paralelo".
- Resolver coincidencias de patrones.
- Preferir coincidencias de prefijo largo.
- Combinar diagramas en uno solo.

Gonzalez Morales Victor Eduardo

17 Oct

TOKEN getRelOp {

TOKEN retToken = new TOKEN();

while (1) {

switch (state) {

case 0: c = nextChar();

if (c == '<') state = 1;

else if (c == '=') state = 5;

else if (c == '>') state = 6;

else fail();

break;

case 1: c = nextChar();

if (c == '=') state = 2;

else if (c == '>') state = 3;

else state = ;

break;

case 2: retToken.setToken("RELOP");

retToken.setCode("LE")

return (retToken); break;

Simulación de Diagramas
de Transición - Aplicación.

21 Oct

Para lenguaje generado por la expresión regular ($a^1 b^2 + a^3 b^2$, crear el DTC (grafo)
generar la tabla de transiciones y
un programa para implementar el analiza-
dar léxico correspondiente a la tabb
que imprima "cadena aceptada" o
"cadena no aceptada" según corresponda.

Gonzalez Morales Victor Eduardo

Fecha: 21/10/0+

Cadena ababba bobba ababb - Resultado:
CADENA NO ACEPTADA

... Program finished with exit code 0
Press ENTER to exit console.

Cadena ababba babb - Resultado: CADENA
ACEPTADA

... Program finished with exit code 0
Press ENTER to exit console.

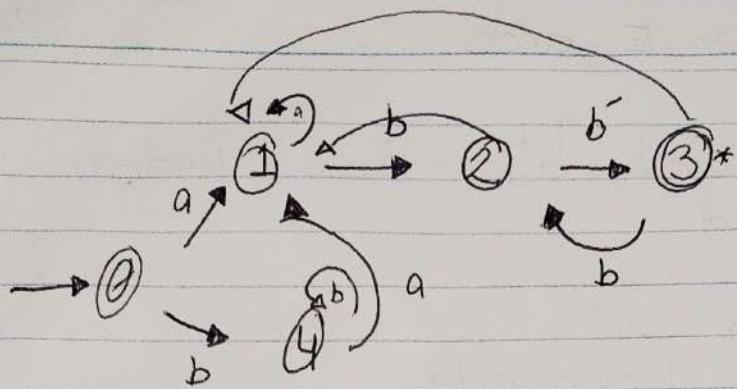
case 3 retToken.setToken('Relop');
retToken.setCode('NE');
break;

case 4 retract();
retToken.setAttribute = LT;
return (retToken);
break;

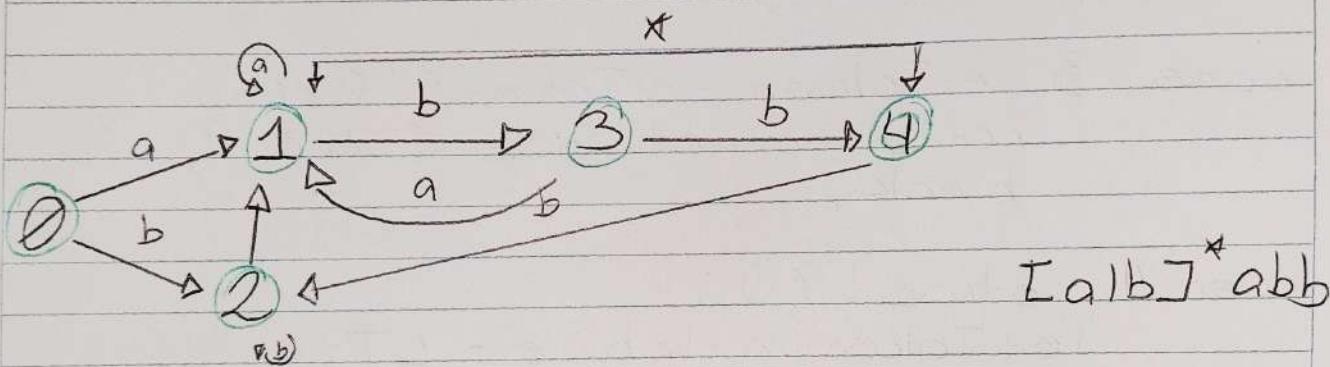
case 5 : retToken.setToken("REMP");
retToken.setCode("FQ");
return (retToken);
break;

case 6 : C = nextChar();
, if (C == '=') state = 7;
else state = 8;
break;

case 7 : retToken.setToken('Relop');
retToken.setCode('GE');
break;



- $\alpha^1 b^2 b^3$ Acepto abbabbabb
- $a^1 b^2 a^1$ NA
- $a^1 b^2 a^1 b^2 b^3$ A
1 2 3 3 3



STATE	a	b	Tolken	Retroceso
0	1	2	-	-
1	1	3	-	-
2	1	2	-	-
3	1	4	-	-
4	1	2	Aceptado	-

- 1) inicio
 - 2) Pedir una cadena para analizar y guardar la cadena en la variable "cadena"
 - 3) Repetir lo siguiente desde el 1er carácter hasta el último carácter de "cadena":
- Leer siguiente carácter y guardarlo en la variable 'dato'

1: Inicio:

2-Inicializa variable "estado" en cero

3- Pedir una cadena para analizar y guardar la cadena en la variable "cadena"

4- Repetir lo siguiente desde el primer caracter hasta el ultimo caracter "cadena"

4.1 En caso de que "estado" sea:

cero: leer siguiente caracter de "cadena" y guardarlo en "dato"

si "dato" es uno 'a' entonces guardar en "estado" un uno.

si no si "dato" es uno 'b' entonces guardar en "estado" un dos.

si no invocar a rutina de error.

fin caso cero.

una: leer siguiente caracter de "cadena" y guardar en "dato".

si "dato" es una 'b' entonces guardar en "estado" un tres.

sino si "dato" no es una 'a' entonces invocar a rutina de error.

fin caso una.

dos: leer siguiente caracter de "cadena" y guardar en "dato".

si "dato" es una 'a' entonces guardar en "estado" en ver uno.

sino si "dato" es una 'b' entonces invocar una rutina de error

fin caso dos.

González Moroles Víctor Eduardo

Fecha 24/10/ct

Tres: leer siguiente carácter de "cadena" y guarda en "dato".
Si "dato" es una 'a' guardar en "estada" uno
Sino si "dato" es una 'b' entonces
guardar en "estada" cuatro
Sino invocar a rutina de error
Fin caso tres.

cuatro: leer siguiente carácter de "cadena" y guardarlo en "dato".
Si "dato" es una 'a' entonces guardar en "estada" uno
Si "dato" es una 'b' entonces guardar en "estada" dos
Sino sino hay más caracteres imprimir "CADENA ACEPTADA"
Si no invocar a rutina de error
Fin caso cuatro.

Fin de casos.

5- Si "estada" no es 4 entonces
Imprimir "CADENA RECHAZADA"

6- Fin.

- 1- Inicio
- 2- Pedir una cadena y analizar y guardarlo en la variable "cadena"
- 3- Repetir lo sig desde el 1er carácter hasta el último carácter de "cadena"
Leer sig carácter y guardarlo en la variable "dato"
Si encontramos "a", empazamos a buscar una "b"

Si después de "a" encontramos una b, buscamos otro "b".

Si encontramos la 2da "b" la cadena es válida.

Si en cualquier momento encontramos algo diferente, volvemos a empezar.

4- Si terminamos encontrando "abb", mostrar "cadena aceptada". Si no mostrar "cadena rechazada".

5- Fin

	a	b
0	1	2
1	1	3
2	1	2
3	1	4
4	1	2 Acepta

Generadores de analizadores léxicos 28 Oct

Herramientas para separar cadenas leídas en elementos lexicográficos que corresponden a tokens.

Lex → C

Flex → C

P Cllex

ANTLR → Java, C, C++, C#, Python, Perl, Delphi, Ada 95, Javascript, Objective-C.

González Morales Víctor Eduardo

Fecha 28-Oct

CUP → Java

Cocktail - Rex

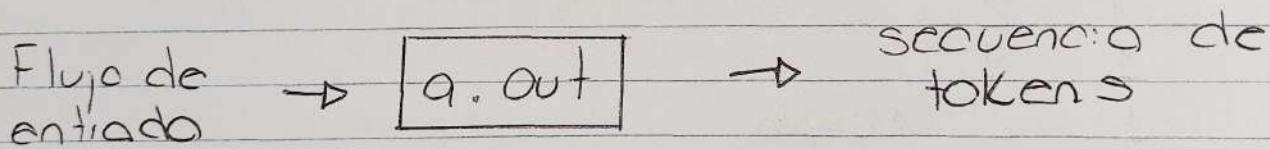
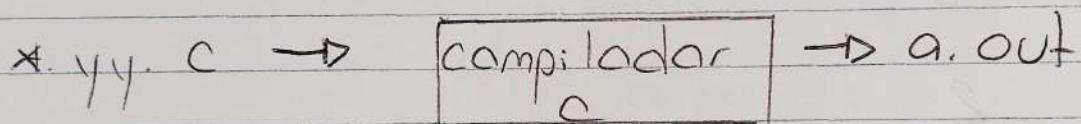
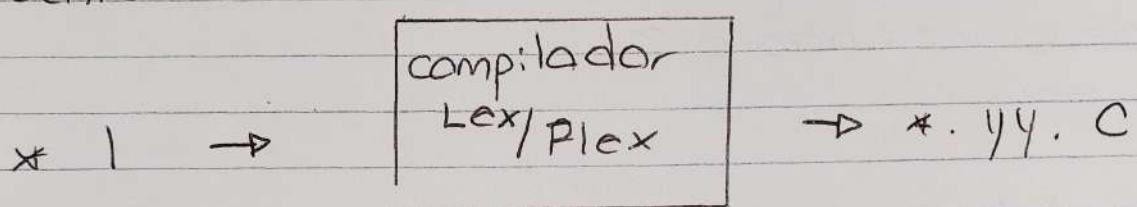
PCCT - DLG

Generan 'lexicos' automáticamente

Enfoque: Lex, Flex

Archivero

Fuente



Estructura de una fuente para Lex

declaraciones => declaración de variables
% %.

reglas => pares patrón - acción
% %.

funciones auxiliares => funciones que pueden compilarse aparte

Ej: %?

#include <stdio.h>

int valores = 0, consonantes = 0;

Máster

3%

Gonzalez Morales Victor Eduardo

Fecha

%. %.
[aeiouAEIOU] { vocales ++; }
(bcdflghjklmnprstuvwxyzBCDFGHJKLMNPQRSTUVWXYZ) { consonantes ++; }
. \n;
. %.
%

29 Oct

Ingrese una cadena: Esto es:

Una linea mas
otra linea. Para ver si funciona:
vocales minusculas: 17
consonantes minusculas: 15
vocales mayúsculas: 4
consonantes mayúsculas: 6
Otras simbolos : 1
No. de lineas: 3

int yywarp() { #2
 return 1;
}

```
int main() {  
    printf("Ingrese una cadena: ");  
    yylex();  
    printf("Vocales minúsculas: %.d\n", vocales);  
    printf("Consonantes minúsculas: %.d\n", consonantes);  
    printf("Vocales mayúsculas: %.d\n", vocalesmayus);  
    printf("Consonantes mayúsculas: %.d\n", consonantesmayus);  
    printf("Otras simbolos: %.d\n", caracter);  
    printf("No. de lineas: %.d\n", saltoFinal);  
    return 0;
```

Emraloz Morales Victor Eduardo

Fecha

29 Oct

1

```
% {  
#include <stdio.h>  
int vocales = 0, consonantes =  
0, vocalesMayus = 0, consonantesMayus = 0,  
saltoLinea = 0, caracter = 0;  
% }  
% /*  
[aeiou] { ++vocales; }  
[AEIOU] { vocalesMayus; }  
[bedfghijklmnpqrstuwyz] { ++consonantes; }  
[BCDFGHIJKLMNOPQRSTUVWXYZ] {  
    ++consonantesMayus; }  
[\n] { ++saltoLinea; }  
[.] { ++caracter; }  
*/
```

Espec. Finales
Léxicas.

30 Oct

Un archivo de especificación léxica para JFlex
consta de 3 partes divididas por una sola
línea que comienza con '%'.:

User Code

/* */

Opciones y declaraciones

/* */

Reglas Léxicas

En todas las partes de la especificación,
se permiten comentarios de la forma
/* */ o texto de comentario /* */ que
de final de línea de estilo Java que
comienzan con /*. Los comentarios de JFlex
se anidan, por lo que el número de /*

Rayter

Gonzalez Morales Victor Eduardo

Fecha
30/10/20+

y λ^* debe estar equilibrado.

Ej: Cómo trabajar con JFlex?

Para demostrar cómo se ve una especificación léxica con JFlex, esta sección presenta un ejemplo que cuenta vocales y consonantes en una cadena ingresada por teclado:

Archivo LetiasLexer.flex:

% %

% class Letiaslexer

% unicode

% standalone

% {

private int vocales = 0;

private int consonantes = 0;

private int lineas = 0;

public void printCounts() {

System.out.println("Número de vocales:
" + vocales);

System.out.println("Número de consonantes:
" + consonantes);

System.out.println("Número de lineas:" +
lineas);

}

% }

VOCAL = [aeiouAEIOUáéíóúÁÉÍÓÚ]

CONSONANTE = [q-zA-Z&[^aeiouAEIOUáéíóúÁÉÍÓÚ
ÁÉÍÓÚ]]

NEW LINE = [\r|\r\n]

1. 1.

{VOCAL} {vocales ++;}
{consonante} {consonantes ++;}
{NEW-LINE} {lincas ++;}

Código a Incluir

La primera sección, el código de usuario:
El texto hasta la primera linea que comienza con i. i. se copia literalmente en la parte superior de la clase de lexergenerado (antes de la declaración de clase `reol`). Junto con las declaraciones de package e import.
Por lo general no hay mucho que hacer.

Sí el código termina con un comentario de clase javadoc, la clase generada obtendrá este comentario, sino, JFlex generará uno automáticamente.

Opciones y declaraciones (macros)

La segunda sección, opciones y declaraciones, es más interesante. Consiste en un conjunto de opciones, código que se incluye dentro de la clase de escáner generada, estás lexicas y declaraciones de macros. Cada opción JFlex debe componer una linea de especificación y comenzar con un i. En nuestro ejemplo se utilizan las sig opciones:

- 1. class LetrasLexer, le dice a JFlex que le dé a la clase generada el nombre LetrasLexer y que escriba el código en un archivo LetrasLexer.java.
- 1. unioade define el conjunto de caracteres en los que trabajará el escáner.
- 1. standalone indica que el analizador léxico generado será un programa independiente. Esto significa que el analizador no dependerá de otro programa o entorno para funcionar.

El código entre 1. E y 13 se copia literal en el código fuente de la clase lexer generada. Aquí puede declarar variables miembro y funciones que se utilizan dentro de las acciones del escáner. Al igual que con todas las acciones de JFlex, tanto 1. { como 1. ; deben comenzar una línea.

La especificación comienza con declaraciones de macros. Las macros son abreviaturas de especificaciones regulares, que se utilizan para facilitar la lectura y comprensión de las especificaciones léxicas. Una declaración de macro consta de un identificador de macro seguido de = y de la expresión regular que representa. Esta expresión regular, puede contener o su vez usos de macros.

En el ejemplo se definen las variables vocales, consonantes y líneas para almacenar

los conteos de vocales, constantes y líneas así como el método printCount que imprime los valores, consonantes y líneas de cada contador.

Aunque esto permite un estilo de especificación similar a la gramática, las macros siguen siendo solo abreviaturas y no terminales, no pueden ser recursivas. JFlex detecta y notifica las ciclas en las definiciones de macros en el momento de la generación.

Reglas y acciones

La sección contiene expresión regulares y acciones que se ejecutan cuando el escáner coincide con la expresión regular asociada. A medida que el escáner lee su entrada, realiza un seguimiento de todas las expresiones regulares y activa la acción de la expresión que tiene lo largo. Si 2 expresiones regulares tienen la coincidencia más larga para una entrada determinada, el analizador elige la acción de la expresión que aparece primero en la especificación.

Aquí cada expresión regular tiene una acción que incrementa el contador correspondiente:

- Si se encuentra una vocal, se incrementa vocales,
- Si se encuentra una consonante, incrementa consonantes.
- Si se encuentra un salto de línea, incrementa líneas.
- Otros caracteres (puntos, espacios, etc.) se ignoran.

Gonzalez Morales Victor Eduardo

Fecha
31/0ct

Ademas de los coincidencias de expresiones regulares, se pueden utilizar estados léxicos para refinar. Un estado léxico oculta como una condición de inicio.

Implementación y ejecución.

Para implementar el analizador léxico generado con JFlex en Java y contar vocales, consonantes y líneas desde consola, necesitarás 2 archivos principales:

- 1- El archivo .flex con las especificaciones para JFlex
- 2- El archivo .java principal que ejecuta el analizador.

Para generar el analizador léxico en Java a partir del archivo .flex, ejecuta el sig. comando en la terminal:

jflex LetrasLexer.flex

Esto generará un archivo llamado LetrasLexer.java en el mismo directorio. Aquí se muestra el código de InputLexer.java para crear el archivo java que se iniciará y ejecutará después leerá una cadena de texto desde el teclado y utilizará el analizador para procesarla.

Se usa un java.io.String Reader para permitir que JFlex lea una cadena proporcionada por el usuario directamente desde el teclado.

```
import java.io.StringReader;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class InputLexer {
    public static void main (String [] args) {
        try {
            BufferedReader reader = new BufferedReader (new InputStreamReader (System.in));
            System.out.print ("Ingresar lo que deseo a analizar:");
            String inputText = reader.readLine();
            LetrasLexer lexer = new LetrasLexer (new StringReader (inputText));
            int token;
            while ((token = lexer.yylex ()) != LetrasLexer.EOF) {
                lexer.printCounts ();
            }
            catch (IOException e) {
                System.out.println ("Error al leer la entrada:");
                + e.getMessage ();
            }
        }
    }
}
```

González Morales Víctor Eduardo

Fecha
31-0ct

Explicación:

Se utiliza BufferedReader junto con InputStreamReader(System.in) para leer lo introducido desde el teclado.

Se utiliza StringReader para crear una instancia del analizador léxico LetrasLexer y pasarsela la cadena leída.

El resto del código procesa la entrada carácter a carácter, llamando a lexer.yy.lex() y luego se imprime el conteo de vocales, consonantes y líneas.

Para ejecutar el analizador, primero se compilan ambas archivos .java en un terminal:

java o InputLexer, java LetrasLexer, java

Luego, ejecuto el programa java InputLexer

Ej. de ejecución:

Ingreso la cadena a analizar:

Hola, cómo estás?

Número de vocales: 6

Número de consonantes: 7

Número de líneas: 1

Gonzalez Morales Victor Eduardo

Fecha

4/11/2018

Hacer un lexer stand alone completo en JFlex
y la clase que lo invoca para que leo
desde un archivo de texto plano las
cadenas de texto en cada linea y reco-
nozco como tokens: números enteros, números
hexadecimales, números octales e identi-
ficadores, usando los siguientes:

Letra: Cualquier letra (mayúscula o minúscula).

Dígito: Cualquier dígito (0 o 9)

Alfanumérico: cualquier letra o dígito.

Número: secuencia de dígitos (0 o más).

Entero: un entero positivo (comenzando en 0
o en un dígito distinto de 0)

Octal: un número en base 8, comenzando con 0

Hexadecimal: un número hexadecimal, comen-
zando con 0x.

Identificador: un identificador que comienza
con una letra y puede tener letras, dígitos
y un guión bajo (`_`) después de la
primera letra.

Cuando identifiques algún lexema que cumple
con algún patrón debe escribirse en un archivo
de texto plano el lexema rellenando
el token correspondiente a ese lexema
y un salto de linea.

Letra = [a - zA - Z]

Dígito = [0 - 9]

Alfanumérico: {LETRA} | {Dígitos}

Número: {Dígitos} +

Gonzalez Morales Victor Eduardo

Fecha
05-nov

Entero = 0 | {Numeros}
Octal 0 [0-7] +

HEXADECIMAL = 0x | x[0-9 a-f A-F] +
IDENTIFICADOR = Letras - ?
& ALFANUMÉRICOS +

01

0103

02

0102

102 ← Entero

0x0-9

0x

07 - NOV

```
import java.io.*;  
public class Main {  
    public static void main(String[] args) {  
        if (args.length < 2) {  
            System.out.println("Uso : java Lexer  
Main <archivo-entrada><archivo-salida>");  
            return;  
        }  
        String archivoEntrada = args[0];  
        String archivoSalida = args[1];  
        try {  
            BufferedReader reader = new BufferedReader(  
                new FileReader(archivoEntrada));  
            BufferedWriter writer = new BufferedWriter(  
                new FileWriter(archivoSalida));  
        }  
        LexerClass lexer = new LexerClass  
        (reader, writer);  
        while (true) {
```

```
lexer.yylex();  
if(lexer.isEOF()) break;  
}
```

```
System.out.println("Análisis completado.  
Tokens guardados en " + archivoSalida);  
} catch (IOException e) {  
e.printStackTrace();  
}  
}
```

```
-----  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.io.Reader;  
//  
//  
// public  
// class LexerClass  
// extends  
// standalone  
// {
```

```
private BufferedWriter writer;  
public boolean isEOF() {  
return zzA+EOF;  
}
```

```
public LexerClass(Reader in, BufferedWriter  
writer, writer) {  
this.zzReader = in;  
this.writer = writer;  
}
```

```
private void escribeToken(String lexema,  
String token) {  
try {
```

González Marañón Víctor Eduardo

Fecha: 07-nov

```
writer.write(exemps + " - " + token + "\n");  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

1.3

DIGITO = [0 - 9]

LETRA = [a - zA - Z]

ALFANUMERICO = {LETRA} {DIGITO}*
IDENTIFICADOR = {LETRA} {ALFANUMERICO}

NUMERO = {DIGITO}*

N-LINEA = "\n"

OTRO = [. \d] +

1.1.

<yy initial> {

```
{ IDENTIFICADOR } { EscribeToken(yyText(),  
"IDENTIFICADOR") };  
{ numero } { EscribeToken(yyText(), "numero") };  
{ N-LINEA } { /* Ignorar cualquier salto */ };  
{ OTRO } { /* Ignorar otro carácter */ };  
}
```

ols 1 12 vc - Identificador

0564 - Número

98547 - Número

mbj - Identificador

zib - Identificador

33 - Número

s - Identificador

0 - Número

XABZ3 - Identificador

105 - Número

F - Identificador

11-nov

Fecha

Agregar al lexer en proceso la siguientes palabras reservadas, operadores simbolos
capacidad para reconocer las
int - INT
float - FLOAT
char - CHAR
CharString - CHARSTRING
if - IF
else - ELSE
do - DO
repeat - REPEAT
for - FOR
begin block -
BEGINBLOCK
end bloc K -
ENDBLOCK
while -
WHILE
(- LEFTPART
) - RIGHTPART
read - READ
end program

< - LT RELATION
> - GT RELATION
<= - EQ RELATION
>= - EG RELATION
:= - EQ RELATION
~= - NEQ RELATION
+ - SUMOP
- - SUBOP
* - PRODOP
/ - QUOTOP
= - ASIGNAOP
and - ANDLOG
or - ORLOG
not - NEGLOG
display - DISPLAY
read - READ

Contenido del Archivo de
Lectura: in.txt
program MI_PROG;
begin block
int x,y;
float pocketo;
display ("DAME X: ~");
read(x);
display ("DAME Y: ~");
read(y);
pocketo = x * y;
if (pocketo > 100)
begin block
display ("CORRECTO");
end block
end program

Mayter

González Morales Víctor Eduardo

Fecha
12/11/2017

IF = "if"

FLOAT = "float"

ELSE = "else"

CHAR = "char"

PARENTESIS = "("

DO = "do"

INT = "int"

REPEAT = "repeat"

DIGITO = [0 - 9]

LETRA = [a - z A - Z]

ALFANUMERICO = {LETRA} {DIGITOS}

IDENTIFICADOR = {LETRA} {ALFANUMERICOS}

NUMERO = {DIGITOS} +

N-LINEA = "\n"

OTRO = [. \r \t]

;

yy init();

{ IDENTIFICADOR } { escribirToken(yy init());
IDENTIFICADOR "); }

{ NUMEROS } { escribirToken(yy text(), "NUMERO"); }

{ N-LINEAS } { /* Ignorar saltos de linea */ }

{ OTROS } { /* Ignorar otro caracter */ }

{ PARENTESIS } { escribirToken(yy text(), "PARENTESIS"); }

{ INT } { escribirToken(yy text(), "INT"); }

{ FLOAT } { escribirToken(yy text(), "FLOAT"); }

{ CHAR } { escribirToken(yy text(), "CHAR"); }

{ DO } { escribirToken(yy text(), "DO"); }

{ REPEAT } { escribirToken(yy text(), "REPEAT"); }

Zalocz Morales Victor Eduardo

Fecha
14-nov

d IV - "Análisis Sintático"

del analizador sintático

independientes de contexto libre

síntesis descendente

síntesis ascendente

por precedencia de operadores

sintáticos LR

ambiguas

síntesis sintáticas

o de errores

programación

ss. Por ej.

expresiones

sintáticas de contexto

señadores y desarrollos

precisos

analizadores sintáticos

solución de ambigüedades y errores

modela la modificación del lenguaje.

- analizador léxico

secuencia de tokens según una gramática.
el programa es válido, genera árbol sintático.

González Morales Víctor Eduardo

Fecha

14-nov

Unidad IV - "Análisis Sintáctico"

- 4.1 - El papel del analizador sintáctico
- 4.2 - Gramáticas independientes de contexto libre
- 4.3 - Análisis sintático descendente
- 4.4 - Análisis sintático ascendente
- 4.5 - Análisis sintático por precedencia de operadores
- 4.6 - Analizadores Sintáticos LR
- 4.7 - Uso de gramáticas ambiguas
- 4.8 - Generadores de analizadores sintáticos

Análisis Sintáctico: Conceptos, manejo de errores

Visión general de la sintaxis de lenguajes de programación

- Los lenguajes siguen reglas sintácticas precisas. Por ej.: un programa tiene: Bloques, sentencias, expresiones y componentes léxicos.
- La sintaxis se describe con: Gramáticas de contexto libre o Notación BNF.

Las gramáticas ayudan a diseñadores y desarrolladores de compiladores:

- Son especificaciones precisas
- Permiten generar analizadores sintáticos
- Facilitan la detección de ambigüedades y errores
- Hacen más sencilla la modificación del lenguaje.

Concepto de analizador léxico

- Verifica secuencia de tokens según una gramática.
- Si programa es válido, genera árbol sintáctico.

- En práctico, dirige el proceso de compilación
- Incluye acciones semánticas para resto de bases.
- Detecta errores sintácticos y se recupera de ellos.
- Controla el flujo de tokens del analizador
- Se designa compilación dirigida por sintaxis.

Manejo de errores en compiladores.

Clasificaciones de errores:

- Léxicos: errores en identificadores, palabras clave, operadores.
- Sintáticos: parentesis o expresiones mal estructurados.
- Semánticas: operadores con operandos incompatibles.

Errores lógicos y de corrección:

- No siempre pueden detectarse
- Requieren análisis de intenciones o flujo de programa

Los compiladores no se fijan en esos errores:

- Errores de sintaxis impiden crear el árbol sintático.
- El manejo de errores desde el inicio mejora la estructura y respuesta del compilador.
- Un buen manejo ayuda a diagnosticar y describir errores.

Manejo de errores de sintaxis en compiladores:

Objetivo: Recuperarse de un error y continuar compilación.
El analizador debe:

- Indicar errores claros y precisamente
- Recuperarse para seguir analizando entiendo
- Distinguir errores y advertencias
- Evitar ralentizar compilación.

Estrategias de manejo de errores

Ignora el problema (Panic Mode).

- Ignora tokens hasta una condición segura (`;` o `End`)

- Desecha tokens entre el error y token seguro

- Continúa análisis desde condición segura

Recuperación a nivel de fiose

- Corrige error insertando tokens (`;`).

- Cuidado: podría provocar recuperaciones infinitas si la corrección introduce errores nuevos.

Reglas de producción adicionales

- Agrega reglas gramaticales para errores comunes.

- Permite dirección y corrección automática de errores.

- Emite advertencias en vez de errores si es posible.

Corrección global

- Genera árbol sintáctico completo a partir de scannos

- Devuelve versión corregida de entrada inicial.

- Crea árbol sintáctico para secuencia sin errores

Gramática en un analizador sintáctico:

- Utiliza gramáticas de contexto libre para economizar sentencias.

- Definición de una gramática G :

N : no terminales

T : terminales

P : Reglas de producción

S : Axioma inicial

Pueden existir múltiples derivaciones en una pseudocadena:

- Izq: reescribe el no terminal más a la izquierda

- Der: reescribe el no terminal más a la derecha.

Derivaciones en gramática:

- Una regla de producción es una regla de reescritura.
- Un no terminal se reemplaza por la pseudocadena en el lado derecho.
- Pseudocadena: Secuencia de terminales y/o no terminales.
- Notación de derivación: $a \Rightarrow b$.
- Las pseudocadenas derivadas del axioma inicial son fórmulas sentenciales.

Árbol sintético en lenguajes de programación:

- Un árbol sintético representa la estructura de una sentencia.
- La raíz es el axioma inicial de la gramática.
- Los nodos internos son los no terminales de las reglas de producción.
- Cada nodo tiene tantas hijas como símbolos en la regla aplicada.
- Si la sentencia es incorrecta, no se genera el árbol.
- Ambigüedad: Una sentencia admite múltiples árboles.
- La ambigüedad se resuelve cambiando la gramática o aplicando reglas.

Gramática y Derivaciones

16 - Nov

Gramática no ambigua que reconoce expresiones aritméticas.

$$\begin{aligned} N &= \{E, T, F\} \\ T &= \{id, num, +, *, (), ()\} \\ S &= E \end{aligned}$$

$$\begin{aligned} P &= \{E \rightarrow E + T \mid T \\ &\quad T \rightarrow T * F \mid F \\ &\quad F \rightarrow id \mid num \mid (E)\} \end{aligned}$$

Objetivo: Construir la cadena de tokens: $id_1 * id_2 + id_3$

Mayte ✓

González Morales Víctor Eduardo

Fecha
16-nov

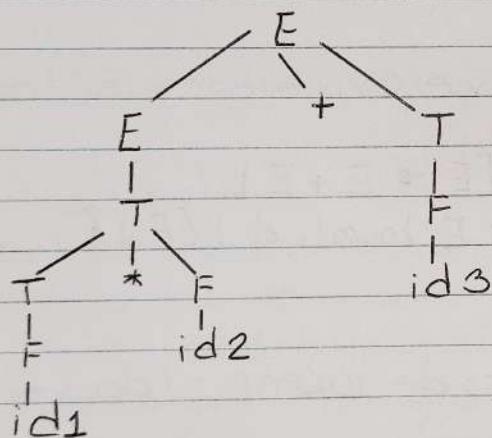
Derivaciones a izquierdo

$$\begin{aligned} E \Rightarrow E + T &\Rightarrow T + T \Rightarrow T * F + T \Rightarrow F * F + T \Rightarrow id_1 \\ * F + T &\Rightarrow id_1 * id_2 + T \Rightarrow id_1 * id_2 + F \Rightarrow id_1 * \\ id_2 + id_3 \end{aligned}$$

Derivaciones a derecho

$$\begin{aligned} E \Rightarrow E + T &\Rightarrow E + F \Rightarrow E + id_3 \Rightarrow T * F + id_3 \Rightarrow T \\ + id_2 * id_3 &\Rightarrow F + id_2 * id_3 \Rightarrow id_1 + id_2 * id_3 \end{aligned}$$

Árboles sintácticos



Gramática y derivaciones

Gramática ambigua que reconoce expresiones aritméticas

$$N = \{E\}$$

$$T = \{id, num, +, *, (,)\}$$

$$S = E$$

$$P = \{E \Rightarrow E + E \mid$$

$$E * E \mid num \mid id \mid (E)\}$$

Objetivo: construir la cadena de tokens: $id_1 * id_2 + id_3$

Derivaciones a izq: $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id_1$
 $* E \Rightarrow id_2 + E \Rightarrow id_1 * id_2 + id_3$

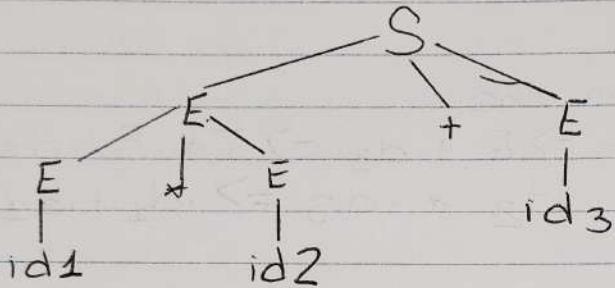
González Morales Víctor

Fecha
16-nov

Derivaciones a derechas

$$E \Rightarrow E + E \Rightarrow E + id_3 \Rightarrow E * E + id_3 \Rightarrow E * id_2 + id_3 \\ \Rightarrow id_1 * id_2 + id_3$$

Árboles Sintácticos



Gramática y derivaciones

Gramática ambigua que reconoce expresiones aritméticas

$$N = \{E\}$$

$$P = \{E \rightarrow E + E\}$$

$$T = \{id, num, +, *, (,)\} \quad E \rightarrow E \text{ Inum}, d \mid (E)$$

$$S = E$$

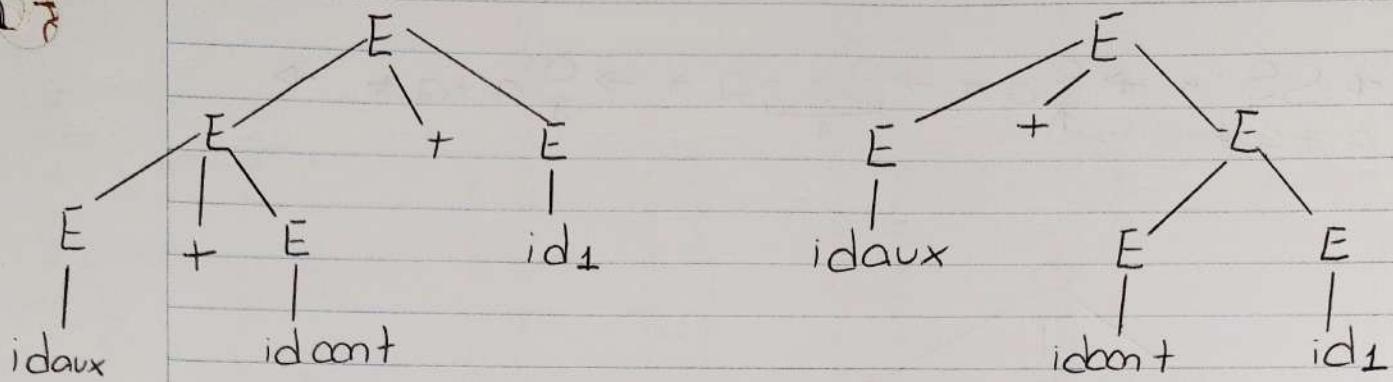
Objetivo construir la cadena de tokens: id1num * id2num + id3num

Derivaciones a derecha

$$E \Rightarrow E + E \Rightarrow E + id_1 \Rightarrow E + E + id_1 \Rightarrow E + id_{cont} + id_1 \\ \Rightarrow id_{aux} + id_{cont} + id_1$$

Derivaciones a izquierdo

$$E \Rightarrow E + E \Rightarrow id_{aux} + E \Rightarrow id_{aux} + E + E \Rightarrow id_{aux} + id_{cont} + E \\ \Rightarrow id_{aux} + id_{cont} + id_1$$



Considerando la siguiente gramática libre de contexto:

$$S \rightarrow SS + I S S * I a$$

Sea la cadena: $a a + a *$

- Realice la derivación a derecho para la cadena
- Realice la derivación a izquierda para la cadena
- Diseñe un árbol sintáctico para la cadena
- Es ambigua esta gramática? Justifique la respuesta

Considerando la gramática libre de contexto.

$$S \rightarrow \emptyset S 1 1 \emptyset 1$$

Sea la cadena $\emptyset \emptyset \emptyset 1 1 1$

Responda las incisos y pregunta del problema 1

Considerando la gramática libre de contexto

$$S \rightarrow S + S I S S I (S) I S * I a$$

Sea la cadena $(a + a) + a$

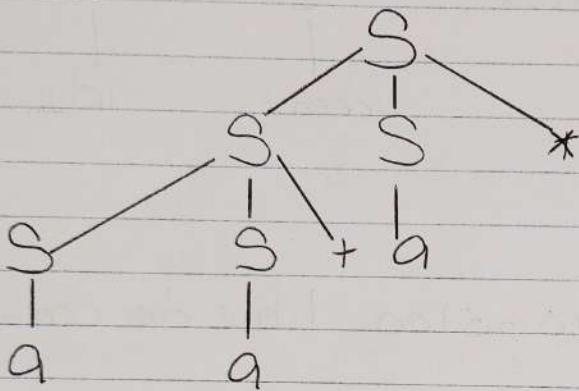
Responda las incisos y preguntas del problema 1

Gonzalez Morales Victor

Fecha 19-nov

a)

$$S \rightarrow SS^* \rightarrow Sa^* \rightarrow \underbrace{SS+a^*}_{\downarrow} \rightarrow Sa+a^* \rightarrow$$



b) $S \rightarrow SS^* \rightarrow SS + S^* \rightarrow aS^* + S^*$
 $\rightarrow aa + \frac{S}{a}^* \rightarrow aa + a^*$

