**INTRODUCCIÓN**

**Traductor**

Es un programa que transforma código fuente en código objeto. Los hay de 1 pasada y de múltiples pasadas.

Tipos de traductores:

* Compilador: traducción y ejecución están separados; ej: C, Pascal
* Intérprete: la traducción se hace a medida que se ejecuta; ej: PHP, javascript
* Compilador JIT: compilación a bytecode y luego interpretación de bytecode a código nativo; ej.: Java, .Net, Python

**Estructura básica de un compilador**

Analizador Léxico

Generador Cód. Intermedio

Analizador Sintáctico

Analizador Semántico

Optimizador

Generador de Código

Tabla de símbolos

Tratamiento de Errores

Entrada Salida

**ANÁLISIS**

**GENERACIÓN**

Fase de análisis

* *Analizador léxico*:aísla la entrada y genera una lista de *tokens*.
* *Analizador sintáctico*:verifica si la secuencia de tokens es correcta según reglas sintácticas (gramática). Genera el árbol sintáctico.
* *Analizador semántico*:recibe una entrada bien formada y verifica que tenga sentido. Reúne información para fases posteriores.

Fase de síntesis

* *Generador de código intermedio*: genera una representación intermedia de más bajo nivel pero independiente del hardware
* *Optimizador*: análisis y modificación del código intermedio para optimizar el procesamiento.
* *Generador de código*: generación de código nativo

Tabla de símbolos

Se completa en las fases de análisis léxico y sintáctico y reúne información sobre los distintos atributos (tipo de atributo, tipo de datos, cantidad y tipos de parámetros, etc.).

Es utilizada para diversas comprobaciones en el análisis semántico (comprobación de tipos, existencia de variables, etc.) y es consultada para la generación de código.

Detección de errores

Cada fase es capaz de reconocer distintos tipos de errores:

* *Léxico*: caracteres que no forman ningún componente léxico.
* *Sintáctico*: secuencias de caracteres mal formadas.
* *Semántico*: errores de significado.

Ante la presencia de errores se puede detener la compilación o tratar de recuperar el error y continuar compilando.

**Teoría de lenguajes**

**Alfabeto**

Es un conjunto no vacío de letras o símbolos. Se representa como .

Las letras griegas minúsculas denotan símbolos genéricos y los símbolos puntuales se denotan con letras latinas minúsculas. Ejemplos de alfabetos: ,

**Palabra**

Es toda secuencia finita formada por las letras de un alfabeto. Ejemplos sobre : juan, pedro y coche.

Para representar palabras se usarán las últimas letras del alfabeto latino; ej.: x = pedro

Las palabras se construyen a partir de letras, es necesario:

* Palabra vacía
* Constructor de palabras (concatenación): a partir de y x forma una palabra .x
* Analizadores inicial y resto.

Longitud de una palabra: Número de letras que la componen. Se representa con la notación |x|. Ej.: |juan| = 4; || = 0

Universo del discurso o Lenguaje universal: Es el conjunto de todas las palabras que se pueden construir con las letras de un alfabeto. Se representa con . Es un conjunto infinito.

Operaciones con palabras

* *Concatenación*: Sean e palabras que pertenecen a , se representa como o a , tal que está formada por las letras de seguidas de las letras de .
* *Potencia*: Se llama potencia i-ésima de una palabra a la operación que consiste en concatenarla veces consigo misma.

Ej: = ( veces)

* *Palabra refleja o inversa*: Sea , se denomina inversa a

**Lenguajes**

Se denomina lenguaje sobre el alfabeto a cualquier subconjunto del lenguaje universal .

*Lenguaje vacío*: . Es distinto de ya que y

Operaciones con lenguajes

Sean y lenguajes que pertenecen a

* *Unión*:

Ej:

* *Concatenación*:

Ej:

* *Potencia*: Consiste en concatenarlo consigo mismo.
* *Cierre o clausura de L*: Unir L con todas sus potencias.
* *Clausura positiva de L*:
* *Lenguaje inverso de L*:

**Gramáticas**

Describe la estructura de las frases y de las palabras de un lenguaje mediante reglas.

Permiten expresar lenguajes infinitos en forma finita.

Las reglas definen ciertos términos en función de otros y se representan mediante la siguiente notación:

Ej.:

Operaciones

* *Derivación directa*:

Es la aplicación de una regla para obtener una palabra a partir de otra. Se dice que deriva directamente de , si , aplicando la regla se llega a .

* *Derivación*:

Es la aplicación de más de una regla para obtener una palabra a partir de otra. Se dice que deriva de , si tras sucesivas derivaciones de , obtengo :

* *Relación de Thue*:

Existe una relación de Thue entre y si o .

**Definición formal de gramática**

Se llama gramática formal a la cuádrupla: ; donde:

* es el alfabeto de símbolos de Terminales.
* es el alfabeto de símbolos de No Terminales.
* es el elemento distinguido o axioma.
* es un conjunto finito de producciones.

Notación de Backus (BNF)

Es una notación abreviada para reglas que comparten la parte izquierda:

|  |  |
| --- | --- |
|  |  |

Lenguaje asociado a una gramática o lenguaje generado por una gramática

Se denomina así al conjunto de todas las sentencias de .

Dos gramáticas son equivalentes cuando describen el mismo lenguaje.

Recursividad: Una producción es recursiva si posee la forma

Si , la gramática es recursiva a izquierda.

Si y, la gramática es recursiva a derecha.

Si un lenguaje es infinito, la gramática que lo representa tiene que ser recursiva.

**Clasificación de Chomsky**

Tipo 0: Gramática sin restricciones

Tipo 1: Gramática sensible al contexto

No se admiten derivaciones en .

Tipo 2: Gramática independiente al contexto

No se admiten derivaciones en .

Tipo 3: Gramática regular o lineal.

Aceptan sólo 3 tipos de producciones:

|  |  |
| --- | --- |
| *Lineales por la izquierda* | *Lineales por la derecha* |

**Árbol de derivación**

Es una representación gráfica de las derivaciones para gramáticas de tipo 1, 2 o 3.

La raíz del árbol se etiqueta con el axioma de la gramática. Por cada derivación directa, desde el nodo etiquetado con el símbolo terminal que se sustituye, se hace surgir un conjunto de arcos que se dirigen a nodos etiquetados con los símbolos de la cadena por la que se sustituye.

Se denomina subárbol a una parte del árbol de derivación que pende de un nodo asociado a un no terminal e incluye todos los nodos que descienden del mismo.

E

E

E

E

x

x

x

x

+

+

y

y

\*

\*

**Gramáticas ambiguas**

Una gramática es ambigua si posee al menos una sentencia ambigua. Una sentencia es ambigua cuando es posible obtenerla mediante más de un árbol de derivación.

Un lenguaje es inherentemente ambiguo si no es posible representarlo mediante una gramática no ambigua.

**Gramáticas limpias y bien formadas**

1. Regla de producción innecesaria

Es de la forma

Hacen la gramática ambigua y no aportan a la generación de palabras.

Estas reglas deben eliminarse

1. Símbolo inaccesible desde el axioma

No es el axioma y no aparece en la parte derecha de ninguna de las reglas alcanzables desde el axioma.

Todo símbolo accesible desde el axioma cumple que

Pasos:

* 1. Hacer una lista de los símbolos de la gramática ( y ) y marcar el distinguido.
  2. Por cada regla de la forma , donde está marcado, marcar todos los símbolos de la derecha.
  3. Repetir paso b hasta que no se marque ningún símbolo.
  4. Eliminar todos los símbolos no marcados de los alfabetos.
  5. Eliminar todas las producciones que contengan alguno de estos símbolos.

1. No terminal no generativo

Cuando el lenguaje generado a partir de ese símbolo es el vacío.

Un símbolo U no es no generativo si

Toda regla que contenga un símbolo no generativo se denomina regla superflua.

Pasos:

* 1. Hacer una lista de los símbolos no terminales de la gramática.
  2. Por cada regla de la forma , donde u está formada únicamente por terminales y no terminales marcados, marcar .
  3. Repetir b hasta que no se marque ningún símbolo.
  4. Eliminar todos los símbolos no marcados del conjunto de no terminales.
  5. Eliminar todas las producciones que contengan alguno de estos símbolos.

1. Gramática reducida es aquella que no posee símbolos inaccesibles desde el axioma, símbolos no generativos ni reglas superfluas.
2. Una gramática está limpia si es reducida y no posee reglas innecesarias.

Ejemplo:

|  |  |
| --- | --- |
| *Gramática original* | *Gramática limpia* |

1. Las reglas de la forma son reglas no generativas.

Si el lenguaje no posee la palabra vacía, se pueden eliminar todas, sino solo hay que dejar la palabra vacía en el axioma.

Pasos:

1. Tomar una regla de la forma y eliminarla de la gramática.
2. Por cada regla de la gramática donde aparece en la parte derecha, , añadir la regla (a menos que ésta exista).
3. Repetir hasta que no haya reglas que deriven en lambda o solo quede una, siendo el axioma la parte izquierda de la misma.

Ejemplo anterior:

|  |  |
| --- | --- |
| *Gramática limpia anterior* | *Sin reglas no generativas* |

1. Las reglas de la forma son reglas de redenominación

Pasos:

* 1. Tomar una regla de la forma y eliminarla de la gramática.
  2. Por cada regla de la gramática de la forma , añadir la regla (a menos que esta exista).
  3. Repetir hasta que no haya reglas de redenominación.

Este algoritmo puede introducir reglas innecesarias.

Ejemplo anterior

|  |  |
| --- | --- |
| *Gramática limpia anterior* | *Sin reglas no generativas* |

1. Gramática bien formada es aquella que está limpia y no posee reglas no generativas o de redenominación.

**ANÁLISIS LÉXICO**

**Expresiones regulares**

Permiten expresar lenguajes regulares**.** Se consideran un metalenguaje.

Si es una expresión regular, entonces es el conjunto descripto por . Podemos decir que la expresión regular denota el lenguaje de la cadena .

Dos expresiones regulares son equivalentes si designan el mismo conjunto regular.

Para poder describir los lenguajes regulares definimos determinadas operaciones relacionadas a las operaciones de los lenguajes regulares.

Operaciones

Sean y dos expresiones regulares:

* *Unión o alternativa* –

Puede aparecer de manera indistinta.

* *Concatenación* -

Es seguido de .

* *Cierre* –

Viene cero o más .

* *Cierre positivo* -

Viene una o más .

Precedencia de las operaciones

1. Paréntesis, 2. Cierres, 3. Concatenación, 4. Alternativa

**Autómatas**

Un autómata reconocedor de un lenguaje funciona de tal forma que cuando recibe en su entrada una cadena de símbolos indica si la misma pertenece o no al lenguaje.

El estado de un autómata es toda la información necesaria en un momento dado para poder deducir, dado un símbolo de entrada en el momento actual, cuál será el símbolo de salida.

Un autómata tiene un determinado número de estados y se encontrará en uno u otro dependiendo de la historia de símbolos que han llegado.

Al arribar un símbolo de entrada se produce una transición de un estado a otro.

Definición formal de autómata finito

Una máquina de estados finitos M es un quíntuplo donde:

es un conjunto de identificadores de estados.

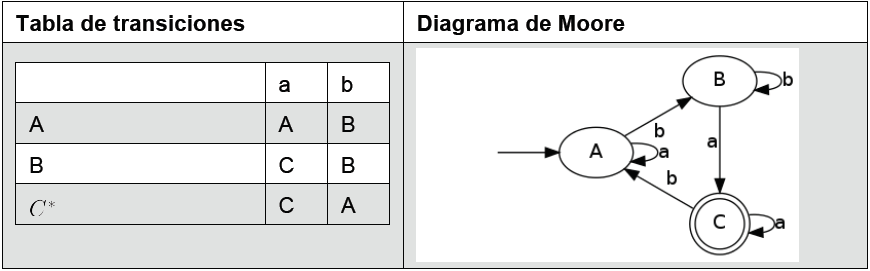
es el alfabeto de entrada.

es el estado inicial.

es un conjunto de estados finales.

es la función de transición, donde a partir de un estado y un símbolo indica el nuevo estado.

Representación de Autómatas



**Determinísticos vs No determinísticos**

Un autómata es determinístico cuando para cada par (estado, símbolo) hay solo un estado relacionado. Es una función.

Un autómata no determinístico permite que por cada par (estado, símbolo) haya más de un estado relacionado.

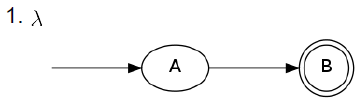
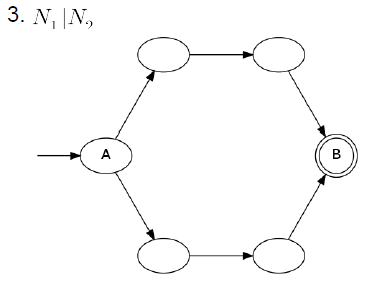
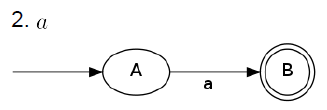
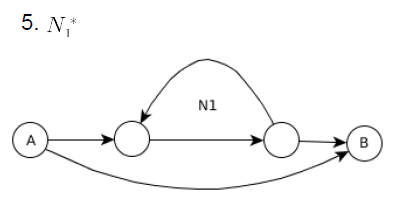
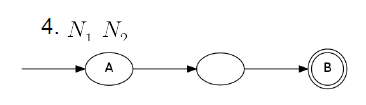
Los autómatas finitos deterministas son más veloces como reconocedores (la velocidad es proporcional al tamaño de la entrada), pero la tabla que los representa es más grande.

Vamos a ver como pasar de ER → AFN → AFD → AFDEM

ER → AFN. Construcción de Thompson

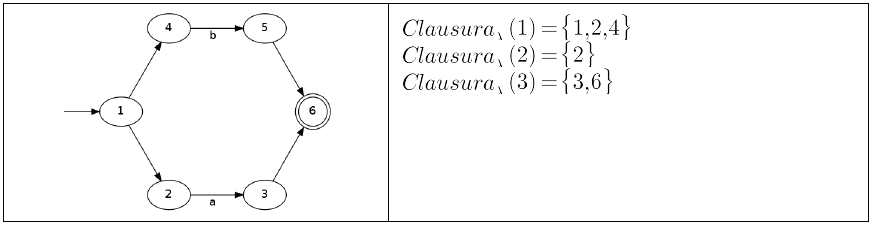
Se dan autómatas básicos para algunas ERs y luego se combinan para formar ERs más complejas.

Todos los AFN tienen un único estado inicial y un único estado final.

AFN → AFD

: es el conjunto de todos los estados alcanzables desde por medio de .



*Estado propio*: vamos a denominar así a los estados que poseen una transición de salida distinta de lambda o son estados de aceptación. En el ejemplo anterior 2 y 4 serían estados propios.

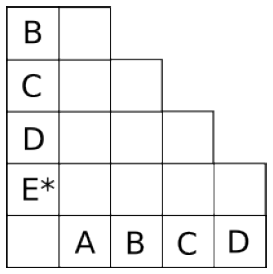
AFD → AFDEM

Un AFD está minimizado si todos sus estados son distinguibles y alcanzables

* Dos estados son *equivalentes* si al unirse en un solo estado pueden reconocer el mismo lenguaje regular que si estuvieran separados.
* Dos estados no equivalentes son estados *distinguibles*.
* Un estado es *alcanzable* si es posible llegar a él desde el estado inicial.

Método 1: Particionar los estados.

Método 2: La escalerita



**Autómatas de pila (AP)**

¿Qué ocurre si queremos reconocer el lenguaje de los paréntesis balanceados con un autómata finito?

Se podría agregar algo a los AF para darle más poder de cómputo; ej. un contador.

¿y si queremos reconocer palíndromos? Ej.: ANITALAVALATINA

Acá vemos que con el contador no alcanza, podemos agregar más memoria; en este caso en forma de pila.

Funcionamiento de un autómata de pila

La pila posee un comportamiento LIFO y **solo se puede modificar su tope**.

La pila va a ser única (sólo una pila va a existir).

La pila posee un alfabeto propio, que puede o no coincidir con el de entrada.

Al comienzo la pila se encuentra vacía, luego se pueden agregar o quitar elementos y al final, para aceptar una palabra, la pila debe quedar vacía.

Las transiciones indican lo que se consume de la entrada, lo que se consume de la pila y lo que se ingresa a la pila.

Primero se saca de la pila y luego se ingresa.

indica que se consume a de la entrada, nada del tope de pila y se apila .

indica que se consume a de la entrada, se quita del tope de pila y se apila . Esta operación falla si no está en el tope de pila.

ídem a la anterior, pero deja en la pila.

Los AP poseen **estados** normales y finales.

Para que una palabra sea aceptada se debe cumplir:

* La palabra se tiene que haber agotado ($ en la entrada)
* El AP debe estar en un estado final.
* La pila debe estar vacía (Z0 en el tope)

**ANÁLISIS SINTÁCTICO**

**Analizador sintáctico**

Convierte la secuencia de tokens de entrada en árboles de derivación

Si el analizador no es capaz de generar un árbol, entonces la cadena no pertenece al lenguaje

Se basan en autómatas de pila y gramáticas libres de contexto

Hay distintas técnicas para obtener el AP; no todas ellas son capaces de reconocer todas las gramáticas

**Técnica LL**

* . Los tokens se leen de izquierda a derecha.
* . Las derivaciones se hacen lo más a la izquierda posible.
* Es una técnica descendente: trata de armar el árbol desde la raíz hacia las hojas.
* Predictivo: predice las reglas necesarias para llegar a las hojas.
* Estando en un NT, tengo que predecir por cual derivación expandir el árbol. Cuando hay más de 1 posibilidad tengo que ver que token sigue en la entrada. Este token se denomina *lookahead*.
* En todo momento con tokens de lookahead debo poder determinar unívocamente que regla derivar:

**Algoritmo**

1. Escribir la gramática
2. Por cada regla escribir el *conjunto first*: Dada una regla, el conjunto First representa los posibles tokens con los que comienza esa regla. Si los conjuntos first de los terminales no son disyuntos, no se puede aplicar la técnica porque sería imposible predecir cuál regla aplicar (habría más de una).
3. Por cada NT escribir el *conjunto follow*: Dado un NT, el conjunto follow representa los posibles tokens que deberían aparecer en la cadena una vez que el NT se haya reducido. Es el conjunto de los terminales que pueden aparecer en cualquier cadena válida luego de un no terminal dado. En los conjuntos follow no puede haber lambda.
4. Por cada regla escribir el *conjunto select*: Idem al conjunto first pero sin lambdas.
5. Si los conjuntos selects son disyuntos para cada NT, entonces la gramática es

Cada elemento de en los conjuntos es una secuencia de tokens de largo .

Reglas Conjunto First (Fi)

1.

2a. Si

2b. Si

3. Si S reemplazar en por

4. Si lamba

Reglas Conjunto Follow (Fo)

1. Si es distiguido

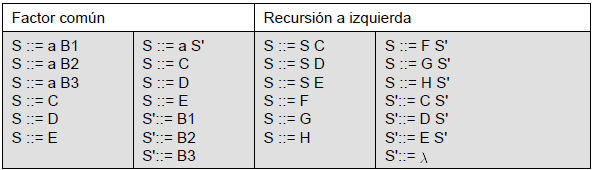
2. Si Si

3. Si

Conjunto Select (Se)

Se calculan reemplazando en los conjuntos las lambdas por los del NT.

**Problemas comunes**

Para solucionar la recursión a izquierda, nos quedamos sólo con las reglas que no tienen recursión y le agregamos un NT’. Este NT’ será la recursión a derecha de las reglas que tenían recursión a izquierda, y se agrega una regla para que este NT’ derive en lambda.

**Implementación recursiva**

1. Por cada NT hacer una rutina donde:
   1. Por cada regla del NT hacer un if para que de acuerdo al lookahead elija que regla usar.
   2. Por cada regla hacer llamadas para los NT e if + consumo de tokens para los T
2. El programa principal consume un token, llama a la rutina del distinguido y si el lookahead es $ la cadena pertenece al lenguaje.

Ventajas del LL

* Asegura una derivación left most determinística.
* Si una gramática es LL no hay ambigüedad.
* Tiempo lineal según la longitud de la cadena.
* Es relativamente fácil de hacer.

Desventajas del LL

* Abarca un pequeño conjunto de gramáticas.
* A veces hay que desdibujar la gramática para aplicarlo.

**Técnica LR**

* L -> Left to right. Los tokens se leen de izquierda a derecha
* R -> Right most. Las derivaciones se hacen lo más a la derecha posible.
* Es una técnica ascendente: trata de armar el árbol desde las hojas hacia la raíz.
* LR(k) -> k indica la cantidad de tokens que lee de la entrada

Ventajas

* Se puede usar para reconocer prácticamente cualquier lenguaje de programación
* El método lee la entrada con una sola pasada
* Detecta errores lo más pronto posible

Desventajas

* Es más complicado armar las tablas

Gramática

|  |  |
| --- | --- |
| Pila | Entrada |
| 0 | Id+id\*id$ |
| Oid5 | +id\*id$ |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

En cada burbuja se agregan los follows de cada regla. En los kernels sólo se copian los de la regla de la burbuja anterior, no se calculan. Los se calculan al hacer la clausura de esa nueva burbuja.

Técnica SLR

1. Agrega una regla falsa E’ -> E
2. Crea una serie de estados: formado por un kernel o núcleo de estado y una clausura.
3. Una vez que se tiene un estado, hay que generar las transiciones desde el mismo a otros estados.

Para llenar la tabla, vamos estado por estado mirando las transiciones.

Para los desplazamientos: si es un NT, se pone el estado al que va. Si es un T, se anota el estado con una D adelante.

Para las reducciones: tendremos reducciones cuando se haya terminado de parsear una producción entera (cuando el . esté al final, solamente en el kernel). Se pone R y el número de regla por la que se debe reducir. Para saber en qué celda se deben colocar las reducciones, se deben calcular los follows.

Reducción por cero = cadena OK!.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Estados | id | + | \* | ( | ) | $ |  | E | T | F |
| 0 | D5 |  |  | D4 |  |  |  | 1 | 2 | 3 |
| 1 |  | D6 |  |  |  | ✔ (o R0) |  |  |  |  |
| 2 |  | R2 | D7 |  | R2 | R2 |  |  |  |  |
| 3 |  | R4 | R4 |  | R4 | R4 |  |  |  |  |
| 4 | D5 |  |  | D4 |  |  |  | 8 | 2 | 3 |
| 5 |  | R6 | R6 |  | R6 | R6 |  |  |  |  |
| 6 | D5 |  |  | D4 |  |  |  |  | 9 | 3 |
| 7 | D5 |  |  | D4 |  |  |  |  |  | 10 |
| 8 |  | D6 |  |  | D11 |  |  |  |  |  |
| 9 |  | R1 | D7 |  | R1 | R1 |  |  |  |  |
| 10 |  | R3 | R3 |  | R3 | R3 |  |  |  |  |
| 11 |  | R5 | R5 |  | R5 | R5 |  |  |  |  |

Gramáticas que son capaces de reconocer: LR contiene a LALR y éste contiene a SLR.

Cantidad de estados:

**LALR**

Busca los kernels iguales que ‘se pueden’ unir y trata de unirlos.

Si en la técnica LR no existen conflictos, al hacer LALR, pueden aparecer conflictos. Pero estos conflictos sólo pueden ser reducción/reducción. Si aparece un conflicto reducción/desplazamiento, está mal.