

# Lenguajes y Automatas II

**Nombre del equipo:** INTERNET EXPLORER 95

**Nombre de integrante:** Rodriguez Perez Luis diego

no control: 161080170 correo: [161080170@iztapalapa.tecnm.mx](mailto:161080170@iztapalapa.tecnm.mx)

**nombre de integrante:** Ramírez Peña Carlos Iván

no control: 161080215 correo: [161080215@iztapalapa.tecnm.mx](mailto:161080215@iztapalapa.tecnm.mx)

**nombre de integrante:** Hernández Ruiz Adrián Felipe

no control: 171080105 correo: [171080105@iztapalapa.tecnm.mx](mailto:171080105@iztapalapa.tecnm.mx)

**nombre de integrante:** Alvarez Rosales Alejandro

no control: 151080010 correo: [151080010@iztapalapa.tecnm.mx](mailto:151080010@iztapalapa.tecnm.mx)

**Grupo:** 7 AM

**NOMBRE DE LA ESCUELA :** Instituto Tecnológico de iztapalapa

**Carrera:** Ing en Sistemas computacionales

**Nombre de la materia:** Lenguajes y Automatas 2

**Nombre del Profesor:** Parra Hernandez Abiel

## Tema: Videos y lectura de los libros



### Actividades semana 4 (Oct 12-16, 2020)

"Introducción a Compiladores" y "Análisis Léxico (Parte 1)"

VIDEO 1

nosotros vamos a hacer la asignación de un programa para hacer más bien nuestro analizador léxico

que sea ejecutable para que nuestro compilador eso no tiene un software las arquitecturas por que son importantes esta construido por una cpu donde se va creando nuestro analizador léxico en nuestra área

para que nuestro analizador necesitamos más nuestro programa para poder crear nuestro simulador de nuestro programa más bien diseñar el compilador para que pueda más bien un mensaje a nuestro cpu

los tipos de algoritmos lo utilizamos más bien en el programa para que pueda analizar nuestro análisis del algoritmo más bien que él lo que está haciendo dentro de nuestra cpu

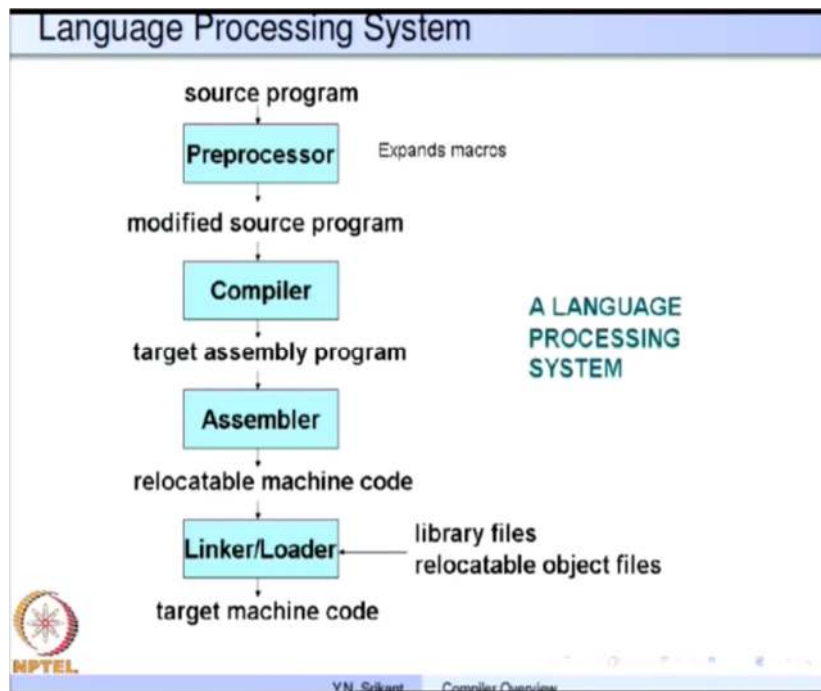
lenguaje de procesamiento del sistema

vemos que como va nuestro proceso de nuestro programa como vemos cómo va mas bien nuestro programa

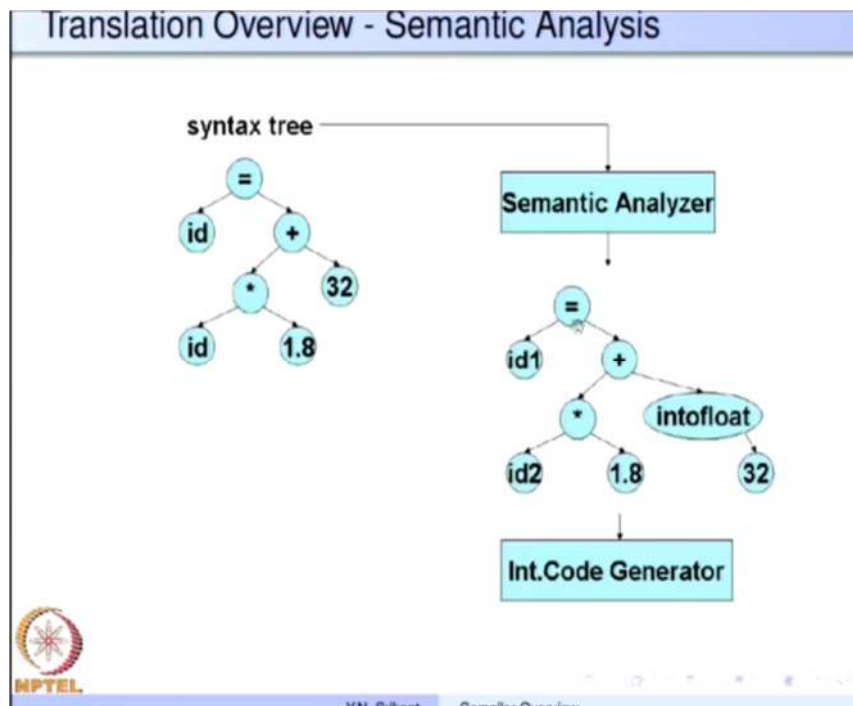
por que vemos como va procesando dentro nuestro analisis lexico y sintactico como va procesando

para que veamos y no vaya ver ningún problema de cómo va entrando/ y saliendo nuestro programa

hay varios programas que se puede utilizar son java,python



aquí vemos cómo se va transformando cuando nosotros ya estamos recopilando aquí vemos que nuestro código fuente son llamados tokens que nosotros vamos a programar por ejemplo nosotros vamos hacer una (suma o multiplicación o cualquiera) nosotros lo vamos a transformar en nuestro código fuente que vamos a poner nuestras constantes y operaciones vamos a ver como pasa nuestro código más bien como lo va transformando en análisis sintáctico por ejemplo cómo va transformando nuestro código fuente pasar por el análisis léxico que son nuestros tokens y se pasa a nuestro análisis sintáctico



## VIDEO 2

sobre el análisis léxico

por qué se debería separar el analista léxico de sintaxis que son los tokens  
 requiere expresión regulares y definiciones regulares y vamos a estudiar todo esto por que  
 es muy importante ver como se va separando del análisis léxico de la sintaxis es una  
 herramienta muy efectiva en generación de analizador léxico para que podamos hacer el  
 primer componente de un compilador

que es el analizador léxico

es un lenguaje de alto nivel en varios lenguajes de programación que son java,c++,python  
 todos estos análisis son más bien controladores que vamos programando en nuestro  
 lenguaje para que mande mensaje de a lo que vamos a querer dentro de nuestro programa  
 y podemos reducir la complejidad de construir nuestro compilador los problemas de e/s  
 tiene un límite al análisis léxico solo los errores que tendríamos en nuestro programa léxico  
 .el análisis léxico se basa más en los autómatas que está finito

que es un patrón

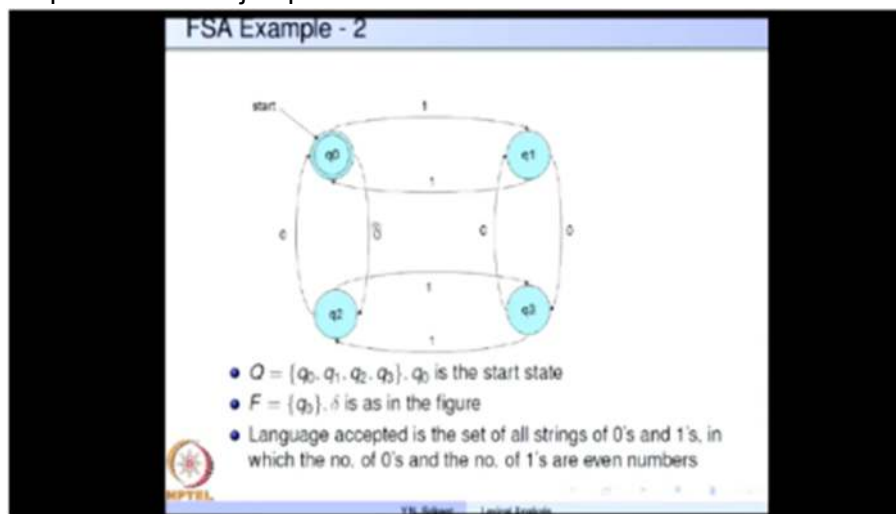
conjunto de cadenas para producir el mismo token se dice que el patrón coincide con cada  
 cadena que se supone que debe coincidir float es un patrón en sí mismo por que ninguna  
 otra cadena coincide con este patrón particular es el que dice que el patrón es el que dice el  
 patrón de la segunda letra más bien para que vaya subrayando en código fuente más bien  
 mientras se va ejecutando todo el proceso de nuestro código fuente

los espacios en blanco no son fotones significativos que pueden aparecer en medio de  
 identificar puede pero en ca c++ pascal etc no están los espacios errores importantes,

excepto los muy simples, como los símbolos ilegales etc y el resto de los errores son detectados por la parcela

los diagramas de transición son más que variables de autómatas de estado finito se utiliza para implementar y traducidos en un lenguaje de programación puede ser en cualquiera en java o en c++ pero tenemos que ver cómo podemos más bien a transformarlo en código para poder programarlo son los tokens que se va a transformar en nuestro código fuente

que son máquinas automáticas y que hacen más bien las máquinas automáticas aceptan lenguajes libres de contexto y puede ser especificado usando gramáticas libres de contexto autómatas lineales acostado aceptar contexto lenguajes sensibles autómatas de estado finito se utiliza para análisis léxico regular y los autómatas se utiliza para analizar los tokens o nuestro código fuente en sí aquí vemos un ejemplo de un autómata



aquí más bien el autómata tiene un inicio y un fin donde más bien vamos a buscar un camino más corto para llegar a nuestro fin pero también hay caminos largo donde podemos llegar al fin pero también hay caminos que no dejan llegar a nuestro destino que es el fin también hay flechas en curva en nuestra autómata que nos indica que volvamos a repetir el mismo camino de nuestro camino

lectura de los libro  
compiladores

los lenguajes de programación son anotaciones que describen los cálculos a las personas y las máquinas aquí en esta más bien como funciona las interfaces dentro de una cpu

## 1.1 compilador

aquí es donde como va procesando nuestro compilador dentro de una entrada y una salida para que mande mensaje a nuestro cpu pueden definirse en cualquier programa para crear nuestro programa en cualquier lenguaje como es java donde nuestro compilador ahora que veamos cómo se va ejecutando

## 1.2 la estructura de un compilador

Un compilador identifica los significados de las diferentes construcciones presentes en la definición del propio lenguaje. En un compilador pueden distinguirse dos fases principales: una fase de análisis, en la que la estructura y el significado del código fuente se analiza; y otra fase de síntesis, en la que se genera el programa objeto.

### 1.2.1 analisis lexico

Es la etapa en la que se realiza un análisis a nivel de caracteres. El objetivo de esta fase es reconocer los componentes léxicos presentes en el código fuente, enviándoles después, junto con sus atributos, al analizador sintáctico.

### 1.2.2 análisis sintáctico

Un analizador sintáctico toma los tokens que le envíe el analizador léxico y creará un árbol sintáctico que refleje la estructura del programa fuente.

En esta fase se comprobará si con dichos tokens se puede formar alguna sentencia válida dentro del lenguaje.

La sintaxis de la mayoría de los lenguajes de programación se define habitualmente por medio de gramáticas libres de contexto. El término libre de contexto se refiere al hecho de que un no terminal puede siempre ser sustituido sin tener en cuenta el contexto en el que aparece.

### 1.2.3 análisis semántico

La semántica se encarga de describir el significado de los símbolos, palabras y frases de un lenguaje, ya sea un lenguaje natural o de programación. Hay que dotar de significado a lo que se ha realizado en la fase anterior de análisis sintáctico.

También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.

### 1.2.4 Generación de código intermedio.

En un modelo en el que se realice una separación de fases en análisis y síntesis dentro de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual se genera después el código objeto.

Entre las ventajas de usar un código intermedio destaca el hecho de que se pueda crear un compilador para una arquitectura distinta sin tocar el front-end de un compilador ya existente para otra arquitectura.

### 1.2.5 optimizacion de codigo

La segunda etapa del proceso de síntesis trata de optimizar el código intermedio, para posteriormente generar código máquina más rápido de ejecutar.

Unos de los tipos de optimización de código más habituales son la eliminación de variables no usadas y el desenrollado de bucles. También resulta muy habitual traducir las expresiones lógicas para que tenga que calcularse simplemente el valor de aquellos operandos necesarios para poder evaluar la expresión (evaluación en corto circuito)

#### 1.2.6 Generación de código

La fase final de un compilador es la generación de código objeto. Cada una de las instrucciones presentes en el código intermedio se debe traducir a una secuencia de instrucciones máquina, donde un aspecto decisivo es la asignación de variables a registros físicos del procesador.

#### 1.2.7 administración de tabla de símbolos

Una función esencial de un compilador es registrar los nombres de las variables que se utilizan en el programa fuente, y recolectar información sobre varios atributos de cada nombre. Estos atributos pueden proporcionar información acerca del espacio de almacenamiento que se asigna para un nombre, su tipo, su alcance (en qué parte del programa puede usarse su valor)

#### 1.2.8 El agrupamiento de fases en pasadas

Las fases del proceso de compilación se agrupan en una etapa inicial y una etapa final.

\*Etapa inicial

Comprende aquellas fases que dependen principalmente del lenguaje fuente y son en gran parte independiente de la máquina objeto.

#### 11.2.9 Herramientas de construcción de compiladores

Al igual que cualquier desarrollador de software, el desarrollador de compiladores puede utilizar para su beneficio los entornos de desarrollo de software modernos que contienen herramientas como editores de lenguaje, depuradores, administradores de versiones, profilers, ambientes seguros de prueba, etc. Algunas herramientas de construcción de compiladores de uso común son:

Generadores de analizadores sintácticos

Generadores de escáneres

Motores de traducción orientados a la sintaxis

Generadores de generadores de código

Motores de análisis de flujos de datos

Kits

### 1.3 La evolución de los lenguajes de programación

Las primeras computadoras electrónicas aparecieron en la década de 1940 y se programaban en lenguaje máquina, mediante secuencias de 0's y 1's que indican de manera explícita a la computadora las operaciones que debía ejecutar, y en qué orden

### 1.3.1 El avance a los lenguajes de alto nivel

El primer paso hacia los lenguajes de programación más amigables para las personas fue el desarrollo de los lenguajes ensambladores a inicios de la década de 1950, los cuales usaban mnemónicos. Al principio, las instrucciones en un lenguaje ensamblador eran sólo representaciones mnemónicas de las instrucciones de máquina.

### 1.3.2 Impactos en el compilador

Los compiladores pueden ayudar a promover el uso de lenguajes de alto nivel, al minimizar la sobrecarga de ejecución de los programas escritos en estos lenguajes. Los compiladores también son imprescindibles a la hora de hacer efectivas las arquitecturas computacionales de alto rendimiento en las aplicaciones de usuario. Desde su diseño, los lenguajes de programación y los compiladores están íntimamente relacionados; los avances en los lenguajes de programación impusieron nuevas demandas sobre los escritores de compiladores.

## 1.4 La ciencia de construir un compilador

Un compilador debe aceptar todos los programas fuente conforme a la especificación del lenguaje; el conjunto de programas fuente es infinito y cualquier programa puede ser muy largo, posiblemente formado por millones de líneas de código. El diseño de compiladores está lleno de bellos ejemplos, en donde se resuelven problemas complicados del mundo real mediante la abstracción de la esencia del problema en forma matemática. Éstos sirven como excelentes ilustraciones de cómo pueden usarse las abstracciones para resolver problemas: se toma un problema, se formula una abstracción matemática que capture las características clave y se resuelve utilizando técnicas matemáticas.

### 1.4.1 Modelado en el diseño e implementación de compiladores

Algunos de los modelos más básicos son las máquinas de estados finitos y las expresiones regulares. Estos modelos son útiles para describir las unidades de léxico de los programas (palabras clave, identificadores y demás) y para describir los algoritmos que utiliza el compilador para reconocer esas unidades.

El estudio de los compiladores es principalmente un estudio de la forma en que diseñamos los modelos matemáticos apropiados y elegimos los algoritmos correctos, al tiempo que logramos equilibrar la necesidad de una generalidad y poder con la simpleza y la eficiencia.

### 1.4.2 La ciencia de la optimización de código

La optimización de código es el conjunto de fases de un compilador que transforma un fragmento de código en otro fragmento con un comportamiento equivalente y que se ejecuta de forma más eficiente, es decir, usando menos recursos de cálculo como memoria o tiempo de ejecución.

Las optimizaciones de compiladores deben cumplir con los siguientes objetivos de diseño:



- La optimización debe ser correcta; es decir, debe preservar el significado del programa compilado.
- La optimización debe mejorar el rendimiento de muchos programas.
- El tiempo de compilación debe mantenerse en un valor razonable.
- El esfuerzo de ingeniería requerido debe ser administrable.

## 1.5 Aplicaciones de la tecnología de compiladores

El diseño de compiladores no es sólo acerca de los compiladores; muchas personas utilizan la tecnología que aprenden al estudiar compiladores en la escuela y nunca, hablando en sentido estricto, han escrito (ni siquiera parte de) un compilador para un lenguaje de programación importante.

### 1.5.1 Implementación de lenguajes de programación de alto nivel

La implementación del diseño de los sistemas orientados a objetos es más fácil usando lenguajes de programación orientados a objetos. En general, no todos los lenguajes de programación

implementan de la misma forma los diferentes conceptos de orientación a objetos. Más aún, los diferentes lenguajes varían en el nivel de apoyo a los conceptos básicos.

- Estructuras de datos: definido de forma declarativa
- Flujo dinámico de control: este puede ser procedural (condiciones, ciclos, llamadas) o declarativo (reglas, restricciones, tablas). No existe apoyo a concurrencia en la mayoría de los lenguajes, excepto Ada. La concurrencia puede ser simulada usando corutinas, modelos de control, o manejadores de eventos. Multi-procesamiento y comunicación entre procesos son provistos por el sistema operativo.
- Transformaciones funcionales: son expresadas por medio de operadores primitivos y subrutinas.

### 1.5.2 Optimizaciones para las arquitecturas de computadoras

La rápida evolución de las arquitecturas de computadoras también nos ha llevado a una insaciable demanda de nueva tecnología de compiladores. Casi todos los sistemas de alto rendimiento aprovechan las dos mismas técnicas básicas Paralelismo

Todos los microprocesadores modernos explotan el paralelismo a nivel de instrucción. Sin embargo, este paralelismo puede ocultarse al programador. Los programas se escriben como si todas las instrucciones se ejecutan en secuencia

#### Jerarquías de memoria

Una jerarquía de memoria consiste en varios niveles de almacenamiento con distintas velocidades y tamaños, en donde el nivel más cercano al procesador es el más rápido, pero también el más pequeño

### 1.5.3 Diseño de nuevas arquitecturas de computadoras

Desde que la programación en lenguajes de alto nivel es la norma, el rendimiento de un sistema computacional se determina no sólo por su velocidad en general, sino también por la forma en que los compiladores pueden explotar sus características.

#### RISC

Uno de los mejores ejemplos conocidos sobre cómo los compiladores influenciaron el diseño de la arquitectura de computadoras fue la invención de la arquitectura RISC. Arquitecturas especializadas. En las últimas tres décadas se han propuesto muchos conceptos sobre la arquitectura, entre los cuales se incluyen las máquinas de flujo de datos, las máquinas VLIW4.

### 1.5.4 Traducciones de programas

Mientras que, por lo general, pensamos en la compilación como una traducción de un lenguaje de alto nivel al nivel de máquina, la misma tecnología puede aplicarse para realizar traducciones entre distintos tipos de lenguajes.

A continuación se muestran algunas de las aplicaciones más importantes de las técnicas de traducción de programas.

Traducción binaria

Síntesis de hardware

Intérpretes de consultas de bases de datos

Simulación compilada

### 1.5.5 Herramientas de productividad de software

Un enfoque complementario interesante y prometedor es utilizar el análisis de flujos de datos para localizar errores de manera estática (es decir, antes de que se ejecute el programa).

El análisis de flujos de datos puede buscar errores a lo largo de todas las rutas posibles de ejecución. Existen herramientas de productividad de software.

Comprobación (verificación) de tipos

Comprobación de límites

Herramientas de administración de memoria

## 1.6 Fundamentos de los lenguajes de programación

En esta sección hablaremos sobre la terminología más importante y las distinciones que aparecen en el estudio de los lenguajes de programación. No es nuestro objetivo abarcar todos los conceptos o todos los lenguajes de programación populares. Asumimos que el lector está familiarizado por lo menos con uno de los lenguajes C, C++, C# o Java, y que tal vez conozca otros.

### 1.6.1 La distinción entre estático y dinámico

Que la estática es un capítulo más de la física es aceptado con total naturalidad desde que Newton estableció las bases de la mecánica clásica. Pero estas dos disciplinas, estática y

dinámica, tienen historias diferentes, con encuentros y desencuentros, desde la Antigüedad hasta el Renacimiento.

En este artículo, se describe parte de este proceso en relación a algunos aspectos de la mecánica, en particular de la estática. Cómo esta disciplina quedó establecida, ya en la Antigüedad, en forma rigurosa y matemática, mientras que la dinámica se enfrentó con dificultades conceptuales y empíricas que comenzaron a esclarecer recién en el Renacimiento.

### 1.6.2 Entornos y estados

Otra distinción importante que debemos hacer al hablar sobre los lenguajes de programación es si los cambios que ocurren a medida que el programa se ejecuta afectan a los valores de los elementos de datos, o si afectan a la interpretación de los nombres para esos datos.

### 1.6.3 Alcance estático y estructura de bloques

La mayoría de los lenguajes, incluyendo a C y su familia, utilizan el alcance estático. Las reglas de alcance para C se basan en la estructura del programa; el alcance de una declaración se determina en forma implícita, mediante el lugar en el que aparece la declaración en el programa. Los lenguajes posteriores, como C++, Java y C#, también proporcionan un control explícito sobre los alcances, a través del uso de palabras clave como `public`, `private` y `protected`.

### 1.6.4 Control de acceso explícito

Mediante el uso de palabras clave como `public`, `private` y `protected`, los lenguajes orientados a objetos como C++ o Java proporcionan un control explícito sobre el acceso a los nombres de los miembros en una superclase. Las clases y las estructuras introducen un nuevo alcance para sus miembros. Si `p` es un objeto de una clase con un campo (miembro) `x`, entonces el uso de `x` en `p.x` se refiere al campo `x` en la definición de la clase.

### 1.6.5 Alcance dinámico

Técnicamente, cualquier directiva de alcance es dinámica si se basa en un factor o factores que puedan conocerse sólo cuando se ejecute el programa. Sin embargo, el término alcance dinámico se refiere, por lo general, a la siguiente directiva: el uso de un nombre `x` se refiere a la declaración de `x` en el procedimiento que se haya llamado más recientemente con dicha declaración.

De hecho, para poder interpretar a `x`, debemos usar la regla de alcance dinámico ordinaria. Examinamos todas las llamadas a funciones que se encuentran activas, y tomamos la función que se haya llamado más recientemente y que tenga una declaración de `x`. A esta declaración esa la que se refiere `x`.

### 1.6.6 Mecanismos para el paso de parámetros

Todos los lenguajes de programación tienen una noción de un procedimiento, pero pueden diferir en cuanto a la forma en que estos procedimientos reciben sus argumentos. En esta sección vamos a considerar cómo se asocian los parámetros actuales. El mecanismo que se utilice será el que determine la forma en que el código de secuencia de llamadas tratará a los parámetros.

#### Llamada por valor

En la llamada por valor, el parámetro actual se evalúa o se copia. El valor se coloca en la ubicación que pertenece al correspondiente parámetro formal del procedimiento al que se llamó.

#### Llamada por referencia

En la llamada por referencia, la dirección del parámetro actual se pasa al procedimiento al que se llamó como el valor del correspondiente parámetro formal.

#### Llamada por nombre

Hay un tercer mecanismo que se utilizó en uno de los primeros lenguajes de programación: Algol 60.

#### 1.6.7 Uso de alias

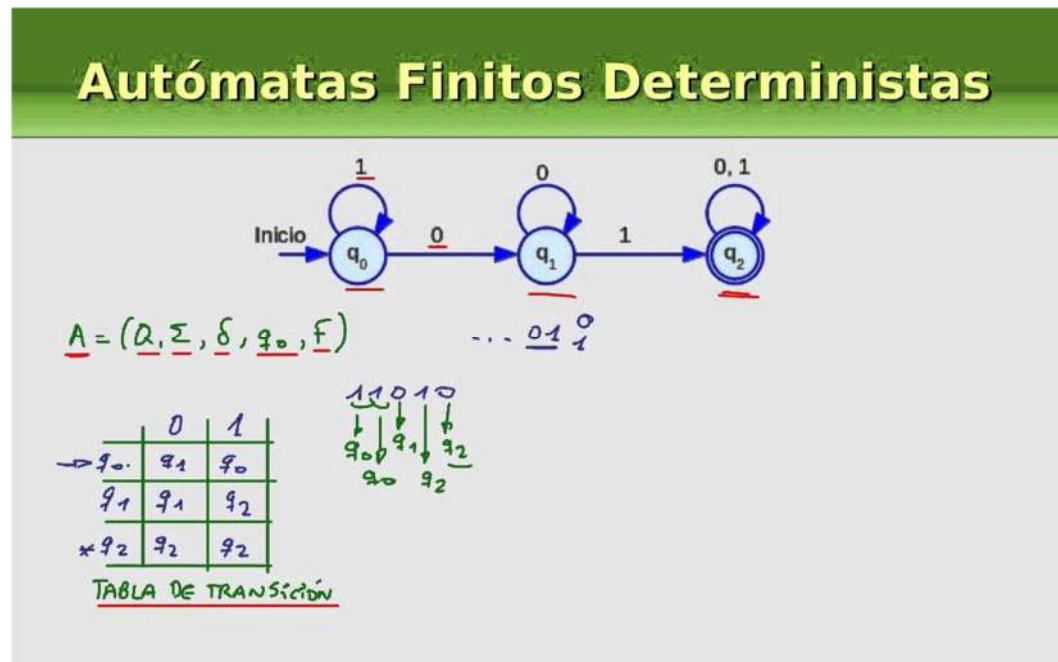
Hay una consecuencia interesante del paso por parámetros tipo llamada por referencia o de su simulación, como en Java, en donde las referencias a los objetos se pasan por valor. Es posible que dos parámetros formales puedan referirse a la misma ubicación. Resulta que es esencial comprender el uso de los alias y los mecanismos que los crean si un compilador va a optimizar un programa.

## Actividades semana 5 (Oct 19-23, 2020)

### Mod-02 Lec-03 Lexical Analysis - Part 2 IVAN VIDEO 1

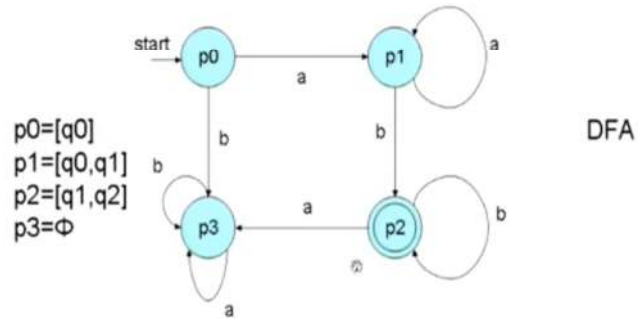
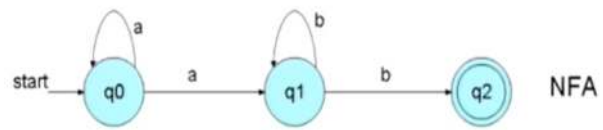
#### autómata determinista

un autómata determinista tiene una transición en cada símbolo alfabético la tabla de transiciones el cual describe los autómatas



En esta imagen tenemos un ejemplo de un autómata determinista y no determinista. en la primer autómata tenemos el no determinista a donde tiene dos transiciones con el mismo valor en el segundo solo tiene uno

## An NFA and an Equivalent DFA



Let  $\Sigma$  be an alphabet. The REs over  $\Sigma$  and the languages they denote (or generate) are defined as below

- 1  $\phi$  is an RE.  $L(\phi) = \phi$
- 2  $\epsilon$  is an RE.  $L(\epsilon) = \{\epsilon\}$
- 3 For each  $a \in \Sigma$ ,  $a$  is an RE.  $L(a) = \{a\}$
- 4 If  $r$  and  $s$  are REs denoting the languages  $R$  and  $S$ , respectively
  - $(rs)$  is an RE,  $L(rs) = R.S = \{xy \mid x \in R \wedge y \in S\}$
  - $(r + s)$  is an RE,  $L(r + s) = R \cup S$
  - $(r^*)$  is an RE,  $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$   
 ( $L^*$  is called the *Kleene closure* or *closure* of  $L$ )

lenguaje aceptado para o reconocido para DFA para tener una representación finita

En este texto vamos a ver uno de los métodos que se usan para transformar autómatas finitos deterministas en expresiones regulares, el método de eliminación de estados.

Cuando tenemos un autómata finito, determinista o no determinista, podemos considerar que los símbolos que componen a sus transiciones son expresiones regulares. Cuando eliminamos un estado, tenemos que reemplazar todos los caminos que pasaban a través de él como transiciones

directas que ahora se realizan con el ingreso de expresiones regulares, en vez de con símbolos.

Crear una FSA es muy parecido a escribir un programa. Es decir, es un proceso creativo sin una "receta" simple que se pueda seguir para llevarte siempre a un diseño correcto. Dicho esto, muchas de las habilidades que aplique a la programación también funcionarán al crear FSA.

Un NFA puede tener cero, uno o más de un movimiento de un estado dado en un símbolo de entrada dado. Un NFA también puede tener movimientos NULL (movimientos sin símbolo de entrada). Por otro lado, DFA tiene un solo movimiento desde un estado dado en un símbolo de entrada dado.

### **Conversión de NFA a DFA**

Suponga que hay un NFA  $N = \langle Q, \Sigma, q_0, \delta, F \rangle$  que reconoce un lenguaje  $L$ . Entonces el DFA  $D = \langle Q', \Sigma, q_0, \delta', F' \rangle$  se puede construir para idioma  $L$  como:

Paso 1: Inicialmente  $Q' = \emptyset$ .

Paso 2: agregue  $q_0$  a  $Q'$ .

Paso 3: Para cada estado en  $Q'$ , encuentre el posible conjunto de estados para cada símbolo de entrada usando la función de transición de NFA. Si este conjunto de estados no está en  $Q'$ , agréguese a  $Q'$ .

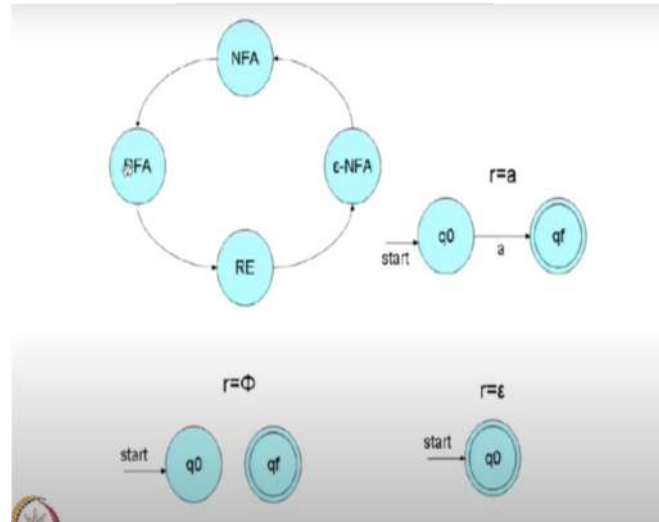
Paso 4: El estado final de DFA serán todos los estados que contengan  $F$  (estados finales de NFA)

### **Equivalencia de FSA**

El algoritmo de llenado de tablanos proporciona una forma fácil de comprobar si dos lenguajes regulares son el mismo. Supongamos que tenemos los lenguajes  $L$  y  $M$ , cada uno de ellos representado de una manera, por ejemplo, uno mediante una expresión regular y el otro mediante un AFN. Convertimos cada una de las representaciones a un AFD. Ahora, imaginemos un AFD cuyos estados sean la unión de los estados de los AFD correspondientes a  $L$  y  $M$ . Técnicamente, este AFD tendrá dos estados iniciales, pero realmente el estado inicial es irrelevante para la cuestión de comprobar la equivalencia de estados, por lo que consideraremos uno de ellos como único estado inicial



Ahora se comprueba si los estados iniciales de los dos AFD originales son equivalentes, utilizando el algoritmo de llenado de tabla. Si son equivalentes, entonces  $L=M$ , y si no lo son, entonces  $L \neq M$ .



Representación del analizador léxico basándose en los anteriores autómatas.

```

Lexical Analyzer Implementation from Trans. Diagrams

TOKEN gettoken() {
    TOKEN mytoken; char c;
    while(1) { switch (state) {
        /* recognize reserved words and identifiers */
        case 0: c = nextchar(); if (letter(c))
            state = 1; else state = failure();
            break;
        case 1: c = nextchar();
            if (letter(c) || digit(c))
                state = 1; else state = 2; break;
        case 2: retract(1);
            mytoken.token = search_token();
            if (mytoken.token == IDENTIFIER)
                mytoken.value = get_id_string();
            return(mytoken);
    }
}

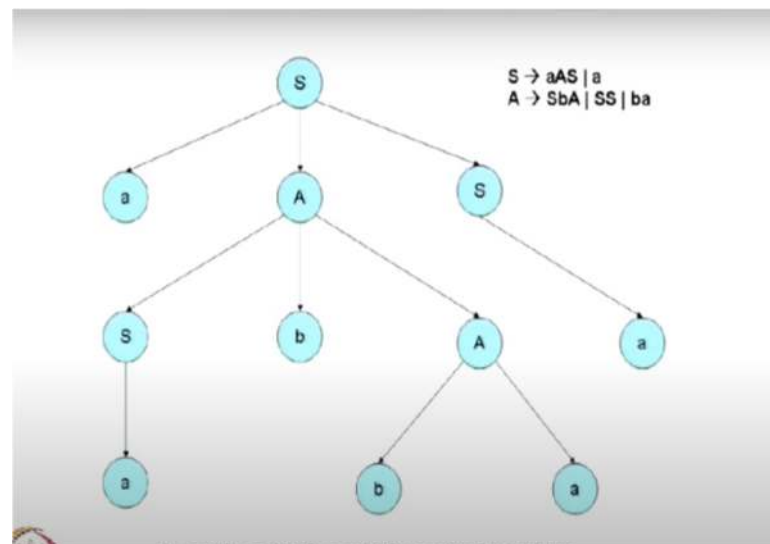
```

### Mod-03 Lec-05 Syntax Analysis VIDEO 3

gramáticas libres de contexto que son la base para la especificación de la programación idiomas que se analizan sin contexto, se describen con precisión y se utiliza una gramática para describir la sintaxis de los lenguajes programación se describen la estructura sintáctica para su ejecución

Las gramáticas se usan generalmente para la especificación de sintaxis de los lenguajes de programación. Durante el análisis lexico idiomas regulares, lenguajes sensibles al contexto en lenguajes de tipo 0. Utilizados para especificar lenguajes libres de contexto y estos son los más utilizados para un análisis sintáctico.

## Gramática de contexto



- Las Gramáticas Libres de Contexto ( Context-Free Languages) o CFL's jugaron un papel central en lenguaje natural desde los 50's y en los compiladores desde los 60's
- Las Gramáticas Libres de Contexto forman la base de la sintaxis BNF
- Son actualmente importantes para XML y sus DTD's (document type definition)

Vamos a ver los CFG's, los lenguajes que generan, los árboles de parseo, el pushdown automata y las propiedades de cerradura de los CFL's.

- Ejemplo: Considere  $L_{pal} = \{w \in \Sigma^* : w = w^R\}$ . Por ejemplo,  $oso \in L_{pal}$ ,  $anitalavalatina \in L_{pal}$ ,
- Sea  $\Sigma = \{0, 1\}$  y supongamos que  $L_{pal}$  es regular.
- Sea dada por el pumping lemma. Entonces  $0^n 10 \in L_{pal}$ . Al leer  $0^n$  el FA debe de entrar a un ciclo.

## Ambigüedad. Árboles de derivación. Gramáticas ambiguas

Def (árbol de derivación) Dada una GI  $(\Sigma, V, I, P)$ , un árbol de derivación tiene las siguientes características:

1. Cada nodo interno está etiquetado con una variable  $\in V$ .
2. Cada hoja está etiquetada con una variable  $\in V$ , con un terminal  $\in \Sigma$  ó con  $\lambda$ .
3. Si un nodo está etiquetado por  $A$ , y sus hijos por  $x_1, x_2, \dots, x_n$ , entonces debe existir una regla en P donde  $A \rightarrow x_1, x_2, \dots, x_n$ .

Ejemplo

Árbol de derivación para  $a * 0 + b$

$$\Sigma = \{a, b, 0, 1, +, *, -, /, (, )\}$$

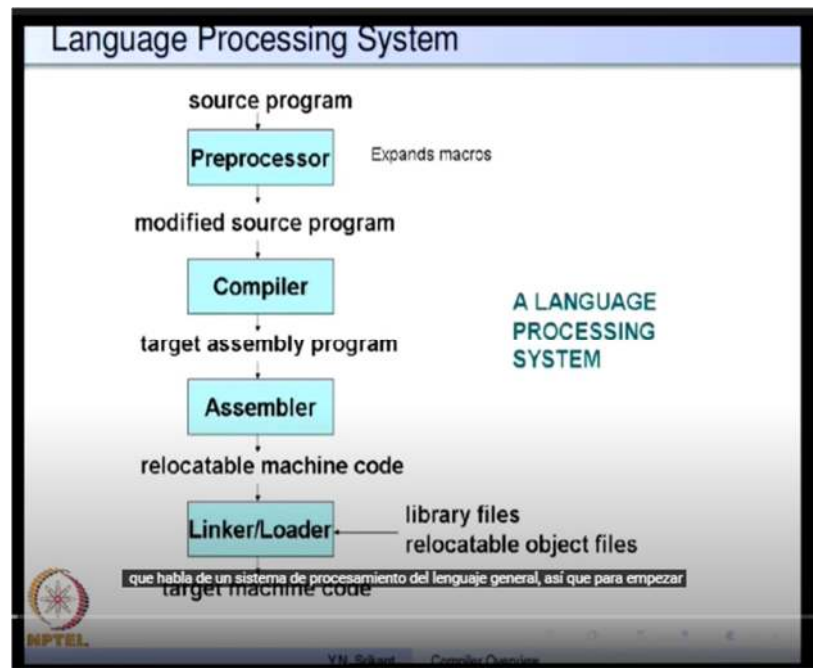
$$E \rightarrow E + E \mid E * E \mid E - E \mid E / E \mid (E) \mid I \mid N$$

$$N \rightarrow 0 \mid 1 \mid N0 \mid N1$$

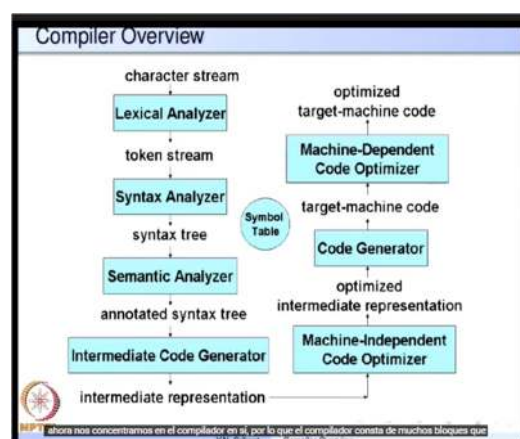
$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

## Apuntes, Alejandro Alvarez Rosales

El video nos invita a analizar la composición de un computador. Para crear un compilador se necesita lenguaje máquina. HTML y JS se basan en compiladores. Se corre el compilador en un CPU simulado para ver su rendimiento y de ser necesario se hacen correcciones. Como no simula el diseño si no las funciones en sí es una simulación mucho más rápida. Un compilador es muy complejo por que usa muchas técnicas y recursos tecnológicos: Álgebra lineal y otras teorías matemáticas, teoría de la probabilidad. Se usan muchos algoritmos como el de ubicación de autómatas finitos, técnicas de escaneo de texto.

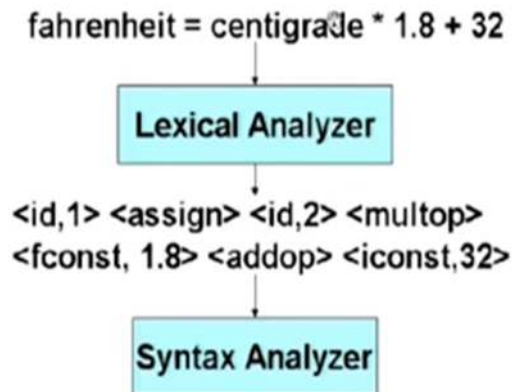


## Fases del compilador



El analizador léxico analiza el alfabeto de cada uno de los caracteres que componen el código. El intérprete es solo un 50 %. El analizador léxico toma el código fuente y lo descompone en tokens y operadores.

## Translation Overview - Lexical Analysis

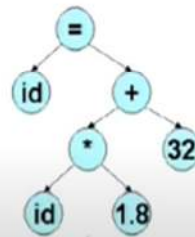


El analizador léxico toma el código fuente y lo descompone en tokens clasificándolos a cada uno con identificadores , operadores y constantes para posteriormente ser analizados por el analizador Semántico

## Translation Overview - Syntax Analysis

<id,1> <assign> <id,2> <multop>  
<fconst, 1.8> <addop> <iconst,32>

**Syntax Analyzer**



**Semantic Analyzer**

expresión estas son todas las comprobaciones de sintaxis que puede realizar el analizador de sintaxis

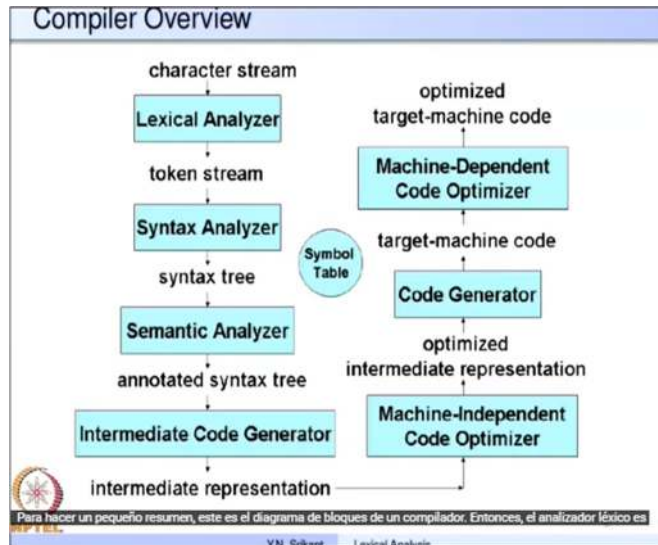


El analizador semántico lo que hace es seleccionar todo en las familias que le corresponden (Alfabetos) y simplifica el código mediante un árbol sintáctico.

[https://docs.google.com/document/d/1h5NLD3YNCek\\_4SZ62fgXHsKweIPt02XYSSwZdy0uWGY/edit](https://docs.google.com/document/d/1h5NLD3YNCek_4SZ62fgXHsKweIPt02XYSSwZdy0uWGY/edit)

Video 2

En este video se habla sobre los analizadores Léxicos.

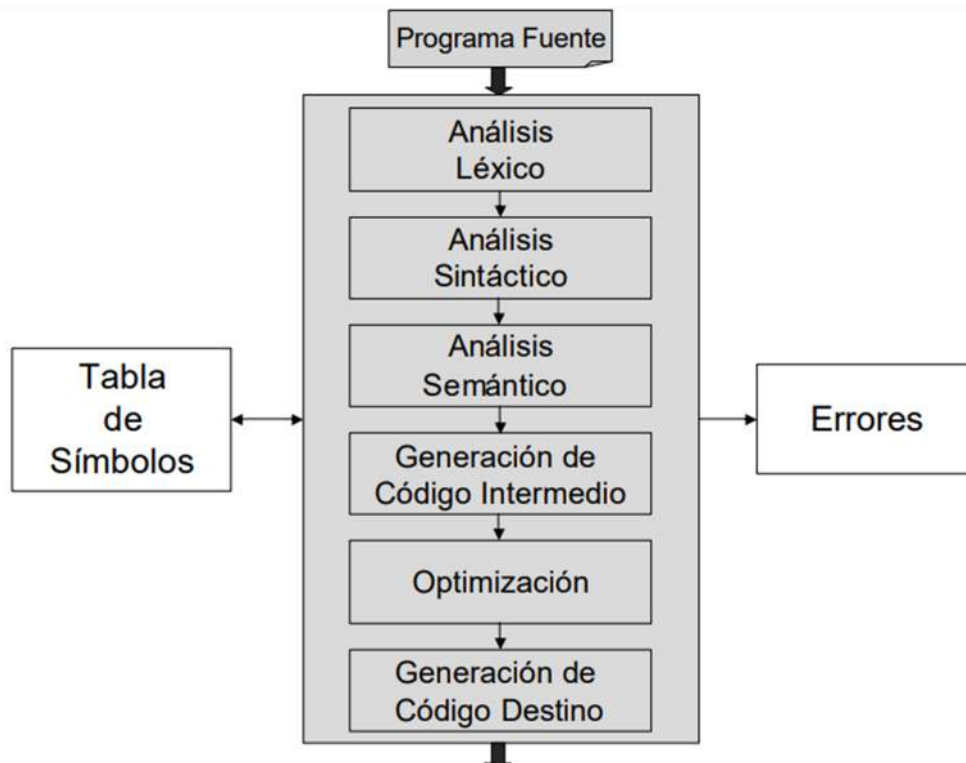


Analizador lexico analiza la lista de tokens , analiza el flujo de tokens linea por linea.

Un **analizador léxico** o **analizador** lexicográfico (en inglés scanner) es la primera fase de un compilador, consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de tokens (componentes léxicos) o símbolos.

El analizador léxico se encarga de desfragmentar esa cadena de caracteres que nosotros conocemos como lenguaje de alto nivel (palabras reservadas) y lo transforma en una cadena de tokens que son más fácilmente interpretados por las siguientes fases del compilador.

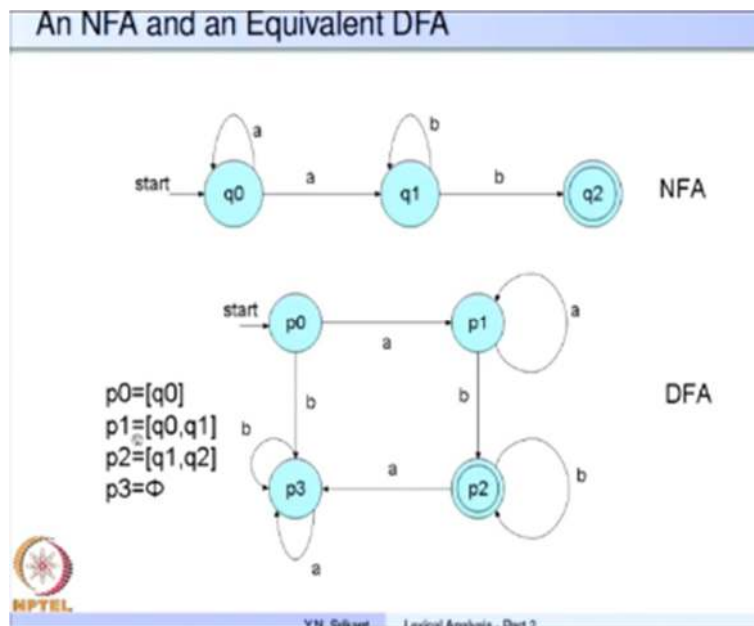
Entre las otras funciones que tiene un analizador léxico están la de eliminar (depurar) los comentarios y espacios en blanco del código fuente, por eso cada lenguaje de programación tiene su manera de identificar sus comentarios ya sea `//` o `<!-- -->` `/**/` y la lista sigue, estas prácticas a parte de no entorpecer nuestro código nos sirven para indicarle al compilador que no se trata de código fuente si no de comentarios los cuales no vale la pena transformar en Tokens, todo lo demás que no sean comentarios o espacios en blanco serán transformados , como por ejemplo paréntesis , comas , operadores y demás signos de puntuación.



Para nosotros es relativamente fácil comprender una serie de símbolos que llamaremos letras, palabras, notas, operadores etc, sin embargo para nuestro compilador es necesaria una serie de métodos para que aprenda a relacionar estos alfabetos entre sí de la manera “Correcta”

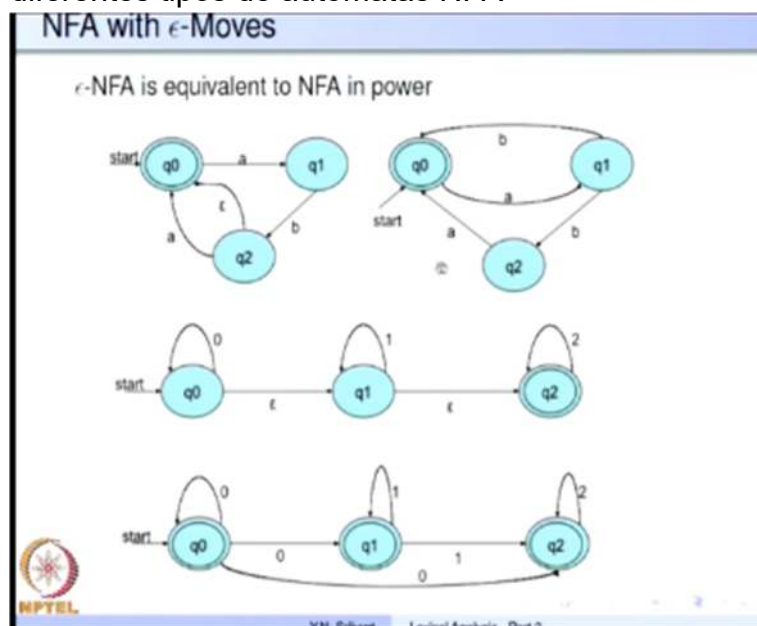
Nombre de integrante:Rodriguez Perez Luis diego  
 analisis lexico parte 2

¿por qué debería separarse el análisis léxico del análisis sintáctico, que es exactamente nuestros tokens,patrones y lexemas y también las dificultades en el análisis léxico y dio una instrucción en estudiar a los automatas infinitos para hacer una instrucción formalmente un autómata determinista tiene exactamente una transición símbolo del alfabeto donde vamos a ver cómo está funcionando nuestro autómata el autómata tiene un inicio y un fin por diferentes caminos pero también hay bucles donde va dar mas bien una vuelta hay diferentes autómatas NFA y DFA



DFA en resumen partiendo del delta de  $q_0$  como  $a$ , es decir, tratamos de averiguar las transiciones del nuevo autómata los nuevos estados del DFA se construye bajo demanda cada subconjunto de los estados NFA es un posible estado DFA, pero no es necesario que todos ser requerido en DFA

diferentes tipos de autómatas NFA



también estudiamos los autómatas del estado finito y analizamos algunas de sus propiedades de conversión de NFA Y DFA que son máquinas, mientras que es posible tener una representación finita de lenguajes regulares en forma de expresión regular conocidas, entonces expresiones regulares son especificaciones y vamos a utilizar expresiones regulares para especificar analizadores léxico es una definición inductiva estas expresiones se combinan



con un plus todas las expresiones regulares es una nueva expresión regular sacando más bien en la tabla de diferentes caminos o posibles caminos en nuestro autómata

los tipos de expresiones regulares

que es una definición regular

es una secuencia de ecuaciones y esto es más que sabes que se usa como una mano corta para escribir especificaciones de analizadores léxicos entonces entendamos cuales son estos.

entonces identifica más en idiomas que son los números enteros. una carta, digamos consideramos sólo cinco letras abcde podemos definirlo como una expresión

regular  $a + b + c + d + e$  que genera el idioma  $a + b + c + d + e$ . 1234 se puede definir como  $0 + 1 + 2 + 3 + 4$  por lo que son demasiados simples

las expresiones regulares extendidas para convertirse en definiciones regulares encabezados por letras y dígitos

que es un identificador

donde las letras seguidas de una letra o un dígito de la estrella entonces combinaciones de letras y dígitos después de una letra y un número sería cualquier número de dígitos así no epsilon por supuesto entonces dígito seguido de cualquier número dígito entonces esto hace que las 2 definiciones identifiquen y el número es un poco más comprensible en lugar de escribir este número como  $0 + 1 + 2 + 3 + 4$  seguido de  $0 + 1 + 2 + 3 + 4$  estrellas entonces sería muy difícil de entender en una definición amplia sin utilizar una abreviatura de estas definiciones regulares

qué pasa con la equivalencia de las expresiones regulares y el estado finito de los autómatas

tenemos un teorema muy fundamental el segundo sea  $r$  una expresión regular entonces existe un autómata de estado finito no determinista con transiciones epsilon que acepta un lenguaje  $L$  de  $r$

entonces este es un teorema muy profundo y esta es la base de construir NFA a partir de especificaciones de definiciones regulares entonces la prueba es por construcción y consideramos la construcción


en detalle lo contrario también es si  $L$  es aceptado por un DFA y luego  $L$  se genera mediante una expresión regular esta prueba es sumamente tediosa y no arroja información sobre el proceso de compilaciones

qué pasa con la generación de un actualizador léxico a partir de diagramas de transición

los diagramas de transiciones mencionados son útiles solo para escribir ya sabes analizadores léxico manualmente entonces veamos cómo se implementa los diagramas de transiciones en código donde devuelve un token esta función get token que devuelve un token y un token en 2 partes puede contener el valor del token, el token y el valor correcto entonces 2 partes en el token en sí y hay un carácter c. entonces aquí este es un ciclo while que continúa para siempre hasta que agotemos la entrada entonces, cambia de estado lee el siguiente carácter este el estado 0 si es una letra ir al estado 1 de lo contrario el estado igual

Lexical Analyzer Implementation from Trans. Diagrams


```
TOKEN gettoken() {
    TOKEN mytoken; char c;
    while(1) { switch (state) {
        /* recognize reserved words and identifiers */
        case 0: c = nextchar(); if (letter(c))
            state = 1; else state = failure();
            break;
        case 1: c = nextchar();
            if (letter(c) || digit(c))
                state = 1; else state = 2; break;
        case 2: retract(1);
            mytoken.token = search_token();
            if (mytoken.token == IDENTIFIER)
                mytoken.value = get_id_string();
            return(mytoken);
    }
```



VN Subant Lexical Analysis - Part 3

Lexical Analyzer Implementation from Trans. Diagrams

```
/* recognize hexa and octal constants */
case 3: c = nextchar();
    if (c == '0') state = 4; break;
    else state = failure();
case 4: c = nextchar();
    if ((c == 'x') || (c == 'X'))
        state = 5; else if (digitoct(c))
            state = 9; else state = failure();
    break;
case 5: c = nextchar(); if (digithex(c))
    state = 6; else state = failure();
    break;
```



VN Subant Lexical Analysis - Part 3

## Lexical Analyzer Implementation from Trans. Diagrams

```
case 6: c = nextchar(); if (digitohex(c))
    state = 6; else if ((c == 'u') ||
    (c == 'U') || (c == 'l') ||
    (c == 'L')) state = 8;
    else state = 7; break;
case 7: retract(1);
/* fall through to case 8, to save coding */
case 8: mytoken.token = INT_CONST;
    mytoken.value = eval_hex_num();
    return(mytoken);
case 9: c = nextchar(); if (digitoct(c))
    state = 9; else if ((c == 'u') ||
    (c == 'U') || (c == 'l') || (c == 'L'))
    state = 11; else state = 10; break;
```



## Lexical Analyzer Implementation from Trans. Diagrams

```
case 10: retract(1);
/* fall through to case 11, to save coding */
case 11: mytoken.token = INT_CONST;
    mytoken.value = eval_oct_num();
    return(mytoken);
```



aquí es donde vamos más bien como vamos transformando nuestro código de como en datos que nos piden en código fuente

análisis léxico parte 3

todo x que genera analizadores léxicos así que para hacer un poco de recapitulación para la transición diagramas o autómatas finitos determinista generalizados pero hay algunas diferencias los borde pueden estar etiquetados por un conjunto de símbolos de símbolos 0 definición algunos estados aceptables pueden indicarse como estado que se retractan y cuando llegamos a un estado de atracción realmente no consumimos el símbolo que nos lleva a ese estado particular y luego con cada estado de aceptación hay una acción que se ejecuta cuando

se alcanza ese estado normalmente usamos esta sección para devolver un token y su valor de atributo pero muy importante luego de que estos diagramas de transición deben combinarse adecuadamente

para hacer un gran diagrama de transición y luego ese diagrama de transición tendrá para ser traducido manualmente a un programa analizador léxico desafortunadamente

hay lenguajes que están enumerados exactamente en ese orden por lo que cuando hay una falla pasa al siguiente diagrama de transición y comienza a mirar ese diagrama en particular así que si esto es seguido sabe que es bastante fácil de programar, sin embargo, esto no utiliza la característica de coincidencia más largas en otras palabras en el XT sería un identificador y una palabra reservada seguido de identificador y no una palabra reservada seguido de identificador ext, por lo que debe no sea mejor pero lo que sucede es si realmente ordena el diagrama de transición el diagrama de transición de palabras reservadas primero, luego sabría que tendría el resultado la palabra luego seguida de identificador ext realmente en realidad es mejor usar la coincidencia más larga de hecho la siguiente debería ser el identificador

cómo obtenemos esta coincidencia más larga

los diagramas de transición debe probarse si sabe que todos deben probarse entonces todas las coincidencias deben registrarse y la coincidencia más larga debe usarse por lo que si esto es hecho o si el programador puede ordenar los diagramas de transición apropiadamente en cualquier caso, la coincidencia más larga se puede usar usando piernas para generar los analizadores léxicos realmente lo hacen fácil para el escritor del compilador.

los analizadores solex es una herramienta disponible en unix solex tiene un lenguaje para describir expresiones regulares que son el núcleo del análisis léxico para que simplemente escriba los patrones que vamos a detectar en el análisis léxico entonces genera un comparador de patrones para las especificaciones de expresiones regulares que son dado a la herramienta lex y una vez esto la herramienta lex genera programas que son apropiadas para el análisis léxico

aquí es donde vemos cómo se va haciendo nuestro código fuente de las instrucciones que estamos pidiendo nos se el color y podemos como poder mandar esa instrucción que color lo podemos programar por medio de números cada número se puede por un color lo que aquí es mandar un mensaje a cpu de lo que es lo que vamos a querer siempre y cuando lo hagamos en código fuente para que podamos mandar ese mensaje

## análisis sintáctico

son las bases para la especificación de lenguajes de programación analizar el idioma sin contexto se basa en autómatas push down al igual que el reconocimiento de lenguaje regular se basa en estado finito autómata estudiaremos dos tipos de análisis : uno es el análisis de arriba hacia abajo, el otro es el análisis de abajo hacia arriba entonces en el análisis de arriba hacia abajo estudiaremos LL(1) y descenso recursivo técnica de análisis sintáctico y en el análisis de abajo hacia arriba estudiaremos las técnicas de análisis de LR Y ahí también es una herramienta llamada yacc que se basa en el análisis LR veremos como ejemplo de como para usarlo para la construcción de analizador sintáctico

## qué es exactamente las gramáticas

Por lo tanto cada lenguaje de programación debe describirse con mucha precisión y se usa una gramática para describir la sintaxis de los lenguajes de programación Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados. En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o notación BNF ( Backus-Naur Form).

Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores.

- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- El proceso de construcción puede llevar a descubrir ambigüedades.
- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

La mayor parte de este tema está dedicada a los métodos de análisis sintáctico de uso típico en compiladores. Primero se introducen los conceptos básicos, después las técnicas adecuadas para la aplicación manual. Además como los programas pueden contener errores sintácticos, los métodos de análisis sintáctico se pueden ampliar para que se recuperen de los errores sintácticos más frecuentes.

S

## ¿Qué es el analizador sintáctico ?

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce. En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

En la práctica, el analizador sintáctico también hace:

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador

semántico).

- Chequeo de tipos ( del analizador semántico).
- Generar código intermedio.
- Generar errores cuando se producen.

En definitiva, realiza casi todas las operaciones de la compilación. Este método de trabajo da lugar a los métodos de compilación dirigidos por sintaxis.

### Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarán mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

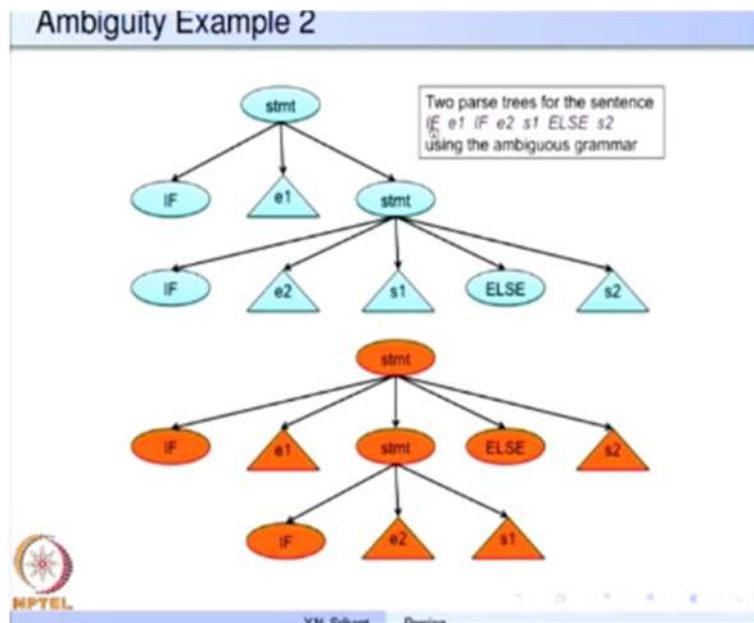
Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores. Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- No ralentizar significativamente la compilación.

vemo como nuestro analizador léxico pasa a semántico porque más bien como se va transformando nuestro código en un diagrama donde va más bien en un diagrama con todos nuestros dato también que pueden guiarse en ese diagrama para ver como va el posible camino ahora si que utilizando solamente nuestros comandos más importantes que necesitamos para poder tener el mensaje que estamos enviando o que resultado estamos pidiendo



## Libro

La fase de análisis de un compilador descompone un programa fuente en piezas componentes y produce una representación interna, a la cual se le conoce como código intermedio. La fase de síntesis traduce el código intermedio en el programa destino.

Por cuestión de simplicidad, consideramos la traducción orientada a la sintaxis de las expresiones infijas al formato postfijo, una notación en la cual los operadores aparecen después de sus operaciones. Por ejemplo, el formato postfijo de la expresión  $9 - 5 + 2$  es  $95 - 2 +$ . La traducción al formato postfijo es lo bastante completa como para ilustrar el análisis sintáctico, y a la vez lo bastante simple como para que se pueda mostrar el traductor por completo en la sección 2.5. El traductor simple maneja expresiones como  $9 - 5 + 2$ , que consisten en dígitos separados por signos positivos y negativos. Una razón para empezar con dichas expresiones simples es que el analizador sintáctico puede trabajar directamente con los caracteres individuales para los operadores y los operandos



Un analizador léxico permite que un traductor maneje instrucciones de varios caracteres como identificadores, que se escriben como secuencias de caracteres, pero se tratan como unidades conocidas como tokens durante el análisis sintáctico; por ejemplo, en la expresión `cuenta+1`, el identificador `cuenta` se trata como una unidad. El analizador léxico en la sección 2.6 permite que aparezcan números, identificadores y “espacio en blanco” (espacios, tabuladores y caracteres de nueva línea) dentro de las expresiones

## Definición de sintaxis

una instrucción if-else es la concatenación de la palabra clave `if`, un paréntesis abierto, una expresión, un paréntesis cerrado, una instrucción, la palabra clave `else` y otra instrucción. Mediante el uso de la variable `expr` para denotar una expresión y la variable `instr` para denotar una instrucción, esta regla de estructuración puede expresarse de la siguiente manera:  $\text{instr} \rightarrow \text{if} ( \text{expr} ) \text{instr} \text{ else } \text{instr}$  en donde la flecha se lee como “puede tener la forma”. A dicha regla se le llama producción. En una producción, los elementos léxicos como la palabra clave `if` y los paréntesis se llaman terminales. Las variables como `expr` e `instr` representan secuencias de terminales, y se llaman no terminales

**Comparación entre tokens y terminales** En un compilador, el analizador léxico lee los caracteres del programa fuente, los agrupa en unidades con significado léxico llamadas lexemas, y produce como salida tokens que representan estos lexemas. Un token consiste en dos componentes, el nombre del token y un valor de atributo

**Análisis sintáctico** El análisis sintáctico (parsing) es el proceso de determinar cómo puede generarse una cadena de terminales mediante una gramática. Al hablar sobre este problema, es más útil pensar en que se va a construir un árbol de análisis sintáctico, aun cuando un compilador tal vez no lo construya en la práctica. No obstante, un analizador sintáctico debe ser capaz de construir el árbol en principio, o de lo contrario no se puede garantizar que la traducción sea correcta

**Recursividad a la izquierda** Es posible que un analizador sintáctico de descenso recursivo entre en un ciclo infinito. Se produce un problema con las producciones “recursivas por la izquierda” como  $\text{expr} \rightarrow \text{expr} + \text{term}$

**2.5.2 Adaptación del esquema de traducción** La técnica de eliminación de recursividad por la izquierda trazada en la figura 2.20 también se puede aplicar a las producciones que contengan acciones semánticas. En primer lugar, la técnica se extiende a varias producciones para `A`. En nuestro ejemplo, `A` es `expr` y hay dos producciones recursivas a la izquierda para `expr`, y una que no es recursiva. La técnica transforma las producciones  $A \rightarrow A\alpha \mid A\beta \mid$

Simplificación del traductor



En primer lugar, ciertas llamadas recursivas pueden sustituirse por iteraciones. Cuando la última instrucción que se ejecuta en el cuerpo de un procedimiento es una llamada recursiva al mismo procedimiento, se dice que la llamada es recursiva por la cola. Por ejemplo, en la función `resto`, las llamadas de `resto()` con los símbolos de preanálisis `+` y `-` son recursivas por la cola, ya que en cada una de estas bifurcaciones, la llamada recursiva a `resto` es la última instrucción ejecutada por esa llamada de `resto`.

```
void resto() {
    while( true ) {
        if ( preanálisis == '+' ) {
            coincidir('+'); term(); print('+'); continue;
        }
        else if preanálisis == '-' ) {
            coincidir('-'); term(); print('-'); continue;
        }
        break ;
    }
}
```

**Análisis léxico** Un analizador léxico lee los caracteres de la entrada y los agrupa en “objetos token”. Junto con un símbolo de terminal que se utiliza para las decisiones de análisis sintáctico, un objeto token lleva información adicional en forma de valores de atributos. Hasta ahora, no hemos tenido la necesidad de diferenciar entre los términos “token” y “terminal”, ya que el analizador sintáctico ignora los valores de los atributos q

## Reconocimiento de palabras clave e identificadores

La mayoría de los lenguajes utilizan cadenas de caracteres fijas tales como `for`, `do` e `if`, como signos de puntuación o para identificar las construcciones. A dichas cadenas de caracteres se les conoce como palabras clave. Las cadenas de caracteres también se utilizan como identificadores para nombrar variables, arreglos, funciones y demás. Las gramáticas tratan de manera rutinaria a los identificadores como terminales para simplificar el analizador sintáctico, el cual por consiguiente puede esperar el mismo terminal, por decir `id`, cada vez que aparece algún identificador en la entrada. Por ejemplo, en la siguiente entrada: `cuenta = cuenta + incrementó; (2.6)` el analizador trabaja con el flujo de terminales `id = id + id`. El token para `id` tiene un atributo que contiene el lexema. Si escribimos los tokens como n-uplas, podemos ver que las n-uplas para el flujo de entrada (2.6) son: `id`, `"cuenta"` = `id`, `"cuenta"` + `id`, `"incremento"` ;

## Un analizador léxico

Hasta ahora en esta sección, los fragmentos de pseudocódigo se juntan para formar una función llamada `escanear` que devuelve objetos token, de la siguiente manera: `Token escanear ()` { omitir espacio en blanco, como en la sección 2.6.1; manejar los números, como en la sección 2.6.3; manejar las palabras reservadas e

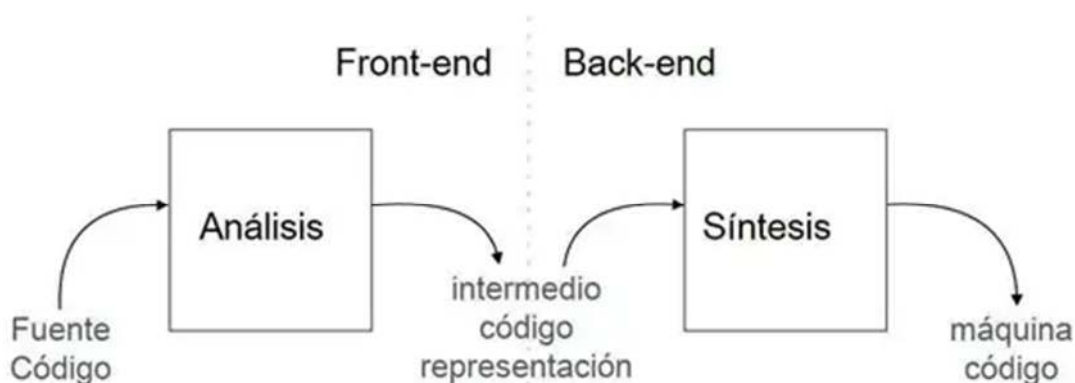
identificadores, como en la sección 2.6.4; /\* Si llegamos aquí, tratar el carácter de lectura de preanálisis vistazo como token \*/ Token t = new Token(vistazo); vistazo = espacio en blanco /\* inicialización, como vimos en la sección 2.6.2 \*/; return t; }

Las técnicas orientadas a la sintaxis que vimos en este capítulo pueden usarse para construir interfaces de usuario (front-end) de compiladores

El punto inicial para un traductor orientado a la sintaxis es una gramática para el lenguaje fuente. Una gramática describe la estructura jerárquica de los programas. Se define en términos de símbolos elementales, conocidos como terminales, y de símbolos variables llamados no terminales. Estos símbolos representan construcciones del lenguaje. Las reglas o producciones de una gramática consisten en un no terminal conocido como el encabezado o lado izquierdo de una producción, y de una secuencia de terminales y no terminales, conocida como el cuerpo o lado derecho de la producción. Un no terminal se designa como el símbolo inicial. Al especificar un traductor, es conveniente adjuntar atributos a la construcción de programación, en donde un atributo es cualquier cantidad asociada con una construcción. Como las construcciones se representan mediante símbolos de la gramática, el concepto de los atributos se extiende a los símbolos de la gramática. Algunos ejemplos de atributos incluyen un valor entero asociado con un terminal num que representa números, y una cadena asociada con un terminal id que representa identificadores

### Secciones del Front-End (Análisis Léxico, Sintáctico y Semántico)

Los compiladores básicamente se dividen en dos partes, **siendo la primera de ellas el “Front End”**, y que es la parte del compilador encargada de analizar y comprobar la validez del código fuente y en base a ella crear los valores de la tabla de símbolos. Esta parte generalmente es independiente del sistema operativo para el cual se está compilando un programa.



La segunda parte del compilador es la llamada "Back End", parte en la cual es generado el código máquina, el cual es creado, de acuerdo a lo analizado en el "Front End", para una plataforma específica, que puede ser Windows, Mac, Linux o cualquier otra.

Cabe destacar que los resultados obtenidos por el "Back End" no pueden ser ejecutados en forma directa, para ello es necesario la utilización de un proceso de enlazado, llamado "Linker", el cual básicamente es un software que recoge los objetos generados en la primera instancia de compilación, incluyendo la información de las bibliotecas, los depura y luego enlaza el código objeto con sus respectivas biblioteca, para finalmente crear un archivo ejecutable.

aquí es donde vamos a ver como funciona nuestro front end y back end como va funcionar nuestro compilador primero hacemos nuestro código fuente ya que lo tengamos se hace un análisis para transformarlo que pasa por un intermedio de código representando se va formando más bien como un diagrama de árbol de nuestro código se hace la síntesis y pasa a código máquina donde se va enviar nuestro código hacia el cpu para que nos muestre qué es lo que vamos a querer o necesitar

## Actividades semana 6 (Oct 26-30, 2020)

### **Gramáticas fuertes de LL (k) Ramirez Peña Carlos Ivan**

#### VIDEO 1

El análisis de arriba hacia abajo mediante el análisis predictivo, rastrea la derivación más a la izquierda de la cadena mientras se construye el árbol de análisis. Comienza desde el símbolo de inicio de la gramática y "predice" la siguiente producción utilizada en la derivación.

Dicha "predicción" es ayudada por tablas de análisis (construidas fuera de línea).

La siguiente producción que se utilizará en la derivación se determina utilizando el siguiente símbolo de entrada para buscar la tabla de análisis (símbolo de anticipación).

Poner restricciones en la gramática asegura que ningún espacio en la tabla de análisis contenga más de una producción.

En el momento de la construcción de la tabla de análisis, si dos producciones se vuelven elegibles para colocarse en el mismo espacio de la tabla de análisis, la gramática se declara no apta para predicciones.

### **Análisis de Descenso Recursivo**

Estrategia de análisis de arriba hacia abajo.

Una función / procedimiento para cada no terminal.

Las funciones se llaman entre sí de forma recursiva, según la gramática.

La pila de recursividad maneja las tareas de la pila del analizador LL (1).

LL (1) condiciones que deben cumplirse para la gramática.

Se puede generar automáticamente a partir de la gramática.

La codificación manual también es fácil.

La recuperación de errores es superior.

### **Análisis de abajo hacia arriba**

Comience en las hojas, construya el árbol de análisis en segmentos pequeños, combine los árboles pequeños para hacer árboles más grandes, hasta que se alcance la raíz.

Este proceso se llama reducción de la oración al símbolo inicial de la gramática.

Una de las formas de "reducir" una oración es seguir la derivación más a la derecha de la oración al revés.

1.- El análisis sintáctico Shift-Reduce implementa dicha estrategia.

2.-Utiliza el concepto de asa para detectar cuándo realizar reducciones.

### **Análisis de Mayús-Reducir**

**Mango:** Mango de una forma de oración correcta  $\gamma$ , es una producción  $A \rightarrow \beta$  y un puesto en  $\gamma$ , donde la cadena  $\beta$  puede ser encontrado y reemplazado por  $A$ , para producir la forma oracional derecha anterior en una derivación más a la derecha de  $\gamma$ .

Es decir, si  $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta w$ , entonces  $A \rightarrow \beta$  en la posición siguiente  $\alpha$  es un mango de  $\alpha \beta w$ .

Un asa siempre aparecerá en la parte superior de la pila, nunca sumergida dentro de la pila.

En el análisis sintáctico S-R, ubicamos el mango y lo reducimos por el LHS de la producción repetidamente, para llegar al símbolo de inicio.

Estas reducciones, de hecho, trazan una derivación más a la derecha de la oración a la inversa. Este proceso se llama poda de mango.

LR-Parsing es un método de análisis sintáctico con reducción de cambios.

## Rodriguez Perez Luis Diego VIDEO 2

gramáticas libres de contexto

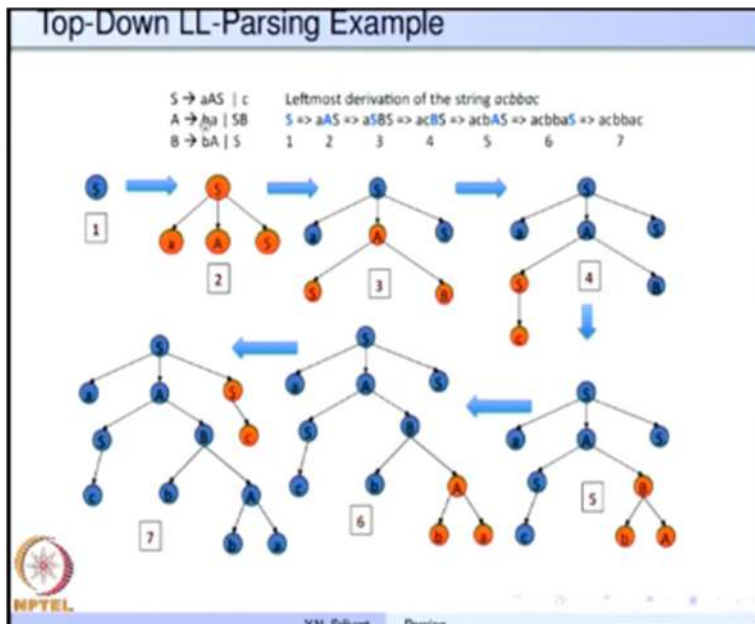
un autómata de empuje hacia abajo M tiene un conjunto finito de estados  $Q$ , un alfabeto de entrada  $\Sigma$ , un alfabeto de pila  $\Gamma$ , hay un estado inicial  $q_0$  y un símbolo de pila inicial  $Z_0$ , hay un conjunto de estados finales  $F$

que es un subconjunto del conjunto  $Q$  Y hay funciones transición que muestra cómo se comporta el autómata, entonces  $\delta$  la función de transición es un mapeo de  $Q \times \Sigma \cup \{\epsilon\} \times \Gamma^* \rightarrow Q$ . entonces  $Q$  es el conjunto de estados,  $\Sigma$  es la entrada del alfabeto y dado que el autómata puede hacer movimientos en epsilon, se es el movimiento silencioso epsilon está permitido y hay un símbolo de la parte superior de la pila que se ve antes haciendo un movimiento y el movimiento puede ser a cualquiera de estos  $Q \times \Sigma \times \Gamma^*$ , tan finito de los subconjuntos de  $Q \times \Sigma \times \Gamma^*$ .

los símbolos  $\Gamma$  será el nuevo de la parte superior de la pila y también definimos la aceptación del lenguaje por un autómata de empuje hacia abajo uno es por estado final el otro es por pila vacía para la aceptación por estado final la máquina debe partir de el estado inicial, pasar a un estado final y vaciar la entrada también y la pila no importa para la aceptación por la pila vacía, comienza desde el inicio el estado se mueve a algún estado pero el proceso no sólo vacía la entrada sino también la pila entonces, el estado al que mueve se mueve no es muy importante por lo tanto, a veces lo ponemos  $F$  igual para este tipo de autómata

los autómatas de empuje hacia abajo son tan iguales como en el caso del estado finito no determinista tenemos autómatas de empuje hacia abajo no determinista y similar a DFA tenemos DPDA sin embargo en el caso de NFA Y DFA se demostró que eran otras palabras equivalentes el idioma que fue aceptado por la NFA es también el idioma aceptado por cualquier equivalente al DFA por lo tanto podemos convertir cada NFA en un DFA, donde como en caso de NPDA, NPDA es

estrictamente más poderoso que la clase DPDA y en NPDA para los que no pueden diseñar DPDA



que es un proceso de análisis sintáctico

el análisis es un proceso de construcción de un árbol de análisis para una oración generada por determinada gramática entonces una gramática genera una cadena que ya vimos y empujamos hacia abajo el autómata para aceptar una cadena, ahora la parte del análisis es proceso de construir un árbol de análisis para una fase

entonces usamos un autómata de empuje hacia abajo con algunas acciones adicionales para construir un análisis entonces básicamente una máquina de análisis no es nada sino un autómata de empuje hacia abajo determinista sino si no hay restricciones en el idioma y la forma de gramática que se utiliza para analizar los lenguajes de árbol de contexto es bastante cara por lo tanto requiere un tiempo de cubo de  $O(n)$  para el análisis, por lo que hay dos algoritmos muy conocidos el algoritmo cocke-younger-kasami y el earley

este en este análisis se basa en una clase de gramáticas llamadas gramáticas LL(1) y utiliza una estrategia de análisis conocida como análisis sintáctico de arriba hacia abajo el análisis sintáctico de reducción de desplazamiento requiere que las gramáticas estén en el LR en formulario esto se basa el análisis de arriba hacia abajo

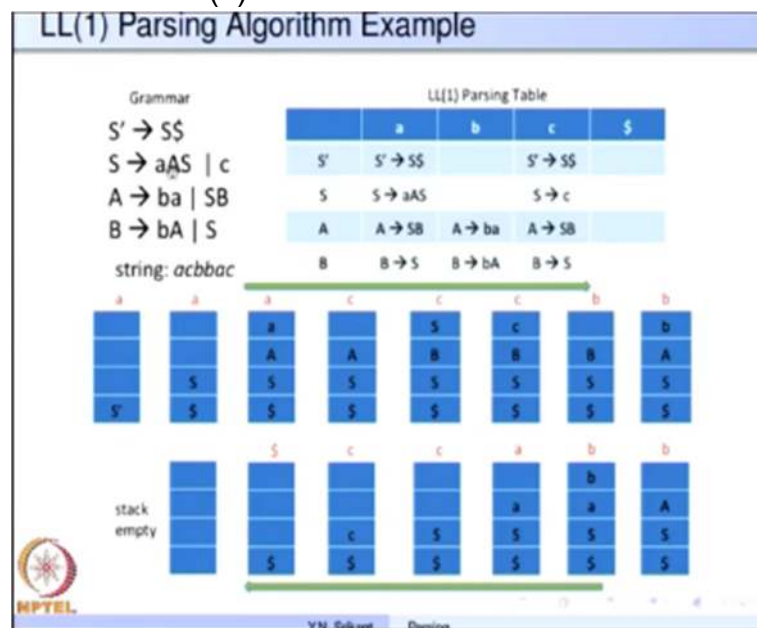
el análisis de arriba hacia abajo

usando predictivo parsing, rastrea la derivación más a la izquierda de la cadena mientras se construye el árbol de análisis entonces comenzamos desde el símbolo inicio y predecimos la producción que se usa en la derivación por lo que una predicción como la que dije leemos información adicional y es conocida como la

tabla de análisis que se construye sin conexión y se almacena de cómo se construye la tabla de análisis

cómo se construye la tabla de análisis

dijo que las gramáticas LL(1) son una subclase de gramáticas libres de contexto así cuando decidimos subclase debemos poner restricciones en la gramática libre de contexto de nuevo dar un mensaje para comprobar si las restricciones se cumplen o no definamos una clase si la gramáticas llamadas gramáticas LL(1) fuertes y luego de ver como se puede usar para nuestro análisis de estrategias de análisis de LL(1)



La sintaxis más bien se describe mediante gramáticas libres de contexto en Una gramática da una especificación sintáctica precisa, y fácil de comprender es de ver un lenguaje de programación Para ciertas clases de gramática se puede construir automáticamente un analizador sintáctico, que determina si un programa está bien construido.



las gramática más bien es diseñada que da una estructura al lenguaje de programación que se utiliza en la traducción del programa fuente en código objeto correcto y para la detección de errores. Existen herramientas que a partir de la descripción de la gramática se puede generar el código del analizador sintáctico dónde vamos cómo se va construyendo por medio de de nuestra tabla sintáctico para que veamos nuevos resultados haciendo más bien movimientos y ya despues se hacen los diagramas donde se hace de arriba hacia abajo para que hagamos nuestro código fuente con nueva estructura

### Computation of *FIRST*: Terminals and Nonterminals

```

{
  for each  $(a \in T)$   $FIRST(a) = \{a\}$ ;  $FIRST(\epsilon) = \{\epsilon\}$ ;
  for each  $(A \in N)$   $FIRST(A) = \emptyset$ ;
  while (FIRST sets are still changing) {
    for each production  $p$  {
      Let  $p$  be the production  $A \rightarrow X_1 X_2 \dots X_n$ ;
       $FIRST(A) = FIRST(A) \cup (FIRST(X_1) - \{\epsilon\})$ ;
       $i = 1$ ;
      while  $(\epsilon \in FIRST(X_i) \ \&\& \ i \leq n - 1)$  {
         $FIRST(A) = FIRST(A) \cup (FIRST(X_{i+1}) - \{\epsilon\})$ ;  $i++$ ;
      }
      if  $(i == n) \ \&\& \ (\epsilon \in FIRST(X_n))$ 
         $FIRST(A) = FIRST(A) \cup \{\epsilon\}$ 
    }
  }
}

```

## análisis sintaxis

Il1 se refiere a mirar hacia adelante y dado que no es fácil implementar es más largo que uno no se preocupa por del I2 I3 y así sucesivamente también definimos primero y seguimos los conjuntos la definición la primera alfa es un conjunto que consta del símbolo  $\alpha$  a qué puede la derivada de alfa por lo que  $a \in T$  y alfa deriva que  $X$  y  $X$  también es la primera cadena terminal de epsilon por definición epsilon y la siguiente es todos esos simbolos en realidad estan despues del no terminal  $a$  en cualquier forma oracional por lo que es por eso que se define como aunque el simbolo  $CA$  tal que se deriva alfa a beta lo que significa que  $a$  ocurre en el forma oracional y recopilamos todos estos símbolos pequeños que a menudo no termina aquí ser un símbolo de terminal o el final del archivo

la definición de la condición de apisonador Il 1 en función de la primera Queremos hacer lo que vamos querer mostrar un algoritmo para calcular el análisis para la tabla Il 1 gramáticas y en esas se basan en la primera y la siguiente supongamos que  $G$  es nuestra gramática  $G$  es Il 1 si cada produccion aqui en alfa o beta se cumple las siguientes condiciones por lo que el punto es durante el análisis 1 cuando hay una variedad de producciones: ir a alfa o beta que deberíamos ser capaz de decir que uno de ellos se aplica mirando el siguiente símbolo de entrada así que estos pueden afirmarse si para cada elección de producciones somos capaces de satisfacer esta condición particular que se establece aquí la condición dice conjunto de símbolos de dirección de intersección alfa conjunto de símbolos de dirección de beta es  $\Phi$

¿cual es el conjunto de símbolos de direcciones ?

el conjunto de símbolos de dirección de gamma es epsilon está en el primer de gamma, luego es el primero de gamma menos epsilon unión siguiendo a, de lo



contrario, es solo el primero de gamma por lo que sí se sabe que gamma significa ya sea alfa o beta, en otras palabras saber lo que es símbolo alfa derivada y beta deriva los primeros símbolo de estos no deben ser lo mismo, de lo contrario la elección no se puede hacer mirando el siguiente símbolo de entrada por lo que hay una formulación equivalente en el conjunto que saben dicen T inscripciones reserva como cordero y diga que T y vamos a uno, dice primero del punto alfa después de un alfa y beta 2, estas condiciones son idénticas, es fácil de ver por qué supongamos que alfa no produce ningún epsilon en ese caso epsilon en el primero de alfa es falso, por lo que el conjunto de símbolos de dirección simplemente se convierte en el primero de alfa por lo que el seguir es intrascendente en ese caso, que alfa produce epsilon por lo que primero de alfa contiene epsilon en tal caso si esto es epsilon entonces el siguiente conjunto de a también se incluirá en el primer cálculo

**LL(1) Table Construction Example 1**

LL(1) Parsing Table for the original grammar

	if	id	else	a	\$
$S'$	$S' \rightarrow S\$$			$S' \rightarrow S\$$	
$S$	$S \rightarrow \text{if id } S$ $S \rightarrow \text{if id } S \text{ else } S$			$S \rightarrow a$	

Original Grammar

$S' \rightarrow S\$$   
 $S \rightarrow \text{if id } S \mid \text{if id } S \text{ else } S \mid a$

tokens: if, id, else, a

$\text{dirsymb}(S\$) = \{\text{if}, a\}$ ;  $\text{dirsymb}(a) = \{a\}$   
 $\text{dirsymb}(\text{if id } S) = \{\text{if}\}$   
 $\text{dirsymb}(\text{if id } S \text{ else } S) = \{\text{if}\}$

$\text{dirsymb}(\text{if id } S) \cap \text{dirsymb}(a) = \emptyset$   
 $\text{dirsymb}(\text{if id } S \text{ else } S) \cap \text{dirsymb}(a) = \emptyset$   
 $\text{dirsymb}(\text{if id } S) \cap \text{dirsymb}(\text{if id } S \text{ else } S) = \emptyset$

Grammar is not LL(1)

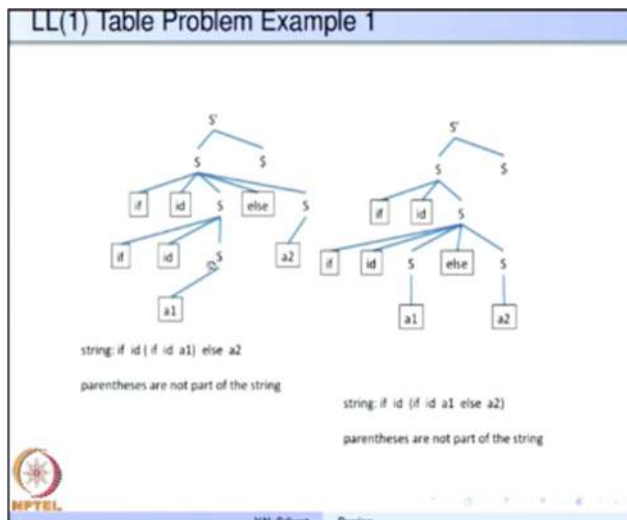
NPTEL

Y.N. Srikant

Las reglas de derivación que se aplican desde el símbolo inicial hasta alcanzar la sentencia reconocer, pueden reemplazar los símbolos no terminales tomando el de más a la izquierda (derivaciones más a la izquierda) o tomando el no terminal de más a la derecha (derivaciones más a la derecha).

Habitualmente se trabaja con derivaciones más a la izquierda. Se puede demostrar que todo árbol de análisis sintáctico tiene asociado una única derivación izquierda y una única derivación derecha. Sin embargo, una sentencia de un lenguaje puede dar lugar a más de un árbol de análisis sintáctico tanto por la derecha como por la izquierda.

Para determinar el árbol más a la izquierda se parte del símbolo inicial S y en la variable parse se van poniendo las producciones aplicadas, utilizando números para indicar los códigos de cada regla aplicada. Siempre se expande el no terminal más a la izquierda en el árbol hasta alcanzar un símbolo terminal



más bien vemos como podemos hacer nuestras diagramas más bien se hace el cálculo de la derivación para poder hacer nuestro diagrama donde podemos ver los resultado hacer las derivaciones más a la izquierda que en la derecha ahora si queremos hacer la derivación Para obtener el árbol más a la derecha se parte del símbolo inicial S y se van aplicando las producciones de forma que se eligen en primer lugar las expansiones del símbolo no terminal más a la derecha en el árbol. Así en la variable producción utilizada (p.u.) se van colocando los números de las producciones utilizadas.

análisis sintáctico con análisis descendente recursivo y análisis de abajo hacia arriba

una transformación de símbolo inútil y ahora también hay dos transformaciones más una en la eliminación de recursividad izquierda y otro conocido como factorización izquierda de las gramáticas recursivas izquierdas post problemas 211 generación de parcelas el problema es que las gramáticas recursivas izquierdas no satisface la condición II 1 así que vamos a entender que las gramáticas recursivas exactamente a la izquierda son así una gramática no tiene una terminal A como que en más de una aplicación de las producciones que produce digamos alfa para que pueda ver que el primer símbolo de esta forma de oración también significa que podemos producir tantos como usted sabe que podemos seguir aplicando las producciones tantas veces como queramos y esto se conoce como recursiva por la izquierda

## Grammar Transformations may not help!

Original Grammar

$$S' \rightarrow SS$$

$$S \rightarrow \text{if } id \ S \mid$$

$$\quad \text{if } id \ S \text{ else } S \mid$$

$$a$$

LL(1) Parsing Table for modified grammar

	if	else	a	\$
$S'$	$S' \rightarrow SS$		$S' \rightarrow SS$	
$S$	$S \rightarrow \text{if } id \ S \mid$	$S \rightarrow \text{if } id \ S \text{ else } S \mid$	$S \rightarrow a$	
$S1$		$S1 \rightarrow \epsilon$ $S1 \rightarrow \text{else } S$		$S1 \rightarrow \epsilon$

Left-Factored Grammar

$$S' \rightarrow SS$$

$$S \rightarrow \text{if } id, S \ S1 \mid a$$

$$S1 \rightarrow \epsilon \mid \text{else } S$$

tokens: if, id, else, a

Grammar is not LL(1)

Choose  $S1 \rightarrow \text{else } S$  instead of  $S1 \rightarrow \epsilon$  on lookahead *else*.  
This resolves the conflict. Associates *else* with the innermost *if*

NPTEL

## análisis de descenso recursivo

es una estrategia de análisis de arriba hacia abajo y en cambio básicamente de una gramática basada en tablas ustedes conocen el analizador basado en las tablas como en el caso de de Il 1 analizador aquí este es un programa alineado que conoce como tal, por lo que estamos va a tener una función o procedimiento para no terminar por lo que estos son difíciles programas codificados en lugar de parcelas controladas por las tablas, las funciones se llaman entre sí recursivamente es por eso que esto se llama un analizador de descenso recursivo basado en la gramática en el analizador podemos generar analizadores de descenso recursivos también automáticamente desde la gramática y la codificación también son muy fáciles incluso si no las genera automáticamente la ventaja del descenso recursivo los analizadores es que la recuperación de errores en tales gramáticas tales paquetes es superior a de los analizadores

## Examples (contd.)

- $E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow id$   
For the string  $= id + id * id$ , two rightmost derivation marked with handles are shown below

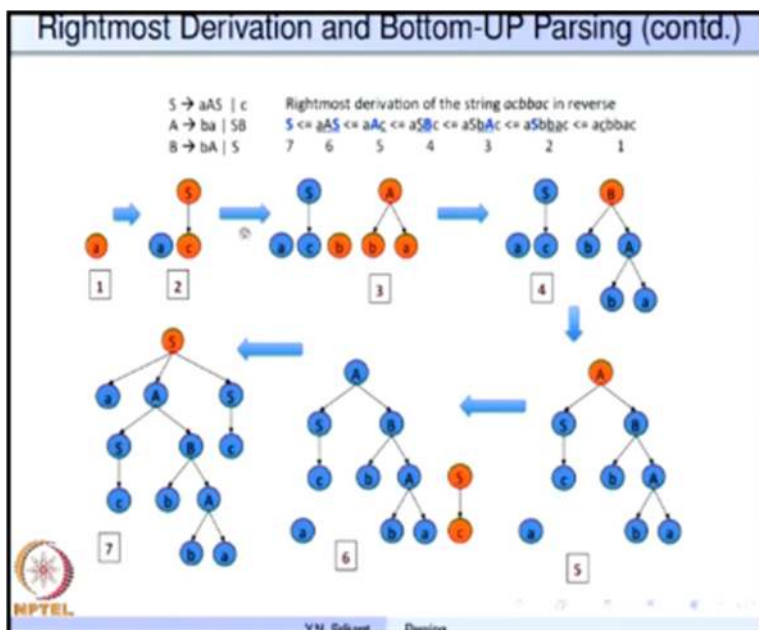
$E \Rightarrow \underline{E + E} \ (E + E, E \rightarrow E + E)$   
 $\Rightarrow \underline{E + E * E} \ (E * E, E \rightarrow E * E)$   
 $\Rightarrow E + E * \underline{id} \ (id, E \rightarrow id)$   
 $\Rightarrow E + \underline{id} * id \ (id, E \rightarrow id)$   
 $\Rightarrow \underline{id} + id * id \ (id, E \rightarrow id)$   
 $E \Rightarrow \underline{E * E} \ (E * E, E \rightarrow E * E)$   
 $\Rightarrow E * \underline{id} \ (id, E \rightarrow id)$   
 $\Rightarrow E + E * \underline{id} \ (E + E, E \rightarrow E + E)$   
 $\Rightarrow E + \underline{id} * id \ (id, E \rightarrow id)$   
 $\Rightarrow \underline{id} + id * id \ (id, E \rightarrow id)$



¿cómo generamos analizadores de descenso recursivo automáticamente ?  
 el esquema se basa en la estructura de las producciones, por lo que el generador mira las producciones y genera el programa las condiciones por lo que vamos a comprobar la condición ll una y luego llama a la función generadora get token obtiene el siguiente token del analizador léxico y lo coloca en el token de variable global este es el error de la función de suposición nuestra impresión muestra un mensaje de error

análisis de abajo hacia arriba

hasta ahora hemos discutido el análisis de arriba hacia abajo la estrategia en la que el símbolo de inicio se fue expandiendo progresivamente y finalmente llegamos a las hojas que corresponde a la cadena que se va a analizar en caso del análisis de abajo hacia arriba, comenzamos en las hojas construimos el árbol del análisis sintáctico en pequeños segmentos al combinar los árboles pequeños para hornear hacer más bien grandes hasta alcanzar la raíz entonces es por eso que se llama una estrategia de abajo hacia arriba entonces, este proceso se llama reducción de la oración al símbolo inicial de la gramática en unas formas de reducir una oración es dejar en barbecho la derivación más a la derecha de la oración al revés . el análisis sintáctico reducido implementa una de estas estrategias y utiliza el concepto de lo que es conocido como un identificador para detectar cuándo realizar tales reducciones



¿una manija?

es un identificador de una forma oracional derecha gamma es una producción A la beta y una posición en gamma entonces, tomemos una derivación de esta forma S yendo a alfa A w es una derivación más a la derecha en muchos pasos y ahora reemplazamos alfa A por beta y el contexto en que aplicamos es alfa en el lado izquierdo y en lado derecho así que ahora, en esta forma enunciativa alfa beta w decidimos que A la beta en la posición siguiente alfa es manejador de la cadena o forma sentencial alfa beta w, entonces básicamente queremos que ubique el lado derecho de la producción que sea aplicable en este punto

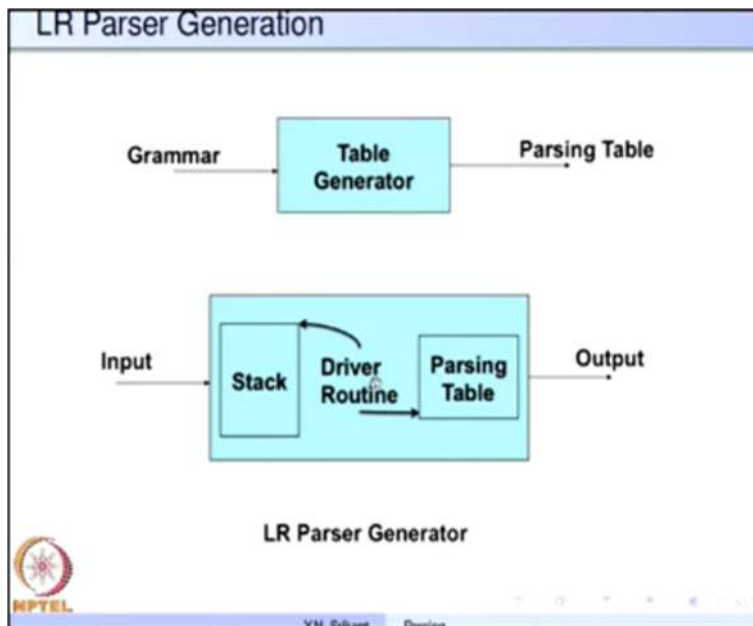
proceso de análisis de abajo hacia arriba

así que hasta dijimos localizamos una manija y luego reemplazar la manija por el lado derecho, por el lado izquierdo no termina de la producción y así sucesivamente no hemos respondido la pregunta de cómo ubicamos exactamente un identificador en la oración correcta en el caso LR el analizador que vamos a estudiar más adelante en detalle utiliza una máquina de estado finito determinista para detectar la condición que maneja ahora está en la pila

¿que producción usar en caso de que haya más de una con el mismo ?

el lado derecho es posible pero la pregunta respondida por el analizador LR usando una tabla de análisis sintáctico similar a una tabla de análisis sintáctico LR para elegir la producción que sea aplicable sucede que el estado DFA nos dice no solo que es un manejador sino que también nos dice la producción que es aplicable en ese punto entonces en el tercer componente es una pila una pila se usa para implementar

un analizador de reducción de desplazamiento y el analizador de reducción de desplazamiento no es más que un autómata de empuje determinista aumentando tiene varias acciones cuatro para ser precisos hay una acción de cambio que cambia el siguiente símbolo de entrada al parte superior de la pila. luego hay una acción de reducción, la mano derecha del mango está en la parte superior de la pila entonces, ubica la mano izquierda del mango dentro de la pila mirando el número de símbolos que componen el lado derecho reemplazar el mango por el LHS de una producción adecuada que sea aplicable en este punto



las gramáticas LR también son un subconjunto de gramáticas libres de contexto general y una gramática particular para un idioma puede no ser LR 1 o LR 2 y es posible que podamos reescribir de estas gramáticas convertirse en LR1 o LR2 pero eso no significa que cada gramática que se escribe para un idioma pasar el texto LR1 o LR2 deberíamos ser lo suficientemente inteligentes como para escribir la gramática de modo que la prueba se complete con éxito entonces las gramáticas LR 1 afortunadamente se pueden escribir con bastante facilidad para prácticamente todas las construcciones de lenguaje de programación para las que el contexto libre las gramáticas se pueden escribir solo necesitas un poco de práctica y el análisis sintáctico LR es la mayoría de los métodos de análisis sintáctico reductor de cambios sin retroceso que se conocen hoy en día también hay una rutina de derivación esto es, este bloque está completo el analizador entonces, el analizador consta de una pila una rutina de derivación y una tabla de análisis que genera automáticamente entonces cuando se da la entrada la rutina de derivación dice la entrada manipula la pila apropiadamente usando la tabla de análisis y los generadores algunos en la salida en forma de árbol analizador o mensajes de error etc

**Hernández Ruiz Adrián Felipe VIDEO 3**

## Condiciones LL (1)

- ❖ Sea  $G$  una gramática libre de contexto
- ❖  $G$  es LL (1) si para cada par de producciones  $A \rightarrow \alpha$  y  $A \rightarrow \beta$ , la siguiente condición se cumple.
  - $\text{dirsymb}(\alpha) \cap \text{dirsymb}(\beta) = \emptyset$ , where

$\text{dirstymb}(\gamma) = \text{if}(\varepsilon \in \text{first}(\gamma)) \text{ then } ((\text{first}(\gamma) - \{\varepsilon\}) \cup \text{follow}(A)) \text{ else first}(\gamma)$  ( $\gamma$  stands for  $\alpha$  or  $\beta$ )

→  $\text{dirstymb}$  significa "conjunto de símbolos de dirección"

❖ Una formulación equivalente (como en el libro de ALSU) es como abajo

→  $\text{first}(\alpha, \text{follow}(A)) \cap \text{first}(\beta, \text{follow}(A)) = \emptyset$

❖ Construcción de la tabla de análisis sintáctico LL (1)

para cada producción  $A \rightarrow \alpha$

para cada símbolo  $s \in \text{dirstymb}(\alpha)$

/\*  $s$  puede ser un símbolo de terminal o  $\$$  \*/

add  $A \rightarrow \alpha$  to  $LLPT[A, s]$

Hacer que cada entrada indefinida de  $LLPT$  sea un error

para cada producción  $A \rightarrow \alpha$

para cada símbolo de terminal  $a \in \text{first}(\alpha)$

add  $A \rightarrow \alpha$  to  $LLPT[A, a]$

if  $\varepsilon \in \text{first}(\alpha)$ {

para cada símbolo de terminal  $b \in \text{follow}(A)$

add  $A \rightarrow \alpha$  to  $LLPT[A, b]$

if  $\$ \in \text{follow}(A)$

add  $A \rightarrow \alpha$  to  $LLPT[A, \$]$

}

Hacer que cada entrada indefinida de  $LLPT$  sea un error

❖ Una vez completada la construcción de la tabla LL (1) (siguiendo cualquiera de los dos métodos), si algún espacio en la tabla LL (1) tiene dos o más producciones, entonces la gramática NO es LL (1)

Ejemplos simples de gramática LL (1):

- P1:  $S \rightarrow \text{if } (a) S \text{ else } S \mid \text{while } (a) S \mid \text{begin } SL \text{ end}$
- P2:  $SL \rightarrow S S'$
- P3:  $S' \rightarrow ;, SL \mid \epsilon$
- {if, while, begin, end, a, (, ), ;} are all terminal symbols
- Clearly, all alternatives of P1 start with distinct symbols and hence create no problem
- P2 has no choices
- Regarding P3,  $\text{dirstymb} (;, SL) = \{ ;, \}$ , and  $\text{dirstymb}(\epsilon) = \{\text{end}\}$ , and the two have no common symbols
- Hence the grammar is LL(1)



**LL(1) Table Construction Example 1**

LL(1) Parsing Table for the original grammar

	if	id	else	a	\$
$S'$	$S' \rightarrow SS$			$S' \rightarrow SS$	
$S$	$S \rightarrow \text{if id } S$	$S \rightarrow \text{id } S$	$S \rightarrow \text{else } S$	$S \rightarrow a$	

Original Grammar

Grammar is not LL(1)

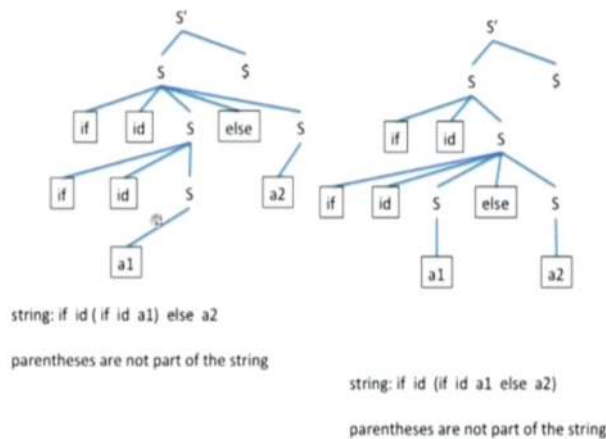
tokens: if, id, else, a

$S' \rightarrow SS$   
 $S \rightarrow \text{if id } S \mid \text{id } S \text{ else } S \mid a$

$\text{dirstymb}(SS) = \{\text{if}, a\}$ ;  $\text{dirstymb}(a) = \{a\}$   
 $\text{dirstymb}(\text{if id } S) = \{\text{if}\}$   
 $\text{dirstymb}(\text{if id } S \text{ else } S) = \{\text{if}\}$

$\text{dirstymb}(\text{if id } S) \cap \text{dirstymb}(a) = \emptyset$   
 $\text{dirstymb}(\text{if id } S \text{ else } S) \cap \text{dirstymb}(a) = \emptyset$   
 $\text{dirstymb}(\text{if id } S) \cap \text{dirstymb}(\text{if id } S \text{ else } S) \neq \emptyset$





### LL(1) Table Construction Example 2

Original Grammar

$$S' \rightarrow SS$$

$$S \rightarrow \text{if id } S \mid \text{if id } S \text{ else } S \mid a$$

LL(1) Parsing Table for modified grammar

	if	else	a	\$
$S'$	$S' \rightarrow SS$		$S' \rightarrow SS$	
$S$	$S \rightarrow \text{if id } S \mid S1$		$S \rightarrow a$	
$S1$		$S1 \rightarrow \epsilon$ $S1 \rightarrow \text{else } S$		$S1 \rightarrow \epsilon$

Left-Factored Grammar

$$S' \rightarrow SS$$

$$S \rightarrow \text{if id } S \mid S1 \mid a$$

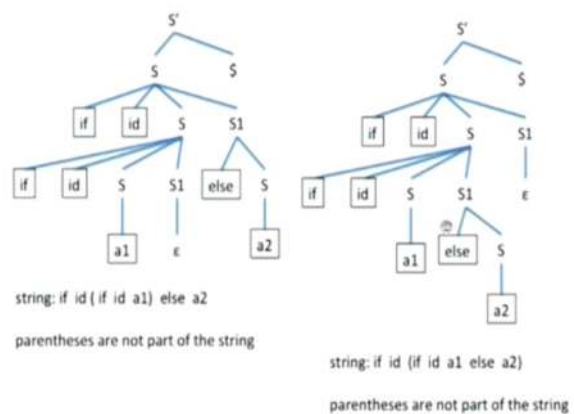
$$S1 \rightarrow \epsilon \mid \text{else } S$$

tokens: if, id, else, a

Grammar is not LL(1)

$\text{dirsymb}(\text{if id } S \mid S1) \cap \text{dirsymb}(a) = \emptyset$

$\text{dirsymb}(\epsilon) \cap \text{dirsymb}(\text{else } S) = \emptyset$



### LL(1) Table Construction Example 3

Original Grammar

$$S' \rightarrow SS$$

$$S \rightarrow aAS \mid c$$

$$A \rightarrow ba \mid SB$$

$$B \rightarrow bA \mid S$$

Grammar is LL(1)

LL(1) Parsing Table

	a	b	c	\$
$S'$	$S' \rightarrow SS$		$S' \rightarrow SS$	
$S$	$S \rightarrow aAS$		$S \rightarrow c$	
$A$	$A \rightarrow SB$	$A \rightarrow ba$	$A \rightarrow SB$	
$B$	$B \rightarrow S$	$B \rightarrow ba$	$B \rightarrow S$	

$\text{first}(S) = \{a, c\}$   
 $\text{first}(A) = \{a, b, c\}$   
 $\text{first}(B) = \{a, b, c\}$

$\text{dirsymb}(aAS) \cap \text{dirsymb}(c) = \emptyset$   
 $\text{dirsymb}(ba) \cap \text{dirsymb}(SB) = \emptyset$   
 $\text{dirsymb}(bA) \cap \text{dirsymb}(S) = \emptyset$

$\text{follow}(S) = \{a, b, c, \$\}$   
 $\text{follow}(A) = \{a, c\}$   
 $\text{follow}(B) = \{a, c\}$

$\text{dirsymb}(SS) = \{a, c\}$   
 $\text{dirsymb}(aAS) = \{a\}$   
 $\text{dirsymb}(c) = \{c\}$

$\text{dirsymb}(ba) = \{b\}$   
 $\text{dirsymb}(SB) = \{a, c\}$   
 $\text{dirsymb}(bA) = \{b\}$   
 $\text{dirsymb}(S) = \{a, c\}$

## Eliminación de los símbolos inútiles

Now we study the grammar transformations, elimination of useless symbols, elimination of left recursion and left factoring

- ❖ Given a grammar  $G = (N, T, P, S)$ , a non-terminal  $X$  is useful if  $S \Rightarrow^* \alpha X \beta \Rightarrow w$ , where,  $w \in T^*$   
Otherwise,  $X$  is useless
- ❖ Two conditions have to be met to ensure that  $X$  is useful



1.  $X \Rightarrow^* w, w \in T^*$  (X derives some terminal string)
2.  $S \Rightarrow^* \alpha X \beta$  (X occurs in some string derivable from S)

## Libro de compiladores

### Análisis léxico

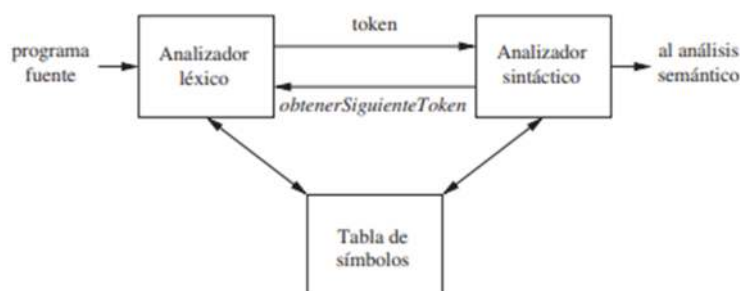
es útil empezar con un diagrama o cualquier otra descripción de los lexemas de cada token. De esta manera, podemos escribir código para identificar cada ocurrencia de cada lexema en la entrada, y devolver información acerca del token identificado.

Podemos producir también un analizador léxico en forma automática, especificando los patrones de los lexemas a un generador de analizadores léxicos, y compilando esos patrones en código que funcione como un analizador léxico. Este método facilita la modificación de un analizador léxico, ya que sólo tenemos que reescribir los patrones afectados, y no todo el programa.

Agiliza también el proceso de implementar el analizador, ya que el programador especifica el software en el nivel más alto de los patrones y se basa en el generador para producir el código detallado.

### La función del analizador léxico

Como la primera fase de un compilador, la principal tarea del analizador léxico es leer los caracteres de la entrada del programa fuente, agruparlos en lexemas y producir como salida una secuencia de tokens para cada lexema en el programa fuente. El flujo de tokens se envía al analizador sintáctico para su análisis. La función del analizador léxico Como la primera fase de un compilador, la principal tarea del analizador léxico es leer los caracteres de la entrada del programa fuente, agruparlos en lexemas y producir como salida una secuencia de tokens para cada lexema en el programa fuente. El flujo de tokens se envía al analizador sintáctico para su análisis.



### Comparación entre análisis léxico y análisis sintáctico

La sencillez en el diseño es la consideración más importante. La separación del análisis léxico y el análisis sintáctico a menudo nos permite simplificar por lo menos una de estas tareas. Por ejemplo, un analizador sintáctico que tuviera que manejar los comentarios y el espacio en blanco como unidades sintácticas sería mucho más complejo que uno que asumiera que el analizador léxico ya ha eliminado los comentarios y el espacio en blanco.

Si vamos a diseñar un nuevo lenguaje, la separación de las cuestiones léxicas y sintácticas puede llevar a un diseño más limpio del lenguaje en general.

### Tokens, patrones y lexemas

Un token es un par que consiste en un nombre de token y un valor de atributo opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico. A partir de este momento, en general escribiremos el nombre de un token en negrita. Con frecuencia nos referiremos a un token por su nombre.

- Un patrón es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.
- Un lexema es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

TOKEN	DESCRIPCIÓN INFORMAL	LEXEMAS DE EJEMPLO
<b>if</b>	caracteres i, f	if
<b>else</b>	caracteres e, l, s, e	Else
<b>comparacion</b>	< > <= >= o == o !=	<=, !=
<b>id</b>	letra seguida por letras y dígitos	pi, puntuacion, D2
<b>numero</b>	cualquier constante numérica	3.14159, 0, 6.02e23
<b>literal</b>	cualquier cosa excepto ", rodeada por "'s	"core dumped"

### Atributos para los tokens

Cuando más de un lexema puede coincidir con un patrón, el analizador léxico debe proporcionar a las subsiguientes fases del compilador información adicional sobre el lexema específico que coincidió. Por ejemplo, el patrón para el token **numero** coincide con 0 y con 1, pero es en extremo importante para el generador de código saber qué lexema se encontró en el programa fuente Ejemplo 3.2: Los nombres de los tokens y los valores de atributo asociados para la siguiente instrucción en Fortran:

```
E = M * C ** 2
```

se escriben a continuación como una secuencia de pares.

<id, apuntador a la entrada en la tabla de símbolos para E>  
<asigna-op>  
<id, apuntador a la entrada en la tabla de símbolos para M>  
<mult-op>  
<id, apuntador a la entrada en la tabla de símbolos para C>  
<exp-op>  
<numero, valor entero 2>

### Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente. Por ejemplo, si la cadena `fi` se encuentra por primera vez en un programa en C en el siguiente contexto:

```
fi ( a == f(x)) ...
```

un analizador léxico no puede saber si `fi` es una palabra clave `if` mal escrita, o un identificador de una función no declarada. Como `fi` es un lexema válido para el token `id`, el analizador léxico debe regresar el token `id` al analizador sintáctico y dejar que alguna otra fase del compilador (quizá el analizador sintáctico en este caso) mande un error debido a la transposición de las letras

### Uso de búfer en la entrada

Antes de hablar sobre el problema de reconocer lexemas en la entrada, vamos a examinar algunas formas en las que puede agilizarse la simple pero importante tarea de leer el programa fuente. Esta tarea se dificulta debido a que a menudo tenemos que buscar uno o más caracteres más allá del siguiente lexema para poder estar seguros de que tenemos el lexema correcto

### Pares de búferes

Debido al tiempo requerido para procesar caracteres y al extenso número de caracteres que se deben procesar durante la compilación de un programa fuente extenso, se han desarrollado técnicas especializadas de uso de búferes para reducir la cantidad de sobrecarga requerida en el procesamiento de un solo carácter de entrada.

### Centinelas

Si utilizamos el esquema en la forma descrita, debemos verificar, cada vez que movemos el apuntador avance, que no nos hayamos salido de uno de los búferes; si esto pasa, entonces también debemos recargar el otro búfer. Así, por cada lectura de caracteres hacemos dos pruebas: una para el final del búfer y la otra para determinar qué carácter se lee (esta última puede ser una bifurcación de varias vías).

**Especificación de los tokens** Las expresiones regulares son una notación importante para especificar patrones de lexemas.

Aunque no pueden expresar todos los patrones posibles, son muy efectivas para especificar los tipos de patrones que en realidad necesitamos para los tokens.

### Cadenas y lenguajes

Un alfabeto es un conjunto finito de símbolos. Algunos ejemplos típicos de símbolos son las letras, los dígitos y los signos de puntuación. El conjunto  $\{0, 1\}$  es el alfabeto binario. ASCII es un ejemplo importante de un alfabeto; se utiliza en muchos sistemas de software.

```
switch ( *avance++) {
case eof :
if (avance está al final del primer búfer ) {
    recargar el segundo búfer;
    avance = inicio del segundo búfer;
}
else if (avance está al final del segundo búfer ) {
    recargar el primer búfer;
    avance = inicio del primer búfer;
}
else /* eof dentro de un búfer marca el final de la entrada */
    terminar el análisis léxico;
break;
```

Casos para los demás caracteres  
}

que incluye aproximadamente 10 000 caracteres de los alfabetos alrededor del mundo, es otro ejemplo importante de un alfabeto.

Una cadena sobre un alfabeto es una secuencia finita de símbolos que se extraen de ese alfabeto. En la teoría del lenguaje, los términos “oración” y “palabra” a menudo se utilizan como sinónimos de “cadena”.

### Operaciones en los lenguajes

En el análisis léxico, las operaciones más importantes en los lenguajes son la unión, la concatenación y la cerradura, las cuales se definen de manera formal en la figura 3.6. La unión es la operación familiar que se hace con los conjuntos.

OPERACIÓN	DEFINICIÓN Y NOTACIÓN
<i>Unión de <math>L</math> y <math>M</math></i>	$L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$
<i>Concatenación de <math>L</math> y <math>M</math></i>	$LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$
<i>Cerradura de Kleene de <math>L</math></i>	$L^* = \bigcup_{i=0}^{\infty} L^i$
<i>Cerradura positivo de <math>L</math></i>	$L^+ = \bigcup_{i=1}^{\infty} L^i$

### Expresiones regulares

Suponga que deseamos describir el conjunto de identificadores válidos de C. Es casi el mismo lenguaje descrito en el punto (5) anterior; la única diferencia es que se incluye el guión bajo entre las letras.

establece de manera que represente a cualquier dígito, entonces podríamos describir el lenguaje de los identificadores de C mediante lo siguiente:

$\text{letra\_}(\text{letra\_} \mid \text{dígito})^*$

La barra vertical de la expresión anterior significa la unión, los paréntesis se utilizan para agrupar las subexpresiones, el asterisco indica “cero o más ocurrencias de”, y la yuxtaposición de  $\text{letra\_}$  con el resto de la expresión indica la concatenación

### Definiciones regulares

Por conveniencia de notación, tal vez sea conveniente poner nombres a ciertas expresiones regulares, y utilizarlos en las expresiones subsiguientes, como si los nombres fueran símbolos por sí mismos. Si  $\Sigma$  es un alfabeto de símbolos básicos, entonces una definición regular es una secuencia de definiciones de la forma:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

en donde:

1. Cada  $d_i$  es un nuevo símbolo, que no está en  $\Sigma$  y no es el mismo que cualquier otro  $d$ .

2. Cada  $r_i$  es una expresión regular sobre el alfabeto  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ .

Al restringir  $r_i$  a  $\Sigma$  y a las  $d$ s definidas con anterioridad, evitamos las definiciones recursivas y podemos construir una expresión regular sobre  $\Sigma$  solamente, para cada  $r_i$ . Para ello, primero sustituimos los usos de  $d_1$  en  $r_2$  (que no puede usar ninguna de las  $d$ s, excepto  $d_1$ ),

### Extensiones de las expresiones regulares

Desde que Kleene introdujo las expresiones regulares con los operadores básicos para la unión, la concatenación y la cerradura de Kleene en la década de 1950, se han agregado muchas extensiones a las expresiones regulares para mejorar su habilidad al especificar los patrones de cadenas. Una o más instancias. El operador unario postfijo  $+$  representa la cerradura positiva de una expresión regular y su lenguaje. Es decir, si  $r$  es una expresión regular, entonces  $(r)^+$  denota el lenguaje Cero o una instancia. El operador unario postfijo  $?$  significa “cero o una ocurrencia”. Es decir,  $r?$  es equivalente a  $r \mid \epsilon$ , o dicho de otra forma,  $L(r?) = L(r) \cup \{\epsilon\}$ . El operador  $?$  tiene la misma precedencia y asociatividad que  $*$  y  $+$ . Clases de caracteres. Una expresión regular  $a_1 a_2 \dots a_n$ , en donde las  $a_i$ s son cada una símbolos del alfabeto, puede sustituirse mediante la abreviación  $[a_1 a_2 \dots a_n]$ .

## Reconocimiento de tokens

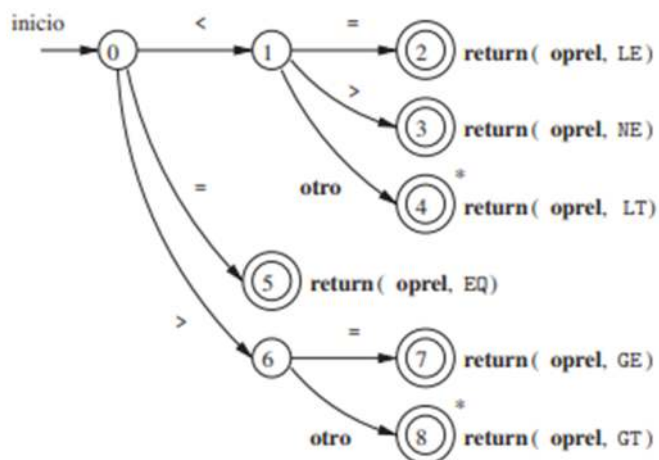
En la sección anterior, aprendimos a expresar los patrones usando expresiones regulares. Ahora debemos estudiar cómo tomar todos los patrones para todos los tokens necesarios y construir una pieza de código para examinar la cadena de entrada y buscar un prefijo que sea un lexema

```
instr → if expr then instr
| if expr then instr else instr
|
expr → term oprel term
| term
term → id
| numero
```

LEXEMAS	NOMBRE DEL TOKEN	VALOR DEL ATRIBUTO
Cualquier <i>ws</i>	-	-
if	if	-
Then	then	-
else	else	-
Cualquier <i>id</i>	id	Apuntador a una entrada en la tabla
Cualquier <i>numero</i>	numero	Apuntador a una entrada en la tabla
<	oprel	LT
<=	oprel	LE
=	oprel	EQ
<>	oprel	NE
>	oprel	GT
>=	oprel	GE

## Diagramas de transición de estados

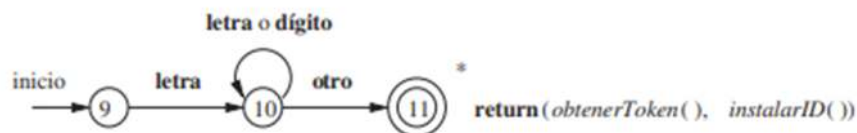
Como paso intermedio en la construcción de un analizador léxico, primero convertimos los patrones en diagramas de flujo estilizados, a los cuales se les llama “diagramas de transición de estados”. En esta sección, realizaremos la conversión de los patrones de expresiones regulares a los diagramas de transición de estados en forma manual, pero en la sección 3.6 veremos que hay una forma mecánica de construir estos diagramas, a partir de colecciones de expresiones regulares



## Reconocimiento de las palabras reservadas

y los identificadores

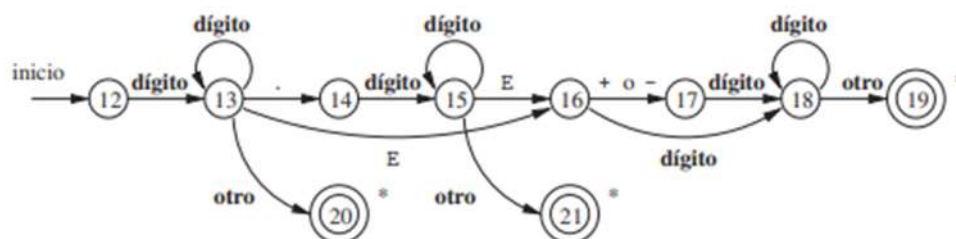
El reconocimiento de las palabras reservadas y los identificadores presenta un problema. Por lo general, las palabras clave como `if` o `then` son reservadas (como en nuestro bosquejo), por lo que no son identificadores, aun cuando lo parecen. Así, aunque por lo general usamos un diagrama de transición de estados como el de la figura 3.14 para buscar lexemas de identificadores, este diagrama también reconocerá las palabras clave `if`, `then` y `else` de nuestro bosquejo.



Finalización del bosquejo

El diagrama de transición para los tokens `id` que vimos en la figura 3.14 tiene una estructura simple. Empezando en el estado 9, comprueba que el lexema empiece con una letra y que pase al estado 10 en caso de ser así. Permanecemos en el estado 10 siempre y cuando la entrada contenga letras y dígitos.

El diagrama de transición de estados para el token `numero` se muestra en la figura 3.16, y es hasta ahora el diagrama más complejo que hemos visto. Empezando en el estado 12, si vemos un dígito pasamos al estado 13. En ese estado podemos leer cualquier número de dígitos



Arquitectura de un analizador léxico basado en diagramas de transición de estados

Hay varias formas en las que pueden utilizarse los diagramas de transición de estados para construir un analizador léxico. Sin importar la estrategia general, cada estado se representa mediante una pieza de código. Podemos imaginar una variable estado que contiene el número del estado actual para un diagrama de transición de estados.

Actividades semana 7 (Nov 2-6, 2020)

## Video 1

Ramirez Peña Carlos ivan

El análisis sintáctico LR es un método de análisis sintáctico de abajo hacia arriba y significa escaneo de izquierda a derecha con la derivación más a la derecha en reversa y  $k$  es el número de mirar hacia adelante tokens. Por supuesto, LR 0 y LR 1 son de gran interés para nosotros en sentido práctico, los analizadores LR, por supuesto, son importantes, porque se pueden generar automáticamente usando generadores de analizadores sintácticos y son un subconjunto de LR sin contexto. Las gramáticas son un subconjunto de gramáticas libres de contexto, para las cuales se pueden construir tales analizadores. Entonces, es fácil escribir gramática LR y es por eso que hoy en día son muy populares. Entonces, veamos el generador de analizador sintáctico, el generador es un dispositivo muy simple que toma gramática como entrada y genera una tabla de análisis sintáctica llamada tabla de análisis sintáctico LR. Y la tabla del analizador encaja en otra caja, que contiene una pila y una rutina de controlador, todo este conjunto es el analizador. Entonces, aquí, por ejemplo, la rutina del controlador de pila y la tabla de análisis juntas hacen el analizador, toma el programa como entrada y entrega como salida posiblemente una sintaxis libre o algo más. Entonces, echemos un vistazo a la operación del analizador, por lo que para entenderlo necesitamos saber exactamente qué es una configuración del analizador LR. La configuración tiene dos partes, una es la pila, la otra es la entrada no gastada o no utilizada y para comenzar con la pila como solo el símbolo de inicio o el estado inicial del analizador y la entrada no expandida como la totalidad entrada, sabe que terminó con un final de archivo o una marca de dólar. Entonces, en algún lugar en el medio, una configuración consistirá en varios estados, entremezclados con símbolos gramaticales y luego el resto de la entrada. La tabla de análisis es un poco más compleja tiene dos partes, la parte de acción y la parte GOTO. La parte de acción tiene cuatro tipos de entrada, cambio, reducción, aceptación y error, la tabla GOTO se utiliza para proporcionar la siguiente información de estado que es realmente necesaria después de un movimiento de reducción.

Entonces, aquí está el algoritmo de análisis. Entonces, antes de eso, miremos esta tabla del analizador para comprender la acción y vayamos a la entrada 03:38 que es un poco mejor. Entonces, la tabla del analizador está indexada por los números de estado en un lado y en el otro lado para la tabla de acción está indexada por los tokens, presente en la entrada y para la tabla de ir a la tabla está indexada por los no terminales presentes en la gramática



```

int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}

```

Samples of static semantic checks in *main*

- Types of *p* and return type of *dot\_prod* match
- Number and type of the parameters of *dot\_prod* are the same in both its declaration and use
- *p* is declared before use, same for *a* and *b*

Busca la parte de acción, si la parte de acción dice shift *p*, entonces presiona a *yp* en la pila en ese orden. Y el puntero de entrada avanza para recoger el siguiente símbolo de entrada si la parte de acción dice reducir por una producción, entonces este es el número que indiqué en la tabla del analizador sintáctico. Salta 2 en símbolos alfa de la pila, la razón 06:18 por la que sacamos 2 veces la longitud alfa de alfa es el número de símbolos en el lado derecho 06:25 de la producción. Y también tenemos el símbolo de estado entre mezcla, por lo tanto, necesitamos eliminar los símbolos alfa. El estado *S* prime está expuesto, ahora el lado izquierdo *A* aquí y el GOTO off *S* prime coma *A* o empujado a la pila, así que esta es una forma en que se usa la tabla GOTO, si la acción es aceptar entonces sale del bucle infinito, de lo contrario hay un error y se llama a la rutina de recuperación de errores. Entonces, echemos un vistazo a un ejemplo para entender todas las operaciones que acabo de describir, la pila contiene 0, el número de estado y la entrada es acbbac, el analizador dice shift. Entonces, esta es la tabla de análisis que tenemos en mente y estas son las producciones que estamos usando. De hecho, he escrito todas las producciones que son necesarias para la reducción, etcétera, en esta diapositiva. Entonces, para

```

int dot_product(int a[], int b[]) {...}

1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};
2 int b[10]={1,2,3,4,5,6,7,8,9,10};
3 printf("%d", dot_product(b));
4 printf("%d", dot_product(a,b,a));
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}

```

In function 'main':

```

error in 3: too few arguments to fn 'dot_product'
error in 4: too many arguments to fn 'dot_product'
error in 5: incompatible types in assignment
warning in 5: format '%d' expects type 'int', but
              argument 2 has type 'int *'

```

Un  
se  
la  
del

DFA cuyos  
estados también  
están enviando a  
pila, por lo que  
este es el estado  
analizador y el  
DFA. El estado

en la parte superior nos dice si es hora de reducción o un cambio, Consideremos una derivación más derecha S deriva en 0 o en muchos pasos  $\phi B t$ , que a su vez deriva  $\phi \beta t$ , en otras palabras, en el último paso B a  $\beta$  es la producción que se ha aplicado.

Entonces, la idea básica en una gramática LR es, deberíamos poder determinar el identificador de manera única, por lo que  $\beta$  es el identificador aquí B a  $\beta$  es la producción. Por lo tanto, deberíamos poder mirar los primeros símbolos de caso de esta t en cualquier derivación de la gramática y determinar qué producción se aplicó en ese punto en particular.

Entonces, se dice que una gramática es LR k, si para cualquier cadena de entrada en cada paso de cualquier derivación del extremo derecho, el identificador  $\beta$  se puede detectar examinando la cadena  $\phi \beta$  y escaneando como máximo los primeros k símbolos de la entrada no utilizada cadena t.

```

int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}

main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}

```

Samples of dynamic semantic checks in *dot\_prod*

- Value of *i* does not exceed the declared range of arrays *x* and *y* (both lower and upper)
- There are no overflows during the operations of "\*" and "+" in *d += x[i]\*y[i]*

Entonces, está phi beta está en la pila, así que si retrocedemos 1 paso. Entonces, este es el contenido de la pila y esto es exactamente lo que queremos decir con phi beta, por supuesto, los números de estado son adicionales. Entonces, el autómata de estado finito cuyos estados han sido rastreados por la pila nos da un método para examinar la cadena phi beta y al observar los símbolos de un primer caso de ty mirar la parte superior del estado de la pila, el analizador sintáctico LR podrá determinar en qué producción se utilizó en este punto, si es una reducción, de lo contrario determinará que es una acción de cambio. Entonces, aquí hay un ejemplo, la gramática es ambigua  $S \rightarrow E$ ,  $E \rightarrow E + E$  o  $E \rightarrow E * E$  o  $\text{id}$ , donde queremos mostrar que esto no es LR 2 que es incluso con  $k$  look antes de 2, no podremos determinar el identificador de forma única. Si hay dos derivaciones que he mostrado aquí,  $S$  deriva  $E$ ,  $E$  deriva  $E + E$ ,  $E + E$  deriva  $E + E * E$ , entonces obtenemos  $E + E * \text{id}$   $E + \text{id} * \text{id}$   $\text{id} + \text{id} * \text{id}$ . Entonces, esta es una derivación más a la derecha, otra derivación  $S \rightarrow E$   $E \rightarrow E * E$  Entonces, en lugar de  $E + E$  tenemos  $E * E$  luego  $E * \text{id}$ , entonces esto se convierte en  $E + E * \text{id}$   $E + \text{id} * \text{id}$   $\text{id} + \text{id} * \text{id}$ . Entonces, el mismo ID de cadena más el ID de estrella tiene dos derivaciones más a la derecha porque, esto es ambiguo, pero el punto que queremos, ya sabes, el énfasis aquí es que no podemos determinar el identificador de manera única. Entonces, considere este paso 6 primo y un paso 6, por lo que el identificador aquí es  $\text{id}$ , el identificador aquí también es  $\text{id}$ , el mismo símbolo primer símbolo y la producción utilizada, por supuesto, es  $E \rightarrow \text{id}$

- Let  $G = (N, T, P, S)$  be a CFG and let  $V = N \cup T$ .
- Every symbol  $X$  of  $V$  has associated with it a set of *attributes* (denoted by  $X.a$ ,  $X.b$ , etc.)
- Two types of attributes: *inherited* (denoted by  $AI(X)$ ) and *synthesized* (denoted by  $AS(X)$ )
- Each attribute takes values from a specified domain (finite or infinite), which is its *type*
  - Typical domains of attributes are, integers, reals, characters, strings, booleans, structures, etc.
  - New domains can be constructed from given domains by mathematical operations such as *cross product*, *map*, etc.
  - *array*: a map,  $\mathcal{N} \rightarrow \mathcal{D}$ , where,  $\mathcal{N}$  and  $\mathcal{D}$  are domains of natural numbers and the given objects, respectively
  - *structure*: a cross-product,  $A_1 \times A_2 \times \dots \times A_n$ , where  $n$  is the number of fields in the structure, and  $A_i$  is the domain of the  $i^{\text{th}}$  field

Entonces, tenemos E más id star id como la forma sentencial en un paso hacia atrás, aquí también id E más id star id es la forma sentencial cuando atravesamos un paso hacia atrás, la reducción desde id a E nos da esta forma enunciativa. El siguiente paso 5 primo y el correspondiente paso 5, nuevamente el identificador es id y la producción, que se ha utilizado nuevamente es E a id y en ambos casos obtenemos la forma sentencial E más E star id E más el ID de estrella E, que son idénticos. Luego, las superficies de diferencia para el paso 4 y el paso 4 priman, el identificador en esta derivación es id mientras que, el identificador en esta derivación es E más E. Entonces, la mirada hacia adelante en este caso, sabes que la entrada no expandida es star id, aquí también la entrada no expandida es star id, así que, si miramos dos símbolos, estaríamos mirando star id en ambos casos.

Entonces, esa es la misma cadena de anticipación, pero al mirar en la cadena no podemos determinar de manera única si el analizador LR debe tomar E más E como el identificador o E más E como el identificador o identificador como el mango. Entonces, esta es precisamente la ambigüedad y una gramática no es LR 2 en este caso, por lo que el identificador no se puede determinar usando la mirada hacia adelante y, por supuesto, la derivación. Entonces, debido a que el contenido de la pila es idéntico, por lo que puede ver que E más E es la pila en ambos casos, la entrada no gastada es star id en ambos casos.

Una vez que derivamos todos los símbolos terminales de T agregamos lo que T derive y luego retenemos esta E más paréntesis, digamos que T deriva i d. Entonces, si agregamos id paréntesis y hash a E más paréntesis, obtenemos una forma de oración correcta

El conjunto forma un lenguaje regular. Entonces, el DFA de este lenguaje puede detectar identificadores durante el análisis de LR, por lo que veremos que muy pronto, el punto es que el DFA alcanza un, llamado estado de reducción e indica que el prefijo viable no puede crecer más. Entonces; eso significa que hay una reducción que es necesaria en este punto, les mostraré qué es exactamente un estado de reducción después de esta diapositiva. Este tipo de DFA puede ser construido por el compilador

usando la gramática y vamos a discutir ese procedimiento un poco más adelante. Todos los analizadores LR tienen un DFA incorporado en ellos, este es el corazón de un analizador LR realmente, así que para hacer eso, construimos una gramática aumentada y si  $S$  es el símbolo de inicio de  $G$ , entonces  $G$  primo contiene todas las producciones de  $G$ , junto con una nueva producción  $S$  prime que va a  $S$

.

- The following CFG  
 $S \rightarrow A B C, A \rightarrow aA \mid a, B \rightarrow bB \mid b, C \rightarrow cC \mid c$   
generates:  $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$
- We define an AG (attribute grammar) based on this CFG to generate  $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
  - $AS(S) = \{equal \uparrow: \{T, F\}\}$
  - $AS(A) = AS(B) = AS(C) = \{count \uparrow: integer\}$

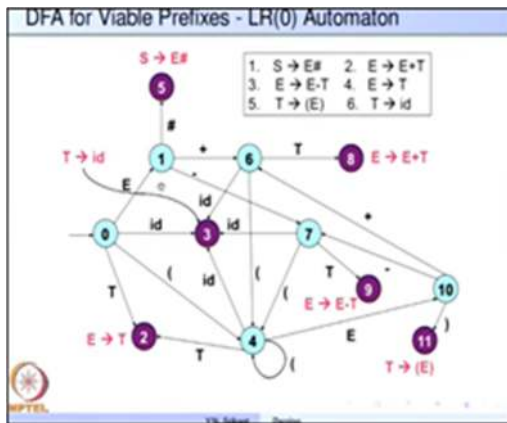
Nombre de integrante:Rodriguez Perez Luis diego

## Video 2

conferencia sobre análisis de sintaxis, parte 6. Así que, hoy continuaremos con el fondo

Estudio de análisis de lo que son las gramáticas de S L R 1, L R 1, etcétera, etcétera. Para hacer un poco de recapitulación. Entonces, discutimos el concepto de los prefijos viables elementos válidos. Y entonces también discutimos el procedimiento para construir el autómata L R 0, aquí está el ejemplo I te mostró la última vez. Y el método de construcción es muy simple, sabes que para empezar con el artículo  $S$  prime al punto  $S$  es el único al que vamos a ir y luego le damos el cierre. Así que, eso le da un conjunto inicial de elementos y ahora aplicamos continuamente la función de ir a y seguir incluyendo tantos grupos de artículos como sea posible en la colección  $C$ . Y cuando no podamos añadir más artículos que detengamos. Así que, cada conjunto de la colección anterior es un estado de la LR 0 DFA, y este DFA reconoce prefijos viables. Así que, cada estado del DFA contiene sólo elementos que son válidos para un prefijo viable particular o un conjunto de

prefijos viables. Entonces, déjame explicarte cómo exactamente el análisis de LR(0) ocurre con respecto a este DFA en particular. Entonces, consideremos la simple cadena  $id$  que se deriva de esta gramática. Entonces, nosotros empezamos con  $S$  ir a  $E$  hash aplicar esta producción y luego cuando aplicamos la producción  $E$  ir a la  $T$  y finalmente, aplicamos la producción  $T$  que va a la  $id$ . Así, obtenemos la cadena  $id$ .



Entonces, veamos cómo se pasa usando este autómata LR(0). Así que, como siempre empezamos con el estado 0, que es el estado inicial y luego en la entrada vamos al número de estado 3. Así que, recuerde el estado 0 en la pila ahora que me empujan a la pila cuando vemos  $y$  te lleva al estado 3 que es de nuevo empujar hacia abajo a la pila. Así que, en lugar de tres dice que es un estado reducido. Así que, y la reducción es por la producción  $T$  va a  $id$ . Así que, la implicación de esto sería hacer estallar el símbolo superior de la pila. Y como el estado también está incluido, hacemos estallar los símbolos de la pila y eso realmente expone

el estado 0 en la pila ahora el no termina en el lado izquierdo es  $t$ . Por lo tanto, aplicamos el ir a la función en el estado 0 y eso nos lleva al estado número 2. Así que, ahora, hemos empujado la  $T$  no terminal en la pila y el estado número 2 también está en la pila ahora dice reducir de nuevo de  $E$  a  $T$ . Así que, sacamos el estado 2 y la  $T$  no terminal de la pila de nuevo obtenemos el estado 0 como la parte superior de la pila. Y ahora, ya que el no terminal del lado izquierdo es  $E$ , aplicamos la función "ir a" en el estado 0 y la no terminal  $E$  que nos lleva a el estado número uno. Ahora, el siguiente símbolo de entrada es el hash porque este es un estado de cambio no es un estado de reducción y en el hash pasa al estado 5 donde una reducción por  $S$  va a  $E$  hash está indicado.

Así que, volvemos a poner los artículos para la pila y tú Conocemos este estado 0 y empujamos la no terminal  $S$  a la pila y eso es una aceptación de la excepción. Así que, así es como se pasa la cadena terminal  $id$ . Así que, vamos a considerar una terminal un poco más complicada, ya sabes, cadena  $id$  más  $id$ . Así que,  $id$  más  $id$  es hash derivar de esta gramática usando  $S$  va a  $E$  hash luego  $E$  va a  $E$  más  $T$

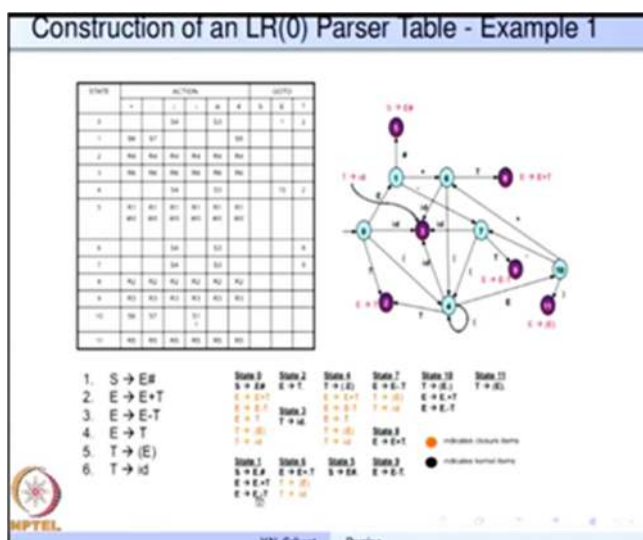
Esta  $T$  irá a la  $id$ , esta  $E$  irá a la  $T$  y luego a la  $id$ . Así que, así es como yo más Si el hash se deriva de nuevo, empezamos de 0 y pasamos al estado 3, que indica una

reducción. volvemos y en la T pasamos al estado 2 que de nuevo indica una reducción. Así que, volvemos al estado 0 símbolos pop y luego ir al estado 1 en E. Así que, ahora, el plus nos lleva al estado 6 y nos lleva al estado 3 de nuevo hay una reducción. Así que, recuerda que el 3 y el 6 están en la pila. Así que, sacamos a estos dos que sabes que lo siento y tres que son de la pila para exponer la no terminal 6 y 6 en la no terminal T nos lleva al estado 8. Así que, aquí tenemos en la pila E más E 0 E uno más 6 T y ocho que indica una reducción de E a E más T. Así que, volvemos al estado 0 y en E volvemos a volver al estado 1. Entonces, y en el hachís hace que vaya al estado 5 y aceptamos la cuerda. Así es como se hace el análisis usando este autómata, por supuesto, todo esto se hace más fácil... una vez que tengamos la mesa en sí. entonces, aquí está la mesa. Así que, antes de ir a esta mesa debo explicar cómo exactamente La mesa está llena. Así que, las acciones de cambio y reducción en el estado están llenas como ahora discutimos si un estado contiene un elemento de la forma un ir a punto alfa que se llama un reducir artículo. Entonces una reducción de a a alfa es la acción en ese estado si no hay reducción artículos en un estado. Entonces es un estado de desplazamiento y el desplazamiento es la acción apropiada podría haber cambio reducir los conflictos o reducir reducir los conflictos. Entonces, ¿cuándo ocurre eso hay un elemento de desplazamiento y también hay un elemento de reducción en el mismo estado están llenos como ahora lo discutimos si un estado contiene un elemento de la forma unir a punto alfa que se llama un reducir artículo. Entonces una reducción de a a alfa es la acción en ese estado si no hay reducción artículos en un estado. Entonces es un estado de desplazamiento y el desplazamiento es la acción apropiada

podría haber cambio reducir los conflictos o reducir reducir los conflictos. Entonces, ¿cuándo ocurre eso hay un elemento de desplazamiento y también hay un elemento de reducción en el mismo estado o hay más de un artículo reducido en el mismo estado. Así que, es que sabes que es normal tener más un elemento de turno en un estado. Así que no hay duda de que tener un conflicto de turnos en absoluto. Así que, si no hay cambios, reduce o reduce los conflictos en cualquier estado de este DFA entonces la gramática se establece para ser L R 0, de lo contrario no es L R 0. Entonces, esta es la, ya sabes, tabla para el conjunto de elementos que ya hemos construido. Entonces, déjame mostrarte el procedimiento para llenarlo. Así que, en realidad miramos los elementos. Así que, tenemos el estado 0 asumimos que no estamos llenos de nada aquí. Así que, toda la mesa está vacía. Así que, en el estado 0 y veamos en qué elementos se indica un cambio en un símbolo de terminal porque el no terminal después del punto realmente indica un ir a. Así que, sólo los símbolos terminales después del punto indican un cambio en la pila. Por lo tanto, hay un paréntesis y también estoy yo. Así que, en el estado 0 para el paréntesis izquierdo hay un cambio y para el i d hay otro turno. Así que, cuando avanzamos el punto después del paréntesis izquierdo ya sabes. Entonces, el estado en el que nos metemos sería, ya sabes, el estado número 4. Así que, aquí es correcto.



Así que, el punto E del paréntesis y luego el otro en i d sabes 0 en i d va al estado T yendo al punto i d. Entonces, ese es el estado número 3. Así que, ahora, el ir a la parte también puede ser llenado para este estado en particular. Por lo tanto, hay una E no terminal después del punto en estos artículos y hay una T no terminal después del punto en este artículo en particular y otros dos que ya hemos considerado. Así que, en la terminal E va a un estado en el que contiene los elementos que van de E a E más el punto T y ese sería el estado número 1. Así que, esto es lo que sabes. Así que, el punto E tiene y luego el punto E más T y luego el punto E menos T. Así que, por supuesto, S va al punto E hash es también un artículo con E en el lado derecho y después del punto. Así que, estos 3 artículos están en el estado 1. Así que, tenemos un goto de 0 en el no terminal E como estado número uno. Así que, y de manera similar 0 ir a de 0 en la terminal T es el número de estado 2 que es esta cosa en particular, ya sabes. Así que, yendo al punto T dará el artículo E va al punto T. Así que, esta es la manera en que vamos a llenar todas las entradas en esta mesa en particular. Así que, ahora, de nuevo si consideras la misma cadena que yo tenía y con nosotros con la ayuda de estos esta mesa y el conjunto de artículos nos permite asegurarnos que entendamos cómo se pasa. Así que, empezamos desde el estado 0 y en i d dice shift y pasar al estado 3. Así que, ese es el estado número tres aquí y vemos que es una reducción artículos que es la razón por la que este estado indica que sabes que este estado indica una reducción por el no por la producción T va a i d y entonces lo sabes; eso significa, que debemos...y que la pila se reviente. Entonces, se elimina el estado 3 y se vuelve a poner el 0 en la pila.



Entonces, la T no terminal del estado nos lleva al estado número 2. Así que, eso es aquí. Entonces, nosotros habíamos empujado la no terminal T y el estado número 2 a la pila. Así que, así que han pasado al estado número 2 aquí y sólo tiene un artículo reducido. Entonces, hay una reducción de E a T que nos devuelve al estado 0 y el estado 0 en E toma nosotros para el estado 1; qué es este estado en particular y el estado 1 contiene una ya sabes S va a E punto hash E va a E punto



más T y E va a E punto menos T, y luego leemos hash we ir al estado número 5. Así que este es el estado número 5, que es un estado de aceptación. Así que., en el estado número 5 la fila entera dice que acepta cualquier símbolo. Así que, así es como la cadena se pasa. Así que, y también puedes ver que hay exactamente una entrada en cada uno de estos sabes que las tragamonedas aquí por supuesto, acepta no es nada, pero un comentario ya sabes. Así que, la reducción

por uno y el dice que es un estado de aceptación. Por lo tanto, no es que haya dos entradas de reducir la naturaleza o algo así. Así que, esta es una gramática que definitivamente LR 0, porque no hay conflictos en ningún estado.

Entonces, vayamos más allá, tomemos otro ejemplo de cómo se construye el autómata LR 0. Así que, esta es la gramática, este es el autómata, estudiaremos la forma de construirlo también.

Entonces, empezamos desde el estado 0. Entonces, esta es la gramática aumentada  $S \rightarrow s$  yendo al punto  $S \rightarrow s$  va a  $S \rightarrow s$  primo va a  $S \rightarrow s$  primo  $S \rightarrow s$  va a  $A S$  y  $S \rightarrow s$  va a  $c$ . Y la inicial El elemento con el que empezamos es siempre  $S \rightarrow s$  yendo a punto  $s$ . Así que, aplicando el cierre añadimos,

estos dos elementos  $S \rightarrow s$  van a punto  $a$   $A S$  y  $S \rightarrow s$  van a punto  $c$  no hay más elementos a se añadirá. Así que, una vez que se construya el estado 0, ya sabes que se construye el estado goto para esto el artículo  $S \rightarrow s$  primo va a punto  $S$  se convierte en  $S \rightarrow s$  primo va a  $S$  punto no más artículos pueden ser añadido aquí que da un estado 1. Así que, cuando avanzamos el punto después de la  $a$  aquí.

Entonces, nosotros obtenemos un punto  $a A S$ . Así que, eso nos da este artículo y ahora el cierre incluye todos estos artículos sabes que empieza con una  $y$  luego hay un elemento con  $S$  después del punto. Entonces, los artículos con  $S$  también se añaden de nuevo. Así que, recuerden que hay dos elementos  $S$  aquí  $S$  yendo a un punto  $a S$  y

$S$  va a poner un punto en la  $A S$ . Pero los puntos están en diferentes posiciones. Así que. No hay nada incorporado en esto. Así que, ahora,  $S$  va al punto  $C$  y nos da el estado 3. Así que, hemos cubierto todos los artículos en el estado 0 esto no nos da más aquí nos da  $S$  va a un punto  $S$  y luego añadimos estos dos elementos para la operación de cierre y luego esto nos da un punto  $a$  de ir a cenizas  $b$  punto  $a$ . Así que, ese es el estado y luego un punto  $p$  de ir a  $S$  que sería este estado con estos artículos de cierre estos artículos de cierre y  $S$  yendo a un punto  $a$

$S$  es el mismo estado número 2 y  $S$  va al punto  $c$  ya está cubierto es el número de estado tres. Así que esto nos da  $S$  yendo a  $A A S$  punto que es el estado número 5 y este ítem nos da  $S$  yendo a un punto  $a A S$  que ya es el estado número 2.

Así que no se añade ningún estado nuevo aquí y esto, por supuesto,  $S$  yendo al punto  $c$  de nuevo no nos da nada más extra. Así que, este estado de nuevo no nos

da ningún estado b punto a punto b a nos da un va a b un punto de nuevo este es un estado reducido y no se puede añadir nada más a esto. Así que, esto por supuesto, nos da un poco más, nos da un estado a ir a S b punto y b va a b punto a es el estado número diez con elementos de cierre extra aquí y esto y esto no nos da ningún estado extra b yendo a S punto es un estado que el estado número doce.

Entonces, este punto b a b que va a b punto a nos da b que va a b punto y otros no generan más estados nuevos. Así que, puedes ver que esta gramática en particular no tiene ningún reducir o reducir los conflictos en cualquier estado hay exactamente un elemento de reducir el estado en un estado siempre que hay 1. Así que, y no hay ningún elemento de cambio también.

Así que, esta es una gramática que es LR(0) aquí es la tabla que conoces para ello. Y aquí están las tablas; los conjuntos de elementos; la gramática y el DFA, todo junto. Así que, tomemos un estado y entendamos de nuevo cómo llenar la tabla. Entonces, el estado número 2.

Entonces, de aquí el estado número 2 en b. Entonces, vamos al estado número 6 a la derecha. Entonces, b punto a. Entonces, El estado número 6 está aquí b punto a. Así que el estado número 2 dice que el turno 6 está a la derecha y luego allí. es una c aquí. Así que, en la c debería ir al estado número 3. Así que, la 2 en la c dice que vayo al número de estado

tres y en un por supuesto, esto sigue siendo usted sabe punto a S esto es un va a punto a A s. Por lo tanto, esto genera un estado que es el número de estado en sí mismo.

A Grammar that is not LR(0) - Example 1

<b>State 0</b> $S \rightarrow \cdot E$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot E - T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$	<b>State 2</b> $E \rightarrow T \cdot$	<b>State 5</b> $E \rightarrow E \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$	<b>State 8</b> $E \rightarrow E \cdot T$
<b>State 1</b> $S \rightarrow E \cdot$ $E \rightarrow E \cdot + T$ $E \rightarrow E \cdot - T$	<b>State 3</b> $T \rightarrow id \cdot$	<b>State 6</b> $E \rightarrow E \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$	<b>State 9</b> $T \rightarrow (E) \cdot$ $E \rightarrow E \cdot + T$ $E \rightarrow E \cdot - T$
<b>State 4</b> $T \rightarrow (E) \cdot$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot E - T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$	<b>State 7</b> $E \rightarrow E \cdot T$	<b>State 10</b> $T \rightarrow (E) \cdot$	

shift-reduce conflicts in state 1  
 indicates closure items  
 indicates kernel items

follow(S) = {E}, where E is EOF  
 Reduction on S, and shifts on + and -, will resolve the conflicts  
 This is similar to having an end marker such as #

Grammar is not LR(0), but is SLR(1)

Por lo tanto, permanece en el estado número 2 y la operación es el cambio. Así que, puedes ver la autodirigida arco aquí a la derecha con respecto a la parte de la mesa. Así que, 2 y S ya sabes. Entonces, 2 en S va al estado número 8. Así que la razón de estar ahí es un símbolo S después del punto a punto S B. Entonces, esto va al estado que contiene un ir a S punto b. Entonces, un ir a S punto b está aquí...el estado derecho número ocho y en el no terminal ocho va al estado número 4. Así

que, eso es porque tenemos S yendo a un punto a A S. Así que, esto nos da el estado número 4 que contiene

a a punto s. Así que esta es la forma en que llenamos toda la mesa y ya que hay exactamente una entrada en cada una de las ranuras esta gramática es de hecho LR 0.

Hasta ahora hemos visto ejemplos en los que la gramática es LR 0. Así que es hora de seguir adelante y encontrar fuera usted sabe si hay gramática que no son LR 0 en absoluto que viola el LR 0 propiedad en efecto. Por lo tanto, tomamos la vieja gramática con más y menos, pero recordamos la vieja La gramática tenía S yendo a E hash realmente quitamos el hash. Así que, ahora, es el momento de explicar por qué quitamos y qué pasa si lo quitamos. Así que, dejamos que se interprete

lo voy a explicar con más detalle otra vez. Pero tenemos este primer artículo y luego el cierre, luego los elementos S que van a punto E S que van a punto E más T y S E E que va al punto E menos T. Le dan aumento al estado número 1, que contiene d que va a E punto E va a E punto más T y E va a E punto menos T .Por lo tanto, es fácil ver que hay un elemento de reducción y hay dos elementos de desplazamiento a la derecha.

Entonces, este es el ítem reducido y estos dos son los ítems de desplazamiento. Por lo tanto, hay un desplazamiento reducir el conflicto en este estado particular. Así que, una vez que lleguemos a este estado particular en un símbolo de entrada particular. Sabes que no podemos decir si el estado debería reducir por la producción S va a E o cambia el siguiente artículo a la pila. Recuerde la tabla Cuando operamos con la mesa, el analizador no ve qué es exactamente el artículo. y luego decide la acción que sabes que es, hay exactamente una acción en cada uno de las tragaperras. Entonces, en 2 en a se convierte en S 2 etcétera. Así que, mientras que, para la reducción allí siempre tenía exactamente un artículo R 4 y éste era el S 7. Así que, en toda la fila que siempre lo hemos puesto es una reducción de 4 ver para el estado número 5 es siempre una reducción por producción en número 2. Y para el estado número 3 siempre es un reducir por la producción número 3 y así sucesivamente. Así que, en otras palabras, para el artículo de turno, ya sabes vemos lo que es exactamente un símbolo y lo empujamos a la pila, pero para el artículo reducido nosotros deberíamos ser capaz de decir que es una reducción de una producción particular sin siquiera molestarse

sobre el siguiente símbolo de entrada. Por lo tanto, esto es un inconveniente de la estrategia de análisis de LR 0 porque hacemos

no considerar que sabes qué es exactamente el siguiente símbolo de entrada para decir si es una reducción o un cambio. Así que, en este caso la gramática falla en la prueba LR pero esto es una reducción a la siguiente clase de gramática llamada S L

R 1 o simple L R 1 gramática. Así que, sucede que la siguiente parte superior es S L R 1. Así que, déjame mencionar lo que sucede allí.

Así que, en realidad calculamos lo que se conoce como el siguiente conjunto de la S no terminal cuando hay una reducción que calculamos el seguimiento del lado izquierdo no terminal que ocurre a un dólar. Así que, si queremos hacer una reducción, entonces lo hacemos sólo en dólares y en más y menos que hagamos un cambio esto realmente resuelve el conflicto y hace que la gramática siga adelante con la misma estrategia de análisis. Así que, hacer esto es similar a tener un marcador final como el hash que teníamos aquí. Así que, si tuviéramos un hash aquí S va a E hash entonces este estado no ha sido un estado de reducción, pero aún así ha sido un estado de cambio. Por lo tanto, el estado de reducción ha sido la siguiente que sabes que S va a E punto de hash. Así que, la suposición aquí es la entrada no importa qué error de sintaxis sabes que ocurre en la entrada siempre habrá 1

El símbolo del hash está llegando. Así que, si el símbolo ha desaparecido entonces los analizadores se detienen con un error en algún lugar en el medio, pero si el símbolo hash definitivamente existe entonces puede seguir adelante, ya sabes. Así que, definitivamente leerá eso y si hay un error proporciona un error, de lo contrario continúa. Entonces, el símbolo del hash hace que esta gramática L R 0 y quitando el símbolo de hash que es un marcador en la entrada hace que este no sea L R 0. Pero afortunadamente hay otra forma de romper este empate y es usar el LSRL Estrategia R 1. Si la gramática no es L R 0, tratamos de resolver los conflictos en el estado usando una mirada a la cabeza símbolo L R 0.

*Pues vemos cómo va más bien funcionando el análisis sintáctico Ir me refiero que más bien que va haciendo reducciones y derivaciones de nuestros tokens "L" es por el examen de la entrada de izquierda a derecha (en inglés, left-to right), la "R" por construir una derivación por la derecha (en inglés, rightmost derivation) en orden inverso, y la k por el número de símbolos de entrada de examen por anticipado utilizados para tomar las decisiones del análisis sintáctico. Cuando se omite, se asume que k es 1. El análisis sintáctico LR es atractivo por varias razones.*

## Video 3

Nombre de integrante: Hernández Ruiz Adrián Felipe

Analizadores LALR (1)

- ❖ Los analizadores sintácticos LR (1) tienen una gran cantidad de estados
  - Para C, muchos miles de estados.

- Un analizador SLR (1) (o LR (0) DFA) para C tendrá unos cientos de estados (con muchos conflictos).
- ❖ Los analizadores sintácticos LALR (1) tienen exactamente el mismo número de estados que los analizadores sintácticos SLR (1) para la misma gramática y se derivan de los analizadores sintácticos LR (1).
  - Los analizadores SLR (1) pueden tener muchos conflictos, pero los analizadores LALR(1) pueden tener muy pocos conflictos.
  - Si el analizador LR (1) no tuvo conflictos S-R, entonces el analizador LALR(1) derivado correspondiente tampoco tendrá ninguno.
  - Sin embargo, esto no es cierto con respecto a los conflictos R-R.
- ❖ Los analizadores sintácticos LALR(1) son tan compactos como los analizadores sintácticos SLR (1) y son casi tan potentes como los analizadores sintácticos LR (1).
- ❖ La mayoría de las gramáticas de los lenguajes de programación también son LALR(1), si son LR (1).

### LALR(1) Parser Construction - Example 1

Grammar  
 $S' \rightarrow S, S \rightarrow aSb, S \rightarrow \epsilon$

Grammar is LALR(1)

**State 0**  
 $S' \rightarrow .S, \$$   
 $S \rightarrow .aSb, \$$   
 $S \rightarrow ., \$$

**State 1**  
 $S' \rightarrow S, .\$$

**State 2**  
 $S \rightarrow a.Sb, \$$   
 $S \rightarrow .aSb, b$   
 $S \rightarrow ., b$

**State 3**  
 $S \rightarrow aS.b, \$$

**State 4**  
 $S \rightarrow aSb., b$   
 $S \rightarrow ., b$

**State 5**  
 $S \rightarrow aSb., \$$

**State 6**  
 $S \rightarrow aSb.b, b$

**State 7**  
 $S \rightarrow aSb., b$

	a	b	\$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		SS7		
57		R: S → aSb	R: S → aSb	

### LALR(1) Parser Construction - Example 1 (contd.)

LR(1) Parser Table

	a	b	\$	S
0	S24		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		SS		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		

LALR(1) Parser Table

	a	b	\$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		SS7		
57		R: S → aSb	R: S → aSb	

### Construcción de analizadores sintácticos LALR (1)

- ❖ La parte principal de los elementos LR (1) (la parte que sigue a la omisión del símbolo de anticipación) es la misma para varios estados de LR (1) (los símbolos de anticipación serán diferentes).
  - Fusionar los estados con el mismo núcleo, junto con los símbolos de anticipación, y cambiarles el nombre.
- ❖ Las partes ACTION y GOTO de la tabla del analizador se modificarán.
  - Fusionar las filas de la tabla del analizador correspondientes a los estados fusionados, reemplazando los nombres antiguos de los estados por los nuevos nombres correspondientes para los estados fusionados.

→ Por ejemplo, si los estados 2 y 4 se fusionan en un nuevo estado 36, todas las referencias a los estados 2, 4, 3 y 6 serán reemplazadas por 24, 24, 36 y 36, respectivamente.

- ❖ Los analizadores LALR (1) pueden realizar algunas reducciones más (pero no cambios) que un analizador LR (1) antes de detectar un error.

LR(1) Parser			LALR(1) Parser		
0	ab\$	shift	0	ab\$	shift
0 a 2	b\$	$S \rightarrow \epsilon$	0 a 24	b\$	$S \rightarrow \epsilon$
0 a 2 3	b\$	shift	0 a 24 36	b\$	shift
0 a 2 3 b 5	\$	$S \rightarrow aSb$	0 a 24 36 b 57	\$	$S \rightarrow aSb$
0 5 1	\$	accept	0 5 1	\$	accept
0	aa\$	shift	0	aa\$	shift
0 a 2	a\$	shift	0 a 24	a\$	shift
0 a 2 a 4	\$	error	0 a 24 a 24	\$	error
0	aab\$	shift	0	aab\$	shift
0 a 2	ab\$	shift	0 a 24	ab\$	shift
0 a 2 a 4	b\$	$S \rightarrow \epsilon$	0 a 24 a 24	b\$	$S \rightarrow \epsilon$
0 a 2 a 4 5 6	b\$	shift	0 a 24 a 24 36	b\$	shift
0 a 2 a 4 5 6 b 7	\$	error	0 a 24 a 24 36 b 57	\$	$S \rightarrow aSb$
			0 a 24 5 36	\$	error

#### Características de los analizadores LALR (1)

- ❖ Si un analizador LR (1) no tiene conflictos S-R, entonces el analizador LALR (1) derivado correspondiente tampoco tendrá ninguno.
  - Los estados del analizador LR (1) y LALR (1) tienen los mismos elementos principales (las búsquedas anticipadas pueden no ser las mismas).
- Si un estado  $s_1$  del analizador LALR (1) tiene un conflicto S-R, deben tener dos elementos  $[A \rightarrow \alpha., a]$  y  $[B \rightarrow \beta, a\gamma, b]$ .
- Uno de los estados  $s_1'$ , a partir del cual se genera  $s_1$ , debe tener los mismos elementos básicos que  $s_1$ .
- Si el artículo  $[A \rightarrow \alpha., a]$  está en  $s_1'$ , entonces  $s_1'$  también debe tener el elemento  $[B \rightarrow \beta, a\gamma, c]$  (la búsqueda anticipada no necesita ser b en  $s_1'$  puede ser b en algún otro estado, pero eso no nos interesa).
- Estos dos elementos en  $s_1'$  todavía crean un conflicto S-R en el analizador LR (1).
- Por lo tanto, la fusión de estados con un núcleo común nunca puede introducir un nuevo conflicto de S-R, porque el cambio depende sólo del núcleo, no de la anticipación.
- Sin embargo, la fusión de estados puede introducir un nuevo conflicto R-R en el analizador LALR (1) aunque el analizador LR (1) original no tuviera ninguno.

→ Tales gramáticas son raras en la práctica.

→ Aquí hay uno del libro de ALSU. Construya los juegos completos de elementos LR (1) como trabajo a domicilio:

$$S' \rightarrow S\$, S \rightarrow aAd \mid bBd \mid aBe \mid bAe \\ A \rightarrow c, B \rightarrow c$$

→ Dos estados contienen los elementos:

$$\{[A \rightarrow c., d]. [B \rightarrow c., e]\} \text{ y } \\ \{[A \rightarrow c., e]. [B \rightarrow c., d]\}$$

→ La fusión de estos dos estados produce el estado LALR (1):

$$\{[A \rightarrow c., d / e]. [B \rightarrow c., d / e]\}$$

→ Este estado LALR (1) tiene un conflicto de reducir-reducir

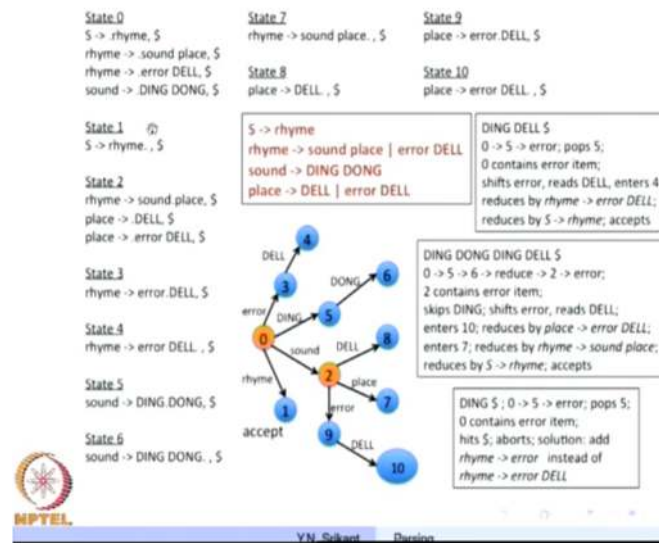
#### Recuperación de errores en analizadores LR: construcción del analizador

- ❖ El escritor del compilador identifica los principales no terminales, como los de programa, declaración, bloque, expresión, etc.
- ❖ Agrega a la gramática, producciones de error de la forma  $A \rightarrow error \alpha$ , Donde A es un importante no terminal y  $\alpha$  es una cadena adecuada de símbolos gramaticales (generalmente símbolos terminales), posiblemente vacía.
- ❖ Asocia una rutina de mensaje de error con cada producción de error.
- ❖ Construye un analizador LALR (1) para la nueva gramática con producciones de error.

#### Recuperación de errores en analizadores LR: funcionamiento del analizador

- ❖ Cuando el analizador encuentra un error, escanea la pila para encontrar el estado superior que contiene un elemento de error del formulario  $A \rightarrow .error \alpha$
- ❖ El analizador luego cambia un error de token como si hubiera ocurrido en la entrada.
- ❖ Si  $\alpha = \epsilon$ , reducido por  $A \rightarrow \epsilon$  e invoca la rutina de mensajes de error asociada con él.
- ❖ Si  $\alpha \neq \epsilon$ , descarta los símbolos de entrada hasta que encuentra un símbolo con el que el analizador puede continuar
- ❖ Reducción por  $A \rightarrow .error \alpha$  ocurre en el momento apropiado Ejemplo: Si la producción de error es  $A \rightarrow .error;$ , y procede como arriba.
- ❖ La recuperación de errores no es perfecta y el analizador puede abortar al final de la entrada.

## LR(1) Parser Error Recovery



## LEX and YACC

### YACC Example

```
%token DING DONG DELL
%start rhyme
%%
rhyme : ^sound place '\n'
      {printf("string valid\n"); exit(0);};
sound : DING DONG ;
place : DELL ;
%%
#include "lex.yy.c"

int yywrap() {return 1;}
yyerror( char* s)
{ printf("%s\n",s); }
main() {yyparse(); }
```

### LEX Specification for the YACC Example

```
%%
ding return DING;
dong return DONG;
dell return DELL;
[ ] * ;
\n|. return yytext[0];
```

#### Compiling and running the parser

```
lex ding-dong.l
yacc ding-dong.y
gcc -o ding-dong.o y.tab.c
ding-dong.o
```

Sample inputs	Sample outputs
ding dong dell	string valid
ding dell	syntax error
ding dong dell\$	syntax error

## Forma de un archivo YACC

- ❖ YACC tiene un lenguaje para describir gramáticas libres de contexto.
- ❖ Genera un analizador LALR (1) para el CFG descrito.
- ❖ Forma de un programa YACC
  - %{declaraciones - opcional
  - %}
  - %%
  - reglas - obligatorio
  - %%
  - programas - opcional



- ❖ YACC utiliza el analizador léxico generado por LEX para hacer coincidir los símbolos terminales del CFG.
- ❖ YACC genera un archivo llamado y.tab.c

#### Declaraciones y Reglas

- ❖ **Tokens:** %token nombre1 nombre2 nombre3, ...
- ❖ **Símbolo de Inicio:** %nombre de inicio
- ❖ **nombres** en reglas: letra (letra | dígito | . | \_)\* la letra puede ser minúscula o mayúscula
- ❖ **Valores de los símbolos y acciones:** Ejemplo

```

A : B
    { $$ = 1; }
    C
    { x = $2; y = $3; $$ = x+y; }
    ;

```

- Now, value of A is stored in \$\$ (second one), that of B in \$1, that of action 1 in \$2, and that of C in \$3.

- ❖ La acción intermedia en el ejemplo anterior se traduce en una -producción de la siguiente manera:

```

$ACT1 : /* empty */
    { $2 = 1; }
    ;
A : B $ACT1 C
    { x = $2; y = $3; $$ = x+y; }
    ;

```

- ❖ Las acciones intermedias pueden devolver valores
- ❖ Por ejemplo, el primer \$\$ del ejemplo anterior está disponible como \$2
- ❖ Sin embargo, las acciones intermedias no pueden referirse a los valores de los símbolos a la izquierda de la acción.
- ❖ Las acciones se traducen en código C que se ejecutan justo antes de que el analizador realice una reducción.

#### Análisis Léxico

- ❖ LA devuelve los números enteros como números simbólicos.

- ❖ Los números de tokens son asignados automáticamente por YACC, comenzando desde 257, para todos los tokens declarados usando la declaración de **%token**.
- ❖ Los tokens pueden devolver no solo números de token sino también otra información (por ejemplo, valor de un número, cadena de caracteres de un nombre, puntero a la tabla de símbolos, etc.).
- ❖ Los valores adicionales se devuelven en la variable, **yylval**, conocida por los analizadores generados por YACC.

#### Symbol Values

- ❖ Los tokens y los no terminales son símbolos de pila.
- ❖ Los símbolos de pila se pueden asociar con valores cuyos **tipos** se declaran en una declaración %union en el archivo de especificación YACC.
- ❖ YACC convierte esto en un tipo de unión llamado YYSTYPE.
- ❖ Con declaraciones de % token y % type, informamos a YACC sobre los tipos de valores que toman los tokens y los no terminales.
- ❖ Automáticamente, las referencias a \$1, \$2, yylval, etc., se refieren al miembro apropiado del sindicato (vea el ejemplo a continuación).

#### Recuperación de errores en YACC

- ❖ Para evitar una cascada de mensajes de error, el analizador permanece en estado de error (después de ingresarlo) hasta que tres tokens se hayan transferido con éxito a la pila.
- ❖ En caso de que ocurra un error antes de esto, no se dan más mensajes y el símbolo de entrada (que causa el error) se elimina silenciosamente.
- ❖ El usuario puede identificar no terminales **mayores** como los de **programa**, **declaración**, o **bloque**, y agregue producciones de error para estos a la gramática.

Ejemplos:

*declaración* → **error** {acción 1}

*declaración* → **error** ';' {acción 2}

## YACC Error Recovery Example

```
%token DING DONG DELL
%start S
%%
S  @: rhyme{printf("string valid\n"); exit(0);}
rhyme : sound place
rhyme : error DELL{yyerror("msg1:token skipped");}
sound : DING DONG ;
place : DELL ;
place : error DELL{yyerror("msg2:token skipped");}
%%
```



VN Srikant YACC

## Actividades semana 8 (Nov 9-13, 2020)

### Vídeo 1

#### Análisis Semántico

La consistencia semántica que no puede ser manejada en la etapa de análisis se maneja aquí.

Los analizadores no pueden manejar las características sensibles al contexto de los lenguajes de programación

Son semánticas estáticas de los lenguajes de programación y pueden ser comprobadas por el analizador semántico.

Las variables se declaran antes de su uso.

Los tipos coinciden en ambos lados de las asignaciones.

Los tipos de parámetros y el número coinciden en la declaración y el uso

Los compiladores sólo pueden generar código para comprobar la semántica dinámica de los lenguajes de programación en tiempo de ejecución.

Si se produce un desbordamiento durante una operación aritmética.

Si los límites de la matriz se cruzaran durante la ejecución.

Si la recursividad cruzara los límites de la pila.

Si la memoria del cabezal es insuficiente.

La información de los tipos se almacena en la tabla de símbolos o en el árbol de sintaxis.

Tipos de variables, parámetros de función, dimensiones de la matriz, etc.

Se utiliza no sólo para la validación semántica sino también para las fases posteriores de la compilación.

La semántica estática de PL puede especificarse utilizando gramáticas de atributos.

Los analizadores semánticos pueden generarse semi automáticamente a partir de atributos.

Las gramáticas de atributos son extensiones de las gramáticas sin contexto.

### **Semántica Estática**

Muestras de comprobaciones semánticas estáticas en principal.

Los tipos de `p` y el tipo de retorno de `dot_prod` coinciden.

El número y el tipo de los parámetros de `dot_prod` son los mismos tanto en su declaración como en su uso.

`p` es declarado antes de su uso, igual para `a` y `b`.

Muestras de comprobaciones semánticas estáticas en `dot_prod`.

`d` e `i` se declaran antes de su uso.

El tipo de `d` coincide con el tipo de retorno de `dot_prod`.

El tipo de `d` coincide con el tipo de resultado de `"*"`.

Los elementos de las matrices `x` e `y` son compatibles con `"*"`.

### **Semántica Dinámica**

Muestras de comprobaciones semánticas dinámicas en `dot_prod`.

El valor de `i` no excede el rango declarado de las matrices `x` e `y` (tanto inferior como superior).

No hay desbordamientos durante las operaciones de `"*"` y  `"+"` en `d += x[i] * y[i]`

Muestras de comprobaciones semánticas dinámicas en `dot_prod`.

El valor de `i` no excede el rango declarado de las matrices `x` e `y` (tanto inferior como superior).

No hay desbordamientos durante las operaciones de `"*"` y  `"+"` en `d += x[i] * y[i]`.

Muestras de comprobaciones semánticas dinámicas de hecho.

La comprobación del programa no se desborda debido a la recursividad.

No hay desbordamiento debido a "\*" en  $n \cdot \text{fact}(n-1)$ .

### **Gramáticas de Atributos**

Dejemos que  $G = (N, T, P, S)$  sea un CFG y dejemos  $V = NT$ .

Cada símbolo  $X$  de  $V$  tiene asociado un conjunto de atributos (denotados por  $X.a$ ,  $X.b$ , etc.).

Dos tipos de atributos: heredados (denotados por  $AI(X)$ ) y sintetizados (denotados por  $AS(X)$ ).

Cada atributo toma valores de un dominio especificado (finito o infinito), que es su tipo.

Los dominios típicos de los atributos son, enteros, reales, caracteres, cadenas, booleanos, estructuras, etc.

Se pueden construir nuevos dominios a partir de dominios dados mediante operaciones matemáticas como producto cruzado, mapa, etc.

formación: un mapa,  $ND$ , dónde,  $N$  y  $D$  son los dominios de los números naturales y los objetos dados, respectivamente.

estructura: un producto cruzado,  $A_1A_2...A_n$ , donde  $n$  es el número de campos en la estructura, y  $A_i$  es el dominio del campo  $i$ .

### **Gráfico de Dependencia de Atributos**

Que  $T$  sea un árbol de análisis generado por el CFG de un AG,  $G$ .

El gráfico de dependencia de atributos (gráfico de dependencia para abreviar) para  $T$  es el gráfico dirigido,  $DG(T) = (V, E)$ , donde

$V = \{b \mid b \text{ es una instancia de atributo de algún nodo del árbol}\}$ , y

$E = \{(b, c) \mid b, c \in V, b \text{ y } c \text{ son atributos de símbolos gramaticales en la misma producción } p \text{ de } B, \text{ y el valor de } b \text{ se utiliza para calcular el valor de } c \text{ en una regla de cálculo de atributos asociada a la producción } p\}$

### **Estrategia de Evaluación de Atributos**

Construye el árbol de análisis.

Construye el gráfico de dependencia.

Realizar una clasificación topológica en el gráfico de dependencia y obtener un orden de evaluación.

Evaluar los atributos según este orden utilizando las reglas de evaluación de atributos correspondientes adjuntas a las producciones respectivas.

Múltiples atributos en un nodo del árbol de análisis pueden hacer que ese nodo sea visitado varias veces.

Cada visita resulta en la evaluación de por lo menos un atributo

### **Gramáticas Atribuidas por la L y la S**

Un AG con sólo atributos sintetizados en una gramática de atributos S.

Los atributos de los SAG pueden ser evaluados en cualquier orden de botones sobre un árbol de análisis (de una sola pasada).

La evaluación de los atributos puede ser combinada con el análisis de LR (YACC).

En la gramática atribuida a la L, las dependencias atribuidas siempre van de izquierda a derecha.

### **Más precisamente, cada atributo debe ser.**

Sintetizada,

Heredado, pero con las siguientes limitaciones: considerar una producción  $p: AX_1X_2...X_n$ . Let  $X_i.a \in A_i(X_i)$ .  $X_i.a$  sólo se puede utilizar.

elementos de  $A_i(A)$ .

elementos de  $A_i(X_k)$  or  $A_i(X_k)$ ,  $k = 1, ..., i - 1$  (es decir, los atributos de  $X_1, ..., X_{i-1}$ )

Nos concentramos en los SAG, y 1 - pasar los LAG, en los que la evaluación de atributos puede combinarse con el análisis de LR, LL o RD.

Nombre: rodriguez perez luis digo

Video 2

Análisis Semántico de atributos, y gramáticas de traducción atribuidas.

Así que, para hacer un poco de recapitulación, las gramáticas de atributos son gramáticas extendidas sin contexto, cada

El símbolo del conjunto  $N$  unión  $T$  tiene asociados algunos atributos.

Hay dos tipos de atributos heredados y sintetizados, por supuesto,

el mismo atributo no puede ser ambos heredado y sintetizado, pero podría haber varios heredados y varios atributos sintetizados asociados a cada símbolo. Cada atributo toma un valor de un dominio específico, como un entero o un producto real o cruzado de estos etcéteras, etcétera. Y asociamos reglas de cálculo de atributos con cada producción, así que

específicamente asociamos reglas para el cálculo de los atributos sintetizados de la mano izquierda no terminal, y nosotros reglas de asociación para el cálculo de los atributos heredados de los no terminales de la derecha de la producción.

Por supuesto, el aspecto más importante de una gramática de atributos es que es estrictamente las reglas estrictamente locales a cada producción P no hay efectos secundarios en las reglas.

Los atributos de los símbolos se evalúan sobre un árbol de parseo haciendo pasadas sobre el parseo árbol, es posible tener más de una pasada y cada pasada sería por lo menos un atributo calculado en los nodos. Así que los atributos sintetizados se calculan en un fondo de las hojas hacia arriba, así que básicamente se sintetizan a partir de los valores de los atributos de los hijos del nodo en el árbol de análisis. Los nodos de la hoja, que son los terminales, tienen sólo sintetizan atributos y estos son inicializados por el analizador léxico y por supuesto, ellos no pueden ser modificados. En lo que respecta a los atributos heredados, estos valores influyen realmente desde el padre o los hermanos hasta el nodo en cuestión y luego, por supuesto, ellos fluyen las palabras de nuevo desde ese nodo hacia abajo.

- AG for the evaluation of a real number from its bit-string representation  
Example: 110.101 = 6.625
- $N \rightarrow L.R, L \rightarrow BL \mid B, R \rightarrow BR \mid B, B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value \uparrow: real\}$ .  
 $AS(L) = \{length \uparrow: integer, value \uparrow: real\}$ 
  - $N \rightarrow L.R \{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$
  - $L \rightarrow B \{L.value \uparrow := B.value \uparrow; L.length \uparrow := 1\}$
  - $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$   
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$
  - $R \rightarrow B \{R.value \uparrow := B.value \uparrow\}$
  - $R_1 \rightarrow BR_2 \{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow) / 2\}$
  - $B \rightarrow 0 \{B.value \uparrow := 0\}$
  - $B \rightarrow 1 \{B.value \uparrow := 1\}$

Un breve resumen de la estrategia de evaluación de atributos, debemos construir el árbol de análisis, entonces construimos el gráfico de dependencia de los atributos.

Realizamos el orden topológico de la dependencia y obtenemos una orden de evaluación, por supuesto, luego se lleva a cabo la evaluación de los atributos...usando esta orden. y la evaluación de los atributos de las respectivas producciones se utilizan en el atributo proceso de evaluación. Entonces, miramos el ejemplo, seguimos mirando el ejemplo que te mostré la última vez, así que esta es la gramática de atributos para la evaluación de un número real a partir de la representación de la cadena de bits.

Por ejemplo, si se tiene una cadena de bits 110.101, su valor decimal es 6.625, aquí hay un contexto La gramática libre N va a L punto R, L genera varios bits en el lado izquierdo del punto. Y si la gramática L va a B L o B, de manera similar R genera los bits en el lado derecho del punto y la gramática es R yendo a B R o B B es por supuesto, un poco o 0 o 1.

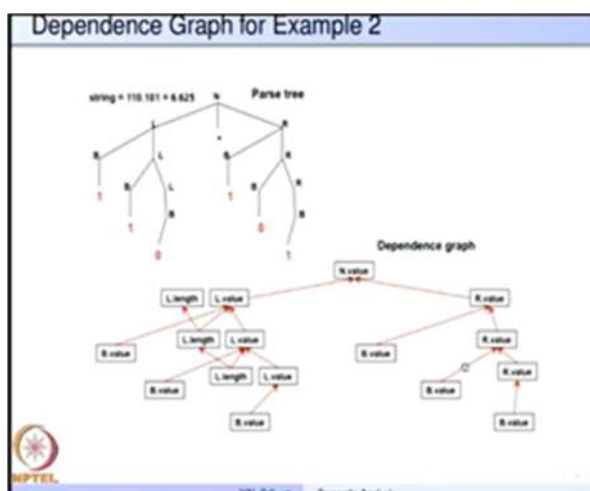
Ahora, los atributos de los varios, así que si miramos esta cadena genera esta gramática genera cadenas binarias con un punto en el medio. Pero, no tenemos ninguna indicación de lo que es el valor decimal, el atributo gramatical se supone que nos da el valor decimal de esta cualquier cadena de bits en particular generada por esta gramática. Los no terminales N R y B tienen un solo valor de atributo sintetizado, que es del dominio de los reales, donde ya que la L no termina tiene una longitud de atributo sintetizada y tiene otro atributo sintetizado valor.

Así que, la longitud es del dominio de los números enteros y real es del dominio de los números reales, ¿por qué requerimos dos atributos para L, donde como uno nos suministra para N L R y B. Así que, para convencer a los nuestros de que esto es necesario, echemos un vistazo a las cuerdas del lado izquierdo del punto, ya sabes, supongamos que consideramos que este es uno de los 110 de la cadena. El poder que se le da a los más bien valor de exponenciación dado a este en particular, es decir, 2 a la potencia 2.

Así, el primero es un 0 2 a la potencia 0, el segundo es 2 a la potencia 1 y el tercero uno es 2 a la potencia 2, esto en realidad se basa en el número de bits a la derecha lado hasta el punto. Así que, hay 2 aquí, si tuviéramos más sería el poder de la base 2, esto no se puede determinar y si miramos la gramática para L esta b indica esto uno que estamos considerando. Y la L a la derecha de ella indica el resto de los bits a la derecha de esta en particular en este caso.

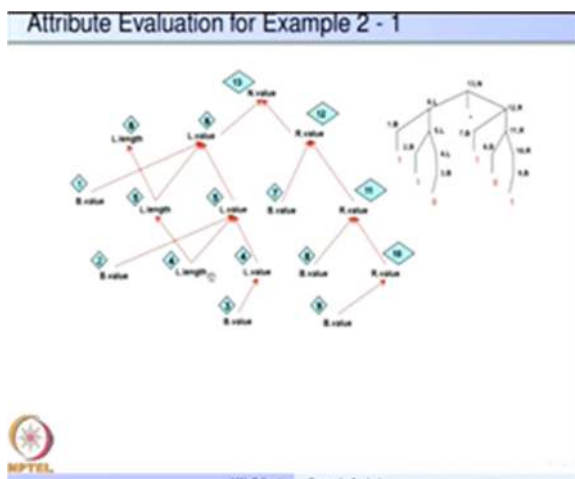
Así que, sin saber la longitud de la cadena que es generada por L, no es posible asignar el peso apropiado a este bit en particular. En el caso de la fracción el peso en realidad comienza con menos 1 justo después del punto, así que esto es 2 a la potencia menos 1, esto es 2 a la potencia menos 2 y el tercer bit es 2 a la potencia menos 3. Y la gramática es también tal que, la B, que está más cerca del punto se genera primero y la R es la el resto de los puntos después de la B. Así que, por lo tanto, asignar un peso a la B es no es para nada difícil y es suficiente tener el valor justo como atributo de este no terminal.

Entonces, veamos el atributo gramatical N que va a L punto R, así que el valor de N es; obviamente, el valor de L más el valor de R, asumiendo que asignamos pesos a la...pedazos en L y R apropiadamente.





Así que, L genera un bit por la producción L yendo a B, por lo que el valor del punto L será sólo el valor del punto B ya sea 0 o 1 y la longitud del punto L es 1 porque, estamos generando sólo 1 bit. L 1 va a b L 2, así que L 2 tiene cierta longitud y B es un bit extra, que ahora se ha generado. Así que, L 1 punto de longitud es; obviamente, L 2 longitud de punto más 1, ¿qué pasa con el valor L 1 valor de punto es; obviamente, tomamos el L El valor de 2 puntos, al igual que tomamos 2 a la potencia 2 más el valor de 1 0 que es 2 que es El total será de 6. Así que, de manera similar tomamos el valor de L 2, así que el valor de L 2 puntos y tomamos el valor del valor del punto del bit B, ya sea 0 o 1 multiplicado por el peso apropiado 2 a la potencia L 2 punto de longitud. Así que, aquí L 2 longitud de punto habría sido 2, así que esto habría sido asignar el valor 2 a la potencia 2. Así que, así es como el valor de L 1 se obtiene, ya sabes que el cálculo de R a B es muy simple valor de punto R igual a la barra de valor de punto B 2 y R 1 van a B R 2, así que hay una razón por la que tenemos el valor del punto B barra 2 es que generamos el, ya sabes, primer bit después de que el punto se aleja 2 a la potencia menos 1, así que el valor del bit dividido por 2 es el valor del punto R. Ahora, R yendo a B R 2 R 1 valor de punto será toma el valor del punto B más el valor del punto R 2 y el conjunto se divide por 2 que es bastante fácil de ver.



Si miras la producción porque, R 2 tenía algún valor, ahora debido a la parte que ha sido desplazada a la derecha por una posición. Por lo tanto, es el valor dividido por 2 es el derecho y el valor de este R 2 en particular y un nuevo bit ha sido introducido, por lo que su valor es El valor del punto B y como dije, siempre debemos dividir el valor del bit por 2. Porque, los pesos comienzan con menos 1 aquí, B a 0 es el valor del punto b igual a 0 y B a 1 será el valor del punto B igual a a la 1.

Así que, aquí está digamos una cadena de muestra 110.101 es un árbol de análisis, las rojas son todas las hojas y este es el gráfico de dependencia basado en este árbol de análisis y la gramática de atributos.



Aquí está el mismo gráfico, el retorno que conoces de una manera ligeramente diferente con el número de nodos según la secuencia de evaluación. Así que, 1, 2 y luego 3 estos son los 3 nodos que son evaluado primero porque, no requieren ningún otro nodo para ser evaluados, por lo que es posible para evaluar la longitud del punto L también junto con estos 2 o más bien estos 3 porque, no requiere nada más en este momento, pero no podemos evaluar el valor del punto L porque, el punto L requiere el valor de punto b. Por lo tanto, preferimos esperar hasta que los valores de los puntos B sean todo completado y luego ir a este nodo 4, que tiene estos 2 atributos y evaluar ambos al mismo tiempo.

Así que, ahora L longitud del punto igual a 1, L valor del punto igual a 0, así que 4 y luego 5 y sabes estos son los nodos, que serán evaluados, en el nodo 4 la producción, aunque todos aquellos como dije el nodo 4 se evalúa primero y luego el nodo 5 porque, los valores del nodo Se requieren 4 para el nodo 5. Así, en el nodo L de un 4 L a B se aplica y el valor puede se calculará utilizando la regla de producción, de modo que el valor del punto L es igual al valor del punto B y el punto L longitud igual a 1.

Así que, eso utiliza apropiadamente el 1 y el 0 aquí, A medida que subimos, la producción aplicada es L 1 que va a B L 2. Por lo tanto, la longitud se incrementa, así que esto se convierte en 2 y el valor del punto L 1 se calcula usando la regla del valor del punto B en 2 a la potencia L 2 longitud de punto más L 2 valor de punto. Así que, el valor de L 2 es 0, la longitud del punto de L 2 es 1, así que esto se convierte en 2 a la potencia 1 en 1 que es S 2.

**Attribute Grammar - Example 4**

- An AG for associating type information with names in variable declarations
- $AI(L) = AI(ID) = \{type \downarrow: \{integer, real\}\}$   
 $AS(T) = \{type \uparrow: \{integer, real\}\}$   
 $AS(ID) = AS(identifier) = \{name \uparrow: string\}$
- $DList \rightarrow D \mid DList ; D$
- $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
- $T \rightarrow int \{T.type \uparrow := integer\}$
- $T \rightarrow float \{T.type \uparrow := real\}$
- $L \rightarrow ID \{ID.type \downarrow := L.type \downarrow\}$
- $L_1 \rightarrow L_2 . ID \{L_2.type \downarrow := L_1.type \downarrow; ID.type \downarrow := L_1.type \downarrow\}$
- $ID \rightarrow identifier \{ID.name \uparrow := identifier.name \uparrow\}$

Example: `int a,b,c; float x,y`  
 a,b, and c are tagged with type *integer*  
 x,y, and z are tagged with type *real*

Así que, ahora vamos más allá, ahora el nodo 6, el nodo 7, 8 y 9 en ese orden serán evaluados, en 6 tenemos el mismo L 1 yendo a b L 2 y, por lo tanto, el valor obtenido aquí sería 2 más 2 al cuadrado. Entonces, eso sería 6 y la longitud se convierte en 3 porque, nosotros se genera 3 bits, por lo que para los nodos 7 y 9, así que 7 y 9 aplicamos B yendo a 1 y en el nodo 8 aplicamos B va a 0, así que el valor aquí B es 1 y estos 2 son esto es 0 y estos 2 son 1.

Luego evaluamos los nodos 10, 11 y 12 y 13 en ese orden, así que esto obtiene el valor del punto R igual a 0,5, debido a la regla R que va a B, esto hace que el valor del punto R sea igual a 0.25.

Porque, este es un 0 que se divide por 2 y entonces esto te hace conocer el valor 0,625 porque, este es un 1 que te da 0,5 y este es 525 finalmente, sumando del lado izquierdo y del derecho, obtenemos el valor de N punto igual a 6.625. Así que, este es el orden en el que la evaluación ocurre sobre el árbol de análisis y sobre la dependencia gráfico.

Así que, tomemos otro ejemplo, así que este ejemplo en particular es de nuevo para el cálculo de un verdadero de la representación de la cadena de bits, pero el método es muy diferente. Así que, mira a la gramática, la gramática anterior tenía L punto R y dos gramáticas diferentes para las producciones de L y R para L y R.

Pero, aquí tenemos sólo un punto X no terminal, ya sabes, X y la producción es N yendo a X punto X, X yendo a B X o B B yendo a 0 o 1, así que cuando el no terminal X produce bits que no es posible saber, si estamos en el lado derecho del punto o en el lado izquierdo del punto. Así que, debido a este problema no podemos usar la misma gramática de atributos que habíamos estudiado antes, la estrategia va a ser muy diferente.

Pero, en cierto sentido es una estrategia más simple también, calculamos el valor de la cadena como es, así que realmente no nos preocupamos por el punto del principio. Así que, 110 tiene valor 6 y 1010 tiene el valor 10 decimal valor 10, ahora hay 4 bits después del punto, así que La longitud de la cuerda aquí es 4, así que simplemente dividimos la parte de la fracción por 2 a la potencia 4 y eso nos da 6 más 10 por 16, que es 6 más 0,625, por lo que obtenemos el valor antiguo 6.625.

Así que, esto es inherente, conoces el valor inherente aunque tuviéramos 0 extra aquí o 0 en el lado izquierdo de este etcétera, etcétera.

Tomemos el caso de que esto se convierta en 10100, así que en ese caso el valor es realmente el doble de 10 que es 20, pero al mismo tiempo, el número de bits también se convierte en 5. Así que, en lugar de 10 por 2 a la potencia 4 tenemos realmente 20 por 2 a la potencia 5, que es la misma 10 por 2 a la potencia 4 y el valor sigue siendo 6.625, aquí nuevamente requerimos, conoces el valor como un...de los atributos N y B. Y en cuanto a la no terminal X..., requerimos la longitud y también el valor, por lo que es muy fácil de ver, este es el y esta es la longitud. Pero, en el cálculo del valor no vamos a ir para diferenciar entre la parte de la fracción y la parte entera. Así que, N va al punto X lo escribimos como X 1 punto X 2 sólo para diferenciar entre las 2 instancias de la X, así que N El valor del punto será X 1 valor del punto más X 2 valor del punto dividido por 2 a la potencia X 2 longitud del punto.

Así que, el valor de x 2 puntos dividido por 2 a la potencia X 2 longitud de punto es lo que hemos hecho aquí, X a B directamente obtenemos el valor del punto X igual al valor del punto B y la longitud del punto X iguala 1.

X yendo a B X nos dará X 1 longitud de punto igual a X 2 longitud de punto más 1, lo cual es muy fácil porque, hemos añadido un poco aquí y el valor del punto X 1 es el valor del punto B en 2 a la potencia X 2 puntos de longitud. Así que, eso es también algo de la gramática anterior más el valor de X 2 puntos, estos dos son tan directos como antes.

Por lo tanto, esta gramática de atributos es diferente para el cálculo del número real de una representación de una cadena de bits. La moraleja de estos dos ejemplos es que, las frases en el idioma pueden ser el mismo, las gramáticas sin contexto pueden ser diferentes y allí por las gramáticas de atributos para el cálculo del mismo valor también puede tener que convertirse en ...diferente.

Así que, sigamos adelante tomemos un ejemplo diferente, una gramática de atributos para asociar teclada la información con los nombres en declaraciones variables. Así que, tenemos la gramática se da aquí, así que veamos cómo funciona DList es D o DList punto y coma D, así que aquí D genera una sola declaración y el DList genera una lista de declaraciones dentro de D tenemos D yendo a T L.

Así que, T es el tipo, así que o bien entero o flotante y L es una lista de nombres de este Así que, eso es cómo se generan estas dos declaraciones, llegando ahora a los atributos de esta gramática, aquí por primera vez introducimos los atributos heredados. Así, por ejemplo, la no terminal L y la identificación no terminal, ambos tienen el tipo de información que tiene un atributo heredado. Asumamos que un tipo escalar definido por el usuario, ya sabe la coma entera real, lo cual está permitido en C, C plus plus, Pascal etcétera.

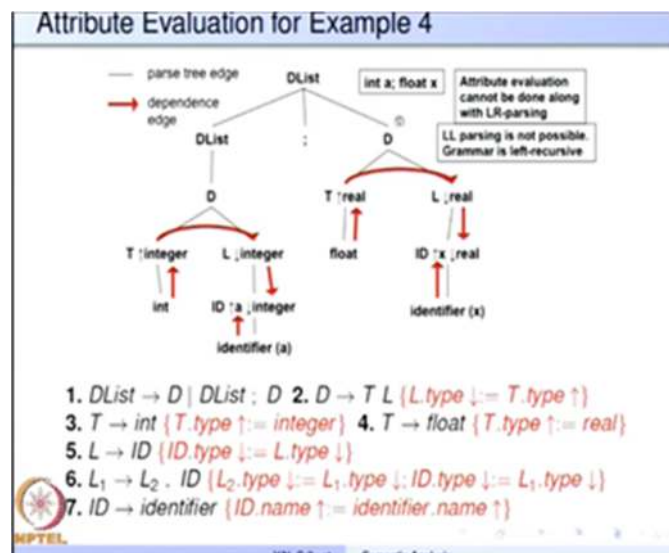
Así que, el tipo es del dominio de dos cantidades enteras coma real y estos son atributos heredados de L y D, donde como la T no terminal tiene un tipo de atributo sintetizado, que a su vez es de tipo entero o real. El no terminal I D también tiene un atributo sintetizado, que es el nombre de la variable y el terminal El identificador de símbolo también tiene nombre ya que es un atributo sintetizado, que no es más que una cadena de personajes.

Entonces, ¿cómo es que este atributo gramatical en particular trabajo, ya sabes, a medida que avanzamos, por ejemplo, deja un ejemplo entero a, b, c. Por lo tanto, el El tipo de las 3 variables a, b, c es entero a es de tipo entero, b es de tipo entero c también es de tipo entero, así que estos nombres a, b y c se generan realmente en este nivel por el identificador de la producción. Para asociar el nombre con él es necesario escribir en realidad tomar este entero en particular y ponerlo a disposición de la lista de nombres que se está generando.

Así que, eso es precisamente lo que hacemos aquí, la primera producción los dos primeros DList que van a D o DList que van a DList punto y coma D tienen no hay cómputo asociado a ellos, D yendo a T L tiene una regla de cómputo de atributos.

Por lo tanto, dice que el tipo de punto L que es el tipo de los nombres generados por L no es nada más que el punto T así que el tipo de punto L se hereda y el tipo de punto T se sintetiza. Así que, estas cosas y sintetizadas y entrando en L veremos un ejemplo con un árbol de análisis dentro de un corto archivo T que va a entero.

Así que, aquí el tipo de punto T es entero, así que eso es muy simple, de manera similar la T que va a flotar nos da el tipo de punto T igual al real, la L que va a la I D. Así que, de nuevo L tiene el tipo como un atributo heredado y que tiene que ser pasado a la I D tiene un atributo heredado, así que I D tipo punto igual a L tipo punto, L 1 va a L 2 coma I D.



Así, la información de tipo que se da a L 1 desde la parte superior se pasa a ambos L 2 y I D aquí, así que, por lo tanto, L 2 tipo de punto igual a L 1 tipo de punto y I D tipo de punto igual a L 1 punto tipo I D va a identificador, así que el nombre del punto I D es el nombre del punto identificador. Así que, a nivel de I D hemos asociado tanto el tipo como el nombre de ese particular...variable.

Así que, a b y c están etiquetadas con el tipo entero x, y, z están etiquetadas con el tipo real en este ejemplo en particular, por lo que debemos observar aquí que la notación para los atributos heredados es el atributo y luego una flecha hacia abajo. Y tenemos reglas de cálculo de las reglas de atributo para los atributos heredados en el lado derecho de la producción, así que L ha heredado por lo que proporcionamos una regla de cálculo para ello.

Pero, no proporcionamos ninguna regla de cálculo para el atributo sintetizado de T porque, está en el lado derecho de esta producción en particular. Mientras que, T está en la mano izquierda de esta producción, por lo que proporcionamos una regla de cálculo para ello, así que de manera similar L 1 va a L 2 coma I D L 2 punto es hereditaria. Por lo tanto, hay una regla de cálculo I D punto se hereda, así que hay una regla de cálculo. Pero, el tipo L 1 punto también se hereda, así que está en el lado izquierdo de la producción, así que no proporcionamos ninguna regla de cálculo para el tipo de punto L 1.

Así que, tomemos este ejemplo y entendamos cómo los atributos heredados fluyen sobre el árbol de análisis. Así, los bordes rojos son todos bordes de dependencia de atributos y los bordes negros son todos analizar los bordes del árbol, por lo que la frase es entera un punto y coma flotante x, por lo que DList nos da DList punto y coma D, esta D genera la flotación x y esta D DList genera D y luego esa D genera Entero a.

Así que, para empezar por supuesto, en este nivel DList va DList punto y coma D allí no es una contribución, sino más bien un cálculo de atributos aquí.

El primer cálculo de atributos se asocia sólo con él, así que tomemos primero esto es más simple. Entonces, tenemos la T y luego eso va a flotar y la L de aquí va a la I D y I D va al identificador, el identificador es x, así que aquí el flotador es un atributo sintetizado y fluye al nodo T y la regla de cálculo es que T va a flotar.

Así que, T el tipo de punto se hace real, así es como este atributo real es calculado, aquí la cadena x fluye a I D, por lo que I D va a identificador y el cómputo es I D nombre de punto igual a identificador nombre de punto, así es como esta x se calcula en este nodo en particular.

Ahora, el atributo real que es sintetizado por T es pasado como un atributo infinito a L y eso sucede en el papel D yendo a T L. Así que, tenemos D yendo a T L aquí, así que el tipo de punto L es el tipo de punto T, así que esa es la dependencia y el cálculo asociado a esta producción.

Así que, de manera similar L va a I D pasará el atributo de L a I D, así que L va a I D, así que I D tipo de punto igual a L tipo de punto, así que la dependencia es correcta, la misma cosa se mantiene aquí también. Así que, el número entero se convierte en un número entero... en T y luego T fluye como un atributo heredado a L y que fluye en I D como un heredado el nombre a fluye como un atributo sintetizado al atributo no terminal D. Así, el flujo de atributos aquí, por lo que hay dos nodos, que se proporcionan aquí, el atributo número uno la evaluación no puede hacerse junto con el análisis LR, porque hay atributos heredados aquí y eso se aclara a medida que avancemos.

El análisis de esta gramática no es posible porque, la gramática dejó de ser recursiva, así que esto también es necesario, porque vamos a ver cómo modificar esta gramática más tarde para habilitar el análisis de L L.

Así que, otro ejemplo de gramática de atributos y este es un ejemplo mucho más complicado, Aquí hay un lenguaje bastante simple de expresiones  $S \rightarrow E$ ,  $E \rightarrow E + T$  más o  $T$  o un nuevo tipo de expresión  $\text{id}$  igual a  $E$  en  $E$ . Entonces, tenemos como siempre  $T$  yendo a la estrella  $T \rightarrow F$  o  $F$ ,  $F$  va al paréntesis  $E$  o al número o  $\text{id}$ . Así que, la novela  $T$  y la especialidad de este lenguaje en particular es que se nos permite definir un valor  $E$  para este nombre en particular  $\text{id}$  y esa unión puede ser usada dentro de esta expresión  $E$ .

Por lo tanto, esto puede ser anidado como se puede ver  $E$  puede ser de nuevo una expresión de la izquierda y esto también puede ser una expresión recursiva. Así que este lenguaje permite que las expresiones se añaden dentro de las expresiones y tienen alcances para los nombres, así que déjenme mostrarles con un ejemplo lo que queremos decir con el alcance de este nombre particular  $\text{id}$ , en esta producción particular  $E \rightarrow E + \text{id}$  igual a  $E$  en  $E$ . Aquí está el ejemplo,  $A$  igual a 5 en ahora  $a$ , así que hemos, hasta ahora dicho que la  $\text{id}$  es igual a  $E$ , así que la  $\text{id}$  es  $A + E$  es 5. Y luego tenemos una nueva expresión este paréntesis entero un término, por lo que es una segunda  $E$ , el segundo  $E$  tiene una estructura similar,  $A$  igual a 6 en una nueva expresión comienza una estrella 7. Y eso se completa y entonces tenemos un menos  $A$ , así que todo esto  $A$  igual a 6 en La estrella 7 es la parte de la  $E$  aquí, y luego tenemos la  $A$ , que es la segunda parte.


para esto los alcances de las variables  $A$  en ambos casos se muestran en diferentes colores. Así que, esta  $A$  igual a 5 es válido sólo para esta  $A$  en particular, donde dentro de esta expresión entre paréntesis está  $A$  igual a 6 es válida aquí en este punto de la expresión.

Entonces, el significado es que está  $A$  toma el valor de 6, lo que nos da 42 y luego este entero La expresión  $A$  igual a 6 en LA estrella 7 tenga el valor 42. Así que, ahora, este particular  $A$  igual a 5 es válido para esta  $A$ , por lo que esta toma el valor 5, esta expresión entera tomó el valor 42.

Así que, 42 menos 5 es 37, por lo que el alcance de esta  $A$  y está  $A$  son claramente diferentes, así que el mismo nombre  $A$ , pero 2 valores diferentes y 2 alcances diferentes. Analizar y evaluar tales expresiones no es, una vez más, trivial, requiere el concepto de una tabla de símbolos para hacer que suceda, vamos a ver eso también.

### Attribute Grammar - Example 5

- 1  $S \rightarrow E \{ E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow \}$
- 2  $E_1 \rightarrow E_2 + T \{ E_2.symtab \downarrow := E_1.symtab \downarrow;$   
 $E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow \}$
- 3  $E \rightarrow T \{ T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow \}$
- 4  $E_1 \rightarrow \text{let } id = E_2 \text{ in } (E_3) \{ E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow;$   
 $E_3.symtab \downarrow := E_1.symtab \downarrow \setminus \{ id.name \uparrow \rightarrow E_2.val \uparrow \} \}$
- 5  $T_1 \rightarrow T_2 * F \{ T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow;$   
 $T_2.symtab \downarrow := T_1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow \}$
- 6  $T \rightarrow F \{ T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow \}$
- 7  $F \rightarrow (E) \{ F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow \}$
- 8  $F \rightarrow \text{number} \{ F.val \uparrow := \text{number.val} \uparrow \}$
- 9  $F \rightarrow id \{ F.val \uparrow := F.symtab \downarrow[id.name \uparrow] \}$



Por lo tanto, lo que vamos a mostrar es un atributo gramatical abstracto para el lenguaje anterior, que utiliza tantos atributos heredados como sintetizados. Y estos atributos pueden, por supuesto, ser evaluado en una pasada por el árbol de parse de izquierda a derecha, les mostraré un ejemplo de cómo se puede hacer eso también. Los atributos heredados no pueden ser evaluados durante LR por lo que esta es una regla general a menos que pongamos una gran cantidad de restricciones en los atributos heredados que esta regla tiene. Por lo tanto, siempre que usamos atributos heredados la suposición es que usamos el análisis de LL. Y cuando sólo hay atributos de síntesis, entonces la gramática puede ser usado junto con los análisis de LR.

Así que, aquí está el atributo gramatical para esa gramática libre de contexto particular, el símbolo La tabla es una entidad, que almacena los nombres y sus valores. La tabla de símbolos, que es válido en un punto particular de la expresión, se transmite realmente a la expresión de él es el padre en el árbol de análisis. Así, por ejemplo, en este caso la tabla de símbolos que es válido dentro de la expresión E es phi porque, este es el símbolo de inicio de la gramática.

Así, E punto simbólico igual a phi y que es E punto simbólico es un atributo heredado y después de que la evaluación se completa E produce un valor y E punto val se asigna ahora a S punto val y ese es el valor de toda la expresión. Tomemos esta producción E 1 que va a E 2 más T, así que E yendo a E más T y las instancias numeradas como 1 y 2, había una tabla de símbolos disponible para E 1 y que es una misma tabla de símbolos, que se pondrá a disposición de ambos E 2 y T sin ninguna modificación.

Así que, E 2 punto simbólico es E 1 punto simbólico, E 1 punto y T punto simbólico es también E 1 punto simbólico, así que recuerda el orden en que estos atributos las reglas de cálculo que se escriben no es muy importante. Porque, esta no es la última El orden E 1 punto val no es más que la suma de los valores producidos por este E 2 y este T, así que E 2 dot val más T dot val, así que este es el valor de E 1 dot val. Así que, E a T es trivial sólo hay una copia de los dos atributos de E a T y así T punto simbólico es E punto simbólico y E



punto val es T punto val. Las cosas importantes suceden en la producción número 4 aquí y luego también ocurre en la producción número 9. Así que, tomemos la producción número 4, que E va a dejar i d igual a E 2 en E 3, por lo que la tabla de símbolos que fue disponible para E 1 se pone a disposición de la expresión E 2. Por lo tanto; eso significa, E 2 punto simbólico es igual a E 1 punto simbólico, también está disponible para E 3, por lo que la tabla de símbolos de E 1 se pone a disposición de E 3 también, pero sucede que esta asociación particular de El valor de los puntos i d y E 2, ahora se introduce en la tabla de símbolos producida por E 1.

Así, que se escribe como E 3 punto simbólico es igual a E 1 punto simbólico sobrepuesto por la entidad o el nombre del punto de la asociación que va a E 2 punto val. Por lo tanto, la consecuencia de esto es doble, por ejemplo, si el nombre i d dot name, ya sabes que está disponible en E 1, entonces hay otra asociación de ese nombre en particular a algún otro valor. Entonces esa particular asociación se olvida y se produce una nueva asociación de i d 2 E 2 punto val, por lo que la antigua la asociación se suprime y la nueva asociación se rige por la expresión E 3.

Dije cuidadosamente suprimido no eliminado porque, tan pronto como el alcance de este fin particular La expresión que es igual a E 2 en E 3 se ha completado. El antiguo valor de la asociación del nombre i d a algún otro valor que estaba presente en E 1 debería estar disponible otra vez.

Así, en este caso, por ejemplo, A igual a 5 es suprimido en la estrella A 7 por la asociación de A a 6, pero tan pronto como esta expresión se complete el valor de A a 5 está disponible en esta expresión particular o la variable A.

Entonces, este operador uno como el operador de anulación y no el operador de borrado, T 1 va a La estrella F de T 2 es muy similar a E 1 yendo a E 2 más T y no requiere demasiada elaboración.

La tabla de símbolos se acaba de copiar T 2 punto simbólico y F punto simbólico o T 1 punto simbólico y T 1 dot val es T 2 dot val estrella F dot val T 2 f es similar a E a t. Así que, y sólo hay copia de los atributos, F va al paréntesis E el paréntesis también tiene sólo una copia de la aquí, el número F 2 dice que F dot val es igual al número dot val y ahora F 2 i d de nuevo aquí está el uso de la variable. Así que, por ejemplo, en esta estrella A 7 A es un uso de esta definición particular y esta A es un uso de esta definición particular.

Así que. Esa definición que usted sabe que el uso habría sido producido por F 2 i d, así que F punto val se obtiene buscando el nombre i d dot name en F dot symtab, que es un F dot symtab está disponible aquí, como un atributo heredado y el nombre del punto i d está disponible como un sintetizado atributo de la terminal. Así que, buscamos este nombre en la tabla de símbolos y producimos el valor, que se asigna a F punto val, por lo que esta operación de búsqueda está indicada por los dos paréntesis y el nombre dentro.

Así que, aquí hay un ejemplo que muestra cómo se produce el flujo de atributos, la expresión es dejar que un igual a 4 en un más 3, así que un igual a 4, así que un más 3 se convierte en 7, por lo que es el que se produce aquí que es correcto. Así que, ahora, S va a E, por lo que inicializa la tabla de símbolos a phi, por eso la tabla de símbolos se muestra como phi aquí, la notación utilizada es ligeramente diferente aquí, el atributo heredado tiene

productions used:  
 $S \rightarrow E$   
 $E \rightarrow \text{let } id = E \text{ in } (E)$

let i : a = number [4] in ( id [a] \* number [3] )

Diagram illustrating Attribute Flow and Evaluation for Example 5. The diagram shows the parse tree for the expression "let i : a = number [4] in ( id [a] \* number [3] )". The root node is S [7], which expands to E [7]. E [7] expands to E [7] \* E [7]. The left E [7] expands to E [7] [4], which further expands to T [4] and F [4]. The right E [7] expands to E [7] [17], which further expands to E [7] [17] (a) and E [7] [17] (4). The diagram uses color-coded arrows to show the flow of attributes (green for synthesis, red for evaluation) and the evaluation of the expression (blue arrows). The final result is 14.

Así que, esto fluye hacia abajo de las palabras de E a T y T a f y aquí está el uso de a, por lo que en esta producción en esta expresión E, por lo que debemos buscar el nombre a en el símbolo y eso es lo que hemos hecho.

Por lo tanto, esto va hacia arriba como el valor de T y este valor se pasa como el valor de E, de manera similar el número F 2 produce 3 que se pasa...y las palabras. Y la regla asociada con E a E más T combina 4 y 3 en una suma para producir el valor 7, que se pasa

hacia arriba como el atributo sintetizado de E y que a su vez va a la S, por lo que esta E como te dije es la E particular. Así que, esa es la que ha producido 7, en realidad estos producen un 7 y ese es el valor que se envía hacia arriba como el valor de la expresión completa. Así que, así es como la evaluación de los atributos ocurre la secuencia en la que ocurre veremos dentro de un rato.

Así que, antes de mirar la secuencia de evaluaciones de atributos para los ejemplos que mostré ...a ti. Veamos una clasificación de gramáticas de atributos llamada gramáticas de atributos L y S gramáticas de atributos, así que si sólo has sintetizado atributos en una gramática de atributos, entonces se llama como una gramática atribuida a la S. Por lo tanto, es una definición muy simple, hay no son atributos heredados en absoluto y en el caso de tal SAG cualquier evaluación de abajo hacia arriba El orden sobre un árbol de análisis puede ser usado para evaluar los atributos. Y, obviamente, un solo análisis sobre el árbol es suficiente, y sucede que ese atributo La evaluación puede ser combinada con el análisis de LR también. Porque, el análisis LR produce una de abajo hacia arriba, ya sabes, hace un análisis de abajo hacia arriba sobre el árbol de análisis, en lugar de la construcción del árbol de análisis se hace de abajo hacia arriba, es bastante, ya sabes, conveniente para hacer la evaluación de atributos también de forma combinada con el análisis de LR.

Por lo tanto, YACC sólo permite atributos sintetizados y las reglas que escribimos para YACC, tú saber que las producciones de YACC se ejecutarán a medida que subamos en el proceso de análisis de LR.

Te mostraré algunos ejemplos de esto muy pronto, ¿qué hay de las gramáticas de L atribuidas, L atribuidas gramáticas que tienen sus dependencias de atributos van de la izquierda a la escritura.

Por lo tanto, el muy preciso, por lo que cada atributo debe ser sintetizado en cuyo caso no hay ningún problema o podría y si se hereda, entonces hay una limitación, supongamos que la producción es, así que ahora, formulamos lo que se conoce como "left to write dependence". Supongamos que la producción es A yendo a  $X_1 X_2$  etcétera  $X_N$  y dejemos el atributo  $X_i$  punto a ser un atributo heredado del símbolo  $X_i$ . Así que, si  $X_i$  punto a es el atributo que puede utilizar sólo los atributos heredados de la no terminal A, el lado izquierdo o el elementos de  $A_i$  de  $X_k$ , lo que significa los atributos heredados de los símbolos a la izquierda de  $i$  o los atributos sintetizados de  $S_k$ ,  $k$  pasando de 1 a  $i$  menos 1. Así que, cuando estamos buscando en la posición  $i-1$  simplemente significa los símbolos a la izquierda de ese particular no terminal  $X_i$ . Por lo tanto, podemos usar el heredado o el sintetizado

atributos de los símbolos a la izquierda de  $X_i$ , así que  $X_1$  a  $X_{i-1}$  solamente.

Así que, eso es lo que queremos decir con la declaración que las dependencias van de izquierda a derecha para escribir, así que no podremos usar cualquier atributo de un símbolo que esté a la derecha de  $X_i$ . Así,  $X_{i+1}$  no puede suministrar cualquier atributo a  $X_i$ , si lo hace entonces no se convierte en L atribuido, la ventaja de las gramáticas atribuidas a la L es que, podemos hacer una serie de evaluaciones de la izquierda para escribir sobre el árbol de análisis y evaluar todos los atributos. Y si restringimos aún más la evaluación de los atributos, de tal manera que se puede hacer en 1 pasada sobre el árbol de análisis de la izquierda para escribir, entonces es llamado LAG 1 o 1 pase LAG.

Así que, 1 pase LAG es el atributo de evaluación puede ser muy se puede hacer fácilmente con L L o sabes que los analizadores de descenso recursivo para el SAG se pueden hacer con LR, LL o RD analizando cualquiera de ellos. Pero, por supuesto, si hay atributos heredados en general no podemos usar el análisis de LR para ir a evaluar los atributos.

Entonces, ¿cómo se hace una evaluación de atributos en el caso de gramáticas con atributos L o LAG, así que asumamos que se nos da un árbol de análisis T con instancias de atributos no evaluados. Y se supone que producimos un árbol de análisis T con evaluación de atributos consistente, valores de atributo después de que la evaluación pase, así que básicamente hacemos una primera búsqueda de profundidad o primera visita de profundidad en el árbol de análisis. Así que, la visita de la raíz n es el punto de partida, Entonces, primero la entrada a un nodo que evaluamos... los atributos heredados, y luego hacer una visita a los internos que bajan al árbol. Y después de que completamos la evaluación de toda la dependencia de este m, evaluamos los atributos sintetizados de n y luego regresar al nivel superior. Así que, primero heredamos entonces visitar el nodo y luego calcular el sintetizado atribuido al nodo y luego regresar, por lo que este es el procedimiento general.

Así que, apliquemos ese procedimiento a nuestros ejemplos aquí, así que este fue el ejemplo de declaración, así que tenemos un usted que sabe que esta gramática también se atribuye a la L. Así que, asegurémonos de que primero, así que aquí el tipo de punto L es igual al tipo de punto T, por lo que el tipo de punto L utiliza sólo el atributo sintetizado de T.

Aquí, por supuesto, sólo tenemos el atributo sintetizado, por lo que no hay violación de ningún regla, lo mismo es cierto para T para flotar así L a I D tenemos I D tipo de punto igual a L tipo, así que el tipo de punto I D es un atributo heredado que utiliza el atributo heredado de L. Así que, ninguna violación L 1 va a L 2 punto I D. Así que, nosotros L 2 punto tipo usa el atributo de los padres y el tipo de punto I D también usa el atributo de los padres, así que no hay violación y el nombre del punto I D es un sintetizado atributo, el nombre del punto de identificación también se sintetiza, por lo que no hay violación de ninguna regla.

Así que, esta es una gramática perfectamente válida atribuida a la L, aquí está el orden de evaluación, así que si lo haces una primera búsqueda de profundidad en este árbol comenzamos por la raíz. Así que, los verdes una vez son todas las visitas por primera vez y la naranja una vez son todas las visitas del mismo nodo una segunda vez.

Por lo tanto, que un total de 25 visitas son necesarias, así que y la evaluación de la visita d f El orden es 1, 2, etc. hasta 25. Así que, eso significa que visitamos 1, luego 2, luego 3 y luego 4, 5 entonces sabes 6 aquí que 7, 8, 9, 10, 11, 12, 13, entonces venimos tú sabes al punto y coma, aunque no tenemos nada que hacer sólo tenemos que visitarlo, luego 15, 16, luego 17, este es el 18, este es el 19, 20, 21, 22, 23, 24, 25.

Así que, cuando hagamos esto, veamos cómo la evaluación de los atributos tiene lugar, así que esta es la primera regla D que va a T L en la que un atributo heredado tendrá que ser evaluado. Pero, ya sabes que realmente no podemos escribir con un punto L igual al tipo de punto T no puede ser evaluado inmediatamente. Así que, lo que realmente hacemos es bajar a 4 y luego 5 y cuando volvamos a 4 en la que es la visita número 6, podemos calcular el atributo sintetizado de T. Entonces estamos listos para calcular el atributo heredado de este L en particular, así que es posible, entonces bajamos de nuevo pasando a I D, este

identificador se pasa a esto. Así que, ese es el final de esta evaluación particular en este subárbol, la evaluación ocurre en este subárbol también de manera similar.

Así que, básicamente debemos hacer una visita  $d f$  y un nodo de enseñanza si las dependencias están satisfechas podremos aplicar una regla de producción más bien de cálculo, evaluar ese particular y luego ir más allá. Así que, aquí a medida que avanzamos, este es el atributo heredado de la  $D$  no es nada, mientras que para la  $T$  no hay ningún atributo heredado, así que bajamos y hay un atributo sintetizado que puede ser evaluado, así que en el camino de retorno, entonces justo antes de la visita a  $L$ , podemos evaluar el atributo heredado de  $L$ . Y luego bajamos a  $I D$  nada que hacer y el atributo heredado de  $I D$  puede ser evaluado y cuando regresemos de aquí, el atributo sintetizado de  $I D$  puede ser evaluado también.

## Vídeo 3

### Gramática de Traducción Atribuida

- ❖ Aparte de las reglas de cálculo de atributos, se añade al AG algún segmento de programa que realiza el cálculo de salida o de algún otro árbol de efectos secundarios.
- ❖ Ejemplos: operaciones de tablas de símbolos, escritura de código generado en un archivo, etc.
- ❖ Como resultado de estos segmentos de código de acción, las órdenes de evaluación pueden verse restringidas.
- ❖ Tales restricciones se añaden al gráfico de dependencia de atributos como bordes implícitos.
- ❖ Estas acciones pueden ser añadidas tanto a los SAG como a los LAG (haciéndolos, SATG y LATG resp.).
- ❖ Nuestra discusión sobre el análisis semántico utilizará LATG (1 - paso) y SATG.

### Ejemplos:



- 1  $Decl \rightarrow DList\$$
- 2  $DList \rightarrow D \mid D ; DList$
- 3  $D \rightarrow T L$
- 4  $T \rightarrow int \mid float$
- 5  $L \rightarrow ID\_ARR \mid ID\_ARR . L$
- 6  $ID\_ARR \rightarrow id \mid id [ DIMLIST ] \mid id BR\_DIMLIST$
- 7  $DIMLIST \rightarrow num \mid num, DIMLIST$
- 8  $BR\_DIMLIST \rightarrow [ num ] \mid [ num ] BR\_DIMLIST$

Note: array declarations have two possibilities  
`int a[10,20,30]; float b[25][35];`

## LATG para Sem. Análisis de Declaraciones Variables - 2

- ❖ La gramática no es LL(1) y por lo tanto no se puede construir un analizador LL(1) a partir de ella.
- ❖ Asumimos que el árbol de análisis está disponible y que la evaluación de atributos se realiza sobre el árbol de análisis.
- ❖ Las modificaciones del CFG para convertirlo en LL(1) y los correspondientes cambios en el AG se dejan como ejercicios.
- ❖ Los atributos y sus reglas de cálculo para las producciones 1 - 4 son como antes y los ignoramos.
- ❖ Proporcionamos el AG sólo para las producciones 5 - 7; el AG para la regla 8 es similar al de la regla 7.
- ❖ El manejo de las declaraciones constantes es similar al de las declaraciones variables.

### Identifier type Information in the Symbol Table

Identifier type information record

name	type	etype	dimlist_ptr
------	------	-------	-------------

1. type: (simple, array)
2. type = simple for non-array names
3. The fields etype and dimlist\_ptr are relevant only for arrays. In that case, type = array
4. etype: (integer, real, error type), is the type of a simple id or the type of the array element
5. dimlist\_ptr points to a list of ranges of the dimensions of an array. C-type array declarations are assumed  
 Ex. `float my_array[5][12][15]`  
 dimlist\_ptr points to the list (5,12,15), and the total number elements in the array is  $5 \times 12 \times 15 = 900$ , which can be obtained by traversing this list and multiplying the elements.

