IC-5701
Compiladores e Intérpretes

Profesor:
Ing. Allan Rodríguez Dávila, MGP

Agenda

TEC

- Árboles Sintácticos
- Ambigüedad y Recursividad
- Autómatas de Pila

Tratamiento de errores

Manejo de Errores

- Reportar la presencia con claridad y precisión.
- Recuperarse de cada error.
- Agregar una sobrecarga mínima.
- Estrategias:
  - Modo pánico
  - Nivel de frase
  - Producción de errores
  - Corrección global

Recuperación Modo pánico

- Al describir un error descarta los símbolos de entrada hasta encontrar un token de sincronización.
  - Delimitadores como ;,) o }
- Es simple y no entra en ciclos

Recuperación a nivel de frase

- Realiza una corrección local sobre la entrada.
  - Sustituye un lexema por alguna cadena que le permita continuar
  - Sustituir una , por ;
  - Insertar ;
- Puede producir ciclos

Producción de errores

- Incluir producciones para identificar errores comunes.
- Detecta errores anticipados cuando entra a una producción de error

Corrección global

- Realizar inserciones, eliminaciones o modificaciones a nivel de árbol sintáctico.
  - Convertir un árbol x con errores en un árbol y sin errores
- Costosos en tiempo y espacio.

CUP

Análisis JFLex y Cup

- **Portafolio #2:** Investigar y documentar sobre Jflex y Cup
  - Características y Estructura del archivo
  - Código dentro de producciones
  - Manejo de errores
  - Instalación
  - Ejemplo y prueba de reconocimiento
- http://jflex.de/
- http://jflex.de/manual.pdf
- http://www2.cs.tum.edu/projects/cup/

Jflex & CUP

- http://www2.cs.tum.edu/projekte/cup/examples.php

```
java -jar jflex-1.6.1.jar
     C:\jflex_1_6_1\lib\Lexer5.flex

java -jar java-cup-11b.jar -interface
     -parser Parser Expresion.cup
```

Jflex & CUP

```
//Genera el archivo del lexer
public void GenerateLexer(String ruta) throws IOException, SilentExit{

    String[] strArr = {ruta};
    jflex.Main.generate(strArr);

}
//Genera los archivos del parser
public void Generateparser(String ruta) throws internal_error, IOException, Exception{
    String[] strArr = {ruta};
    java_cup.Main.main(strArr);

}
```

Lexer

```java
//Usa symbol de cup
public void ejercicioLexer1(String rutaScanear) throws IOException
{
    Reader reader = new BufferedReader(new FileReader (rutaScanear));
    reader.read();
    MyLexer lex = new MyLexer(reader);
    int i = 0;
    Symbol token;
    while(true)
    {
        token = lex.next_token();
        if(token.sym != 0){
            System.out.println("Token: "+token.sym+ ", Valor: "+(token.value==null?lex.yytext():token.value.toString()));
        }
        else{
            System.out.println("Cantidad de lexemas encontrados: "+i);
            return;
        }
        i++;
    }
}
```

Lexer

```
//Usa Yylex
public void ejercicioLexer2(String rutaScanear) throws IOException
{
    Reader reader = new BufferedReader(new FileReader (rutaScanear));
    reader.read();
    BasicLexer.Yylex lex = new BasicLexer.Yylex(reader);
    do {
        lex.yylex();
        System.out.println("++++++++++++++++++++++++++++++++++");
        System.out.println(" Valor: "+lex.yytext());
        System.out.println("++++++++++++++++++++++++++++++++++");
      } while (!lex.zzAtEOF);
}
```

Parser

```java
public void ejercicioParser1(String rutaparsear) throws Exception{
    Reader inputLexer = new FileReader(rutaparsear);
    MyLexer myLexer = new MyLexer(inputLexer);

    parser myParser = new parser(myLexer);
    myParser.parse();
}
```

Parser

```
public static void main(String[] args) throws
IOException, Exception {
   Reader reader =
      new BufferedReader(new FileReader
("C:\\jflex_1_6_1\\ejercicios\\pruebaScanner.c")
);
        reader.read();
        Scanner lex = new Scanner(reader);
        Parser p = new Parser(lex);
      p.parse();
}
```

Ambigüedad

Ambigüedad

- Una gramática es ambigua si produce más de un árbol de análisis sintáctico para cierta cadena.

- En otras palabras es aquella permite más de una derivación de una cadena

TEC

Ambigüedad

expresion ::= *expresion* + *expression*

expresion ::= *expresion* * *expression*

expresion ::= (*expresion*)

expresion ::= **id**

TEC

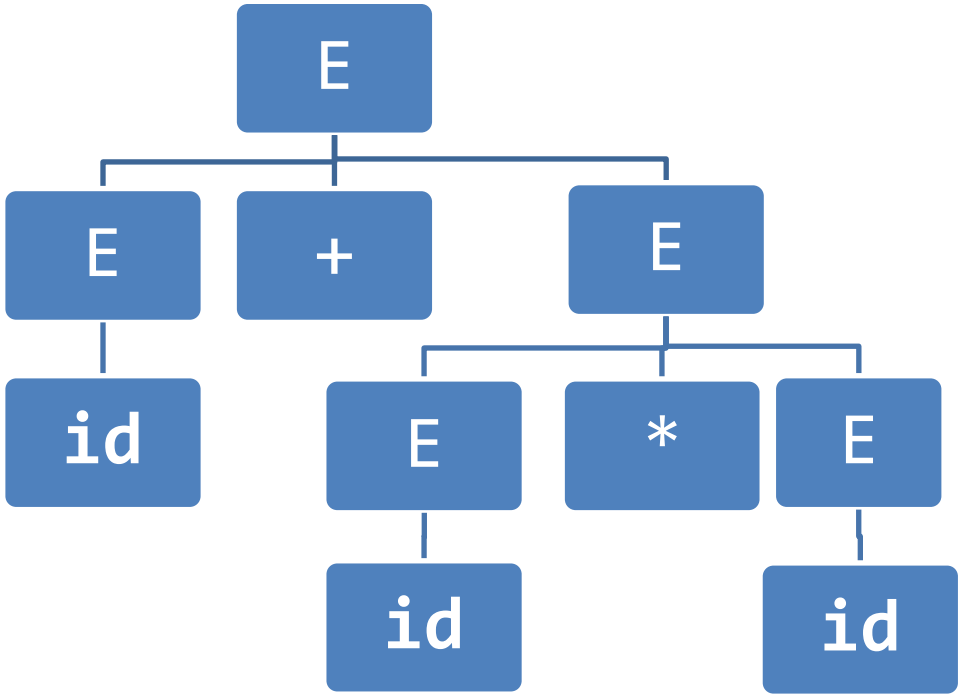
Ambigüedad

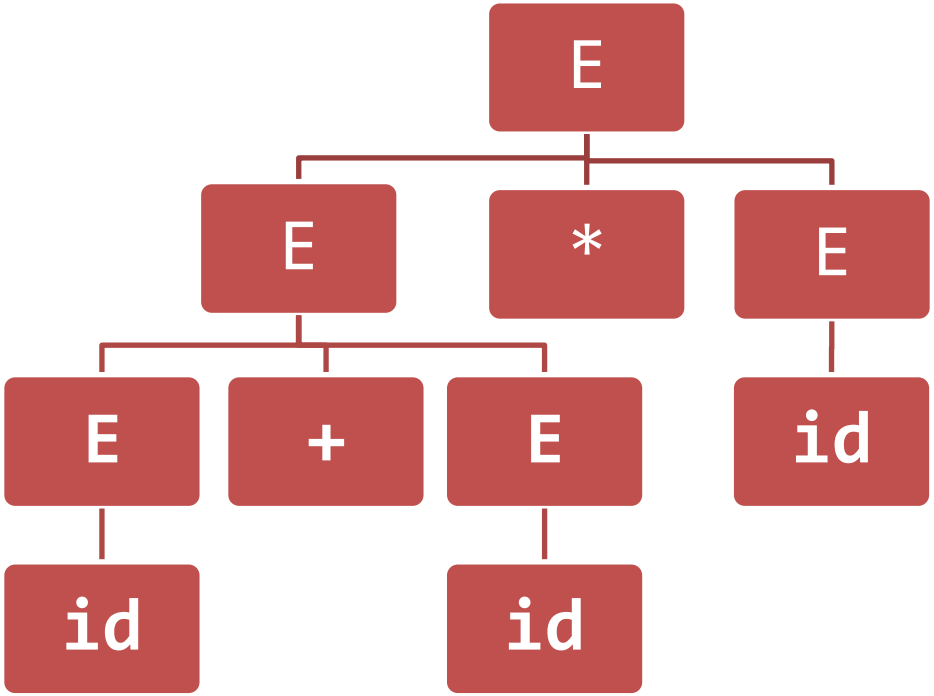
id + id * id

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

TEC

Ambigüedad





TEC

Solución de Ambigüedad

expresion ::= *expresion* + *termino*

expresion ::= *termino*

termino ::= *termino* * *factor*

termino ::= *factor*

factor ::= (*expresion*)

factor ::= **id**

TEC

Ambigüedad

- Uno de los problemas más famosos es el problema del else colgante.

instr → **if** *expr* **then** *instr*

instr → **if** *expr* **then** *instr* **else** *instr*

instr → **otra**

- Genera al menos dos arboles sintácticos para:
  - **if** E_1 **then** S1 **else if** E2 **then** S2 **else** S_3

TEC

Ambigüedad

- Uno de los problemas más famosos es el problema del else colgante.

instr → **if** *expr* **then** *instr*

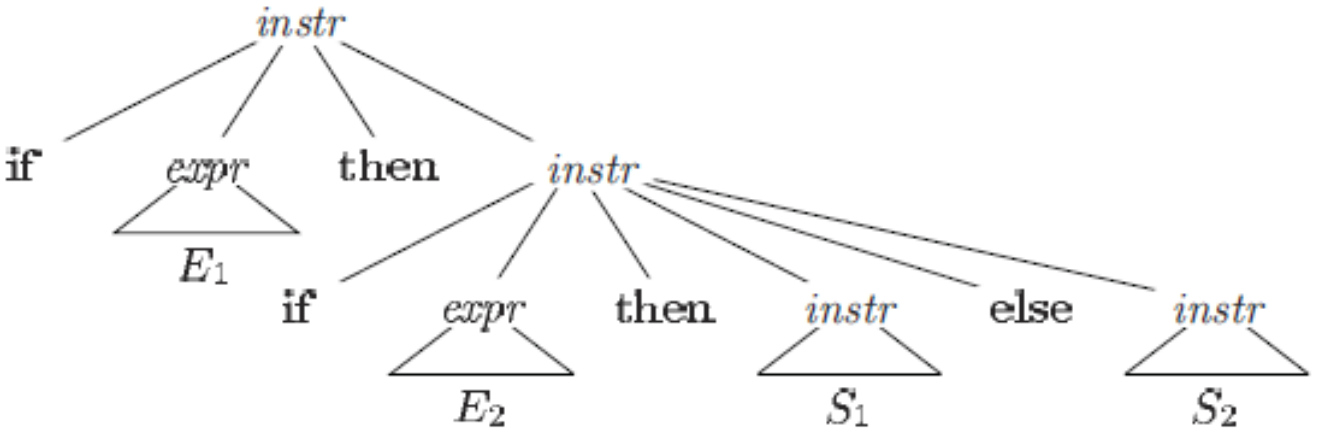
instr → **if** *expr* **then** *instr* **else** *instr*

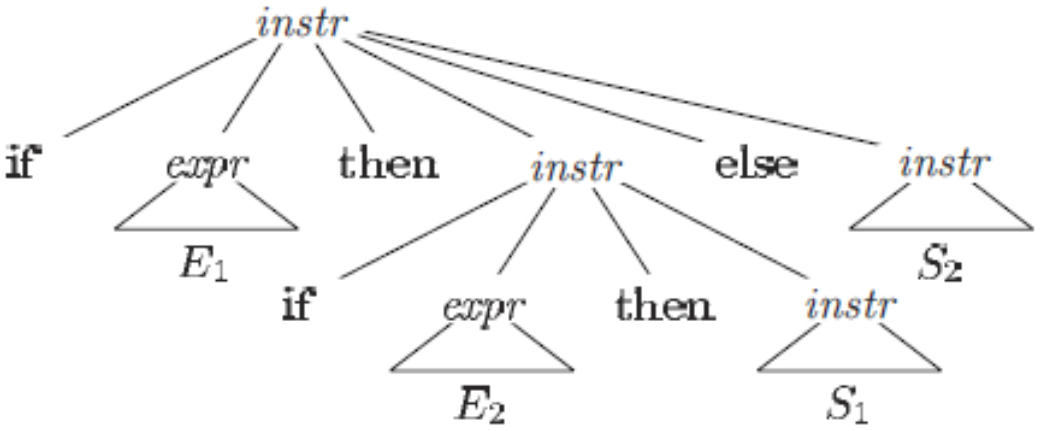
instr → **otra**

- Genera al menos dos arboles sintácticos para:
  - **if** E_1 **then if** E_2 **then** S_1 **else** S_2

TEC

Ambigüedad





Ambigüedad

- En la medida de lo posible se deben evitar ambigüedades en la gramática.
- Los lenguajes con instrucciones condicionales por lo general relacionan cada `else` con el `then` más cercano.
- La corrección puede incorporarse a la gramática, pero en la práctica lo solucionan por "código"

Autómatas de Pila

TEC

Autómatas de Pila

- Las Expresiones regulares generan Lenguajes Regulares.

- Los Autómatas Finitos se utilizan para reconocer Lenguajes Regulares.

Autómatas de Pila

- Las Gramáticas Libres de Contexto generan Lenguajes Libres de Contexto.

- Los Autómatas Autómatas de Pila se utilizan para reconocer Lenguajes Libres de Contexto.

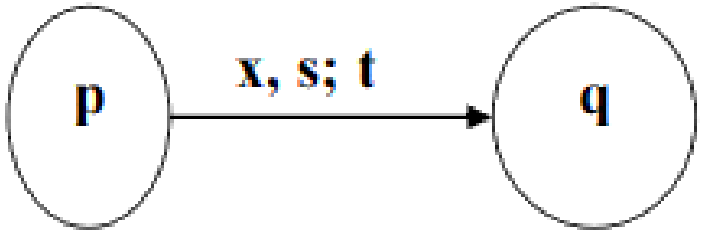
TEC

Autómatas de Pila

- Son una extensión de los Autómatas Finitos No Deterministas.
- Almacenan gran cantidad de símbolos en una pila.
  - # significa pila vacía o símbolo inicial de pila
- Las transiciones se realizan basados en la cadenas de entrada y en los símbolos de la pila

Autómatas de Pila

- Las transiciones se representan como `(p, x, s; q, t)`
  - `p` = estado inicial
  - `q` = estado al que llega
  - `x` = símbolo de la cadena de entrada
  - `s` = símbolo que se desapila
  - `t` = símbolo que se apila



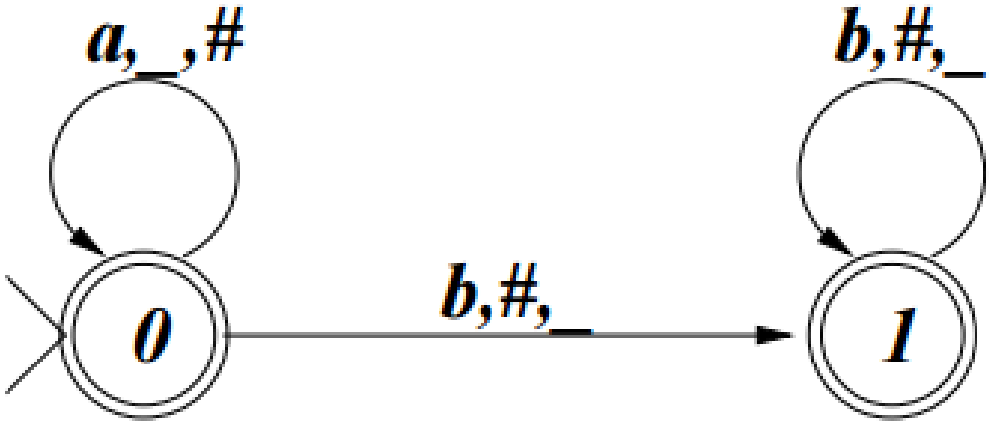
TEC

Autómatas de Pila

- Se define como $(\Sigma, P, Q, A_0, q_0, f, F)$:
  - Σ = alfabeto de entrada
  - P = alfabeto de la pila
  - Q = conjunto de estados
  - A_0 = símbolo inicial de la pila (#)
  - q_0 = símbolo inicial del conjunto de estados
  - f = función de transición de estados
  - F = conjunto de estados de aceptación

TEC

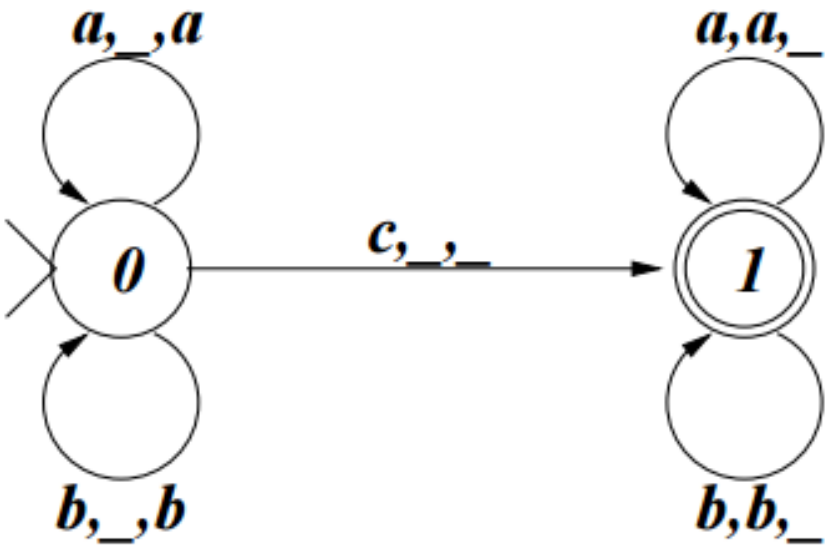
Autómatas de Pila



TEC

Autómatas de Pila

- Gramática:
  - S → A
  - A → aAa
  - A → bAb
  - A → c



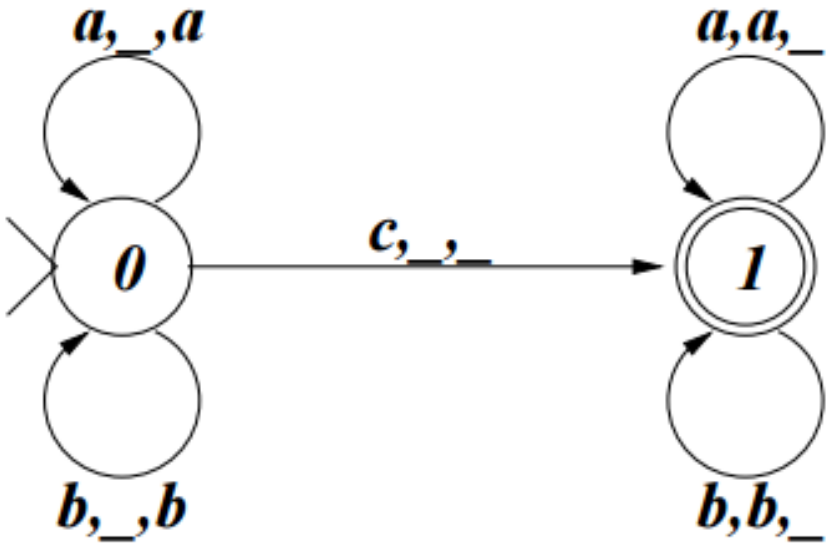
Simulación con Gramática

- Gramática:
  - S → A
  - A → aAa
  - A → bAb
  - A → c
- Ejemplo:
  - abcba

Pila	Entrada	Acción
#	abcba	Leer();
#	abcba	S→A
A#	abcba	A→aAa
aAa#	abcba	Desapila a; Leer();
Aa#	bcba	A→bAb
bAba#	bcba	Desapila b; Leer();
Aba#	cba	A→c
cba#	cba	Desapila c; Leer();
ba#	ba	Desapila b; Leer();
a#	a	Desapila a; Leer();
#	?	Aceptar();

Simulación con Autómata

- AP:



- Ejemplo:
  - abcba
  - abcab

Pila	Entrada	Acción
#	abcba	-
#	abcba	(0,a,_,a;0)
a#	abcba	(0,b,_,b;0)
ba#	abcba	(0,c,_,_;1)
ba#	abcba	(1,b,b,_;1)
a#	abcba	(1,a,a,_;1)
#	abcba?	-

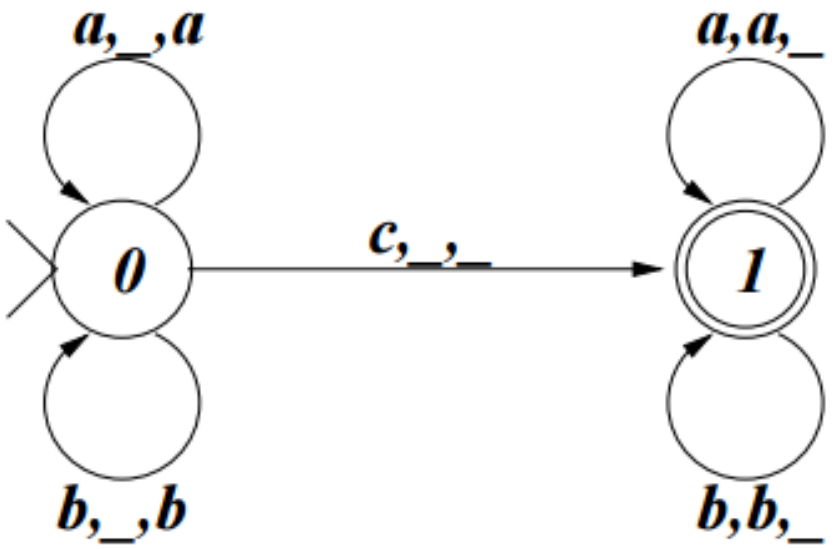
Bonus: Simulación con Gramática

- Gramática:
  - S → A
  - A → aAa
  - A → bAb
  - A → c
- Ejercicios:
  - Ababacababa
  - int main () { int a = read()$ a = a*2$}
    - Gramática Propuesta por el profesor

TEC

Bonus: Simulación con AP

- Autómata de Pila:
  - S → A
  - A → aAa
  - A → bAb
  - A → c



- Ejercicios:
  - abababacababa
  - bbbcbbb
  - bbbacabbb

Bibliografía

- Programming Language Processors in Java: compilers and interpreters. Watt, David, Brown, Deryck. Pearson Education. 2000

- Compilers: principles, techniques and tools (2da. ed.). Aho, Alfred. Pearson Education. 2007


TEC
Tecnológico
de Costa Rica