Compiladores

Notas de clase basadas en
Construcción de compiladores de Kenneth C. Louden

Dr. Francisco Javier Zaragoza Martínez
franz@correo.azc.uam.mx

UAM Azcapotzalco
Departamento de Sistemas

Trimestre 2011 Invierno

Contenido

Exámenes y tareas

- Cinco exámenes parciales:
  - Individuales.
  - Cada dos semanas.
  - Por cada tema del curso.
  - Fuera del horario de clase (Moodle).
- Cinco tareas de programación:
  - Individuales.
  - En C o C++ (ningún otro lenguaje).
  - En gcc o g++ para Linux (ningún otro ambiente).
  - Deberán funcionar en el servidor callix.
- Cada examen y cada tarea valdrá 10 puntos.

Otros detalles

- Para aprobar se requiere al menos 30 puntos en cada parte y:
  - Al menos 60 puntos para S.
  - Al menos 73 puntos para B.
  - Al menos 87 puntos para MB.
- Los exámenes tendrán un horario amplio pero un tiempo limitado de aplicación.
- El código fuente de las tareas se deberá enviar a tiempo desde callix.
- No guardaré calificación.

Contenido

Contenido

¿Qué es un compilador?

Programa fuente ⟶ Compilador ⟶ Programa objetivo

- Un compilador es un programa que traduce de un lenguaje a otro.
- Su entrada es un programa escrito en un lenguaje fuente.
- Su salida es un programa escrito en un lenguaje objetivo.
- Los programas fuente y objetivo son equivalentes.
- Normalmente el lenguaje fuente es de alto nivel como C o C++.
- Normalmente el lenguaje objetivo es de bajo nivel como C o ASM.

¿Por qué estudiar compiladores?

- Los compiladores suelen ser programas muy complejos, difíciles de escribir y de entender.
- Pero los compiladores se usan en todos los aspectos prácticos de la computación.
- Una tarea frecuente es la de escribir aplicaciones con interfaces e intérpretes de instrucciones, más pequeños que un compilador pero usando las mismas técnicas.
- En este curso estudiaremos las técnicas básicas que permiten la escritura de compiladores y de estas otras aplicaciones.

Contenido

Lenguaje máquina y lenguaje ensamblador

- Con las computadoras de programa almacenado se comenzaron a escribir secuencias de códigos para realizar cálculos (40s).
- Estos programas se escribían en lenguaje de máquina. Por ejemplo

  C7 06 0000 0002

  quiere decir mover el número 2 a la dirección 0 en un procesador x86.
- Esta actividad fue reemplazada por la escritura de programas en lenguaje ensamblador donde las instrucciones y direcciones de memoria pueden ser simbólicas. El ejemplo anterior se escribiría

  MOV X, 2

  si el símbolo X representa la dirección 0.
- Los programas que traducen de lenguaje ensamblador a lenguaje de máquina se llaman ensambladores.

Lenguaje ensamblador e independencia de la máquina

- El lenguaje ensamblador todavía se usa cuando se requiere una gran velocidad o un código muy pequeño: videojuegos, sistemas embebidos.
- Sin embargo tiene varios defectos:
  - Aún no es fácil de escribir.
  - Es difícil de leer y comprender.
  - Es completamente dependiente de la máquina.
  - Por lo que el código debe reescribirse todo el tiempo.
- Por lo tanto, el siguiente paso natural era la posibilidad de escribir programas en una notación más natural, independientes de la máquina y que todavía se pudieran traducir a lenguaje de máquina.
- Por ejemplo $X = 2$.

Compiladores y teoría de lenguajes

- Al principio se creyó que esto era imposible o poco eficiente.
- Esto fue desmentido con el desarrollo de FORTRAN por Backus (50s).
- Al mismo tiempo Chomsky comenzó a estudiar la teoría de lenguajes.
- La jerarquía de Chomsky clasifica a los lenguajes según la complejidad de sus gramáticas y los algoritmos necesarios para reconocerlos.
- Los lenguajes regulares nos resultarán útiles para llevar a cabo el proceso de análisis léxico.
- Los lenguajes libres de contexto son la forma estándar de representar la estructura de los lenguajes de programación y son fundamentales para el proceso de análisis sintáctico.

Generación de código

- El desarrollo de métodos para la generación de código objeto es mucho más complejo y continúa hasta nuestros días.
- Estas técnicas de mejoramiento de código tienen como objetivo aumentar la velocidad del código ejecutable o disminuir su longitud.
- También se logró automatizar parte del desarrollo de los compiladores con los generadores de analizadores sintácticos y los generadores de analizadores léxicos (70s).

Contenido

Intérpretes y preprocesadores

- Un intérprete es un traductor que en vez de generar código objeto ejecuta el programa fuente inmediatamente.
- Cualquier lenguaje se puede interpretar o compilar, pero a veces se prefiere un intérprete dependiendo de la situación específica:
  - Algunos lenguajes son usualmente interpretados: BASIC, LISP, PHP.
  - En situaciones de enseñanza o de desarrollo de software.
- Un preprocesador es un programa separado que se ejecuta antes de la traducción real de un programa fuente.
- Generalmente se usan para eliminar comentarios, incluir archivos, hacer sustituciones de macros, efectuar compilación condicional, etc.

Ensambladores, ligadores y cargadores

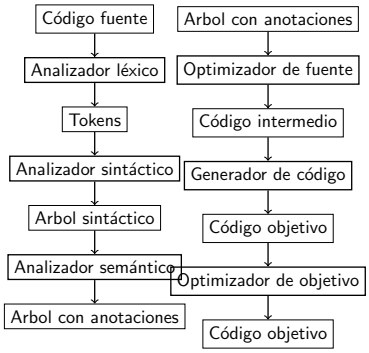
- Un ensamblador es un traductor del lenguaje ensamblador al lenguaje de máquina de una computadora en particular.
- Un ligador es un programa que recopila el código objeto de varios programas que se compilaron o ensamblaron de forma separada (por ejemplo las bibliotecas) en un solo programa ejecutable.
- Un cargador es un programa que se encarga de llevar un programa ejecutable a cualquier parte de la memoria de modo que se pueda ejecutar.

Editores, depuradores, perfiladores y administradores

- Un editor es un programa que se puede usar para escribir cualquier archivo de texto. Sin embargo, hay editores especializados para escribir programas llamados editores basados en estructura.
- Un depurador es un programa que se usa para determinar los errores de ejecución de un programa compilado.
- Un perfilador es un programa que junta estadísticas acerca de la ejecución de un programa, como el número de veces que se llamó a un procedimiento o la cantidad de tiempo que ocupa cada uno de ellos.
- Un administrador de proyecto es un programa que sirve para coordinar el trabajo de escritura de un gran proyecto de software en el que se involucra a más de un programador.

Contenido

Fases del proceso de traducción



Componentes auxiliares de un compilador

- Tabla de literales.
- Tabla de símbolos.
- Manejador de errores.

Analizador léxico

- Esta fase lee el programa fuente como un flujo de caracteres.
- El análisis léxico junta secuencias de caracteres en tokens.
- Como ejemplo, el código en C

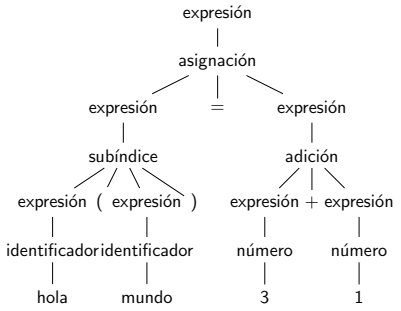
  hola[mundo]=3+1

  contiene 15 caracteres, pero sólo ocho tokens:
  1. hola (identificador)
  2. [(corchete izquierdo)
  3. mundo (identificador)
  4.] (corchete derecho)
  5. = (asignación)
  6. 3 (número)
  7. + (suma)
  8. 1 (número)
- Un analizador léxico también puede introducir identificadores en la tabla de símbolos y literales en la tabla de literales.

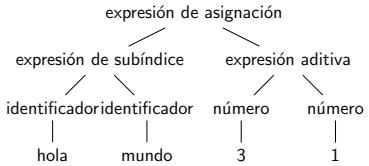
Analizador sintáctico

- Esta fase recibe la secuencia de tokens y determina los elementos estructurales del programa.
- El resultado del análisis sintáctico es un árbol sintáctico, también llamado árbol de análisis gramatical.
- Estos árboles son auxiliares útiles para visualizar la sintaxis de un programa pero no son una representación eficiente.
- En lugar de eso, se usan los árboles sintácticos abstractos.

Ejemplo de árbol sintáctico



Ejemplo de árbol sintáctico abstracto



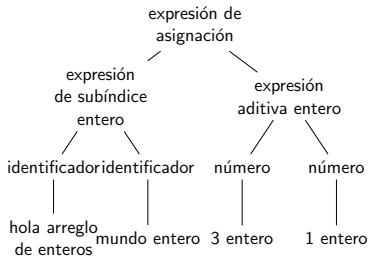
Analizador semántico

- La semántica de un programa es su significado.
- La semántica determina el comportamiento de un programa en ejecución.
- La mayoría de los lenguajes de programación tienen características que se pueden determinar antes de la ejecución.
- A estas se les llama la semántica estática.
- Típicamente incluyen a la verificación de tipos.
- Por ejemplo, el código en C

  hola[mundo]=3+1

  requiere que hola sea arreglo de enteros y que mundo sea entero.

Ejemplo de árbol sintáctico abstracto con anotaciones



Optimizador (mejorador) de código fuente

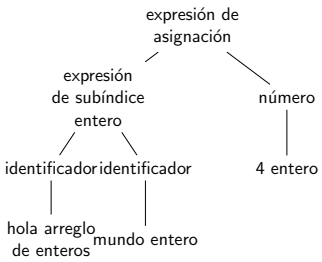
- Muchos compiladores incluyen etapas para el mejoramiento del código.
- El primer lugar donde se puede hacer esto es después del análisis semántico.
- Algunas de estas mejoras sólo dependen del código fuente.
- Por ejemplo, el código en C

  hola[mundo]=3+1

  permite precalcular la expresión del lado derecho para que quede

  hola[mundo]=4.
- A esto se le llama incorporación de constantes.

Ejemplo de árbol sintáctico abstracto con incorporación de constantes



Generador de código

- El generador de código toma cualquier representación interna del código (código intermedio) y genera código para la máquina objetivo.
- Aquí usaremos código ensamblador, aunque la mayoría de los compiladores genera código máquina de manera directa.
- Es en esta etapa cuando las propiedades de la máquina objetivo se convierten en el factor principal: se deben usar instrucciones que existan y se debe tener cuidado con la representación interna de los datos.

Ejemplo de código generado

- Una posible secuencia de código generado para nuestro ejemplo sería:
- Usamos la convención de C para los modos de direccionamiento.

```
MOV R0, mundo ;; valor de mundo a R0
MUL R0, 2     ;; doble valor en R0
MOV R1, &hola ;; dirección de hola a R1
ADD R1, R0    ;; sumar R0 a R1
MOV *R1, 4    ;; constante 4 a dirección en R1
```


Optimizador (mejorador) de código objetivo

- En esta fase se intenta mejorar el código objetivo generado por el generador de código.
- Dichas mejoras incluyen:
  - Selección de modos de direccionamiento para mejorar el rendimiento.
  - Reemplazar instrucciones lentas por rápidas.
  - Eliminación de operaciones redundantes o innecesarias.
- Nuestro ejemplo podría quedar así:

```
MOV R0, mundo    ;; valor de mundo a R0
SHL R0           ;; doble valor en R0
MOV &hola[R0], 4 ;; constante 4 a dirección de hola + R0
```


Contenido

Estructuras de datos principales en un compilador

- Existe una gran interacción entre las fases del compilador y las estructuras de datos que soportan esas fases.
- Por lo tanto es importante que cuando se implementen se haga de la forma más eficaz posible sin aumentar la complejidad.
- Idealmente, un compilador debe compilar un programa en tiempo proporcional al número de caracteres del mismo.
- A continuación describiremos algunas de las estructuras de datos principales que existen en un compilador.

Tokens

- Cuando el analizador léxico reúne los caracteres en un token, generalmente lo representa de manera simbólica.
- Para esto se usa un valor de un tipo de datos enumerado que represente al conjunto de tokens del lenguaje.
- A veces se necesita también almacenar información adicional: el nombre de un token identificador o el valor de un token literal.
- En la mayoría de los lenguajes sólo se procesa un token a la vez, pero en otros se requiere un arreglo de tokens.

Árbol sintáctico

- El árbol sintáctico se construye como una estructura de datos dinámica que crece conforme se lleva a cabo el análisis sintáctico.
- Cada nodo del árbol es un registro cuyos campos almacenan la información obtenida por el análisis sintáctico (y posteriormente el análisis semántico).
- Por ejemplo, el tipo de datos de una expresión se puede almacenar en un campo del nodo correspondiente.

Tabla de símbolos

- Esta estructura mantiene la información asociada con los identificadores:
  - Funciones.
  - Variables.
  - Constantes.
  - Tipos de datos.
- La tabla de símbolos interactúa con todas las fases del compilador.
- Debido a esto, las operaciones de inserción, eliminación y acceso deben ser muy eficientes, incluso de tiempo constante.
- Generalmente se usa una tabla de dispersión.

Tabla de literales

- Esta estructura mantiene la información asociada con las literales (constantes y cadenas usadas en el programa).
- La búsqueda e inserción rápida también son esenciales, pero la eliminación no ocurre en esta estructura.
- Esta tabla es importante porque permite la reutilización de constantes y cadenas y, por lo tanto, la reducción de tamaño del programa.

Código intermedio

- Dependiendo de la clase de código intermedio generado, se puede almacenar de diversas formas:
  - Arreglo de cadenas de texto.
  - Archivo de texto temporal.
  - Lista ligada.
- La representación empleada también influye en el tiempo de ejecución de las etapas de mejoramiento de código.

Contenido

Análisis y síntesis

- La estructura de un compilador también se puede describir como una etapa de análisis seguida de una etapa de síntesis.
- El análisis corresponde con las etapas del compilador que analizan el programa fuente para calcular sus propiedades:
  - Análisis léxico.
  - Análisis sintáctico.
  - Análisis semántico.
  - Técnicas matemáticas y algorítmicas.
- La síntesis corresponde con las etapas del compilador que generan el código traducido.
  - Generación de código.
  - Mejoramiento de código.
  - Técnicas más especializadas.

Etapa inicial y etapa final

- Otra forma de ver el proceso de compilación lo divide en etapas que dependen del lenguaje fuente o del lenguaje objetivo.
- La etapa inicial depende sólo del lenguaje fuente.
- La etapa final depende sólo del lenguaje objetivo.
- Estas dos etapas quedan divididas por el código intermedio, lo cual resulta muy importante para la portabilidad del compilador.
- Desafortunadamente este ideal ha sido imposible de conseguir.

Pasadas

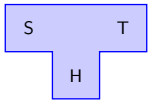
- A cada lectura del programa fuente se le llama una pasada.
- Algunos compiladores llevan a cabo más de una pasada:
  - Una para el análisis léxico.
  - Una para construir el árbol sintáctico.
  - Una para construir el código intermedio.
  - Una para construir el código objetivo, etc.
- Algunos lenguajes como C permiten la compilación en una pasada.
- Sin embargo, los compiladores con mejoramiento de código suelen hacer varias pasadas: ocho no es fuera de lo común.

Contenido

Lenguajes involucrados en el proceso

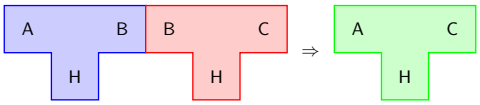
- En realidad hay tres lenguajes involucrados en el proceso de compilación:
  - El lenguaje fuente.
  - El lenguaje objeto.
  - El lenguaje anfitrión
- Este último es el lenguaje en el que está escrito el compilador.
- Históricamente, este lenguaje comenzó siendo el lenguaje de máquina.
- Pero actualmente se escriben en lenguajes para los que ya existen compiladores para las máquinas objetivo.

Diagramas T



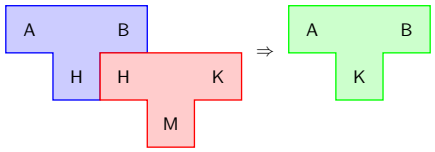
- Muchas situaciones se pueden describir mediante un diagrama en el que se indican los lenguajes fuente S, objetivo T y anfitrión H.
- Este diagrama es equivalente a decir que el compilador correspondiente traduce de S a T ejecutándose sobre una máquina H.
- Típicamente H es igual a T, pero esto puede variar.

Primer tipo de combinación de diagramas T



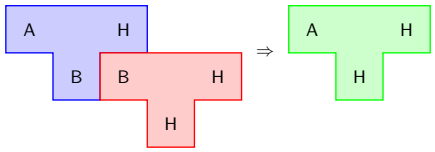
- Si tenemos dos compiladores de A a B y de B a C que se ejecutan en la misma máquina H entonces obtenemos un compilador de A a C que se ejecuta también en la máquina H.

Segundo tipo de combinación de diagramas T



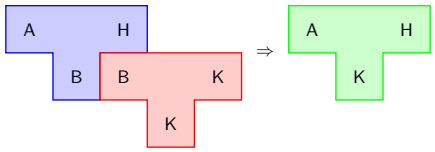
- Podemos usar un compilador de la máquina H a la máquina K para traducir el lenguaje de implementación de otro compilador de H a K.

Traducción de compiladores



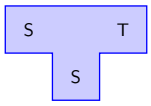
- Podemos utilizar un compilador para el lenguaje B en la máquina H para traducir un compilador de A a H escrito en el lenguaje B.

Compilación cruzada



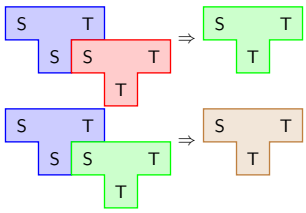
- Podemos utilizar un compilador para el lenguaje B en la máquina K para obtener un compilador cruzado de A a H en la máquina K.

Compilador escrito en el lenguaje que compila



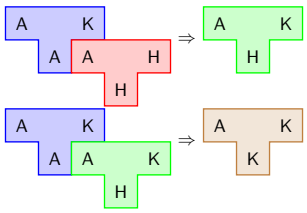
- Es común escribir un compilador en el mismo lenguaje que compila.
- Aunque parece que esto es un problema debido a que el compilador no se puede compilar a sí mismo, esto resulta muy útil.
- Veamos dos casos.

Arranque automático



- El compilador azul está escrito en su propio lenguaje.
- El compilador rojo (limitado) está escrito en lenguaje máquina.
- El compilador verde es ineficiente.
- El compilador café es la versión final.

Portabilidad



- El compilador azul está redirigido a la máquina K.
- El compilador rojo se ejecuta en la máquina H.
- El compilador verde es un compilador cruzado.
- El compilador café se ejecuta en la máquina K.

Contenido

Lenguaje TINY

- Un programa en TINY tiene una estructura muy simple: es una secuencia de sentencias separadas por puntos y coma.
- Todas las variables son enteras y se declaran automáticamente.
- Existen solamente dos secuencias de control: if y repeat.
- Existen sentencias de lectura y escritura.
- Se permiten comentarios no anidados.
- Las expresiones son aritméticas y booleanas.

Ejemplo de programa en TINY

```
{ Programa de muestra
  en lenguaje TINY -
  calcula el factorial
}
read x; { introducir un entero }
if x > 0 then { no calcule si x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1;
  until x = 0;
  write fact { salida del factorial de x }
end
```


Máquina TM

- El lenguaje objetivo será un ensamblador para la máquina TM, de la cual mostramos un ejemplo de traducción del código C `a[i] = 6`.
- Note que hay tres modos de direccionamiento: LDC es constante de carga, LD es carga de memoria y LDA es dirección de carga.
- Todas las direcciones se dan como registro más desplazamiento.

```
LDC 1,0(0)  carga 0 en registro 1
* supone que i está en la posición 10 de memoria
LD  0,10(1) carga valor para 10+R1 en R0
LDC 1,2(0)  carga 2 en registro 1
MUL 0,1,0   pon R1*R0 en R0
LDC 1,0(0)  carga 0 en registro 1
* supone que a comienza en la posición 20 de memoria
LDA 1,20(1) carga 20+R1 en R0
ADD 0,1,0   pon R1+R0 en R0
LDC 1,6(0)  carga 6 en registro 1
ST  1,0(0)  almacena R1 en 0+R0
```

Contenido

Contenido

El proceso del análisis léxico

- El trabajo del analizador léxico es leer los caracteres del código fuente y agruparlos en unidades lógicas llamadas tokens.
- Formar tokens es parecido a deletrear palabras en español.
- En un compilador los tokens se definen como un tipo enumerado:

```
typedef enum {
   IF, THEN, ELSE, PLUS, MINUS, NUM, ID, ...} TokenType;
```

- Los tokens caen en diversas categorías:
  - Las palabras reservadas como `IF` y `THEN` que representan a las cadenas de caracteres `if` y `then`.
  - Los símbolos especiales como `PLUS` y `MINUS` que representan a los caracteres `+` y `-`.
  - Las literales como `NUM` que representa a los números.
  - Los identificadores como `ID` que representa a los nombres.

Lexemas

- Note que los primeros dos tipos de token representan a una única cadena de caracteres y que los últimos dos representan a varias cadenas de caracteres.
- Para distinguir claramente entre un token y la cadena que representa se le llama a esta última su lexema.
- Por ejemplo, el token `IF` tiene a la cadena `if` como lexema.
- Los tokens que corresponden con las palabras reservadas y los símbolos especiales tienen un único lexema.
- Los tokens que corresponden con literales representan una cantidad infinita de lexemas, como `NUM` y los números.
- Los tokens que corresponden con identificadores también representan una cantidad infinita de lexemas.

Atributos

- Cualquier valor asociado con un token se denomina atributo.
- Ejemplos de atributos de tokens pueden ser:
  - El lexema del token.
  - Un valor de cadena (lo que está escrito en el código fuente).
  - Un valor numérico (que se calcula del valor de cadena).
  - El significado de un símbolo especial (como un operador).
- El analizador léxico debe calcular tantos atributos de cada token como sea necesario para la siguiente fase, pero a veces calcula todos los posibles.

Registro de token

- A menudo es útil recolectar todos los atributos de un token en un tipo estructurado al que llamaremos registro de token:

```
typedef struct {
   TokenType tokenval;
   char*     stringval;
   int       numval;
} TokenRecord;
```

- O posiblemente como una unión, suponiendo que no se usan los valores numérico y de cadena al mismo tiempo:

```
typedef struct {
   TokenType tokenval;
   union {
      char* stringval;
      int   numval;
   } attribute;
} TokenRecord;
```


Funcionamiento del analizador léxico

- Aunque la tarea del analizador léxico es la de convertir todo el programa fuente en una secuencia de tokens, normalmente no lo hace todo a la vez.
- En lugar de eso lo hará bajo el control del analizador sintáctico, simplemente devolviendo el siguiente token de la entrada:

```
TokenType getToken(void);
```

- Esta función devolverá el siguiente token de la entrada y calculará sus atributos, guardando los resultados en un buffer.

Ejemplo del funcionamiento de `getToken`

- Considere la línea de código fuente:

```
a[index] = 4 + 2
```

- Suponga que esta línea de entrada se almacena en un buffer:

	a	[	i	n	d	e	x	]		=		4		+		2

- Entonces una llamada a `getToken` deberá saltarse el primer espacio, reconocer a la cadena a como un token y devolver el valor de token ID, de modo que la siguiente llamada a `getToken` comenzará a leer el carácter `[`.

Contenido

Expresiones regulares

- Las expresiones regulares representan patrones simples de caracteres.
- Una expresión regular r define un lenguaje $L(r)$: el conjunto de las cadenas con las que concuerda.
- A $L(r)$ se le llama el lenguaje generado por r.
- Este lenguaje depende del conjunto de caracteres disponible.
- A este conjunto se le llama alfabeto y se suele denotar por Σ.

Definición de expresiones regulares

- Las expresiones regulares se definen de manera recursiva como sigue:
  - La cadena vacía ϵ es una expresión regular con $L(\epsilon) = \{\epsilon\}$.
  - Si $a \in \Sigma$ entonces a es una expresión regular con $L(a) = \{a\}$.
  - El conjunto vacío $\emptyset$ es una expresión regular con $L(\emptyset) = \{\}$
  - Note la diferencia entre $L(\epsilon)$ y $L(\emptyset)$.
  - Si r y s son expresiones regulares entonces $r|s$ es una expresión regular con $L(r|s) = L(r) \cup L(s)$, la unión de r y s.
  - Si r y s son expresiones regulares entonces rs es una expresión regular con $L(rs) = L(r)L(s)$, la concatenación de r y s.
  - Si r es una expresión regular entonces r^* es una expresión regular con $L(r^*) = L(r)^*$, la cerradura de Kleene de r.
- Ninguna otra cosa es una expresión regular.

Ejemplos de expresiones y lenguajes regulares

- $L(a|b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$.
- $L((a|b)c) = L(a|b)L(c) = \{a, b\}\{c\} = \{ac, bc\}$.
- $L(((a|b)c)^*) = \{ac, bc\}^* = \{\epsilon, ac, bc, acac, acbc, bcac, bcbc, \ldots\}$.
- Si $D = L(0|1|2|3|4|5|6|7|8|9)$ entonces D es el conjunto de dígitos.
- Si $E = DD^*$ entonces E es el conjunto de los enteros en decimal.
- Si $V = L(a|e|i|o|u)$ entonces V es el conjunto de vocales.
- El lenguaje a^*ba^* consta de las cadenas con exactamente una b.
- El lenguaje $a^*(b|\epsilon)a^*$ consta de las cadenas con una b o menos.

Lenguajes que no son regulares

- Es muy fácil encontrar ejemplos de lenguajes que no son regulares.
- Uno de ellos es el lenguaje

$$S = \{a^n b a^n | n \geq 0\}$$

  que consta de las cadenas con la misma cantidad de a antes y después de una b.
- Demostrar que S no es regular requiere de un resultado sobre lenguajes regulares llamado el lema de bombeo.

Extensiones para las expresiones regulares

- Si r es una expresión regular entonces $r^+ = rr^*$.
- La expresión regular Σ representa a cualquier carácter.
- El intervalo $a|b|\cdots|z$ también se escribe $[a-z]$.
- Cualquier subconjunto $A \subseteq \Sigma$ se escribe A.
- Observe que $\neg A = \Sigma \setminus A \subseteq \Sigma$.
- Si r es una expresión regular entonces $r? = \epsilon|r$.

Expresiones regulares para tokens

- No es difícil dar expresiones regulares que definan a los diferentes tokens de un lenguaje de programación.
- En particular, veremos los siguientes casos:
  - Números enteros.
  - Números flotantes.
  - Palabras reservadas.
  - Identificadores.
  - Comentarios.

Números enteros

- Los dígitos decimales son $0, 1, \ldots, 9$, por lo tanto podemos usar la expresión regular

$$D = 0|1|\cdots|9 = [0-9].$$

- Los números naturales están hechos de uno o más digitos, por lo que podemos usar la expresión regular

$$N = D^+ = [0-9]^+.$$

- Si se quiere excluir a los que empiezan con cero (pero no al 0) entonces

$$N = 0|[1-9][0-9]^*.$$

- Los números enteros son los naturales posiblemente con signo

$$Z = (\epsilon|+|-)N = (+|-)?N.$$

Números flotantes

- Hay dos formas de escribir los números flotantes: con un punto decimal o con notación exponencial.
- En el primer caso son un entero seguido de un punto seguido de otro natural, por lo que podemos usar la expresión regular

$$F = Z.N.$$

- En el segundo caso debemos considerar que posiblemente incluyan la letra e y un entero para el exponente

$$E = FeZ.$$

Palabras reservadas

- Estas son las más fáciles de escribir como expresiones regulares.
- Simplemente necesitamos una lista de cadenas que correspondan con estas.
- $R = \text{if}|\text{while}|\text{do}|\ldots$

Identificadores

- Generalmente un identificador comienza con una letra y sigue con una secuencia de letras o dígitos.
- Recordemos que los dígitos se definen así

$$D = 0|1|\cdots|9 = [0-9].$$

- Y las letras se pueden definir así

$$L = a|b|\cdots|z|A|B|\cdots|Z = [a-zA-Z].$$

- Por lo que los identificadores se pueden describir con

$$I = L(L|D)^*.$$

Comentarios

- Curiosamente, casi siempre es más difícil describir un comentario como una expresión regular excepto para casos sencillos.
- En Pascal un comentario comienza con una llave izquierda y termina con una llave derecha, así que

$$C_{\text{Pascal}} = \{(\neg\})^*\}.$$

- En Scheme un comentario comienza con un punto y coma y termina con un carácter de nueva línea, así que

$$C_{\text{Scheme}} = ;(\neg NL)^*NL.$$

- En C la situación se complica porque los delimitadores de comentarios constan de más de un carácter.
- En Modula el problema es que los comentarios pueden estar anidados.

Ambigüedad

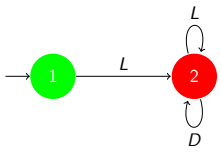
- En ocasiones una misma cadena se puede interpretar como distintas sucesiones de tokens.
- Por ejemplo una palabra reservada también es un identificador.
- Otro ejemplo es la cadena <> que se podría interpretar como dos tokens menor que y mayor que o un token distinto.
- Esto generalmente se resuelve aplicando una de dos reglas:
  - Si una cadena puede ser palabra reservada se interpreta de esta forma.
  - Si una cadena puede extenderse para formar un token se hará esto.
- Así que la única pregunta adicional que queda es cómo saber dónde termina un token y a esto generalmente se le llama un espacio en blanco, el cual puede ser una nueva línea, un espacio, un tabulador o un comentario.

Contenido

Autómatas finitos

- Los autómatas finitos son modelos matemáticos para describir cierto tipo de algoritmos.
- En particular se pueden usar para reconocer patrones de cadenas.
- Por lo tanto se pueden usar para construir analizadores léxicos.
- Se puede ver que los autómatas finitos reconocen exactamente a los lenguajes regulares.

Ejemplo de un autómata finito



- Si las letras están definidas por L y los dígitos están definidos por D entonces los identificadores se pueden definir por $I = L(L|D)^*$.
- Esto representa a cualquier cadena que comienza con una letra y sigue con cero o más dígitos o letras.

¿Qué significa el diagrama anterior?

- Los círculos representan estados.
- Las flechas representan transiciones debidas a sus etiquetas.
- El círculo verde representa el estado inicial.
- Los círculos rojos representan estados de aceptación.
- El proceso de reconocimiento se puede representar por una secuencia de estados que comienza en el estado inicial y que tiene una transición por cada carácter de la entrada hasta llegar a un estado de aceptación, como en

$$\rightarrow 1 \rightarrow^{x} 2 \rightarrow^{t} 2 \rightarrow^{m} 2 \rightarrow^{p} 2 \rightarrow^{7} 2 \rightarrow^{2} 2.$$

Autómatas finitos determinísticos

Definición

Un autómata finito determinístico M consta de un alfabeto Σ, un conjunto de estados S, una función de transición $T : S \times \Sigma \rightarrow S$, un estado inicial $s_0 \in S$ y un conjunto de estados de aceptación $A \subseteq S$.

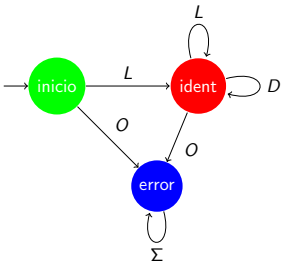
Lenguaje aceptado

El lenguaje $L(M)$ aceptado por M es el conjunto de cadenas de caracteres $c = c_1 c_2 \cdots c_n \in \Sigma^*$ tales que $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, $ldots$, $s_n = T(s_{n-1}, c_n)$ con $s_n \in A$.

Diferencias entre la definición y nuestro uso

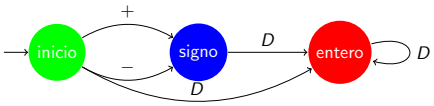
- Una primera diferencia es que no etiquetamos las transiciones con caracteres del alfabeto sino con subconjuntos del mismo.
- Esto se debe a que sería abrumador dibujar una enorme cantidad de transiciones idénticas para cada elemento de este conjunto.
- Una segunda diferencia es que no dibujamos transiciones para todos los caracteres desde cada estado.
- Estas transiciones faltantes significan que no tenemos un token como el esperado o que el reconocimiento del token ha concluido.

Ejemplo de un autómata finito con todas las transiciones



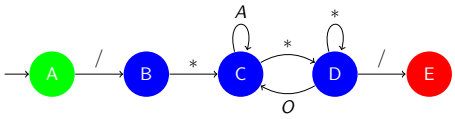
- L son la letras.
- D son los dígitos.
- $O = \neg(L|D)$ son los otros caracteres.

Un autómata para reconocer enteros con signo



- D son los dígitos.
- No dibujamos las transiciones o estado de error.

Un autómata para reconocer comentarios en C

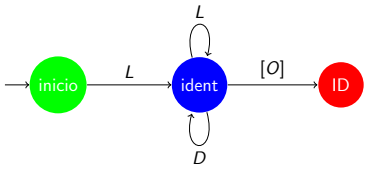


- A representa a cualquier carácter excepto $*$.
- O representa a cualquier carácter excepto $*$ y $/$.
- No dibujamos las transiciones o estado de error.

Acciones en un autómata finito

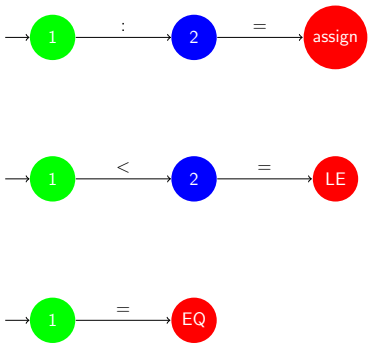
- La definición matemática de un autómata no indica todos los aspectos de su funcionamiento.
- Por ejemplo, no indica qué hacer cuando se alcance un estado de error o de aceptación.
- Cuando se hace una transición generalmente se agrega el carácter de entrada a una cadena que eventualmente será el lexema de un token.
- Cuando se alcanza un estado de aceptación se devuelve el token encontrado y sus atributos.
- Cuando se alcanza un estado de error se devuelve un carácter a la entrada o se genera un token de error.

Ejemplo de un autómata finito con acciones

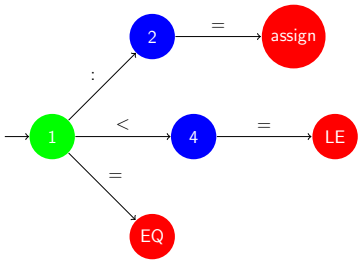


- El estado inicial tiene otras transiciones en $\neg L$ no mostradas.
- En el estado final se devuelve el token ID.
- En la transición $[O]$ se devuelve el carácter leído a la entrada.

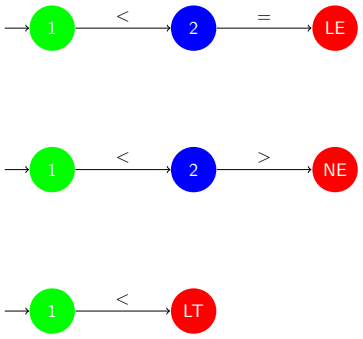
Dos o más autómatas con caracteres iniciales distintos



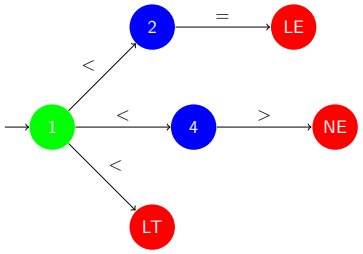
Combinados en un solo autómata



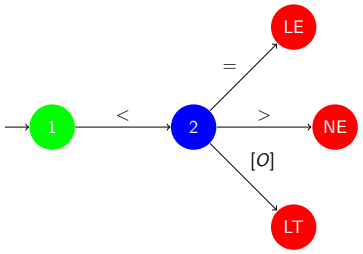
Dos o más autómatas con caracteres iniciales iguales



Combinados equivocadamente en un solo autómata



Combinados correctamente en un solo autómata

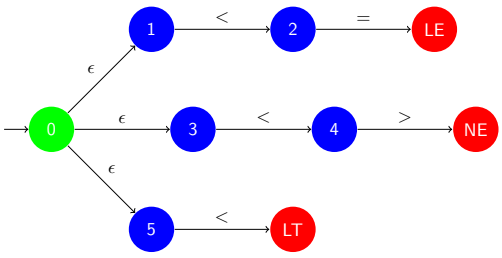


Contenido

Autómatas no determinísticos

- En principio se pueden combinar los autómatas correspondientes a todos los tipos de token en uno gigante.
- Sin embargo esto se vuelve una tarea muy complicada.
- Una posibilidad es ampliar la definición de autómata para permitir cero o más transiciones desde un estado con un mismo símbolo.
- A estos autómatas los llamaremos no deterministicos.
- También permitiremos transiciones con la cadena vacía ϵ.
- Estos autómatas no representan algoritmos.

Autómatas combinados en uno no determinístico



Autómatas finitos no deterministicos

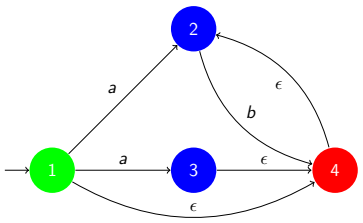
Definición

Un autómata finito no determinístico M consta de un alfabeto Σ, un conjunto de estados S, una función de transición $T : S \times (\Sigma \cup \epsilon) \to 2^S$, un estado inicial $s_0 \in S$ y un conjunto de estados de aceptación $A \subseteq S$.

Lenguaje aceptado

El lenguaje $L(M)$ aceptado por M es el conjunto de cadenas de caracteres $c = c_1 c_2 \cdots c_n \in (\Sigma \cup \epsilon)^*$ tales que $s_1 \in T(s_0, c_1)$, $s_2 \in T(s_1, c_2)$, *ldots*, $s_n \in T(s_{n-1}, c_n)$ con $s_n \in A$.

Ejemplo de un autómata finito no determinístico

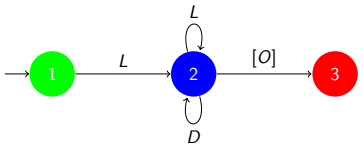


- La cadena *abb* puede aceptarse por cualquiera de las transiciones:
- $\to 1 \to^a 2 \to^b 4 \to^\epsilon 2 \to^b 4$ o
- $\to 1 \to^a 3 \to^\epsilon 4 \to^\epsilon 2 \to^b 4 \to^\epsilon 2 \to^b 4$.

Implementación de autómatas finitos en código

- Existen diversas maneras de implementar un autómata finito en un programa.
- No todas son apropiadas en general, pero algunas son sencillas.
- Por el momento analizaremos tres opciones básicas:
  1. Condicionales anidados.
  2. Casos anidados.
  3. Tabla de transición.

Código con condicionales anidados



if siguiente carácter es una letra **then**
  avanzar en la entrada
  while siguiente carácter es una letra o un dígito **do**
    avanzar en la entrada
  aceptar
else
  error u otros casos

Código con casos anidados

$s \leftarrow 1$
while $s = 1$ **or** $s = 2$ **do**
  case s **of**
  1: **case** c **of**
  letra: avanzar en la entrada y $s \leftarrow 2$
  else $s \leftarrow 4$ (error)
  2: **case** c **of**
  letra o dígito: avanzar en la entrada y $s \leftarrow 2$
  else $s \leftarrow 3$
if $s = 3$ **then**
  aceptar
else
  error

Tabla de transición

- En los ejemplos anteriores el autómata se alambró en el código.
- Otra opción es la de expresar el autómata en una estructura de datos.
- Esto nos permitirá escribir un código genérico.
- Esta estructura puede ser una tabla de transición T en la que $T(s, c)$ representa el estado que se alcanza al estar en s y leer el carácter c de la entrada. Como ejemplo:

s,c	L	D	O	acepta	error
1	2			no	no
2	2	2	[3]	no	no
3				sí	no

- Llamaremos A a la columna de aceptación, E a la columna de error y $L[s, c]$ a la decisión de si se debe leer un caracter de la entrada o no.

Código con tabla de transición

$s \leftarrow 1$
c se lee de la entrada
while not $A[s]$ **and not** $E[s]$ **do**
  $n \leftarrow T[s, c]$
  if $L[s, c]$ **then**
    c se lee de la entrada
  $s \leftarrow n$
if $A[s]$ **then**
  aceptar
else
  error

Contenido

Tokens del lenguaje de muestra

Palabras reservadas	Símbolos	Otros
if	+ MAS	número
then	- MEN	ident
else	* POR	
end	/ DIV	
repeat	= EQU	
until	< MEQ	
read	(PIZ	
write	) PDE	
	; PYC	
	:= ASI	

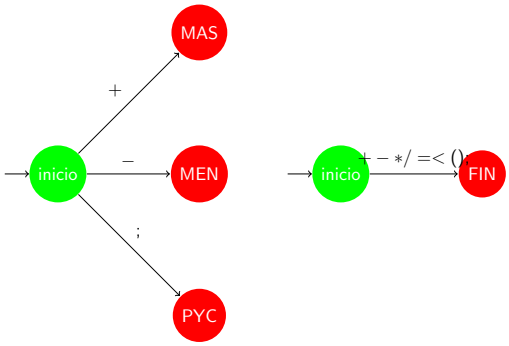
Convenciones léxicas adicionales

- Los comentarios están encerrados entre llaves {}.
- Los comentarios no pueden estar anidados.
- El código es de formato libre.
- El espacio en blanco consta de espacios, tabuladores y nuevas líneas.
- Se sigue el principio de la subcadena más larga.

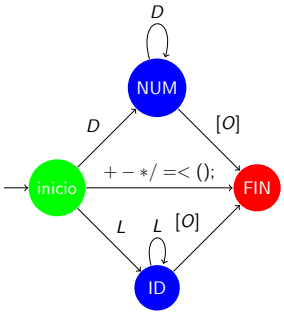
Diseño del analizador léxico

- Ya tenemos autómatas para los números y los identificadores.
- El autómata para los comentarios es particularmente simple.
- Los autómatas para los demás tokens son sencillos pues constan de cadenas fijas.

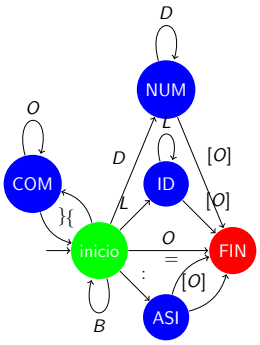
Autómata para los símbolos de un caracter



Autómata para los símbolos, números e identificadores



Autómata para el analizador léxico



Palabras reservadas

- ¿Qué pasa con las palabras reservadas?
- El analizador léxico las reconoce como identificadores y después hace una búsqueda en una tabla.
- Esta búsqueda debe ser tan eficiente como sea posible:
  - La búsqueda lineal resulta muy lenta.
  - La búsqueda binaria es razonable si no hay demasiadas palabras reservadas.
  - Una tabla de dispersión resulta casi siempre adecuada.

Contenido

Contenido

Análisis sintáctico

Secuencia de tokens ⟶ Analizador sintáctico ⟶ Árbol sintáctico

- La tarea del analizador sintáctico es determinar la estructura sintáctica de un programa a partir de los tokens producidos por el analizador léxico.
- Además debe construir de manera implícita o explícita un árbol de análisis gramatical o árbol sintáctico.
- Se puede ver al analizador sintáctico como una función que toma como entrada la secuencia de tokens producida por el analizador léxico y que produce como salida el árbol sintáctico correspondiente.

Analizador sintáctico

- Por lo regular la secuencia de tokens no es un parámetro de entrada explícito para el analizador sintáctico.
- En realidad el analizador sintáctico llama a una función del analizador léxico como `getToken` para obtener el siguiente token de la entrada cuando lo necesite.
- La etapa de análisis sintáctico del compilador se reduce a una llamada al analizador léxico como `syntaxTree = parse()`.
- En un compilador de una sola pasada no es necesario construir el árbol sintáctico explícito, por lo que una llamada a `parse()` hará todo el trabajo.
- En un compilador de varias pasadas la primera pasada usará la secuencia de tokens como entrada y las siguientes pasadas usarán el árbol sintáctico como entrada.

Árbol sintáctico

- La estructura del árbol sintáctico dependerá fuertemente de la estructura sintáctica tanto del lenguaje como del programa que está siendo compilado.
- Generalmente el árbol sintáctico es una estructura dinámica en la que cada nodo contiene en sus campos toda la información necesaria para continuar con el proceso de compilación.
- A veces los nodos son estructuras variables para ahorrar espacio.
- Los campos de atributo también pueden ser estructuras dinámicas.

Tratamiento de errores

- En el analizador léxico si aparece un caracter inesperado simplemente se genera un token de error.
- De esa manera se le deja la labor al analizador sintáctico.
- El analizador sintáctico deberá generar un mensaje de error adecuado.
- Además debe recuperarse del error para continuar con el proceso y encontrar tantos errores como sea posible.
- A veces también lleva a cabo una reparación de error cuando éste es tan simple que se puede inferir el código correcto.
- Ninguna de estas labores es fácil pues el error a veces se detecta mucho después de que ocurrió.

Contenido

Gramáticas libres de contexto

- Una gramática libre de contexto es un tipo más sofisticado de especificación sintáctica de un lenguaje.
- Son similares a las expresiones regulares pero permiten la recursión.
- Un ejemplo de lenguaje que se puede expresar con una gramática libre de contexto es el que representa a las expresiones simples aritméticas con operaciones de suma, resta y multiplicación.
- Estas expresiones se pueden dar mediante la siguiente gramática:
  1. $E \rightarrow EOE|(E)|N$.
  2. $O \rightarrow +|-|*$.
- Donde N representa a un número entero.

Notación para gramáticas libres de contexto

- En el ejemplo anterior se usó el símbolo $\rightarrow$ para indicar que un símbolo genera cualquiera de las cadenas a su derecha separadas por el símbolo $|$.
- Como con las expresiones regulares se usa la concatenación, pero a diferencia de éstas no se usa la cerradura de Kleene para la repetición.
- Esto se debe a que la repetición se puede expresar recursivamente.
- El alfabeto de una gramáticas libres de contexto suele constar de tokens (definidos a su vez por expresiones regulares) y caracteres simples.
- A esto se le llama la forma Backus-Naur (BNF).

Especificación de una gramática libre de contexto

- Una regla gramatical libre de contexto en BNF es una cadena de símbolos donde:
  - El primer símbolo es el nombre de una estructura.
  - El segundo símbolo es $\rightarrow$.
  - Los siguientes símbolos son caracteres del alfabeto, nombres de estructuras o $|$.
- Informalmente, una regla gramatical define una estructura. En nuestro ejemplo anterior:
  - $E \rightarrow EOE|(E)|N$ define una expresión E.
  - $O \rightarrow +|-|*$ define un operador O.

Especificación alternativa de una gramática

- Sería perfectamente válido especificar nuestro ejemplo como:
  1. $E \rightarrow EOE$.
  2. $E \rightarrow (E)$.
  3. $E \rightarrow N$.
  4. $O \rightarrow +$.
  5. $O \rightarrow -$.
  6. $O \rightarrow *$.
- Sin embargo, normalmente agruparemos todas las opciones que definan a una estructura en una sola regla.

Cadenas de tokens legales

- Las reglas gramaticales libres de contexto determinan un conjunto de cadenas legales de tokens para las estructuras definidas.
- Por ejemplo, la expresión aritmética

$$(34 - 3) * 42$$

corresponde a la cadena legal de siete tokens

$$(N - N) * N.$$

- Aquí el token N (número) tiene su estructura determinada por el analizador léxico.
- Como segundo ejemplo, la cadena

$$(34 - 3 * 42$$

no es una cadena legal pues no tiene paréntesis derecho y la regla $E \rightarrow (E)$ indica que los paréntesis deben venir en pares.

Derivaciones

- Una derivación es una secuencia de reemplazos de nombres de estructuras por alguna de las opciones en los lados derechos de las reglas gramaticales que las definen.
- Una derivación comienza con un nombre de estructura simple y termina con una cadena de tokens o caracteres.
- Las cadenas que se pueden obtener como resultado de estas derivaciones son las cadenas sintácticamente legales definidas por las reglas gramaticales.

Ejemplo de una derivación

- Una derivación para la cadena $(N - N) * N$:

$$\begin{aligned} E &\Rightarrow EOE && (1)\\ &\Rightarrow EON && (2)\\ &\Rightarrow E * N && (3)\\ &\Rightarrow (E) * N && (4)\\ &\Rightarrow (EOE) * N && (5)\\ &\Rightarrow (EON) * N && (6)\\ &\Rightarrow (E - N) * N && (7)\\ &\Rightarrow (N - N) * N. && (8) \end{aligned}$$

- Observe que usamos el símbolo $\Rightarrow$ para cada paso de una derivación.
- También observe que pudimos obtener la misma cadena con otra secuencia de pasos.

Lenguaje definido por una gramática

- El conjunto de todas las cadenas obtenidas por derivaciones desde un símbolo de estructura es el lenguaje definido por la gramática.
- Esto se puede escribir de manera simbólica como

$$L(G) = \{s : E \Rightarrow^* s\}$$

donde G es la gramática, s es una cadena arbitraria de tokens o caracteres, E es la estructura inicial y $\Rightarrow^*$ significa una secuencia de pasos de derivación.

- Cada estructura define un lenguaje, pero en general sólo nos interesa el lenguaje definido por la estructura más general de la gramática.
- A las estructuras se les llama también no terminales, a los tokens o caracteres se les llama también terminales y a las reglas se les llama también producciones.

Ejemplo de gramática para paréntesis anidados

- Considere la gramática G con única regla gramatical:

$$E \rightarrow (E)|a.$$

- Esta gramática tiene un no terminal E y tres terminales $(,), a$.
- El lenguaje generado por esta gramática es

$$L(G) = \{(^n a)^n : n \geq 0\}$$

es decir el lenguaje de cierta cantidad de paréntesis izquierdos, seguidos de una a y seguida de la misma cantidad de paréntesis derechos.

Ejemplo de una gramática para sumas

- Considere la gramática G con única regla gramatical:

$$E \rightarrow E + a|a.$$

- Esta gramática tiene un no terminal E y dos terminales $+, a$.
- El lenguaje generado por esta gramática es

$$L(G) = \{a, a + a, a + a + a, \ldots\}$$

es decir el lenguaje de todas las cadenas de a separadas por signos $+$.

Ejemplo de una gramática para decisiones anidadas

- Considere la gramática G con tres reglas gramaticales:

$$\begin{aligned} S &\rightarrow I \mid \text{c} \\ I &\rightarrow \text{if } (D)S \mid \text{if } (D)S \text{ else } S \\ D &\rightarrow 0 \mid 1. \end{aligned}$$

- Esta gramática tiene tres no terminales S, I, D y siete terminales $0, 1, (,),$ c, if, else.
- El lenguaje generado por esta gramática es una simplificación de las sentencias de decisiones anidadas (con decisiones simplificadas a 0 o 1 y las demás sentencias agrupadas en c).

Recursión por la izquierda y por la derecha

- Antes dijimos que las gramáticas libres de contexto no usan a la cerradura de Kleene para expresar la repetición.
- Esto resulta innecesario puesto que tanto la regla

$$A \rightarrow Aa \mid a$$

como la regla

$$A \rightarrow aA \mid a$$

generan al mismo lenguaje que la expresión regular a^+.
- A la primera de esas reglas se le llama recursiva por la izquierda y a la segunda recursiva por la derecha por la ubicación del no terminal.
- Si queremos generar el lenguaje a^* entonces necesitamos una producción vacía como $A \rightarrow \epsilon$ para obtener $A \rightarrow Aa|\epsilon$ o $A \rightarrow aA|\epsilon$.

Ejemplo de gramática para paréntesis balanceados

- Considere la gramática G con única regla gramatical:

$$E \rightarrow (E)E \mid \epsilon.$$

- Esta gramática tiene un no terminal E y dos terminales $(,)$.
- El lenguaje generado por esta gramática es el lenguaje de los paréntesis balanceados.
- Ejemplo de una derivación:

$$\begin{aligned} E &\Rightarrow (E)E \Rightarrow ((E)E)E \\ &\Rightarrow ((E)E)(E)E \Rightarrow (()E)(E)E \\ &\Rightarrow (()E)()E \Rightarrow (())()E \\ &\Rightarrow (())(). \end{aligned}$$

Ejemplo de otra gramática para decisiones anidadas

- Considere la gramática G con cuatro reglas gramaticales:

$$\begin{aligned} S &\rightarrow I \mid \text{c} \\ I &\rightarrow \text{if } (D)SE \\ E &\rightarrow \text{else } S \mid \epsilon \\ D &\rightarrow 0 \mid 1. \end{aligned}$$

- Esta gramática tiene cuatro no terminales S, I, D, E y siete terminales $0, 1, (,),$ c, if, else.
- Observe cómo el ϵ indica que la parte E es opcional.

Ejemplos de gramáticas para secuencia de sentencias

- La gramática con dos reglas gramaticales:

$$\begin{aligned} P &\rightarrow S;P \mid S \\ S &\rightarrow s \end{aligned}$$

define el lenguaje de secuencias de sentencias

$$\{s, s; s, s; s; s, \ldots\}$$

- La gramática con dos reglas gramaticales:

$$\begin{aligned} P &\rightarrow S;P \mid \epsilon \\ S &\rightarrow s \end{aligned}$$

define el lenguaje de secuencias de sentencias

$$\{\epsilon, s; , s; s; , s; s; s; , \ldots\}$$

- En el primero el ; es un separador y en el segundo un terminador.

Contenido

Múltiples derivaciones

- Otra derivación para la cadena $(N - N) * N$:

$$\begin{aligned} E &\Rightarrow EOE && (9) \\ &\Rightarrow (E)OE && (10) \\ &\Rightarrow (EOE)OE && (11) \\ &\Rightarrow (NOE)OE && (12) \\ &\Rightarrow (N - E)OE && (13) \\ &\Rightarrow (N - N)OE && (14) \\ &\Rightarrow (N - N) * E && (15) \\ &\Rightarrow (N - N) * N. && (16) \end{aligned}$$

- La diferencia entre esta derivación y la que vimos antes es el orden de los reemplazos
- Esto es en realidad una diferencia superficial.

Árboles de análisis gramatical

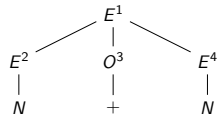
- Necesitamos una representación que mantenga la estructura de una cadena de terminales, abstrayendo las características esenciales de una derivación al mismo tiempo que se eliminan las diferencias superficiales.
- Un árbol de análisis gramatical correspondiente a una derivación es un árbol en el cual los nodos interiores están etiquetados por no terminales, las hojas están etiquetadas por terminales y los hijos de cada nodo interno representan el reemplazo del no terminal asociado en un paso de la derivación.

Ejemplo de un árbol de análisis gramatical en preorden

- La derivación

$$E \Rightarrow^1 EOE \Rightarrow^2 NOE \Rightarrow^3 N + E \Rightarrow^4 N + N$$

corresponde al árbol de análisis gramatical:



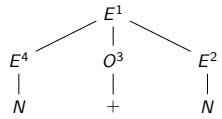
- Observe que la numeración de los nodos internos fue en preorden.

Ejemplo de un árbol de análisis gramatical en posorden

- La derivación

$$E \Rightarrow^1 EOE \Rightarrow^2 EON \Rightarrow^3 E + N \Rightarrow^4 N + N$$

corresponde al árbol de análisis gramatical:

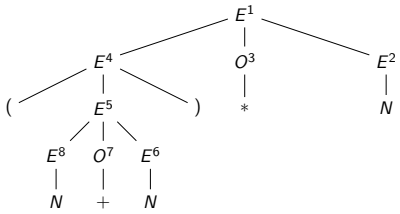


- Observe que la numeración de los nodos internos fue en posorden inverso.

Derivaciones por la izquierda y por la derecha

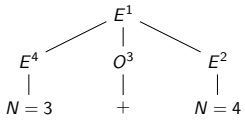
- Un árbol de análisis gramatical corresponde en general a muchas derivaciones que representan la misma estructura básica.
- Sin embargo podemos distinguir dos derivaciones particulares:
- Una derivación por la izquierda es aquella en la que se reemplaza el no terminal más a la izquierda en cada paso. Corresponde a la numeración en preorden de los nodos internos.
- Una derivación por la derecha es aquella en la que se reemplaza el no terminal más a la derecha en cada paso. Corresponde a la numeración en posorden inverso de los nodos internos.

Un ejemplo más complejo

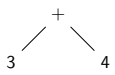


Exceso de información

- Un árbol de análisis gramatical es útil pero contiene información no necesaria.
- Por ejemplo, el árbol de análisis gramatical



se puede simplificar al árbol

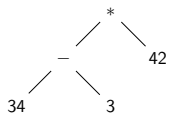


que contiene estrictamente la información necesaria.

- En este último árbol se muestran los valores verdaderos de los tokens.

Otro ejemplo de un árbol simplificado

- En el siguiente árbol simplificado algunos tokens desaparecieron:



- Por ejemplo los paréntesis, pero el significado se mantiene.

Árboles sintácticos abstractos

- Estos árboles simplificados representan abstracciones de las secuencias de tokens.
- Las secuencias de tokens originales no se pueden recuperar de estos árboles.
- Pero contienen toda la información necesaria para llevar a cabo la traducción.
- Estos árboles se conocen como árboles sintácticos abstractos o simplemente como árboles sintácticos.

Ejemplo de estructura para un árbol sintáctico abstracto

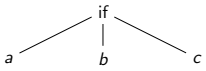
- Los árboles sintácticos abstractos para expresiones aritméticas simples se pueden representar con las siguientes estructuras en C:

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpKind,ConstKind} ExpKind;
typedef struct streenode {
   ExpKind kind;
   OpKind op;
   struct streenode *lchild, *rchild;
   int val;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

- Observe que se usaron tipos enumerados para definir los tipos de nodos del árbol.

Ejemplo de árbol sintáctico abstracto para decisiones

- Un árbol sintáctico abstacto para la cadena `if (a) b else c` sería:



- En este ejemplo usamos el token `if` como raíz del árbol y a, b, c corresponden con los árboles sintácticos abstractos correspondientes a la condición, el caso verdadero y el caso falso, respectivamente.

Ejemplo de estructura para decisiones

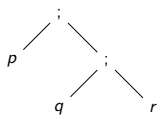
- Los árboles sintácticos abstractos para decisiones se pueden representar con las siguientes estructuras en C:

```
typedef enum {ExpK,StmtK} NodeKind;
typedef enum {Zero,One} ExpKind;
typedef enum {IfK,OtherK} StmtKind;
typedef struct streenode {
   NodeKind kind;
   ExpKind ekind;
   StmtKind skind;
   struct streenode *test, *thenpart, *elsepart;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

- Observe que se usaron tipos enumerados para definir los tipos de nodos del árbol.

Ejemplo de árbol sintáctico abstracto para secuencias de sentencias

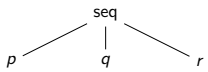
- Un árbol sintáctico abstracto para la cadena `p;q;r` sería:



- En este ejemplo usamos el token `;` como raíz del árbol y p, q, r corresponden con los árboles sintácticos abstractos correspondientes a las sentencias `p,q,r`, respectivamente.

Ejemplo alternativo de árbol sintáctico abstracto para secuencias de sentencias

- Otro árbol sintáctico abstacto para la cadena `p;q;r` podría ser:



- En este ejemplo usamos el token `seq` como raíz del árbol y p, q, r corresponden con los árboles sintácticos abstractos correspondientes a las sentencias `p,q,r`, respectivamente.
- Un problema de esta representación es que el número de hijos de `seq` es variable.
- Esto se puede resolver reemplazando esta estructura por una lista ligada.

Contenido

Gramáticas ambiguas

- Los árboles de análisis gramatical y los árboles sintácticos expresan la estructura de la sintaxis de un programa.
- Hemos visto cómo a partir de un árbol de análisis gramatical se puede obtener una derivación por la izquierda y una derivación por la derecha.
- Sin embargo, una gramática puede permitir que una cadena tenga más de un árbol de análisis gramatical.
- Considere la cadena $N - N * N$ generada por la gramática

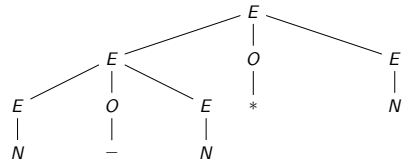
$$E \to EOE|(E)|N \text{ y } O \to +|-|*$$

que tiene dos árboles de análisis gramatical distintos.

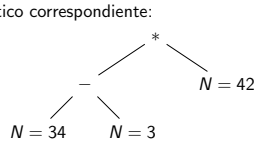
- Por lo tanto, esta gramática es ambigua.

Primer árbol de análisis gramatical

- Primer árbol de análisis gramatical:



- Y su árbol sintáctico correspondiente:



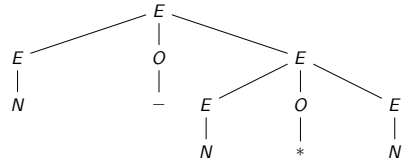
Primera derivación izquierda

- Una derivación izquierda para la cadena $N - N * N$:

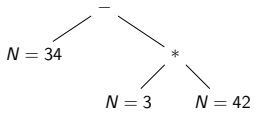
$$\begin{aligned} E &\Rightarrow EOE && (17)\\ &\Rightarrow EOEOE && (18)\\ &\Rightarrow NOEOE && (19)\\ &\Rightarrow N - EOE && (20)\\ &\Rightarrow N - NOE && (21)\\ &\Rightarrow N - N * E && (22)\\ &\Rightarrow N - N * N. && (23) \end{aligned}$$

Segundo árbol de análisis gramatical

- Segundo árbol de análisis gramatical:



- Y su árbol sintáctico correspondiente:



Segunda derivación izquierda

- Otra derivación izquierda para la cadena $N - N * N$:

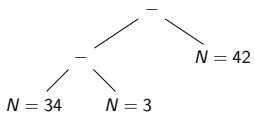
$$\begin{aligned} E &\Rightarrow EOE && (24)\\ &\Rightarrow NOE && (25)\\ &\Rightarrow N - E && (26)\\ &\Rightarrow N - EOE && (27)\\ &\Rightarrow N - NOE && (28)\\ &\Rightarrow N - N * E && (29)\\ &\Rightarrow N - N * N. && (30) \end{aligned}$$

Problemas de las gramáticas ambiguas

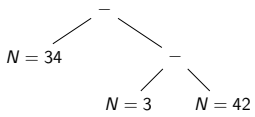
- Una gramática ambigua representa un problema para un compilador porque no indica con precisión la estructura sintáctica de un programa.
- Por lo tanto debe considerarse como una especificación incompleta de un lenguaje y debe evitarse.
- Para tratar de resolver las ambigüedades se usan dos métodos básicos:
  - Establecer una regla que determine cuál de los árboles sintácticos es el correcto.
  - Modificar la gramática de modo que no exista la ambigüedad.
- En cualquier caso, debemos decidir cuál es el árbol sintáctico correcto, es decir, cuál es el que refleja correctamente el significado del programa.

Otro ejemplo de ambigüedad

- La cadena $N - N - N$ nos presenta otro ejemplo de ambigüedad.
- Primer árbol sintáctico, que representa a $(34 - 3) - 42 = -11$:

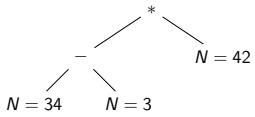


- Segundo árbol sintáctico, que representa a $34 - (3 - 42) = 73$:

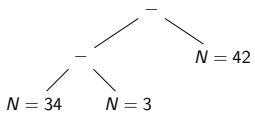


Precedencia y asociatividad

- El primer ejemplo de ambigüedad ($N - N * N$) se resuelve cuando consideramos que la multiplicación tiene precedencia sobre la resta:



- El segundo ejemplo de ambigüedad ($N - N - N$) se resuelve si consideramos que la resta es una operación asociativa por la izquierda:



Cascada de precedencia

- Para manejar la precedencia de las operaciones en la gramática debemos agrupar a los operadores en grupos de igual precedencia y para cada uno de ellos debemos escribir una regla diferente:
  - $E \to ESE|T$
  - $S \to +|-$
  - $T \to TMT|F$
  - $M \to *$
  - $F \to (E)|N$
- En esta gramática las multiplicaciones se agrupan en la regla T, mientras que las sumas y restas se agrupan en la regla E.
- Como el caso base de E es T esto significa que las sumas y restas aparecerán más arriba en un árbol de sintaxis y por lo tanto tendrán una precedencia más baja.
- A estas agrupaciones se les llama cascadas de precedencia.

Ambigüedad, recursión y asociatividad

- La gramática que acabamos de presentar sigue siendo ambigua y todavía no representa la asociatividad de los operadores.
- La razón es la recursión en ambos lados de las reglas $E \to ESE$ y $T \to TMT$.
- La solución es reemplazar una de las recursiones con el caso base, forzando las repeticiones del lado en el que aparece la recursión.
- Por ejemplo, la regla

$$E \to EST|T$$

hace que la suma y la resta sean asociativas por la izquierda, mientras que la regla

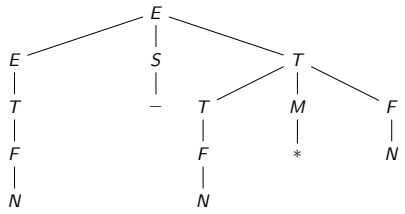
$$E \to TSE|T$$

hace que la suma y la resta sean asociativas por la derecha.

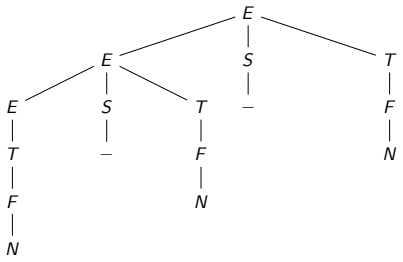
Solución con operadores asociativos por la izquierda

- Si aplicamos estos cambios obtenemos la nueva gramática:
  - $E \to EST|T$
  - $S \to +|-$
  - $T \to TMF|F$
  - $M \to *$
  - $F \to (E)|N$
- Ahora todos los operadores son asociativos por la izquierda.

Árbol de análisis gramatical para $N - N * N$



Árbol de análisis gramatical para $N - N - N$



El problema del else ambiguo

- Considere la gramática simplificada para decisiones anidadas:

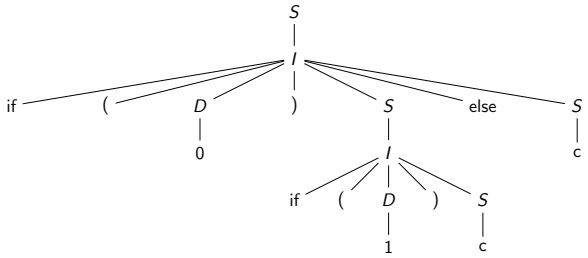
$$\begin{aligned} S &\rightarrow I \mid \text{c} \\ I &\rightarrow \text{if } (D)S \mid \text{if } (D)S \text{ else } S \\ D &\rightarrow 0 \mid 1. \end{aligned}$$

- Esta gramática es ambigua debido al else opcional.
- Considere por ejemplo la cadena

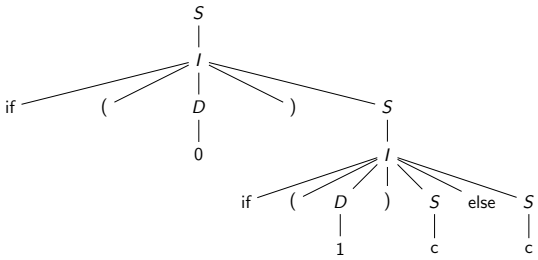
  if (0) if (1) c else c

  que tiene dos árboles de análisis gramatical.

Primer árbol de análisis gramatical



Segundo árbol de análisis gramatical



¿Cuál es el árbol correcto?

- La respuesta depende de si queremos asociar el else con el primero o con el segundo if.
- El siguiente fragmento de código en C nos ayudará a decidir:

```
if (x != 0.0)
   if (y == 1.0/x) ok = 1;
   else z = 1.0/x;
```

- Es decir: esperamos que el else se asocie con el último if no asociado.
- A esto se le llama la regla de la anidación más cercana.

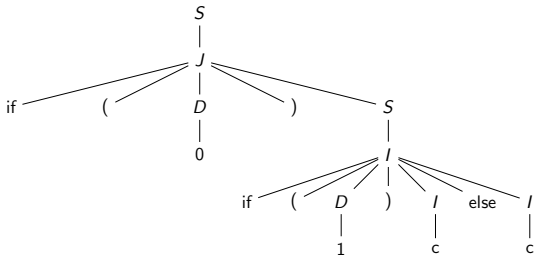
Solución al problema

- Una solución para el problema es como sigue:

$$\begin{aligned} S &\rightarrow I \mid J \\ I &\rightarrow \text{if } (D)I \text{ else } I \mid c \\ J &\rightarrow \text{if } (D)S \mid \text{if } (D)I \text{ else } J \\ D &\rightarrow 0 \mid 1. \end{aligned}$$

- Esto funciona al permitir que llegue solamente una sentencia igualada J antes que un else en una sentencia if.

Nuevo árbol de análisis gramatical



Ambigüedad no esencial

- En ocasiones una gramática puede ser ambigua y aún así producir árboles sintácticos únicos.
- Un ejemplo de esto es la gramática para secuencias de sentencias

$$\begin{aligned} P &\rightarrow S; P|S \\ S &\rightarrow s \end{aligned}$$

  o incluso una gramática donde la primera regla fuera $P \rightarrow P; P|S$.
- A esto lo llamaremos una ambigüedad no esencial puesto que la semántica asociada no depende de cuál regla de eliminación de ambigüedad se use.
- Otro ejemplo aparece con los operadores asociativos como la suma. En este caso $(a+b)+c = a+(b+c)$ y por lo tanto no importa cuál de los dos árboles sintácticos se use para representar $a+b+c$.

Contenido

Notación EBNF

- Las construcciones repetitivas y opcionales son muy comunes en las estructuras de los lenguajes de programación.
- Es por eso que en ocasiones se extiende la notación BNF con notaciones especiales para estos casos.
- Una de estas notaciones es la BNF extendida o EBNF.
- La repetición se simboliza con llaves $\{\}$, como en $A \rightarrow \{\alpha\}\beta$ en lugar de la regla recursiva por la izquierda $A \rightarrow A\alpha|\beta$.
- La opción se simboliza con corchetes [], como en $A \rightarrow [\alpha]\beta$ en lugar de $A \rightarrow \alpha\beta|\beta$.

Ejemplos de repetición

- En el caso de secuencias de sentencias tenemos que la regla

$$P \rightarrow S; P|S$$

  se escribiría en EBNF como

$$P \rightarrow S\{; S\}$$

  lo que resulta en recursión por la izquierda.
- En el caso de las expresiones aritméticas tenemos que la regla

$$E \rightarrow EST|T$$

  se escribiría en EBNF como

$$E \rightarrow \{TS\}T$$

  lo que implica asociatividad por la izquierda.

Ejemplos de opción

- En el caso de secuencias de sentencias tenemos que la regla

$$P \rightarrow S; P|S$$

  se escribiría en EBNF como

$$P \rightarrow S[; P]$$

  lo que resulta en recursión por la derecha.
- En el caso de las expresiones aritméticas tenemos que la regla

$$E \rightarrow EST|T$$

  se escribiría en EBNF como

$$E \rightarrow T[SE]$$

  lo que implica asociatividad por la derecha.

Contenido

Una gramática libre de contexto para Tiny

Programa	$P \rightarrow P; S\|S$
Sentencia	$S \rightarrow I\|R\|A\|L\|W$
If	$I \rightarrow$ if E then P end|if E then P else P end
Repeat	$R \rightarrow$ repeat P until E
Asignación	$A \rightarrow D := E$
Lectura	$L \rightarrow$ read D
Escritura	$W \rightarrow$ write E
Expresión	$E \rightarrow XCX\|X$
Comparación	$C \rightarrow < \| =$
Exp. Simple	$X \rightarrow XUT\|T$
Suma	$U \rightarrow +\|-$
Término	$T \rightarrow TMF\|F$
Multiplicación	$M \rightarrow *\|/$
Factor	$F \rightarrow (E)\|N\|D$ (Número e iDentificador).

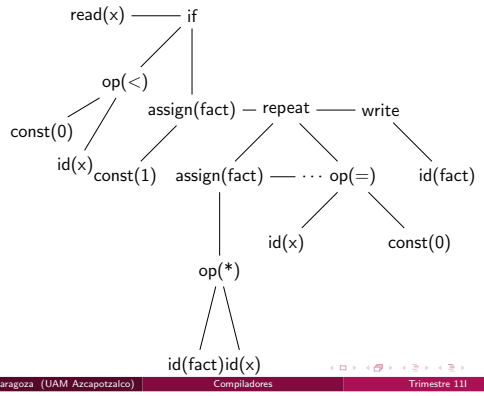
Declaraciones en C para un nodo de árbol sintáctico

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;
typedef enum {Void, Integer, Boolean} ExpType;

#define MAXCHLIDREN 3

typedef struct treeNode {
   struct treeNode *child[MAXCHILDREN];
   struct treeNode *sibling;
   int lineno;
   NodeKind nodekind;
   union {StmtKind stmt; ExpKind exp;} kind;
   union {TokenType op; int val; char *name;} attr;
   ExpType type;
} TreeNode;
```

Ejemplo de un programa en Tiny

```
{ Programa de muestra en lenguaje Tiny -
  Calcula el factorial }

read x; { entrada de un entero }
if 0 < x then { no calcula si x <= 0 }
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1;
   until x = 0;
   write fact { salida factorial de x }
end
```


Ejemplo de un árbol sintáctico para un programa



Contenido

Contenido

Análisis sintáctico descendente

- Un algoritmo de análisis sintáctico descendente analiza una cadena de tokens de entrada mediante una búsqueda de los pasos de una derivación por la izquierda.
- Estos algoritmos se llaman así porque el recorrido implicado del árbol de análisis gramatical es en preorden, por lo que va de la raíz a las hojas.
- Existen dos tipos de analizadores sintácticos descendentes:
  - Analizadores sintácticos inversos.
  - Analizadores sintácticos predictivos.
- Estos últimos (los únicos que estudiaremos) intentan predecir la siguiente construcción sintáctica usando uno o más tokens por adelantado.

Analizadores sintácticos predictivos

- Las dos clases de analizadores sintácticos predictivos que estudiaremos son:
  - Analizadores sintácticos descendentes recursivos: Son muy versátiles y adecuados para diseñarse e implementarse a mano.
  - Analizadores sintácticos LL(1): No se usan a menudo en la práctica, pero son útiles para analizar algunos de los problemas que aparecen en el análisis descendente recursivo.
- Por el momento basta saber que LL(1) significa:
  - La primera L que la entrada se lee de izquierda a derecha.
  - La segunda L que la derivación buscada es por la izquierda.
  - El (1) que sólo se usa un símbolo de la entrada para decidir.

Contenido

El método básico descendente recursivo

- La idea es considerar la regla gramatical para un no terminal A como una definición de un procedimiento (posiblemente recursivo) que reconocerá una A.
- El lado derecho de la regla gramatical especifica la estructura del código para este procedimiento:
- La secuencia de terminales y no terminales en una selección corresponde a concordancias de la entrada y llamadas a otros procedimientos, respectivamente.
- Las selecciones corresponden a las alternativas (`switch` o `if`) dentro del código.

Ejemplo con la gramática de expresión

- Considere la gramática de expresión vista anteriormente:
  - $E \rightarrow EST|T$
  - $S \rightarrow +|-$
  - $T \rightarrow TMF|F$
  - $M \rightarrow *$
  - $F \rightarrow (E)|N$
- Entonces un procedimiento que reconoce un factor F sería:

if token es un (**then**
  empata un (
  reconoce una expresión E
  empata un)
else if token es un número **then**
  empata un número
else
  señala un error.

Aclaraciones

- En el algoritmo anterior suponemos que existe una variable que mantiene el token actual de la entrada.
- También suponemos que existe un procedimiento que empata el token actual con su parámetro, avanza en la entrada si tiene éxito y señala un error en caso contrario:

if el token es el esperado **then**
  lee un nuevo token
else
  señala un error.

- Por el momento suponemos que el señalar un error imprime un mensaje y termina el proceso de compilación.
- Observe también que reconocer un factor es un procedimiento recursivo. Esto se debe a que debe reconocer una expresión, el procedimiento para reconocer una expresión debe reconocer un término y el procedimiento para reconocer un término debe reconocer un factor.

Uso de EBNF para selección

- Considere la gramática simplificada para una sentencia de decisión

$$D \rightarrow \text{if } (E)S| \text{ if } (E)S \text{ else } S.$$

- Esto se puede reescribir en EBNF como:

$$D \rightarrow \text{ if } (E)S[\text{else } S].$$

- Esto se puede traducir al algoritmo de reconocimiento de decisiones:

empata un `if`
empata un (
reconoce una expresión E
empata un)
reconoce una sentencia S
if el token es un `else` **then**
  empata un `else`
  reconoce una sentencia S.

Uso de EBNF para repetición

- No se puede usar directamente la regla $E \rightarrow EST|T$ para reconocer una expresión puesto que esto implica que lo primero que haga ese procedimiento sea llamarse recursivamente, es decir, un ciclo infinito.
- En vez de eso se reescribe en EBNF para obtener $E \rightarrow T\{ST\}$.
- Esto se puede traducir al algoritmo de reconocimiento de expresiones:

reconoce un término T
while token es un $+$ o un $-$ **do**
  empata un token (ya sea $+$ o $-$)
  reconoce un término T.

- Aquí se eliminó el no terminal S (que sólo puede ser $+$ o $-$).

Otro uso de EBNF para repetición

- De manera similar no se puede usar la regla $T \rightarrow TMF|F$ para reconocer un término.
- En vez de eso se reescribe en EBNF para obtener $T \rightarrow F\{MF\}$.
- Esto se puede traducir al algoritmo de reconocimiento de expresiones:

reconoce un factor F
while token es un $*$ **do**
  empata un $*$
  reconoce un factor F.

- Aquí se eliminó el no terminal M (que sólo puede ser $*$).

Conversión de EBNF a código

- Este método es muy poderoso y lo usaremos después.
- Pero debemos tener cuidado de seguir algunas reglas.
- Por ejemplo, para mantener la variable de token:
  - Esta debe inicializarse antes de que comience el análisis sintáctico.
  - Debe leerse un token precisamente después de haber comprobado otro token.
- Un cuidado similar debe seguirse al programar la construcción del árbol sintáctico.
- Esto será vital para mantener la asociatividad de los operadores.

Construcción del árbol sintáctico de una expresión

- El siguiente algoritmo corresponde a la construcción del árbol sintáctico de una expresión E:

$t \leftarrow$ árbol de un término
while token es un $+$ o un $-$ **do**
  $n \leftarrow$ nuevo nodo de operador con token
  empata un token
  hijo izquierdo de $n \leftarrow t$
  hijo derecho de $n \leftarrow$ árbol de un término
  $t \leftarrow n$
return t

- En este algoritmo t y n son dos variables temporales locales.

Construcción del árbol sintáctico de una decisión

- El siguiente algoritmo corresponde a la construcción del árbol sintáctico de una decisión D:

empata un `if`
empata un (
$t \leftarrow$ nuevo nodo de decisión
hijo de prueba de $t \leftarrow$ árbol de una expresión
empata un)
hijo verdadero de $t \leftarrow$ árbol de una sentencia
if el token es un `else` **then**
  empata un `else`
  hijo falso de $t \leftarrow$ árbol de una sentencia
else
  hijo falso de $t \leftarrow$ nulo.

- En este algoritmo t es una variable temporal local.

Contenido

El método básico del análisis sintáctico LL(1)

- El análisis sintáctico LL(1) usa una pila explícita en vez de recursión.
- Representaremos una pila como una lista que crece hacia la derecha.
- Para comenzar haremos un ejemplo con la gramática

$$S \to (S)S|\epsilon$$

que genera cadenas de paréntesis balanceados.

- Al inicio la pila contendrá un símbolo especial $\int$ (que permanecerá en el fondo de la pila) y el símbolo inicial de la estructura a reconocer (en este caso S).
- En nuestro ejemplo reconoceremos la cadena () seguida de un símbolo especial $\int$ que señala el fin de la entrada.

Ejemplo de reconocimiento de una cadena

Paso	Pila	Entrada	Acción
1	$\int S$	$()\int$	$S \to (S)S$
2	$\int S)S($	$()\int$	empata (
3	$\int S)S$	$)\int$	$S \to \epsilon$
4	$\int S)$	$)\int$	empata)
5	$\int S$	$\int$	$S \to \epsilon$
6	$\int$	$\int$	acepta

Caso general

- El analizador sintáctico comenzará con el símbolo especial $\int$ y el símbolo inicial en la pila.
- Aceptará una cadena de entrada si después de una serie de acciones la pila y la entrada terminan únicamente con el símbolo especial.

Paso	Pila	Entrada	Acción
1	$\int S$	$ABC\int$	acción
$\vdots$	$\int \cdots$	$\cdots \int$	acción
N	$\int$	$\int$	acepta

Acciones básicas de un analizador sintáctico

- El analizador sintáctico realiza su trabajo al reemplazar un no terminal en la parte superior de la pila por una de las selecciones en la regla gramatical para ese no terminal.
- La idea es producir el token en la entrada en la parte superior de la pila.
- De esta manera, las dos acciones básicas de un analizador sintáctico son:

  Generar: Reemplazar un no terminal A en la parte superior de la pila por una cadena α usando la regla gramatical $A \to \alpha$.

  Empatar: Empatar el token en la parte superior de la pila con el siguiente token de la entrada.

- Observe que la cadena α se debe insertar en reversa.
- También observe que la lista de acciones corresponde con una derivación por la izquierda.

Tabla de análisis sintáctico LL(1)

- Cuando un terminal está en la parte superior de la pila sólo hay dos opciones:
  - Si coincide con el siguiente token de la entrada se empata.
  - Si no coincide entonces se ha detectado un error de sintaxis.
- En cambio, cuando un no terminal está en la parte superior de la pila se debe tomar una decisión, basada en el token de la entrada, que selecciona la regla gramatical que reemplaza al no terminal.
- Estas decisiones de pueden codificar en una tabla de análisis sintáctico.

Construcción de la tabla de análisis sintáctico

- Esta tabla es un arreglo bidimensional $M[N, T]$, donde N es el conjunto de no terminales y T es el conjunto de terminales (incluyendo al símbolo especial $\int$).
- Al inicio cada una de las entradas de la tabla está vacía.
- Agregaremos entradas a la tabla de acuerdo a dos reglas.
- Cada entrada de la tabla que al final quede vacía es un error potencial.

Reglas para la tabla de análisis sintáctico

- Estas son las dos reglas que usaremos:
  1. Si $A \to \alpha$ es una opción de producción y existe una derivación $\alpha \Rightarrow^* a\beta$ en la que a es un token, entonces se agrega $A \to \alpha$ a la entrada $M[A, a]$.
  2. Si $A \to \alpha$ es una opción de producción y existen derivaciones $\alpha \Rightarrow^* \epsilon$ y $S \Rightarrow^* \beta Aa\gamma$, entonces se agrega $A \to \alpha$ a la entrada $M[A, a]$.
- Las ideas detrás de estas dos reglas son:
  1. Si A está en el tope de la pila y a es el token siguiente entonces debemos escoger una regla $A \to \alpha$ tal que α genere una a que podamos empatar con la entrada.
  2. Si A está en el tope de la pila y genera la cadena vacía con la derivación $A \Rightarrow \alpha \Rightarrow^* \epsilon$ entonces debemos escoger una regla $A \to \alpha$ si a es un token que puede venir legalmente después de A en una derivación.

Ejemplo de tabla de análisis sintáctico

- Considere la gramática $S \to (S)S|\epsilon$.
- Como sólo hay una producción no vacía de S

$$S \to (S)S$$

entonces cada cadena derivable de S es vacía o comienza con un (y por lo tanto esta producción se agrega a $M[S,(]$.
- Como $S \to (S)S$ entonces la segunda regla se aplica con

$$\alpha = \epsilon, \beta = (, A = S, a =) \text{ y } \gamma = S$$

de manera que $S \to \epsilon$ se agrega a $M[S,)]$.
- Como $S \Rightarrow^* S$ entonces $S \to \epsilon$ se agrega a $M[S, \int]$.
- Esto completa la tabla:

$M[N,T]$	(	)	$\int$
S	$S \to (S)S$	$S \to \epsilon$	$S \to \epsilon$

Gramáticas LL(1)

- Observe que la tabla presentada contiene a lo más una opción para cada combinación de no terminal en la pila y token en la entrada.
- A las gramáticas que cumplen esto se les llama gramáticas LL(1).
- Esto también significa que una gramática que cumple esta propiedad no puede ser ambigua (aunque en ocasiones permitiremos una ambigüedad si tenemos una regla de eliminación de ambigüedad).
- A continuación mostraremos un algoritmo que es capaz de realizar el análisis sintáctico usando la tabla de análisis sintáctico para una gramática LL(1).

Algoritmo de análisis sintáctico LL(1)

Inserta $\int$ y el símbolo inicial en la pila
while el tope no es $\int$ **and** el token no es $\int$ **do**
  if el tope es a y el token es a **then**
    elimina el tope de la pila
    avanza un token en la entrada
  else if el tope es A **and** el token es a **and** $M[A,a]$ contiene $A \to X_1X_2 \ldots X_n$ **then**
    elimina el tope de la pila
    inserta $X_n, \ldots, X_2, X_1$ en la pila
  else
    señala un error
if el tope es $\int$ **and** el token es $\int$ **then**
  acepta
else
  señala un error.

Ejemplo con la gramática simplificada de decisiones

- Considere la gramática simplificada para decisiones anidadas:

$$\begin{aligned} S &\to I|\ \mathrm{c} \\ I &\to \text{if } (D)SE \\ E &\to \epsilon|\text{else } S \\ D &\to 0|1. \end{aligned}$$

- Entonces la tabla de análisis sintáctico correspondiente es:

$M[N,T]$	if	c	else	0	1	$\int$
$S \to$	I	c				
$I \to$	if $(D)SE$					
$E \to$			ϵ|else S			ϵ
$D \to$				0	1	

- En la entrada $M[E,\text{else}]$ se prefiere la producción $E \to \text{else } S$.

Ejemplo de análisis sintáctico LL(1)

Paso	Pila	Entrada	Acción
1	$\int S$	$i(0)i(1)cec\int$	$S \to I$
2	$\int I$	$i(0)i(1)cec\int$	$I \to i(D)SE$
3	$\int ES)D(i$	$i(0)i(1)cec\int$	empata i
4	$\int ES)D($	$(0)i(1)cec\int$	empata (
5	$\int ES)D$	$0)i(1)cec\int$	$D \to 0$
6	$\int ES)0$	$0)i(1)cec\int$	empata 0
7	$\int ES)$	$)i(1)cec\int$	empata)
8	$\int ES$	$i(1)cec\int$	$S \to I$
9	$\int EI$	$i(1)cec\int$	$I \to i(D)SE$
10	$\int EES)D(i$	$i(1)cec\int$	empata i
11	$\int EES)D($	$(1)cec\int$	empata (

Ejemplo de análisis sintáctico LL(1)

Paso	Pila	Entrada	Acción
12	$\int EES)D$	$1)cec\int$	$D \to 1$
13	$\int EES)1$	$1)cec\int$	empata 1
14	$\int EES)$	$)cec\int$	empata)
15	$\int EES$	$cec\int$	$S \to c$
16	$\int EEc$	$cec\int$	empata c
17	$\int EE$	$ec\int$	$E \to eS$
18	$\int ESe$	$ec\int$	empata e
19	$\int ES$	$c\int$	$S \to c$
20	$\int Ec$	$c\int$	empata c
21	$\int E$	$\int$	$E \to \epsilon$
22	$\int$	$\int$	aceptar

Eliminación de recursión y factorización por la izquierda

- En el análisis LL(1) se presentan los mismos problemas relacionados a la repetición y la selección que se presentan en el análisis recursivo.
- Para el análisis recursivo la solución fue usar la notación EBNF.
- Pero ahora no podemos hacer lo mismo y debemos reescribir la gramática.
- Las dos técnicas estándares que se usan son:
  - Eliminación de la recursión por la izquierda.
  - Factorización por la izquierda.
- Aunque ninguna de estas dos técnicas garantiza que el resultado será una gramática LL(1), en lo general son útiles en las situaciones prácticas.

Eliminación de la recursión por la izquierda

- Esta técnica se usa para hacer operaciones asociativas por la izquierda.
- Dos ejemplos simples son

$$E \to EST|T$$

y

$$E \to E + T|E - T|T$$

en los que existe recursión inmediata por la izquierda.
- Un ejemplo más complicado involucraría recursión indirecta por la izquierda:

$$A \to Bb|\ldots \text{ y } B \to Aa|\ldots$$

aunque esto casi nunca ocurre en la práctica.

Recursión inmediata por la izquierda simple

- En este caso estamos considerando reglas gramaticales de la forma

$$A \to A\alpha|\beta$$

donde α y β son cadenas de terminales y no terminales y β no comienza con A.
- Estas reglas generan cadenas de la forma $\beta\alpha^*$.
- El caso base es $A \to \beta$ y el recursivo es $A \to A\alpha$.
- Para eliminar la recursión inmediata por la izquierda se reemplazan estas reglas por otras dos reglas: una que primero genera β

$$A \to \beta A'$$

y otra que genera las repeticiones de α usando recursión por la derecha

$$A' \to \alpha A'|\epsilon.$$

Ejemplo de recursión inmediata por la izquierda simple

- Considere la regla recursiva inmediata por la izquierda

$$E \to EST|T$$

que genera expresiones simples.
- Esta regla se puede reescribir como

$$E \to TE'$$

para generar un término y luego

$$E' \to STE'|\epsilon$$

para generar las siguientes sumas de términos.

Recursión inmediata por la izquierda general

- En este caso estamos considerando reglas gramaticales de la forma

$$A \to A\alpha_1|A\alpha_2|\cdots|A\alpha_n|\beta_1|\beta_2|\cdots|\beta_m$$

donde α_i y β_j son cadenas de terminales y no terminales y ninguna de las β_j comienza con A.
- Estas reglas generan cadenas de la forma $\beta_j\{\alpha_i : 1 \leq i \leq n\}^*$.
- El caso base es $A \to \beta_j$ y el recursivo es $A \to A\alpha_i$.
- Para eliminar la recursión inmediata por la izquierda se reemplazan estas reglas por otros dos conjuntos de reglas: uno que primero genera β_j

$$A \to \beta_1 A'|\beta_2 A'|\cdots|\beta_m A'$$

y otro que genera las repeticiones de α_i usando recursión por la derecha

$$A' \to \alpha_1 A'|\alpha_2 A'|\cdots|\alpha_n A'|\epsilon.$$

Ejemplo de recursión inmediata por la izquierda general

- Considere la regla recursiva inmediata por la izquierda

$$E \to E + T|E - T|T$$

que genera expresiones simples.
- Esta regla se puede reescribir como

$$E \to TE'$$

para generar un término y luego

$$E' \to +TE'|-TE'|\epsilon$$

para generar las siguientes sumas de términos.

Recursión por la izquierda general

- Enseguida mostramos un algoritmo que sólo está garantizado para funcionar con gramáticas sin producciones vacías ($A \to \epsilon$) ni ciclos de al menos un paso ($A \Rightarrow^+ A$).
- Se supone que los no terminales tienen un orden $A_1, \ldots, A_m$.

for $i = 1$ **to** m **do**
  for $j = 1$ **to** $i - 1$ **do**
    Reemplaza cada selección de regla gramatical de la forma $A_i \to A_j\beta$ por la regla $A_i \to \alpha_1\beta|\cdots|\alpha_k\beta$ donde $A_j \to \alpha_1|\cdots|\alpha_k$ es la regla actual para A_j.
  Elimina la recursión inmediata por la izquierda de A_i.

Ejemplo de recursión por la izquierda general

- Considere la gramática $A \to Ba|Aa|c$ y $B \to Bb|Ab|d$ con el orden A, B.
- Primero se elimina la recursión inmediata por la izquierda de A de modo que la primera regla se reemplaza por $A \to BaA'|cA'$ y $A' \to aA'|\epsilon$.
- Segundo se elimina la regla $B \to Ab$ de modo que la segunda regla se reemplaza por $B \to Bb|BaA'b|cA'b|d$.
- Tercero se elimina la recursión inmediata por la izquierda de B de modo que la segunda regla se reemplaza por $B \to cA'bB'|dB'$ y $B' \to bB'|aA'bB'|\epsilon$.

Ejemplo de eliminación de recursión por la izquierda

- Si eliminamos la recursión por la izquierda de la gramática de expresiones (que era recursiva por la izquierda para representar la asociatividad de las operaciones) obtenemos:
  - $E \to TE'$.
  - $E' \to STE'|\epsilon$.
  - $S \to +|-$.
  - $T \to FT'$.
  - $T' \to MFT'|\epsilon$.
  - $M \to *$.
  - $F \to (E)|N$.
- Esta gramática no es recursiva por la izquierda y es LL(1), pero sus árboles de análisis gramaticales ya no expresan la asociatividad izquierda de la resta.

Árbol de análisis gramatical de $a - b - c$

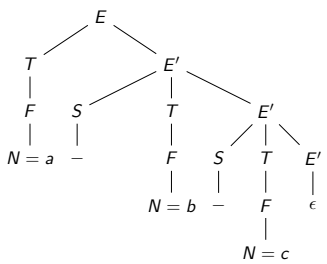


Tabla LL(1) para las expresiones simples

$M[N,T]$	(	N	)	+	−	*	∫
$E \to$	TE'	TE'					
$E' \to$			ϵ	STE'	STE'		ϵ
$S \to$				+	−		
$T \to$	FT'	FT'					
$T' \to$			ϵ	ϵ	ϵ	MFT'	ϵ
$M \to$						*	
$F \to$	(E)	N					

Factorización por la izquierda

- La factorización por la izquierda se requiere cuando dos o más opciones de reglas gramaticales comparten un prefijo común

$$A \to \alpha\beta|\alpha\gamma.$$

- Esta situación ocurre en la regla recursiva por la derecha para programas

$$P \to S;P|S$$

o en esta versión de decisiones

$$I \to \text{if } (E)S|\text{if } (E)S \text{ else } S.$$

- En este caso la solución es simplemente factorizar la α por la izquierda y reescribir la regla con las dos reglas $A \to \alpha A'$ y $A' \to \beta|\gamma$.

Algoritmo para factorización por la izquierda

while existan cambios en el lenguaje **do**
  for cada no terminal A **do**
    Sea α un prefijo de longitud máxima que se comparte por dos o más opciones de producción para A.
    if $\alpha \neq \epsilon$ **then**
      Sean $A \to \alpha_1|\cdots|\alpha_n$ todas las selecciones de producción para A y supongamos que $\alpha_1|\cdots|\alpha_k$ comparten α, de manera que $A \to \alpha\beta_1|\cdots|\alpha\beta_k|\alpha_{k+1}|\cdots|\alpha_n$, las β_j no comparten prefijos comunes y las $\alpha_{k+1},\ldots,\alpha_n$ no comparten α.
      Reemplace la regla $A \to \alpha_1|\cdots|\alpha_n$ por las reglas $A \to \alpha A'|\alpha_{k+1}|\cdots|\alpha_n$ y $A' \to \beta_1|\cdots|\beta_k$.

Primer ejemplo de factorización por la izquierda

- Considere la producción

$$P \to S;P|S$$

de las secuencias de sentencias.
- La producción tiene un prefijo compartido S que se puede factorizar por la izquierda.
- Esto nos genera las dos producciones

$$P \to SP'$$

y

$$P' \to ;S|\epsilon.$$

Segundo ejemplo de factorización por la izquierda

- Considere la producción

$$I \to \text{if } (E)S|\text{if } (E)S \text{ else } S$$

para decisiones simples.
- La producción tiene un prefijo compartido if $(E)S$ que se puede factorizar por la izquierda.
- Esto nos genera las dos producciones

$$I \to \text{if } (E)SI'$$

y

$$I' \to \text{ else } S|\epsilon.$$

Construcción del árbol sintáctico en análisis LL(1)

- La construcción del árbol sintáctico cuando se usa análisis sintáctico LL(1) no es trivial.
- Una razón es que la eliminación de la recursión izquierda y la factorización por la izquierda oscurecen la gramática.
- Pero la razón principal es que la pila de análisis sintáctico no contiene tokens vistos, sino predicciones de tokens.
- Por lo tanto, se debe posponer la creación de nodos del árbol hasta que se eliminen las estructuras de la pila.
- En general, esto requiere que se use una pila extra para seguir la pista de los nodos del árbol sintáctico y que se inserten marcadores de acción en la pila de análisis sintáctico.

Ejemplo de cálculo de valores

- En lugar de presentar un ejemplo de la creación del árbol sintáctico, presentaremos un ejemplo de cálculo de valores usando expresiones muy simples (sumas).
- Usaremos la gramática $E \to NE'$ y $E' \to +NE'|\epsilon$.
- Para calcular el valor de una expresión necesitamos dos acciones sobre una pila de valores:
  - Insertar un valor cuando se empata en la entrada: esto se hará en el procedimiento empata.
  - Sumar dos números en la pila de valores: esto se hará insertando un símbolo especial Σ en la pila de análisis sintáctico.
- El símbolo Σ ahora debe igualar un $+$ en la regla para E', es decir, $E' \to +N\Sigma E'|\epsilon$.
- Esto hace que la suma se programa después de haber leído el siguiente número, pero antes de que se procese el siguiente no terminal.
- Así se garantiza la asociatividad por la izquierda.

Ejemplo de pila de análisis sintáctico con pila de valores

Pila AS	Entrada	Acción	Pila V
$\int E$	$3+4+5\int$	$E \to NE'$	$\int$
$\int E'n$	$3+4+5\int$	empatar e insertar	$\int$
$\int E'$	$+4+5\int$	$E' \to +N\Sigma E'$	$3\int$
$\int E'\Sigma N+$	$+4+5\int$	empatar	$3\int$
$\int E'\Sigma N$	$4+5\int$	empatar e insertar	$3\int$
$\int E'\Sigma$	$+5\int$	sumar en la pila	$4\,3\int$
$\int E'$	$+5\int$	$E' \to +N\Sigma E'$	$7\int$
$\int E'\Sigma N+$	$+5\int$	empatar	$7\int$
$\int E'\Sigma N$	$5\int$	empatar e insertar	$7\int$
$\int E'\Sigma$	$\int$	sumar en la pila	$5\,7\int$
$\int E'$	$\int$	$E' \to \epsilon$	$12\int$
$\int$	$\int$	aceptar	$12\int$

Contenido

Conjunto primero

- Si X es un símbolo de la gramática (terminal, no terminal o ϵ) entonces el conjunto primero de X, llamado $P(X)$, compuesto de terminales y posiblemente ϵ se define de la siguiente manera:
  1. Si X es un terminal o ϵ entonces $P(X) = \{X\}$.
  2. Si X es no terminal entonces para cada producción $X \to X_1 X_2 \dots X_n$ el conjunto $P(X)$ contiene a $P(X_1) \setminus \{\epsilon\}$. Si también para alguna $i < n$ todos los conjuntos $P(X_1), \dots, P(X_i)$ contienen ϵ entonces el conjunto $P(X)$ contiene a $P(X_{i+1}) \setminus \{\epsilon\}$. Si todos los conjuntos $P(X_1), \dots, P(X_n)$ contienen ϵ entonces el conjunto $P(X)$ contiene a ϵ.
- De manera similar se define $P(\alpha)$ para cualquier cadena α de terminales y no terminales.

Algoritmo para calcular el conjunto primero

for todo no terminal A **do**
  $P(A) \leftarrow \emptyset$
while existan cambios a cualquier $P(A)$ **do**
  for cada selección de producción $A \to X_1 X_2 \dots X_n$ **do**
    $k \leftarrow 1$
    $c \leftarrow$ **true**
    while $c =$ **true and** $k \leq n$ **do**
      agregar $P(X_k) \setminus \{\epsilon\}$ a $P(A)$
      if $\epsilon \notin P(X_k)$ **then**
        $c \leftarrow$ **false**
      $k \leftarrow k + 1$
    if $c =$ **true then**
      agregar ϵ a $P(A)$

Algoritmo simplificado para calcular el conjunto primero

- Si no hay producciones $A \to \epsilon$ se puede simplificar el algoritmo:

  for todo no terminal A **do**
    $P(A) \leftarrow \emptyset$
  while existan cambios a cualquier $P(A)$ **do**
    for cada selección de producción $A \to X_1 X_2 \dots X_n$ **do**
      agregar $P(X_1)$ a $P(A)$
- En caso de que sí haya producciones $A \to \epsilon$, el algoritmo original también calcula los no terminales anulables, es decir, aquellos no terminales A para los que existe alguna derivación $A \Rightarrow^* \epsilon$.
- Un no terminal A es anulable si y sólo si $P(A)$ contiene a ϵ.

Ejemplo de cálculo del conjunto primero para expresiones

- Considere la gramática de expresiones simples con nueve producciones:

$$\begin{array}{lll} E \to EST & E \to T & S \to + \\ S \to - & T \to TMF & T \to F \\ M \to * & F \to (E) & F \to N \end{array}$$

- Como no tiene producciones ϵ podemos usar el algoritmo simplificado.
- En el primer paso se actualiza $P(S) = \{+\}$, $P(S) = \{+, -\}$, $P(M) = \{*\}$, $P(F) = \{(\}$ y $P(F) = \{(, N\}$.
- En el segundo paso se actualiza $P(T) = \{(, N\}$.
- En el tercer paso se actualiza $P(E) = \{(, N\}$.
- En el cuarto paso ya no hay actualizaciones.

Ejemplo de cálculo del conjunto primero para decisiones

- Considere la gramática de decisiones con siete producciones:

$$\begin{array}{ll} S \to I & S \to c \\ I \to \text{if } (D) SE & E \to \text{else } S \\ E \to \epsilon & D \to 0 \\ D \to 1 & \end{array}$$

- Como tiene producciones ϵ debemos usar el algoritmo completo.
- En el primer paso se actualiza $P(S) = \{c\}$, $P(I) = \{\text{if}\}$, $P(E) = \{\text{else}\}$, $P(E) = \{\text{else}, \epsilon\}$, $P(D) = \{0\}$ y $P(D) = \{0, 1\}$.
- En el segundo paso se actualiza $P(S) = \{c, \text{if}\}$.
- En el tercer paso ya no hay actualizaciones.

Conjunto siguiente

- Si A es un no terminal de la gramática entonces el conjunto siguiente de X, llamado $S(X)$, compuesto de terminales y posiblemente $\int$ se define de la siguiente manera:
  1. Si A es el símbolo inicial entonces $\int$ está en $S(A)$.
  2. Si hay una producción $B \to \alpha A \gamma$ entonces $P(\gamma) \setminus \{\epsilon\}$ está en $S(A)$.
  3. Si hay una producción $B \to \alpha A \gamma$ tal que ϵ está en $P(\gamma)$ entonces $S(A)$ contiene a $S(B)$.
- Observe que $\int$ siempre está en $S(A)$ para el símbolo inicial A y que ϵ nunca es un elemento de $S(A)$.

Algoritmo para calcular el conjunto siguiente

S(símbolo inicial) $\leftarrow \{\int\}$
for todo no terminal $A \neq$ símbolo inicial **do**
  $S(A) \leftarrow \emptyset$
while existan cambios a cualquier $S(A)$ **do**
  for cada selección de producción $A \to X_1 X_2 \dots X_n$ **do**
    for cada X_i que sea un no terminal **do**
      agregar $P(X_{i+1} X_{i+2} \dots X_n) \setminus \{\epsilon\}$ a $S(X_i)$
      if $\epsilon \in P(X_{i+1} X_{i+2} \dots X_n)$ **then**
        agregar $S(A)$ a $S(X_i)$

Ejemplo de cálculo del conjunto siguiente para expresiones

- Considere la gramática de expresiones simples con nueve producciones:

$$\begin{array}{lll} E \to EST & E \to T & S \to + \\ S \to - & T \to TMF & T \to F \\ M \to * & F \to (E) & F \to N \end{array}$$

- Para la que obtuvimos $P(E) = \{(, N\}$, $P(T) = \{(, N\}$, $P(F) = \{(, N\}$, $P(S) = \{+, -\}$ y $P(M) = \{*\}$.
- Comenzamos con $S(E) = \{\int\}$ y $S(T) = S(F) = S(S) = S(M) = \emptyset$.
- En el primer paso se actualiza $S(E) = \{\int, +, -\}$, $S(S) = \{(, N\}$, $S(T) = \{\int, +, -\}$, $S(T) = \{\int, +, -, *\}$, $S(M) = \{(, N\}$, $S(F) = \{\int, +, -, *\}$ y $S(E) = \{\int, +, -, *\}$.
- En el segundo paso se actualiza $S(T) = \{\int, +, -, *,)\}$ y $S(F) = \{\int, +, -, *,)\}$.
- En el tercer paso ya no hay actualizaciones.

Ejemplo de cálculo del conjunto siguiente para decisiones

- Considere la gramática de decisiones con siete producciones:

$$\begin{array}{ll} S \to I & S \to c \\ I \to \text{if } (D)SE & E \to \text{else } S \\ E \to \epsilon & D \to 0 \\ D \to 1 & \end{array}$$

- Para la que obtuvimos $P(S) = \{c, \text{if}\}$, $P(I) = \{\text{if}\}$, $P(E) = \{\text{else}, \epsilon\}$ y $P(D) = \{0, 1\}$.
- Comenzamos con $S(S) = \{\int\}$ y $S(I) = S(E) = S(D) = \emptyset$.
- En el primer paso se actualiza $S(I) = \{\int\}$, $S(D) = \{)\}$, $S(S) = \{\int, \text{else}\}$ y $S(E) = \{\int\}$.
- En el segundo paso se actualiza $S(I) = \{\int, \text{else}\}$ y $S(E) = \{\int, \text{else}\}$.
- En el tercer paso ya no hay actualizaciones.

Construcción de tablas de análisis sintáctico LL(1)

- Considere las reglas para la creación de la tabla LL(1):
  1. Si $A \to \alpha$ es una opción de producción y existe una derivación $\alpha \Rightarrow^* a\beta$ en la que a es un token, entonces se agrega $A \to \alpha$ a la entrada $M[A, a]$.
  2. Si $A \to \epsilon$ es una opción de producción y existe una derivación y $S \int \Rightarrow^* \alpha A a \beta$, donde S es el símbolo inicial y a es un token o $\int$, entonces se agrega $A \to \epsilon$ a la entrada $M[A, a]$.
- Obviamente el token a en la primera regla está en $P(\alpha)$ y el token a en la segunda regla está en $S(A)$.
- Esto nos da un algoritmo para la creación de la tabla LL(1).
- Repita los siguientes dos pasos para cada no terminal A y opción de producción $A \to \alpha$:
  1. Para cada token a en $P(\alpha)$ agregue $A \to \alpha$ a la entrada $M[A, a]$.
  2. Si ϵ está en $P(\alpha)$, para cada elemento $a \in S(A)$ agregue $A \to \alpha$ a la entrada $M[A, a]$.

Ejemplo para el cálculo de la tabla LL(1) de expresiones

- Consideremos la gramática de expresión simple a la que le hemos eliminando la recursión por la izquierda:
  - $E \to TE'$.
  - $E' \to STE' | \epsilon$.
  - $S \to + | -$.
  - $T \to FT'$.
  - $T' \to MFT' | \epsilon$.
  - $M \to *$.
  - $F \to (E) | N$.

Tabla LL(1) para las expresiones simples

- Si calculamos los conjuntos primero y siguiente obtenemos:

	E	E'	S	T	T'	M	F
P	$(, N$	$+, -, \epsilon$	$+, -$	$(, N$	$*, \epsilon$	$*$	$(, N$
S	$\int,)$	$\int,)$	$(, N$	$\int,), +, -$	$\int,), +, -$	$(, N$	$\int,), +, -, *$

- Y de esto obtenemos la tabla:

$M[N, T]$	$($	N	$)$	$+$	$-$	$*$	$\int$
$E \to$	TE'	TE'					
$E' \to$			ϵ	STE'	STE'		ϵ
$S \to$				$+$	$-$		
$T \to$	FT'	FT'					
$T' \to$			ϵ	ϵ	ϵ	MFT'	ϵ
$M \to$						$*$	
$F \to$	(E)	N					

Ejemplo para el cálculo de la tabla LL(1) de decisiones

- Considere la gramática simplificada para decisiones anidadas:

$$\begin{array}{lllllll} S & \to & I | \text{ c} & I & \to & \text{if } (D)SE \\ E & \to & \epsilon | \text{else } S & D & \to & 0 | 1. \end{array}$$

- Si calculamos los conjuntos primero y siguiente obtenemos:

	S	I	E	D
P	if,c	if	else, ϵ	0, 1
S	$\int$, else	$\int$, else	$\int$, else	$)$

- Y de esto obtenemos la tabla:

$M[N, T]$	if	c	else	0	1	$\int$
$S \to$	I	c				
$I \to$	if $(D)SE$					
$E \to$			$\epsilon \mid$else S			ϵ
$D \to$				0	1	

- .

- .