Compiladores 1

Definición de Compilador

Los compiladores son programas de computadora que traducen de un lenguaje a otro. Un compilador toma como su entrada un programa escrito en lenguaje fuente y produce su equivalente escrito en lenguaje objeto.

Generalmente al lenguaje fuente se le asocia como lenguaje de alto nivel, mientras que al lenguaje objeto se le conoce como código objeto o código máquina, escrito específicamente para una máquina objeto. A lo largo del proceso de traducción, el compilador debe informar la presencia de errores en el lenguaje fuente.

Diseñar y desarrollar un compilador, no es tarea fácil, sin embargo, los compiladores se utilizan en casi todas las formas de la computación, y cualquiera involucrado en esta área debería conocer la organización y el funcionamiento básico de un compilador.

Historia de los Compiladores

A finales de la década de **1940,** comenzaron a construirse las primeras computadoras digitales y fue necesario implementar un lenguaje capaz de realizar los cálculos, es aquí donde aparece el lenguaje máquina que representaba secuencias de códigos numéricos.

C7 06 0000 0002
INSTRUCCIÓN QUE MUEVE EL NÚMERO 2 A LA DIRECCIÓN 0000

Desafortunadamente, éste lenguaje era tedioso de seguir y complicado de mantener, por lo que esta forma de codificación fue reemplazada por el lenguaje ensamblador, en el cual las instrucciones y las localidades de memoria son formas simbólicas. Un

ensamblador traduce de los códigos simbólicos al lenguaje máquina. Aún con esta mejora, el lenguaje ensamblador sigue siendo demasiado difícil de mantener.

MOV X, 2

INSTRUCCIÓN DE ENSAMBLADOR QUE MUEVE UN 2 EQUIVALENTE A LA ANTERIOR

En este punto, se presenta la necesidad de lenguajes que permitan escribir los programas de forma concisa, similar a una notación matemática y que se pudieran traducir a código ejecutable para una máquina específica.

En la década de **1950**, **G. M. Hooper**, acuña el término compilador y aparecen los primeros trabajos sobre compiladores relacionados con la traducción de formulas aritméticas a código de máquina.

John Backus lideró un grupo de trabajo en IBM para realizar un traductor de código máquina a fórmulas matemáticas. Resultando con gran éxito la especificación de un lenguaje de alto nivel (*FORTRAN Formule Translation*) y la realización de un traductor para una máquina *IBM-704*. Trabajaron 18 personas durante más de un año durante el proyecto. Fue un compilador ad hoc, pues no existía una teoría formal, sino que se iban resolviendo las construcciones una a una, para cada situación particular.

Más o menos al mismo tiempo **Noam Chomsky** continúa con sus estudios sobre la estructura del lenguaje natural. Sus estudios los condujeron a la clasificación de los lenguajes de acuerdo a una jerarquía de sus gramáticas, sus estudios sobre los algoritmos de reconocimiento derivaron en una automatización del proceso de traducción más eficiente.

Durante la década de **1960**, se diseña el lenguaje **LISP**. En un principio, el código LISP se traducía manualmente a código máquina también para una *IBM-704*. Se escribió en LISP un programa capaz de interpretar programas LISP, que se tradujo manualmente a código de máquina, construyendo de este modo un intérprete ejecutable de LISP. A demás se desarrollan la mayoría de las técnicas de análisis sintáctico (**Knuth**).

En la década de **1970**, se presentan los mayores avances en el área de lenguajes de programación y compiladores. Aparecen los primeros programas que automatizan los procesos de análisis léxico y análisis sintáctico. Surge la llamada *Torre de Babel* debido a la proliferación de la teoría para la construcción de compiladores.

Niklaus Wirth, del Instituto Politécnico de Zúrich diseña *Pascal*, pensado para la enseñanza. Wirth propone el concepto de representación intermedia de código, separando el proceso de traducción en dos fases, el *front-end* encargado de analizar el programa fuente (operaciones dependientes soló del lenguaje fuente) y el *back-end*

encargado de generar el código para la máquina objeto (operaciones dependientes sólo del lenguaje objeto). A demás surge el concepto de *memoria dinámica*.

Ya para la década de **1980**, comienzan a proliferar las técnicas de mejoramiento de código (optimización), se consolida y prolifera el concepto de asignación y liberación de memoria dinámica. La programación orientada a objetos es extensamente utilizada y madura.

A partir de la década de **1990**, los lenguajes de programación y compiladores son muy similares a lo que tenemos actualmente, surgen los ambientes de desarrollo, los lenguajes interpretados comienzan a ganar terreno en aplicaciones de internet, y el código intermedio es altamente utilizado.

Tipos de Traductores

Un traductor es un programa que convierte un archivo de un lenguaje base a un correspondiente lenguaje objeto. Así pues, apreciaremos que un compilador en realidad es un tipo específico de traductor. Existen diversos traductores y los más significativos son:

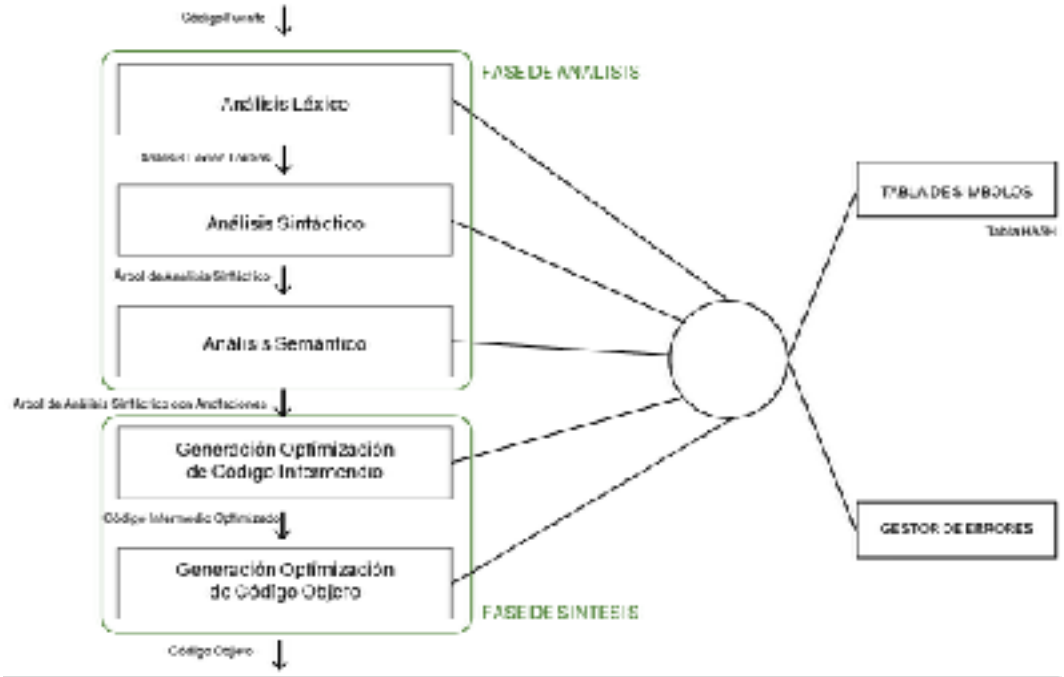
- **Ensamblador.** Programa que convierte del lenguaje ensamblador al lenguaje máquina, generando un archivo con el código objeto equivalente al código fuente completo, junto con información necesaria para el montaje.
- **Formadores de texto.** Toma como entrada una cadena de caracteres que incluye el texto a componer y órdenes para indicar capítulos, secciones, párrafos, tablas, etcétera. Por ejemplo HTML.
- **Intérpretes.** Ejecutan las instrucciones del programa fuente inmediatamente en lugar de esperar a que esté traducido por completo el código fuente a código máquina. Necesitan menos memoria, pero son más lento que los compiladores. *LISP* y *BASIC*, históricamente, fueron de los más utilizados debido a que los recursos de memoria eran escasos.

Ventajas y Desventajas de un Compilador

Ventajas de Compilador sobre Intérprete	Ventajas de Intérprete sobre Compilador
- Tiempo de ejecución. - Se compila una vez y se ejecuta n veces. - En ciclos, la compilación genera código equivalente al ciclo, pero un intérprete traduce tantas veces cada línea del ciclo mientras veces se repita el mismo. - El compilador tiene una visión global del programa, por lo que la información de mensajes de error es más detallado	- Menor consumo de memoria. - No es dependiente de la máquina. - Permite una mayor interactividad con el código en tiempo de desarrollo.

Definición de las fases de un Compilador

Un compilador se compone internamente de varias etapas o fases, que realizan operaciones lógicas. Es útil pensar en estas fases como piezas separadas (ventajas: facilita trabajar cada una por separado; si tengo que modificarlo, solo sería en el análisis; modificación, mantenimiento y actualización es mejor por separado.) dentro del compilador, y pueden en realidad escribirse como operaciones codificadas separadamente, aunque en la práctica, a menudo se integran juntas. A continuación se describe cada una de ellas.

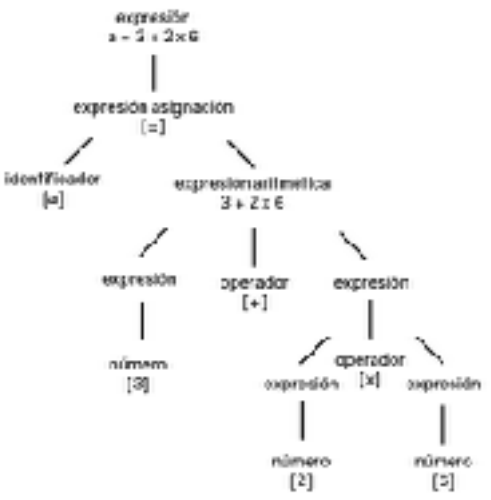


Especificación Detallada de Cada Fase

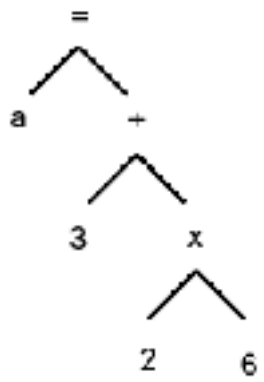
- **Analizador Léxico.** Lee la secuencia de caracteres de izquierda a derecha del programa fuente y las agrupa en unidades con significado propio (componentes léxicos o tokens). Por ejemplo: palabras clave, identificadores, operadores, constantes numéricas, signos de puntuación como separadores de sentencia, llaves, etcétera. Además de eliminación de comentarios e inclusión de archivos. La estructura léxica la modelaremos mediante expresiones regulares.

COMPONENTE LÉXICO	CATEGORÍA
int	palabra reservada
vec	identificador
[	separador de sentencia
indice	identificador
]	separador de sentencia
=	operador
3	constante numérica
+	operador
2.16	constante numérica
;	terminador de sentencia

- **Análisis Sintáctico.** Determina si la secuencia de componentes léxicos sigue la sintaxis del lenguaje y obtiene la estructura jerárquica del programa en forma de árbol, donde los nodos son las construcciones de alto nivel del lenguaje. Se determinan las relaciones estructurales entre los componentes léxicos, esto es semejante a realizar el análisis gramatical sobre una frase del lenguaje natural. La estructura sintáctica la definiremos mediante reglas gramaticales libres de contexto.



ÁRBOL DE ANÁLISIS GRAMATICAL

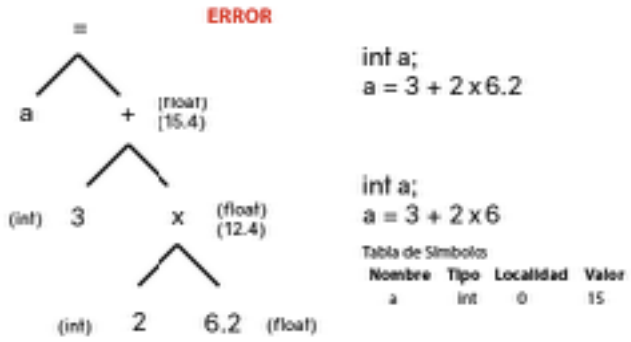


Los nodos internos del árbol de análisis gramatical están etiquetados con los nombres de las estructuras que representan y las hojas del árbol representan la secuencia de tokens.

Los árboles de análisis gramatical son útiles para visualizar la sintaxis de un programa, pero no es eficaz en la representación de esa estructura. Los analizadores sintácticos tienden a generar un árbol sintáctico abstracto, que es una simplificación de la información contenida en el árbol de análisis gramatical.

ÁRBOL SINTÁCTICO ABSTRACTO

- **Análisis Semántico.** Realiza comprobaciones necesarias sobre el árbol sintáctico para determinar el correcto *significado* del programa, por ejemplo: verificación e inferencia de tipos en asignaciones y expresiones, declaración antes de uso (variables, funciones, tipos), correcto uso de operadores, ámbito de variables, correcta llamada de funciones.



Nos limitaremos al *análisis semántico estático* (en tiempo de compilación), donde es necesario hacer uso de la *Tabla de Símbolos,* como estructura de datos para almacenar información sobre los identificadores que van surgiendo a lo largo del programa. El análisis semántico suele agregar atributos (como tipos de datos) a la estructura del árbol semántico.

Siguiendo como ejemplo de la expresión en C, el analizador semántico extrae la información de que **a** es un arreglo de valores enteros y que **índice** es una variable entera.

- **Generación optimización de Código Intermedio.** La optimización consiste en la calibración del árbol sintáctico, donde ya no aparecen construcciones de alto nivel, generando un código mejorado, ya no estructurado, más fácil de traducir directamente a código ensamblador o máquina, compuesto de un código de tres direcciones (cada instrucción tiene un operador, y la dirección de dos operando a demás de un lugar donde guardar el resultado, también conocida como código intermedio.

La etapa de optimización sólo depende del lenguaje fuente (y no de la máquina), se busca principalmente: eliminar subexpresiones comunes, identificar código muerto,

sustituir operaciones aritméticas, cálculo previo de constantes, propagación de copias o código inalcanzable. Suele ser una frase lenta y compleja.

Código no Optimizado	Código Optimizado
t = 2 x 6 **t2 = 3 + t** **a = t2**	a = 15

- **Generación Optimización de Código Objeto.** Toma como entrada la representación intermedia y genera el código ensamblador o máquina. La optimización depende de la máquina, es necesario conocer el conjunto de instrucciones, la representación de los datos (número de bytes), modo de direccionamiento, número y propósito de los registros, jerarquía de memoria, encausamientos, etcétera. Suele implementarse a mano, y son complejos, por que la generación de un buen código objeto requiere la consideración de muchos casos particulares.

También se está investigando la generación de generadores de código automático. La idea es automáticamente hacer corresponder una representación intermedia a plantillas de instrucciones objeto. Permitiendo generar fácilmente el código objeto para una nueva máquina con tan solo cambiar el conjunto de plantillas.

- **Tabla de Símbolos.** Es una estructura tipo diccionario con operaciones de inserción, borrado y búsqueda, que almacena información sobre los símbolos que van apareciendo a lo largo del programa, como son: los identificadores, etiquetas y tipos definidos por el usuario. Además almacena el tipo de dato, método de paso de parámetros, tipo de retorno y argumentos de una función, el ámbito de referencia de identificadores y la dirección de memoria. Interacciona con el analizador léxico, sintáctico y semántico, que introducen información conforme se procesa la entrada. La fase de generación de código y optimización también la usa.
- **Gestor de Errores.** Detecta e informa de errores en cada fase del compilador. Debe generar mensajes significativos y reanudar la traducción, encuentra errores en:
  - En tiempo de compilación. Errores léxicos (ortográficos), sintácticos (construcciones incorrectas) y semánticos (por ejemplo, uso de variables no declariadas, errores de tipo, etcétera).
  - En tiempo de ejecución. Direccionamiento de vectores fuera de rango, direcciones por cero, etcétera.

Se tratan solo errores estáticos (en tiempo de compilación). Respecto a los errores en tiempo de ejecución, es necesario que el traductor genere código para la comprobación de errores específicos, su adecuado tratamiento y los mecanismos de tratamiento de excepciones para que el programa se continúe ejecutando.

La mayoría de los compiladores son dirigidos por la sintaxis, es decir, el proceso de traducción es dirigido por el analizador sintáctico. El análisis sintáctico genera la estructura del programa fuente a través de *tokens*. El análisis semántico proporciona significado del programa basándose en la estructura del árbol de análisis sintáctico.

Las fases de análisis léxico y análisis sintáctico se pueden automatizar de manera relativamente fácil, las fases de análisis semántico, generación y optimización de código requieren un poco más de trabajo.

El número de pasadas, es decir, el número de veces que hay que analizar el código fuente, está en función del grado de optimización. Típicamente se realiza una pasada para realizar el análisis léxico y sintáctico, otra pasada para el análisis semántico y optimización del lenguaje intermedio y una tercera pasada para generación de código y optimizaciones dependientes de la máquina.

Estructuras de Datos Empleadas en un Compilador

Las principales estructuras de datos que se necesitan en el proceso de traducción y que sirven para comunicarse entre las fases son:

- **Componentes Léxicos.** Estructura tipo registro con dos campos, el tipo de componente léxico que se representa por un tipo enumerado y el lexema como una cadena de caracteres.
- **Árbol Sintáctico.** Se genera de forma dinámica como una estructura estándar basada en apuntadores conforme se avanza en el proceso de análisis sintáctico. Cada nodo es un registro con información obtenida por el analizador léxico (lexema), sintáctico (tipo) y semántico (valor y/o tipo de una expresión). Esta información depende del tipo de construcción del lenguaje que ese nodo representa.
- **Tabla de Símbolos.** Contiene información sobre los identificadores, funciones, variables, ámbito de referencia, constantes y literales numéricas, tipos de datos o incluso la dirección de memoria. Es importante que las operaciones de inserción, búsqueda y eliminación sean de costo casi constante, por lo cual se implementará una tabla hash.
- **Código Intermedio.** Se implementa como una lista de registros, donde cada registro tiene cuatro campos (operador, la dirección de los operando y del resultado) se puede usar también un archivo de texto.

Modelo de Análisis y Síntesis (Agrupamiento de Fases)

La estructura de un compilador se puede ver desde varias perspectivas, en la sección anterior se revisaron las fases que representan su estructura lógica. Los constructores

de compiladores deben estar familiarizados con los diversos tipos estructurales del compilador, ya que de esto depende su buen mantenimiento, eficiencia y confiabilidad.

En el modelo de análisis y síntesis las operaciones del compilador que analizan el programa fuente y calculan sus propiedades se clasifican como análisis del compilador, mientras que las operaciones involucradas con la traducción al código objeto se conoce como **síntesis del compilador.**

La intención de separar las etapas de análisis y síntesis, es principalmente, para realizar mantenimientos y actualizaciones más eficientes.

Compilador Cruzado

Un compilador cruzado es un compilador capaz de crear código ejecutable para una plataforma distinta a aquella en la que él se ejecuta. Esta herramienta es útil cuando quiere compilarse código para una plataforma a la que no se tiene acceso o cuando es imposible compilar en dicha plataforma.

Es perfectamente normal construir un compilador de *pascal* que genere código para *ms-dos* y que el compilador funcione en *linux* y se haya escrito en *c++*. Existe también la variante que implica un compilador para una máquina abstracta, que facilita la transportabilidad de compiladores de un lenguaje fuente a varias máquinas objeto. La construcción de este tipo de compiladores se realiza en dos etapas, el *frontend* o el etapa inicial y el *backend* o etapa final. Si se cambia el lenguaje fuente, entonces, se reescribe el *frontend*. Si se cambia la máquina objeto entonces se re escribe el *backend.* Si aparece una nueva arquitectura, basta con desarrollar un traductor de lenguaje intermedio para esa nueva máquina.

Herramientas de Construcción de Compiladores.

Son programas de ayuda en el proceso de escritura de compiladores. Sistemas generadores de traductores a los cuales también se les conoce como herramientas de escritura de compiladores, generadores de compiladores o compiladores de compiladores. Los más conocidos son:

1. **Generadores de analizadores léxicos.** Ordinariamente trabajan a partir de una especificación basada en expresiones regulares, como LEX y FLEX.
2. **Generadores de analizadores sintácticos.** A partir de una entrada que es la gramática libre de contexto que representa la estructura sintáctica del lenguaje, generan un árbol sintáctico, como YACC / BISON.

Unidad 2: Análisis Léxico

Función de Análisis Léxico (Scanner)

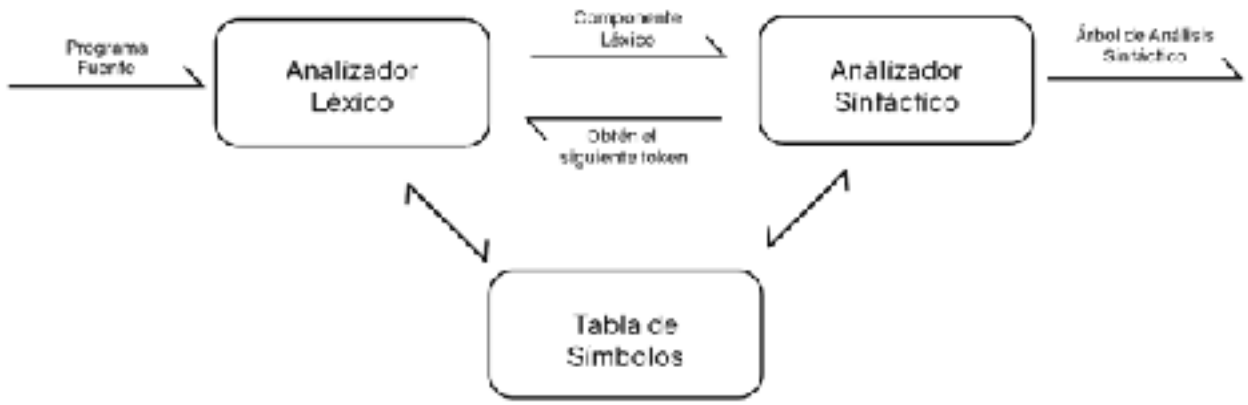
Manejo del buffer de entrada.

La fase de rastreo (scanner), tiene las funciones de leer el programa fuente como un archivo de caracteres y dividirlo en tokens. Los tokens son las palabras reservadas de un lenguaje, secuencias de caracteres que representan una unidad de información en el programa fuente. En cada caso un token representa un cierto patrón de caracteres que el analizador léxico reconoce, o ajusta desde el inicio de los caracteres de entrada. De tal manera, es necesario generar un mecanismo computacional que nos permite identificar el patrón de transiciones entre los caracteres de entrada, generando tokens que posteriormente serán clasificados. Este mecanismo, es posible crearlo a partir de un tipo específico de máquina de estados llamado autómata finito.

Su principal función consiste en leer la secuencia de caracteres del programa fuente carácter a carácter, y elaborar como salida la secuencia de componentes léxicos que utilizará el analizador sintáctico. El analizador sintáctico emite la orden al analizador léxico para que agrupen los caracteres y forme unidades con significado propio llamados componentes léxicos (tokens). Los componentes léxicos representan:

- **palabras reservadas:** if, while, do…
- **identificadores:** variables, funciones, etiquetas…
- **operadores:** = > < + - …
- **símbolos especiales:** ; ( ) { } …
- **constantes numéricas:** enteras y reales.
- **constantes de caracter:** cadenas de caracteres.

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme a analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática. Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden obtiene el siguiente componente léxico, el analizador léxico, lee los caracteres de entrada hasta identificar el siguiente componente léxico.



A demás el analizador léxico es responsable de:

- Manejo de apertura y cierre de archivo, lectura de caracteres y gestión de posibles errores de lectura.
- Eliminar comentarios, espacios en blanco, tabuladores y saltos de línea.
- Inclusión de archivos.
- Contabilizar número de líneas y de columnas para emitir mensajes de error.

Componentes Léxicos, Patrones y Lexemas

En la fase de análisis léxico, los términos *componente léxico* (token), *patrón* y *lexema*, se emplean con significados específicos. Un analizador léxico, inicialmente lee los lexemas y les asigna un significado propio.

Componente léxico. Es la secuencia lógica y coherente de caracteres relativos a una categoría.

Patrón. Es una regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico.

Lexema. Es una cadena de caracteres que concuerda con un patrón que describe un componente léxico.

Ejemplo: **int a = 3.1415; cout << “Buenas Tardes”;**

Lexema	Componente Léxico	Patrón
int	palabra reservada	int, do, while, for…
a	identificador	letra(letra | digito)*
=	operadores	= < > + - …

Lexema	Componente Léxico	Patrón
3.1415	constante numérica	número real
;	símbolos especiales	; ( ) { } …
cout	palabra reservada	int, do, while, for…
<<	operadores	= < > + - …
"Buenas Tardes”	constante de caracter	“ (caracteres)* “
;	símbolos especiales	; ( ) { } …

El analizador léxico recoge información sobre los componentes léxicos en sus atributos asociados. Los tokens influyen en las decisiones del análisis sintáctico, y los atributos en la traducción de los tokens. En la práctica los componentes léxicos suelen tener solo un atributo, pero para efectos de diagnóstico puede considerarse tanto en lexema para un identificador como el número de línea en el que se encontró por primera vez. Esta información puede ser almacenada en la tabla de símbolos para el identificador.

Especificación de Componentes Léxicos

Las expresiones regulares son una notación importante para especificar patrones. Cada patrón concuerda con una serie de cadenas de modo que las expresiones regulares servirán como nombres para el conjunto de cadenas. El término alfabeto denota cualquier conjunto finito de símbolos, por ejemplo, el alfabeto binario (0,1). Una cadena es una secuencia finita de símbolos de un alfabeto y un lenguaje se refiere al conjunto de cadenas de un alfabeto fijo.

---

1. Expresiones Regulares

El origen de las expresiones regulares surge de la teoría de autómatas y la teoría de lenguajes formales, ambas partes de las ciencias computacionales teóricas. Este campo estudia los modelos computacionales (autómata) y la manera de describir y clasificar los lenguajes formales. Un lenguaje formal puede ser especificado de varias maneras, tales como:

1. Cadenas producidas por alguna gramática formal.
2. Cadenas producidas por expresiones regulares.
3. Cadenas aceptadas por algunos autómatas, tales como las *máquinas de Turing* o autómatas de estado finito.

A las expresiones regulares frecuentemente se les llama patrones, ya que son expresiones que describen a un conjunto de cadena. Frecuentemente son usadas para dar una descripción concisa de un conjunto sin tener que listar todos sus elementos.

Dado un alfabeto ∑ las siguientes constantes son definidas:

L(ø) denota el conjunto vacío {}

L(ε) denota el conjunto {}

L(a), a es elemento de ∑ denota el conjunto {a}

Operaciones para expresiones regulares

Existen tres operaciones básicas en las expresiones regulares:

1. **Selección de Alternativas.** Se indica mediante el metacaracter **I**. Si r y s son expresiones regulares entonces r I s es una expresión regular que define cualquier cadena que concuerda con r o con s. En términos de lenguaje, el lenguaje de r I s es la unión de r y s.

$$SI\ R\ Y\ S \rightarrow R\ I\ S \rightarrow L(\ R\ I\ S) \rightarrow L(R)\ U\ L(S)$$

Ejemplo: Considere el alfabeto ∑ = (a,b) y la expresión regular alb. Esta corresponde al carácter de a como al carácter de b.

$$alb \rightarrow L(alb) \rightarrow L(a)\ U\ L(b) = \{a,b\}$$

La selección se puede extender a más de una alternativa, de manera que la siguiente expresión es válida.

$$(alblcld) = L(alblcld) \rightarrow L(a)\ U\ L(b)\ U\ L(c)\ U\ L(d) = \{a,b,c,d\}$$

2. **Concatenación.** La concatenación de dos expresiones regulares r y s se escribe como rs, y corresponde a cualquier cadena que sea la concatenación de dos cadenas, con la primera de ellas correspondiendo a r y la segunda correspondiendo a s. La concatenación en términos de lenguaje se definir como sigue:

$$R\ Y\ S \rightarrow RS = L(RS) = L(R)\ L(S) = \{RS\}$$

Ejemplo: La expresión regular ab corresponde solo a la cadena ab, mientras que la expresión regular (alb)c corresponde a las cadenas ac y bc.

$$(alb)c \rightarrow \{ac,bc\}$$

3. **Repetición.** La operación de repetición de una expresión regular, denominada también en ocasiones como *cerradura de Kleene* se escribe r*, donde r es una expresión regular. La expresión regular r*corresponde a cualquier concatenación finita de cadenas, cada una de las cuales corresponde a r.

Ejemplo: La expresión regular a* corresponde a las cadenas

$$a^* = \{\varepsilon, a, aa, aaa, \ldots\}$$

Identificadores.

(a|b|c|…|z|A|B|C|…|Z|_) (a|b|c|…|z|A|B|C|…|Z|_|0|1|…|9)*.

Precedencia de Operadores y Uso de Paréntesis

La descripción precedente no toma en cuenta la cuestión de la precedencia de las operaciones de selección, concatenación y repetición.

Por ejemplo, dada la expresión regular a|b* se debería de representar como (a|b)* o a|(b)*. Entre las tres operaciones se le da a la repetición la precedencia más alta, a la concatenación la que sigue y a la selección la precedencia más baja, de este modo la interpretación correcta de la expresión regular antes descrita es: a|(b)*.

Ejercicio:

dígitos = (0|1|2|3|4|5|6|7|8|9)

signo = (+|-|ε)

minúsculas = (a|b|c|…|x|y|z)

hexadecimal = (0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F)

- Escribir una expresión regular para números enteros.

  dígitos dígitos*

- Escribir una expresión regular para números enteros con o sin signo.

  signo dígitos dígitos*

- Escribir una expresión regular para números naturales que no comiencen con cero.

  (1|2|3|4|5|6|7|8|9) dígitos*

- Escribir una expresión regular en un alfabeto ∑={a,b,c} de todas las cadenas dentro de este alfabeto que contengan como máximo una b.

  (a|c)* (b|ε) (a|c)*

- Una expresión regular que tenga todos los números pares.

  signo dígitos* (0|2|4|6|8)

- Considere las cadenas en el alfabeto ∑={a,b,c}, que no contienen dos b consecutivas y escriba una expresión regular para dichas cadenas.

(b|ε) (a|c)* ((a|c) (b|ε) (a|c)*)*

((a|c)* (b|ε) (a|c))* (b|ε)

(a|c|ba|bc)* (b|ε)

- Hacer una expresión regular considerando el alfabeto ∑={a,b,c} para todas las cadenas que contengan un número par de a.

(((b|c)* a (b|c)* a (b|c)*) | (b|c))*

((b|c)* a (b|c)* a (b|c)*)* (b|c)*

- Hacer una expresión regular para todas las cadenas de letras minúsculas que comiencen y finalicen con a.

minúsculas = (a|b|c|…|x|y|z)

(a minúsculas* a | a)

(a minúsculas* a)* (a|ε)

- Hacer una expresión regular para números hexadecimales.

hexadecimal = (0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F)

hexadecimal hexadecimal*

- Hacer una expresión regular de todas las cadenas de dígitos tales que todos los números 2 se presenten antes que todos los 9.

auxiliar = (0|1|3|4|5|6|7|8)

(2|auxiliar)* (9|auxiliar)*

números = (0|1|3|4|5|6|7|8|ε)

(números 2 9 números)*

Extensiones para las Expresiones Regulares

- Una o más repeticiones. metasímbolo “+” Ejemplo:

(0|1) (0|1)* == $(0|1)^+$

dígito (0|1|2|3|4|5|6|7|8|9)

dígito dígito* = $dígito^+$

- Cualquier caracter metasímbolo “.” ∑=(a,b,c) E.R. Todas las cadenas que tengan al menos una b

(alc)* b (alc)* = .* b .*

- Subexpresiones opcionales. Metasímbolo “?” Número entero con o sin signo.

(+l-lε) digito^{+} = (+l-)? dígito^{+}

Los que ocupamos:

identificadores, palabras reservadas, operadores, números enteros.

números reales con o sin signo.

dígito = (0l1l2l3l4l5l6l7l8l9)

(+l-)? dígito^{+} “.” dígito*

Comentarios (una línea o múltiples líneas).

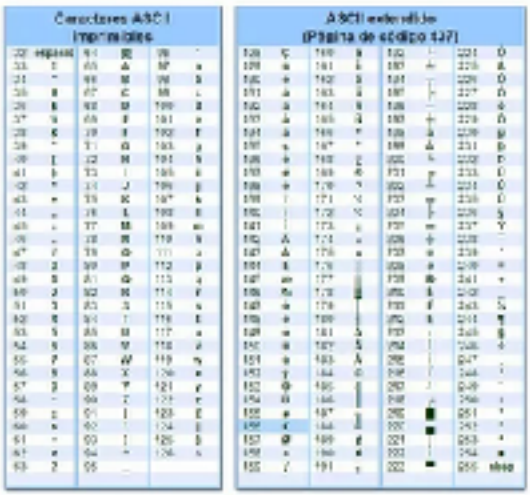
caracteres = (alt+32l…lalt+254)

/ / caracteres*

/ “ * ” caracteres* “ * ” /

Cadenas.

“ caracteres* “



Ambigüedad, espacios en blanco y búsqueda hacia adelante

A menudo en la descripción de los tokens de lenguajes de programación utilizando expresiones regulares, algunas cadenas se pueden definir mediante varias expresiones diferentes, por ejemplo, cadenas tales como ***if*** y ***while*** podrían ser identificadores o palabras reservadas. De manera semejante la cadena ***++*** podría interpretarse como la representación de 2 tokens de adición o como un token simple de incremento. Una definición de lenguaje de programación debe establecer cual interpretación es la correcta, y las expresiones regulares por si mismas no pueden resolver eso. Así, una definición de lenguaje debe proporcionar reglas de no ambigüedad que implicarán cuál significado es el correcto para cada uno de los casos.

Dos reglas típicas para resolver estos problemas son:

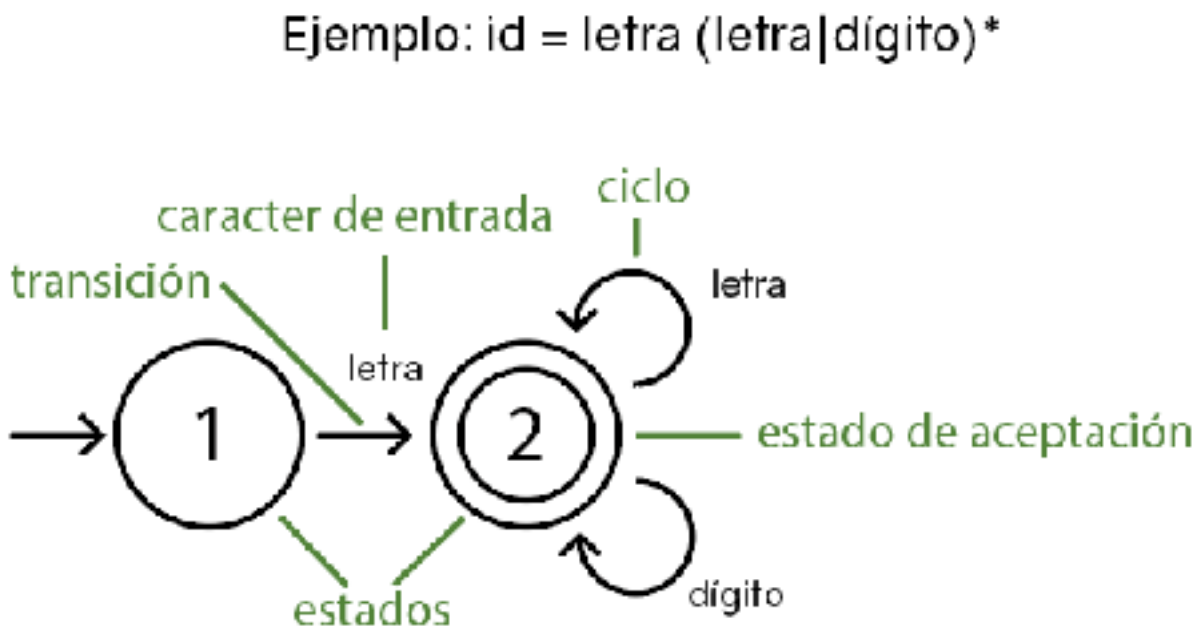
- La primera establece que, cuando una cadena puede ser un identificador o una palabra reservada, se prefiere la interpretación como palabra reservada.
- La segunda establece que, cuando una cadena puede ser un token simple o una secuencia de varios tokens, por lo regular se prefiere la interpretación de un token simple. Esta preferencia se conoce como el *principio de la subcadena más larga*.

Reconocimiento de Componentes Léxicos

Autómatas de Estado Finito

Los autómatas finitos, o máquinas de estado finito, son una manera matemática para describir clases particulares de algoritmos. En particular, los autómatas finitos se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo, se pueden utilizar para construir analizadores léxicos.

Una máquina de estado finito o autómata finito, es un modelo computacional que consiste de un conjunto de estados, un estado de inicio, un alfabeto de entrada y una función de transición que traza un mapa a un siguiente estado a partir del símbolo de entrada y el estado actual.



Autómatas Finitos Deterministico (DFA)

Un autómata finito determinista (DFA), es un modelo, donde el siguiente estado de la transición está dado particularmente por el estado actual y el carácter de entrada. Si ningún estado de transición es especificado, la cadena entrante es rechazado.

Formalente dun DFA se define como:

$$DFA (S , \sum , T , s , A)$$

donde:

S: Es un conjunto finito no vacío de elementos llamados estados.

∑: Alfabeto de entrada.

T: Función de transición.

s: s ε S estado inicial.

A: A ⊆S conjunto **NO** vacío de estados finales.

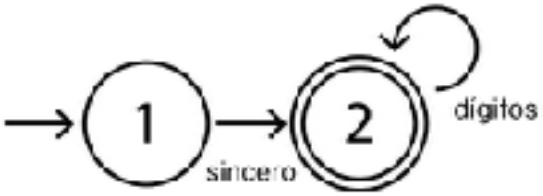
El DFA se encuentra en un estado inicial y lee una entrada de cadena de caracteres de izquierda a derecha. La función de transición T define los estados de transición y es denotada por $T (S_i , C) = S_{i+1}$ donde S_i y S_{i+1} son estados de S y C es un carácter del alfabeto de entrada. La función de transición indica que cuando un autómata está en un estado S_i y recibe el siguiente símbolo de entrada C, el autómata deberá de cambiar al estado S_{i+1}. El carácter de entrada nunca usará transición en caso de ser un carácter vacío.

dígitos = (0|1|2|3|4|5|6|7|8|9)

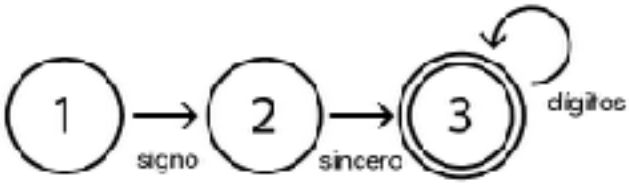
sincero = (1|2|3|4|5|6|7|8|9)

signo = (+|-|ε)

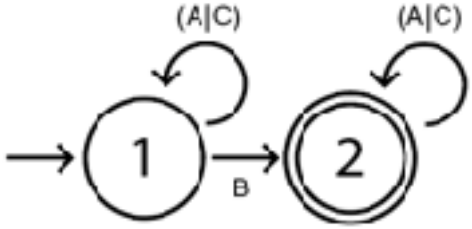
- Hacer un autómata para números enteros que no inicien con cero.



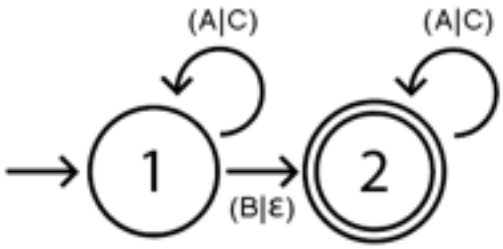
- Para números enteros con o sin signo.



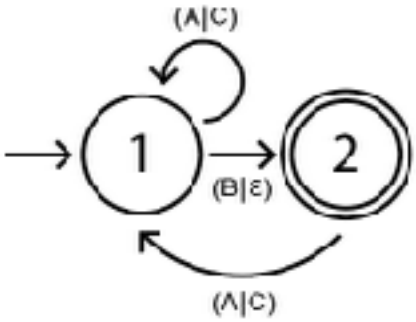
- Para todas las cadenas de A, B, y C que contengan exactamente una B.



- Para todas las cadenas de A, B y C que tengan como máximo una B.



- Para todas las cadenas de A, B y C que no tengan 2 B consecutivas.



- Para todas las cadenas de A, B y C que contengan un número par de A.

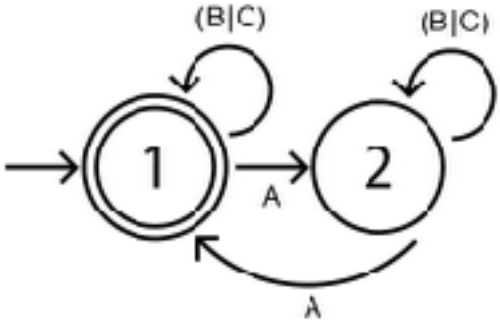
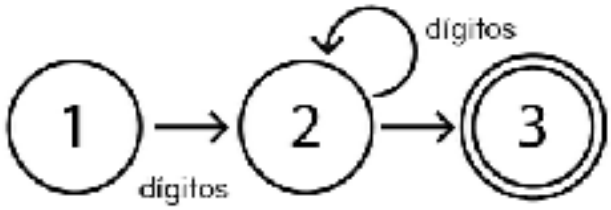


Tabla de Transición

ENTRADA > ESTADO V	dígitos	otro	aceptación
1	2	-	-
2	2	3	-
3	-	-	acepta cadena



Algoritmo de

A continuación dos maneras de realizar el mismo algoritmo.

```
{ inicio estado 1 }
if ( caracter == letra )
     avanzaEntrada; //concatena a lexema.
     { estado = 2 }
     while( caracter==letra ll caracter==dígito)
          avanzaEntrada;
     fin while
     {ir al estado 3 SIN consumir el caracter}
     aceptarCadena; //categorizo token y
                          //almaceno lexema
else
     {error u otros casos}
end
```

```
estado := 1; { inicio }
while estado = 1 o 2 do
    case estado of
    1:  case carácter de entrada of
        letra: avanza en la entrada;
                  estado := 2;
        else estado := … {error u otro};
        end case;
    2:  case carácter de entrada of
        letra dígito: avanza en la entrada;
                  estado := 2; {realmente innecesario}
        else estado := 3;
        end case;
    end case;
end while;
if estado = 3 then aceptar else error;
```

Compilador Tiny

Lenguaje y compilador de muestra. No tipado con variables enteras.

Tiene una estructura muy simple: es simplemente una secuencia de sentencias separadas pedante signos de punto y coma en una sintaxis semejante a la de Ada o Pascal.

Sentencias de control: if else end, repeat until.

Sentencias de lectura y escritura: read, write, {comentarios}.

Expresiones: aritméticas enteras, booleanas (< = comparación) .

Operadores: + - * /.

Palabras Reservadas	Símbolos Especiales	Otros
if	+	número (1 o más dígitos)
then	-	identificador (1 o más letras)
else	*	
end	/	
repeat	=	
until	<	
read	(	

Palabras Reservadas	Símbolos Especiales	Otros
write	)	
	;	
	:=	

Unidad 3. Análisis Sintáctico.

El análisis gramatical es la tarea de determinar la sintaxis del programa por esa razón también se le conoce como análisis sintáctico. La sintaxis de un lenguaje de programación se determina mediante las reglas gramaticales de una gramática libre de contexto, de una manera similar como se determina mediante expresiones regulares la estructura léxica.

Una gramática libre de contexto utiliza convenciones para nombrar y operaciones muy similares a las expresiones regulares con la diferencia de que las reglas gramaticales son recursivas.

Las estructuras de datos utilizadas para representar la estructura sintáctica de un lenguaje deben ser re cursivas en lugar de lineales.

Existen dos categorías de análisis sintáctico: ascendente y descendente, esto se debe a la manera en la que reconstruyen el árbol de análisis gramatical o árbol sintáctico.

La tarea pues del analizador sintáctico es determinar la estructura sintáctica de un programa a partir de los tokens producidos por el analizador léxico y ya sea de manera explícita o implícita construir un árbol de análisis gramatical o árbol sintáctico de ésta manera se puede ver al analizador sintáctico como una función que toma como entrada los tokens y produce un árbol de análisis sintáctico.

La estructura del árbol sintáctico depende en gran medida de la estructura sintáctica particular del lenguaje. el árbol resultante de este análisis por lo regular se define como una estructura de datos dinámica, en la cual cada nodo se compone de un registro cuyos campos incluyen los atributos necesarios para el proceso de compilación.

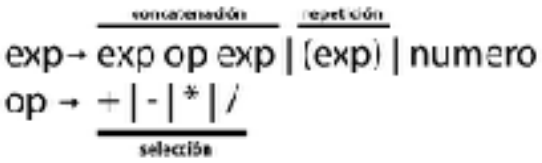
Un problema más difícil de resolver para el analizador sintáctico que para el léxico es el tratamiento de errores.

En el analizador léxico si hay un carácter que no puede ser parte de un token legal, es suficiente con generar un token de error y consumir el carácter en problema. Por otra parte el analizador sintáctico no solo debe mostrar un mensaje de error, sino que debe recuperarse del error y continuar con el análisis sintáctico. En ocaciones, un analizador sintáctico puede efectuar reparación de errores, en la cual infiera una posible versión de código corregida a partir de la versión incorrecta. Un aspecto particularmente importante de la recuperación de errores es la exhibición de mensajes de errores significativos y la reanudación del análisis sintáctico tan próximo al error como sea posible.

Gramáticas libres de contexto.

Una gramática libre de contexto es una especificación para la estructura sintáctica de un lenguaje de programación. Una especificación así es muy similar a la especificación de la estructura léxica de un lenguaje utilizando expresiones regulares, excepto que una gramática libre de contexto involucra reglas de recursividad.

Ejemplo: gramática para expresiones aritméticas simples.



Las reglas gramaticales utilizan notaciones similares a las expresiones regulares. La barra vertical " | " todavía aparece como el metasímbolo para selección. La concatenación también se utiliza como operación estándar y no existe ningún símbolo para la repetición ya que esta se obtiene a través de reglas recursivas. Las reglas gramaticales se usaron por primera vez en la descripción del lenguaje *Algol60.* La notación fue desarrollada por *John Backus* y adaptada por *Peter Naur.* De este modo, generalmente se dice que las reglas gramaticales en esta forma están en la forma *Backus-Naur* o *BNF*.

Dado un alfabeto, una regla gramatical libre de contexto en *BNF* se compone de una cadena de símbolos. El primer símbolo es un nombre para una estructura. El segundo símbolo, el metasímbolo "→". Este símbolo está seguido por una cadena de símbolos, cada uno de los cuales es un símbolo del alfabeto, un nombre para una estructura o el metasímbolo pipe " | ".

exp → exp op exp | (exp) | numero

op → + | - | * | /

No terminales: exp op Terminal: TOKEN

Derivación y Lenguaje definido por una gramática

Las reglas gramaticales libres de contexto determinan el conjunto de cadenas sintácticamente legales de símbolos de token para las estructuras definidas por la regla. Éstas cadenas legales de símbolos se determinan por medio de derivaciones. Una derivación es una secuencia de reemplazo de nombres de estructura por selecciones en los lados derechos de las reglas gramaticales. Una derivación comienza con un nombre de estructura simple y termina con una cadena de símbolos de token. En cada etapa de una derivación se hace un reemplazo simple utilizando una selección de una regla gramatical.

(34 - 3) * 42

exp → exp op exp

→ (exp) op exp

→ (exp op exp) op exp

→ (numero op exp) op exp

→ (34 - exp) op exp

→ (34 - numero) op exp

→ (34 - 3) * exp

→ (34 - 3) * numero

→ (34 - 3) * 42

(((34 - 3) * 42) / 7) * 2

exp → exp op exp

→ (exp) op exp

→ (exp op exp) op exp

→ ((exp) op exp) op exp

→ ((exp op exp) op exp) op exp

→ (((exp) op exp) op exp) op exp

→ (((exp op exp) op exp) op exp) op exp

→ (((numero op numero) op numero) op numero) op numero

→ (((numero - numero) op numero) op numero) op numero

→ (((numero - numero) * numero) op numero) op numero

→ (((numero - numero) * numero) / numero) op numero

→ (((numero - numero) * numero) / numero) * numero

→ (((34 - 3) * 42) / 7) * 2

3 + 7 - 32 * 25 / 4

exp → exp op exp

→ exp op exp op exp

→ exp op exp op exp op exp

→ exp op exp op exp op exp op exp

→ numero op exp op exp op exp op exp

→ numero op numero op exp op exp op exp

→ numero op numero op numero op exp op exp

→ numero op numero op numero op numero op exp

→ numero op numero op numero op numero op numero

→ numero + numero op numero op numero op numero

→ numero + numero - numero op numero op numero

→ numero + numero - numero * numero op numero

→ numero + numero - numero * numero / numero

→ 3 + 7 - 32 * 25 / 4

32 - 3 * 42

exp → exp op exp

→ numero - exp

→ numero - exp op exp

→ numero - numero op ex

→ numero - numero * exp

→ numero - numero * numero

exp → exp op exp

→ exp op numero

→ exp * 42

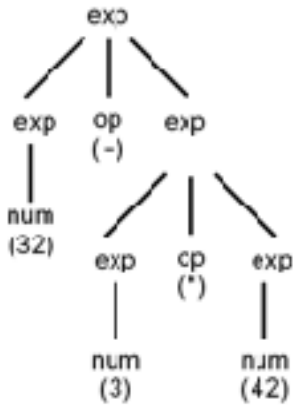
→ exp op numero * 42

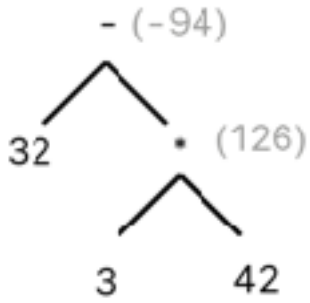
→ exp - 3 * 42

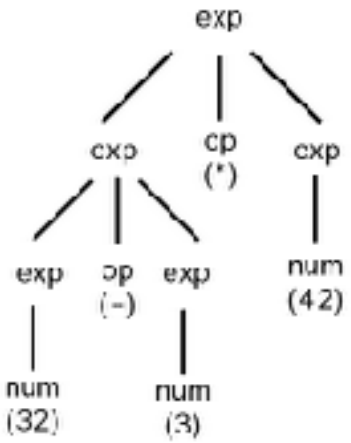
→ numero - 3 * 42

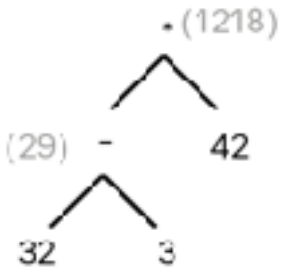
→ 32 - 3 * 42

ÁRBOL DE ANÁLISIS GRAMATICAL	ÁRBOL SINTÁCTICO







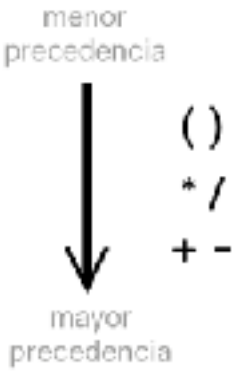


Gramáticas Ambiguas

Los árbol de análisis gramatical y los árboles sintácticos expresan de manera única la estructura de la sintaxis. Una gramática que genera una cadena con dos árboles de análisis gramatical distintos se denomina gramática ambigua. Una gramática de ésta clase representa un serio problema para un analizador sintáctico, ya que no especifica con precisión la estructura sintáctica de un programa.

Una gramática ambigua debe por lo tanto, considerarse como una especificación incompleta de la sintaxis de un lenguaje, y como tal deberían evitarse.

Para tratar con las ambigüedades se utilizan dos métodos básicos. Uno consiste en establecer una regla que especifique en cada caso ambiguo cuál de los árboles de análisis gramatical es el correcto. Una regla de esta clase se conoce como regla de no ambigüedad o eliminación de ambigüedades. La segunda alternativa es cambiar la gramática a una forma que obligue a construir el árbol de análisis gramatical correcto, de tal manera que se elimine la ambigüedad.



exp → exp ops exp l termino

ops → + l -

termino → termino opm termino l factor

opm → * l /

factor → (exp) l numero

3 + 4 * 5

exp → exp ops exp

→ termino ops exp

→ factor ops exp

→ numero ops exp

→ numero + exp

→ numero + termino

→ numero + termino opm termino

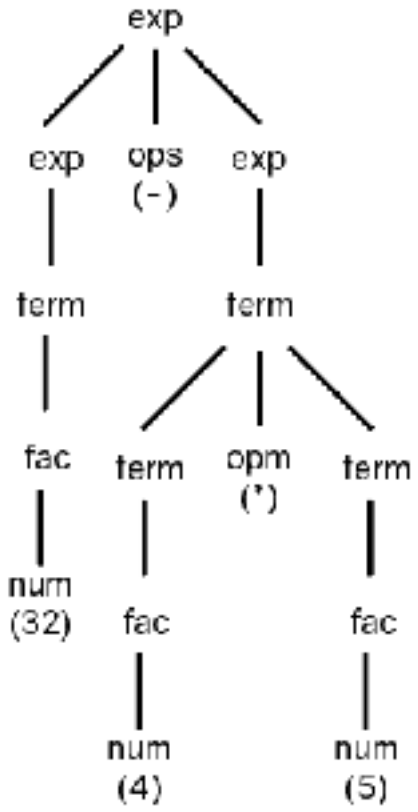
→ numero + factor opm termino

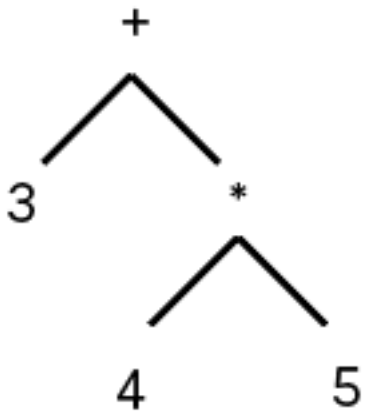
→ numero + numero opm termino

→ numero + numero * termino

→ numero + numero * factor

→ numero + numero * numero





3 - 4 - 5

exp → exp ops exp

→ exp ops exp ops exp

→ termino ops exp ops exp

→ factor ops exp ops exp

→ numero ops exp ops exp

→ numero - exp ops exp

→ numero - termino ops exp

→ numero - factor ops exp

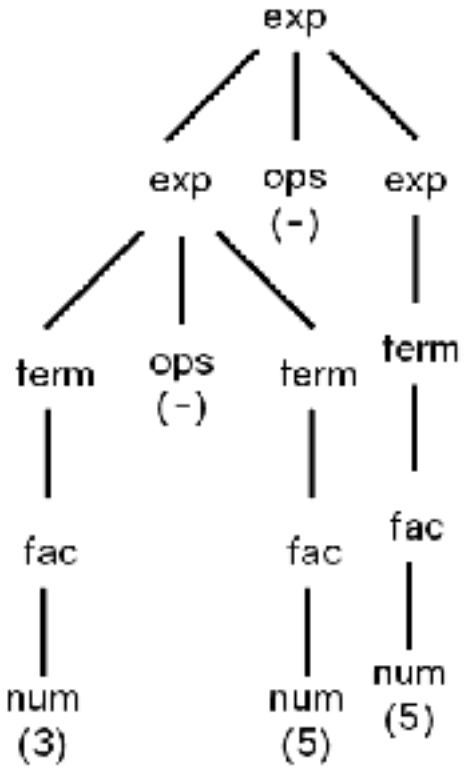
→ numero - numero ops exp

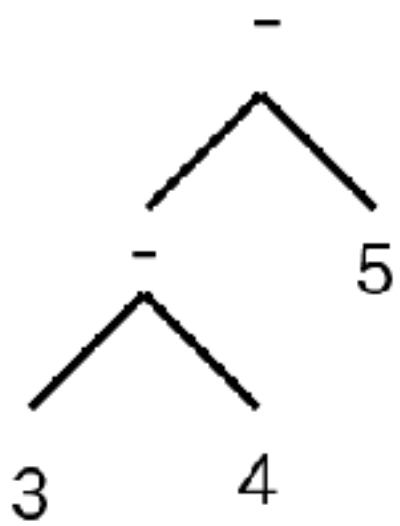
→ numero - numero - exp

→ numero - numero - termino

→ numero - numero - factor

→numero - numero - numero





34 - 3 - 42

exp → exp ops exp

→ termino ops exp

→ factor ops exp

→ numero ops exp

→ numero - exp

→ numero - exp ops exp

→ numero - termino ops exp

→ numero - factor ops exp

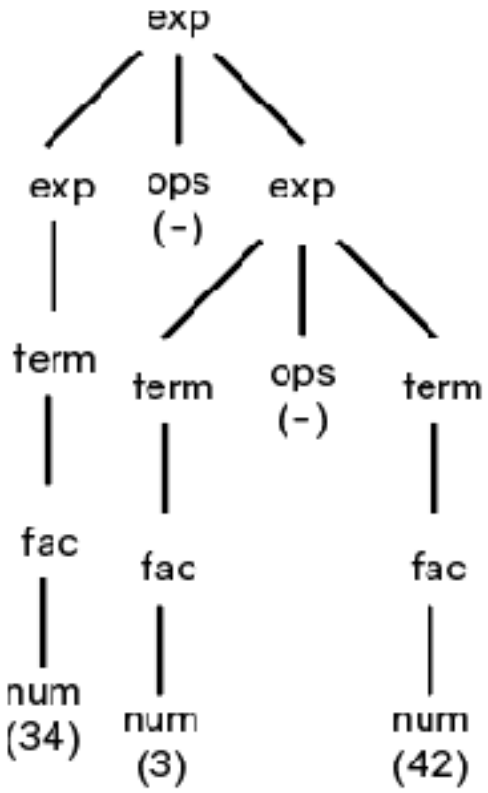
→ numero - numero ops exp

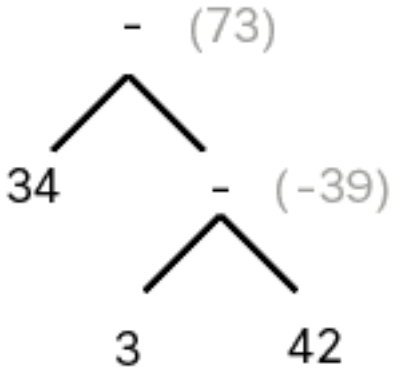
→ numero - numero - exp

→ numero - numero - termino

→ numero - numero - factor

→numero - numero - numero





exp → exp ops exp

→ exp ops termino

→ exp ops factor

→ exp ops numero

→ exp - numero

→ exp ops exp - numero

→ exp ops termino - numero

→ exp ops factor - numero

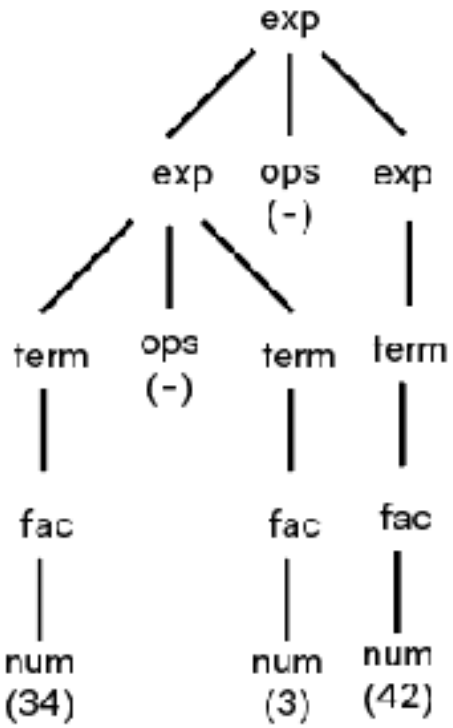
→ exp ops numero- numero

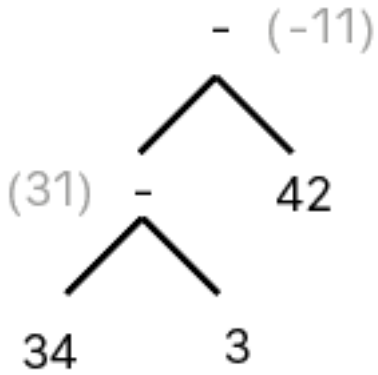
→ exp - numero - numero

→ termino - numero - numero

→ factor - numero - numero

→ numero - numero - numero





***gramática ambigua por asociatividad**

exp → exp ops termino | termino

ops → + | -

termino → termino opm factor | factor

opm → * | /

factor → (exp) | numero

34 - 3 - 42

exp → exp ops termino

→ termino ops termino

→ factor ops termino

→ numero ops termino

→ numero - termino

→ numero - factor

→ numero - (exp)

→ numero - (exp ops termino)

→ numero - (termino ops termino)

→ numero - (factor ops termino)

→ numero - (numero ops termino)

→ numero - (numero - termino)

→ numero - (numero - factor)

→ numero - (numero - (exp))

→ numero - (numero - (termino))

→ numero - (numero - (factor))

→ numero - (numero - (numero))

Gramáticas EBNF

Las construcciones repetitivas y opcionales son muy comunes en los lenguajes de programación, lo mismo que las reglas gramaticales BNF. Por lo tanto, es común que la notación BNF se extienda en ocasiones con el fin de incluir notaciones especiales para estas dos situaciones. Estas extensiones comprenden una notación que se denomina BNF EXTENDIDA o EBNF.

Ejemplo:

A → A α I ß

ß

ß α

ß α α

ß α α α

BNF → ß α *

EBNF → ß {a} // { } ciclo EBNF

A → α A I ß

ß

α ß

α α ß

α α α ß

BNF → α * ß

EBNF → {a} ß

SecSen → sen; SecSen I sen

sen → s

EBNF → {sen;} sen

exp → exp ops term | term

ops → + | -

term → term opm fac | fac

opm → * | /

fac → (exp) | num

exp → term { ops term }

ops → + | -

term → fac { opm fac }

opm → * | /

fac → (exp) | num

EBNF → { { fac opm } fac ops } { fac opm } fac

EBNF → { ({ { fac opm } fac ops } { fac opm } fac) } { { { fac opm } fac ops } { fac opm } fac }

// [ ] opcional EBNF

TERMINO

```
procedure factor :
begin
        case token of
        (:
                match(();
                term;
                match());
        :

        else error;
        end case;
end factor;

procedure match(expectedToken) :
end match ;

procedure term :
begin
        case token of
        term:
                term;
                match(*|/);
                factor;
        factor:
                factor;
        else error;
        end case;
end term;


procedure exp :
begin
        case token of
        exp:
                exp;
                match(+|-);
                term;
        term:
                term;
        else error;
        end case;
end exp;
```

Compiladores 2

Tabla de Símbolos

La tabla de símbolos es el principal atributo heredado en un compilador, y después del árbol sintáctico también forma la principal estructura de datos. En los compiladores prácticos, la tabla de símbolos con frecuencia, está íntimamente involucrada con el analizador sintáctico e incluso con el analizador léxico, cualquiera de los cuales pueden necesitar introducir información de manera directa en la tabla de símbolos o consultarla para resolver ambigüedades. No obstante, en lenguajes diseñados muy cuidadosamente, es posible e incluso razonable postponer las operaciones de la tabla de símbolos hasta después de realizar un análisis sintáctico completo, cuando se sepa que el programa que se está traduciendo es sintácticamente correcto. Las principales operaciones de la tabla de símbolos son inserción, búsqueda y eliminación. La operación de **inserción** se utiliza para almacenar la información proporcionada por las declaraciones. La operación de **búsqueda** es necesaria para recuperar información asociada a un nombre cuando este es utilizado en algún código. La operación de **eliminación** es necesaria para eliminar la información proporcionada por una declaración que ya no aplica. Las propiedades de estas operaciones son dictadas por las reglas del lenguaje de programación que se esté traduciendo. La información que se necesita almacenar en la tabla de símbolos está en función de la estructura y propósito de las declaraciones. Incluye información del tipo de datos, ámbito de las variables e información acerca de la ubicación posible de la memoria.

Estructura de la Tabla de Símbolos

La tabla de símbolos en un compilador es una típica estructura de datos tipo diccionario. La eficiencia de las tres operaciones básicas de inserción, búsqueda y eliminación, varía de acuerdo con la organización de la estructura de datos.

Las implementaciones típicas de las tablas de símbolos incluye listas ligadas, diversas estructuras de árbol de búsqueda (árboles binarios, árboles ABL y árboles B) así como tablas de dispersión.

Ejemplo:

```
read x
if x > 0 then
        fact = 1
        repeat
```

```
            fact = fact * x
            x = x - 1
        until x = 0
    write fact
```

Nombre Variable	Localidad (#Registro)	Número Línea	Valor	Tipo
x	0	1, 2, 5, 6, 6, 7	-	int
fact	1	3, 5, 5, 8	1	int

Tanto las listas lineales como las estructuras de árbol son mecanismos eficientes para implementar tablas de símbolos, aunque las tablas de dispersión siempre resultan ser la mejor solución.

Una tabla de dispersión es un arreglo de entradas que indizadas mediante un intervalo entero, generalmente con valores de 0 a n-1. Para el manejo de los datos dentro de la tabla de dispersión, es necesario una función de dispersión (hash) que se encarga de convertir la clave de búsqueda (nombre del identificador) en un valor entero de dispersión en el intervalo del índice y el elemento que corresponde a la clave.

Se debe tener claro que la función de dispersión es un mecanismo de manipulación de índices, por lo que debe garantizar un mínimo de colisiones, ya que ella se encarga de distribuir los valores de manera uniforme. Permitiendo que las operaciones de búsqueda, inserción y eliminación, sean lo más eficiente posible.

Código Intermedio

La tarea final de un compilador, es generar código ejecutable para una máquina objeto que sea una fiel representación de la semántica del código fuente. La generación de código es la fase más compleja de un compilador, puesto que involucra información detallada acerca de la arquitectura objetivo, la estructura del ambiente de ejecución y el sistema operativo que esté corriendo en la máquina objetivo. La generación de código por lo regular también realiza algún intento de optimización, mejora la velocidad y/o tamaño del código objetivo.

Debido a la complejidad de la generación de código, un compilador por lo regular divide esta fase en varios pasos, los cuales involucran varias estructuras de datos intermedias, y a menudo incluyen alguna forma de código abstracto, denominado código intermedio. Un compilador puede detener la generación de código ejecutable real y en su lugar generar alguna forma de código ensamblador que deba ser procesado adicionalmente

por un ensamblador, un ligador y un cargador, los cuales pueden ser proporcionados por el sistema operativo, o compactados en el compilador.

Código Intermedio y Estructuras de Datos para la Generación de Código

Una estructura de datos que represente el programa fuente, durante la traducción se denomina representación intermedia.

En nuestro compilador, hasta ahora se ha usado un árbol sintáctico como la principal estructura de datos para la traducción.

El código intermedio puede tomar muchas formas, existen casi tantos estilos diferentes como códigos intermedios en los compiladores. Sin embargo, todos representan alguna forma de linealización del árbol sintáctico, es decir, una representación del árbol sintáctico de forma secuencial. El código intermedio puede ser de muy alto nivel y representar todas las operaciones de manera casi tan abstracta como la de un árbol sintáctico, o parecerse mucho al código objetivo. Puede utilizar o no información detallada acerca de la máquina objetivo y el ambiente de ejecución. Puede o no incorporar toda la información contendida en la tabla de símbolos, tal como los ámbitos, los niveles de admiración y los desplazamientos de las variables. Si lo hace, entonces la generación de código objeto puede basarse sólo en el código intermedio; si no, el compilador debe retener la tabla de símbolos para la generación de código objeto.

Existen dos formas populares de código intermedio, código de tres direcciones y código p.

Código de 3 direcciones.

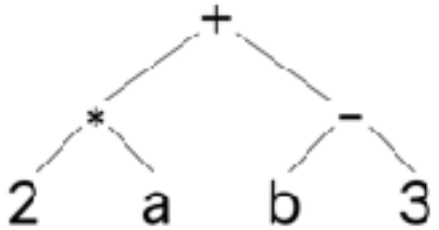
Las instrucciones básicas de código de tres direcciones están diseñadas para representar la evaluación de expresiones aritméticas que tienen la siguiente forma general X = Y op Z

El nombre, del código de tres direcciones viene de esta forma de instrucción, ya que cada una de las variables representan una dirección en la memoria.

Ejemplo: Representar en código de 3 direcciones la expresión siguiente:

2 * a + (b - 3)

1. Construir el árbol sintáctico.



2. Código de tres direcciones

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

El código de tres direcciones requiere que el compilador genere nombres para elementos temporales. Estos elementos temporales corresponden a los nodos interiores del árbol sintáctico y representar sus valores calculados con el último elemento temporal representando el valor de la raíz. La manera en la que estos elementos temporales son asignados a la memoria no se especifica por el código de tres direcciones, por lo regular, los elementos temporales serán asignados a registros, pero también se pueden conservar en registros de activación. El código de tres direcciones regularmente representa una linealización de izquierda a derecha del árbol sintáctico, pero es posible que un compilador necesite un orden diferente en algunas circunstancias.

Ejemplo: Considere el siguiente programa escrito en tiny y obtenga el código de 3 direcciones que lo representa:

```
read x;
if 0 < x then
    fact := 1
    repeat
        fact := fact * x;
        x := x-1;
    until x = 0;
    write fact
end
```

código de tres direcciones

```
read x
t1 = x > 0
if_else t1 goto L1
fact = 1
label L2
```

```
t2 = fact * x

fact = t2

t3 = x - 1

x = t3

t4 = x == 0

if_false t4 goto L2

write fact

label L1

halt
```

Estructuras de Datos para la implementación de código de tres direcciones

El código de tres direcciones por lo regular no está implementado en forma textual. En lugar de esto, cada instrucción de tres direcciones está implementada como una estructura de registros que contiene varios datos, y la secuencia completa de instrucciones está implementada como un arreglo o lista enlazada, la cual se debe conservar en memoria o en archivos temporales.

La implementación más común consiste en almacenar el código de tres direcciones en un registro con cuatro campos; uno para la operación y tres para las direcciones. Para las instrucciones que necesitan menos de tres direcciones, uno o más de los campos de dirección proporcionan un valor nulo o vacío.

```
(rd , x , _ , _ )

( gt , x , 0 , t )

( if_f , t1 , L1 , _ )

( asn , 1 , fact , _ )

( lab , L2 , _ , _ )

( mul , fact , x , t2 )

( asn , t2 , fact , _ )

( sub , x , 1 , t3 )

( asn , t3 , x , _ )

( eq , x1 , 0 , t4 )
```

```
( if_f , t4 , L2 , _ )
( wri , fact , _ , _ )
( lab , L1 , _ , _ )
( halt , _ , _ , _ )

typedef enum { rd , gt , if_f , asn , lab , mul , sub , eq , wri , halt , … }
        opkind;
typedef enum { Empty , IntConst , String } Addr kind;
typedef struct {
        Addrkind kind;
        union {
                int val;
                char *name;
        } contents;
} Address;
typedef struct {
        opkind op;
        Address addr1, addr2, addr3;
} Quad;
```

Código P

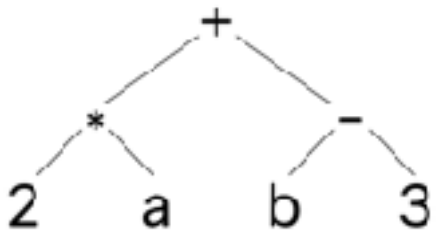
El código P comenzó como un código ensamblador objetivo estándar producido por varios compiladores de pascal en la década de los 70's y principios de los 80's. Fue diseñado para hacer el código real de una máquina de pila hipotética, denominada máquina P, para la que fue escrito un intérprete en varias máquinas reales. La idea era hacer que los compiladores de pascal se transportarán fácilmente requiriendo solo que se volviera a escribir el intérprete de la máquina P para una nueva plataforma. El código P también ha probado ser útil como código intermedio, y se han utilizado varias extensiones y modificaciones del mismo en diversos compiladores del código nativo.

El código P fue diseñado para ser directamente ejecutado, contiene una descripción implícita de un ambiente de ejecución particular que incluye tamaños de datos a demás de mucha información específica para la máquina P.

Como ejemplo para el compilador Tiny la máqiuna P está compuesta por una **memoria de código**, una **memoria de datos no especificados** para variables nombradas, y una **pila para datos temporales**, junto con cualquier registro que sea necesario para mantener la pila y apoyar la ejecución.

Ejemplo: considere la expresión 2 * a + (b - 3)

1. Construir el árbol sintáctico.



2. Código intermedio

ldc 2;	cargar constante 2
lod a;	cargar el valor de la variable a
mpr;	multiplicación entera
lod b;	cargar el valor de la variale b
ldc 3;	cargar al constante 3
sbi;	resta entera
adi;	adición entera

Ejemplo de un código P

x := y + 1

lda x; código dirección de x

lod y; carga valor de y

ldc 1; carga constante 1

adi; suma

sto; almacena tope a dirección debajo del tope y extrae ambas

Ejemplo

```
read x
if 0 < x then
    fact := 1
    repeat
        fact := fact * x;
        x := x - 1;
    until x = 0;
    write fact
end
```

```
lda x
rdi
lod x
ldc 0
grt
fjp L1
lda fact
ldc 1
sto
lab L2
```

```
lda fact
lod fact
lod x
mpi
sto
lda x
lod x
ldc 1
sbi
sto
```

```
lod x
ldc 0
equ
fjp  L2
lod fact
wri
lab L1
stp
```

Técnicas para la Generación de Código Intermedio

La generación de código intermedio (o generación de código objeto directa sin código intermedio) se puede ver como un cálculo de atributo similar a los atributos de valor o de tipo. Si el código generado se ve como un atributo de cadena, entonces este código se convierte en un atributo sintetizado que se puede definir utilizando una gramática con atributos, y generando directamente el código intermedio durante el análisis semántico, o mediante un recorrido posorden del árbol sintáctico.

Ejemplo: Considere la siguiente gramática que representa un pequeño subconjunto de expresiones en c

exp → id = exp | aexp

aexp → aexp + factor | factor

factor → (exp) | num | id

Pcode = atributo de cadena

++ → concatena instrucciones con saltos de línea

|| → concatena instrucciones sin saltos de linea

Gramática

Regla Gramatical	Regla Semántica
exp1 → id = exp2	exp1.pcode="lda" || id.strval ++ exp2.pcode ++ "sto"
exp → aexp	exp.pcode = aexp.code
aexp1 → aexp2 + factor	aexp1.pcode = aexp2.pcode ++ factor.pcode ++ "adi"
aexp → factor	aexp.pcode = factor.pcode
factor → (exp)	factor.pcode = exp.pcode
factor → num	factor.pcode = "ldc" || num.strval
factor → id	factor.pcode = "lod" || id.strval

(x = x + 3) + 4

exp → aexp

→ aexp + factor

→ factor + factor

→ (exp) + factor

→ (id = exp) + factor

→ (id = aexp) + factor

→(id = aexp + factor) + factor

→(id = factor + factor) + factor

→ (id = id + factor) + factor

→ (id = id + num) + factor

→ (id = id + num) + num

PCODE

pcode [ ]

pcode [ ]

adi /// **completar esta mare**

Gramática Expandida

exp → id = exp | aexp

aexp → aexp op1 multi | multi

op1 → + | -

multi → multi op2 factor | factor

op2 → * | /

factor → (aexp) | num | id

Regla Gramatical	Regla Semántica
exp1 → id = exp2	exp1.pcode=“lda” || id.strval ++ exp2.pcode ++ “sto”
exp → aexp	exp.pcode = aexp.code
aexp1 → aexp2 + multi	aexp1.pcode = aexp2.pcode ++ multi.pcode ++ “adi”
aexp1 → aexp2 - multi	aexp1.pcode = aexp2.pcode ++ multi.pcode ++ “sbi”
aexp → multi	aexp.pcode = multi.pcode
multi1 → multi2 * factor	multi1.pcode = multi2.pcode ++ factor.pcode ++ “mpr”
multi1 → multi2 / factor	multi1.pcode = multi2.pcode ++ factor.pcode ++ “dvi”
multi → factor	multi.pcode = factor.pcode
factor → (exp)	factor.pcode = exp.pcode
factor → num	factor.pcode = “ldc” || num.strval
factor → id	factor.pcode = “lod” || id.strval

Ejemplo: a = x + 3 * 2 + (x - 4)

exp → id = exp

→ id = aexp

→ id = aexp op1 multi

→ id = aexp op1 multi op1 multi

→ id = multi op1 multi op1 multi

→ id = factor op1 multi op1 multi

→ id = id op1 multi op1 multi

→ id = id + multi op1 multi

→ id = id + multi op2 factor op1 multi

→ id = id + factor op2 factor op1 multi

→ id = id + num op2 factor op1 multi

→ id = id + num * factor op1 multi

→ id = id + num * num op1 multi

→ id = id + num * num + multi

→ id = id + num * num + factor

→ id = id + num * num + (aexp)

→ id = id + num * num + (aexp op1 multi)

→ id = id + num * num + (multi op1 multi)

→ id = id + num * num + (factor op1 multi)

→ id = id + num * num + (id op1 multi)

→ id = id + num * num + (id - multi)

→ id = id + num * num + (id - factor)

→ id = id + num * num + (id - num)

código P

lda a

lod x

ldc 3

ldc 2

mpi

adi

lod x

ldc 4

sbi

adi

sto

falta apunte del 31

Generación de código de sentencias de control y expresiones lógicas.

La generación de código intermedio para sentencias de control, tanto en código de tres direcciones como en código P, involucra la generación de etiquetas de una manera similar a la generación de nombres temporales en el código de tres direcciones, pero en este caso representan direcciones en el código objetivo a las que se harán los saltos.

Las expresiones lógicas, o boleanas, que se utilizan como pruebas de control, y que también se pueden emplear de manera independiente como datos se evalúan en corto circuito, en la cual difieren de las expresiones aritméticas.

Ejemplo:

sent-if → if (exp) sent l if (exp) sent else sent

sent-write → while (exp) sent

El problema principal en la generación de código para la sentencias de control es traducir, las características de control estructurado en un equivalente no estructurado que involucre saltos, mismos que se muestran en los siguientes esquemas:

sent → sent_if l sent_while l break l other

sent_if → if (exp) sent l if (exp) sent else sent

sent_while → while (exp) sent

exp → true l false

```
while
label L1
< código para evaluar E >
fjp L2
< código para sentencias >
ujp L1
label L2
```

```
if
< código para evaluar E >
fjp L1
< código para sentencias >
ujp L2
Label L1
< código para sentencias >
Label L2
```

```
void genCode ( SyntaxTree t, char * label) {
        char code codestr[CODESIZE];
        char * lab1, * lab2;
        if ( t != NULL ) switch ( t -> kind ) {
                case ExpKind:
                        if ( t-> val == 0 )
                                emitCode( “ldc false” );
                        else
                                emitCode( “ldc true” );
                        break;
                case IfKind:
                        genCode ( t -> child[0] , label );
                        lab1 = genLabel();
                        sprintf( codestr, ”%s %s” , ”fjp” , lab1 );
                        emitCode( codestr );
                        genCode( t -> child[1] , label );
```

```
            if ( t -> child[1] != NULL ) {
                    lab2 = genLabel();
                    sprintf( codestr , ”%s %s” , ”ujp” , lab2 );
                    emitCode( codestr );
            }
            sprintf( codestr , ”%s %s” , ”lab” , lab1 );
            emitCode( codestr );
            if ( t -> child[2] != NULL ) {
                    genCode ( t -> child[2] , label );
                    sprinf( codestr , ”%s %s” , ”lab” , lab2 );
                    emitCode( codestr );
            }
            break;
    case WhileKind:
            lab1 = genLabel();
            sprinf( codestr , ”%s %s” , ”lab” , lab1 );
            emitCode( codestr );
            genCode( t -> child[0] , label );
            lab2 = genLabel();
            sprinf( codestr , ”%s %s” , ”fjp” , lab2 );
            emitCode( codestr );
            genCode( t -> child[1] , lab2 );
            sprinf( codestr , ”%s %s” , ”ujp” , lab1 );
            emitCode( codestr );
            sprinf( codestr , ”%s %s” , ”lab” , lab2 );
            emitCode( codestr );
            emitCode( codestr );
            break;
```

```
                case BreakKind:
                        sprinf( codestr , ”%s %s” , ”ujp” , label );
                        emitCode( codestr );
                        break;
                case OtherKind:
                        emitCode( “Other” );
                        break;
                default:
                        emitCode( “Error” );
                        break;
                }
        }
}
```

código de 3 direcciones apéndice A del libro // reimplementar esa máquina