



Instituto tecnológico de Iztapalapa

Ingeniería en Sistemas Computacionales

Lenguajes y automatas 2

Santana González Jesús Salvador: 171080127

Cabrera Ramírez Gerardo: 171080187

Morales Carrillo Gerardo: 171080120

Actividad semana 14

16/10/2020

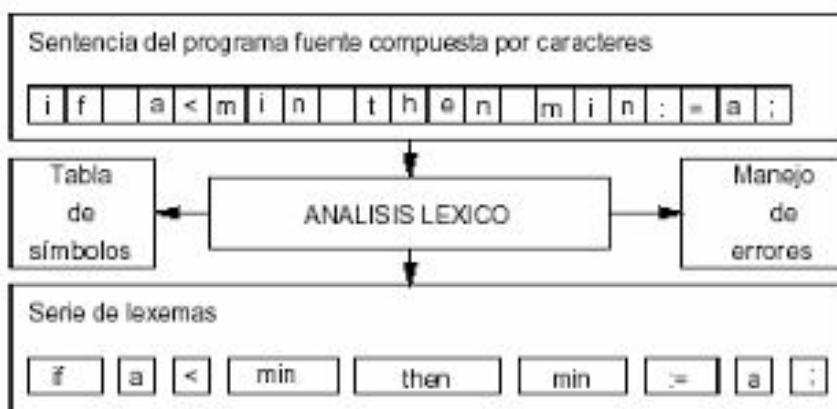


"Análisis Léxico parte 1"

Conceptos:

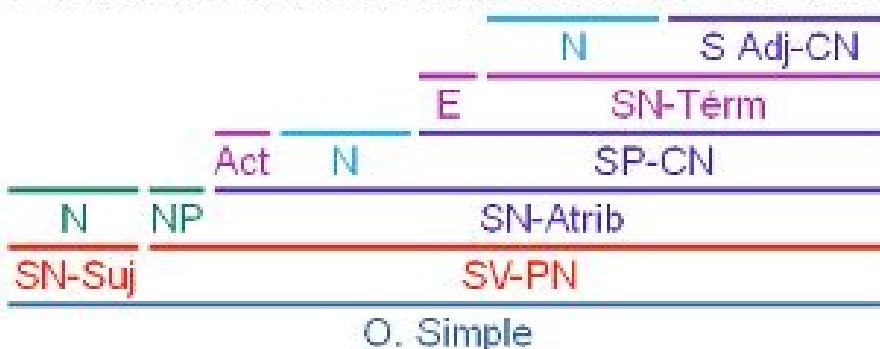
Análisis de léxico

A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias.

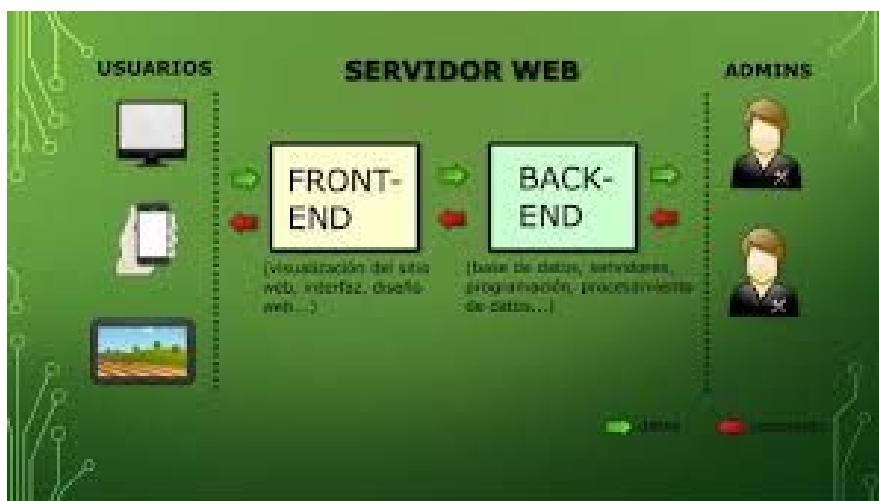


Análisis sintáctico La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens.

EdAS es un editor de análisis sintáctico



Front End: El Front End es la parte del compilador que interactúa con el usuario y por lo general, es independiente de la plataforma en la que se trabaja.



Back End: Esta parte del compilador es la encargada de generar el código en formato de máquina, a partir del trabajo hecho por el Front End.

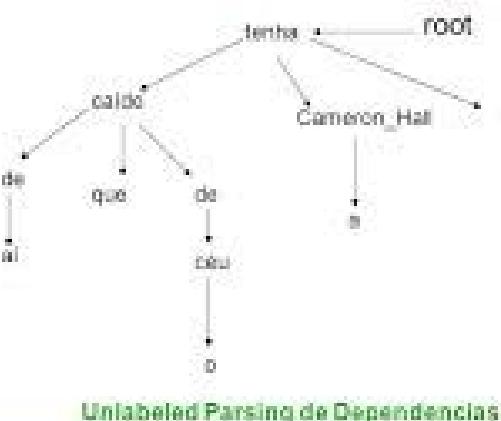


Gráficos de dependencias Un grafo de dependencias describe el flujo de información entre las instancias de atributos en un árbol de análisis sintáctico específico; una flecha de una instancia de atributo a otra significa que el valor de la primera se necesita para calcular la segunda.



Ejemplo de Grafo de dependencias (1)

| | |
|----------------|----|
| 1De | 7 |
| 2 al | 1 |
| 3 qué | 7 |
| 4 a | 5 |
| 5 Cameron_Hall | 6 |
| 6 tenta | 0 |
| 7 caldo | 6 |
| 8 -de | 7 |
| 9 o | 10 |
| 10 céu | 8 |
| 11 - | 6 |



Unlabeled Parsing de Dependencias

Gestión de información de errores.

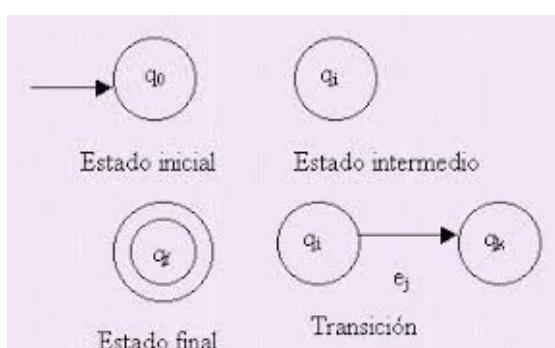
Si los compiladores tuvieran que procesar solamente programas correctos, su diseño e implementación se simplificaría en buena medida. Pero los programadores escriben programas incorrectos frecuentemente, y un buen compilador debe ayudar al programador a localizar e identificar los errores.

Generación del código objeto.

La fase final de un compilador es la de generación del código objeto, consistente en código máquina o código ensamblador.

Representación de lenguajes:

En general existen dos esquemas diferentes para definir un Lenguaje, los cuales se conocen como esquema generador y esquema reconocedor. En el caso de los esquemas generadores se trata de un mecanismo que nos permite “generar” las diferentes sentencias del lenguaje



Concepto de gramática:

La gramática es un ente o modelo matemático que permite especificar un lenguaje, es decir, es el conjunto de reglas capaces de generar todas las posibilidades combinatorias de ese lenguaje, y sólo las de dicho lenguaje, ya sea éste un lenguaje formal o un lenguaje natural.

Santana Gonzales Jesús Salvador

"Análisis Léxico (partes 2 y 3)" y "Análisis Sintáctico (parte 1)"

Funciones del analizador léxico y sus ventajas

El analizador léxico realiza varias funciones, siendo la fundamental la de agrupar los caracteres que va leyendo uno a uno del programa fuente y formar los tokens.

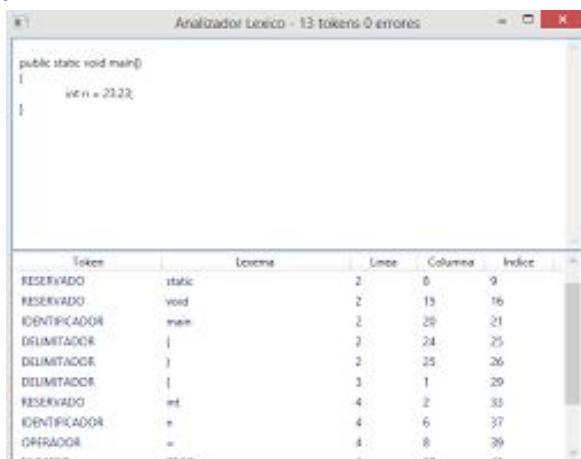
Implementación de un analizador léxico Hay varias formas de implementar un analizador léxico:

Utilizando un generador de analizadores léxicos: son herramientas que a partir de las expresiones regulares generan un programa que permite reconocer los tokens o componentes léxicos. Estos programas suelen estar escritos en C, donde una de las herramientas es FLEX, o pueden estar escritos en Java, donde las herramientas pueden ser JFLEX o JLEX.

Utilizando un lenguaje de alto nivel: a partir del diagrama de transiciones y del pseudocódigo correspondiente se programa el analizador léxico (véase un ejemplo en Louden, 2004).

Errores léxicos

Los errores léxicos son detectados, cuando durante el proceso de reconocimiento de caracteres, los símbolos que tenemos en la entrada no concuerdan con ningún patrón.



Analizador Léxico - 13 tokens 0 errores

```
public static void main()
{
    int n = 23.23;
}
```

| TOKEN | TEXTO | LÍNEA | COLUMNA | ÍNDICE |
|---------------|--------|-------|---------|--------|
| RESERVADO | static | 2 | 0 | 9 |
| RESERVADO | void | 2 | 15 | 16 |
| IDENTIFICADOR | main | 2 | 20 | 21 |
| DELIMITADOR | { | 2 | 24 | 25 |
| DELIMITADOR | } | 2 | 25 | 26 |
| DELIMITADOR | , | 3 | 1 | 29 |
| RESERVADO | int | 4 | 2 | 33 |
| IDENTIFICADOR | n | 4 | 6 | 37 |
| OPERADOR | = | 4 | 8 | 39 |
| OTRO | | 5 | 18 | 40 |

¿Qué es el analizador sintáctico ?

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

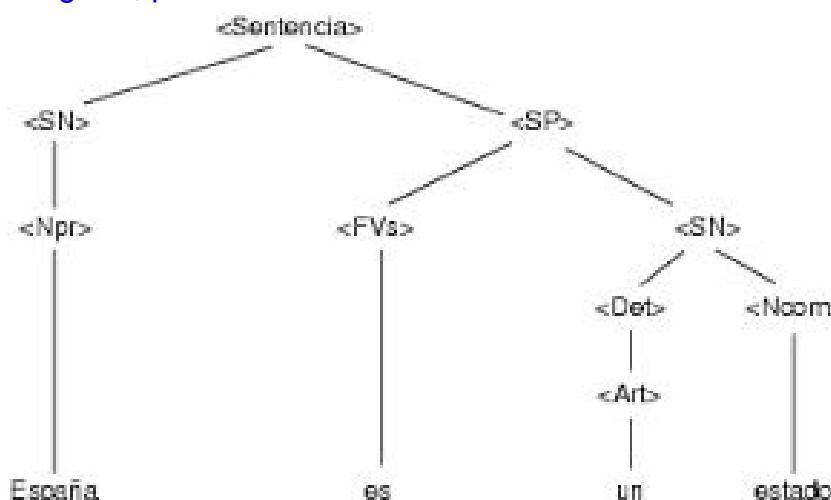
- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar código intermedio.
- Generar errores cuando se producen.

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

Manejo de errores sintácticos

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

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



Parte 4

ANÁLISIS SEMÁNTICO EN PROCESADORES DE LENGUAJE

La fase de análisis semántico de un procesador de lenguaje es aquélla que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación .

Sintaxis de un lenguaje de programación es el conjunto de reglas formales que especifican la estructura de los programas pertenecientes a dicho lenguaje.

Semántica de un lenguaje de programación es el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente válida. Finalmente, el análisis semántico1 de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

La sintaxis del lenguaje C indica que las expresiones se pueden formar con un conjunto de operadores y un conjunto de elementos básicos. Entre los operadores, con sintaxis binaria infija, se encuentran la asignación, el producto y la división. Entre los elementos básicos de una expresión existen los identificadores y las constantes enteras sin signo (entre otros).

Su semántica identifica que en el registro asociado al identificador superficie se le va a asociar el valor resultante del producto de los valores asociados a base y altura, divididos por dos (la superficie de un triángulo).

Especificación Semántica de Lenguajes de Programación

Existen dos formas de describir la semántica de un lenguaje de programación: mediante especificación informal o natural y formal. La descripción informal de un lenguaje de programación es llevada a cabo mediante el lenguaje natural. Esto hace que la especificación sea inteligible (en principio) para cualquier persona.

La experiencia nos dice que es una tarea muy compleja, si no imposible, el describir todas las características de un lenguaje de programación de un modo preciso. Como caso particular, véase la especificación del lenguaje ISO/ANSI C++ [ANSIC++]. La descripción formal de la semántica de lenguajes de programación es la descripción rigurosa del significado o comportamiento de programas, lenguajes de programación, máquinas abstractas o incluso cualquier dispositivo hardware.

- Revelar posibles ambigüedades existentes implementaciones de procesadores de lenguajes o en documentos descriptivos de lenguajes de programación.

- Ser utilizados como base para la implementación de procesadores de lenguaje.
- Verificar propiedades de programas en relación con pruebas de corrección o información relacionada con su ejecución.
- Diseñar nuevos lenguajes de programación, permitiendo registrar decisiones sobre construcciones particulares del lenguaje, así como permitir descubrir posibles irregularidades u omisiones.
- Facilitar la comprensión de los lenguajes por parte del programador y como mecanismo de comunicación entre diseñador del lenguaje, implementador y programador. La especificación semántica de un lenguaje, como documento de referencia, aclara el comportamiento del lenguaje y sus diversas construcciones.
- Estandarizar lenguajes mediante la publicación de su semántica de un modo no ambiguo. Los programas deben poder procesarse en otra implementación de procesador del mismo lenguaje exhibiendo el mismo comportamiento.

Especificación Formal de Semántica

Si bien la especificación formal de la sintaxis de un lenguaje se suele llevar a cabo mediante la descripción estándar de su gramática en notación BNF (Backus-Naur Form), en el caso de la especificación semántica la situación no está tan clara; no hay ningún método estándar globalmente extendido. El comportamiento de las distintas construcciones de un lenguaje de programación, puede ser descrito desde distintos puntos de vista.

Tareas y Objetivos del Análisis Semántico

El análisis semántico¹¹ de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

Comprobaciones pospuestas por el analizador sintáctico

A la hora de implementar un procesador de un lenguaje de programación, es común encontrarse con situaciones en las que una gramática libre de contexto puede representar sintácticamente propiedades del lenguaje; sin embargo, la gramática resultante es compleja y difícil de procesar en la fase de análisis sintáctico. En estos casos es común ver cómo el desarrollador del compilador escribe una gramática más sencilla que no representa detalles del lenguaje, aceptándolos como válidos cuando realmente no pertenecen al lenguaje.

Comprobaciones dinámicas

Todas las comprobaciones semánticas descritas en este punto suelen llevarse a cabo en fase de compilación y por ello reciben el nombre de “estáticas”. Existen

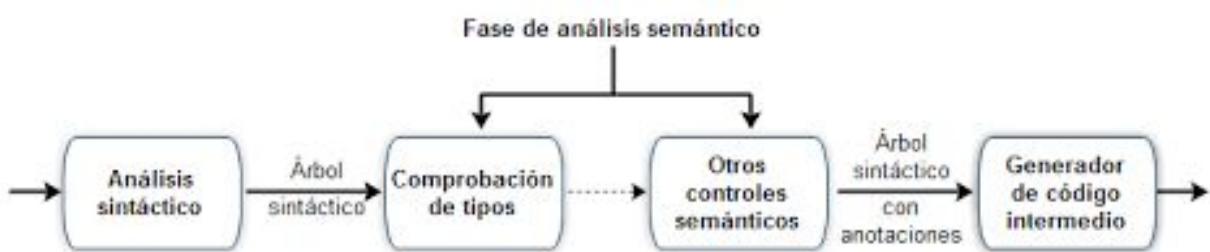


comprobaciones que, en su caso más general, sólo pueden ser llevadas a cabo en tiempo de ejecución y por ello se llaman “dinámicas”. Éstas suelen ser comprobadas por un intérprete o por código de comprobación generado por el compilador –también puede darse el caso de que no se comprueben. Diversos ejemplos pueden ser acceso a un vector fuera de rango, utilización de un puntero nulo o división por cero.

Comprobaciones de tipo

Sin duda, este tipo de comprobaciones es el más exhaustivo y amplio en fase de análisis semántico. Ya bien sea de un modo estático (en tiempo de compilación), dinámico (en tiempo de ejecución) o en ambos, las comprobaciones de tipo son necesarias en todo lenguaje de alto nivel. De un modo somero, el analizador semántico deberá llevar a cabo las dos siguientes tareas relacionadas con los tipos: Tareas y Objetivos del Análisis Semántico.

Comprobar las operaciones que se pueden aplicar a cada construcción del lenguaje. Dado un elemento del lenguaje, su tipo identifica las operaciones que sobre él se pueden aplicar. Por ejemplo, en el lenguaje Java el operador de producto no es aplicable a una referencia a un objeto. De un modo contrario, el operador punto sí es válido. 2. Inferir el tipo de cada construcción del lenguaje. Para poder implementar la comprobación anterior, es necesario conocer el tipo de toda construcción sintácticamente válida del lenguaje. Así, el analizador semántico deberá aplicar las distintas reglas de inferencia de tipos descritas en la especificación del lenguaje de programación, para conocer el tipo de cada construcción del lenguaje.



Santana Gonzalez Jesus Salvador

PARTE 5

Análisis Sintáctico.

Tiene como función etiquetar cada uno de los componentes sintácticos que aparecen en la oración y analizar como las palabras se combinan para formar construcciones gramaticalmente correctas. El resultado de este proceso consiste en



generar la estructura correspondiente a las categorías sintácticas formadas por cada una de las unidades léxicas que aparecen en la oración. Las gramáticas, tal como se muestran en la siguiente figura, están formadas por un conjunto de reglas cuales son:

43 O --> SN,
SV SN --> Det,
N SN --> Nombre Propio
SV --> V,
SN SV --> V SP --> Preposición,
SN SN = sintagma nominal
SV = sintagma verbal Det = determinante

El analizador léxico tiene como entrada el código fuente en forma de una sucesión de caracteres, el analizador sintáctico tiene como entrada los lexemas que le suministra el analizador léxico y su función es comprobar que están ordenados de forma correcta (dependiendo del lenguaje que queramos procesar) los dos analizadores suelen trabajar unidos e incluso el léxico suele ser una subrutina del sintáctico

Análisis Semántico.

Es la extensión del análisis sintáctico para la comprensión de las redes semánticas y la base de conocimientos tanto para ver si tiene sentido y que este dirigida al lenguaje natural ya que es a eso a lo que se enfoca, para decirlo en otras palabras es darle significado asociado a las estructuras formales del lenguaje.

Este analizador computa información adicional necesaria para el procesamiento de un lenguaje una vez que se halla tomando bien un análisis completo y exacto del analizador sintáctico y genere resultados verídicos su objetivo primordial o principal del analizador semántico es que el programa analizado satisfaga las reglas requeridas por la especificación del lenguaje para garantizar su correcta ejecución.

Ventajas del análisis semántico

- La descripción formal de la semántica de lenguajes de programación es la descripción rigurosa del significado o comportamiento de programas, el lenguaje de programación además de verificar propiedades de relación con pruebas de corrección o información relacionada con su ejecución.
- Ser utilizados como base para la implementación de procesadores de lenguaje.
- Facilitar la comprensión de los lenguajes por parte del programador y como mecanismo de comunicación entre el diseñador del lenguaje e implementador, la especificación semántica de un lenguaje como documento



de referencia aclara el comportamiento del lenguaje y sus diversas construcciones.

- Revela posibles ambigüedades existentes.
- Crear implementaciones de procesadores de lenguajes o en documentos descriptivos del procesamiento del lenguaje natural.

El analizador semántico detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico. El analizador semántico suele trabajar simultáneamente al analizador sintáctico y en estrecha cooperación se entiende por semántica como el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente correcta y escrita en un determinado lenguaje.

Las rutinas semánticas deben realizar la evaluación de los atributos de las gramáticas siguiendo las reglas semánticas asociadas a cada producción de la gramática. El análisis sintáctico es la fase en la que se trata de determinar el tipo de los resultados intermedios, comprobar que los argumentos que tiene un operador pertenecen al conjunto de los operadores posibles y si son compatibles entre sí, etc. En definitiva, comprobará que el significado de lo que se va leyendo es válido.

La salida “teórica” de la fase de análisis semántico sería un árbol semántico. Consiste en un árbol sintáctico en el que cada una de sus ramas ha adquirido el significado que debe tener. En el caso de los operadores polimorfismo (un único símbolo con varios significados), el análisis semántico determina cual es el aplicable.

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológicos y sintácticos. El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones 52 semánticas y preparadas la generación de código. Las rutinas semánticas suelen hacer uso de una pila (la pila semántica) que contiene la información semántica asociada a los operandos en forma de registro semánticos.

Front end

El Frontend hace uso de las tecnologías o lenguajes de estilo o programación del lado del cliente para la estructuración, el maquetado y la animación de los sitios web. Los lenguajes de estilo o programación a los que nos referimos son: HTML (lenguaje de marcas de hipertexto), CSS (hojas de estilo en cascada) y JavaScript, entre otros:



El HTML es un lenguaje de marcado que sirve para definir la estructura del contenido de tu web.

El CSS es un lenguaje de estilo que sirve para codificar la estructura creada por el HTML (darle color al texto, incluir márgenes, cambiar la tipografía del contenido...).

El JavaScript es un lenguaje de programación con el que puedes programar la interacción con el usuario.

Una vez aclarados estos tres términos, ¿qué es el Frontend? El Frontend es una tecnología que se encarga del diseño de una página web, es decir, se hace cargo de la estética, borradores, mockups (maqueta del diseño a escala o a tamaño real), interfaz, usabilidad de usuario, estructura, colores, tipografías y efectos. Asimismo, a través de JavaScript está la posibilidad de programar eventos, y validar formularios de contacto, entre otras funciones. En otras palabras, el Frontend es la parte del software encargada de interactuar con los visitantes de la web.

Es la parte de componente visible para el usuario. El front-end en diseño de software y desarrollo web hace referencia a la visualización del usuario navegante o, dicho de otra manera, es la parte que interactúa con los usuarios.

El Frontend abarca aquellas tecnologías que hacen referencia exclusivamente de la comodidad visual del usuario. Estiliza la página web mediante las técnicas que ofrece la User Experience (experiencia del usuario). También conocerá acerca de diseño web para que la estructura del sitio facilite una visualización ordenada y una comodidad para quien esté navegando por la página. Además, se ocupará de generar una web intuitiva para el usuario.

Estas tecnologías se generalizan en los siguientes lenguajes de programación:

- HTML: Encargado de ordenar el contenido de un sitio web.
- CSS: Forma parte del diseño gráfico y se ocupa de la creación y estructuras de los documentos webs.
- JAVA SCRIPT.:Permite la creación de actividades complejas en el desarrollo web como actualizaciones instantáneas, mapas, infografías, 3D, etcétera.

El programador encargado del frontend manejará solo estas 3 tecnologías, conociendo también a la perfección acerca del backend aunque no trabaje con él, ya que están íntimamente relacionados y uno necesita del otro.

En pocas palabras, esta tecnología web se encargará de optimizar visualmente la página, estructurarla cómodamente a los ojos del usuario y mantener una adecuada estética del sitio para que la interacción en él (consulta, solicitud, clics) sea correcta y eficaz. Por ello, una de las cualidades más valiosas para obtener una web responsive es la creatividad para poder diseñar sitios llamativos y atractivos que sean óptimos para todo tipo de dispositivos y resoluciones.

Los componentes que se encuentran en la parte frontal del sistema son los siguientes:

- pruebas de usabilidad y accesibilidad;
- lenguajes de diseño y marcado como HTML, CSS y JavaScript;
- diseño gráfico y herramientas de edición de imágenes;
- posicionamiento en buscadores o SEO;
- rendimiento web y compatibilidad del navegador.

- A configuration of an LR parser is:
 $(s_0 X_1 s_2 X_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$, where,
stack unexpected input
 s_0, s_1, \dots, s_m , are the states of the parser, and X_1, X_2, \dots, X_m , are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 \dots a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 \dots a_n$ is the string to be parsed
- Two parts in the parsing table: **ACTION** and **GOTO**
 - The **ACTION** table can have four types of entries: **shift**, **reduce**, **accept**, or **error**
 - The **GOTO** table provides the next state information that will be used after a *reduce* move



Santana Gonzalez Jesus Salvador

Parte 6

Analizador Léxico, Sintáctico y Semántico

Los ordenadores son una mezcla equilibrada de Software y Hardware

Los compiladores son programas de computadora que traducen de un lenguaje a otro un lenguaje escrito en lenguaje fuente y produce un programa equivalente escrito en lenguaje objeto

Un compilador se compone internamente de varias etapas o faces que realizan operaciones lógicas y estas son:

Analizador léxico

Lee la secuencia de caracteres de izquierda a derecha del programa fuente y agrupa la secuencia de caracteres en unidades con significado propio (componentes léxicos o tokens)

Las palabras clave identificadores, operadores, constantes numéricas, signos de puntuación como separadores de sentencia, llaves, parentesis, etcétera. son diversas clasificaciones de componentes léxicos.

Análisis léxico (Scanner)

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, secuencia de caracteres que representa 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 permita identificar el patrón de transición entre los caracteres de entrada, generando tokens, que posteriormente serán clasificados. Este mecanismo es posible crearlo a partir de un tipo específico de maquina de estados llamado autómata finito.

Representación de un Analizador léxico

Los componentes léxicos se representan:

1. Palabras reservadas: if, while, do, ...
2. Identificadores: variables, funciones, tipos definidos por el usuario, etiquetas,
...
3. Operadores: =, >, <, >=, <=, +, *, ...
4. Símbolos especiales: ;, (), { }, ...
5. Constantes numéricas. literales que representan valores enteros y flotantes.



6. Constantes de carácter: literales que representan cadenas de caracteres.

¿Qué es un analizador léxico?

Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones. La entrada del analizador léxico podemos definirla como una secuencia de caracteres.

El analizador léxico tiene que dividir la secuencia de caracteres en palabras con significado propio y después convertirlo a una secuencia de terminales desde el punto de vista del analizador sintáctico, que es la entrada del analizador sintáctico. El analizador léxico reconoce las palabras en función de una gramática regular de manera que sus NO TERMINALES se convierten en los elementos de entrada de fases posteriores.

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 determina las relaciones estructurales entre los componentes léxicos esto es semejante a realizar el análisis gramatical sobre una fase en el lenguaje natural. La estructura sintáctica se define mediante las gramáticas independientes del contexto

¿Qué es el analizador sintáctico?

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

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar código intermedio.

Análisis Semántico

El análisis semántico dota de un significado coherente a lo que hemos hecho en el análisis sintáctico.

El chequeo semántico se encarga de que los tipos que intervienen en las expresiones sean compatibles o que los parámetros reales de una función sean coherentes con los parámetros formales

Funciones principales

Identificar cada tipo de instrucción y sus componentes

Completar la Tabla de Símbolos

Realizar distintas comprobaciones y validaciones:

Comprobaciones de tipos.

Comprobaciones del flujo de control.

Comprobaciones de unicidad.

Ejemplo de un Analizador-Lexico-Sintactico-Semantico:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- You may freely edit this file. See commented blocks below for -->
<!-- some examples of how to customize the build. -->
<!-- (If you delete it and reopen the project it will be recreated.) -->
<!-- By default, only the Clean and Build commands use this build script. -->
<!-- Commands such as Run, Debug, and Test only use this build script if -->
<!-- the Compile on Save feature is turned off for the project. -->
<!-- You can turn off the Compile on Save (or Deploy on Save) setting -->
<!-- in the project's Project Properties dialog box.-->
<project name="CompiladorFinal" default="default" basedir=".">
    <description>Builds, tests, and runs the project CompiladorFinal.</description>
    <import file="nbproject/build-impl.xml"/>
    <!--
```

There exist several targets which are by default empty and which can be used for execution of your tasks. These targets are usually executed before and after some main targets. They are:

| | |
|----------------------------|--|
| -pre-init: | called before initialization of project properties |
| -post-init: | called after initialization of project properties |
| -pre-compile: | called before javac compilation |
| -post-compile: | called after javac compilation |
| -pre-compile-single: | called before javac compilation of single file |
| -post-compile-single: | called after javac compilation of single file |
| -pre-compile-test: | called before javac compilation of JUnit tests |
| -post-compile-test: | called after javac compilation of JUnit tests |
| -pre-compile-test-single: | called before javac compilation of single JUnit test |
| -post-compile-test-single: | called after javac compilation of single JUnit test |
| -pre-jar: | called before JAR building |
| -post-jar: | called after JAR building |
| -post-clean: | called after cleaning build products |

(Targets beginning with '-' are not intended to be called on their own.)

Example of inserting an obfuscator after compilation could look like this:

```
<target name="-post-compile">
    <obfuscate>
        <fileset dir="${build.classes.dir}"/>
```

```
</obfuscate>
</target>
```

For list of available properties check the imported nbproject/build-impl.xml file.

Another way to customize the build is by overriding existing main targets.

The targets of interest are:

- init-macrodef-javac: defines macro for javac compilation
- init-macrodef-junit: defines macro for junit execution
- init-macrodef-debug: defines macro for class debugging
- init-macrodef-java: defines macro for class execution
- do-jar: JAR building
- run: execution of project
- javadoc-build: Javadoc generation
- test-report: JUnit report generation

An example of overriding the target for project execution could look like this:

```
<target name="run" depends="CompiladorFinal-impl.jar">
    <exec dir="bin" executable="launcher.exe">
        <arg file="${dist.jar}" />
    </exec>
</target>

Notice that the overridden target depends on the jar target and not only on the compile target as the regular run target does. Again, for a list of available properties which you can use, check the target you are overriding in the nbproject/build-impl.xml file.

-->
</project>
```

Ejemplo frontend:

```
nombre-proyecto/
|__ src
|   |__ scss
|   |   |__ style.scss
|   |   |__ inc
|   |       |__ mixins.scss
|   |       |__ normalize.scss
|   |       |__ colors.scss
|   |       |__ variables.scss
|   |       |__ components.scss
|
|   |__ jade
|       |__ page.jade
```



```
|- |__ inc
|   |__ mixins.jade
|   |__ variables.jade
|   __ template
|     |__ templatename.jade

|- __ js
|   |__ functions.js

|- __ images
|   |__ sprites

|- __ gruntfile.js
|- __ package.json
|- __ bower.json
|- __ .editorconfig
|- __ .gitignore
|- __ .htmlhintrc
|- __ .jshintrc

|- __ dist
|   |- __ page.html
|   |- __ assets
|     |- __ css
|       |__ style.css
|       |- __ libs
|         |__ anyexternallib.css
|     |- __ js
|       |__ functions.js
|       |- __ libs
|         |__ jquery-1.11.3.min.js
|         |__ modernizr.js
|         |__ detectizr.js
|         |__ lt-ie-9.min.js
|         |__ anyexternallib.js
|
|     |- __ images
|       |__ sprites.png

|- __ build
|   |- __ page.html
|   |- __ assets
```



```
__ css
|   __ styles.min.css
__ js
|   __ functions.min.js
|   __ libs
|       __ jquery-1.11.3.min.js
|       __ modernizr-detectizr.min.js
|       __ ie.min.js
|       __ plugins.min.js
|
__ images
|   __ sprites.png
```

Santana Gonzalez Jesus Salvador

3.2.2 "Gramáticas independientes del contexto (Context-Free Grammars)"

Una gramática $G = h\Sigma, \Gamma, S, \rightarrow_i$ es independiente del contexto si todas las reglas de producción tienen una de las dos siguientes formas

$$A \rightarrow \alpha A \rightarrow ,$$

donde $A \in \Gamma$.

CFG designará el conjunto de gramáticas independientes del contexto.

El lenguaje generado por G se denotará con $L(G)$.

Un lenguaje L es independiente del contexto si existe una gramática $G \in \text{CFG}$ tal que

$$L = L(G).$$

CFL es el conjunto de lenguajes independientes del contexto

Lenguaje de palíndromas. Sea $GP = h\{a, b\}, \{S\}, S, \rightarrow Pi$, donde

$$S \rightarrow P aSa | bSb | .$$

2 Lenguaje de paréntesis. Sea $GD = h\{(,)\}, \{S\}, S, \rightarrow Di$, con las siguientes reglas

$$S \rightarrow D(S) | SS | .$$

En la teoría del lenguaje formal, una gramática libre de contexto (CFG) es una gramática formal en la que cada regla de producción tiene la forma donde es un solo símbolo no terminal y es una cadena de terminales y / o no terminales (puede estar vacío). Una gramática formal se considera "libre de contexto" cuando sus reglas de producción se pueden aplicar independientemente del contexto de un no terminal.

Independientemente de los símbolos que lo rodeen, el no terminal único del lado izquierdo siempre se puede reemplazar por el lado derecho. Esto es lo que lo distingue de una gramática sensible al contexto. Una gramática formal es esencialmente un conjunto de reglas de producción que describen todas las cadenas posibles en un lenguaje formal dado. Las reglas de producción son reemplazos simples. Por ejemplo, la primera regla de la imagen, reemplaza con . Puede haber varias reglas de reemplazo para un símbolo no terminal dado. El lenguaje generado por una gramática es el conjunto de todas las cadenas de símbolos terminales que pueden derivarse, mediante aplicaciones repetidas de reglas, de algún símbolo no terminal particular ("símbolo de inicio"). Los símbolos no terminales se utilizan durante el proceso de derivación, pero es posible que no aparezcan en su cadena de resultado final.

Gramáticas Libres de Contexto

Def. Una *gramática libre de contexto* (*context-free grammar*) es un cuadraplo $G = \langle V, \Sigma, S, P \rangle$, en donde: V , conjunto de símbolos terminales, Σ alfabeto, S símbolo inicial y en la cual las producciones P son de la forma:

$$A \rightarrow \alpha$$

en donde A es una categoría sintáctica y α una cadena de símbolos terminales o categorías sintácticas.

Las gramáticas libres de contexto generan lenguajes libres de contexto.

Ejemplos de lenguajes libres de contexto:

- Los palindromos que son palabras que se leen igual si se leen en cualquier dirección. Para un vocabulario de 0's y 1's:
 $\{x, 0, 1, 00, 11, 010, 000, 101, 111, \dots\}$ (x es la cadena vacía)

| | | |
|------------------------------|-------------------------|-------------------------|
| R1. $S \rightarrow \epsilon$ | R3. $S \rightarrow 1$ | R5. $S \rightarrow 1S1$ |
| R2. $S \rightarrow 0$ | R4. $S \rightarrow 0S0$ | |

- $\{a^n b^n | n \geq 0\}$

| | |
|-----------------------|------------------------|
| R1. $S \rightarrow a$ | R2. $S \rightarrow ab$ |
|-----------------------|------------------------|

Una gramática regular es una 4-tupla $G=(\Sigma, N, S, P)$, donde Σ es el alfabeto, N es una colección de símbolos no terminales, S es un símbolo no terminal llamado símbolo inicial y P es un conjunto de reglas de sustitución, llamadas producciones de la forma $A \rightarrow w$ donde $A \in N$ y $w \in (\Sigma \cup N)^*$ satisfaciendo:

- 1) w contiene un no terminal como máximo.
- 2) Si w contiene un no terminal, entonces es el símbolo que está en el extremo derecho de w .

El lenguaje generado por la gramática se representa como $L(G)$.

Sea L el lenguaje regular reconocido por un AFD $M = (\Sigma, Q, s, F, \delta)$. La gramática regular que genera el lenguaje L será $G = (\Sigma, N, S, P)$ tal que: $\Sigma = \Sigma N = Q S = s P =$

$\{q \rightarrow ap \mid \delta(q, a) = p\} \cup \{q \rightarrow \epsilon \mid q \in F\}$ z Todas aquellas palabras que reconoce el AFD M, pueden ser generadas a partir de las producciones de G.

Una Gramática Independiente del Contexto (GIC) es una 4-tupla
 $G = (\Sigma, N, S, P)$

donde Σ es el alfabeto (conjunto de terminales), N es la colección finita de los no terminales, S es un no terminal llamado símbolo inicial y $P \subseteq N \times (N \cup \Sigma)^*$ es un conjunto de producciones.

El lenguaje generado por la GIC se denota $L(G)$ y se llama Lenguaje Independiente del Contexto (LIC).

Relación GR y GIC:

$$\begin{aligned} GR &\subseteq GIC \\ LR &\subseteq LIC \end{aligned}$$

Sea L el lenguaje generado por la gramática regular $G = (\Sigma, N, S, P)$. El AFN que reconoce el lenguaje L será $M = (\Sigma, Q, s, F, \Delta)$ tal que:

$$Q = N \cup \{f\}, \text{ siendo } f \text{ un símbolo nuevo}$$

$$s = S$$

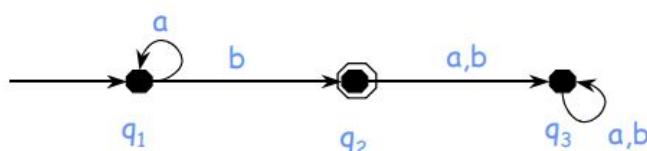
$$F = \{f\}$$

Δ se definen:

- Si $A \rightarrow \sigma_1 \dots \sigma_n B$ con A y B no terminales, entonces se añadirán a Q los nuevos estados q_1, q_2, \dots, q_{n-1} y las transformaciones siguientes:
 $\Delta(A, \sigma_1 \dots \sigma_n) = \Delta(q_1, \sigma_2 \dots \sigma_n) = \dots = \Delta(q_{n-1}, \sigma_n) = B$

- Si $A \rightarrow \sigma_1 \dots \sigma_n$ con A no terminal, entonces se añadirán a Q los nuevos estados q_1, q_2, \dots, q_{n-1} y las transformaciones siguientes:
 $\Delta(A, \sigma_1 \dots \sigma_n) = \Delta(q_1, \sigma_2 \dots \sigma_n) = \dots = \Delta(q_{n-1}, \sigma_n) = f$

- Ejemplo: Sea el AFD M ,



Santana Gonzalez Jesus Salvador

Gramáticas independientes del contexto (context-free grammars): usan reglas que predicen las palabras que pueden posiblemente seguir a la última palabra reconocida, reduciendo el número de palabras candidatas a evaluar para reconocer la siguiente pronunciación del speaker (hablante o fichero de voz). Las reglas están formadas por dos tipos de símbolos: palabras y directrices. Ejemplo: =ALT(SEQ("Jesús", "Moreno"), "Miguel Alonso") que significa algo como "lo que en la categoría que se puede dar como válido al reconocer el habla es una de las siguientes alternativas: la secuencia Jesús y detrás Moreno o Miguel Alonso" Estas gramáticas usan "pedazos" (chunks) de información para comunicarse con el motor de reconocimiento: Words, Rules, Exported Rules, Imported Rules, Run-Time Lists.

Una Gramática independientes del contexto (GIC) es una gramática formal en la que cada regla de producción es de la forma:

Exp → x

Donde Exp es un símbolo no terminal y x es una cadena de terminales y / o no terminales. El término independiente del contexto se refiere al hecho de que el no terminal Exp puede siempre ser sustituido por x sin tener en cuenta el contexto en el que ocurre. Un lenguaje formal es independiente de contexto si hay una gramática libre de contexto que lo general, este tipo de gramática fue creada por Backus-Naur y se utiliza para describir la mayoría de los lenguajes de programación.

Una GIC está compuesta por 4 elementos:

1. Símbolos terminales (elementos que no generan nada)
2. No terminales (elementos del lado izquierdo de una producción, antes de la flecha "->")
3. Producciones (sentencias que se escriben en la gramática)
4. Símbolo inicial (primer elemento de la gramática)

Ejemplo 1: Teniendo un lenguaje que genera expresiones de tipo:

9 + 5 - 2

Para determinar si una GIC esta bien escrita se utilizan los arboles de análisis sintáctico , así:

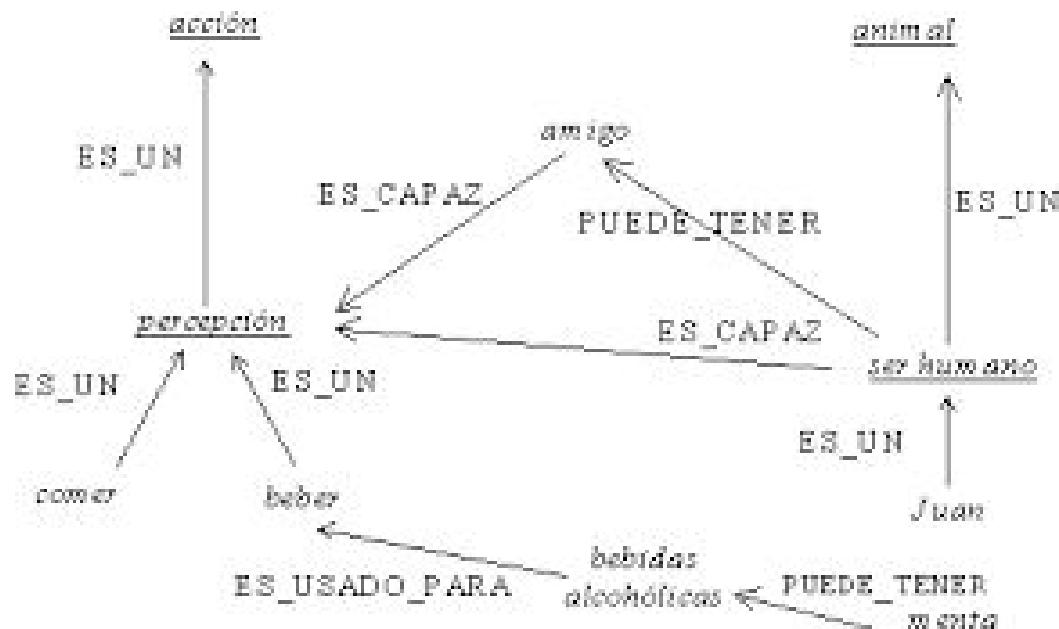
Producciones:

lista -> lista + digito

lista -> lista - digito

lista -> digito

digito -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Las Gramáticas Libres de Contexto (‘Context-Free Languages) o CFLs jugaron un papel central en lenguaje natural desde los 50's y en los compiladores desde los 60's

Las Gramáticas Libres de Contexto forman la base de la sintaxis BNF

Son actualmente importantes para XML y sus DTDs (document type definition)

Una gramática libre de contexto se define con ' $G = (V, T, P, S)$ ' donde:

- V es un conjunto de variables
 - T es un conjunto de terminales
 - P es un conjunto finito de producciones de la forma $A \rightarrow \alpha$, donde A es una variable y $\alpha \in (V \cup T)^*$
 - S es una variable designada llamada el símbolo inicial

Ejemplo:

$G_{pal} = (\{P\}, \{0, 1\}, A, P)$, donde $A = \{P \rightarrow , P \rightarrow 0, P \rightarrow 1, P \rightarrow 0P0, P \rightarrow 1P1\}$.

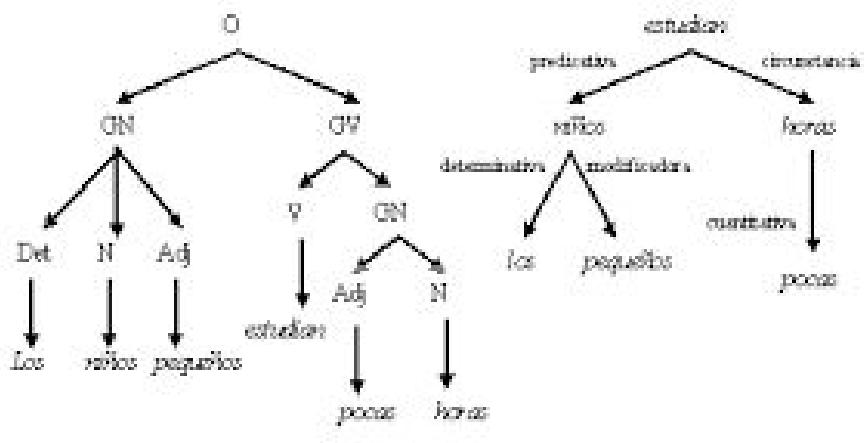
- Muchas veces se agrupan las producciones con la misma cabeza, e.g., $A = \{P \rightarrow |0|1|0P_0|1P_1\}$.



- Ejemplo: Expresiones regulares sobre $\{0, 1\}$ se pueden definir por la gramática:
 $\text{Gregex} = (\{E\}, \{0, 1\}, A, E)$, donde $A = \{E \rightarrow 0, E \rightarrow 1, E \rightarrow E.E, E \rightarrow E + E, E \rightarrow E^*, E \rightarrow (E)\}$.

Estas gramáticas, conocidas también como gramáticas de tipo 2 o gramáticas independientes del contexto, son las que generan los lenguajes libres o independientes del contexto.

Los lenguajes libres del contexto son aquellos que pueden ser reconocidos por un autómata de pila determinista o no determinístico.



A) Árbol de constituyentes

B) Árbol de dependencias

Las gramáticas libres del contexto se escriben, frecuentemente, utilizando una notación conocida como BNF (Backus-Naur Form). BNF es la técnica más común para definir la sintaxis de los lenguajes de programación.

En esta notación se deben seguir las siguientes convenciones:

- los no terminales se escriben entre paréntesis angulares $< >$
- los terminales se representan con cadenas de caracteres sin paréntesis angulares
- el lado izquierdo de cada regla debe tener únicamente un no terminal (ya que es una gramática libre del contexto)
- el símbolo $::=$, que se lee “se define como” o “se reescribe como”, se utiliza en lugar de \rightarrow
- varias producciones del tipo $::= ::= \dots ::=$ se pueden escribir como $::= \dots$

Árbol de derivación

Un árbol de derivación permite mostrar gráficamente cómo se puede derivar cualquier cadena de un lenguaje a partir del símbolo distinguido de una gramática que genera ese lenguaje.

Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas, llamadas arcos.

Un arco conecta dos nodos distintos.



Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades:

- hay un único nodo distinguido, llamado raíz (se dibuja en la parte superior) que no tiene arcos incidentes.
- todo nodo c excepto el nodo raíz está conectado con un arco a otro nodo k, llamado el padre de c (c es el hijo de k). El padre de un nodo, se dibuja por encima del nodo.
- todos los nodos están conectados al nodo raíz mediante un único camino.
- los nodos que no tienen hijos se denominan hojas, el resto de los nodos se denominan nodos interiores. El árbol de derivación tiene las siguientes propiedades:
 - el nodo raíz está rotulado con el símbolo distinguido de la gramática;
 - cada hoja corresponde a un símbolo terminal o un símbolo no terminal;
 - cada nodo interior corresponde a un símbolo no terminal.

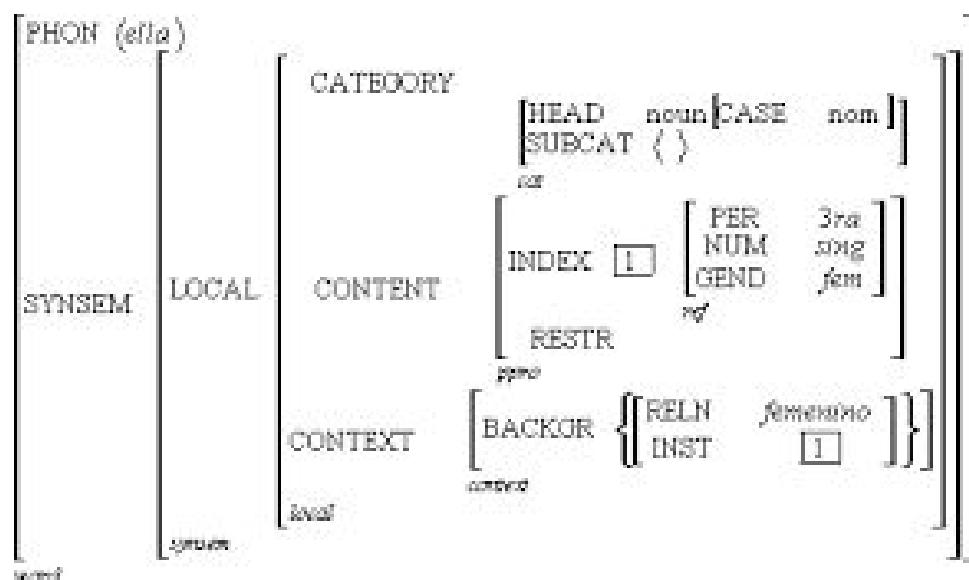
Gramáticas ambiguas

Una gramática es ambigua si permite construir dos o más árboles de derivación distintos para la misma cadena. Por lo tanto, para demostrar que una gramática es ambigua lo único que se necesita es encontrar una cadena que tenga más de un árbol de derivación.

Una gramática en la cual, para toda cadena generada w, todas las derivaciones de w tienen el mismo árbol de derivación es no ambigua. En algunos casos, dada una gramática ambigua, se puede encontrar otra gramática que produzca el mismo lenguaje pero que no sea ambigua.

Si todas las gramáticas independientes del contexto para un lenguaje son ambiguas, se dice que el lenguaje es un lenguaje independiente del contexto inherentemente ambiguo.

Por ejemplo, el lenguaje $L = \{ ai \text{ } bj \text{ } ck / i = j \text{ ó } j = k \}$



Santana Gonzalez Jesus Salvador

Semana 14

Una representación intermedia es una estructura de datos que representa al programa fuente durante el proceso de la traducción a código objeto. Hasta ahora hemos usado el árbol de análisis sintáctico como representación intermedia, junto con la tabla de símbolos que contenía información sobre los nombres (variables, constantes, tipos y funciones) que aparecían en el programa fuente. Aunque el árbol de análisis sintáctico es una representación válida, no se parece ni remotamente al código objeto, en el que solo se emplean saltos a direcciones en memoria en vez de construcciones de alto nivel, como sentencias if-then-else. Es necesario generar una nueva forma de representación intermedia. A esta representación intermedia, que se parece al código objeto pero que sigue siendo independiente de la máquina, se le llama código intermedio. El código intermedio puede tomar muchas formas. Todas ellas se consideran como una forma de linearización del árbol sintáctico, es decir, una representación del árbol sintáctico de forma secuencial. El código intermedio más habitual es el código de 3-direcciones. El código de tres direcciones es una secuencia de proposiciones de la forma general $x = y$ op z donde p representa cualquier operador; x,y,z representan variables definidas por el programador o variables temporales generadas por el compilador. y,también pueden representar constantes o literales. op representa cualquier operador: un operador aritmético de punto fijo o flotante, o un operador lógico sobre datos booleanos. No se permite ninguna expresión aritmética compuesta, pues sólo hay un operador en el lado derecho. Por ejemplo, $x+y$ *se debe traducir a una secuencia, donde t1, t2 son variables temporales generadas por el compilador. $t1 = y * z$ $t2 = x + t1$ Las expresiones compuestas y las proposiciones de flujo de control se han de descomponer en proposiciones de este tipo, definiendo un conjunto suficientemente amplio de operadores. Se le llama código de 3-direcciones porque cada proposición contiene, en el caso general, tres direcciones, dos para los operandos y una para el resultado. (Aunque aparece el nombre de la variable, realmente corresponde al puntero a la entrada de la tabla de símbolos de dicho nombre). El código de tres direcciones es una representación linealizada (de izquierda a derecha) del árbol sintáctico en la que los nombres temporales corresponden a los nodos internos. Como estos nombres temporales se representan en la memoria no se especifica más información sobre ellos en este tipo de código. Normalmente se asignan directamente a registros o se almacenan en la tabla de símbolos. $2*a+b-3$.

La forma de código de 3-direcciones es insuficiente para representar todas las construcciones de un lenguaje de programación (saltos condicionales, saltos incondicionales, llamadas a funciones, bucles, etc), por tanto es necesario introducir nuevos operadores. El conjunto de proposiciones (operadores) debe ser lo suficientemente rico como para poder implantar las operaciones del lenguaje fuente. Las proposiciones de 3-direcciones van a ser en cierta manera análogas al código ensamblador. Las proposiciones pueden tener etiquetas simbólicas y existen instrucciones para el flujo de control (goto). Una etiqueta simbólica representa el índice de una proposición de 3-direcciones en la lista de instrucciones.



Las proposiciones de 3-direcciones más comunes:

1. Proposiciones de la forma $x = y \text{ op } z$ donde op es un operador binario aritmético, lógico o relacional.
2. Instrucciones de la forma $x = \text{op } y$, donde op es un operador unario (operador negación lógico, operadores de desplazamiento o conversión de tipos).
3. Proposiciones de copia de la forma $x = y$, donde el valor de y se asigna a x.
4. Salto incondicional goto etiq. La instrucción con etiqueta etiq es la siguiente que se ejecutará.
5. Saltos condicionales como if false x goto etiq.
6. param x y call f para apilar los parámetros y llamadas a funciones (los procedimientos se consideran funciones que no devuelven valores). También return y, que es opcional, para devolver valores. Código generado como parte de una llamada al procedimiento p(x 1,x 2,...,x n). param x1 param x2 ... param xn call p,n
7. Asignaciones con índices de la forma $x = y[i]$, donde se asigna a x el valor de la posición en i unidades de memoria más allá de la posición y. O también $x[i] = y$.
8. Asignaciones de direcciones a punteros de la forma $x = &y$ (el valor de x es la dirección de y), $x = *y$ (el valor de x se iguala al contenido de la dirección indicada por el puntero y) o $*x = y$ (el objeto apuntado por x se iguala al valor de y).

Ejemplo. Consideremos el código que calcula el factorial de un número. La tabla 7.1 muestra el código fuente y el código de 3- direcciones. Existe un salto condicional if false que se usa para traducir las sentencias de control if-then, repeat-until que contiene dos direcciones: el valor condicional de la expresión y la dirección de salto.

La proposición label sólo tiene una dirección.

Las operaciones de lectura y escritura, read, write, con una sola dirección.

Y una instrucción de parada halt que no tiene direcciones.

```
read x; read x
if 0<x then t1 = 0 < x
fact:=1; if false t1 goto L1
repeat fact=1
fact:=fact*x; label L2
x:=x-1; t2=fact * x
until x=0; fact=t2
write fact; t3=x-1
end; x=t3
t4=x==0
if false t4 goto L2
write fact
label L1
halt
```

Procesadores de lenguaje

Los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. Nuestra percepción del mundo en que vivimos depende de los lenguajes de programación, ya que todo el software que se ejecuta en todas las computadoras se escribió en algún lenguaje de programación. Los sistemas de software que se encargan de esta traducción se llaman compiladores. Un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje destino). El programa destino en lenguaje máquina que produce un compilador es, por lo general, más rápido que un intérprete al momento de asignar las entradas a las salidas.

La estructura de un compilador

La parte del análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se pasa junto con la representación intermedia a la parte de la síntesis. La parte de la síntesis construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el front-end del compilador; la parte de la síntesis (propriamente la traducción) es el back-end. Algunos compiladores tienen una fase de optimización de código independiente de la máquina, entre el front-end y el back-end. El propósito de esta optimización es realizar transformaciones sobre la representación intermedia, para que el back-end pueda producir un mejor programa destino de lo que hubiera producido con una representación intermedia sin optimizar.

Análisis de léxico

El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas.

Análisis sintáctico

El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.

Las fases siguientes del compilador utilizan la estructura gramatical para ayudar a analizar el programa fuente y generar el programa destino.

Análisis semántico

Utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio. Una parte importante del

análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tiene operandos que coincidan.

Cabrera Ramírez Gerardo

("Análisis Léxico (partes 2 y 3)")

Traducción binaria

La tecnología de compiladores puede utilizarse para traducir el código binario para una máquina al código binario para otra máquina distinta, con lo cual se permite a una máquina ejecutar los programas que originalmente eran compilados para otro conjunto de instrucciones. La traducción binaria también puede usarse para ofrecer compatibilidad inversa.

Síntesis de hardware

No sólo la mayoría del software está escrito en lenguajes de alto nivel; incluso hasta la mayoría de los diseños de hardware se describen en lenguajes de descripción de hardware de alto nivel, como Verilog y VHDL (Very High-Speed Integrated Circuit Hardware Description Language, Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad).

Simulación compilada

La simulación es una técnica general que se utiliza en muchas disciplinas científicas y de ingeniería para comprender un fenómeno, o para validar un diseño. Por lo general, las entradas de los simuladores incluyen la descripción del diseño y los parámetros específicos de entrada para esa ejecución específica de la simulación. Las simulaciones pueden ser muy costosas. Por lo general, necesitamos simular muchas alternativas de diseño posibles en muchos conjuntos distintos de entrada, y cada experimento puede tardar días en completarse, en una máquina de alto rendimiento.

Al hablar sobre el análisis léxico, utilizamos tres términos distintos, pero relacionados:

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

Un patrón es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres

que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.

Un lexema es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

Atributos para los tokens

Cuando más de un lexema puede coincidir con un patrón, el analizador léxico debe proporcionar a las subsiguientes fases del compilador información adicional sobre el lexema específico que coincidió.

Otras funciones que realiza :

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, ..., y tratarlos correctamente con respecto a la tabla de símbolos (sólo en los casos que debe de tratar con la tabla de símbolos).
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información sobre dónde se ha producido.

Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente. La estrategia de recuperación más simple es la recuperación en “modo de pánico”. Eliminamos caracteres sucesivos del resto de la entrada, hasta que el analizador léxico pueda encontrar un token bien formado al principio de lo que haya quedado de entrada. Esta técnica de recuperación puede confundir al analizador sintáctico, pero en un entorno de computación interactivo, puede ser bastante adecuado. Otras de las posibles acciones de recuperación de errores son:

1. Eliminar un carácter del resto de la entrada.
2. Insertar un carácter faltante en el resto de la entrada.
3. Sustituir un carácter por otro.
4. Transponer dos caracteres adyacentes.

Las transformaciones como éstas pueden probarse en un intento por reparar la entrada. La estrategia más sencilla es ver si un prefijo del resto de la entrada puede transformarse en un lexema válido mediante una transformación simple. Esta estrategia tiene sentido, ya que en la práctica la mayoría de los errores léxicos involucran a un solo carácter. Una estrategia de corrección más general es encontrar el menor número de transformaciones necesarias para convertir el programa fuente en uno que consista

sólo de lexemas válidos, pero este método se considera demasiado costoso en la práctica como para que valga la pena realizarlo.

Uso de búfer en la entrada

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

En C, los operadores de un solo carácter como `-`, `=` o `<` podrían ser también el principio de un operador de dos caracteres, como `->`, `==` o `<=`. Por ende, vamos a presentar un esquema de dos búferes que se encarga de las lecturas por adelantado extensas sin problemas. Después consideraremos una mejora en la que se utilizan “centinelas” para ahorrar tiempo al verificar el final de los búferes.

Pares de búferes

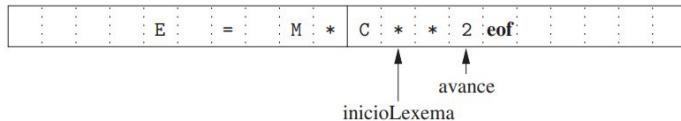
Debido al tiempo requerido para procesar caracteres y al extenso número de caracteres que se deben procesar durante la compilación de un programa fuente extenso, se han desarrollado técnicas especializadas de uso de búferes para reducir la cantidad de sobrecarga requerida en el procesamiento de un solo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

Cada búfer es del mismo tamaño N , y por lo general N es del tamaño de un bloque de disco (es decir, 4 096 bytes). Mediante el uso de un comando de lectura del sistema podemos leer N caracteres y colocarlos en un búfer, en vez de utilizar una llamada al sistema por cada carácter. Si quedan menos de N caracteres en el archivo de entrada, entonces un carácter especial, representado por `eof`, marca el final del archivo fuente y es distinto a cualquiera de los posibles caracteres del programa fuente. Se mantienen dos apuntadores a la entrada:

1. El apuntador `inicioLexema` marca el inicio del lexema actual, cuya extensión estamos tratando de determinar.
2. El apuntador `avance` explora por adelantado hasta encontrar una coincidencia en el patrón; durante el resto del capítulo cubriremos la estrategia exacta mediante la cual se realiza esta determinación.

Una vez que se determina el siguiente lexema, `avance` se coloca en el carácter que se encuentra en su extremo derecho. Después, una vez que el lexema se registra como un valor de atributo de un token devuelto al analizador sintáctico, `inicioLexema` se coloca en el carácter que va justo después del lexema que acabamos de encontrar.

Centinelas



Si utilizamos el esquema en la forma descrita, debemos verificar, cada vez que movemos el apuntador avance, que no nos hayamos salido de uno de los búferes; si esto pasa, entonces también debemos recargar el otro búfer. Así, por cada lectura de caracteres hacemos dos pruebas: una para el final del búfer y la otra para determinar qué carácter se lee (esta última puede ser una bifurcación de varias vías). Podemos combinar la prueba del final del búfer con la prueba del carecer actual si extendemos cada búfer para que contenga un valor centinela al final. El centinela es un carácter especial que no puede formar parte del programa fuente, para lo cual una opción natural es el carácter eof.

Cadenas y lenguajes

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

Una cadena sobre un alfabeto es una secuencia finita de símbolos que se extraen de ese alfabeto. En la teoría del lenguaje, los términos “oración” y “palabra” a menudo se utilizan como sinónimos de “cadena”. La longitud de una cadena s , que por lo general se escribe como $|s|$, es el número de ocurrencias de símbolos en s .

Operaciones en los lenguajes

En el análisis léxico, las operaciones más importantes en los lenguajes son la unión, la concatenación y la cerradura

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

Figura 3.6: Definiciones de las operaciones en los lenguajes

Expresiones regulares

Se usa para describir a todos los lenguajes que puedan construirse a partir de estos operadores, aplicados a los símbolos de cierto alfabeto. En esta notación, si letra_ se establece de manera que represente a cualquier letra o al guion bajo, y dígito_ se Figura 3.6: Definiciones de las operaciones en los lenguajes OPERACIÓN DEFINICIÓN Y NOTACIÓN Unión de L y M $L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$ Concatenación de L y M $LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$ Cerradura de Kleene de L $L^* = \bigcup_{i=0}^{\infty} L^i$ Cerradura positivo de L $L^+ = \bigcup_{i=1}^{\infty} L^i$



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

letra_(letra_ | dígito)*

La barra vertical de la expresión anterior significa la unión, los paréntesis se utilizan para agrupar las subexpresiones, el asterisco indica "cero o más ocurrencias de", y la yuxtaposición de letra_ con el resto de la expresión indica la concatenación. Las expresiones regulares se construyen en forma recursiva a partir de las expresiones regulares más pequeñas, usando las reglas que describiremos a continuación. Cada expresión regular r denota un lenguaje $L(r)$, el cual también se define en forma recursiva, a partir de los lenguajes denotados por las subexpresiones de r . He aquí las reglas que definen las expresiones regulares sobre cierto alfabeto Σ , y los lenguajes que denotan dichas expresiones.

BASE: Hay dos reglas que forman la base:

1. a es una expresión regular, y $L(a) = \{a\}$; es decir, el lenguaje cuyo único miembro es la cadena vacía.
2. Si a es un símbolo en Σ , entonces a es una expresión regular, y $L(a) = \{a\}$, es decir, el lenguaje con una cadena, de longitud uno, con a en su única posición. Tenga en cuenta que por convención usamos cursiva para los símbolos, y negrita para su correspondiente expresión regular.

INDUCCIÓN: Hay cuatro partes que constituyen la inducción, mediante la cual las expresiones regulares más grandes se construyen a partir de las más pequeñas. Suponga que r y s son expresiones regulares que denotan a los lenguajes $L(r)$ y $L(s)$, respectivamente.

1. $(r)l(s)$ es una expresión regular que denota el lenguaje $L(r) \cup L(s)$.
2. $(r)(s)$ es una expresión regular que denota el lenguaje $L(r)L(s)$.
3. $(r)^*$ es una expresión regular que denota $L((r))^*$.
4. (r) es una expresión regular que denota a $L(r)$. Esta última regla dice que podemos agregar pares adicionales de paréntesis alrededor de las expresiones, sin cambiar el lenguaje que denotan.

Según su definición, las expresiones regulares a menudo contienen pares innecesarios de paréntesis. Tal vez sea necesario eliminar ciertos pares de paréntesis, si adoptamos las siguientes convenciones:

- a) El operador unario $*$ tiene la precedencia más alta y es asociativo a la izquierda.
- b) La concatenación tiene la segunda precedencia más alta y es asociativa a la izquierda.

c) | tiene la precedencia más baja y es asociativo a la izquierda.

A un lenguaje que puede definirse mediante una expresión regular se le llama conjunto regular. Si dos expresiones regulares r y s denotan el mismo conjunto regular, decimos que son equivalentes y escribimos $r = s$. Por ejemplo, $(a|b) = (b|a)$. Hay una variedad de leyes algebraicas para las expresiones regulares; cada ley afirma que las expresiones de dos formas distintas son equivalentes. La figura 3.7 muestra parte de las leyes algebraicas para las expresiones regulares arbitrarias r , s y t .

| LEY | DESCRIPCIÓN |
|----------------------------------|--|
| $r s = s r$ | es conmutativo |
| $r (s t) = (r s) t$ | es asociativo |
| $r(st) = (rs)t$ | La concatenación es asociativa |
| $r(s t) = rs rt; (s t)r = sr tr$ | La concatenación se distribuye sobre |
| $\epsilon r = r\epsilon = r$ | ϵ es la identidad para la concatenación |
| $r^* = (r \epsilon)^*$ | ϵ se garantiza en un cerradura |
| $r^{**} = r^*$ | * es idempotente |

Extensiones de las expresiones regulares

Desde que Kleene introdujo las expresiones regulares con los operadores básicos para la unión, la concatenación y la cerradura de Kleene en la década de 1950, se han agregado muchas extensiones a las expresiones regulares para mejorar su habilidad al especificar los patrones de cadenas.

1. Una o más instancias. El operador unario postfijo $+$ representa la cerradura positivo de una expresión regular y su lenguaje. Es decir, si r es una expresión regular, entonces $(r)+$ denota el lenguaje $(L(r))_+$. El operador $+$ tiene la misma precedencia y asociatividad que el operador $*$. Dos leyes algebraicas útiles, $r^* = r+|$ y $r+ = rr^* = r^*r$ relacionan la cerradura de Kleene y la cerradura positiva.
2. Cero o una instancia. ¿El operador unario postfijo $?$ significa “cero o una ocurrencia”. Es decir, $r?$ es equivalente a $r|$, o dicho de otra forma, $L(r?) = L(r) \cup \{\}$. El operador $?$ tiene la misma precedencia y asociatividad que $*$ y $+$.
3. Clases de caracteres. Una expresión regular $a_1|a_2|\dots|a_n$, en donde las a_i s son cada una símbolos del alfabeto, puede sustituirse mediante la abreviación $[a_1a_2\dots a_n]$. Lo que es más importante, cuando a_1, a_2, \dots, a_n forman una secuencia lógica, por ejemplo, letras mayúsculas, minúsculas o dígitos consecutivos, podemos sustituirlos por a_1-a_n ; es decir, sólo la primera y última separadas por un guion corto. Así, $[abc]$ es la abreviación para $a|b|c$, y $[a-z]$ lo es para $a|b|\dots|z$.

Reconocimiento de tokens

Para operar, usamos los operadores de comparación de lenguajes como Pascal o SQL, en donde = es “es igual a” y <> es “no es igual a”, ya que presenta una estructura interesante de lexemas. Las terminales de la gramática, que son if, then, else, opel, id y numero, son los nombres de tokens en lo que al analizador léxico respecta.

| | |
|----------------|---|
| <i>digito</i> | $\rightarrow [0-9]$ |
| <i>digitos</i> | $\rightarrow digito^+$ |
| <i>numero</i> | $\rightarrow digitos \ (. \ digitos)? \ (\ E \ [+-]? \ digitos \)?$ |
| <i>letra</i> | $\rightarrow [A-Za-z]$ |
| <i>id</i> | $\rightarrow letra \ (letra \mid digito)^*$ |
| <i>if</i> | $\rightarrow if$ |
| <i>then</i> | $\rightarrow then$ |
| <i>else</i> | $\rightarrow else$ |
| <i>oprel</i> | $\rightarrow < \mid > \mid <= \mid >= \mid = \mid <>$ |

Para este lenguaje, el analizador léxico reconocerá las palabras clave if, then y else, así como los lexemas que coinciden con los patrones para opel, id y numero. Para simplificar las cosas, vamos a hacer la suposición común de que las palabras clave también son palabras reservadas; es decir, no son identificadores, aun cuando sus lexemas coinciden con el patrón para identificadores.

Además, asignaremos al analizador léxico el trabajo de eliminar el espacio en blanco, reconociendo el “token” ws definido por:

$$ws \rightarrow (\text{blanco} \mid \text{tab} \mid \text{nuevalinea})^+$$

Aquí, blanco, tab y nuevalinea son símbolos abstractos que utilizamos para expresar los caracteres ASCII de los mismos nombres. El token ws es distinto de los demás tokens porque cuando lo reconocemos, no lo regresamos al analizador sintáctico, sino que reiniciamos el analizador léxico a partir del carácter que va después del espacio en blanco. El siguiente token es el que se devuelve al analizador sintáctico.

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

Diagramas de transición de estados



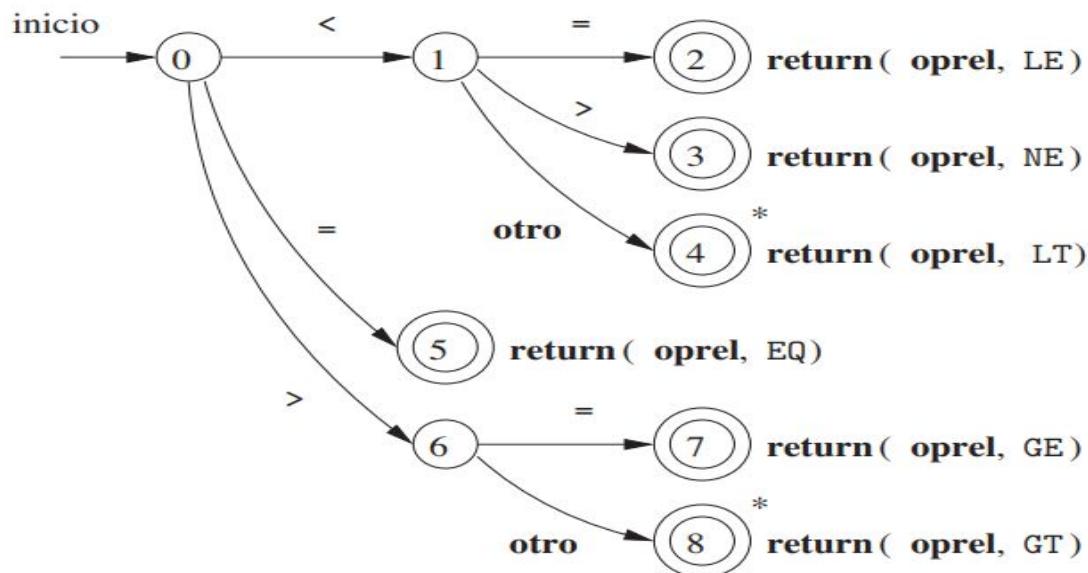
Como paso intermedio en la construcción de un analizador léxico, primero convertimos los patrones en diagramas de flujo estilizados, a los cuales se les llama “diagramas de transición de estados”.

Los diagramas de transición de estados tienen una colección de nodos o círculos, llamados estados. Cada estado representa una condición que podría ocurrir durante el proceso de explorar la entrada, buscando un lexema que coincida con uno de varios patrones. Podemos considerar un estado como un resumen de todo lo que debemos saber acerca de los caracteres que hemos visto entre el apuntador inicioLexema y el apuntador avance.

Las líneas se dirigen de un estado a otro del diagrama de transición de estados. Cada línea se etiqueta mediante un símbolo o conjunto de símbolos. Si nos encontramos en cierto estado s , y el siguiente símbolo de entrada es a , buscamos una línea que salga del estado s y esté etiquetado por a (y tal vez por otros símbolos también). Si encontramos dicha línea, avanzamos el apuntador avance y entramos al estado del diagrama de transición de estados al que nos lleva esa línea. Asumiremos que todos nuestros diagramas de transición de estados son deterministas, lo que significa que nunca hay más de una línea que sale de un estado dado, con un símbolo dado de entre sus etiquetas.

Algunas convenciones importantes de los diagramas de transición de estados son:

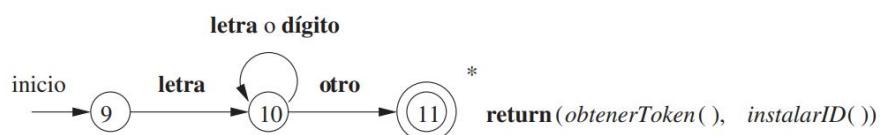
1. Se dice que ciertos estados son de aceptación, o finales. Estos estados indican que se ha encontrado un lexema, aunque el lexema actual tal vez no consista de todas las posiciones entre los apuntadores inicioLexema y avance. Siempre indicamos un estado de aceptación mediante un círculo doble, y si hay que realizar una acción (por lo general, devolver un token y un valor de atributo al analizador sintáctico), la adjuntaremos al estado de aceptación.
2. Además, si es necesario retroceder el apuntador avance una posición (es decir, si el lexema no incluye el símbolo que nos llevó al estado de aceptación), entonces deberemos colocar de manera adicional un * cerca del estado de aceptación.
3. Un estado se designa como el estado inicial; esto se indica mediante una línea etiquetada como “inicio”, que no proviene de ninguna parte. El diagrama de transición siempre empieza en el estado inicial, antes de leer cualquier símbolo de entrada.



Por otro lado, si en el estado 0 el primer carácter que vemos es =, entonces este carácter debe ser el lexema. De inmediato devolvemos ese hecho desde el estado 5. La posibilidad restante es que el primer carácter sea >. Entonces, debemos pasar al estado 6 y decidir, en base al siguiente carácter, si el lexema es \geq (si vemos a continuación el signo =), o sólo > (con cualquier otro carácter). Observe que, si en el estado 0 vemos cualquier carácter además de <, = o >, no es posible que estemos viendo un lexema oprel, por lo que no utilizaremos este diagrama de transición de estados.

Reconocimiento de las palabras reservadas y los identificadores

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

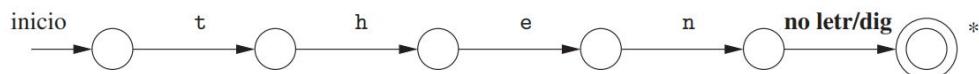


Hay dos formas en las que podemos manejar las palabras reservadas que parecen identificadores:

1. Instalar las palabras reservadas en la tabla de símbolos desde el principio. Un campo de la entrada en la tabla de símbolos indica que estas cadenas nunca serán identificadores ordinarios, y nos dice qué token representan. Al encontrar un identificador, una llamada a instalarID lo coloca en la tabla de símbolos, si no se encuentra ahí todavía, y devuelve un apuntador a la entrada en la tabla de símbolos para

el lexema que se encontró. La función obtenerToken examina la entrada en la tabla de símbolos para el lexema encontrado, y devuelve el nombre de token que la tabla de símbolos indique que representa este lexema; ya sea id o uno de los tokens de palabra clave que se instaló en un principio en la tabla.

2. Crear diagramas de transición de estados separados para cada palabra clave. Si adoptamos este método, entonces debemos dar prioridad a los tokens, para que los tokens de palabra reservada se reconozcan de preferencia en vez de id, cuando el lexema coincide con ambos patrones.



Finalización del bosquejo

El diagrama de transición de estados para el token numero se muestra en la figura 3.16, y es hasta ahora el diagrama más complejo que hemos visto. Empezando en el estado 12, si vemos un dígito pasamos al estado 13. En ese estado podemos leer cualquier número de dígitos adicionales. No obstante, si vemos algo que no sea un dígito o un punto, hemos visto un número en forma de entero; 123 es un ejemplo. Para manejar ese caso pasamos al estado 20, en donde devolvemos el token numero y un apuntador a una tabla de constantes en donde se introduce el lexema encontrado. Esta mecánica no se muestra en el diagrama, pero es análoga a la forma en la que manejamos los identificadores.

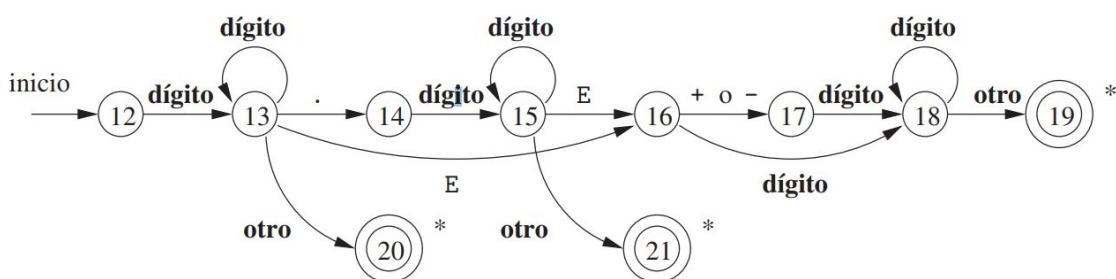


Figura 3.16: Un diagrama de transición para los números sin signo

Si en vez de ello vemos un punto en el estado 13, entonces tenemos una “fracción opcional”. Pasamos al estado 14, y buscamos uno o más dígitos adicionales; el estado 15 se utiliza para este fin. Si vemos una E, entonces tenemos un “exponente opcional”, cuyo reconocimiento es trabajo de los estados 16 a 19. Si en el estado 15 vemos algo que no sea una E o un dígito, entonces hemos llegado al final de la fracción, no hay exponente y devolvemos el lexema encontrado, mediante el estado 21.

El diagrama de transición de estados final, que se muestra en la figura 3.17, es para el espacio en blanco. En ese diagrama buscamos uno o más caracteres de “espacio en blanco”, representados por delim en ese diagrama; por lo general estos caracteres son los espacios, tabuladores, caracteres de nueva línea y tal vez otros caracteres que el diseño del lenguaje no considere como parte de algún token.

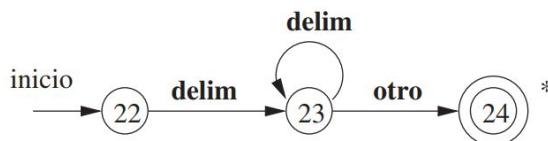


Figura 3.17: Un diagrama de transición para el espacio en blanco

Observe que, en el estado 24, hemos encontrado un bloque de caracteres de espacio en blanco consecutivos, seguidos de un carácter que no es espacio en blanco. Regresemos la entrada para que empiece en el carácter que no es espacio en blanco, pero no regresamos nada al analizador sintáctico, sino que debemos reiniciar el proceso del análisis léxico después del espacio en blanco.

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

Una instrucción switch con base en el valor de estado nos lleva al código para cada uno de los posibles estados, en donde encontramos la acción de ese estado. A menudo, el código para un estado es en sí una instrucción switch o una bifurcación de varias vías que determina el siguiente estado mediante el proceso de leer y examinar el siguiente carácter de entrada.

El generador de analizadores léxicos Lex

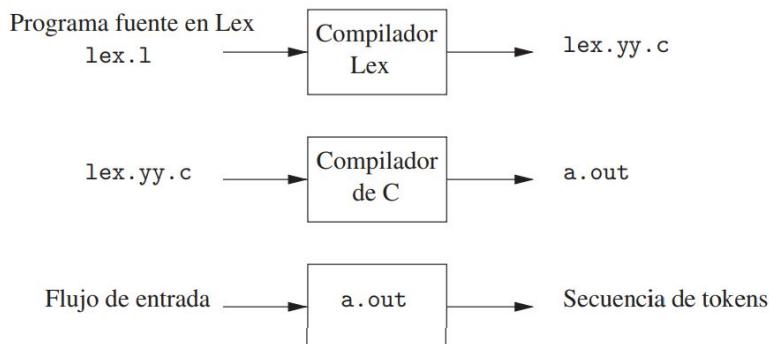
Nos permite especificar un analizador léxico mediante la especificación de expresiones regulares para describir patrones de los tokens. La notación de entrada para la herramienta Lex se conoce como el lenguaje Lex, y la herramienta en sí es el compilador Lex. El compilador Lex transforma los patrones de entrada en un diagrama de transición y genera código, en un archivo llamado lex.yy.c, que simula este diagrama de transición. La mecánica de cómo ocurre esta traducción de expresiones regulares a diagramas de transición es el tema de las siguientes secciones; aquí sólo aprenderemos acerca del lenguaje Lex.

Uso de Lex

Un archivo de entrada, al que llamaremos lex.l, está escrito en el lenguaje Lex y describe el analizador léxico que se va a generar. El compilador Lex transforma a lex.l en un programa en C, en un archivo que siempre se llama lex.yy.c. El compilador de C compila este archivo en un archivo llamado a.out, como de costumbre. La salida del compilador



de C es un analizador léxico funcional, que puede recibir un flujo de caracteres de entrada y producir una cadena de tokens



Estructura de los programas en Lex

Un programa en Lex tiene la siguiente forma:

```
declaraciones
%%
reglas de traducción
%%
funciones auxiliares
```

La sección de declaraciones incluye las declaraciones de variables, constantes de manifiesto (identificadores que se declaran para representar a una constante; por ejemplo, el nombre de un token) y definiciones regulares.

Cada una de las reglas de traducción tiene la siguiente forma

:

```
Patrón { Acción }
```

Cada patrón es una expresión regular, la cual puede usar las definiciones regulares de la sección de declaraciones. Las acciones son fragmentos de código, por lo general, escritos en C, aunque se han creado muchas variantes de Lex que utilizan otros lenguajes. La tercera sección contiene las funciones adicionales que se utilizan en las acciones. De manera alternativa, estas funciones pueden compilarse por separado y cargarse con el analizador léxico. El analizador léxico que crea Lex trabaja en conjunto con el analizador sintáctico de la siguiente manera. Cuando el analizador sintáctico llama al analizador léxico, éste empieza a leer el resto de su entrada, un carácter a la vez, hasta que encuentra el prefijo más largo de la entrada que coincide con uno de los patrones P_i . Después ejecuta la acción asociada A_i . Por lo general, A_i regresará al analizador sintáctico, pero si no lo hace (tal vez debido a que P_i describe espacio en blanco o comentarios), entonces el analizador léxico procede a buscar lexemas adicionales, hasta que una de las acciones correspondientes provoque un retorno al analizador sintáctico. El analizador léxico devuelve un solo valor, el nombre del token, al

analizador sintáctico, pero utiliza la variable entera compartida `yylval` para pasarle información adicional sobre el lexema encontrado, si es necesario.

Resolución de conflictos en Lex

Nos hemos referido a las dos reglas que utiliza Lex para decidir acerca del lexema apropiado a seleccionar, cuando varios prefijos de la entrada coinciden con uno o más patrones:

1. Preferir siempre un prefijo más largo a uno más corto.
2. Si el prefijo más largo posible coincide con dos o más patrones, preferir el patrón que se lista primero en el programa en Lex.

El operador adelantado

Lex lee de manera automática un carácter adelante del último carácter que forma el lexema seleccionado, y después regresa la entrada para que sólo se consuma el propio lexema de la entrada. No obstante, algunas veces puede ser conveniente que cierto patrón coincida con la entrada, sólo cuando vaya seguido de ciertos caracteres más. De ser así, tal vez podamos utilizar la barra diagonal en un patrón para indicar el final de la parte del patrón que coincide con el lexema. Lo que va después de / es un patrón adicional que se debe relacionar antes de poder decidir que vimos el token en cuestión, pero que lo que coincide con este segundo patrón no forma parte del lexema.

Autómatas finitos

En el corazón de la transición se encuentra el formalismo conocido como autómatas finitos. En esencia, estos consisten en gráficos como los diagramas de transición de estados, con algunas diferencias:

1. Los autómatas finitos son reconocedores; sólo dicen "sí" o "no" en relación con cada posible cadena de entrada.
2. Los autómatas finitos pueden ser de dos tipos:
 - (a) Los autómatas finitos no deterministas (AFN) no tienen restricciones en cuanto a las etiquetas de sus líneas. Un símbolo puede etiquetar a varias líneas que surgen del mismo estado, y , la cadena vacía, es una posible etiqueta.
 - (b) Los autómatas finitos deterministas (AFD) tienen, para cada estado, y para cada símbolo de su alfabeto de entrada, exactamente una línea con ese símbolo que sale de ese estado.

Tanto los autómatas finitos deterministas como los no deterministas son capaces de reconocer los mismos lenguajes. De hecho, estos lenguajes son exactamente los mismos lenguajes, conocidos como lenguajes regulares, que pueden describir las expresiones regulares.

Autómatas finitos no deterministas

Un autómata finito no determinista (AFN) consiste en:

1. Un conjunto finito de estados S .
2. Un conjunto de símbolos de entrada Σ , el alfabeto de entrada. Suponemos que ϵ , que representa a la cadena vacía, nunca será miembro de Σ .
3. Una función de transición que proporciona, para cada estado y para cada símbolo en $\Sigma \cup \{\epsilon\}$, un conjunto de estados siguientes.
4. Un estado s_0 de S , que se distingue como el estado inicial.
5. Un conjunto de estados F , un subconjunto de S , que se distinguen como los estados aceptantes (o estados finales).

Podemos representar un AFN o AFD mediante un gráfico de transición, en donde los nodos son estados y los flancos Indecidibles representan a la función de transición. Hay un flanco Indecidable a , que va del estado s al estado t si, y sólo si t es uno de los estados siguientes para el estado s y la entrada a . Este gráfico es muy parecido a un diagrama de transición, excepto que:

- a) El mismo símbolo puede etiquetar flancos de un estado hacia varios estados distintos.
- b) Un flanco puede etiquetarse por ϵ , la cadena vacía, en vez de, o además de, los símbolos del alfabeto de entrada.

Tablas de transición

También podemos representar a un AFN mediante una tabla de transición, cuyas filas corresponden a los estados, y cuyas columnas corresponden a los símbolos de entrada y a ϵ . La entrada para un estado dado y la entrada es el valor de la función de transición que se aplica a esos argumentos. Si la función de transición no tiene información acerca de ese par estado-entrada, colocamos \emptyset en la tabla para ese estado.

Aceptación de las cadenas de entrada mediante los autómatas

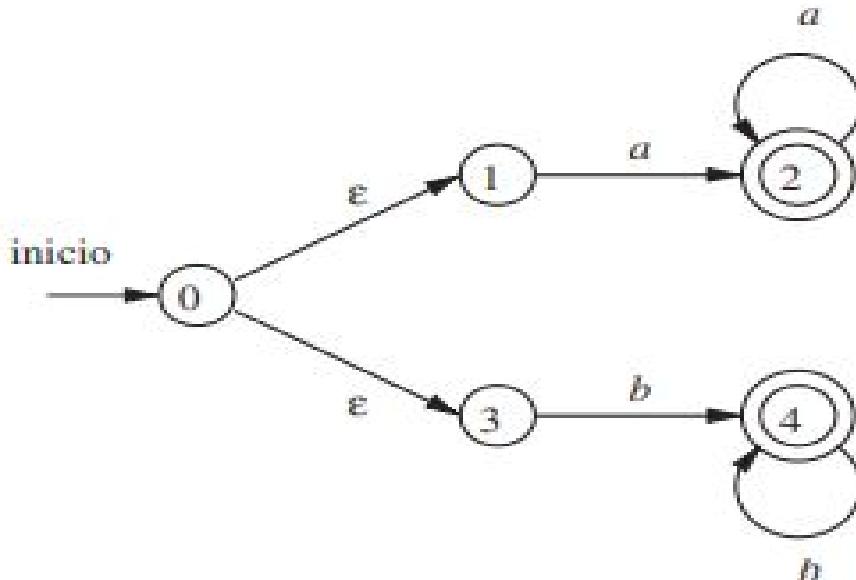
Un AFN acepta la cadena de entrada x si, y sólo si hay algún camino en el grafo de transición, desde el estado inicial hasta uno de los estados de aceptación, de forma que los



símbolos a lo largo del camino deletreen a x. Observe que las etiquetas ϵ a lo largo del camino se ignoran, ya que la cadena vacía no contribuye a la cadena que se construye a lo largo del camino.

Autómatas finitos deterministas

Un autómata finito determinista (AFD) es un caso especial de un AFN, en donde:



1. No hay movimientos en la entrada ϵ .
2. Para cada estado s y cada símbolo de entrada a, hay exactamente una línea que surge de s, Indecidible como a.

Si utilizamos una tabla de transición para representar a un AFD, entonces cada entrada es un solo estado. Por ende, podemos representar a este estado sin las llaves que usamos para formar los conjuntos.

Mientras que el AFN es una representación abstracta de un algoritmo para reconocer las cadenas de cierto lenguaje, el AFD es un algoritmo simple y concreto para reconocer cadenas. Sin duda es afortunado que cada expresión regular y cada AFN puedan convertirse en un AFD que acepte el mismo lenguaje, ya que es el AFD el que en realidad implementamos o simulamos al construir analizadores léxicos. El siguiente algoritmo muestra cómo aplicar un AFD a una cadena.

Conversión de un AFN a AFD

La idea general de la construcción de subconjuntos es que cada estado del AFD construido corresponde a un conjunto de estados del AFN. Después de leer la entrada $a_1a_2 \dots a_n$, el AFD se encuentra en el estado que corresponde al conjunto de estados que el AFN puede alcanzar, desde su estado inicial, siguiendo los caminos etiquetados como $a_1a_2 \dots a_n$. Es posible que el número de estados del AFD sea exponencial en el número de estados del AFN, lo cual podría provocar dificultades al tratar de implementar este AFD. No obstante,



parte del poder del método basado en autómatas para el análisis léxico es que para los lenguajes reales, el AFN y el AFD tienen aproximadamente el mismo número de estados, y no se ve el comportamiento exponencial.

Algoritmo 3.20: La construcción de subconjuntos de un AFD, a partir de un AFN.

ENTRADA: Un AFN N.

SALIDA: Un AFD D que acepta el mismo lenguaje que N.

MÉTODO: Nuestro algoritmo construye una tabla de transición Dtran para D. Cada estado de D es un conjunto de estados del AFN, y construimos Dtran para que D pueda simular “en paralelo” todos los posibles movimientos que N puede realizar sobre una cadena de entrada dada. Nuestro primer problema es manejar las transiciones de N en forma apropiada. En la figura 3.31 vemos las definiciones de varias funciones que describen cálculos básicos en los estados de N que son necesarios en el algoritmo. Observe que s es un estado individual de N, mientras que T es un conjunto de estados de N.

| OPERACIÓN | DESCRIPCIÓN |
|--------------------------|--|
| ϵ -cerradura(s) | Conjunto de estados del AFN a los que se puede llegar desde el estado s del AFN, sólo en las transiciones ϵ . |
| ϵ -cerradura(T) | Conjunto de estados del AFN a los que se puede llegar desde cierto estado s del AFN en el conjunto T, sólo en las transiciones ϵ ; = $\bigcup_{s \in T} \epsilon\text{-cerradura}(s)$. |
| mover(T, a) | Conjunto de estados del AFN para los cuales hay una transición sobre el símbolo de entrada a, a partir de cierto estado s en T. |

Figura 3.31: Operaciones sobre los estados del AFN

Debemos explorar esos conjuntos de estados en los que puede estar N después de ver cierta cadena de entrada. Como base, antes de leer el primer símbolo de entrada, N puede estar en cualquiera de los estados de ϵ -cerradura(s0), en donde s0 es su estado inicial. Para la inducción, suponga que N puede estar en el conjunto de estados T después de leer la cadena de entrada x. Si a continuación lee la entrada a, entonces N puede pasar de inmediato a cualquiera de los estados en mover(T, a). No obstante, después de leer a también podría realizar varias transiciones ; por lo tanto, N podría estar en cualquier estado de ϵ -cerradura(mover(T, a)) después de leer la entrada xa. Siguiendo estas ideas, la construcción del conjunto de estados de D, Destados, y su función de transición Dtran.

La estructura del analizador generado

La figura 3.49 presenta las generalidades acerca de la arquitectura de un analizador léxico generado por Lex. El programa que sirve como analizador léxico incluye un programa fijo que simula a un autómata; en este punto dejamos abierta la decisión de si el autómata es determinista o no. El resto del analizador léxico consiste en componentes que se crean a partir del programa Lex, por el mismo Lex.

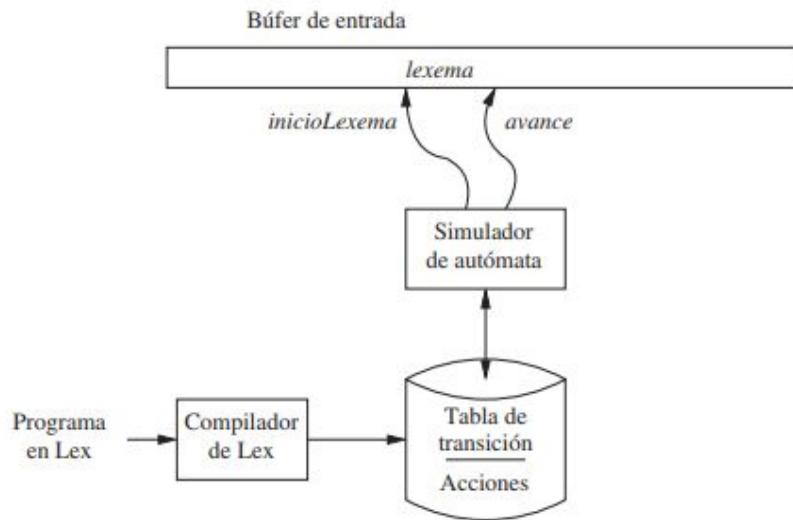


Figura 3.49: Un programa en Lex se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

Coincidencia de patrones con base en los AFNs

Si el analizador léxico simula un AFN como el de la figura 3.52, entonces debe leer la entrada que empieza en el punto de su entrada, al cual nos hemos referido como *inicioLexema*. A medida que el apuntador llamado *avance* avanza hacia delante en la entrada, calcula el conjunto de estados en los que se encuentra en cada punto.

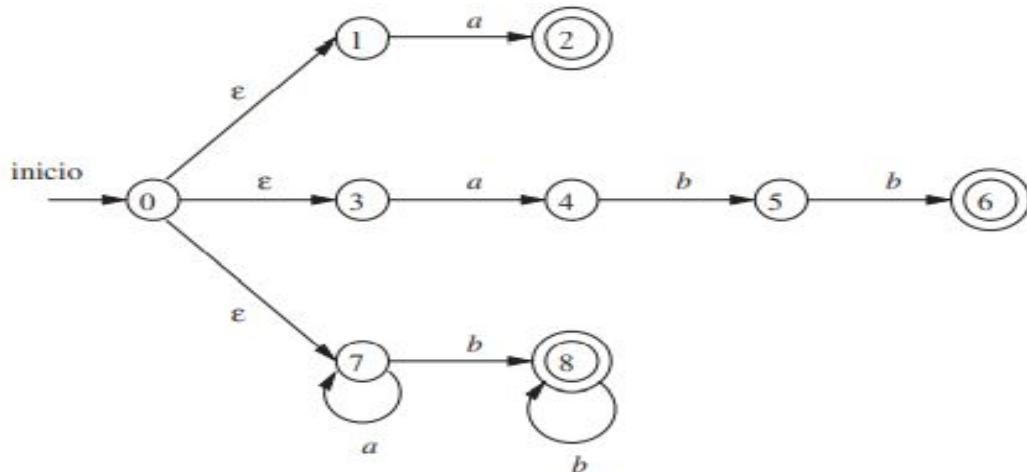


Figura 3.52: AFN combinado

En algún momento, la simulación del AFN llega a un punto en la entrada en donde no hay siguientes estados. En ese punto, no hay esperanza de que cualquier prefijo más largo de la entrada haga que el AFN llegue a un estado de aceptación; en vez de ello, el conjunto de estados siempre estará vacío. Por ende, estamos listos para decidir sobre el prefijo más largo que sea un lexema que coincide con cierto patrón.



Buscamos hacia atrás en la secuencia de conjuntos de estados, hasta encontrar un conjunto que incluya uno o más estados de aceptación. Si hay varios estados de aceptación en ese conjunto, elegimos el que esté asociado con el primer patrón π en la lista del programa en Lex. Retrocedemos el apuntador avance hacia el final del lexema, y realizamos la acción A_i asociada con el patrón π .

AFDs para analizadores léxicos

Dentro de cada estado del AFD, si hay uno o más estados aceptantes del AFN, se determina el primer patrón cuyo estado aceptante se representa, y ese patrón se convierte en la salida del estado AFD.

Implementación del operador de preanálisis

Al convertir el patrón r_1/r_2 en un AFN, tratamos al / como si fuera , por lo que en realidad no buscamos un / en la entrada. No obstante, si el AFN reconoce un prefijo xy del búfer de entrada, de forma que coincide con esta expresión regular, el final del lexema no es en donde el AFN entró a su estado de aceptación. En vez de ello, el final ocurre cuando el AFN entra a un estado s tal que:

1. s tenga una transición en el / imaginario.
2. Hay un camino del estado inicial del AFN hasta el estado s , que deletrea a x .
3. Hay un camino del estado s al estado de aceptación que deletrea a y .
4. x es lo más largo posible para cualquier xy que cumpla con las condiciones 1-3.

Si sólo hay un estado de transición en el / imaginario en el AFN, entonces el final del lexema ocurre cuando se entra a este estado por última vez, como se ilustra en el siguiente ejemplo. Si el AFN tiene más de un estado de transición en el / imaginario, entonces el problema general de encontrar el estado s actual se dificulta mucho más.

Optimización de los buscadores por concordancia de patrones basados en AFD

En esa sección presentaremos tres algoritmos que se utilizan para implementar y optimizar buscadores por concordancia de patrones, construidos a partir de expresiones regulares.

1. El primer algoritmo es útil en un compilador de Lex, ya que construye un AFD directamente a partir de una expresión regular, sin construir un AFN intermedio. Además, el AFD resultante puede tener menos estados que el AFD que se construye mediante un AFN.
2. El segundo algoritmo disminuye al mínimo el número de estados de cualquier AFD, mediante la combinación de los estados que tienen el mismo comportamiento a futuro.



El algoritmo en sí es bastante eficiente, pues se ejecuta en un tiempo $O(n \log n)$, en donde n es el número de estados del AFD.

3. El tercer algoritmo produce representaciones más compactas de las tablas de transición que la tabla estándar bidimensional.

Estados significativos de un AFN

Durante la construcción de subconjuntos, pueden identificarse dos conjuntos de estados del AFN (que se tratan como si fueran el mismo conjunto) si:

1. Tienen los mismos estados significativos.
2. Ya sea que ambos tengan estados de aceptación, o ninguno.

El AFN construido sólo tiene un estado de aceptación, pero éste, que no tiene transiciones de salida, no es un estado significativo. Al concatenar un único marcador final # derecho con una expresión regular r , proporcionamos al estado de aceptación para r una transición sobre #, con lo cual lo marcamos como un estado significativo del AFN para $(r)\#$. En otras palabras, al usar la expresión regular aumentada $(r)\#$, podemos olvidarnos de los estados de aceptación a medida que procede la construcción de subconjuntos; cuando se completa la construcción, cualquier estado con una transición sobre # debe ser un estado de aceptación.

Los estados significativos del AFN corresponden directamente a las posiciones en la expresión regular que contienen símbolos del alfabeto. Como pronto veremos, es conveniente presentar la expresión regular mediante su árbol sintáctico, en donde las hojas corresponden a los operandos y los nodos interiores corresponden a los operadores. A un nodo interior se le llama nodo-concat, nodo-o o nodo-asterisco si se etiqueta mediante el operador de concatenación (punto), el operador de unión |, o el operador *, respectivamente.

Funciones calculadas a partir del árbol sintáctico

Para construir un AFD directamente a partir de una expresión regular, construimos su árbol sintáctico y después calculamos cuatro funciones: anulable, primerapos, ultimapos y siguientepos, las cuales se definen a continuación. Cada definición se refiere al árbol sintáctico para una expresión regular aumentada $(r)\#$ específica.

1. anulable(n) es verdadera para un nodo n del árbol sintáctico si, y sólo si, la subexpresión representada por n tiene a en su lenguaje. Es decir, la subexpresión puede "hacerse nula" o puede ser la cadena vacía, aun cuando pueda representar también a otras cadenas.

2. $\text{primerapos}(n)$ es el conjunto de posiciones en el subárbol con raíz en n , que corresponde al primer símbolo de por lo menos una cadena en el lenguaje de la subexpresión con raíz en n .

3. $\text{ultimapos}(n)$ es el conjunto de posiciones en el subárbol con raíz en n , que corresponde al último símbolo de por lo menos una cadena en el lenguaje de la subexpresión con raíz en n .

4. $\text{siguientenpos}(p)$, para una posición p , es el conjunto de posiciones q en todo el árbol sintáctico, de tal forma que haya cierta cadena $x = a_1a_2 \dots a_n$ en una $L((r)\#)$ tal que para cierta i , haya una forma de explicar la membresía de x en $L((r)\#)$, haciendo que a_i coincida con la posición p del árbol sintáctico y a_{i+1} con la posición q

Cálculo de anulable, primerapos y ultimapos

Por último, necesitamos ver cómo calcular siguientenpos . Sólo hay dos formas en que podemos hacer que la posición de una expresión regular siga a otra.

1. Si n es un nodo-concat con el hijo izquierdo c_1 y con el hijo derecho c_2 , entonces para cada posición i en $\text{ultimapos}(c_1)$, todas las posiciones en $\text{primerapos}(c_2)$ se encuentran en $\text{siguientenpos}(i)$.

2. Si n es un nodo-asterisco e i es una posición en $\text{ultimapos}(n)$, entonces todas las posiciones en $\text{primerapos}(n)$ se encuentran en $\text{siguientenpos}(i)$.

Cálculo de siguientepos

Por último, necesitamos ver cómo calcular siguientepos . Sólo hay dos formas en que podemos hacer que la posición de una expresión regular siga a otra. 1. Si n es un nodo-concat con el hijo izquierdo c_1 y con el hijo derecho c_2 , entonces para cada posición i en $\text{ultimapos}(c_1)$, todas las posiciones en $\text{primerapos}(c_2)$ se encuentran en $\text{siguientepos}(i)$. 2. Si n es un nodo-asterisco e i es una posición en $\text{ultimapos}(n)$, entonces todas las posiciones en $\text{primerapos}(n)$ se encuentran en $\text{siguientepos}(i)$.

Conversión directa de una expresión regular a un AFD

Algoritmo 3.36: Construcción de un AFD a partir de una expresión regular r .

ENTRADA: Una expresión regular r .

SALIDA: Un AFD D que reconoce a $L(r)$.

MÉTODO:



1. Construir un árbol sintáctico T a partir de la expresión regular aumentada $(r)^*$.
2. Calcular anulable, primerapos, ultimapos y siguientepos para T, mediante los métodos de las secciones 3.9.3 y 3.9.4.
3. Construir Destados, el conjunto de estados del AFD D, y Dtran, la función de transición para D, mediante el procedimiento de la figura 3.62. Los estados de D son estados de posiciones en T. Al principio, cada estado está “sin marca”, y un estado se “marca” justo antes de que consideremos sus transiciones de salida. El estado inicial de D es primerapos(n_0), en donde el nodo n_0 es la raíz de T. Los estados de aceptación son los que contienen la posición para el símbolo de marcador final #.

Intercambio de tiempo por espacio en la simulación de un AFD

La manera más simple y rápida de representar la función de transición de un AFD es una tabla bidimensional indexada por estados y caracteres. Dado un estado y el siguiente carácter de entrada, accedemos al arreglo para encontrar el siguiente estado y cualquier acción especial que debemos tomar; por ejemplo, devolver un token al analizador sintáctico. Como un analizador léxico ordinario tiene varios cientos de estados en su AFD e involucra al alfabeto ASCII de 128 caracteres de entrada, el arreglo consume menos de un megabyte.

No obstante, los compiladores también aparecen en dispositivos muy pequeños, en donde hasta un megabyte de memoria podría ser demasiado. Para tales situaciones, existen muchos métodos que podemos usar para compactar la tabla de transición. Por ejemplo, podemos representar cada estado mediante una lista de transiciones (es decir, pares carácter-estado) que se terminen mediante un estado predeterminado, el cual debe elegirse para cualquier carácter de entrada que no se encuentre en la lista. Si elegimos como predeterminado el siguiente estado que ocurra con más frecuencia, a menudo podemos reducir la cantidad de almacenamiento necesario por un factor extenso.

Hay una estructura de datos más sutil que nos permite combinar la velocidad del acceso a los arreglos con la compresión de listas con valores predeterminados. Podemos considerar esta estructura como cuatro arreglos. El arreglo base se utiliza para determinar la ubicación base de las entradas para el estado s, que se encuentran en los arreglos siguiente y comprobacion. El arreglo predeterminado se utiliza para determinar una ubicación base alternativa, si el arreglo comprobacion nos indica que el que proporciona base[s] es inválido.

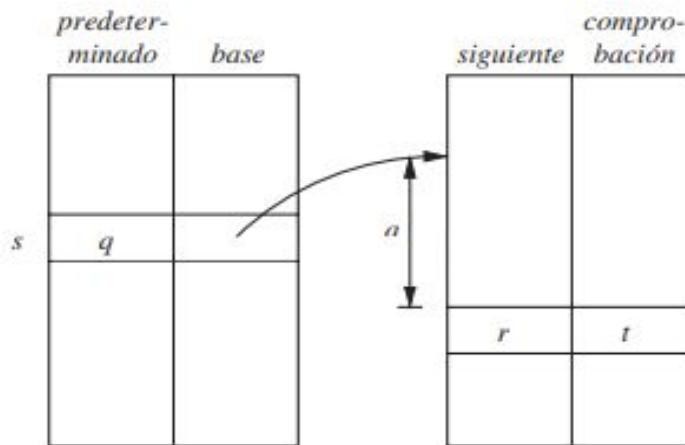


Figura 3.66: Estructura de datos para representar tablas de transición

Para calcular siguienteEstado(s, a), la transición para el estado s con la entrada a , examinamos las entradas siguiente y comprobacion en la ubicación $l = base[s]+a$, en donde el carácter a se trata como entero, supuestamente en el rango de 0 a 127. Si $comprobacion[l] = s$, entonces esta entrada es válida y el siguiente estado para el estado s con la entrada a es $siguiente[l]$. Si $comprobacion[l] \neq s$, entonces determinamos otro estado $t = predeterminado[s]$ y repetimos el proceso, como si t fuera el estado actual. De manera más formal, la función siguienteEstado se define así:

```
int siguienteEstado(s, a) {
    if ( comprobacion[base[s]+a] == s ) return siguiente[base[s] + a];
    else return siguienteEstado(predeterminado[s], a);
}
```

(“Análisis Sintáctico (parte 1)”)

De hecho, el árbol de análisis sintáctico no necesita construirse en forma explícita, ya que las acciones de comprobación y traducción pueden intercalarse con el análisis sintáctico, como veremos más adelante. Por ende, el analizador sintáctico y el resto de la interfaz de usuario podrían implementarse sin problemas mediante un solo módulo.

Una gramática proporciona una especificación sintáctica precisa, pero fácil de entender, de un lenguaje de programación.



A partir de ciertas clases de gramáticas, podemos construir de manera automática un analizador sintáctico eficiente que determine la estructura sintáctica de un programa fuente. Como beneficio colateral, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y puntos problemáticos que podrían haberse pasado por alto durante la fase inicial del diseño del lenguaje.

La estructura impartida a un lenguaje mediante una gramática diseñada en forma apropiada es útil para traducir los programas fuente en código objeto correcto, y para detectar errores.

Una gramática permite que un lenguaje evolucione o se desarrolle en forma iterativa, agregando nuevas construcciones para realizar nuevas tareas. Estas nuevas construcciones pueden integrarse con más facilidad en una implementación que siga la estructura gramatical del lenguaje.

Existen tres tipos generales de analizadores para las gramáticas: universales, descendentes y ascendentes. Los métodos universales de análisis sintáctico como el algoritmo de Cocke-Younger-Kasami y el algoritmo de Earley pueden analizar cualquier gramática (vea las notas bibliográficas). Sin embargo, estos métodos generales son demasiado ineficientes como para usarse en la producción de compiladores.

Los métodos que se utilizan, por lo regular, en los compiladores pueden clasificarse como descendentes o ascendentes. Los métodos descendentes y ascendentes más eficientes sólo funcionan para subclases de gramáticas, pero varias de estas clases, en especial las gramáticas LL y LR, son lo bastante expresivas como para describir la mayoría de las construcciones sintácticas en los lenguajes de programación modernos. Los errores léxicos incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador tamañoElipse en vez de tamañoElipse, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.

Los errores sintácticos incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, “{” o “}”. Como otro ejemplo, en C o Java, la aparición de una instrucción case sin una instrucción switch que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).

Otra de las razones para enfatizar la recuperación de los errores durante el análisis sintáctico es que muchos errores parecen ser sintácticos, sea cual fuere su causa, y se exponen cuando el análisis sintáctico no puede continuar. Algunos errores semánticos, como los conflictos entre los tipos, también pueden detectarse con eficiencia; sin



embargo, la detección precisa de los errores semánticos y lógicos en tiempo de compilación es, por lo general, una tarea difícil.

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

Representación de gramáticas

Las construcciones que empiezan con palabras clave como while o int son muy fáciles de analizar, ya que la palabra clave guía la elección de la producción gramatical que debe aplicarse para hacer que coincida con la entrada. Por lo tanto, nos concentraremos en las expresiones, que representan un reto debido a la asociatividad y la precedencia de operadores. La asociatividad y la precedencia se resuelvan en la siguiente gramática, que es similar a las que utilizamos en el capítulo 2 para describir expresiones, términos y factores. E representa a las expresiones que consisten en términos separados por los signos +, T representa a los términos que consisten en factores separados por los signos *, y F representa a los factores que pueden ser expresiones entre paréntesis o identificadores:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Manejo de los errores sintácticos

El resto de esta sección considera la naturaleza de los errores sintácticos y las estrategias generales para recuperarse de ellos. Dos de estas estrategias, conocidas como recuperaciones en modo de pánico y a nivel de frase, se describirán con más detalle junto con los métodos específicos de análisis sintáctico. Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificaría en forma considerable. No obstante, se espera que un compilador ayude al programador a localizar y rastrear los errores que, de manera inevitable, se infiltran en los programas, a pesar de los mejores esfuerzos del programador. Aunque parezca increíble, son pocos los lenguajes que se diseñan teniendo en mente el manejo de errores, aun cuando éstos son tan comunes. Nuestra civilización sería radicalmente distinta si los lenguajes hablados tuvieran los mismos requerimientos en cuanto a precisión sintáctica que los lenguajes de computadora. La mayoría de las especificaciones de los lenguajes de programación no describen la forma en que un compilador debe responder a los errores; el manejo de los mismos es responsabilidad del diseñador del compilador. La planeación del manejo de los errores desde el principio

puede simplificar la estructura de un compilador y mejorar su capacidad para manejar los errores. Los errores de programación comunes pueden ocurrir en muchos niveles distintos.

- Los errores léxicos incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador tamanioElipce en vez de tamanioElipse, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.
- Los errores sintácticos incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, “{” o “}”. Como otro ejemplo, en C o Java, la aparición de una instrucción case sin una instrucción switch que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).
- Los errores semánticos incluyen los conflictos de tipos entre los operadores y los operandos. Un ejemplo es una instrucción return en un método de Java, con el tipo de resultado void.
- Los errores lógicos pueden ser cualquier cosa, desde un razonamiento incorrecto del programador en el uso (en un programa en C) del operador de asignación =, en vez del operador de comparación ==. El programa que contenga = puede estar bien formado; sin embargo, tal vez no refleje la intención del programador.

La precisión de los métodos de análisis sintáctico permite detectar los errores sintácticos con mucha eficiencia. Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan un error lo más pronto posible; es decir, cuando el flujo de tokens que proviene del analizador léxico no puede seguirse analizando de acuerdo con la gramática para el lenguaje. Dicho en forma más precisa, tienen la propiedad de prefijo viable, lo cual significa que detectan la ocurrencia de un error tan pronto como ven un prefijo de la entrada que no puede completarse para formar una cadena válida en el lenguaje. Otra de las razones para enfatizar la recuperación de los errores durante el análisis sintáctico es que muchos errores parecen ser sintácticos, sea cual fuere su causa, y se exponen cuando el análisis sintáctico no puede continuar. Algunos errores semánticos, como los conflictos entre los tipos, también pueden detectarse con eficiencia; sin embargo, la detección precisa de los errores semánticos y lógicos en tiempo de compilación es, por lo general, una tarea difícil. El mango de errores en un analizador sintáctico tiene objetivos que son simples de declarar, pero difíciles de llevar a cabo:

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

Por fortuna, los errores comunes son simples, y a menudo basta con un mecanismo simple para su manejo. ¿De qué manera un mango de errores debe reportar la presencia de un error? Como mínimo, debe reportar el lugar en el programa fuente en donde se detectó un error, ya que hay una buena probabilidad de que éste en sí haya ocurrido en uno de los pocos tokens anteriores. Una estrategia común es imprimir la línea del problema con un apuntador a la posición en la que se detectó el error.

Estrategias para recuperarse de los errores

Una vez que se detecta un error, ¿cómo debe recuperarse el analizador sintáctico? Aunque no hay una estrategia que haya demostrado ser aceptable en forma universal, algunos métodos pueden aplicarse en muchas situaciones. El método más simple es que el analizador sintáctico termine con un mensaje de error informativo cuando detecte el primer error. A menudo se descubren errores adicionales si el analizador sintáctico puede restaurarse a sí mismo, a un estado en el que pueda continuar el procesamiento de la entrada, con esperanzas razonables de que un mayor procesamiento proporcione información útil para el diagnóstico. Si los errores se apilan, es mejor para el compilador desistir después de exceder cierto límite de errores, que producir una molesta avalancha de errores “falsos”. El resto de esta sección se dedica a las siguientes estrategias de recuperación de los errores: modo de pánico, nivel de frase, producciones de errores y corrección global.

Recuperación en modo de pánico

Con este método, al describir un error el analizador sintáctico descarta los símbolos de entrada, uno a la vez, hasta encontrar un conjunto designado de tokens de sincronización. Por lo general, los tokens de sincronización son delimitadores como el punto y coma o }, cuya función en el programa fuente es clara y sin ambigüedades. El diseñador del compilador debe seleccionar los tokens de sincronización apropiados para el lenguaje fuente. Aunque la corrección en modo de pánico a menudo omite una cantidad considerable de entrada sin verificar errores adicionales, tiene la ventaja de ser simple y, a diferencia de ciertos métodos que consideraremos más adelante, se garantiza que no entrará en un ciclo infinito.

Recuperación a nivel de frase

Al descubrir un error, un analizador sintáctico puede realizar una corrección local sobre la entrada restante; es decir, puede sustituir un prefijo de la entrada restante por alguna cadena que le permita continuar. Una corrección local común es sustituir una coma por un punto y coma, eliminar un punto y coma extraño o insertar un punto y coma faltante. La elección de la corrección local se deja al diseñador del compilador. Desde luego que debemos tener cuidado de elegir sustituciones que no nos lleven hacia ciclos infinitos, como sería, por ejemplo, si siempre insertáramos algo en la entrada adelante del símbolo de entrada actual. La sustitución a nivel de frase se ha utilizado en varios



compiladores que reparan los errores, ya que puede corregir cualquier cadena de entrada. Su desventaja principal es la dificultad que tiene para arreglárselas con situaciones en las que el error actual ocurre antes del punto de detección.

Producciones de errores

Al anticipar los errores comunes que podríamos encontrar, podemos aumentar la gramática para el lenguaje, con producciones que generen las construcciones erróneas. Un analizador sintáctico construido a partir de una gramática aumentada por estas producciones de errores detecta los errores anticipados cuando se utiliza una producción de error durante el análisis sintáctico. Así, el analizador sintáctico puede generar diagnósticos de error apropiados sobre la construcción errónea que se haya reconocido en la entrada.

Corrección global

Lo ideal sería que un compilador hiciera la menor cantidad de cambios en el procesamiento de una cadena de entrada incorrecta. Hay algoritmos para elegir una secuencia mínima de cambios, para obtener una corrección con el menor costo a nivel global. Dada una cadena de entrada incorrecta x y una gramática G , estos algoritmos buscarán un árbol de análisis sintáctico para una cadena y relacionada, de tal forma que el número de inserciones, eliminaciones y modificaciones de los tokens requeridos para transformar x en y sea lo más pequeño posible. Por desgracia, estos métodos son en general demasiado costosos para implementarlos en términos de tiempo y espacio, por lo cual estas técnicas sólo son de interés teórico en estos momentos. Hay que observar que un programa casi correcto tal vez no sea lo que el programador tenía en mente. Sin embargo, la noción de la corrección con el menor costo proporciona una norma para evaluar las técnicas de recuperación de los errores, la cual se ha utilizado para buscar cadenas de sustitución óptimas para la recuperación a nivel de frase.

Gramáticas libres de contexto

Si utilizamos una variable sintáctica $instr$ para denotar las instrucciones, y una variable $expr$ para denotar las expresiones, la siguiente producción:

$$instr \rightarrow if (expr) instr \text{ else } instr$$

especifica la estructura de esta forma de instrucción condicional. Entonces, otras producciones definen con precisión lo que es una $expr$ y qué más puede ser una $instr$. En esta sección repasaremos la definición de una gramática libre de contexto y presentaremos la terminología para hablar acerca del análisis sintáctico. En especial, la noción de derivaciones es muy útil para hablar sobre el orden en el que se aplican las producciones durante el análisis sintáctico.

La definición formal de una gramática libre de contexto

1. Los terminales son los símbolos básicos a partir de los cuales se forman las cadenas. El término “nombre de token” es un sinónimo de “terminal”; con frecuencia usaremos la palabra “token” en vez de terminal, cuando esté claro que estamos hablando sólo sobre el nombre del token. Asumimos que las terminales son los primeros componentes de los tokens que produce el analizador léxico. Los terminales son las palabras reservadas if y else, y los símbolos (“” y ””).
2. Los no terminales son variables sintácticas que denotan conjuntos de cadenas. Instr y expr son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.
3. En una gramática, un no terminal se distingue como el símbolo inicial, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero.
4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:
(a) Un no terminal, conocido como encabezado o lado izquierdo de la producción; esta producción define algunas de las cadenas denotadas por el encabezado. (b) El símbolo →. Algunas veces se ha utilizado ::= en vez de la flecha. (c) Un cuerpo o lado derecho, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

Convenciones de notación

Para evitar siempre tener que decir que “éstos son los terminales”, “éstos son los no terminales”, etcétera, utilizaremos las siguientes convenciones de notación para las gramáticas durante el resto de este libro:

1. Estos símbolos son terminales:
 - (a) Las primeras letras minúsculas del alfabeto, como a, b, c.
 - (b) Los símbolos de operadores como +, *, etcétera.
 - (c) Los símbolos de puntuación como paréntesis, coma, etcétera.
 - (d) Los dígitos 0, 1, ..., 9. (e) Las cadenas en negrita como id o if, cada una de las cuales representa un solo símbolo terminal.
2. Estos símbolos son no terminales:



- (a) Las primeras letras mayúsculas del alfabeto, como A, B, C.
- (b) La letra S que, al aparecer es, por lo general, el símbolo inicial.
- (c) Los nombres en cursiva y minúsculas, como expr o instr.
- (d) Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para las expresiones, los términos y los factores se representan a menudo mediante E, T y F, respectivamente.
3. Las últimas letras mayúsculas del alfabeto, como X, Y, Z, representan símbolos gramaticales; es decir, pueden ser no terminales o terminales.
 4. Las últimas letras minúsculas del alfabeto, como u, v, ..., z, representan cadenas de terminales (posiblemente vacías).
 5. Las letras griegas minúsculas α , β , γ , por ejemplo, representan cadenas (posiblemente vacías) de símbolos gramaticales. Por ende, una producción genérica puede escribirse como $A \rightarrow \alpha$, en donde A es el encabezado y α el cuerpo.
 6. Un conjunto de producciones $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_k$ con un encabezado común A (las llamaremos producciones A), puede escribirse como $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$. A α_1 , α_2 , ..., α_k les llamamos las alternativas para A.
 7. A menos que se indique lo contrario, el encabezado de la primera producción es el símbolo inicial.

Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura. Empezando con el símbolo inicial, cada paso de rescritura sustituye a un no terminal por el cuerpo de una de sus producciones. Esta vista derivacional corresponde a la construcción descendente de un árbol de análisis sintáctico, pero la precisión que ofrecen las derivaciones será muy útil cuando hablemos del análisis sintáctico ascendente. Como veremos, el análisis sintáctico ascendente se relaciona con una clase de derivaciones conocidas como derivaciones de “más a la derecha”, en donde el no terminal por la derecha se reescribe en cada paso.

Por ejemplo, considere la siguiente gramática, con un solo no terminal E, la cual agrega una producción $E \rightarrow -E$ a la gramática:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$$



La producción $E \rightarrow -E$ significa que si E denota una expresión, entonces $-E$ debe también denotar una expresión. La sustitución de una sola E por $-E$ se describirá escribiendo lo siguiente:

$$E \Rightarrow -E$$

lo cual se lee como “ E deriva a $-E$ ”. La producción $E \rightarrow (E)$ puede aplicarse para sustituir cualquier instancia de E en cualquier cadena de símbolos gramaticales por (E) ; por ejemplo, $E * E \Rightarrow (E) * E$ o $E * E \Rightarrow E * (E)$. Podemos tomar una sola E y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

A dicha secuencia de sustituciones la llamamos una derivación de $-(\mathbf{id})$ a partir de E . Esta derivación proporciona la prueba de que la cadena $-(\mathbf{id})$ es una instancia específica de una expresión. Para una definición general de la derivación, considere un no terminal A en la mitad de una secuencia de símbolos gramaticales, como en $\alpha A \beta$, en donde α y β son cadenas arbitrarias de símbolos gramaticales. Suponga que $A \rightarrow \gamma$ es una producción. Entonces, escribimos $\alpha A \beta \Rightarrow \alpha \gamma \beta$. El símbolo \Rightarrow significa, “se deriva en un paso”. Cuando una secuencia de pasos de derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ se rescribe como α_1 a α_n , decimos que α_1 deriva a α_n . Con frecuencia es conveniente poder decir, “deriva en cero o más pasos”. Para este fin, podemos usar el símbolo \Rightarrow^* . Así,

1. $\alpha \stackrel{*}{\Rightarrow} \alpha$, para cualquier cadena α .
2. Si $\alpha \stackrel{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \stackrel{*}{\Rightarrow} \gamma$.

De igual forma, \Rightarrow significa “deriva en uno o más pasos”. Si $S \Rightarrow^* \alpha$, en donde S es el símbolo inicial de una gramática G , decimos que α es una forma de frase de G . Observe que una forma de frase puede contener tanto terminales como no terminales, y puede estar vacía. Un enunciado de G es una forma de frase sin símbolos no terminales. El lenguaje generado por una gramática es su conjunto de



oraciones. Por ende, una cadena de terminales w está en $L(G)$, el lenguaje generado por G , si y sólo si w es un enunciado de G ($\alpha \Rightarrow^* w$). Un lenguaje que puede generarse mediante una gramática se considera un lenguaje libre de contexto. Si dos gramáticas generan el mismo lenguaje, se consideran como equivalentes.

Cada no terminal se sustituye por el mismo cuerpo en las dos derivaciones, pero el orden de las sustituciones es distinto. Para comprender la forma en que trabajan los analizadores sintácticos, debemos considerar las derivaciones en las que el no terminal que se va a sustituir en cada paso se elige de la siguiente manera:

1. En las derivaciones por la izquierda, siempre se elige el no terminal por la izquierda en cada de frase. Si $\alpha \Rightarrow \beta$ es un paso en el que se sustituye el no terminal por la izquierda en α , escribimos $\alpha \Rightarrow lm \beta$.
2. En las derivaciones por la derecha, siempre se elige el no terminal por la derecha; en este caso escribimos $\alpha \Rightarrow rm \beta$.

La derivación (4.8) es por la izquierda, por lo que puede rescribirse de la siguiente manera:

$$E \xrightarrow{lm} -E \xrightarrow{lm} -(E) \xrightarrow{lm} -(E+E) \xrightarrow{lm} -(\text{id}+E) \xrightarrow{lm} -(\text{id}+\text{id})$$

Si utilizamos nuestras convenciones de notación, cada paso por la izquierda puede escribirse como $wAy \Rightarrow lm w\delta y$, en donde w consiste sólo de terminales, $A \rightarrow \delta$ es la producción que se aplica, y y es una cadena de símbolos gramaticales. Para enfatizar que α deriva a β mediante una derivación por la izquierda, escribimos $\alpha \Rightarrow^* lm \beta$. Si $S \Rightarrow^* lm \alpha$, decimos que α es una forma de frase izquierda de la gramática en cuestión. Las análogas definiciones son válidas para las derivaciones por la derecha. A estas derivaciones se les conoce algunas veces como derivaciones canónicas.

Árboles de análisis sintáctico y derivaciones

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación.



Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama producto o frontera del árbol. Para ver la relación entre las derivaciones y los árboles de análisis sintáctico, considere cualquier derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$, en donde α_1 es un sólo no terminal A. Para cada forma de frase α_i en la derivación, podemos construir un árbol de análisis sintáctico cuyo producto sea α_i . El proceso es una inducción sobre i.

BASE: El árbol para $\alpha_1 = A$ es un solo nodo, etiquetado como A.

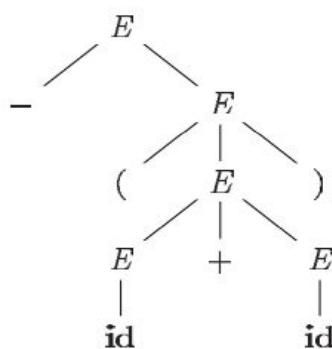
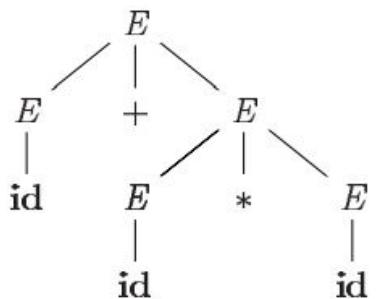


Figura 4.3: Árbol de análisis sintáctico para $-(\text{id} + \text{id})$

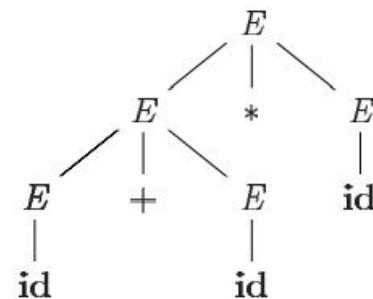
INDUCCIÓN: Suponga que ya hemos construido un árbol de análisis sintáctico con el producto $\alpha_{i-1} = X_1 X_2 \dots X_k$ (tenga en cuenta que, de acuerdo a nuestras convenciones de notación, cada símbolo gramatical X_i es un no terminal o un terminal). Suponga que α_i se deriva de α_{i-1} al sustituir X_j , un no terminal, por $\beta = Y_1 Y_2 \dots Y_m$. Es decir, en el i -ésimo paso de la derivación, la producción $X_j \rightarrow \beta$ se aplica a α_{i-1} para derivar $\alpha_i = X_1 X_2 \dots X_{j-1} \beta X_{j+1} \dots X_k$. Para modelar este paso de la derivación, buscamos la j -ésima hoja, partiendo de la izquierda en el árbol de análisis sintáctico actual. Esta hoja se etiqueta como X_j . A esta hoja le damos m hijos, etiquetados Y_1, Y_2, \dots, Y_m , partiendo de la izquierda. Como caso especial, si $m = 0$ entonces $\beta = \epsilon$, y proporcionamos a la j -ésima hoja un hijo etiquetado como ϵ .

Ambigüedad

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única qué árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con reglas para eliminar la ambigüedad, las cuales “descartan” los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.



(a)



(b)

Figura 4.5: Dos árboles de análisis sintáctico para **id+id*id**

Verificación del lenguaje generado por una gramática

Aunque los diseñadores de compiladores muy raras veces lo hacen para una gramática de lenguaje de programación completa, es útil poder razonar que un conjunto dado de producciones genera un lenguaje específico. Las construcciones problemáticas pueden estudiarse mediante la escritura de una gramática abstracta y concisa, y estudiando el lenguaje que genera. A continuación vamos a construir una gramática de este tipo, para instrucciones condicionales. Una prueba de que una gramática G genera un lenguaje L consta de dos partes: mostrar que todas las cadenas generadas por G están en L y, de manera inversa, que todas las cadenas en L pueden generarse sin duda mediante G .

Comparación entre gramáticas libres de contexto y expresiones regulares

Antes de dejar esta sección sobre las gramáticas y sus propiedades, establecemos que las gramáticas son una notación más poderosa que las expresiones regulares. Cada construcción que puede describirse mediante una expresión regular puede describirse mediante una gramática, pero no al revés. De manera alternativa, cada lenguaje regular es un lenguaje libre de contexto, pero no al revés. Por ejemplo, la expresión regular $(a|b)^*$ abb y la siguiente gramática:

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

describen el mismo lenguaje, el conjunto de cadenas de as y bs que terminan en abb.

Podemos construir de manera mecánica una gramática para reconocer el mismo lenguaje que un autómata finito no determinista (AFN). La gramática anterior se construyó a partir del AFN de la figura 3.24, mediante la siguiente construcción:

1. Para cada estado i del AFN, crear un no terminal A_i .
2. Si el estado i tiene una transición al estado j con la entrada a , agregar la producción $A_i \rightarrow a A_j$. Si el estado i pasa al estado j con la entrada ϵ , agregar la producción $A_i \rightarrow A_j$.
3. Si i es un estado de aceptación, agregar $A_i \rightarrow \cdot$.
4. Si i es el estado inicial, hacer que A_i sea el símbolo inicial de la gramática.

Escritura de una gramática

Las gramáticas son capaces de describir casi la mayoría de la sintaxis de los lenguajes de programación. Por ejemplo, el requerimiento de que los identificadores deben declararse antes de usarse, no puede describirse mediante una gramática libre de contexto. Por lo tanto, las secuencias de los tokens que acepta un analizador sintáctico forman un superconjunto del lenguaje de programación; las fases siguientes del compilador deben analizar la salida del analizador sintáctico, para asegurar que cumpla con las reglas que no verifica el analizador sintáctico.

Comparación entre análisis léxico y análisis sintáctico

sería razonable preguntar: “¿Por qué usar expresiones regulares para definir la sintaxis léxica de un lenguaje?” Existen varias razones.

1. Al separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas, se proporciona una manera conveniente de colocar en módulos la interfaz de usuario de un compilador en dos componentes de un tamaño manejable.
2. Las reglas léxicas de un lenguaje son con frecuencia bastante simples, y para describirlas no necesitamos una notación tan poderosa como las gramáticas.
3. Por lo general, las expresiones regulares proporcionan una notación más concisa y fácil de entender para los tokens, en comparación con las gramáticas.
4. Pueden construirse analizadores léxicos más eficientes en forma automática a partir de expresiones regulares, en comparación con las gramáticas arbitrarias.

No hay lineamientos firmes en lo que se debe poner en las reglas léxicas, en contraste a las reglas sintácticas. Las expresiones regulares son muy útiles para describir la estructura de las construcciones como los identificadores, las constantes, las palabras reservadas y el espacio en blanco. Por otro lado, las gramáticas son muy útiles para describir estructuras anidadas, como los paréntesis balanceados, las instrucciones begin-end relacionadas, las instrucciones if-then-else correspondientes, etcétera. Estas estructuras anidadas no pueden describirse mediante las expresiones regulares.

Eliminación de la ambigüedad

Algunas veces, una gramática ambigua puede rescribirse para eliminar la ambigüedad. Como ejemplo, vamos a eliminar la ambigüedad de la siguiente gramática del “else colgante”:



instr → **if** *expr then instr*
| **if** *expr then instr else instr*
| **otra**

Aquí, “otra” representa a cualquier otra instrucción. De acuerdo con esta gramática, la siguiente instrucción condicional compuesta:

if *E₁* **then** *S₁* **else** **if** *E₂* **then** *S₂* **else** *S₃*

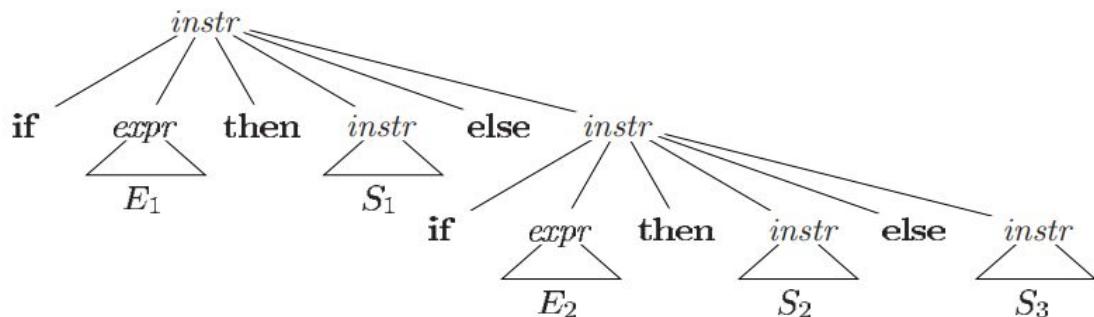


Figura 4.8: Árbol de análisis sintáctico para una instrucción condicional

Eliminación de la recursividad por la izquierda

Una gramática es recursiva por la izquierda si tiene una terminal A tal que haya una derivación $A \Rightarrow A\alpha$ para cierta cadena α . Los métodos de análisis sintáctico descendentes no pueden manejar las gramáticas recursivas por la izquierda, por lo que se necesita una transformación para eliminar la recursividad por la izquierda.

La recursividad inmediata por la izquierda puede eliminarse mediante la siguiente técnica, que funciona para cualquier número de producciones A. En primer lugar, se agrupan las producciones de la siguiente manera:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

en donde ninguna β_i termina con una A. Después, se sustituyen las producciones A mediante lo siguiente:



$$\begin{array}{c} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}$$

El no terminal A genera las mismas cadenas que antes, pero ya no es recursiva por la izquierda. Este procedimiento elimina toda la recursividad por la izquierda de las producciones A y A' (siempre y cuando ninguna α_i sea ϵ), pero no elimina la recursividad por la izquierda que incluye a las derivaciones de dos o más pasos. Por ejemplo, considere la siguiente gramática:

$$\begin{array}{c} S \rightarrow A \ a \mid b \\ A \rightarrow A \ c \mid S \ d \mid \epsilon \end{array}$$

El no terminal S es recursiva por la izquierda, ya que $S \Rightarrow Aa \Rightarrow Sda$, pero no es inmediatamente recursiva por la izquierda.

Cabrera Ramírez Gerardo

Compilador: es un programa que puede leer programas en un lenguaje (el lenguaje fuente) y traducirlo en un programa en otro lenguaje (el lenguaje destino).

programa fuente  compilador  programa destino

Si el programa destino es un programa ejecutable en lenguaje máquina, entonces el usuario puede ejecutarlo para procesar las entradas y producir salidas.

entrada  programa destino  salida

Intérprete: Este en vez de producir un programa destino como traducción, este nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen (fuente) con las entradas proporcionadas por el usuario.

programa fuente 



entrada

intérprete

salida

En la estructura de un compilador hay dos procesos:

Análisis: Que divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Si esta parte detecta un error de sintaxis en el programa fuente, este debe enviar un mensaje de error al usuario para que se corrija.

Síntesis: Construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos.

Análisis del léxico: Este lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma: nombre-token, valor-atributo

Análisis sintáctico: Utiliza los primeros componentes de los tokens generados por el analizador de léxico para crear una representación que describa la estructura gramatical del flujo de tokens.

Análisis semántico: Utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. Lo importante de esto es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tenga operandos que coincidan.

Generación de código: recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones de memoria para cada variable que utiliza el programa.

y como ya todos sabemos el lenguaje máquina es el que puede leer las computadoras como todos sus componentes ya sea la RAM, procesador, etc. y el de alto nivel es el que el usuario puede ordenar a la máquina que hacer a través de instrucciones en nuestro lenguaje.

Análisis Sintáctico (parte 1)

Es el proceso de determinar cómo puede generarse una cadena de terminales mediante una gramática.

Esta sección introduce un método de análisis sintáctico conocido como “descenso recursivo”, este puede usarse tanto para el análisis sintáctico, como para la implementación de traductores orientados a la sintaxis.

La mayoría de los métodos de análisis sintáctico se adaptan a una de dos clases, llamadas métodos descendente y ascendente. Estos se refieren al orden en el que se construyen los nodos en el árbol de análisis sintáctico. En los analizadores tipo descendente, la construcción empieza en la raíz y procede hacia las hojas, mientras que en los analizadores tipo ascendente, la construcción empieza en las hojas y procede hacia la raíz.



Análisis sintáctico tipo arriba-abajo

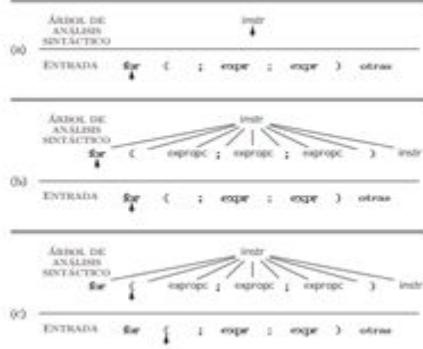
La construcción descendente de un árbol de análisis sintáctico como el de la figura 2.17 se realiza empezando con la raíz, etiquetada con el no terminal inicial instr

instr → expr ;

- | if (expr) instr
- | for (expropc ; expropc ; expropc) instr
- | otras

expropc → ε

- | expr



Análisis sintáctico predictivo

Aquí consideraremos una forma simple de análisis sintáctico de descenso recursivo, conocido como análisis sintáctico predictivo, en el cual el símbolo de preanálisis determina sin ambigüedad el flujo de control a través del cuerpo del procedimiento para cada no terminal.

Árboles sintácticos para las instrucciones

| | |
|-------------------------|--|
| programa → bloque | { return Bloque; n; } |
| bloque → '{' instr '}' | { Bloque.n = instr.n; } |
| instr → instrs instr | { instrs.n = new Seq(instrs.n, instr.n); } |
| | { instrs.n = null; } |
| instr → expr ; | { instr.n = new Eval(expr.n); } |
| if (expr) instr | { instr.n = new If(expr.n, instr.n); } |
| while (expr) instr | { instr.n = new While(expr.n, instr.n); } |
| do instr while (expr) | { instr.n = new Do(instr.n, expr.n); } |
| bloque | { instr.n = Bloque.n; } |
| expr → rel * expr | { expr.n = new Axiop(*, rel.n, expr.n); } |
| rel | { expr.n = rel.n; } |
| rel → rel < adic | { rel.n = new Rel(<,'<', rel.n, adic.n); } |
| rel ≤ adic | { rel.n = new Rel(≤,'≤', rel.n, adic.n); } |
| adic | { rel.n = adic.n; } |
| adic → adic1 + term | { adic.n = new Op('+', adic1.n, term.n); } |
| term | { adic.n = term.n; } |
| term → term1 * factor | { term.n = new Op(*, term1.n, factor.n); } |
| factor | { term.n = factor.n; } |
| factor → (expr) | { factor.n = expr.n; } |
| num | { factor.n = new Num(num.valor); } |

Figura 2.20: Construcción de árboles sintácticos para expresiones e instrucciones

Árboles sintácticos para las expresiones



| SINTAXIS CONCRETA | SINTAXIS ABSTRACTA |
|-------------------------|--------------------|
| = | asigna |
| | cond |
| && | cond |
| == != | rel |
| < <= > >= > | rel |
| + - | op |
| * / % | op |
| ! | not |
| - <small>unario</small> | menos |
| [] | acceso |

Análisis Léxico (partes 2 y 3)

Un analizador léxico lee los caracteres de la entrada y los agrupa en “objetos token”. Junto con un símbolo de terminal que se utiliza para las decisiones de análisis sintáctico, un objeto token lleva información adicional en forma de valores de atributos. El analizador léxico en esta sección permite que aparezcan números, identificadores y “espacio en blanco” (espacios, tabuladores y nuevas líneas) dentro de las expresiones.

Lectura adelantada

Si el siguiente carácter es =, entonces > forma parte de la secuencia de caracteres >=, el lexema para el token del operador “mayor o igual que”. En caso contrario, > por sí solo forma el operador “mayor que”, y el analizador léxico ha leído un carácter de más. Un método general para leer por adelantado en la entrada es mantener un búfer de entrada, a partir del cual el analizador léxico puede leer y devolver caracteres.

Un analizador léxico

Hasta ahora en esta sección, los fragmentos de pseudocódigo se juntan para formar una función llamada escanear que devuelve objetos token, de la siguiente manera:

```
Token escanear () {  
    omitir espacio en blanco;  
    manejar los números;  
    manejar las palabras reservadas e identificadores;  
    /* Si llegamos aquí, tratar el carácter de lectura de preanálisis vistazo como token */  
    Token t = new Token(vistazo);  
    vistazo = espacio en blanco /* inicialización, como lo vimos antes */;  
    return t;  
}
```

Tokens, patrones y lexemas

- Un token es un par que consiste en un nombre de token y un valor de atributo opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico. A partir de este momento, en general escribiremos el nombre de un token en negrita. Con frecuencia nos referiremos a un token por su nombre.
- Un patrón es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.
- Un lexema es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

| TOKEN | DESCRIPCIÓN INFORMAL | LEXEMAS DE EJEMPLO |
|--------------------|--|----------------------------|
| if | caracteres i, f | if |
| else | caracteres e, l, s, e | Else |
| comparacion | < o > o <= o >= o == o != | <=, != |
| id | letra seguida por letras y dígitos | p1, puntuacion, D2 |
| numero | cualquier constante numérica | 3.14159, 0, 6.02e23 |
| literal | cualquier cosa excepto ", rodeada por "s | "core dumped" |

Atributos para los tokens

Cuando más de un lexema puede coincidir con un patrón, el analizador léxico debe proporcionar a las siguientes fases del compilador información adicional sobre el lexema específico que coincidió.

Otras funciones que realiza :

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, ..., y tratarlos correctamente con respecto a la tabla de símbolos (sólo en los casos que debe de tratar con la tabla de símbolos).
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información sobre dónde se ha producido.

Ejemplo: Los nombres de los tokens y los valores de atributo asociados para la siguiente

instrucción en Fortran:

$E = M * C^{**} 2$

se escriben a continuación como una secuencia de pares.

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

Observe que en ciertos pares en especial en los operadores, signos de puntuación y palabras clave, no hay necesidad de un valor de atributo.

Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente.

Ejemplo:

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

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

La estrategia de recuperación más simple es la recuperación en “modo de pánico”. Eliminamos caracteres sucesivos del resto de la entrada, hasta que el analizador léxico pueda encontrar un token bien formado al principio de lo que haya quedado de entrada. Esta técnica de recuperación puede confundir al analizador sintáctico, pero en un entorno de computación interactivo, puede ser bastante adecuado.

Otras de las posibles acciones de recuperación de errores son:

1. Eliminar un carácter del resto de la entrada.
2. Insertar un carácter faltante en el resto de la entrada.
3. Sustituir un carácter por otro.
4. Transponer dos caracteres adyacentes.

Uso de búfer en la entrada

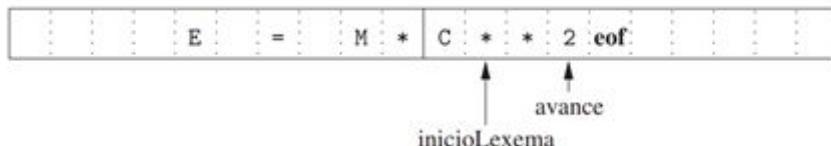
Esta tarea se dificulta debido a que a menudo tenemos que buscar uno o más caracteres más allá del siguiente lexema para poder estar seguros de que tenemos el lexema correcto.



Pares de búferes

Se han desarrollado técnicas especializadas de uso de búferes para reducir la cantidad de sobrecarga requerida en el procesamiento de un solo carácter de entrada.

Un esquema importante implica el uso de dos búferes que se recargan en forma alterna:



Cada búfer es del mismo tamaño N , y por lo general N es del tamaño de un bloque de disco (es decir, 4 096 bytes). Mediante el uso de un comando de lectura del sistema podemos leer N caracteres y colocarlos en un búfer, en vez de utilizar una llamada al sistema por cada carácter.

Se mantienen dos apuntadores a la entrada:

1. El apuntador `inicioLexema` marca el inicio del lexema actual, cuya extensión estamos tratando de determinar.
2. El apuntador `avance` explora por adelantado hasta encontrar una coincidencia en el patrón; durante el resto del capítulo cubriremos la estrategia exacta mediante la cual se realiza esta determinación.

Especificación de los tokens

Las expresiones regulares son una notación importante para especificar patrones de lexemas. Aunque no pueden expresar todos los patrones posibles, son muy efectivas para especificar los tipos de patrones que en realidad necesitamos para los tokens.

Operaciones en los lenguajes

La concatenación de lenguajes es cuando se concatenan todas las cadenas que se forman al tomar una cadena del primer lenguaje y una cadena del segundo lenguaje, en todas las formas posibles. La cerradura (Kleene) de un lenguaje L , que se denota como L^* , es el conjunto de cadenas que se obtienen al concatenar L cero o más veces.

Observe que L^0 , la “concatenación de L cero veces”, se define como $\{\}$, y por inducción, L^i es $L^{i-1}L$. Por último, la cerradura positiva, denotada como L^+ , es igual que la cerradura de Kleene, pero sin el término L^0 . Es decir, no estará en L^+ a menos que esté en el mismo L .

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



Definiciones regulares

Si Σ es un alfabeto de símbolos básicos, entonces una definición regular es una secuencia de definiciones de la forma:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

...

$$d_n \rightarrow r_n$$

En donde:

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

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

Al restringir r_i a Σ y a las d_s definidas con anterioridad, evitamos las definiciones

recursivas y podemos construir una expresión regular sobre Σ solamente, para cada r_i .

Reconocimiento de tokens

Ahora debemos estudiar cómo tomar todos los patrones para todos los tokens necesarios y construir una pieza de código para examinar la cadena de entrada y buscar un prefijo que sea un lexema que coincida con uno de esos patrones.

$$\text{instr} \rightarrow \text{if expr then instr}$$

$$| \text{if expr then instr else instr}$$

$$| \epsilon$$

$$\text{expr} \rightarrow \text{term opre term}$$

$$| \text{term}$$

$$\text{term} \rightarrow \text{id}$$

$$| \text{numero}$$

Para oprel, usamos los operadores de comparación de lenguajes como Pascal o SQL, en donde = es “es igual a” y $<>$ es “no es igual a”, ya que presenta una estructura interesante de lexemas.

Las terminales de la gramática, que son if, then, else, oprel, id y numero, son los nombres de tokens en lo que al analizador léxico respecta.

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



Diagramas de transición de estados

en la construcción de un analizador léxico, primero convertimos los patrones en diagramas de flujo estilizados, a los cuales se les llama “diagramas de transición de estados”.

Los diagramas de transición de estados tienen una colección de nodos o círculos, llamados estados. Cada estado representa una condición que podría ocurrir durante el proceso de explorar la entrada, buscando un lexema que coincida con uno de varios patrones.

Las líneas se dirigen de un estado a otro del diagrama de transición de estados. Cada línea se etiqueta mediante un símbolo o conjunto de símbolos. Si nos encontramos en cierto estado s , y el siguiente símbolo de entrada es a , buscamos una línea que salga del estado s y esté etiquetado por a (y tal vez por otros símbolos también).

Ejemplo:

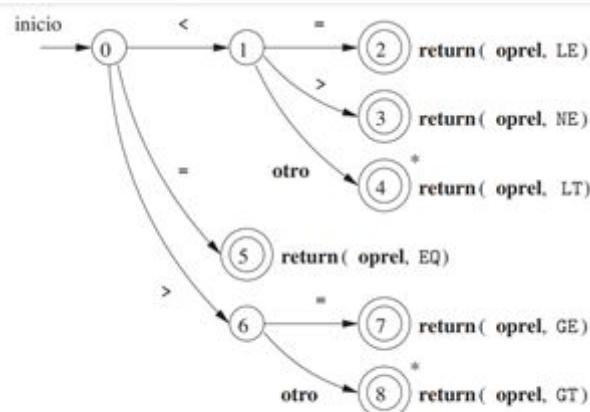


Figura 3.13: Diagrama de transición de estados para **oprel**

Reconocimiento de las palabras reservadas y los identificadores

El reconocimiento de las palabras reservadas y los identificadores presenta un problema. Por lo general, las palabras clave como if o then son reservadas (como en nuestro bosquejo), por lo que no son identificadores, aun cuando lo parecen.

Para buscar lexemas de identificadores, este diagrama también reconocerá las palabras clave if, then y else de nuestro bosquejo.

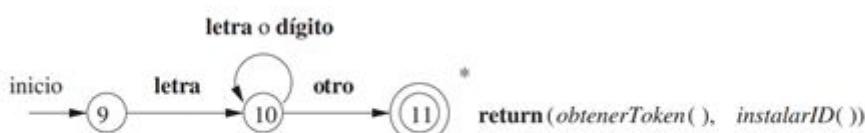


Figura 3.14: Un diagrama de transición de estados para identificadores (**id**) y palabras clave

Finalización del bosquejo

El diagrama de transición para los tokens id que vimos en la figura tiene una estructura simple. Empezando en el estado 9, comprueba que el lexema empiece con una letra y



que pase al estado 10 en caso de ser así. Permanecemos en el estado 10 siempre y cuando la entrada contenga letras y dígitos. Al momento de encontrar el primer carácter que no sea letra ni dígito, pasamos al estado 11 y aceptamos el lexema encontrado.

Arquitectura de un analizador léxico basado en diagramas de transición de estados
Hay varias formas en las que pueden utilizarse los diagramas de transición de estados para construir un analizador léxico. Cada estado se representa mediante una pieza de código. El código para un estado es en sí una instrucción switch o una bifurcación de varias vías que determina el siguiente estado mediante el proceso de leer y examinar el siguiente carácter de entrada.

Ejemplo:

En la figura 3.18 vemos un bosquejo de obtenerOpRel(), una función en C++ cuyo trabajo es simular el diagrama de transición de la figura 3.13 y devolver un objeto de tipo TOKEN

```
TOKEN obtenerOpRel()
{
    TOKEN tokenRet = new (OPREL);
    while(1) { /* repite el procesamiento de caracteres hasta que
        ocurre un retorno o un fallo */
        switch(estado) {
            case 0: c = sigCar();
                if ( c == '<' ) estado = 1;
                else if ( c == '=' ) estado = 5;
                else if ( c == '>' ) estado = 6;
                else fallo(); /* el lexema no es un oprel */
                break;
            case 1: ...
            ...
            case 8: retractar();
                tokenRet.attributo = GT;
                return(tokenRet);
        }
    }
}
```

Figura 3.18: Bosquejo de la implementación del diagrama de transición **oprel**

El generador de analizadores léxicos Lex

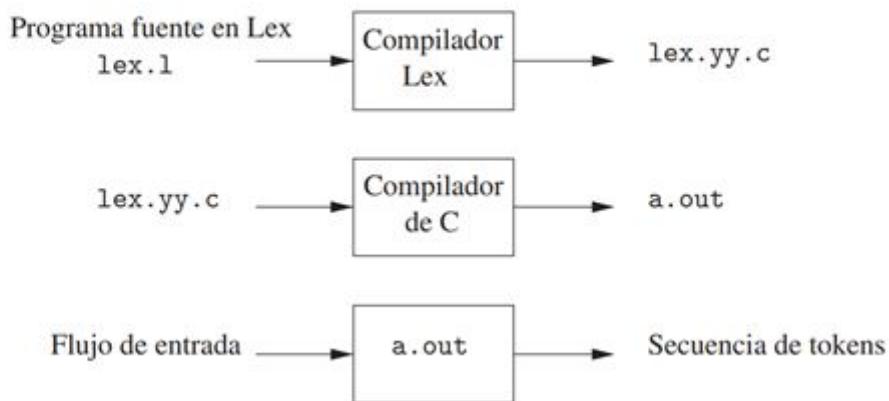
La notación de entrada para la herramienta Lex se conoce como el lenguaje Lex, y la herramienta en sí es el compilador Lex. El compilador Lex transforma los patrones de entrada en un diagrama de transición y genera código, en un archivo llamado lex.yy.c, que simula este diagrama de transición.

Uso de lex

Un archivo de entrada, al que llamaremos lex.1, está escrito en el lenguaje Lex y describe el analizador léxico que se va a generar. El compilador Lex transforma a lex.1 en un programa en C, en un archivo que siempre se llama lex.yy.c. El compilador de C compila este archivo en un archivo llamado a.out, como de costumbre. La salida del compilador de C es un analizador léxico funcional, que puede recibir un flujo de caracteres de entrada y producir una cadena de tokens.



El generador de analizadores léxicos Lex



Estructura de los programas en Lex

Un programa en Lex tiene la siguiente forma:

```
declaraciones
%%
reglas de traducción
%%
funciones auxiliares
```

Cada patrón es una expresión regular, la cual puede usar las definiciones regulares de la sección de declaraciones. Las acciones son fragmentos de código, por lo general, escritos en C, aunque se han creado muchas variantes de Lex que utilizan otros lenguajes.

El analizador léxico que crea Lex trabaja en conjunto con el analizador sintáctico de la siguiente manera. Cuando el analizador sintáctico llama al analizador léxico, éste empieza a leer el resto de su entrada, un carácter a la vez, hasta que encuentra el prefijo más largo de la entrada que coincide con uno de los patrones Pi. Después ejecuta la acción asociada Ai. Por lo general, Ai regresará al analizador sintáctico, pero si no lo hace (tal vez debido a que Pi describe espacio en blanco o comentarios), entonces el analizador léxico procede a buscar lexemas adicionales, hasta que una de las acciones correspondientes provoque un retorno al analizador sintáctico. El analizador léxico devuelve un solo valor, el nombre del token, al analizador sintáctico, pero utiliza la variable entera compartida yyval para pasarle información adicional sobre el lexema encontrado, si es necesario.

Resolución de conflictos en Lex

Nos hemos referido a las dos reglas que utiliza Lex para decidir acerca del lexema apropiado a seleccionar, cuando varios prefijos de la entrada coinciden con uno o más patrones:

1. Preferir siempre un prefijo más largo a uno más corto.
2. Si el prefijo más largo posible coincide con dos o más patrones, preferir el patrón que se lista primero en el programa en Lex.

Autómatas finitos

Estos consisten en gráficos como los diagramas de transición de estados, con algunas diferencias:

1. Los autómatas finitos son reconocedores; sólo dicen “sí” o “no” en relación con cada posible cadena de entrada.
2. Los autómatas finitos pueden ser de dos tipos:
 - (a) Los autómatas finitos no deterministas (AFN) no tienen restricciones en cuanto a las etiquetas de sus líneas. Un símbolo puede etiquetar a varias líneas que surgen del mismo estado, y , la cadena vacía, es una posible etiqueta.
 - (b) Los autómatas finitos deterministas (AFD) tienen, para cada estado, y para cada símbolo de su alfabeto de entrada, exactamente una línea con ese símbolo que sale de ese estado.

Autómatas finitos no deterministas

Un autómata finito no determinista (AFN) consiste en:

1. Un conjunto finito de estados S .
2. Un conjunto de símbolos de entrada Σ , el alfabeto de entrada. Suponemos que , que representa a la cadena vacía, nunca será miembro de Σ .
3. Una función de transición que proporciona, para cada estado y para cada símbolo en $\Sigma \cup \{\}$, un conjunto de estados siguientes.
4. Un estado s_0 de S , que se distingue como el estado inicial.
5. Un conjunto de estados F , un subconjunto de S , que se distinguen como los estados aceptantes (o estados finales).

Tablas de transición

También podemos representar a un AFN mediante una tabla de transición, cuyas filas corresponden a los estados, y cuyas columnas corresponden a los símbolos de entrada y a . La entrada para un estado dado y la entrada es el valor de la función de transición que se aplica a esos argumentos. Si la función de transición no tiene información acerca de ese par estado-entrada, colocamos \emptyset en la tabla para ese estado.

Ejemplo:



Autómatas finitos

| ESTADO | <i>a</i> | <i>b</i> | ϵ |
|--------|-------------|-------------|-------------|
| 0 | {0, 1} | {0} | \emptyset |
| 1 | \emptyset | {2} | \emptyset |
| 2 | \emptyset | {3} | \emptyset |
| 3 | \emptyset | \emptyset | \emptyset |

Autómatas finitos deterministas

Un autómata finito determinista (AFD) es un caso especial de un AFN, en donde:

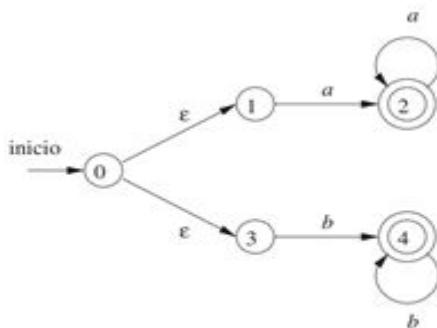


Figura 3.26: AFN que acepta a $aa^*|bb^*$

1. No hay movimientos en la entrada .
 2. Para cada estado s y cada símbolo de entrada a , hay exactamente una línea que surge de s , Indicible como a .
- Si utilizamos una tabla de transición para representar a un AFD, entonces cada entrada es un solo estado. Por ende, podemos representar a este estado sin las llaves que usamos para formar los conjuntos. Mientras que el AFN es una representación abstracta de un algoritmo para reconocer las cadenas de cierto lenguaje, el AFD es un algoritmo simple y concreto para reconocer cadenas.

Ejemplo AFN:

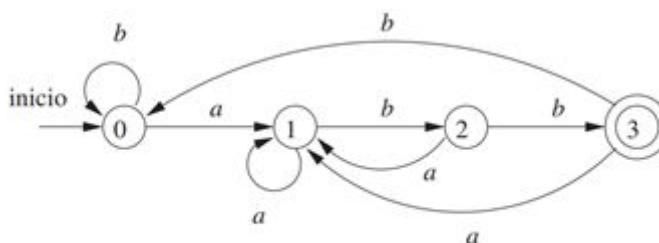


Figura 3.28: AFD que acepta a $(a|b)^*abb$

La estructura del analizador generado

La figura 3.49 presenta las generalidades acerca de la arquitectura de un analizador léxico generado por Lex. El programa que sirve como analizador léxico incluye un



programa fijo que simula a un autómata; en este punto dejamos abierta la decisión de si el autómata es determinista o no. El resto del analizador léxico consiste en componentes que se crean a partir del programa Lex, por el mismo Lex.

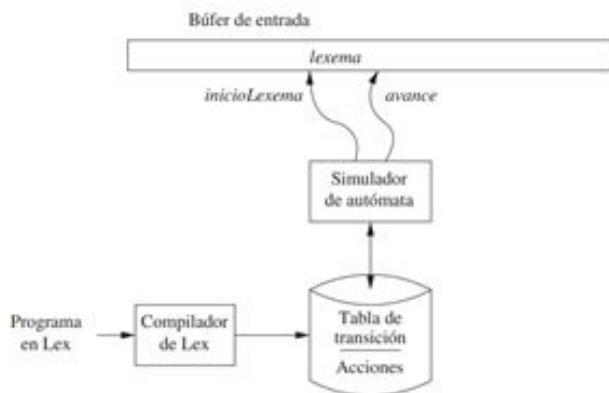


Figura 3.49: Un programa en Lex se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

Estos componentes son:

1. Una tabla de transición para el autómata.
2. Las funciones que se pasan directamente a través de Lex a la salida.
3. Las acciones del programa de entrada, que aparece como fragmentos de código que el simulador del autómata debe invocar en el momento apropiado.

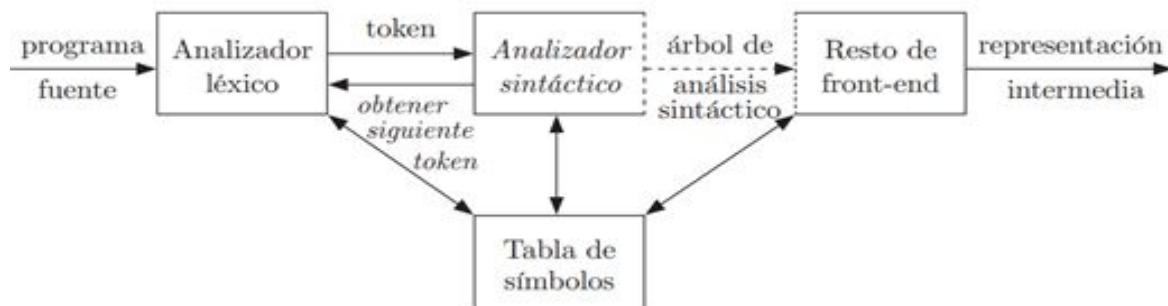
Optimización de los buscadores por concordancia de patrones basados en AFD

1. El primer algoritmo es útil en un compilador de Lex, ya que construye un AFD directamente a partir de una expresión regular, sin construir un AFN intermedio. Además, el AFD resultante puede tener menos estados que el AFD que se construye mediante un AFN.
2. El segundo algoritmo disminuye al mínimo el número de estados de cualquier AFD, mediante la combinación de los estados que tienen el mismo comportamiento a futuro. El algoritmo en sí es bastante eficiente, pues se ejecuta en un tiempo $O(n \log n)$, en donde n es el número de estados del AFD.
3. El tercer algoritmo produce representaciones más compactas de las tablas de transición que la tabla estándar bidimensional.

Análisis sintáctico

La función del analizador sintáctico

el analizador sintáctico obtiene una cadena de tokens del analizador léxico, como se muestra en la imagen:



Y verifica que la cadena de nombres de los tokens pueda generarse mediante la gramática para el lenguaje fuente. Esperamos que el analizador sintáctico reporte cualquier error sintáctico en forma inteligible y que se recupere de los errores que ocurren con frecuencia para seguir procesando el resto del programa.

Para los programas bien formados, el analizador sintáctico construye un árbol de análisis sintáctico y lo pasa al resto del compilador para que lo siga procesando.

Existen tres tipos generales de analizadores para las gramáticas: universales, descendentes y ascendentes.

Los métodos universales de análisis sintáctico como el algoritmo de Cocke-Younger Kasami y el algoritmo de Earley pueden analizar cualquier gramática. Sin embargo, estos métodos generales son demasiado inefficientes como para usarse en la producción de compiladores.

Los métodos que se utilizan, por lo regular, en los compiladores pueden clasificarse como descendentes o ascendentes.

Métodos descendentes: Construyen árboles de análisis sintáctico de la parte superior (raíz) a la parte inferior (hojas).

Métodos ascendentes: Empiezan de las hojas y avanzan hasta la raíz.

En cualquier caso, la entrada al analizador se explora de izquierda a derecha, un símbolo a la vez.

Representación de gramáticas

Las construcciones que empiezan con palabras clave como while o int son muy fáciles de analizar, ya que la palabra clave guía la elección de la producción gramatical que debe aplicarse para hacer que coincida con la entrada.

La asociatividad y la precedencia se resuelvan en la siguiente gramática:

Para describir expresiones, términos y factores. E representa a las expresiones que consisten en términos separados por los signos +, T representa a los términos que consisten en factores separados por los signos *, y F representa a los factores que pueden ser expresiones entre paréntesis o identificadores:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

(4.1)

$F \rightarrow (E) \mid id$

La gramática para expresiones (4.1) pertenece a la clase de gramáticas LR que son adecuadas para el análisis sintáctico ascendentes.

Manejo de los errores sintácticos

Los errores de programación comunes pueden ocurrir en muchos niveles distintos.

- Los errores léxicos incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador tamanioElipce en vez de tamanioElipse, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.
- Los errores sintácticos incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, "{" o "}". Como otro ejemplo, en C o Java, la aparición de una instrucción case sin una instrucción switch que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).
- Los errores semánticos incluyen los conflictos de tipos entre los operadores y los operandos. Un ejemplo es una instrucción return en un método de Java, con el tipo de resultado void.
- Los errores lógicos pueden ser cualquier cosa, desde un razonamiento incorrecto del programador en el uso (en un programa en C) del operador de asignación =, en vez del operador de comparación ==. El programa que contenga = puede estar bien formado; sin embargo, tal vez no refleje la intención del programador.

El mango de errores en un analizador sintáctico tiene objetivos que son simples de declarar, pero difíciles de llevar a cabo:

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

Estrategias para recuperarse de los errores

Recuperación en modo de pánico

Con este método, al describir un error el analizador sintáctico descarta los símbolos de entrada, uno a la vez, hasta encontrar un conjunto designado de tokens de sincronización

Aunque la corrección en modo de pánico a menudo omite una cantidad considerable de entrada sin verificar errores adicionales, tiene la ventaja de ser simple y, a diferencia de ciertos métodos.

Recuperación a nivel de frase



Al descubrir un error, un analizador sintáctico puede realizar una corrección local sobre la entrada restante; es decir, puede sustituir un prefijo de la entrada restante por alguna cadena que le permita continuar. La elección de la corrección local se deja al diseñador del compilador. Desde luego que debemos tener cuidado de elegir sustituciones que no nos lleven hacia ciclos infinitos.

La sustitución a nivel de frase se ha utilizado en varios compiladores que reparan los errores, ya que puede corregir cualquier cadena de entrada.

Producciones de errores

Al anticipar los errores comunes que podríamos encontrar, podemos aumentar la gramática para el lenguaje, con producciones que generen construcciones erróneas. Un analizador sintáctico construido a partir de una gramática aumentada por estas producciones de errores detecta los errores anticipados cuando se utiliza una producción de error durante el análisis sintáctico. Así, el analizador sintáctico puede generar diagnósticos de error apropiados sobre la construcción errónea que se haya reconocido en la entrada.

Corrección global

Lo ideal sería que un compilador hiciera la menor cantidad de cambios en el procesamiento de una cadena de entrada incorrecta.

Dada una cadena de entrada incorrecta x y una gramática G , estos algoritmos buscarán un árbol de análisis sintáctico para una cadena y relacionada, de tal forma que el número de inserciones, eliminaciones y modificaciones de los tokens requeridos para transformar x en y sea lo más pequeño posible.

Gramáticas libres de contexto

Si utilizamos una variable sintáctica instr para denotar las instrucciones, y una variable expr para denotar las expresiones, la siguiente producción:

$$\text{instr} \rightarrow \text{if} (\text{ expr }) \text{ instr } \text{else} \text{ instr} \quad (4.4)$$

especifica la estructura de esta forma de instrucción condicional. Entonces, otras producciones definen con precisión lo que es una expr y qué más puede ser una instr .

La definición formal de una gramática libre de contexto

1. Los terminales son los símbolos básicos a partir de los cuales se forman las cadenas. El término “nombre de token” es un sinónimo de “terminal”; con frecuencia usaremos la palabra “token” en vez de terminal, cuando esté claro que estamos hablando sólo sobre el nombre del token. Asumimos que las terminales son los primeros componentes de los tokens que produce el analizador léxico. En (4.4), los terminales son las palabras reservadas if y else , y los símbolos “(” y “)”.



2. Los no terminales son variables sintácticas que denotan conjuntos de cadenas. En (4.4), instr y expr son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.
3. En una gramática, un no terminal se distingue como el símbolo inicial, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero.
4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:
 - (a) - Un no terminal, conocido como encabezado o lado izquierdo de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
 - (b) - El símbolo \rightarrow . Algunas veces se ha utilizado ::= en vez de la flecha.
 - (c) Un cuerpo o lado derecho, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

En esta gramática, los símbolos de los terminales son:

id + - * / ()

Los símbolos de los no terminales son expresión, term y factor, y expresión es el símbolo inicial.

expresión \rightarrow expresión + term
expresión \rightarrow expresión - term
expresión \rightarrow term
term \rightarrow term * factor
term \rightarrow term / factor
term \rightarrow factor
factor \rightarrow (expresión)
factor \rightarrow id

Convenciones de notación

1. Estos símbolos son terminales:

- (a) Las primeras letras minúsculas del alfabeto, como a, b, c.
- (b) Los símbolos de operadores como +, *, etcétera.
- (c) Los símbolos de puntuación como paréntesis, coma, etcétera.
- (d) Los dígitos 0, 1, ..., 9.
- (e) Las cadenas en negrita como id o if, cada una de las cuales representa un solo símbolo terminal.

2. Estos símbolos son no terminales:

- (a) Las primeras letras mayúsculas del alfabeto, como A, B, C.
- (b) La letra S que, al aparecer es, por lo general, el símbolo inicial.
- (c) Los nombres en cursiva y minúsculas, como expr o instr.
- (d) Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para las expresiones, los términos y los factores se representan a menudo mediante E, T y F, respectivamente.

3. Las últimas letras mayúsculas del alfabeto, como X, Y, Z, representan símbolos gramaticales; es decir, pueden ser no terminales o terminales.

4. Las últimas letras minúsculas del alfabeto, como u, v, ..., z, representan cadenas de terminales (posiblemente vacías).

5. Las letras griegas minúsculas α, β, γ, por ejemplo, representan cadenas (posiblemente vacías) de símbolos gramaticales. Por ende, una producción genérica puede escribirse como $A \rightarrow \alpha$, en donde A es el encabezado y α el cuerpo.

6. Un conjunto de producciones $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$ con un encabezado común A (las llamaremos producciones A), puede escribirse como $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$. A $\alpha_1, \alpha_2, \dots, \alpha_k$ les llamamos las alternativas para A.

7. A menos que se indique lo contrario, el encabezado de la primera producción es el símbolo inicial.

$$\begin{array}{lcl} E & \rightarrow & E \\ & \rightarrow & * \\ & \rightarrow & (E) \end{array} \quad \mid \quad \begin{array}{l} E - \\ \mid \\ id \end{array}$$

Las convenciones de notación nos indican que E, T y F son no terminales, y E es el símbolo inicial. El resto de los símbolos son terminales.

Derivaciones



Considere la siguiente gramática, con un solo no terminal E, la cual agrega una producción $E \rightarrow -E$

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id \quad (4.7)$$

La producción $E \rightarrow -E$ significa que si E denota una expresión, entonces $-E$ debe también denotar una expresión. La sustitución de una sola E por $-E$ se describirá escribiendo lo siguiente:

$$E \Rightarrow -E$$

lo cual se lee como “E deriva a $-E$ ”. La producción $E \rightarrow (E)$ puede aplicarse para sustituir cualquier instancia de E en cualquier cadena de símbolos gramaticales por (E) .

Podemos tomar una sola E y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

A dicha secuencia de sustituciones la llamamos una derivación de $-(id)$ a partir de E.

La cadena $-(id + id)$ es un enunciado de la gramática (4.7), ya que hay una derivación $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$ (4.8)

Las cadenas $E, -E, -(E), \dots, -(id + id)$ son todas formas de frases de esta gramática. Escribimos $E \Rightarrow -*(id + id)$ para indicar que $-(id + id)$ puede derivarse de E.

Árboles de análisis sintáctico y derivaciones

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación. Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama producto o frontera del árbol.

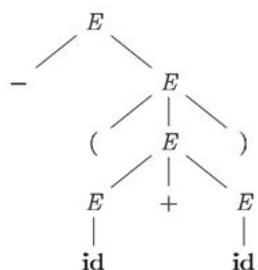


Figura 4.3: Árbol de análisis sintáctico para $-(id + id)$

Ambigüedad



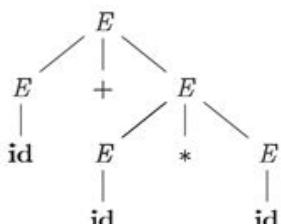
Una gramática ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

Ejemplo 4.11:

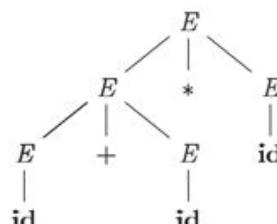
La gramática de expresiones aritméticas (4.3) permite dos derivaciones por la izquierda distintas para el enunciado $\text{id} + \text{id} * \text{id}$:

$$\begin{array}{ll} E \Rightarrow E \quad E & E \Rightarrow E * E \\ \Rightarrow \text{id} \quad E & \Rightarrow E \quad E * E \\ \Rightarrow \text{id} \quad E * E & \Rightarrow \text{id} \quad E * E \\ \Rightarrow \text{id} \quad \text{id} * E & \Rightarrow \text{id} \quad \text{id} * E \\ \Rightarrow \text{id} \quad \text{id} * \text{id} & \Rightarrow \text{id} \quad \text{id} * \text{id} \end{array}$$

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única qué árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con reglas para eliminar la ambigüedad, las cuales “descartan” los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.



(a)



(b)

Figura 4.5: Dos árboles de análisis sintáctico para $\text{id} + \text{id} * \text{id}$

Comparación entre gramáticas libres de contexto y expresiones regulares

Toda estructura que se pueda describir mediante expresiones regulares se puede describir mediante sintaxis, pero no de forma opuesta. O, todo lenguaje convencional es un lenguaje sin contexto, pero no al revés.

Por ejemplo:

La expresión regular $(a|b)^* abb$ y la siguiente gramática:



$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

Podemos construir de manera mecánica una gramática para reconocer el mismo lenguaje que un autómata finito no determinista (AFN), mediante la siguiente construcción:

1. Para cada estado i del AFN, crear un no terminal A_i .
2. Si el estado i tiene una transición al estado j con la entrada a , agregar la producción $A_i \rightarrow aA_j$.
Si el estado i pasa al estado j con la entrada ϵ , agregar la producción $A_i \rightarrow A_j$.
3. Si i es un estado de aceptación, agregar $A_i \rightarrow \cdot$.
4. Si i es el estado inicial, hacer que A_i sea el símbolo inicial de la gramática.

Escritura de una gramática

Las gramáticas son capaces de describir casi la mayoría de la sintaxis de los lenguajes de programación.

El requerimiento de que los identificadores deben declararse antes de usarse, no puede describirse mediante una gramática libre de contexto. Por lo tanto, las secuencias de los tokens que acepta un analizador sintáctico forman un superconjunto del lenguaje de programación; las fases siguientes del compilador deben analizar la salida del analizador sintáctico, para asegurar que cumpla con las reglas que no verifica el analizador sintáctico.

Comparación entre análisis léxico y análisis sintáctico

1. Al separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas, se proporciona una manera conveniente de colocar en módulos la interfaz de usuario de un compilador en dos componentes de un tamaño manejable.
2. Las reglas léxicas de un lenguaje son con frecuencia bastante simples, y para describirlas no necesitamos una notación tan poderosa como las gramáticas.
3. Por lo general, las expresiones regulares proporcionan una notación más concisa y fácil de entender para los tokens, en comparación con las gramáticas.
4. Pueden construirse analizadores léxicos más eficientes en forma automática a partir de expresiones regulares, en comparación con las gramáticas arbitrarias.

No hay lineamientos firmes en lo que se debe poner en las reglas léxicas, en contraste a las reglas sintácticas.

Morales Carrillo Gerardo

EXAMEN

Capítulo 2, ejercicio 9

Algorithm to Remove Left Recursion with an example:

Suppose we have a grammar which contains left recursion:

$S \rightarrow S \text{ a} / S \text{ b} / S \text{ c} / S \text{ d}$

Check if the given grammar contains left recursion, if present then separate the production and start working on it.

In our example,

$S \rightarrow S \text{ a} / S \text{ b} / S \text{ c} / S \text{ d}$

Introduce a new nonterminal and write it at the last of every terminal. We produce a new nonterminal S' and write new production as,

$S \rightarrow cS' / dS'$

Write newly produced nonterminal in LHS and in RHS it can either produce or it can produce new production in which the terminals or non terminals which followed the previous LHS will be replaced by new nonterminal at last.

$S' \rightarrow ? / aS' / bS'$

So after conversion the new equivalent production is

$S \rightarrow cS' / dS'$

$S' \rightarrow ? / aS' / bS'$

Indirect Left Recursion:

A grammar is said to have indirect left recursion if, starting from any symbol of the grammar, it is possible to derive a string whose head is that symbol.

For example,

$A \rightarrow Br$

$B \rightarrow Cd$

$C \rightarrow At$

Where A, B, C are non-terminals and r, d, t are terminals.

Here, starting with A, we can derive A again on substituting C to B and B to A.

Algorithm to remove Indirect Recursion with help of an example:

A1 → A2 A3

A2 → A3 A1 / b

A3 → A1 A1 / a

Where A1, A2, A3 are non terminals and a, b are terminals.

Identify the productions which can cause indirect left recursion. In our case,

A3 → A1 A1 / a

Substitute its production at the place the terminal is present in any other production substitute A1 → A2 A3 in production of A3. A3 → A2 A3 A1.

Now in this production substitute A2 → A3 A1 / b and then replace this by,

A3 → A3 A1 A3 A1 / b A3 A1

Now the new production is converted in form of direct left recursion, solve this by direct left recursion method.

Eliminating direct left recursion in the above,

A3 → a | b A3 A1 | aA' | b A3 A1A'

A' → A1 A3 A1 | A1 A3 A1A'

The resulting grammar is then:

A1 → A2 A3

A2 → A3 A1 | b

A3 → a | b A3 A1 | aA' | b A3 A1A'

A' → A1 A3 A1 | A1 A3 A1

Capítulo 3, ejercicio 1

Iniciamos con la gramática (el paso 1 ya está hecho):

- $E \rightarrow E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1$
- $T \rightarrow T * E | (E) | a | b | Ia | Ib | I0 | I1$
- $F \rightarrow (E) | a | b | Ia | Ib | I0 | I1$
- $I \rightarrow a | b | Ia | Ib | I0 | I1$

Para el paso 2, introducimos nuevas variables y nos quedan

las siguientes reglas:

$A \rightarrow a, B \rightarrow b, Z \rightarrow 0, O \rightarrow 1$

$P \rightarrow +, M \rightarrow *, L \rightarrow (, R \rightarrow)$

y al reemplazar obtenemos la gramática:

$E \rightarrow EPT | TMF | LER | a | b | IA | IB | IZ | IO$

$T \rightarrow TPE | LEL | a | b | IA | IB | IZ | IO$

$F \rightarrow LER | a | b | IA | IB | IZ | IO$

$I \rightarrow a | b | IA | IB | IZ | IO$

$A \rightarrow a, B \rightarrow b, Z \rightarrow 0, O \rightarrow 1$

$P \rightarrow +, M \rightarrow *, L \rightarrow (, R \rightarrow)$

Para el paso 3, reemplazamos:

• $E \rightarrow EPT$ por $E \rightarrow EC1, C1 \rightarrow PT$

• $E \rightarrow TMF, T \rightarrow TMF$ por

$E \rightarrow TC2, T \rightarrow TC2, C2 \rightarrow MF$

• $E \rightarrow LER, T \rightarrow LER, F \rightarrow LER$ por

$E \rightarrow LC3, T \rightarrow LC3, F \rightarrow LC3, C3 \rightarrow ER$

La gramática CNF final es:

• $E \rightarrow EC1 | TC2 | LC3 | a | b | IA | IB | IZ | IO$

• $T \rightarrow TC2 | LC3 | a | b | IA | IB | IZ | IO$



EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE IZTAPALAPA



- **F → LC3|a|b|IA|IB|IZ|IO**
- **I → a|b|IA|IB|IZ|IO**
- **C1 → PT, C2 → MF, C3 → ER**
- **A → a, B → b, Z → 0, O → 1**
- **P → +, M → *, L → (, R →)**