COMPILADOR DE PSEUDOCÓDIGO COMO HERRAMIENTA PARA EL APRENDIZAJE EN LA CONSTRUCCIÓN DE ALGORITMOS

RAFAEL ANIBAL VEGA CASTRO

UNIVERSIDAD DEL NORTE
PROGRAMA DE INGENIERÍA DE SISTEMAS
DIVISIÓN DE INGENIERÍAS
2008

COMPILADOR DE PSEUDOCÓDIGO COMO HERRAMIENTA PARA EL APRENDIZAJE EN LA CONSTRUCCIÓN DE ALGORITMOS

RAFAEL ANIBAL VEGA CASTRO

Tesis para optar al título de Ingeniero de Sistemas

Ing. ALFONSO MANUEL MANCILLA HERRERA
Director del Proyecto

UNIVERSIDAD DEL NORTE
PROGRAMA DE INGENIERÍA DE SISTEMAS
DIVISIÓN DE INGENIERÍAS
2008

Nota de aceptación

_____
_____
_____
_____
_____

_____
Firma del presidente del jurado

_____
Firma del jurado

_____
Firma del jurado

Barranquilla, Diciembre de 2008

DEDICATORIA

A Dios, que siempre me iluminó.

A mi padre y madre, por el apoyo que siempre me han brindado, por confiar en mi, por darme todo lo que hoy tengo y por hacerme todo lo que soy.

A mi hermano, quien siempre se alegra por los logros que consigo y quien quiere seguir los mismo pasos que yo seguí gracias a la vocación que Dios nos dio.

A mi novia, que siempre me apoyaba y me llenaba de fuerzas para continuar cuando quería rendirme..

A mi primo, que desde el cielo siempre me acompaña, al que nunca olvidaré.

Al Ing. Alfonso Mancilla, quien siempre ha confiado en mi, me ha tendido su mano y me ha aconsejado en la toma de muchas de las decisiones mas importantes en mi vida.

A mis tías, abuelas, primos, demás familiares y amigos.,

AGRADECIMIENTOS

En mi vida he tenido la posibilidad de trabajar en varios proyectos, pero ninguno que me diera la satisfacción que siento al terminar los objetivos de este. Digo terminar los objetivos porque a este proyecto se le puede llevar muy lejos.

Agradezco enormemente a todas las ayudas recibidas por parte del Ingeniero Jose Marquez, jefe del programa de Ingeniería de Sistemas de la Universidad del Norte, quien siempre sacaba un rato de su valioso tiempo para resolver mis dudas o darme consejos para resolver mis problemas. No tengo como agradecer al Ingeniero Alfonso Mancilla, docente de planta del programa de Ingeniería de Sistemas de la Universidad del Norte, mi director de tesis, quien me dio la oportunidad y me apoyó en la idea que le plantee en un principio, la cual era un poco ambiciosa, pero el día de hoy nos damos cuenta que cualquier cosa puede lograrse si tiene la dedicación y las ganas que se necesitan.

Rafael Anibal Vega Castro

Barranquilla, Colombia

Octubre 29 de 2008

Tabla de Contenido

Lista de Figuras

Lista de Tablas

INTRODUCCIÓN

Ciertas personas que han hecho saber sus dudas y sugerencias, cuando un estudiante se encuentra por primera vez frente a un algoritmo, se toma determinado tiempo para capar o entender cómo funciona este nuevo concepto. Las personas que se ven ante esta situación, son los estudiantes de primer semestre de Ingeniería de Sistemas en la asignatura de Fundamentos de Programación I y los estudiantes de otros programas.

Después de realizar un análisis del contexto en el que este grupo de estudiantes se encuentran cuando toman dicha asignatura, se ha pensado en desarrollar un compilador de código fuente en pseudocódigo, para que los estudiantes puedan realizar la comprobación de sus algoritmos en un nuevo software, en donde la estructura del código que ellos ingresan se exactamente la misma que la de los algoritmos estudiados en clase.

Esta idea surgió debido a que es incómodo e implica cierto esfuerzo acostumbrase a dos tipos de estructuras cuando se esta aprendiendo algo por primera vez, es decir, para el aprendizaje de algoritmos es más fácil usar siempre la misma estructura en vez de utilizar una estructura en clase y otra cuando se va a realizar la comprobación de dado algoritmo en una computadora.

Ahora, encontraremos una parte innovadora en este proyecto y será la posibilidad de que escribiendo el pseudocódigo en un portal Web dicho portal devuelva un archivo ejecutable, esto sin la necesidad de que el usuario no deba instalar ningún tipo de software para crear sus algoritmos.

1. OBJETIVOS

1.1. OBJETIVO GENERAL

Desarrollar un compilador de código fuente en pseudocódigo para la construcción de algoritmos.

1.2. OBJETIVOS ESPECÍFICOS

- Desarrollar un compilador de código fuente en pseudocódigo para la construcción de algoritmos.
- Desarrollar un Portal Web de compilación de código fuente en pseudocódigo para la construcción de algoritmos.
- Desarrollar un IDE (Integrated Development Environment o Entorno Integrado de Desarrollo) stand alone de compilación de código fuente en pseudocódigo para la construcción de algoritmos.

2. TEORÍA DE COMPILADORES

"Los traductores son programas de computadoras que traducen un lenguaje[1] a otro. Un compilador toma como su entrada un programa escrito en un **lenguaje fuente** y produce un programa equivalente escrito en su **lenguaje objeto**. Por lo regular, el lenguaje fuente es un lenguaje de alto nivel, tal como C o C++, mientras que el lenguaje objetivo es código objeto (también llamado en ocasiones código de máquina) para la máquina objetivo, es decir, código escrito en las instrucciones de máquina correspondientes a la computadora en la cual se ejecutará."[15]

2.1. INTRODUCCIÓN

2.1.1. Concepto de traductor

Un traductor esta definido como un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino produciendo, si cabe, mensajes de error. Los traductores abarcan a los compiladores y a los interpretes[2]. La figura 2.1 muestra el esquema básico de como esta formado un traductor (compilador/intérprete).

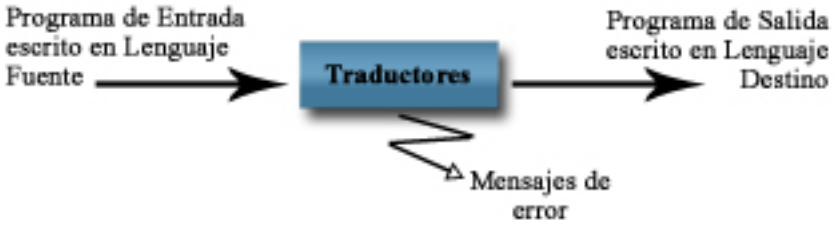

Figura 2.1: Esquema preliminar de un traductor

[1] Un **lenguaje de programación** es un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones, y utilizado para controlar el comportamiento físico y lógico de una máquina.

[2] Un **interprete** es un programa capaz de ejecutar las instrucciones que se encuentra dadas en un lenguaje de programación especifico.

Sin duda alguna la velocidad con la que hoy en día se puede implementar un traductor es incomparable con la década anterior. En la década de 1950, se consideraba a los traductores como programas completamente difíciles de escribir.

2.1.2. Tipo de traductores

Desde el inicio de la computación, ha existido un gran abismo en la forma como las personas expresan sus necesidades y la forma como un computador es capaz de interpretar esas instrucciones. Los traductores han tratado de disminuir este abismo para facilitar el trabajo a los seres humanos, lo que ha llevado a aplicar la teoría de autómatas[3] a diferentes campos y áreas concretas de la informática, dando lugar a los distintos tipos de traductores que existen.

Traductores de idioma

Estos programas traducen de un idioma a otro, por ejemplo del español al inglés. Este tipo de traductores tienen una gran cantidad de problemas:

- Necesita de inteligencia artificial, el problema de la inteligencia artificial actual es que es poco de inteligencia y mucho de artificial, por este motivo en la actualidad resulta casi imposible traducir frases con un el mismo sentido al idioma origen.
- Tienen difícil formalización en la especificación del significado de las palabras.
- El cambio que tiene el sentido de las palabras segundo el contexto.

Compiladores

Son aquellos traductores que tienen como entrada un sentencia de algún lenguaje formal y como salida tienen un programa ejecutable, es decir, que realiza una traducción de un código de alto nivel a código máquina. También un compilador es un programa que genera un archivo objeto en lugar de un

[3] Un **autómata** es una manera matemática para describir clases particulares de algoritmos. En particular los autómatas se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo se pueden utilizar para construir analizadores léxicos.

programa ejecutable final.

Intérpretes

En esencia, es como un compilador, la única diferencia es que la salida de los intérpretes es una ejecución y no un programa ejecutable. El programa entrada se reconoce y ejecuta a la vez. No se produce un resultado físico sino lógico.

La ventaja que tiene los intérpretes es que permiten una fácil depuración. Entre los inconvenientes se encuentra en primer lugar la lentitud de ejecución, ya que al ejecuta a la vez que se traduce no puede aplicarse un alto grado de optimización. Otro inconveniente es que durante la ejecución, el interprete debe residir en memoria, por lo que consume más recursos.

Algunos lenguajes de programación intentan igualar las ventajas de los intérpretes con respecto a los compiladores, estos son los lenguajes pseudointerpretados. En estos últimos, el archivo fuente pasa por un pseudocompilador que genera un pseudoejecutable. Para ejecuta este pseudoejecutable se pasa por un motor de ejecución que lo interpreta de manera relativamente eficiente. Esto tiene ventaja de portabilidad, ya que el pseudoejecutable es independiente de la máquina en que vaya a ejecutarse, ya que basta con que dicha máquina tenga instalado el motor de ejecución apropiado para poder interpretarlo.

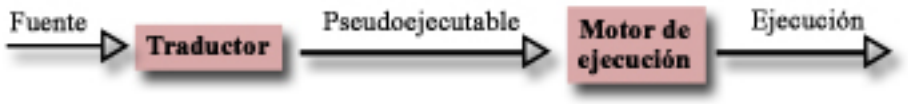

Figura 2.2: Esquema de traducción/ejecución de un programa interpretado

Preprocesadores

Permiten modificar el programa fuente antes de la verdadera compilación. Hacen uso de macroinstrucciones y directivas de compilación. Los preprocesadores suelen actuar de manera transparente para el programador, pudiendo incluso considerarse que son una fase preliminar del compilador.

Intérpretes de comandos

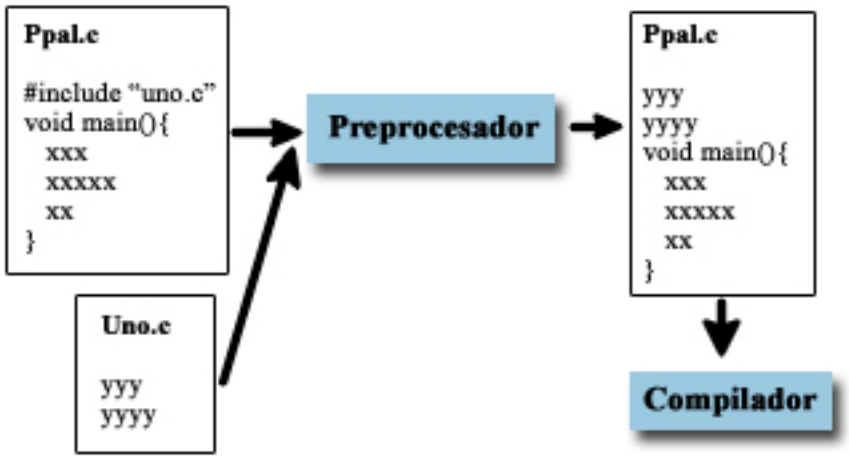

Figura 2.3: Funcionamiento de la directiva de preprocesamiento `#include`

Un intérprete de comandos traduce sentencia simples a invocaciones a programas de una biblioteca o librería. Se utilizan comúnmente en los sistemas operativos. Los programas invocados pueden residir en el núcleo del sistema o estar almacenados en algún dispositivo externo. Ejemplos de este tipo de traductores son: shell[4] de Unix, bash[5] de GNU/Linux y el conocido y tradicional MS-DOS[6] de Microsoft Windows.

Conversores fuente-fuente

Permite traducir desde un lenguaje de alto nivel a otro lenguaje de alto nivel, con lo que se consigue una mayor portabilidad en los programas de alto nivel.

Por ejemplo, si una computadora solo dispone de un compilador de Pascal, y queremos ejecuta un

[4]La **shell** actúa como un intermediario entre el sistema operativo y el usuario gracias a líneas de comando que este último introduce. Su función es la de leer la línea de comandos, interpretar su significado, llevar a cabo el comando y después arrojar el resultado por medio de las salidas.

[5]**Bash** es un shell de Unix (intérprete de órdenes de Unix) escrito para el proyecto GNU. Su nombre es un acrónimo de **b**ourne-**a**gain **sh**ell (otro shell bourne) Ů haciendo un juego de palabras (born-again significa renacimiento) sobre el Bourne shell (sh), que fue uno de los primeros intérpretes importantes de Unix.

[6]**MS-DOS** son las signas de **M**icro**S**oft **D**isk **O**perating **S**ystem, Sistema operativo de disco de Microsoft. Es un sistema operativo comercializado por Microsoft perteneciente a la familia DOS. Fue un sistena operativo para el IBM PC que alcanzó gran difusión.

programa escrito en lenguaje C, entonces un conversor de C a Pascal solucionará el problema. De cualquier forma, el programa fuente resultado puede necesitar de ciertos retoques manuales debido a varios motivos:

- Hay ocasiones en que el lenguaje destino no tiene instrucciones que el lenguaje origen si tiene, por ejemplo, de Java a C, se necesitarían modificaciones ya que C no tiene recolector de basura.
- En situaciones en que la traducción no es inteligente y los programas destino son altamente ineficientes.

Compilador cruzado

En un tipo de compilador que genera código para ser ejecutado en otra máquina. Se utilizan en la fase de desarrollo de nuevos ordenadores. De esta manera es posible construir un sistema operativo de una nueva computadora recurriendo a un lenguaje de alto nivel, incluso antes de que dicha computadora disponga de siquiera un compilador.

2.1.3. Teoría básica relacionada con la traducción

Veremos a continuación una diversa terminología relacionada con el proceso de compilación y de construcción de compiladores.

Pasadas de compilación

Es el número de veces que un compilador debe leer el programa fuente para generar el código. Hay algunas situaciones en las que, para realizar la compilación, no es suficiente con leer el fichero fuente solo una vez.

En la actualidad, cuando un programa necesita hacer varias pasadas de compilación, suele colocar en memoria un representación abstracta del fichero fuente, de manera que las pasadas de compilación se realizan sobre dicha representación en lugar de sobre el fichero de entrada, lo que soluciona el problema de la ineficiencia debido a los operadores de e/s[7]

[7] **Entrada/Salida**, es la entrada y/o salida de datos mediante dispositivos de hardward.

Compilación incremental

Cuando se desarrolla un programa fuente, éste se recompila varias veces hasta obtener una versión definitiva libre de errores. Pues bien, en una compilación incremental sólo se recompilan las modificaciones realizadas desde la última compilación. Lo ideal es que sólo se recompilen aquellas partes que contenían los errores o que, en general, hayan sido modificadas. Sin embargo, esto es muy difícil de conseguir y no suele ahorrar tiempo de compilación más que en algunos casos muy concretos.

Autocompilador

Es un compilador escrito en el mismo lenguaje que compila (o parecido). Normalmente, cuando se extiende entre muchas máquinas diferentes el uso de un compilador, y éste se desea mejorar, el nuevo compilador se escribe utilizando el lenguaje del antiguo, de manera que pueda ser compilado por todas esas máquinas diferentes, y dé como resultado un compilador más potente de este mismo lenguaje.

Descompilador

Un descompilador realiza una labor de traducción inversa, esto es, pasa de un código máquina al equivalente en el lenguaje que lo generó. Cada descompilador trabaja con un lenguaje de alto nivel concreto.

La descompilación suele ser una labor casi imposible, porque al código máquina generado casi siempre se le aplica una optimización en una fase posterior, de manera que un mismo código máquina ha podido ser generado a partir de varios códigos fuentes. Es por esta razón que en la actualidad sólo existen descompiladores de aquellos lenguajes en los que existe una relación biyectiva entre el código destino y el código fuente.

Metacompilador

Este es uno de los conceptos más importantes con los que vamos a trabajar. Un metacompilador es un compilador de compiladores. Se trata de un programa que acepta como entrada la descripción de un lenguaje y produce el compilador de dicho lenguaje. Hoy en día no existen metacompiladores

completos, pero sí parciales en los que se acepta como entrada una gramática de un lenguaje y se genera un autómata que reconoce cualquier sentencia del lenguaje. A este autómata podemos añadirle código para completar el resto del compilador. Entre los metacompiladores mas conocidos se encuentran: Lex, YACC, FLex, JavaCC, JLex, JFlex, CUP, etc.

Los metacompiladores se suelen dividir entre los que pueden trabajar con gramáticas de contexto libre y los que trabajan con gramáticas regulares. Los primeros se dedican a reconocer la sintaxis del lenguaje y los segundos dividen los archivos fuentes y los dividen en palabras.

2.1.4. Estructura de un traductor

Un traductor divide su labor básicamente en dos etapas: una que analiza la entrada y genera estructuras intermedias y otra que sintetiza la salida a partir de dichas estructuras. Por tanto, el esquema de un traductor pasa tener la siguiente forma: Básicamente los objetivos de la etapa de análisis son:

Figura 2.4: Esquema por etapas de un traductor

- Controlar la corrección del programa fuente.
- Generar las estructuras necesarias para comenzar la etapa de síntesis.

Para llevar esto a cabo, la etapa de análisis consta de las siguientes fases:

- **Análisis lexicográfico.** Divide el programa fuente en los componentes básicos del lenguaje a compilar. Cada componente básico es una subsecuencia de caracteres del programa fuente, y pertenece a una categoría gramatical, es decir: números, identificadores, palabras reservadas, signos de puntuación, etc.

- **Análisis sintáctico.** Comprueba que la estructura de los componentes básicos sea correcta según las reglas gramaticales del lenguaje que se compila.
- **Análisis semántico.** Comprueba que el programa fuente respeta las directrices del lenguaje que se compila, es decir , todo lo relacionado con el significado: chequeo de tipos, rangos de valores, existencia de variables, etc.

Cualquiera de estas tres fases puede emitir mensajes de error derivados de fallos cometidos por el programador en la redacción de los textos fuente. Mientras más errores controle un compilador, menos problemas dará un programa en tiempo de ejecución. Por ejemplo, el lenguaje C no controla los límites de un array, lo que provoca que en tiempo de ejecución puedan producirse comportamientos del programa de difícil explicación.

La etapa de síntesis construye el programa objeto deseado (equivalente semánticamente al fuente) a partir de las estructuras generadas por la etapa de análisis. Para ello se compone de tres fases fundamentales:

- **Generación de código intermedio.** Genera un código independiente de la máquina muy parecido al ensamblador. No se genera código máquina directamente porque así es más fácil hacer pseudocompiladores y además se facilita la optimización de código independientemente del microprocesador.
- **Generación del código máquina.** Crea un bloque de código máquina ejecutable, así como los bloques necesarios destinados a contener los datos.
- **Fase de optimización.** La optimización puede realizarse sobre el código intermedio (de forma independiente de las características concretas del microprocesador), sobre el código máquina, o sobre ambos. Y puede ser una aislada de las dos anteriores, o estar integrada con ellas.

2.1.5. Tabla de símbolos

Una función esencial de un compilador es registrar los identificadores de usuario (nombres de variables, de funciones, de tipos, etc.) utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, la dirección de memoria en que se almacenará en tiempo de ejecución, su tipo, su ámbito (la parte del programa donde es visible), etc.

Pues bien, la tabla de símbolos es una estructura de datos que posee información sobre los identificadores definidos por el usuario, ya sean constantes, variables, tipos u otros. Dado que puede contener información de diversa índole, debe hacerse de forma que su estructura no sea uniforme, esto es, no se guarda la misma información sobre una variable del programa que sobre un tipo definido por el usuario. Hace funciones de diccionario de datos y su estructura puede ser una tabla hash, un árbol binario de búsqueda, etc., con tal de que las operaciones de acceso sean lo bastante eficientes.

Tanto la etapa de análisis como la de síntesis acceden a esta estructura, por lo que se halla muy acoplada al resto de fases del compilador. Por ello conviene dotar a la tabla de símbolos de una interfaz lo suficientemente genérica como para permitir el cambio de las estructuras internas de almacenamiento sin que estas fases deban ser retocadas. Esto es así porque suele ser usual hacer un primer prototipo de un compilador con una tabla de símbolos fácil de construir (y por tanto, ineficiente), y cuando el compilador ya ha sido finalizado, entonces se procede a sustituir la tabla de símbolos por otra más eficiente en función de las necesidades que hayan ido surgiendo a lo largo de la etapa de desarrollo anterior. Siguiendo este criterio, el esquema general definitivo de un traductor se detalla en la siguiente figura:

2.2. ANÁLISIS LEXICOGRÁFICO

En esta sección estudiaremos la fase inicial de un compilador, es decir, su análisis lexicográfico, tam-

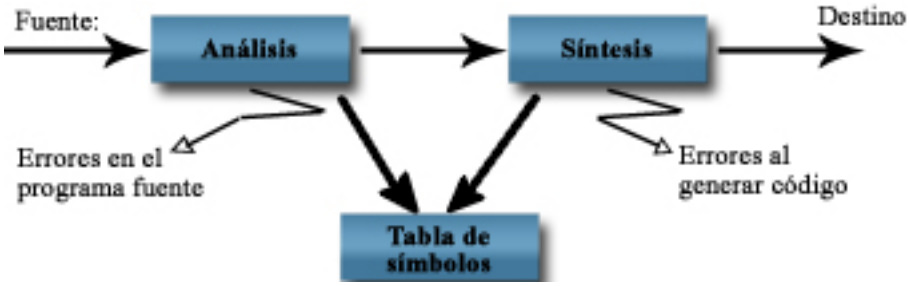

Figura 2.5: Esquema por etapas definitivo de un traductor

bién llamado análisis léxico. Las técnicas que se utilizan para la construcción de analizadores léxicos también se pueden implementar en otras áreas como, por ejemplo, a lenguajes de consulta y sistemas de recuperación de información. Sin importar el área de aplicación, el problema básico es la especificación y diseño de programas que ejecuten las acciones activadas por palabras que siguen ciertos patrones dentro de cadenas a reconocer. Debido a que la programación dirigida por patrones es sin duda alguna de gran utilidad, existen gran cantidad de metalenguajes[8] que permiten la programación de pares que tengan la forma *patrón-acción*, de tal manera que la acción se ejecute una vez se encuentre el patron dentro de una cadena.

La fase de análisis léxico de un compilador tiene como función leer el programa fuente como un conjunto de caracteres y separarlo en tokens[9]. Como básicamente la función de un analizador léxico es un caso especial de coincidencia de patrones, se necesita estudiar métodos de especificación y reconocimiento de patrones en la medida en que se aplican al proceso de análisis léxico. Estos métodos son principalmente las **expresiones regulares** y **autómatas finitos**[10] un analizador léxico debe funcionar de manera tan eficiente como sea posible. Por esto, también se debe prestar mucha atención a los detalles prácticos de la estructura del analizador.

[8] Un **metalenguaje** es un lenguaje usado para hacer referencia a otros lenguajes.

[9] Un **token** es una secuencia de caracteres que representa una unidad de información en el programa fuente, como: `if, while`.

[10],

2.2.1. Proceso de un analizador léxico

"*El trabajo del analizador léxico es leer los caracteres del código fuente y formarlos en unidades lógicas para que lo aborden las partes siguientes del compilador (generalmente el analizador sintáctico). Las unidades lógicas que genera el analizador léxico se denominan* ***tokens****, y formar caracteres en tokens es muy parecido a formar palabras a partir de caracteres con una oración en un lenguaje natural como el inglés o cualquier otro y decidir lo que cada palabra significa. En esto se asemeja a la tarea del deletreo.*"[15]

Los tokens tiene diversas categorías, una de ellas son las palabras reservadas como: *IF* , *THEN*, las cuales representan las cadenas de caracteres *if* y *then*. Otra categoría es la que representa a los símbolos especiales, como *MAS* o *MENOS*, los cuales hacen referencia a los caracteres + y -. Por último, existen tokens que puede representar cadenas con diversos tipos de caracteres, por ejemplo: el NUM y el ID, que representa números e identificadores respectivamente.

2.2.2. Expresiones Regulares

Una expresión regular es un patron que permite encontrar determinados caracteres, palabras o cadenas. Las expresiones regulares se escriben en un lenguaje formal que puede ser interpretado por un procesador de expresiones regulares.

Las expresiones regulares son usadas por muchos editores de texto y lenguajes de programación para buscar y manipular textos basándose en patrones. "Una expresión regular r se encuentra completamente definida mediante el conjunto de cadenas con las que concuerda."[15]

Una expresión regular puede contener caracteres del alfabeto, pero dichos caracteres tienen un significado diferente: en toda expresión regular, cada uno de los símbolos indican patrones, es decir: **a** es el carácter *a* usado como patrón. También, una expresión regular r puede contener caracteres que tienen significados especiales. A esta clase de caracteres se les llaman **metacaracteres** o **metasímbolos**.

Definición de expresiones regulares

Ahora debemos describir el significado de las expresiones regulares al establecer cuáles lenguajes genera cada patrón. Esto se hará en varias etapas. Se comenzará por describir el conjunto de expresiones regulares básicas, las cuales se componen de símbolos individuales. Continuaremos con la descripción de las operaciones que generan nuevas expresiones regulares a partir de las ya existentes. Esto es similar a la manera en que se construyen las expresiones aritméticas: las expresiones aritméticas básicas son los números, tales como 45 y 3.6. Entonces las operaciones aritméticas, como la suma y la multiplicación se pueden utilizar para formar nuevas expresiones a partir de las existentes, como en el caso de 45 * 3.6 y 45 + 3.6 + 2.8.

Expresiones regulares básicas: Éstas son los caracteres simples del alfabeto, los cuales se corresponden a sí mismos. Dado cualquier carácter *a* del alfabeto $\sum$, indicamos que la expresión regular `a` corresponde al carácter *a* escribiendo $L(a) = \{a\}$. Existen otros dos símbolos que necesitaremos en situaciones especiales. Necesitamos poder indicar una concordancia con la cadena vacía, es decir, la cadena que no contiene ningún carácter. Se utilizará el símbolo ϵ (épsilon) para denotar la cadena vacía, y definiremos el metasímbolo ϵ (ϵ en negrillas) estableciendo que $L(\epsilon) = \{\epsilon\}$. También necesitaremos ocasionalmente ser capaces de describir un símbolo que corresponda a la ausencia de cadenas, es decir, cuyo lenguaje sea el conjunto vacío, el cual escribiremos como { }. Emplearemos para esto el símbolo ϕ y escribiremos $L(\phi) = \{\}$. La diferencia entre { } y $\{\epsilon\}$ es que el conjunto { } no contiene ninguna cadena, mientras que el conjunto $\{\epsilon\}$ contiene la cadena vacía, es decir, la que no se compone de ningún carácter.

Operaciones de expresiones regulares: Existen tres operaciones básicas en las expresiones regulares:

1. *Selección entre alternativas, la cual se indica mediante el metacarácter:* | *(barra vertical).*
   Si r y s son expresiones regulares, entonces $r|s$ es una expresión regular que define cualquier cadena que concuerda con r o con s. En términos de lenguajes, el lenguaje de $r|s$ es la unión de los lenguajes de r y s, o $L(r|s) = L(r) \cup L(s)$.

2. *Concatenación, que se indica mediante la yuxtaposición.*

La concatenación de dos expresiones regulares r y s se describe como rs, y corresponde a cualquier cadena que sea la concatenación de dos cadenas, con la primera de ellas correspondiendo a r y la segunda correspondiendo a s. Ahora, la operación de concatenación para expresiones regulares se define como $L(rs) = L(r)L(s)$.

3. *Repetición o "Cerradura", la cual se indica mediante el metacarácter: **

   La operación de repetición de una expresión regular, denominada también en ocasiones *cerradura (de Kleene)*, se escribe r^*, donde r es una expresión regular. La expresión regular r^* corresponde a cualquier concatenación finita de cadenas, cada una de las cuales corresponde a r. En términos de lenguajes, $L((a|bb)^*) = L(a|bb)^* = \{a, bb\}^* = \{\epsilon, a, bb, aa, abb, bbbb, aaa, abba,$
   $abbbb, bbaa, \ldots\}$

Precedencia de operaciones y el uso de los paréntesis: La descripción precedente no toma en cuenta la cuestión de la precedencia de las operaciones de elección, concatenación y repetición. Por ejemplo, dada la expresión regular $a|b^*$, ¿deberíamos interpretar esto como $(a|b)^*$ o como $a|(b)^*$?. La convención estándar es que la repetición debería tener mayor precedencia, por lo tanto, la segunda interpretación es la correcta. En realidad, entre las tres operaciones, se le da al * la precedencia más alta, a la concatenación se le da la precedencia que sigue y a la | se le otorga la precedencia más baja. Cuando deseemos indicar una precedencia diferente, debemos usar paréntesis para hacerlo.

Nombres para expresiones regulares: A menudo es útil como una forma de simplificar la notación proporcionar un nombre para una expresión regular larga, de modo que no tengamos que escribir la expresión misma cada vez que deseemos utilizarla.

El uso de una definición regular es muy conveniente, pero introduce la complicación agregada de que el nombre mismo se convierta en un metasímbolo y se deba encontrar un significado para distinguirlo de la concatenación de sus caracteres. En nuestro caso hicimos esa distinción al utilizar letra cursiva para el nombre. Hay que tener en cuenta que no se debe emplear el nombre del término en su propia definición (es decir, de forma recursiva): debemos poder eliminar nombres reemplazándolos sucesivamente con las expresiones regulares para las que se establecieron.

Extensiones para las expresiones regulares

Se ha formulado una definición para las expresiones regulares que emplean un conjunto mínimo de operaciones comunes a todas las aplicaciones, y podríamos limitarnos a utilizar sólo las tres operaciones básicas más los paréntesis en todos los casos. Sin embargo, escribir expresiones regulares utilizando solo estos operadores en ocasiones es poco manejable, ya que se crean expresiones regulares que son más complicadas de lo que set'rian si se dispusiera de un conjunto de operaciones más expresivo.

La siguiente es una nueva listas de operaciones no básicas definidas para hacer las expresiones regulares mas manejables:

- *UNA O MÁS REPETICIONES*

  Dada una expresión regular r, la repetición de r se describe utilizando la operación de cerradura estándar, que se escribe r^*. Esto permite que r se repita 0 o más veces. Una situación típica que surge es la necesidad de una o más repeticiones en lugar de ninguna, lo que garantiza que aparece por lo menos una cadenas correspondiente a r, y no permite la cadena vacía ϵ.

- *UN INTERVALO DE CARACTERES*

  Frecuentemente se necesita escribir un intervalo de caracteres, como el de todas las letras minúsculas o el de todos los dígitos. Has ahora esto se podría realizar con el operador | : *a*|*b*|*c*|...|*z* para las letras o *0*|*1*|...|*9* para los dígitos. Una alternativa es tener una notación especial para esta situación, y una que es común es la de emplear corchetes y un guión, como en [a-z] para las letras minúsculas y [0-9] para los dígitos. Esto también se puede emplear para alternativas individuales, de modo que a|b|c puede escribirse como [abc]. También se puede incluir los intervalos múltiples, de manera que [a-zA-Z] representa todas las letras minúsculas y mayúsculas.

- *CUALQUIER CARÁCTER QUE NO ESTÉ EN UN CONJUNTO DADO*

  A menudo es de utilidad poder excluir un carácter simple del conjunto de caracteres por generar. Esto se puede conseguir con un metacarácter para indicar la operación de negación o complementaria sobre un conjunto de alternativas. Utilizaremos el operador $\wedge$ para al efecto. Es decir, que $\wedge$[abc] representa cualquier carácter diferente de a, b y c.

- *SUBEXPRESIONES OPCIONALES*

  Por último, hay un caso que se presenta comúnmente y es el de cadenas que contienen partes opcionales que pueden o no aparecer en cualquier cadena en particular. Esto se puede convertir rápidamente en algo voluminoso, e introduciremos el metacarácter de signo de interrogación $r?$ para indicar que las cadenas que coincidan con r son opcionales, esto quiere decir que la expresión regular r puede o no aparecer en la cadena.

2.3. GRAMÁTICAS LIBRES DE CONTEXTO

"El análisis gramatical es la tarea de determinar la sintaxis, o estructura, de un programa. Por esta razón también se le conoce como **análisis sintáctico.** La sintaxis de un lenguaje de programación por lo regular se determina mediante las **reglas gramaticales** de una **gramática libre de contexto**, de manera similar como se determina mediante expresiones regulares la estructura léxica de los tokens reconocida por el analizador léxico. En realidad, una gramática libre de contexto utiliza convenciones para nombrar y operaciones muy similares a las correspondientes en las expresiones regulares. Con la única diferencia de que las reglas de una gramática libre de contexto son **recursivas**."[15]

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante gramáticas independientes del contexto. Por ejemplo, se puede tener una proporción condicional definida por una regla como:
Si S_1 y S_2 son proposiciones y E es una expresión, entonces "**if** E **then** S_1 **else** S_2" es una proposición

No se puede especificar esta forma de proposición condicional usando la notación de las expresiones regulares. Por otro lado, utilizando la variable sintáctica $prop$ para denotar la clase de las proposiciones y $expr$ para la clase de las expresiones, ya se puede expresar la regla anterior usando la siguiente producción gramatical:

$$prop \rightarrow \textbf{if } expr \textbf{ then } prop \textbf{ else } prop \tag{2.1}$$

Una gramática independiente del contexto consta de terminales, no terminales, un símbolo inicial y producciones.

- Los terminales son los símbolos básicos con que se forman las cadenas. “Componente léxico” es un sinónimo de “terminal” cuando se trata de gramáticas para lenguajes de programación. En (1.1), cada una de las palabras clave **if, then** y **else** es un terminal.

- Los no terminales son variables sintácticas que denotan conjuntos de cadenas. En (1.1), *prop* y *expr* son no terminales. Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática. También imponen una estructura jerárquica sobre el lenguaje que es útil tanto para el análisis sintáctico como para la traducción.

- En una gramática, un no terminal es considerado como el símbolo inicial, y el conjunto de cadenas que representa es el lenguaje definido por la gramática.

- Las producciones de una gramática especifican cómo se pueden combinar los terminales y los no terminales para formar cadenas. Cada producción consta de un no terminal, seguido por una flecha (a veces se usa el símbolo ::=, en lugar de la flecha), seguida por una cadena de no terminales y terminales.

2.3.1. Convenciones de notación

Para evitar tener que establecer siempre que “estos son los terminales”, “estos son los no terminales”, etcétera, a partir de ahora se emplearán las siguientes convenciones de notación con respecto a las gramáticas.

1. Estos símbolos son terminales:

    a) Las primeras letras minúsculas del alfabeto, como *a, b, c*.

    b) Los símbolos de operador, como $+$, $-$, etcétera.

    c) Los símbolos de puntuación, como paréntesis, coma, etcétera.

    d) Los dígitos 0,1,...,9.

    e) Cadenas en **negrillas**, como **id** o **if**.

2. Estos símbolos son no terminales:

a) Las primeras letras mayúsculas del alfabeto, como *A, B, C*.

b) La letra S, que cuando aparece suele ser el símbolo inicial.

c) Los nombres en cursivas minúsculas, como *expr* o *prop*.

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

4. Las últimas letras minúsculas del alfabeto, principalmente *u, v,..., z*, representan cadenas de terminales.

5. Las letras griegas minúsculas, α, β, γ, por ejemplo, representa cadenas de símbolos gramaticales. Por tanto, una producción genérica podría escribirse $A \rightarrow \alpha$, indicando que hay un solo no terminal A a la izquierda de la flecha y una cadena de símbolos gramaticales α a la derecha de la flecha.

6. Si $A \rightarrow \alpha_1, A \rightarrow \alpha_2, ..., A \rightarrow \alpha_k$ son todas producciones con A a la izquierda, entonces se pueden escribir $A \rightarrow \alpha_1|\alpha_2|...|\alpha_k$.

7. A menos que se diga otra cosa, el lado izquierdo de la primera producción es el símbolo inicial.

2.3.2. Derivación

Hay varias formas de considerar el proceso mediante el cual una gramática define un lenguaje. La idea central es que se considera una producción como una regla reescrita, donde el no terminal de la izquierda es sustituido por la cadena del lado derecho de la producción.
Por ejemplo, consideremos la siguiente gramática para expresiones aritméticas, donde el no terminal E representa una expresión.

$$E \rightarrow E + E|E * E|(E)| - E|\mathbf{id} \tag{2.2}$$

La producción $E \rightarrow -E$ significa que una expresión precedida por un signo menos es también una expresión. Esta producción se puede usar para generar expresiones más complejas a partir de expresiones más simples permitiendo sustituir cualquier presencia de E por $-E$. En el caso más simple, se puede sustituir una sola E por $-E$. Se puede describir esta acción escribiendo.

$$E \Rightarrow -E$$

que se lee "E deriva $-E$. La producción $E \rightarrow (E)$ establece que también se podría sustituir una presencia de una E en cualquier cadena de símbolos gramaticales por (E); por ejemplo, $E * E \Rightarrow (E) * E$ o $E * E \Rightarrow E * (E)$.

Se puede tomar una sola E y aplicar repetidamente producciones en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

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

A dicha secuencia de sustituciones se le llama *derivación* de -(**id**) a partir de E. Esta derivación proporciona una prueba de que un caso determinado de una expresión es la cadena -(**id**).

De forma más abstracta, se dice que $\alpha A \beta \Rightarrow \alpha\gamma\beta$ si $A \rightarrow \gamma$ es una producción y α y β son cadenas arbitrarias de símbolos gramaticales. Si $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$, se dice que α_1 deriva a α_n. El símbolo $\Rightarrow$ significa "deriva en un paso". A menudo se desea decir "deriva en cero o más pasos". Para este propósito se puede usar el símbolo $\overset{*}{\Rightarrow}$. Así:

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

Del mismo modo se puede usar $\overset{+}{\Rightarrow}$ para expresar "derivada en uno o más pasos".

2.3.3. Ambigüedad

"Se dice que una gramática que produce más de un árbol de análisis sintáctico para alguna frase es *ambigua*. O, dicho de otro modo, una gramática ambigua es la que produce más de una derivación por la izquierda o por la derecha para la misma frase. Para algunos tipos de analizadores sintácticos es preferible que la gramática no se ambigua, pues si lo fuera, no se podría determinar de manera exclusiva qué árbol de análisis sintáctico seleccionar para una frase. Para algunas aplicaciones se considerarán también los métodos mediante los cuales se pueden utilizar ciertas gramáticas ambiguas, junto con *reglas para eliminar ambigüedades* que desechan árboles de análisis sintático indeseables, dejando sólo un árbol para cada frase."[2]

2.3.4. Escritura de una gramática

"Las gramáticas son capaces de describir la mayoría, pero no todas, de las sintaxis de los lenguajes de programación. Un analizador léxico efectúa una cantidad limitada de análisis sintáctico conforme produce la secuencia de componentes léxicos a partir de los caracteres de entrada. Ciertas limitaciones de la entrada, como el requisito de que los identificadores se declaren antes de ser utilizados, no pueden describirse mediante una gramática independiente del contexto. Por tanto, las secuencias de componentes léxicos aceptadas por un analizador sintáctico forman un superconjunto de un lenguaje de programación; las fases posteriores deben analizar la salida del analizador sintáctico para garantizar la obediencia a reglas que el analizador sintáctico no comprueba."[2]

Cada método del análisis sintáctico puede manejar sólo gramáticas de una cierta forma, quizá se debe reescribir la gramática inicial para hacerla analizable por el método elegido. Frecuentemente se construyen gramáticas adecuadas para expresiones utilizando la información sobre la precedencia y la asociatividad.

Expresiones regulares, o gramáticas libres de contexto

Cualquier construcción que se pueda escribir como una expresión regular también puede ser representada por medio de una gramática. Por ejemplo, la expresión regular $(a|b)^*abb$ y la gramática

$A_0 \rightarrow aA_0|bA_0|aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \epsilon$

describen el mismo lenguaje, el conjunto de cadenas de caracteres a Y b que terminan en abb.

Debido a que toda expresión regular puede ser expresada también como una gramática libre de contexto, es razonable preguntarse: ¿Por qué utilizar expresiones regulares para definir la sintaxis lexicográfica de un lenguaje? Existen varias razones.

1. Las reglas lexicográficas de un lenguaje a menudo son bastantes sencillas, y para describirlas no se necesita un notación tan poderosa como la utilizada en las gramáticas.

2. Las expresiones regulares por lo general proporcionan una notación más concisa y fácil de entender para los componentes léxicos que una gramática.

3. Se pueden construir automáticamente analizadores léxicos más eficientes a partir de expresiones regulares que de gramáticas arbitrarias.

4. Separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas proporciona una forma convenientes de modularizar la etapa inicial de un compilador en dos componentes de tamaño razonable.

No hay normas fijas para poner reglas lexicográficas, en vez de las reglas sintácticas. Las expresiones regulares son muy útiles para describir la estructura de las construcciones léxicas, como identificadores, constantes, palabras clave, etcétera. Por otra lado, las gramáticas son muy útiles para describir estructuras anidadas, como paréntesis equilibrados, concordancia de las palabras clave, etcétera. Como ya se ha señalado, estas estructuras anidadas no se pueden escribir con expresiones regulares.

Comprobación del lenguaje generado por una gramática

Aunque los diseñadores de compiladores rara vez lo hacen para una gramática completa de un lenguaje de programación, es importante ser capaz de razonar que un conjunto dado de producciones genera un lenguaje determinado. Las construcciones problemáticas se pueden estudiar escribiendo una gramática abstracta concisa y estudiando el lenguaje que genera.
Una prueba de que una gramática G genera un lenguaje L tiene dos partes

- Se debe demostrar que toda cadena generada por G esta en L
- Se debe demostrar que toda cadena que este en L pueda ser generada por G

Eliminación de la recursión por la izquierda

Una gramática es *recursiva por la izquierda* si tiene un no terminal A tal que existe una derivación $A \stackrel{+}{\Rightarrow} A\alpha$ para alguna cadena α. Los métodos de análisis sintáctico descendente no puede manejar gramáticas recursivas por la izquierda, así que se necesita una transformación que elimine la recursión

por la izquierda. Veremos el caso general, para producciones recursivas por la izquierda $A \rightarrow A\alpha|\beta$ podían sustituirse por las producciones no recursivas por la izquierda

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'|\epsilon$

sin modificar el conjunto de cadenas derivables de A. Esta regla ya es suficiente en muchas gramáticas.

Factorización por la izquierda

La factorización por la izquierda es una transformación útil para producir una gramática adecuada para el análisis sintáctico predictivo. La idea básica es que cuando no está claro cuál de dos producciones alternativas utilizar para ampliar un terminal A, se pueden reescribir las producciones de A para retrasar la decisión hasta haber visto lo suficiente de la entrada como para elegir la opción correcta. Por ejemplo, si se tienen las producciones

$prop \rightarrow$ **if** $expr$ **then** $prop$ **else** $prop$

| **if** $expr$ **then** $prop$

al ver el componente léxico de entrada **if**, no se puede saber de inmediato qué producción elegir para expandir *prop*. En general, si $A \rightarrow \alpha\beta_1|\alpha\beta_2$ son dos producciones de A y la entrada comienza con una cadena no vacía derivada de α, no se sabe si expandir A a $\alpha\beta_1$ o a $\alpha\beta_2$. Sin embargo, se puede retrasar la decisión expandiendo A a $\alpha A'$. Entonces, después de ver la entrada derivada de α, se puede expandir A' a β_1 o a β_2. Es decir, factorizadas por la izquierda, las producciones originales se convierten en

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1|\beta_2$

2.4. ANÁLISIS SINTÁCTICO ASCENDENTE

*El algoritmo ascendente más general se denomina **análisis sintáctico LR(1)** (La L indica que la entrada se procesa de izquierda a derecha, "**L**eft-to-right", la R indica que se produce una derivación por la derecha, "**R**ightmost derivation", mientras que el número 1 indica que se utiliza un símbolo*

de búsqueda hacia adelante). Una consecuencia de la potencia del análisis sintáctico ascendente es el hecho de que también es significativo hablar de ***análisis sintáctico LR(0)****, donde no se consulta ninguna búsqueda hacia adelante al tomar decisiones de análisis sintáctico. Una optimización en el análisis sintáctico LR(0) que hace algún uso de la búsqueda hacia adelante se conoce como* ***análisis sintáctico SLR(1)*** *(sus siglas en inglés son LR(1) simple). Un método que es un poco más potente que el análisis sintáctico SLR(1) pero menos complejo que análisis sintáctico LR(1) genera se denomina* ***análisis sintáctico LALR(1)*** *(por las siglas de "lookahead LR(1) parsing', es decir, "análisis sintáctico LR(1) de búsqueda hacia adelante")".[15]*

2.4.1. Visión general

Un analizador sintáctico ascendente utiliza una pila explícita para realizar un análisis sintáctico, de igual forma como lo hacen otros tipos de analizadores no recursivos. Dicha pila tendrá tokens, no terminales y también alguna información de estado adicional que mostraremos mas adelante. La pila está vacía al inicio del análisis sintáctico ascendente y al final de dicho análisis en caso de ser exitoso tendrá el símbolo inicial. El esquema para el análisis sintáctico ascendente sería algo como esto:

$	CadenaEntrada	$
...	...	
...	...	
$ SímboloInicial		$ aceptar

donde la pila de análisis sintáctico está a la izquierda, la entrada está en el centro y las acciones del analizador sintáctico están ubicadas a la derecha (en este caso, "aceptar" es la única acción indicada).

Un analizador sintáctico ascendente tiene dos posibles acciones (aparte de "aceptar"):

1. **Desplazar** un terminal a la parte frontal de la entrada hasta la parte superior de la pila.
2. **Reducir** una cadena α en la parte superior de la pila a un no termina A, dada la selección BNF[11] $A \rightarrow \alpha$

[11]BNF o Backus-Naur Form, es la manera formal matemática de escribir un lenguaje, fue desarrollado por

Por esta razón es muy común escuchar nombre a un analizador sintáctico como **analizador sintáctico de reducción por desplazamiento**. Las acciones de desplazamiento se indican al escribir la palabra *shift* o *desplazamiento*. Las acciones de reducción se indican al escribir la palabra *reduce* o *reducción* y proporcionar la selección BNF utilizada en la reducción. Otro aspecto de los analizadores sintácticos ascendentes es que, por razones técnicas, la gramáticas siempre se aumentan con un nuevo símbolo inicial. Esto es que si S es el símbolo inicial, se agrega a la gramática un nuevo símbolo inicial S', con una producción simple correspondiente al símbolo inicial anterior:

$$S' \to S$$

Ejemplo: considere la siguiente gramática aumentada para paréntesis balanceados:

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

Un análisis sintáctico ascendente de la cadena $()$ en el que se utiliza esta gramática se proporciona en la siguiente tabla:

	Pila de análisis sintáctico	Entrada	Acción
1	$	()$	desplazamiento
2	$(	)$	reducción $S \to \epsilon$
3	$(S	)$	desplazamiento
4	$(S)	$	reducción $S \to \epsilon$
5	$(S) S	$	reducción $S \to (S)S$
6	$S	$	reducción $S \to S$
7	$S’	$	aceptar

Tabla 2.1: Acciones de análisis sintáctico de un analizador sintáctico ascendente

Un analizador ascendente puedes transferir símbolos de entrada en la pila hasta que determine qué acción realizar (suponiendo que se puede determinar una acción que no requiera los símbolos para se

John Backus para describir la sintaxis del lenguaje de programación *Algol 60*.

transferida de regreso a la entrada). Sin embargo, un analizador analizador sintáctico ascendente puede necesitar examinar más allá de la parte superior de la pila para determinar qué acción realizar.

2.4.2. Elementos LR(0) y análisis sintáctico LR(0)

Elemento LR(0)

Un **elemento LR(0)** de una gramática libre de contexto es una regla de producción con una posición distinguida en su lado derecho. Indicaremos en esta posición distinguida mediante un punto. De este modo, si $A \rightarrow \alpha$ es una regla de producción, y si β y γ son dos cadenas cualquiera de símbolos, tales como $\beta\gamma = \alpha$, entonces $A \rightarrow \beta.\gamma$ es un elemento LR(0). Éstos se conocen como elementos LR(0), porque no contienen referencia explícita de la búsqueda hacia delante.

Ejemplo: Considere la gramática desarrollada en la tabla 2.1:

$$S' \rightarrow S$$
$$S \rightarrow (S)S|\epsilon$$

Esta gramática tiene tres reglas de producción y ocho elementos: La idea detrás del concepto de un elemento es que un elemento registra un paso intermedio en el reconocimiento del lado derecho de una opción de regla gramatical específica. En particulas, el elemento $A \rightarrow \beta.\gamma$ construido por la regla gramatical $A \rightarrow \alpha$ (con $\alpha = \beta\gamma$) significa que ya se ha visto β y que se pueden derivar los siguientes tokens de entrada de γ. En términos de la pila de análisis sintáctico, esto significa que β deberá aparecer en la pare superior de la pila. Un elemento $A \rightarrow .\alpha$ significa que podemos estar cerca de reconocer una A mediante el uso de la selección de regla gramatical $A \rightarrow \alpha$ (a los elementos de esta clase se denominan **elementos iniciales**). Un elemento $A \rightarrow \alpha.$ significa que α reside ahora en la parte superior de la pila de análisis sintáctico y puede ser el controlador, si se va a utilizar $A \rightarrow \alpha$ para la siguiente reducción (tales elementos se conocen como **elementos completos**).

El algoritmo de análisis sintáctico LR(0)

$$S' \rightarrow .S$$
$$S' \rightarrow S.$$
$$S \rightarrow .(S)S$$
$$S \rightarrow (.S)S$$
$$S \rightarrow (S.)S$$
$$S \rightarrow (S).S$$
$$S \rightarrow (S)S.$$
$$S \rightarrow .$$

Tabla 2.2: Reglas de producción de la tabla 2.1

Ahora estamos listos para establecer el algoritmo de análisis sintáctico LR(0). Como el algoritmo depende del seguimiento del estado actual en el DFA de conjuntos de elementos, debemos modificar la pila de análisis sintáctico para no sólo almacenar símbolos sino también números de estado. Haremos esto mediante la inserción del nuevo número de estado en la pila del análisis sintáctico después de cada inserción de un símbolo. De hecho, los estados mismos contienen toda la información acerca de los símbolos, así que podríamos prescindir del todo de los símbolos y mantener sólo los números de estado en la pila de análisis sintáctico. Aun así, mantendremos los símbolos en la pila por conveniencia y claridad.

Para comenzar un análisis sintáctico insertamos el marcado inferior \$ y el estado inicial 0 en la pila, de manera que al principio del análisis la situación se puede representar como:

Pila de análisis sintáctico	Entrada
\$ 0	*CadenaEntrada* \$

Supongamos ahora que el siguiente paso es desplazar un token n a la pila y vayamos al estado 2. Esto se representa de la manera siguiente:

Pila de análisis sintáctico	Entrada
\$ 0 n 2	*Resto de CadenaEntrada* \$

Definición

"***El algoritmo de análisis sintáctico LR(0)***. Sea s el estado actual (en la parte superior de la pila de análisis sintáctico). Entonces las acciones se definen como sigue:

1. Si el estado s contiene cualquier elemento de la forma $A \rightarrow \alpha.X\beta$, donde X es un terminal, entonces la acción es desplazar el token de entrada actual a la pila. Si este toke es X, y el estado s contiene el elemento $A \rightarrow \alpha.X\beta$, entonces el nuevo estado que se insertará en la pila es el estado que contiene el elemento $A \rightarrow \alpha X.\beta$. Si este token no es X para algún elemento en el estado s de la forma que se acaba de describir, se declara un error.

2. Si el estado s contiene cualquier elemento completo (un elemento de la forma $A \rightarrow \alpha.$, entonces la acción es reducir mediante la regla $A \rightarrow \alpha.$. Una reducción mediante la regla $S' \rightarrow S$, donde S es el estado inicial, es equivalente a la aceptación, siempre que la entrada esté vacía, y aun error si no es así. En todos los otros casos el nuevo estado se calcula como sigue. Elimine la cadena α y todos su estados correspondientes de la pila de análisis sintáctico (la cadena α debe estar en la parte superior de la pila, de acuerdo con la manera en que se esté construido el DFA). De la misma manera, retroceda en el DFA hacia el estado en el cual comenzó la construcción de α (éste debe ser el estado descubierto por la eliminación de α=. Nuevamente mediante la construcción del DFA, este estado debe contener un elemento de la forma $B \rightarrow \alpha.A\beta$. Inserte A en la pila, e inserte (como nuevo estado) el estado que contiene el elemento $B \rightarrow \alpha A.\beta$."[15]

2.4.3. Análisis sintáctico LALR(1) y LR(1) general

EL algoritmo de análisis sintáctico LR(1)

"***El algoritmo de análisis sintáctico LR(a) general***. Sea s el estado actual (en la parte superior de la

pila de análisis sintáctico). Entonces las acciones se definen de la manera siguiente:

1. Si el estado s contiene cualquier elemento LR(1) de la forma $[A \rightarrow \alpha.X\beta, a]$, donde X es un terminal, y X es el token siguiente en la cadena de entrada, entonces la acción es desplazar el token de entrada actual a la pila, y el nuevo estado que se insertará en la pila es el estado que contiene el elemento LR(1) $[A \rightarrow \alpha X.\beta, a]$.

2. Si el estado s contiene el elemento completo LR(1)$[A \rightarrow \alpha., a]$, y el token siguiente en la cadena de entrada es a, entonces la acción es reducir mediante la regla $A \rightarrow \alpha$. Una reducción mediante la regla $S' \rightarrow S$, donde S es el estado inicial, es equivalente a la aceptación. (Esto ocurrirá sólo si el token de entrada es \$.) En los otro casos el nuevo estado se calcula del modo que se describe a continuación. Elimine la cadena α y todos sus estados correspondientes de la pila de análisis sintáctico. De la misma manera, retroceda en el DFA hasta el estado en el que comenzó la construcción de α. Por construcción, este estado debe contener un elemento LR(1) de la forma $B \rightarrow \alpha.A\beta, b]$. Inserte A en la pila, e inserte el estado que contiene el elemento $B \rightarrow \alpha A.\beta, b]$.

3. Si el siguiente token de entrada es de tal naturaleza que ninguno de los dos casos anteriores aplique, de declara un error.

Análisis sintáctico LALR(1)

El análisis sintáctico LALR(1) está basado en la observación de que, en muchos casos, el tamaño del DFA de conjuntos de elementos LR(1) se debe en parte a la existencia de muchos estados diferentes que tienen el mismo conjunto de primeros componentes en sus elementos (los elementos LR(0)), mientras que difieren sólo en sus segundos componentes (los símbolos de búsqueda hacia delante).

El algoritmo de análisis sintáctico LALR(1) expresa el hecho de que tiene sentido identificar todos estos estados y combinar su búsqeuda hacia delante. Al hacerlo así, siempre debemos finalizar con un DFA que sea idéntico al DFA de los elementos LR(0), excepto si cada estado se compone de elementos con conjuntos de búsqueda hacia delante. En el caso de elementos completos estos conjun-

tos de búsqueda hacia delante son frecuentemente más pequeños que los correspondientes conjuntos siguiente. De este modo, el análisis sintáctico LALR(1) retiene algunos de los beneficios del análisis LR(1), mientras que mantiene el tamaño más pequeño del DFA de los elementos LR(0).

El **núcleo** de un esado del DFA de los elementos LR(1) es el conjunto de elementos LR(0) que se compone de los primeros componentes de todos los elementos LR(1) en el estado. Puesto que la construcción del DFA de los elementos LR(1) utiliza transiciones que son las mismas que en la construcción del DFA de los elementos LR(0), excepto por sus efectos sobre las partes de búsqueda hacia delante de los elementos, obtenemos los siguientes dos hechos, que forman la base para la construcción del análisis sintáctico LALR(1).

- **PRIMER PRINCIPIO DEL ANÁLISIS SINTÁCTICO LALR(1)**
  El núcleo de un estado del DFA de elementos LR(1) es un estado del DFA de elementos LR(0)
- **SEGUNDO PRINCIPIO DEL ANÁLISIS SINTÁCTICO LALR(1)**
  Dados dos estados s_1 y s_2 del DFA de elementos LR(1) que tenga el mismo núcleo, suponga que hay una transición con el símbolo X desde s_1 hasta un estado t_1. Entonces existe también una transición con X del estado s_2 al estado t_2, y los estados t_1 y t_2 tienen el mismo núcleo.

Tomados juntos, estos dos principios nos permiten construir el **DFA de los elementos LALR(1)**, el cual se construye a partir del DFA de los elementos LR(1) al identificar todos los estados que tienen el mismo núcleo y formar la unción de los símbolos de búsqueda hacia delante para ada elemento LR(0) como su primer componente y un conjunto de tokens de búsqueda hacia delante como su segundo componente. [12]

[12]Los DFA de elementos LR(1), de hecho, también podrían utilizar conjuntos de símbolos de búsqueda hacia delante para representar elementos múltiples en el mismo estado en que comparten sus primeros componentes, pero se considera conveniente emplear esta representación para la construcción LALR(1), donde es más apropiado.

3. JFLEX - ANALIZADOR LÉXICO

JFlex es un analizador léxico para Java hecho en Java. JFlex es homólogo de la poderosa herramienta *Lex*[13] para C/C++, de hecho JFlex es la versión de Lex reescrita para Java.

3.1. PRINCIPALES CARACTERÍSTICAS

Las principales características del diseño de JFlex son:

- Soporte completo con caracteres unicode[14]
- Rápida generación de analizadores
- Una conveniente especificación de sintaxis
- Es independiente de la plataforma
- Es compatible con CUP (Analizador sintáctico)

3.2. ESCRIBIR UN ANALIZADOR LÉXICO CON JFLEX

3.2.1. Estructura de un archivo jflex

En teoría, un archivo jflex esta dividido en 3 secciones:

- Código de usuario
- Opciones y declaraciones
- Reglas lexicográficas

[13] **Lex** es un programa que genera analizadores léxicos ("scanners" o "lexers")

[14] **Unicode**, es una gran tabla, que en la actualidad asigna un código a cada uno de los más de cincuenta mil símbolos, los cuales abarcan todos los alfabetos europeos, ideogramas chinos, japoneses, coreanos, muchas otras formas de escritura, y más de un millar de símbolos especiales.

Para efectos de explicar como esta conformado lo dividiremos en 6 partes o fragmentos de código, la primera parte del archivo es el bloque donde se importaran los paquete que se van a utilizar para nuestro analizador, es decir, si en nuestro programa utilizaremos componentes del paquete *util* debemos importar aquí dicho paquete:

```
import java.util.*;
```

Luego sigue un par de signos de porcentaje (%) para indicar que empezará la definición del bloque de configuración del analizador. El bloque de configuración se define por el conjunto de parámetros que se especifican en el archivo para decirle a nuestro a analizador como se debe comportar, cada parámetro empieza con el símbolo % y se escribe solo uno por linea, es decir, uno de bajo del otro.

```
%unicode
%line
%column
```

En el siguiente fragmento de código podremos incluir código Java el cual podemos utilizar en el analizador, cabe notar que el código que aquí se escriba será incluido sin ninguna alteración al resultado final del analizador, dicho fragmento ira enmarcado entre las etiquetas %{ al inicio del código y %} al final del mismo.

```
%{

  public static void escribir(String cadena)
  {
    System.out.println(cadena);
  }

%}
```

El siguiente fragmento formará parte esencial dentro del funcionamiento del analizador, en este se definirán el conjunto de expresiones regulares que se utilizarán durante el proceso de análisis, a continuación se presentan unos ejemplos de este tipo de declaraciones:

```
FinDeLinea = \r | \n | \r\n
Variable = [:jletter:][:jletterdigit:]*
Si = "Si"
Entero = 0 | [1-9][0-9]*
```

En la próxima línea de código se especifican los posible estados en los que se encontrará el analizador, esta funcionalidad es muy útil a la hora de identificar el contexto en el que se esta realizando el análisis.

```
%state CADENA
```

Nuestro sexto y último fragmento de código es donde se le especificará al analizador los tokens y que acciones realizar cuando se encuentren dichos tokens, para inicial este fragmento debemos insertar nuevamente un doble símbolo de porcentaje (% %):

```
"Inicio" {System.out.println("Se encontró la palabra Inicio")}
"+" {System.out.println("Se encontró el símbolo +")}
{FinDeLinea} {System.out.println("Se encontró un final de línea")}
```

3.2.2. Opciones

A continuación presentaremos las posibles opciones que pueden especificar en un archivo jflex.

- **%class Lexer:** Esta opción le dice a JFlex que el archivo .java a generar lleve el nombre de

Lexer, aquí podremos indicar cualquier nombre.

- **%unicode:** La opción unicode nos permite trabajar con archivos que tienen este tipo de caracteres.
- **%cup:** CUP es un analizador sintactico el cual se mostrará mas adelante, esta opción nos permite integrar JFlex y CUP.
- **%line:** Le dice al analizador que lleve el conteo de la linea que se esta analizando.
- **%line:** Le dice al analizador que lleve el conteo de la columna que se esta analizando.

3.2.3. Reglas y acciones

La sección de "Reglas léxicas" de JFlex contiene expresiones regulares y acciones (Código Java) que son ejecutadas cuando el analizador encuentra cadenas asociadas a las expresiones regulares. Así como el escáner lee las entradas, también hace seguimiento a todas las expresiones regulares y activa la acción del patrón que tenga la mas grande coincidencia, por ejemplo, para la especificación anterior, la cadena *Sino* coincide en parte con la expresión *Si*, pero como la expresión regular *Variable* tiene mayor numero de coincidencias (coincide totalmente) entonces es esta ultima quien ejecuta la acción. Para el caso en que dos expresiones regulares coincidan en su totalidad con una cadena entrante entonces es el patrón que se ha especificado primero el que ejecuta la acción.

Estados léxicos Adicionalmente a las expresiones regulares, se pueden utilizar estados léxicos para hacer las especificaciones mas exactas. Un estado léxico como una condición de inicio, una cadena, entre otros. Si el escáner se encuentra en estado CADENA, entonces solo las expresiones regulares que se estén precedidas por la condición inicial <CADENA>podrán ser evaluados. La condición inicial de una expresión regular puede contener mas de un estado léxico, de ser así, dicha expresión regular se evaluará cuando el escáner se encuentre en cualquiera de esos estados iniciales. El estado léxico inicial del escáner es YYINITIAL y es con este el cual se empieza a escanear, si una expresión regular no tiene condición inicial esto quiere decir que podrá ser evaluada en cualquier estado léxico

que se encuentre el escáner.

```
<YYINITIAL>"Inicio" {System.out.println("Se encontró la palabra Inicio")}
```

Dos métodos importantes que se utilizan en el código de acción de una expresión regular son *yybegin* y *yytext*. **yybegin** se utiliza para decirle al escáner que cambie el estado léxico, por ejemplo: `yybegin(CADENA)`, esto indica al escáner que a partir del llamado de esa función el escáner se encontrará en el estado léxico CADENA, por otro lado tenemos **yytext** la cual devuelve la entrada la cual coincidió con la respectiva expresión regular.

```
{Variable} {System.out.println("Se encontró la variable "+yytext())}
```

3.3. MÉTODO DE ESCANEO

En esta sección podremos ver que el método que realiza el escaneo puede personalizarse. Se puede redefinir el nombre, el tipo que retorna el método, es posible declarar excepciones que se dejen lanzar en el código de acción de una expresión regular. Si no hay un tipo de retorno especificado, el escáner declarará el valor de retorno la clase Yytoken.

- **%function "nombre"**

  Esta línea causa que el nombre el método de escaneo tome un nombre especifico. Si la directiva %function no se encuentra presente en las especificaciones entonces el método tomará el nombre de "yylex". El modificar esta directiva implica que se debe hacer una implementación del método con el nuevo nombre en el código generado por CUP lo cual hace mas complicado la integración de los dos analizadores.

- **%Integer**

  %int

  Ambas directivas causan que el valor retornado por el método de escaneo sea de tipo `int`. Esto permitiría que las acciones de las expresiones regulares retornen valores de tipo `int` como

tokens. Para tal caso el valor por defecto del final de archivo sería `YYEOF` el cual esta declarado como `public static final int`.

- **%type "nombreDelTipo"**

  Causa que el método de escaneo retorne un valor de algún tipo de dato en especifico, esto quiere decir que las acciones de las expresiones regulares podrán retornar valores de tipo `nombreDelTipo` como tokens. Para este caso el valor por defecto de fin de archivo es `null`. Al igual, como se dijo en el primer item estos cambios dificultan la integración con el analizador sintáctico CUP.

3.4. GENERACIÓN DE CÓDIGO

La siguiente opción nos permite decidir que tipo de código generará el analizador léxico. `%pack` es el valor por defecto cuando no se especifica ninguna directiva.

- **%switch**

  Básicamente esta directiva se debe utilizar cuando consideremos que nuestro analizador tendrá mas de 300 estados, este trabaja de tal forma que no excederá los 64 KB que tiene java como límite en la Máquina Virtual, pero el rendimiento no es el mejor.

- **%table**

  Esta directiva lo que hace es producir una clásica tabla que codifica el AFD en un arreglo para dirigir el escáner. El único inconveniente es que el arreglo es cargado en memoria, por tanto es dependiente de la máquina virtual de java (64KB). Se recomienda utilizar esta directiva si se considera que existen menos de 300 estados en el analizador léxico.

- **%pack**

  %pack causa que JFlex comprima la tabla del AFD generado en una o mas cadenas de caracteres. JFlex manipula el tamaño de las cadenas para no exceder el límite permitido. El problema es que la cadenas deberán ser descomprimidas la primera vez que se cree un objeto del analizador léxico, y como es de esperarse consume algo de tiempo y recursos. Este método es recomendado para menos de 300 estados en el analizador léxico.

3.5. CODIFICACIÓN DE CARACTERES

- **%7bit**

  Permite usar entradas de 7 bits, es decir entre 0-127. Si una entrada es mayor que 127 se generará un error en tiempo de ejecución y se lanzará una excepción de tipo `ArrayIndexOutofBoundsException`.

- **%full**

  %8bit

  Permite usar entradas de 8 bits, es decir entre 0-255. Si una entrada es mayor que 255 se generará un error en tiempo de ejecución y se lanzará una excepción de tipo `ArrayIndexOutofBoundsException`.

- **%unicode**

  %16bit

  Esto significa que la entrada puede ser cualquier tipo de caracter, es decir, entre 0-65535. Para esta directiva no se produce ningún tipo de error en tiempo de ejecución.

3.6. REGLAS LÉXICAS

Las *reglas léxicas* contienen un conjunto de expresiones regulares y acciones.

3.6.1. Sintaxis

La sintaxis de las *reglas léxicas* de JFlex esta descrita por la siguiente gramática (Los símbolos terminales se encuentra entre 'comillas':

```
LexicalRules ::= Rule+
```

```
Rule ::= [StateList] ['^'] RegExp [LookAhead] Action
         | [StateList] '«EOF»' Action
         | StateGroup
StateGroup ::= StateList '{'Rule+ '}'
StateList ::= '<' Identifier (',' Identifier)* '>'
LookAhead ::= '$'| '/'RegExp
Action ::= '{'JavaCode '}'| '|'
RegExp ::= RegExp '|'RegExp
         | RegExp RegExp
         | '('RegExp ')'
         | ('!'|'~') RegExp
         | RegExp ('*'|'+'|'?')
         | RegExp "{"Number [","Number] "}"
         | '['['^'] (Character|Character'-'Character)* ']'
         | PredefinedClass
         | '{' Identifier '}'
         | '"'StringCharacter+ '"'
         | Character
PredefinedClass ::= '[:jletter:]'
            | '[:jletterdigit:]'
            | '[:letter:]'
            | '[:digit:]'
            | '[:uppercase:]'
            | '[:lowercase:]'
            | '.'
```

La gramática usa los siguiente símbolos terminales:

- **JavaCode**

  Es una secuencia que describe las especificaciones del lenguaje Java.

- **Number** Un número entero no negativo.

- **Identifier**

  Una secuencia de letras seguidas de cero o mas letras, digitos o rallas al pie (_);

- **Character**

  Una secuencia de caracteres que no sean ninguno de los siguientes:

  `| ( ) { } [ ] < > \ . * + ? ^ $ / " ~ !`

- **StringCharacter**

  Una secuancia de caracteres que no sean ninguno de los siguientes:

  `\ "`

3.6.2. Operadores en las expresiones regulares

Ahora se mostrarán los operadores que se pueden utilizar en la definición de expresiones regulares en el analizador léxico JFlex.

Sean a y b expresiones regulares, entonces:

- **a | b** *Unión*

  Es una expresión regular que encuentra todas las entradas que sean validas para *a* ó *b*.

- **a b** *Concatenación*

  Es una expresión regular que encuentra todas las entradas que sean validas para *a* seguida de *b*.

- **a*** *Cerradura de Kleene*

  Es una expresión regular que encuentra todas las entradas que sean validas para cero o mas repeticiones de *a*.

- **a+** *Iteración*

  Es una expresión regular que encuentra todas las entradas que sean validas para una o mas repeticiones de *a*. Es equivalente a `aa*`

- **a?** *Opción*

  Es una expresión regular que encuentra todas las entradas que sean validas para cero o una ocurrencia de *a*.

- **!a** *Negación*

  Es una expresión regular que encuentra todas las entradas que sean validas para cualquier expresión diferente de *a*.

- **a{n}** *Repetición*

  Es una expresión regular que encuentra todas las entradas que sean validas para exactamente n repeticiones de *a*.

- **a{n}{m}**

  Es una expresión regular que encuentra todas las entradas que sean validas para entre n y m repeticiones de *a*.

- **a**

  Es una expresión regular que encuentra todas las entradas que coincidan exactamente con *a*.

3.6.3. Precedencia de operadores

JFlex usa la siguiente precedencia de operadores estándar para expresiones regulares:

- Operadores unarios posfijos ('*', '+', '?', {n}, {n,m})
- Operadores unarios prefijos ('!', '∼')
- Concatenación (RegExp::= RegExp Regexp)
- Unión (RegExp::= RegExp 'l'RegExp)

Entonces la expresión `a | abc | !cd*` terminará convertida en `(a|(abc)) | ((!c)(d*))`.

3.7. MÉTODOS Y ATRIBUTOS DE JFLEX ASEQUIBLES EN EL CÓDIGO DE ACCIÓN

Todos los métodos y atributos asequibles para el código de acción de JFlex tiene el prefijo *yy* para evitar que al momento de que el usuario los use generen conflicto con otros métodos o atributos de las demás clases. JFlex no tiene métodos ni atributos públicos o privados, por el contrario utiliza los prefijos *yy* y *zz* para indicar que estos pueden ser usados en el código de acción o para el código interno respectivamente.

El API actual que se puede utilizar en el código de acción es:

- `String yytext()`

  Devuelve la cadena que coincidió con la respectiva expresión regular.

- `int yylength()`

  Devuelve el tamaño de la cadena que coincidió con la respectiva expresión regular.

- `char yycharat(int pos)`

  Devuelve el carácter que se encuentra en la posición pos de la cadena que coincidió con la respectiva expresión regular.

- `void yyclose()`

  Cierra el flujo de la entrada de datos, por tanto a todas las entradas siguientes arrojara fin de archivo.

- `int yystate()`

  Devuelve el estado léxico actual del analizador.

- `void yybegin(int estado)`

  Cambia el estado léxico actual del analizador por el nuevo estado especificado como parámetro.

- `int yyline`

  Contiene el número de la línea en la que se encuentra el analizador. (Solo si se incluyó la directiva %line)

- `int yychar`

  Contiene el número del carácter que se esta analizando.

- `int yycolumn`

  Contiene el número de la columna en la que se encuentra el analizador. (Solo si se incluyó la directiva %column)

4. CUP - ANALIZADOR SINTÁCTICO

CUP es un paquete utilizado para generar un analizadores LALR con especificaciones simples. CUP es el homólogo para Java del programa YACC utilizado en C, por tanto ofrece la mayoría de las funciones de YACC. De cualquier forma, CUP es para Java escrito en Java.

Antes de empezar a describir la sintaxis de un archivo CUP podemos familiarizarnos con el siguiente ejemplo:

```
import java_cup.runtime.*;

init with {: scanner.init(); :};
scan with {: scan with {:  return scanner.next_token(); :};

terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal UMINUS, LPAREN, RPAREN;
terminal Integer NUMBER;

non terminal expr_list, expr_part;
non terminal Integer expr, term, factor;

TIMES, DIVIDE, MOD; precedence left UMINUS;

expr_list ::= expr_list expr_part |
              expr_part;
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
```

```
| expr DIVIDE expr
| expr MOD expr
| MINUS expr %prec UMINUS
| LPAREN expr RPAREN
| NUMBER;
```

4.1. ESPECIFICACIÓN DE LA SINTAXIS DE UN FICHERO .CUP

Ahora despues de haber visto el pequeño ejemplo se presentará una descripción completa de las partes de un archivo .CUP.

Un archivo de entrada CUP consta de las siguientes cinco partes:

1. Definición de paquete y sentencias `import`.
2. Sección de código de usuario.
3. Declaración de símbolos terminales y no terminales.
4. Declaraciones de precedencia.
5. Definición del símbolo inicial de la gramática y reglas de producción.

4.1.1. Definición de paquete y sentencias `import`

Las especificaciones comienzan de forma opcional con las directivas `package` y `import` Estas tienen la misma sintaxis y juegan el mismo rol que el `package` y el `import` de un programa normal escrito en Java. La declaración de un paquete tiene la forma:

```
package nombre_del_paquete;
```

donde nombre_del_paquetes es el nombre del paquete al que se esta incluyendo la clase en Java. En general, CUP implementa las convenciones léxicas de Java, como por ejemplo, soporta los dos tipos

de comentarios que soporta Java.

Después de la declaración opcional de `package`, luego se pueden importar cero o mas paquetes Java. Al igual que un programa escrito en Java importar un paquete se hace de la forma:

```
import nombre_del_paquete.nombre_de_la_clase;
```

o

```
import nombre_del_paquete.*;
```

En general, la declaración del paquete indica en donde se van a incluir las clases `sym` y `parser` que son generadas por el analizador. Todos los `import` que aparezcan en el archivo fuente parecerán luego en el archivo de la clase `parser` permitiendo de esta forma utilizar las clases incluidas en el código de acción.

4.1.2. Sección de código de usuario

Después de la declaración opcional de `import` y `package`, viene una serie de declaraciones opcionales que permiten al usuario escribir código que luego hará parte del analizador generado como parte del archivo `parser`, pero separado en una clase no-pública que contendrá todo el código escrito por el usuario. Este código va incluido entre la siguiente directiva:

```
action code {: ... :};
```

en donde el código que se incluirá en el archivo `parser` será el que esta incluido entre las etiquetas `{: :}`.

Luego de la declaración del código de acción se puede hacer la declaración opcional del código del analizador el cual se agregará directamente a la clase `parser`, este código es útil cuando se va a personalizar algunos de los métodos del analizador:

```
parser code {: ... :};
```

Otra vez, el código que se copia dentro de la clase `parser` es el que se encuentra entre las etiquetas `{: :}`.

La siguiente declaración opcional es:

```
init with {: ... :};
```

Esta declaración provee el código que se va a ejecutar antes de que el analizador llame al primer token. Esta declaración es usada usualmente para inicializar el escáner con tablas y cierto tipos de datos que luego podrán ser utilizados por las acciones semánticas. En este caso, el código se escribirá en un metodo `void` que se encuentra dentro de la clase `parser`.

La siguiente declaración opcional permite indicar al analizador como debe preguntar por el siguiente token del escáner.

```
scan with {: ... :};
```

Al igual que la declaración `init` el código de este bloque se incluye en un método dentro de la clase `parser`, de cualquier forma este método deberá devolver un objeto de tipo `java_cup.runtime.Symbol`, en consecuencia con esto el código que sea incluido dentro de la declaración `scan with` deberá devolver un objeto de este tipo.

4.1.3. Declaración de símbolos terminales y no terminales

Seguido de las declaraciones de código de usuario viene la primera parte requerida de las especificaciones: la lista de símbolos. Esta declaración es la responsable de listar y asignar un tipo para cada símbolo terminal o no terminal que aparece en la gramática. Como se mostrará, cada símbolo terminal o no terminal esta representado en tiempo real por un objeto de tipo `Symbol`. En el caso de los terminales, estos son retornados por el escáner y colocados en la pila del analizador. En el caso de los no terminales reemplazan una serie de objetos `Symbol` en la pila del analizador siempre y cuando este concuerde con la parte derecha de alguna producción. Para efectos de especificar al analizador que tipo de objeto es cada símbolo terminal o no terminal, se hace de la siguiente forma:

```
terminal Nombre_de_la_clase simbolo1, simbolo2, ... ;
terminal simbolo1, simbolo2, ... ;
non terminal Nombre_de_la_clase simbolo1, simbolo2, ... ;
non terminal simbolo1, simbolo2, ... ;
```

donde Nombre_de_la_clase es la clase a la que pertenece el objeto simbolo*x*.

4.1.4. Declaraciones de precedencia

La siguiente sección que es opcional, es donde se especifica la precedencia y asociatividad de los terminales. Esta herramientas es muy útil a la hora de analizar gramáticas ambiguas. Como se muestra en el siguiente ejemplo existen tres tipos de declaraciones:

```
precedence left     terminal, terminal, ...;
precedence right    terminal, terminal, ...;
precedence nonassoc terminal, terminal, ...;
```

La coma separa los terminales que deben tener asociatividad y en ese nivel de precedencia que se esta declarando. El orden de la precedencia del mayor a menor es de abajo a arriba. En el siguiente ejemplo el producto y la división son asociativos y tiene mayor precedencia que la suma y la resta que son asociativos entre ellos.

```
precedence left  SUMA, RESTA;
precedence left PRODUCTO, DIVISION;
```

La precedencia resuelve problemas de reducción. Por ejemplo, en la entrada $3 + 5 * 3$, el analizador no sabe si reducir la pila, si por el $+$ o por el $*$, de cualquier forma, utilizando precedencia se tiene que el $*$ tiene mayor precedencia que el $+$ por lo tanto reducirá primero por el $*$.

⚠ Todos los terminales a los que no se les especifique precedencia serán tratados con la mas baja

precedencia. Si resulta un error al reducir dos de esos terminales no podrá ser resuelto y presentará conflictos en el analizador, en tal caso, este error será reportado.

4.1.5. Definición del símbolo inicial de la gramática y reglas de producción

Para definir el símbolo inicial de la gramática se utiliza la construcción `start with...;`

```
start with Prog;
```

Para ver cómo definir las reglas de producción de la gramática veamos un ejemplo:

```
Prog ::= PROG IDENT:i1 In:i2 Out:o Local:l Body:b
         {:RESULT=new Progv1(i1, i2, o, l, b); :}
       | PROG IDENT:i1 In:i2 Out:o Body:b
         {:RESULT=new Progv2(i1,i2,o,b); :} ;
```

Para definir todas las reglas de producción que tengan a un mismo símbolo no terminal como antecedente, se escribe el símbolo no terminal en cuestión (en el ejemplo, Prog), seguido de `::=` y a continuación las reglas de producción que le tengan como antecedente, separadas por el símbolo `|` (en el ejemplo hay 2 reglas de producción). Después de la última regla de producción se termina con punto y coma.

Si nos fijamos, por ejemplo, en el consecuente de la primera regla de producción, vemos que está formado por una secuencia de símbolos terminales y no terminales `(PROG IDENT:i1 In:i2 Out:o Local:l Body:b)`, algunos de los cuales llevan adyacente un símbolo de dos puntos seguido de un identificador. Recordemos que cuando se presentó la declaración de símbolos terminales y no terminales, se dijo que éstos podían tener asociado un objeto Java. Los identificadores que vienen después de un símbolo terminal o no terminal representan variables Java en las que se guarda

el objeto asociado a ese símbolo terminal o no terminal. Estas variables Java pueden ser utilizadas en la parte `{: ... :}` que viene a continuación.

Entre `{: ... :}` se incluye el código Java que se ejecutará cuando se reduzca la regla de producción en la que se encuentre dicha cláusula `{: ... :}`.

Si el no terminal antecedente tiene asociada una cierta clase Java, obligatoriamente dentro de la cláusula `{: ... :}` habrá una sentencia `RESULT=...;`. El objeto Java guardado en la variable RESULT será el objeto Java que se asocie al no terminal antecedente cuando se reduzca la regla de producción en la que se encuentre esa cláusula `{: ... :}`.

A una regla de producción también se le puede asociar directamente una precedencia. Por ejemplo:

```
expr ::= MENOS expr:e
        {: ... :}
        %prec UMINUS
```

La directiva `%prec` permite asociar una precedencia a una regla de producción equivalente a la precedencia definida en la sección de declaraciones de precedencia al terminal que viene a continuación de `%prec` (en el ejemplo UMINUS).

4.2. EJECUTANDO CUP

Como se ha mencionado, CUP esta escrito en Java. Para invocarlo se necesita un interprete Java para invocar el método estático java_cup.Main(), pasando un arreglo de $string$ que contiene las opciones. La forma mas fácil de invocarlo es directamente desde la línea de comando de la siguiente forma:

```
java -jar java-cup-11a.jar opciones erchivo_de_entrada
```

Si todo ha salido bien se deberán generar dos archivos .java, el $sym.java$ y el $parser.java$. La

siguiente es la lista de opciones que se pueden pasar al archivo que genera el código del analizador:

- **package** $name$**:** Se le especifica al analizador que las clases `sym` y `parser` serán agregadas al paquete $name$. Por defecto estas clases no serán agregadas a ningún paquete.
- **parser** $name$**:** Hace que el archivo del analizador se llame $name$ en vez de `parser`
- **symbols** $namr$**:** Hace que el archivo de símbolos se llame $name$ en vez de `sym`
- **expect** $number$**:** Por lo general el analizador resuelve problemas de precedencia, esta opción coloca un limite a ese número de errores que se pueden corregir, una vez exceda este limite el analizador se detendra por si solo.
- **nowarn:** Evita que analizador arroje mensajes de prevención o alertas (En ingles: warnings)
- **nosummary:** Normalmente, el sistema imprime una lista con cierto tipo de cosas como los terminales, no terminales, estados del analizador, etc. al final de cada ejecución. Esta opción elimina esta funcionalidad.
- **progress:** Esta opción causa que el sistema imprima pequeños mensajes indicando varias partes del proceso de la creación del analizador.
- **dump:** Esta opción causa que el sistema imprima pedazos de la gramática, estados del analizador y tablas de analisis con el fin de resolver conflictos en el analisis.
- **time:** Causa que el sistema muestre detalles de las estadisticas sobre los tiempos resultantes del analizador. Muy útil para futuros mantenimientos y optimización del analizador.
- **version:** Causa que CUP imprima la versión actual con la que se esta trabajando.

4.3. USO DE LAS CLASES JAVA

Para utilizar las clases Java obtenidas con CUP para realizar un análisis sintáctico, se seguirá el proceso descrito a continuación.

Como sabemos, el analizador sintáctico consume los tokens generados por el analizador léxico. En CUP, al crear el analizador sintáctico (que será un objeto de la clase parser creada por CUP al ejecutar java_cup.Main), se le pasa al constructor como argumento un objeto que es el analizador léxico:

```
lexer l= new ...; parser par; par= new parser(l);
```

El analizador léxico (en el código de ejemplo anterior, el objeto de la clase lexer) sólo debe de cumplir el requisito de ser de una clase Java que implemente el siguiente interfaz:

```
public interface Scanner {
    public Symbol next_token() throws java.lang.Exception;
}
```

`next_token()` devuelve un objeto de la clase Symbol que representa el siguiente Token de la cadena de Tokens que será la entrada para realizar el análisis sintáctico. El primer Token que se le pasa al analizador sintáctico al invocar next_token será el de más a la izquierda. Un analizador léxico obtenido con JLex se ajusta a estos requisitos.

Para realizar el análisis sintáctico se invoca el método `parse()`, que devuelve un objeto de la clase Symbol que representa al símbolo no terminal raíz del árbol de derivación que genera la cadena de Tokens de entrada. Si queremos obtener el objeto Java asociado a dicho símbolo no terminal, deberemos acceder al atributo value del objeto de la clase Symbol obtenido:

```
p= par.parse().value;
```

Como el objeto almacenado en el atributo value es de la clase Object, normalmente se realizará un casting para convertirlo a la clase adecuada, como por ejemplo:

```
p= (Prog) par.parse().value;
```

4.4. GESTIÓN DE ERRORES EN CUP

Cuando se produce un error en el análisis sintáctico, CUP invoca a los siguientes métodos:

```
public void syntax_error(Symbol s); public void
unrecovered_syntax_error(Symbol s) throws
                              java.lang.Exception;}
```

Una vez que se produce el error se invoca el método syntax_error. Después, se intenta recuperar el error (el mecanismo que se utiliza para recuperar errores no se explica en este documento; si no se hace nada especial, el mecanismo para recuperar errores falla al primer error que se produzca). Si el intento de recuperar el error falla, entonces se invoca el método unrecovered_syntax_error. El objeto de la clase Symbol representa el último Token consumido por el analizador. Estos métodos se pueden redefinir dentro de la declaración parser code `{: ... :}`.

En el ejemplo, se redefinen los métodos syntax_error y unrecovered_syntax_error:

```
parser code {: public void syntax_error(Symbol s) {
  report_error(''Error de sintaxis en linea '' + s.left, null);
}

public void unrecovered_syntax_error(Symbol s) throws
  java.lang.Exception {
  report_fatal_error('' '', null);
} :};
```

Se supone que el atributo left del objeto de la clase Symbol que representa cada Token tiene un valor igual al número de la línea en la que estaba el Token en el fichero de entrada del programa que está

siendo analizado.

⚠ **La siguiente lista son palabras reservadas del analizador CUP:**

code, action, parser, terminal, non, nonterminal, init, scan, with, start, precedence, left, right, nonassoc, import, package.

4.5. GRAMÁTICA DE LAS ESPECIFICACIONES DE UN ARCHIVO CUP

A continuación se mostrar la gramática que define la forma en como debería esta formado correctamente un archivo CUP:

```
java_cup_spec      ::= package_spec import_list code_parts
             symbol_list precedence_list start_spec
             production_list
package_spec       ::= PACKAGE multipart_id SEMI | empty
import_list        ::= import_list import_spec | empty
import_spec        ::= IMPORT import_id SEMI
code_part          ::= action_code_part | parser_code_part
                     | init_code | scan_code
code_parts         ::= code_parts code_part | empty
action_code_part   ::= ACTION CODE CODE_STRING opt_semi
parser_code_part   ::= PARSER CODE CODE_STRING opt_semi
init_code          ::= INIT WITH CODE_STRING opt_semi
scan_code          ::= SCAN WITH CODE_STRING opt_semi
symbol_list        ::= symbol_list symbol | symbol
symbol             ::= TERMINAL type_id declares_term |
                      NON TERMINAL type_id declares_non_term |
             NONTERMINAL type_id declares_non_term |
             TERMINAL declares_term |
             NON TERMINAL declares_non_term |
```

```
            NONTERMIANL declared_non_term
term_name_list     ::= term_name_list COMMA new_term_id
                       | new_term_id
non_term_name_list ::= non_term_name_list COMMA
                 new_non_term_id |new_non_term_id
declares_term      ::= term_name_list SEMI
declares_non_term  ::= non_term_name_list SEMI
precedence_list    ::= precedence_l | empty
precedence_l       ::= precedence_l preced + preced;
preced             ::= PRECEDENCE LEFT terminal_list SEMI
                 | PRECEDENCE RIGHT terminal_list SEMI
                 | PRECEDENCE NONASSOC terminal_list SEMI
terminal_list      ::= terminal_list COMMA terminal_id
                     | terminal_id
start_spec         ::= START WITH nt_id SEMI | empty
production_list    ::= production_list production
                     | production
production         ::= nt_id COLON_COLON_EQUALS
                       rhs_list SEMI
rhs_list           ::= rhs_list BAR rhs | rhs
rhs                ::= prod_part_list PERCENT_PREC
                        term_id | prod_part_list
prod_part_list     ::= prod_part_list prod_part | empty
prod_part          ::= symbol_id opt_label | CODE_STRING
opt_label          ::= COLON label_id | empty
multipart_id       ::= multipart_id DOT ID | ID
import_id          ::= multipart_id DOT STAR
                     | multipart_id
type_id            ::= multipart_id
```

```
terminal_id         ::= term_id
term_id             ::= symbol_id
new_term_id         ::= ID
new_non_term_id     ::= ID
nt_id               ::= ID
symbol_id           ::= ID
label_id            ::= ID
opt_semi            ::= SEMI | empty
```

5. OTRAS HERRAMIENTAS UTILIZADAS

A continuación mencionaremos otras herramientas aparte de JFlex y CUP, las cuales fueron utilizadas para lograr construir el compilador de código fuente en pseudocódigo.

5.1. C++

Al surgir la Programación Orientada a Objetos (POO), el lenguaje de programación C se vio en la necesidad de adaptarse al gran cambio que trajo consigo el nuevo paradigma de programación, de allí surgió C++. Basado en los solidos fundamentos de C, C++ añade el soporte para POO sin perder la capacidad, estilo y flexibilidad de C. Tanto, que muchos programadores ven C++ como la versión mejorada de C, sin importar si permite o no el paradigma de la programación orientada a objetos.

5.1.1. E/S Consola de C++

El hecho de que C++ sea un superconjunto de C, significa que todos los elementos del lenguaje C están contenidos en el lenguaje C++. Esto quiere decir que todos los programas en C también son por homologación programas en C++. Por ello, es posible escribir programas en C++ que luzcan como programas en C.

Una característica específica muy común utilizada por los programadores de C++ es su enfoque a la E/S[15] por consola. Aunque como se ha mencionado antes se pueden seguir usando las funciones **prinf()** y **scanf()**, C++ brinda un método nuevo y mejor de realizar estas dos operaciones. El operador de salida es $<<$ y el operador de entrada es $>>$.

5.1.2. Variables

Nombre

En C++, las variables se nombran de igual forma que en el lenguaje C, a continuación veremos la forma que debe tener el nombre de una variable en C++:

[15] **E/S**, hace referencia a las Entras y Salidas por medio de dispositivos físicos de la computadora.

- El nombre de una variable debe empezar con cualquier letra.
- El nombre de una variable solo puede contener caracteres numéricos, letras y/o guión al pie (_).
- C++ al igual que C es *case sensitive*, es to quiere decir que los caracteres en mayúscula son diferentes a los caracteres en minúscula. Es decir, `Variable1` es diferente de `variable1`.

Tipos de datos

Entre los principales y mas comunes tipos de datos de C++ tenemos:

- **int**, son variables de tipo entero, contiene solo números sin punto flotante.
- **char**, las variables de este tipo pueden contener cualquier tipo de carácter ASCCI.
- **float**, representa las variables reales con de punto flotante.

5.1.3. Estructuras de control

If - Si

La estructura de control `if` sirve para controlar la ejecución de ciertas instrucciones siempre y cuando la condición que se especifique en dicho condicional sea verdadera, es decir, las instrucciones que se encuentran dentro del bloque `if` se ejecutan si la condición es verdadera, o no se ejecuta si la condición es falsa.

If-Else - Si-Sino

En forma similar a la estructura de control `if`, esta ejecuta un conjunto de instrucciones 1 en caso de que la condición que se especifique dentro del condicional sea verdadera, en caso de que la condición sea falsa se ejecutará un conjunto de instrucciones 2.

SINTAXIS

`if(condición)`
`{`
`  Instrucciones en C++`
`}`

Tabla 5.3: Sintaxis del condicional `if` en C++.

SINTAXIS

`if(condición)`
`{`
`  Conjunto de Instrucciones 1`
`}`
`else`
`{`
`  Conjunto de Instrucciones 2`
`}`

Tabla 5.4: Sintaxis de la estructura de control `if-else` en C++.

switch - Dependiendo-De

La estructura de control `switch` sirve en el caso que se tenga una variable y se quiera ejecutar algún conjunto de instrucciones dependiendo del valor que tenga la variable que se encuentra en el `switch`. Si se tiene una variable a dentro de una estructura de control switch: `switch(a)` y n casos con n conjuntos de instrucciones, se ejecutará el conjunto de instrucciones que se encuentre en el caso que coincida con el valor de a.

SINTAXIS

`switch(variable)`
`{`
`  case 1: Instrucciones 1`
`  break;`
`  case 2: Instrucciones 1`
`  break;`
`  ...`
`  case n: Instrucciones 1`
`  break;`
`}`

Tabla 5.5: Sintaxis de la estructura de control `switch` en C++.

5.1.4. Ciclos

La primitiva algorítmica por excelencia de los ciclos repetitivos es el *Mientras Que*, el *Para* y el *Haga-Hasta* son derivados del *Mientras Que*.

while - Mientras-Que

El ciclo repetitivo *while* permite ejecutar determinado conjunto de instrucciones siempre y cuando la condición que se encuentra en el *while* sea verdadera.

for - Para

El ciclo repetitivo *for* nos permite ejecutar un conjunto de instrucciones n cantidad de veces, para ello se necesita un valor inicial, un valor final y un incremento.

SINTAXIS

`while(condición)`
`{`
`  Instrucciones en C++`
`}`

Tabla 5.6: Sintaxis del ciclo repetitivo `while` en C++.

SINTAXIS

`for(valor\_inicial;condición\_final;incremento)`
`{`
`  Instrucciones en C++`
`}`

Tabla 5.7: Sintaxis del ciclo repetitivo `for` en C++.

do-while - Haga-Hasta

El ciclo *do-while* funciona de manera similar al ciclo *while* con la diferencia que el *do-while* primero ejecuta el conjunto de instrucciones y luego evalúa la condición para decidir si volver a ejecutar las instrucciones o no.

5.2. JAVA

Java fue creado por *Sun Microsystems* fue desarrollado en el año 1991, pero solo 10 años después se convertiría en el lenguaje de programación más usado por la comunidad mundial de programadores.

Java es el lenguaje de programación que más impacto ha tenido en los últimos años, especialmente en la comunidad de programadores de servicios y aplicaciones desktop[16]. Esta especie de "Revolución

[16]Una **aplicación desktop** es aquella que se ejecuta localmente en una maquina.

SINTAXIS

`do`
`{`
`  Instrucciones en C++`
`}while(condición)`

Tabla 5.8: Sintaxis del ciclo repetitivo `do-while` en C++.

del Software" que ha producido Java lo lleva a que uno se pregunte: ¿Qué tiene Java que no tengan los demás lenguajes de programación? Como lenguaje de programación, Java no tiene mucha diferente del resto de los lenguajes de programación orientados a objetos, pero Java es más que un lenguaje y tiene varias características que lo hacen diferente.

- **Lenguaje orientado a objetos en su totalidad.**

  Todos los conceptos en los que se basa la técnica de programación orientada a objetos (POO), es decir, encapsulación, herencia, polimorfismo, etcétera, están todos ellos presentes en Java.

- **Disponibilidad de un amplio conjunto de librerías.**

  La programación de aplicaciones con Java no solo se basa en el empleo de instrucciones que componen el lenguaje, sino, fundamentalmente, en la posibilidad de utilizar el amplio conjunto de clases que Sun pone a disposición del programador y con las cuales es posible realizar, prácticamente, cualquier tipo de aplicación.

- **Aplicaciones multiplataforma.**

  Que las aplicaciones Java sean multiplataforma significa que, cuando se compila el programa, éste puede ser ejecutado en diferentes sistemas operativos sin necesidad de hacer cambios en el código fuente y sin que se debe volver a compilar el programa, es lo que en el mundo Java se conoce como "compila en un lugar y ejecuta en cualquier parte".

- **Amplio soporte de fabricantes de software.**

  Esta característica proviene del hecho que las aplicaciones Java no están vinculadas a un determinado sistema operativo.

Hoy en día, encontramos una amplia variedad de productos de software de diferentes fabricantes que dan soporte a Java, como puede ser el caso de los entornos de desarrollo o los servidores de aplicaciones.

5.2.1. Entornos de desarrollo para Java

Cuando se desarrollan aplicaciones que contienen un numero elevado de líneas código se hace necesaria la utilización de herramientas diferentes a las del SDK[17] para la compilación y ejecución del código ya que este puede resultar engorroso, además de dificultar la detección y solución de errores, tanto al momento de compilar como de ejecutar.

En esos casos resulta mucho más practico la utilización de un IDE[18]. Un IDE proporciona todos los elementos indispensables para la codificación, compilación, depuración y ejecución de programas en un entorno gráfico amigable y fácil de utilizar.

Los IDE de Java usan internamente las herramientas básicas del SDK en la realización de las operaciones, aun así, el programador no tendrá que hacer uso de la consola para ejecutar estos comandos, dado que el entorno le ofrecerá una forma alternativa de utilización, formada por menús y barras de herramientas.

Existen en el mercado una gran lista de IDE's para desarrollar aplicaciones en Java, a continuación se muestra una lista con los editores mas usados y poderosos para el desarrollo de aplicaciones:

5.2.2. Sintaxis del lenguaje

[17] **SDK**, siglas de **S**oftware **D**evelopment **K**it, es un kit de desarrollo de software es generalmente un conjunto de herramientas de desarrollo que le permite a un programador crear aplicaciones para un sistema bastante concreto.

[18] **IDE**, Entorno de Desarrollo Integrado

Entorno de desarrollo	Fabricante	Sitio Web oficial
NetBeans	Sun Microsystem	http://www.netbeans.org
Jbuilder	Borland	http://www.borland.com
Jdeveloper	Oracle	http://www.oracle.com
Eclipse	Eclipse Foundation	http://www.eclipse.org

Tabla 5.9: Entornos de desarrollo para aplicaciones Java.

Sintaxis básica

Antes de entrar en la sintaxis del lenguaje, veamos algunos aspectos sintácticos generales:

- **Lenguaje sensible a mayúsculas y minúsculas**. El compilador java hace distinción entre mayúsculas y minúsculas, no es lo mismo escribir `void` que `Void`.
- **Las sentencias finalizan con ";"**. Todas las sentencias escritas en Java finalizan con ";".
- **Los bloques de instrucciones se delimitan con llaves**.
- **Comentarios de una línea y multilínea**. En Java, un comentario de una línea va precedido por "//" mientras que los que ocupan varias líneas se delimitan por "/*" y "*/".

```
//comentario de una línea
/* comentario de
varias líneas */
```

Tabla 5.10: Comentarios en un programa Java.

Tipo de datos primitivos

Toda la información que se maneja en Java puede estar representada por un objeto o por un *dato*

básico o de tipo primitivo. Java soporta los 8 tipos de datos primitivos que se muestran a continuación: Los tipos de datos primitivos en Java se pueden categorías en cuatro grupos:

Tipo básico	Tamaño
byte	8 bits
short	16 bits
int	32 bits
long	64 bits
char	16 bits
float	32 bits
double	64 bits
boolean	-

Tabla 5.11: Tipos primitivos Java.

- **Numéricos enteros**, son los tipos *byte, short, int* y *long*. Los cuatro representan números enteros con signo.
- **Carácter**, el tipo de dato *char* representa un carácter codificado en el sistema unicode.
- **Numérico decimal**, los tipos *float* y *double* representan números decimales con punto flotante.
- **Lógicos**, el tipo *boolean* es el tipo de dato lógico, los dos únicos posibles valores que puede representar un dato lógico son *true* y *false*.

Declaración de variables

Una variable se declara de la siguiente forma:

```
tipo_dato nombre_variable;
```

Un nombre de variable válido debe cumplir las siguientes reglas:

- Debe comenzar por un carácter alfabético.
- No puede contener espacios, signos de puntuación o secuencias de escape.
- No puede utilizarse como nombre de variable una palabra reservada Java.

Válidas	No válidas
int k,cod;	boolean 7q; //comienza por un número
long p1;	int num uno; //contiene un espacio
char cad_2;	long class; //utiliza una palabra reservada

Tabla 5.12: Ejemplos de declaraciones de variables.

5.3. PHP

5.3.1. ¿Qué es PHP?

PHP es el acrónimo de Hipertext Preprocesor. Es un lenguaje de programación del lado del servidor gratuito e independiente de plataforma, rápido, con una gran librería de funciones y mucha documentación.

PHP fue creado originalmente en 1994 por Rasmus Lerdorf, pero como PHP está desarrollado en política de código abierto, a lo largo de su historia ha tenido muchas contribuciones de otros desarrolladores.

PHP logra una mezcla muy útil entre el código HTML utilizado en las páginas Web y un lenguaje de alto nivel, para dotar a las páginas Web de Dinamismo haciéndolas capaces de procesar y manejar grandes cantidades de información de una manera eficiente y sencilla.

Un lenguaje del lado del servidor es aquel que se ejecuta en el servidor Web, justo antes de que se envíe la página a través de Internet al cliente. Las páginas que se ejecutan en el servidor pueden realizar accesos a bases de datos, conexiones en red, y otras tareas para crear la página final que verá el cliente. El cliente solamente recibe una página con el código HTML resultante de la ejecución del código PHP. Como la página resultante contiene únicamente código HTML, es compatible con todos los navegadores.

PHP puede procesar la información de formularios, generar páginas con contenidos dinámicos, o enviar y recibir cookies. Y esto no es todo, se puede hacer mucho más.

Existen tres campos bases en los que se usan scripts escritos en PHP.

- Scripts del lado del servidor. Este es el campo más tradicional y el principal foco de trabajo. Se necesitan tres cosas para que esto funcione. El intérprete PHP (CGI ó módulo), un servidor Web en este caso se escogerá Apache y un navegador como Internet Explorer. Es necesario correr el servidor Web con PHP instalado. El resultado del programa PHP se puede obtener a través del navegador, conectándose con el servidor Web.

- Scripts en la línea de comandos. Puede crear un script PHP y correrlo sin ningún servidor Web o navegador. Solamente necesita el intérprete PHP para usarlo de esta manera. Este tipo de uso es ideal para scripts ejecutados regularmente desde Linux debido a que PHP fue creado originalmente para que trabajara bajo este sistema operativo, pero hoy en día PHP es libre de plataforma y también puede ser usado bajo Windows y se pueden correr scripts en el Planificador de tareas. Estos scripts también pueden ser usados para tareas simples de procesamiento de texto.

- Escribir aplicaciones de interfaz gráfica. Este uso no es el más adecuado para trabajar con PHP, Debido a que no posee un óptimo manejo de interfaces, pero si se conoce PHP y quisiera utilizar algunas de sus características avanzadas en programas clientes, puede utilizar PHP-GTK para escribir dichos programas. También es posible escribir aplicaciones independientes de una plataforma. PHP-GTK es una extensión de PHP, no disponible en la distribución principal.

Como se dijo anteriormente PHP es independiente de plataforma y puede ser utilizado en cualquiera de los principales sistemas operativos del mercado, incluyendo Linux, muchas variantes Unix (incluyendo HP-UX, Solaris y OpenBSD), Microsoft Windows, Mac OS X, RISC OS y probablemente algunos más. PHP soporta la mayoría de servidores Web de hoy en día, incluyendo Apache, Microsoft Internet Information Server, Personal Web Server, Netscape e iPlanet, Oreilly Website Pro server, Caudium, Xitami, OmniHTTPd y muchos otros. PHP tiene módulos disponibles para la mayoría de los servidores, para aquellos otros que soporten el estándar CGI, PHP puede usarse como procesador CGI.

De modo que, con PHP se tiene la libertad de elegir el sistema operativo y el servidor de su gusto. También se tiene la posibilidad de usar programación procedimental o programación orientada a objetos. Aunque no todas las características estándar de la programación orientada a objetos están implementadas en la versión actual de PHP, muchas bibliotecas y aplicaciones grandes están escritas íntegramente usando programación orientada a objetos.

Con PHP no se encuentra limitado a resultados en HTML. Entre las habilidades de PHP se incluyen: creación de imágenes, archivos PDF y películas Flash (usando libswf y Ming) sobre la marcha. También puede presentar otros resultados, como XHTM y archivos XML. PHP puede autogenerar estos archivos y almacenarlos en el sistema de archivos en vez de presentarlos en la pantalla.

PHP también cuenta con soporte para comunicarse con otros servicios usando protocolos tales como LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (en Windows) y muchos otros. También se pueden crear sockets puros. PHP soporta WDDX para el intercambio de datos entre lenguajes de programación en Web. Y hablando de interconexión, PHP puede utilizar objetos Java de forma transparente como objetos PHP Y la extensión de CORBA puede ser utilizada para acceder a objetos remotos.

5.3.2. Una explicación sencilla

Aunque PHP no solamente está diseñado para la creación de paginas Web, como se dijo anteriormente puede hacer muchas mas cosas pero este es su uso más común.

Las páginas Web que utilizan PHP son tratadas como páginas de HTML comunes y corrientes, se puede crearlas y editarlas de la misma manera que lo hace con documentos normales de HTML.

5.3.3. Instrucciones en PHP

Operadores

En PHP existen tres tipos de operadores: unarios, binarios y ternarios, los primeros actúan sobre un solo valor o expresión, los segundos operan sobre dos variables o expresiones, en este grupo se encuentran la gran mayoría de operadores, y el tercer grupo es usado para seleccionar dos operaciones en base a una tercera. PHP ofrece los siguientes operadores:

Operadores aritméticos.

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
+	Suma aritmética	Binario	$a+$b;
-	Resta aritmética	Binario	$a-$b;
*	Multiplicación aritmética	Binario	$a*$b;
/	División aritmética	Binario	$a/$b;
%	Modulo división	Binario	$a %$b;

Tabla 5.13: Operadores Aritméticos

Operador de asignación

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
=	Se define a	Binario	$a=5;

Tabla 5.14: Operador de asignación

Operadores de cadena

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
.	Concatenación	Binario	$cad=$cad1.$cad2;
.=	Concatena con lo antes guardado en la variable	Binario	$cad.=“joel”;

Tabla 5.15: Operadores de Cadena

Operadores de incremento y decremento

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
++	Incremento	Unario	$a++; ó ++$a;
–	Decremento	Unario	$a–; ó –$a;

Tabla 5.16: Operadores de incremento y decremento

Operadores de Comparación

Es necesario aclarar que en estos casos si el operador se encuentra antes de la variable trabaja de

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
==	Igual	Binario	$a==$b;
!=	Diferente	Binario	$a!=$b;
===	Idéntico(Igual y mismo tipo)	Binario	$a===$b;
!==	No Idéntico	Binario	$a!==$b;
<>	Diferente	Binario	$a<>$b;
>	Mayor que	Binario	$a>$b;
<	Menor que	Binario	$a<$b;
>=	Mayor o igual que	Binario	$a>=$b;
<=	Menor o igual que	Binario	$a<=$b;

Tabla 5.17: Operadores de Comparación

la siguiente forma: incrementa o decrementa en uno la variable y después devuelve la variable, si se encuentra después de la variable devuelve la variable y después la incrementa o decrementa.

Operadores lógicos

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
and	Y Lógico	Binario	$a and $b;
or	O Lógico	Binario	$a or $b;
&&	Y Lógico	Binario	$a && $b;
||	O Lógico	Binario	$a || $b;

Tabla 5.18: Operadores lógicos

Estructuras de control

Condicionales

Los condicionales son unas de las estructuras de control más importantes en los lenguajes de programación debido a que permiten ejecutar segmentos de código siempre y cuando se cumplan ciertas condiciones, en PHP también se encuentran disponibles de la siguiente forma:

SINTAXIS

`if (expresión)`
`{`
`  Instrucciones`
`}`

Tabla 5.19: Sintaxis de condicional if

Pero si se necesita que si no se cumple cierta condición se ejecuten otras instrucciones, se utiliza además de la estructura if la estructura else, es decir “sino”.

SINTAXIS

`if (expresión)`
`{`
`  Instrucciones PHP, Si se cumple la expresión`
`}else{`
`  Instrucciones PHP, Si no se cumple la expresión`
`}`

Tabla 5.20: Sintaxis de condicional if con else

Ciclos o bucles

Los ciclos o bucles son estructuras de control que se ejecutan repetidamente siempre y cuando se cumplan ciertas condiciones, para el programador son muy útiles debido a que le ahorra tiempo en los casos que se requiera que una serie de instrucciones se ejecuten de manera repetida hasta que se cumpla cierta condición.

While (mientras que)

Es uno de los ciclos más sencillos, se ejecutan repetidamente las instrucciones que se encuentren dentro del ciclo, siempre que la expresión que la expresión que lo acompañe sea evaluada como verdadera. El ciclo termina cuando la expresión tome un valor de falso.

SINTAXIS

`while (expresión)`
`{`
`  Instrucciones PHP, Si se cumple la expresión`
`}`

Tabla 5.21: Sintaxis del ciclo While (Mientras que)

Es muy importante que en algún momento del ciclo la expresión tome el valor de falso, por que sino, el ciclo se ejecutaría indefinidamente causando errores de programación y despilfarro de memoria en la cpu.

For (para)

Este tipo de ciclo es un poco más controlado que el While, en su declaración posee tres expresiones, la primera se ejecuta siempre y solo la primera vez que se ejecuta el ciclo, la segunda es la que tiene la potestad de terminar la ejecución del ciclo si la expresión es evaluada como falsa, y la tercera se ejecuta al final de cada realización del ciclo.

SINTAXIS

`for (expresión1; expresión2 ; expresión3)`
`{`
`  Instrucciones PHP, Si se cumple la expresión2`
`}`

Tabla 5.22: Sintaxis del ciclo For (Para)

Foreach

Esta estructura solo funciona con arreglos en caso de tratar de ser usado con otro tipo de datos generara un error. Este ciclo recorre uno a uno los elementos del arreglo asignándoles a otra variable que se encuentra presente en la declaración del ciclo.

SINTAXIS

`foreach(matriz as variable)`
`{`
`  Instrucciones PHP`
`}`

Tabla 5.23: Sintaxis del ciclo Foreach (Para cada)

5.3.4. Sesiones en PHP

Las sesiones son una herramienta que permite guardar datos cuando el navegante se desplaza de una página Web a otra sin cerrar la página que ha abierto su navegador, esto permite que datos que el usuario haya ingresado en cierto punto del sitio Web se conserven o propaguen mientras el usuario se mantenga en el sitio sin cerrar su navegador.

Actualmente en PHP para el manejo de sesiones se utilizan las siguientes funciones:

- **Session_start( )**: Crea una sesión o continúa con la actual ayudada del identificador de sesión SID que indica que sesión esta abierta actualmente.
- **Session_name( )**: Esta función permite asignar un nombre nuevo a la sesión actual

Para propagar a través de las páginas Web una sesión es necesaria la ayuda del identificador de sesión, como se dijo anteriormente, una forma de utilizarlo es la siguiente: cuando exista un link hacia otra página Web dentro del mismo sitio, de la forma <a href="pagina2.php">es necesario pasar el identificador de sesión con la ayuda de la variable de PHP, $SID que lo guarda, para esto dentro del código HTML se embebe una sentencia PHP : <a href="pagina2.php?<? echo $SID; ? >">así se pasa la información del identificador de sesión a la otra página Web.

5.4. HTML

5.4.1. ¿Qué es HTML?

El HTML o lenguaje etiquetado de hipertexto. Es un lenguaje que maneja ciertas etiquetas para definir y representar como se verán los diferentes componentes utilizados en las páginas Web, no es un lenguaje de programación como tal, solo ayuda a que ciertas características sean introducidas dentro del documento de una forma ordenada y sencilla para que el navegador Web pueda interpretarlas correctamente.

5.4.2. Sintaxis de las Etiquetas

Como se dijo anteriormente las etiquetas ayudan a incluir características de manera ordenada para los elementos que componen una página Web, La forma básica de escribir una etiqueta en HTML es la siguiente: **<etiqueta>** lo más común es que después de escribir la etiqueta sea necesario indicar que en cierto punto se requiere el uso de otra etiqueta y que la que se estaba usando anteriormente ya ha cumplido con su labor, para lo cual es necesario cerrar la etiqueta de la siguiente forma: **</etiqueta>**. Existen etiquetas que no necesitan de un cierre, pero la mayoría si lo requieren.

Algunas etiquetas poseen ciertos atributos que indican en que forma se van a comportar los elementos que se encuentren dentro de ellas, estos atributos se incluyen dentro de la etiqueta de la siguiente forma: **<etiqueta atributo1 atributo2 atributo n>** dependiendo de el número de atributos que sean validos para esa etiqueta. Para cerrar una etiqueta con atributos se utiliza el cierre normal de etiqueta: **</etiqueta>**.

5.4.3. Etiquetas HTML

HTML

La etiqueta **<HTML>** define el comienzo de un documento HTML, esta etiqueta debe estar presente en todo documento HTML al inicio del documento, y dentro de ella se deben encontrar todas las demás etiquetas antes de su etiqueta de cierre: **</HTML>**.

HEAD

La etiqueta **<HEAD>** contiene la información no visible del documento HTML, conocido como cabecera del documento y el titulo de la página Web que será visible en la barra de titulo del navegador. Su etiqueta de cierre es **</HEAD>**.

TITLE

La etiqueta **<TITLE>** contiene el titulo del documento HTML, que será visible por la barra de titulo del navegador Web. Debe ir dentro de la etiqueta <HEAD>, su etiqueta de cierre es **</TITLE>**.

BODY

La etiqueta **<BODY>** contiene todo el cuerpo del documento, todo lo que será visible por el navegador: formas, imágenes, enlaces, etc. Deberá ir siempre después de la etiqueta </HEAD>Su etiqueta de cierre es **</BODY>**.

La etiqueta <BODY>posee los siguientes atributos:

- BGCOLOR: Muestra un color de fondo para la página.
- TEXT: especifica un color para el texto de la página.
- LINK: especifica el color de los enlaces no visitados presentes en el documento.
- VLINK: especifica el color de los enlaces visitados presentes en el documento.
- ALINK: especifica el color de los enlaces cuando se hace clic sobre ellos.

P

La etiqueta **<P>** crea un párrafo con el texto contenido antes de su etiqueta de cierre que es **</P>**.

La etiqueta <P>posee el siguiente atributo:

- ALIGN: Se utiliza para especificar la alineación del párrafo, las opciones que acepta son: Right para alinear el párrafo a la derecha, Left para alinear el párrafo a la izquierda, y Center para centrar el párrafo.

FONT

La etiqueta **<FONT>** hace referencia a los atributos para la fuente de texto que se va a utilizar en el documento HTML.

La etiqueta <FONT>posee los siguientes atributos:

- FACE: Se utiliza para especificar la fuente que se quiere utilizar en el documento. Por Ejemplo: Arial, Times New Roman, Etc.

- SIZE: Se utiliza para especificar el tamaño de la fuente a utilizar. Por Ejemplo:1, 2, 3,+2,-1, Etc.
- COLOR: Se utiliza para especificar el color de la fuente utilizada, el color debe ser ingresado en formato hexadecimal. Por Ejemplo: #00cc66.

La etiqueta <FONT>solo modificara el texto que se encuentre entre ella misma y su etiqueta de cierre </FONT>.

A

Esta etiqueta da la propiedad de vincular elementos con páginas y páginas con otras páginas, por eso es indispensable a la hora de crear enlaces a otras páginas.

Sus atributos son:

- HREF: este atributo brinda la posibilidad de enlazar la página que se esta visualizando con cualquier otra página de la siguiente forma: HREF="dirección de la página a visitar".
- TARGET: indica en que página del documento HTML se quiere que se cargue el contenido de la nueva página a visitar, los posibles valores para este tributo son: _BLANK que carga la nueva página abriendo otra ventana en el navegador, _SELF que carga la página en la misma ventana que se esta visualizando.

La etiqueta de cierre es: </A>.

TABLE

Esta etiqueta inserta una tabla dentro del documento HTML, para esta etiqueta existen dos subetiquetas principales **<TR><TD>** que solo pueden ir entre la etiqueta **<TABLE>** y su etiqueta de cierre **</TABLE>**.

Los atributos validos para la etiqueta <TABLE> son:

- ALIGN: Especifica la alineación que se desea dar a la tabla. Los valores validos para este atributo son: CENTER (centro), LEFT (izquierda), RIGHT (derecha).

- BGCOLOR: Da un color de fondo a la tabla, el color debe ser introducido en formato hexadecimal para evitar complicaciones.
- BORDER: Especifica el tamaño del borde de la tabla medido en pixeles.
- BORDERCOLOR: Asigna un color a los bordes de la tabla.
- CELLSPACING: Especifica el espacio entre celdas.
- CELLPADDING: Especifica el espacio entre la celda y su contenido.
- WIDTH: Especifica el ancho de la tabla con respecto al documento, se puede medir en pixeles o en porcentaje.

TR

Esta etiqueta representa una fila para la tabla que se debe haber creado previamente su etiqueta de cierre es </TR>.

La etiqueta <TR>hereda los atributos ALIGN y BGCOLOR de la etiqueta <TABLE>, Así que pueden ser usados igual que en la etiqueta anterior.

TD

Esta etiqueta representa una columna para la tabla que se debe haber creado previamente su etiqueta de cierre es </TD>.

La etiqueta <TD>hereda de la etiqueta <TABLE>los atributos ALIGN y BGCOLOR, pero además posee los suyos propios como por ejemplo:

- COLSPAN: Indica la cantidad de celdas que se desplazaran a la derecha para transformarse en una sola.
- ROWSPAN: Indica la cantidad de celdas que se desplazaran hacia abajo para transformarse en una sola.
- WIDTH: Indica el ancho de la celda, medido en porcentaje o pixeles.

- HEIGHT: Indica el alto de la celda, medido en porcentaje o pixeles.

FORM

La etiqueta <FORM>define un formulario con los diversos campos que puede contener, su etiqueta de cierre es </FORM>.

Esta etiqueta posee los siguientes atributos que son muy importantes a la hora de procesar la información del formulario.

- NAME: Da un nombre al formulario que se va a utilizar.
- METHOD: Especifica que método se utilizará al enviar los datos del formulario, Los valores que puede tomar este atributo son GET ó POST.
- ACTION: Indica hacia donde serán enviados los datos capturados por el formulario. Puede ser una la dirección de una página Web u otro tipo de función.

INPUT

Esta etiqueta permite crear cada uno de los campos que de los que se compone el formulario, esta etiqueta posee los siguientes atributos:

- TYPE: Indica el tipo del elemento que se quiere utilizar, los valores permitidos para este atributo son: Checkbox, Radio, hidden, submit, reset, Password, Text, Button, File.
- NAME: Indica el nombre especifico de cada campo, debido a que cuando se reciben los valores del formulario se recopilan como nombre=valor.
- VALUE: Indica un valor para determinado campo.

SELECT

La etiqueta <SELECT>crea una lista o menú desplegable que permite al usuario seleccionar una o varias opciones, para agregar una opción al menú es necesario utilizar la etiqueta <OPTION>para cada opción dentro de la etiqueta <SELECT>y antes de su cierre la etiqueta </SELECT>.
La etiqueta <SELECT>posee el atributo NAME, el cual es obligatorio.

OPTION

La etiqueta <OPTION>añade una nueva opción a la lista, la lista tendrá tantas opciones como se desee, pero es necesario que las opciones siempre se encuentren entre la etiqueta <OPTION>y su cierre </OPTION>.

Esta etiqueta tiene un atributo VALUE que guarda la el valor que se quiere para la opción elegida. Por ejemplo si el valor deseado para la opción BARRANQUILLA es CIUDAD_1 el código para la opción sería <OPTION VALUE =“CIUDAD_1”>BARRANQUILLA</OPTION>.

TEXTAREA

La etiqueta <TEXTAREA>crea un área de texto donde el usuario puede ingresar una cadena de texto un poco más larga que la ingresada con el campo <INPUT>del tipo texto. Su etiqueta de cierre es </TEXTAREA>.

Posee los siguientes atributos:

- NAME: Indica el nombre del campo área de texto.
- ROWS: Indica el numero de filas del campo área de texto.
- COLS: Indica el numero de columnas del campo área de texto.
- READONLY: Indica que el usuario solo puede leer lo que este en el campo y no puede modificarlo.

5.5. JAVASCRIPT

Los formularios son un componente esencial para en un sitio Web, debido a que permiten la interacción con el usuario, mediante un formulario se puede recoger información acerca de sus gustos, opiniones, información acerca de pedidos, en general lo que el usuario necesite transmitir mediante la página Web, esta información se debe procesar en el servidor Web para su correcta utilización e interpretación.

Para procesar la información en el servidor Web se debe utilizar un lenguaje del lado del servidor en este caso PHP, pero hay procesos que se pueden realizar en el cliente como validaciones, y en general cualquier tipo de interacción con los campos que componen el formulario, para esto la herramienta más adecuada con la que se cuenta es sin lugar a dudas JAVASCRIPT que es un lenguaje del lado del cliente, que no solo permite validar los datos del formulario antes de ser enviados al servidor, sino interactuar con cada uno de los campos que lo componen y así modificar sus propiedades haciendo más útiles sus funciones.

5.5.1. Encabezado JavaScipt

El código de JavaScript también se escribe junto con el código HTML en la pagina Web, pero en este caso como es un lenguaje del lado del cliente el código si esta visible para el usuario. La porción de código de JavaScript debe ir dentro de las etiquetas: <SCRIPT>y su etiqueta de cierre </SCRIPT>.

Esta etiqueta indica la utilización de un script que generalmente es JavaScript, pero puede ser utilizada para indicar porciones de código de otros lenguajes, es por esto que es recomendable escribir la etiqueta con el siguiente encabezado:

<SCRIPT language="JavaScript"type="text/javascript» y su etiqueta de cierre se conserva como en el caso anterior, que indica que después de este punto no hay más código JavaScript </SCRIPT>.

5.5.2. El Objeto Form

El objeto form es considerado en JavaScript como un sub-objeto del objeto document, y este a su vez

forma parte de otro objeto predefinido window, por lo que la sintaxis para acceder las propiedades o métodos del objeto form es la siguiente:

window.document.forms.nombre_formulario.nonbre_campo.propiedad

En la que se pueden omitir las cadenas forms y window, debido a que el navegador considera al formulario particular un objeto por sí mismo.

Además, el objeto forms dentro del objeto documents posee dos sub-propiedades:

- Index: Hace referencia a un arreglo que contiene todos los formularios de una página, la sintaxis es la siguiente: document.forms [index].
  Donde se debe tener en cuenta que el primer formulario viene identificado por el índice 0, por ejemplo para acceder al primer formulario de una página se deberá utilizar la expresión: document.forms [0].
- Length: Que contiene el número de formularios que hay en la página, y cuya sintaxis es: document.forms.length.

La forma más sencilla para acceder a un campo del formulario es la siguiente:

document.forms.FORM_NAME.CAMPO.(PROPIEDAD ó FUNCIÓN)

Por ejemplo si se tiene un formulario de nombre joel que contiene un campo llamado texto1 y se desea asignarle un valor determinado utilizando JavaScript, se escribiría lo siguiente: document.forms.joel.texto1.value="valor que se quiere asignar".

5.5.3. Eventos

JavaScript ofrece el uso de diferentes eventos para que sean utilizados de diversas formas, los más conocidos son los siguientes:

- OnChange: Este evento capta cuando a cambiado el valor de cierto campo.

- OnClick: Este evento capta cuando se ha hecho click en determinado campo.
- OnFocus: Este evento capta cuando se ha seleccionado determinado campo, cuando tiene foco.
- OnBlur: Este evento capta cuando se ha cambiado de campo o ha perdido el foco.
- OnMouseOver: Capta cuando se ha pasado el puntero del mouse por determinado campo.
- OnMouseOut: Capta cuando se ha quitado el puntero del mouse de determinado campo.

5.5.4. Variables en JavaScript

El comportamiento de las variables en JavaScript es similar al comportamiento en PHP, no es necesario declarar el tipo de variable que se va a utilizar, aunque puede aceptar los tipos ya conocidos: enteros, cadenas, float, etc. Solo hay que anteponer antes del nombre de la variable la frase var. Tampoco es necesario inicializar dichas variables.

5.5.5. Operadores En JavaScript

En JavaScript la mayoría de los operadores actúa de la misma forma que en PHP y otros lenguajes de programación, salvo pocas diferencias como en el caso de la concatenación de cadenas en la que el operador utilizado es (+).

OPERADOR	SIGNIFICADO	TIPO	SINTAXIS
+	Suma aritmética o concatenación	Binario	a+b ; “hola” + “como estas”
-	Resta aritmética	Binario	a-b
*	Multiplicación aritmética	Binario	a*b
/	División aritmética	Binario	a/b
%	Modulo división	Binario	a %b
=	Asignación	Binario	a=5
!=	Diferente	Binario	a!=b
==	Igual	Binario	a==b
>=	Mayor o igual que	Binario	a>=b
<=	Menor o igual que	Binario	a<=b
++	Incremento	Unario	a++
–	Decremento	Unario	a–

Tabla 5.24: Sintaxis y Ejemplos de Operadores en JavaScript

5.5.6. Estructuras De Control En JavaScript

JavaScript también posee estructuras de control que resultan muy útiles para el programador a la hora de realizar ciertas tareas. Su comportamiento es similar al de las estructuras de control en PHP y en general a las de cualquier otro lenguaje de programación como C ó Java, por eso solo se presentaran las sintaxis acompañadas de ejemplos.

Condicionales

En JavaScript el uso y la sintaxis de los condicionales es análogo al uso de los mismos en PHP, se utilizan de la siguiente forma:

SINTAXIS

`if (expresión){`
`  Instrucciones, en caso de`
`  que se cumpla la expresión`
`}else{`
`  Instrucciones, en caso de que`
`  no se cumpla la expresión`
`}`

Tabla 5.25: Sintaxis de Condicionales en JavaScript

Bucles o Ciclos en JavaScript

Como se menciono anteriormente el uso de las estructuras de control en JavaScript es casi el mismo que en PHP y otros lenguajes y los ciclos no son la excepción.

While (Mientras Que)

Su sintaxis es la siguiente:

SINTAXIS

`while (expresión)`
`  Instrucciones, en caso de`
`  que se cumpla la expresión`
`}`

Tabla 5.26: Sintaxis del Ciclo While en JavaScript

for (Para)

La sintaxis para este ciclo es:

SINTAXIS

`for (expresión1; expresión2 ; expresión3)`
`  Instrucciones, en caso de`
`  que se cumpla la expresión2`
`}`

Tabla 5.27: Sintaxis del Ciclo For en JavaScript

Funciones en JavaScript

El manejo de funciones en JavaScript es semejante al manejo de funciones en PHP su sintaxis es la siguiente:

SINTAXIS

`function nombre_funcion(parametros){`
`  Instrucciones`
`}`

Tabla 5.28: Sintaxis de Funciones en JavaScript

Una función se puede llamar con el disparar de un evento, por ejemplo presionando un botón, o desde otra función como en el ejemplo anterior. Pueden o no retornar valores.

5.5.7. Funciones Importantes En JavaScript

- **Submit():** Envia el formulario correspondiente, se utiliza de la siguiente forma: document.forms.nombre_formulario.submit(), por ejemplo:
  document.forms.joe.submit().

- **Reset():** Borra los datos que se han ingresado en el formulario, su sintaxis es la siguiente: document.forms.nombre_formulario.reset(), por ejemplo:
  document.forms.joe.reset ().

- **Confirm:** Muestra en pantalla una ventana con una cadena de texto y dos botones uno de aceptar y otro de cancelar para que el usuario elija lo que quiere hacer, en caso de elegir aceptar devuelve además un valor de verdadero y si se elige cancelar devuelve un valor de falso . Se utiliza de la siguiente forma: confirm ("cadena de texto"), por ejemplo confirm ("Se van a enviar los datos, ¿esta usted de acuerdo?").

- **Alert:** Muestra en pantalla una ventana con un aviso, es muy útil para interactuar con el usuario. Se utiliza de la siguiente forma: alert ("cadena de texto"), por ejemplo alert (" el campo de texto esta vacío, digite algo en el campo!").

5.6. CSS

CSS[19], son hojas de estilo representan un avance importante para los diseños de páginas web, ya que les dan una gran variedad de posibilidades para mejorar el aspecto de estas. Las hojas de estilos surgieron, ya que la gente para ese entonces estaba más preocupada por el contenido de sus páginas que por su presentación. .A medida que la Web era descubierta por personas con distintas visiones, HTML se vio bastante limitado y se convirtió en fuente de continua frustración, para contrarrestar esto sus autores se vieron forzados a superar las limitaciones estéticas del HTML. Aunque las intenciones han sido buenas (mejorar la presentación de las páginas web), las técnicas para conseguirlo han tenido efectos secundarios negativos. Entre estas técnicas, que dan buenos resultados para algunas personas, algunas veces, pero no siempre y no para todas las personas, se incluyen:

- La utilización de extensiones propietarias del HTML

- La utilización de extensiones propietarias del HTML

- Conversión del texto en imágenes

[19] Siglas en Inglés de Cascade Style Sheet

- Utilización de imágenes para controlar el espacio en blanco
- La utilización de tablas para la organización de las páginas
- Escribir programas en lugar de usar HTML[20].

Dichas técnicas aumentan la complejidad de las páginas web, por lo ofrecen una flexibilidad limitada. Las hojas de estilo resuelven estos problemas al mismo tiempo que no limitan el rango de mecanismos para la presentación en HTML. Con las hojas de estilo es más fácil especificar el tipo de fuente al texto.

Adicionalmente, Las hojas de estilo tienen dos ventajas principales:

- **La hoja de estilo es un archivo único y centralizado**, que se puede aplicar a tantos archivos HTML como se desee.
- **Las propiedades gráficas se definen una sola vez en la hoja de estilo**, sea cual sea el número de veces que se aplican en el HTML.

[20]W3C Recomendation "Hojas de estilo ". Fecha de Consulta: Septiembre 30 de 2008. [On-Line] URL: http://html.conclase.net/w3c/html401-es/present/styles.html

6. CONCLUSIONES

Este proyecto ha dejado varias cosas claras y también ha permitido elevar la vista y mirar más allá de lo que se ve en el momento. Pudimos darnos cuenta que es posible desarrollar un compilador de un lenguaje de programación totalmente adaptado a nuestras necesidades basándonos en la teoría de compiladores.

Por otro lado, podemos concluir que para realizar un compilador no es necesario iniciar desde cero sino que se pueden tomar un conjunto de herramientas e integrarlas de cierto modo que cada una cumpla una función especifica dentro del compilador para así poder lograr el objetivo final que es realizar un programa completo capaz de compilar código fuente de cualquier tipo de lenguaje de programación.

El logro mas interesante de este proyecto es haber podido compilar el código fuente en pseudocódigo de manera remota por medio de una página Web. Esto nos muestra y nos confirma la gran teoría que en ocasiones parece un mito: “En el futuro todo se hará por medio de Internet”. El hecho de que no sea necesario tener un compilador instalado en tu máquina para compilar un programa y que lo puedas hacer desde una máquina remota en cualquier lugar del mundo que tenga acceso a Internet abre un nuevo camino en la utilización de herramientas de programación, editores de código, compiladores, etcétera.

Bibliografía

[1] ADELL, J. La Internet como Telaraña: El World-Wide Web [artículo en internet] 2007 Disponible en: <http://www.uv.es/∼biblios/mei3/Web022.html>[acceso el 26 de septiembre de 2008].

[2] ALFRED V. Aho, Compiladores Principios, Técnicas y Herramientas, Addison-Wesley 1998, ISBN 968-444-333-1

[3] _____, Foundations of Computer Sciece, W.H. Freeman and Company 1995, ISBN 0-7167-8284-7

[4] _____, The design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company 1974, ISBN 0-201-00029-6

[5] ALVAREZ, R; LENGUAJE HTML Y PHP 2003, Disponible en: http://www.desarrolloweb.com/articulos/303.php?manual=12 [acceso el 1 de octubre de 2008]

[6] CABERO ALMENARA Julio. Nuevas Tecnologías, Comunicación y Educación [artículo en internet] 1996 Disponible en: <http://www.ull.es/departamentos/didinv/tecnologiaeducativa/doc-cabero.htm>[acceso el 25 de septiembre de 2008].

[7] Definición de PHP [Página Oficial de PHP] 2007 Disponible en: <http://www.php.net/>[acceso el 29 de septiembre de 2008]

[8] GERWIN Klein, JFLEX User's Manual. <http://www.jflex.de/manual.pdf>. Acceso el 28 de agosto de 2008

[9] GUITIERREZ Abraham, PHP5 a través de ejemplos, AlfaOmega 2005, ISBN 970-15-1083-6

[10] HARRIR Andy, HTML, XHTML, and CSS, Wiley Publishing 2008, ISB 978-0-470-18627-5

[11] HARVEY M. Deitel / Paul J. Deitel, Cómo programar en Java, quinta edición, Editorial Pearson 2004, ISB 970-26-0518-0

[12] Hipertexto e hipermedia en la enseñanza universitaria [artículo en internet] 2007 Disponible en: <http://www.sav.us.es/pixelbit/articulos/n1/n1art/art12.htm>[acceso el 27 de septiembre de 2008].

[13] INSTITUTO COLOMBIANO DE NORMAS TÉCNICAS, ICONTEC. Normas Técnicas Colombianas: Presentación de Tesis, trabajos de grado y otros trabajos de investigación. 5ta Actualización. Bogotá, ICONTEC 2002, 34 p., NTC 1486.

[14] JACOBO Pavón Puertas, Creación de un portal con PHP y MySQL, AlfaOmega Grupo Editor 2005, ISBN 970-115-1082-8

[15] KANNETH C. Louden, Construcción de Compiladores Principios y práctica, Editorial Thomson 2004, ISBN 970-686-299-4

[16] LEE Babin, Introducción a Ajax con PHP, Grupo ANAYA 2007, ISBN 978-84-415-2200-8

[17] LERDORF Rasmus, TATROE Kevin, PROGRAMMING PHP, O'Reilly & Associates, 2002, 524p.

[18] MARK Allen Weiss, Estructuras de Datos en JAVA, Addison Wesley 2000, ISBN 84-7829-035-4

[19] MICHAEL S. Jenkins, Abstract Data Types in JAVA, MacGraw-Hill 1998, ISBN 0-07-913270-7

[20] NIEDERST Jennifer, Web Design, O'Reilly Media 2006, ISBN 1-80033-012-6

[21] NIELSEN, Jackop. Workshop: Cómo hacer un test de usabilidad de un sitio [en línea] 2007 Disponible en: <http://www.gaiasur.com.ar/infoteca/siggraph99/test-de-usabilidad-de-un-sitio.html>[acceso el 29 de septiembre de 2008]

[22] SCHMITT Christopher, CSS, Grupo ANAYA 2007, 978-84-415-1954-1

[23] SCOTT E. Hudson, CUP User's Manual. <http://www2.cs.tum.edu/projects/cup/manual.html>. Acceso el 29 de agosto de 2008

[24] STEVEN S. Muchnick, Compiler Design and Implementation, Morgen Kaufmann Publishers 1997,ISBN 1-55860-320-4

[25] THOMAS Pittman / Jame Peters, The Art of Compiler Design Theory And Practice, Prentice-Hall 1992, ISBN 0-13-048190-4

[26] THOMSON Laura, WELLING Luke, DESARROLLO WEB CON PHP Y MYSQL, Anaya Multimedia, 2003, 912p

[27] VALADE Janet, PHP & MySQL Web Development, Wiley Publishing 2008, ISBN 978-0-470-16777-9