01132
84

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

DISEÑO Y DESARROLLO DE UN COMPILADOR PARA UN LENGUAJE SIMBOLICO.

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A N :

JOSE LUIS ROMERO CAMARENA
LILIANA MURILLO GRANADOS
ENRIQUE LEON MAZON
ELIZABETH HERNANDEZ CHAVEZ
SERGIO PABLO FABIAN

DIRECTOR DE TESIS: ING. NORMA ELVA CHAVEZ

JUNIO DEL 2003

A

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

DEDICATORIAS

Y

AGRADECIMIENTOS

Autorizo a la Dirección General de Bibliotecas de la UNAM a difundir en formato electrónico e impreso el contenido de mi trabajo recepcional.
NOMBRE: Liliana Murillo Granados
FECHA: 12-JUN-03
FIRMA: Liliana Murillo G.

TESIS CON
FALLA DE ORIGEN

Dedicatorias de José Luis Romero Camarena

El esfuerzo de elaboración de este trabajo lo dedico a:

Aurora (q.e.p.d.) y Gustavo, mis padres, por que ellos me dedicaron una parte importante de su vida y en agradecimiento a todo el amor que me dieron.

A Gustavo, David, Víctor y Adriana, mis hermanos, que me han enseñado mucho más de lo que ellos piensan.

A Fanny, mi amorosa mujer, camarada, amante, compañera, amiga y cómplice, porque me halló entre computadoras cuando yo estaba perdido y desde entonces vamos haciendo el mismo camino juntos.

A Aviva y David. Sin yo saberlo, ellos fueron las personas para las que hice este programa. ¡Va para ustedes, niños!

A mis amigos de entonces y de siempre, de los que nunca me he alejado y que nunca se alejaron (me refiero a ustedes, Gabriel y Pepe).

A mis maestros de la Facultad y a mis maestros de la vida; a ambos por igual porque son una cosa y la otra a la vez.

A Niklaus Wirth y Alan Turing, en humilde agradecimiento a un par de obras geniales que cambiaron radicalmente mi vida.

Al Dr. Enrique Calderón Alzati, hombre visionario, y a su sentido del humor. Le agradezco públicamente sus enseñanzas y le reconozco su inagotable talento.

A mis compañeros del Grupo de Tesis: Elizabeth, Liliana, Enrique y Sergio. ¡Somos un grupo genial !

¿Y porqué no? A la primera computadora que programé en mi vida: una venerable Burroughs 6700, al universo que en ella descubrí y al que no dejo de asomarme todos los días de mi vida en otras computadoras, pero universo al fin.

C

Dedicatorias y Agradecimientos de Liliana Murillo Granados

Primero que nada gracias a Dios quien me ha dado la fortaleza para seguir adelante y agradezco su ayuda en los momentos difíciles de mi vida.

Gracias mamita por darme tanto apoyo incondicional, por tu cariño, tu ternura, tus regaños, por tu amor, por todos esos detalles que hacen que seas única, también gracias por quererme y aceptarme como soy. Has sido mi ejemplo a seguir (aunque algunas veces me desvío) pero tienes toda mi admiración y respeto como mujer, como mamá y como amiga. Gracias a ti pude realizar este trabajo e hiciste que nuevamente confiara en mí y pude por fin terminar lo que hace años empecé..con cariño te dedico esta tesis.

Gracias a mi hermanito Sergio por todo su apoyo y darme lecciones de vida con el solo hecho de levantarse todos los días, eres mi héroe.

Gracias a mi otro hermanito Eduardo por echarme porras, en su muy particular forma y por enseñarme que no solo yo la riego, aunque no parezca te quiero muchísimo.

Irma..ahora sé lo que es tener una hermana...gracias.

A mi abuelito José (q.e.p.d) porque fue el primero en tener la paciencia de explicarme el mundo de las matemáticas y me enseño a tener fe en mí.

A mi abuelito Thomás (q.e.p.d)..gracias por tu cariño...y por enseñarme que no todo en la vida es estudiar, te extraño.

A mis amigos: Claudia (amigas por siempre), Manuel, Cecilia, Amelia, Adolfo, Eduardo Carrasco (te admiro), Eli Flores, Israel Corona, Héctor Narváez, Silvia, Elizabeth (eres única), Bertha e Iván (amigos en todo momento).

A toda mi familia de Veracruz va por ustedes.....ahora si ya soy Inge.

A Carlos...

A mi querida Universidad Nacional Autónoma de México, por hacer de mí un ser integral y por abrir un mundo nuevo en mi vida.

A mi querido profesor Ing. Eric Castañeda de la Isla Puga quien me enseño que lo difícil puede ser un reto y que todos tenemos la capacidad de aprender si nos lo proponemos, mi mayor admiración como profesor y como ser humano. Gracias por haber sido mi guía cuando la desesperación llegaba.

Al Ing. Villalobos, sin usted no lo habríamos logrado..mil gracias

Gracias a mis compañeros de tesis, joche luigi, enriquiux, lizonja, sergiopafa, por fin ...

TESIS CON
FALLA DE ORIGEN

D

A Dios :

Por permitirme transitar en este espacio.

A mi país:

Por permitirme ser MEXICANO !!!!!!

A la UNAM:

Por darme una formación, un modo de pensar y una carrera.

A mis Padres:

Por su apoyo incondicional, moral y económico, por su empuje y por creer siempre en mi a pesar de las circunstancias.

A mis hermanos:

Por ser mis compañeros de juegos y de aventuras, por regalarme sus sonrisas y sus tristezas y por la confianza que depositaron en mí.

Gloria:

Por ser toda una señora, en la extensión de la palabra, gracias por tus consejos y por tu apoyo. Pero sobre todo gracias por la dicha de haberme dejado conocerte. Te la prometí y aquí está, y desde donde estés, sé que estarás tan feliz como yo.

Ara:

Por tu comprensión y por tu espera, por tu apoyo como esposa, como mujer y como amiga, gracias por tus horas de desvelo, por tus cuidados, tus caricias y tus besos.

A mis amigos:

Que con sus ocurrencias y sus desatinos hacen mi vida más ligera.

A mis enemigos:

Gracias por perder su confianza en mí, gracias por sus malas vibras, gracias por desear que esto no fuera posible, por que gracias a todo eso me hicieron más fuerte y más seguro de mi mismo y provocaron que esto saliera mejor de lo que esperaba.

A la Facultad de Ingeniería:

Por permitirme ser un orgulloso ingeniero, por darme "posada" y abrigo durante tanto tiempo.

Enrique León Mazón

TESIS CON
FALLA DE ORIGEN

DEDICATORIAS DE ELIZABETH HERNÁNDEZ CHÁVEZ

El presente trabajo, es ofrecido como un profundo reconocimiento, al enorme esfuerzo de mis padres, quienes me apoyaron para concluirlo.

A mi padre, El hombre más noble que jamás conocí.

A mi madre con amor y respeto.

Elizabeth.

AGRADECIMIENTOS DE SERGIO PABLO FABIAN

A mi esposa Evelyn, por su amor y comprensión y por prestarme parte de su tiempo para terminar este trabajo. Por que me ha dado dos hermosas hijas y porque luchamos juntos en la búsqueda de un hogar perfecto. Sin su apoyo hubiera sido más difícil continuar. Gracias Amor, te Amo.

A mis hijas Evelyn Mariana y Jimena por que son el motivo principal que me impulsó a terminar lo que un día inicié. Porque son parte de mi, que hizo que concluyera mi trabajo de tesis. Las Amo.

A mis padres Álvaro y Catalina por que han sabido guiarme por buen camino en la vida y por el apoyo incondicional que he tenido de cada uno. En las buenas y en las malas su cariño y amor están conmigo y me cobijan para salir siempre adelante. Los Amo.

A mis hermanos Mercedes y Jorge por que siempre confiaron en mi, por que convivimos muchos años juntos y por que espero no defraudarlos nunca como hermano mayor. Los amo

A mis suegros Moises y Marcelina y a mis cuñados Vania, Agustín y Carola por haber aceptado ser parte de mi familia. Los quiero.

A todos mis **amigos y familiares** que siempre me dieron su cariño y amistad incondicional.

A la **Universidad Nacional Autónoma de México** por haberme aceptado como hijo suyo y a la **Facultad de Ingeniería** por que en ella pase los años más importantes de mi vida como estudiante y que me ha dado todo lo que ahora soy como profesional.

A mis Maestros por su sabiduría y energía para enseñar, particularmente a nuestra directora de Tesis, **Norma Elba Chávez.**

A mis Compañeros de tesis porque juntos hemos logrado el objetivo de este trabajo.

A Dios, por todo lo anterior. Gracias Señor.

SINCERAMENTE
Sergio

TESIS CON
FALLA DE ORIGEN

ÍNDICE

TESIS CON
FALLA DE ORIGEN

TESIS CON
FALLA DE ORIGEN

TESIS CON
FALLA DE ORIGEN

INTRODUCCIÓN

Hoy en día se sabe, que el desarrollo de programas es una de las ramas de la computación que más se explota, y para ello se necesita de gente experta en la creación de programas de toda índole, pero, ¿que tan difícil es aprender a programar?, algunos coinciden que la manera más fácil de aprender es practicando, pero aún así no es del todo fácil, ya que pensamos que se necesita algo más que sólo práctica, tal como un sentido lógico, y un gran gusto por innovar.

Por ello es que hemos pensado, en base a la idea original de Richard Pattis, presentar en este trabajo de tesis una opción para que cualquier persona pueda aprender a programar de una manera fácil y sencilla, y por este medio proporcionar un método que estimule la imaginación y creatividad en el desarrollo de programas.

El sistema a desarrollar, incorporará los aspectos básicos para que el propio usuario cree y dé mantenimiento a sus programas. La forma mas personal de usar un lenguaje de programación es precisamente aprender un lenguaje de desarrollo sencillo y que permita aplicarlo a algunos problemas interesantes, así como generar soluciones prácticas, es decir, sus propias soluciones.

El lenguaje que se propone, carece de variables, constantes, operadores, estructuras aritméticas o de expresiones algebraicas, lo cual lo clasificaría como un lenguaje simbólico. El hecho de ser un lenguaje simbólico lo hace muy simple de aprender, sin embargo posibilita la creación de programas elaborados capaces de resolver problemas complejos.

Hemos llamado **PATHFINDER** al lenguaje que se va a diseñar, por la semejanza en la operación del robot que se envió a Marte, y por su similitud con las acciones que realiza.

Diseñar e implementar un lenguaje que además de permitir el manejo de un pequeño robot virtual como lo es Pathfinder utilizando instrucciones fáciles, se convierta en una herramienta fundamental para apoyar el aprendizaje de la programación desde un punto de vista sencillo no es fácil.

Entre los principales puntos de este requerimiento se encuentra el desarrollo de un lenguaje simbólico el cual permita programar al robot con estructuras y comandos. Esto sin lugar a dudas permitirá una mejor comprensión del concepto de la programación y por supuesto del proceso de compilación de un lenguaje de alto nivel.

Otro requerimiento es el ambiente en el cual se debe desenvolver Pathfinder. Pattis, en su obra, proponía originalmente, un plano cartesiano en el cual existen calles y avenidas. Para el caso de

TESIS CON
FALLA DE ORIGEN

nuestro proyecto no será el cruce de coordenadas lo que nos dirá la posición del Pathfinder, si no una matriz de cuadros sobre la que se desplazará, y definirá a su mundo como Planeta ó Terreno.

Además, tenemos la ventaja de utilizar gráficos. El uso de gráficos permite al usuario final la conceptualización del código que se programa, y por medio de una ventana de programación en tiempo real ve exactamente que es lo que sucede durante la ejecución del mismo.

El propósito de éste tópico, es crearle al usuario un ambiente amigable y común. Con esto, el usuario dispondrá de una herramienta sencilla de operar y que no requerirá de un alto conocimiento de conceptos de programación avanzada.

Se ha decidido, darle vida a este robot virtual, creando un compilador del lenguaje que se llamará ***Lenguaje ULP*** (Un Lenguaje de Programación) y creando también un sistema que sea capaz de ejecutar la compilación de programas fuentes escritos en este lenguaje. El resultado será la aplicación que se ofrece en este proyecto.

Para poder compilar programas escritos en ULP, será necesario crear una aplicación que incluya las siguientes fases:

- **Análisis**. En esta fase se comprobará que el programa fuente funcione correctamente, siguiendo las reglas que definen el lenguaje en los distintos niveles (léxico, sintáctico, semántico).
    - *Léxico*. los símbolos que se pueden usar en el lenguaje.
    - *Sintáctico*: las construcciones válidas de símbolos.
    - *Semántico*: significado de las construcciones.

- **Traducción**: se realiza la traducción propiamente dicha hacia un código objeto o bien hacia un código ejecutable por una máquina ejecutora de este código. Si bien se puede optar por efectuar una *interpretación* del código, se tratará de realizar un proceso de *compilación* del mismo. En consecuencia la fase de traducción de esta aplicación realizará una compilación del código fuente analizado.

Este proceso de traducción no genera un código máquina nativo del procesador del equipo de cómputo en el cual se compila el sistema, es decir, el proceso de generación de código ejecutable, no genera código de procesador Intel. El código producido es generado para una Máquina Ejecutora de Código Intermedio. En este trabajo se le ha llamado Máquina-P. Esta Máquina-P es lo más aproximado a una Máquina de Turing.

TESIS CON
FALLA DE ORIGEN

Una vez detallado el proceso de creación del compilador, se dedicará una porción de este trabajo para mostrar algunos programas escritos en ULP, capaces de realizar tareas asombrosas. Poder observar, por ejemplo, la manera en la que es posible realizar operaciones aritméticas básicas empleando ULP, o bien la manera en la que es posible resolver un laberinto complicado con un programa hecho a base de muy pocas instrucciones.

También se encontrará dentro de este trabajo una sección dedicada a las pruebas que se le hicieron al sistema enseñando a niños a programar en ULP y los resultados que se obtuvieron de estas, es decir que uno de los objetivos que se buscaron es el de que mediante este proyecto se ofrezca una herramienta rentable y práctica para la enseñanza de la programación a cualquier nivel.

A lo largo de todo el proyecto se buscará que los conceptos presentados sean claros y legibles, también que la didáctica sea una constante a lo largo del trabajo, y así darle un uso más allá de un simple trabajo de tesis y poderlo aplicar como instrumento de trabajo para la impartición de materias de la carrera de Ing. en Computación.

De hecho, y como propuesta, se pretende que esta aplicación pueda llegar a ser una herramienta didáctica valiosa para apoyar la enseñanza de materias como lenguajes formales y autómatas, programación estructurada y compiladores, dentro de la Facultad de Ingeniería de la UNAM.

CAPÍTULO I

Fundamentos

Teóricos

TESIS CON
FALLA DE ORIGEN

CAPÍTULO I. FUNDAMENTOS TEÓRICOS

I.1. Compiladores

A grandes rasgos, un compilador lee un programa escrito en un lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto (Véase Fig. I.1.). Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente. Por tradición los compiladores se han escrito en lenguaje ensamblador de la PC empleada, sin embargo la tendencia es ahora escribir compiladores con lenguajes de alto nivel, como C, Visual Basic, etc, a causa de la reducción del tiempo de programación necesario y del tiempo de depuración, así como por ser más legible el compilador una vez terminado.

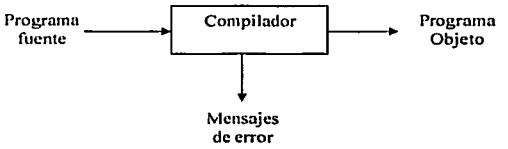


Fig I.1 Compilador Básico

Los lenguajes objeto son igualmente variados; un lenguaje objeto puede ser otro lenguaje de programación o el lenguaje máquina de cualquier PC entre un microprocesador y una supercomputadora. Los compiladores a menudo se clasifican como de una pasada, de múltiples pasadas, de carga y ejecución, de depuración o de optimación, dependiendo como hayan sido construidos o de que función se supone que realizan. A pesar de esta aparente complejidad, las tareas básicas que debe realizar cualquier compilador son esencialmente las mismas. Al comprender tales tareas, se pueden construir compiladores para una gran diversidad de lenguajes fuente y máquinas objeto utilizando las mismas técnicas básicas.

I.1.1. Modelo de análisis y síntesis de compilación

En la compilación hay dos partes: análisis y síntesis. La parte del análisis divide el programa fuente en sus elementos componentes y crea una representación intermedia del programa fuente. La parte de síntesis construye el programa objeto deseado a partir de la representación intermedia. De las dos partes, la síntesis es la que requiere técnicas más especializadas.

Durante el análisis, se determinan las operaciones que implica el programa fuente y se registran en una estructura jerárquica llamada árbol. A menudo, se usa una clase especial de árbol sintáctico, donde cada nodo representa una operación y los hijos de un nodo son los argumentos de la operación.

TESIS CON
FALLA DE ORIGEN

CAPÍTULO I. FUNDAMENTOS TEÓRICOS

Muchas herramientas de software que manipulan programas fuente realizan primero algún tipo de análisis. Algunos ejemplos de tales herramientas son:

- **Editores de estructuras.** Un editor de estructuras toma como entrada una secuencia de órdenes para construir un programa fuente. El editor de estructuras no sólo realiza las funciones de creación y modificación de textos de un editor de textos ordinario, sino que también analiza el texto del programa, imponiendo al programa fuente una estructura jerárquica apropiada. De esa manera el editor de estructuras puede realizar tareas adicionales útiles para la preparación de programas.
- **Impresoras estéticas.** Una impresora estética analiza un programa y lo imprime de forma que la estructura del programa resulte claramente visible. Por ejemplo, los comentarios pueden aparecer con un tipo de letra especial, y las proposiciones pueden aparecer con una indentación proporcional a la profundad de su anidamiento en la organización jerárquica de las proposiciones.
- **Verificadores estáticos.** Un verificador estático lee un programa, lo analiza e intenta descubrir errores potenciales sin ejecutar el programa. La parte del análisis a menudo es similar a la que se encuentra en los compiladores de optimación. Así, un verificador estático puede detectar si hay partes de un programa que nunca se podrán ejecutar o si cierta variable se usa antes de ser definida. Además, puede detectar errores de lógica, como intentar utilizar una variable real como apuntador, empleando las técnicas de verificación.
- **Intérpretes.** En lugar de producir un programa objeto como resultado de un traducción, un intérprete realiza las operaciones que implica el programa fuente. Muchas veces los intérpretes se usan para ejecutar lenguajes de órdenes, pues cada operador que se ejecuta en un lenguaje de órdenes suele ser una invocación de una rutina compleja, como un editor o un compilador. Del mismo modo, algunos lenguajes de "muy alto nivel ", como el APL, normalmente son interpretados, porque hay muchas cosas sobre los datos, como el tamaño y la forma de las matrices, que no se pueden deducir en el momento de la compilación.

Tradicionalmente, se concibe un compilador como un programa que traduce un programa fuente, al lenguaje ensamblador o de máquina o de máquina de alguna PC. Sin embargo, hay lugares, al parecer, no relacionados, donde la tecnología de los compiladores se usa con regularidad. La parte de análisis de cada uno de los siguientes ejemplos es parecida a la de un compilador convencional.

- **Formadores de textos.** Un formador de textos toma como entrada una cadena de caracteres, la mayor pare de la cual es texto para componer, pero alguna incluye órdenes para indicar párrafos, figuras o estructuras matemáticas, como subíndices o superíndices.
- **Compiladores de circuitos de silicio.** Un compilador de circuitos de silicio tiene un lenguaje fuente similar o idéntico a un lenguaje de programación convencional. Sin embargo, las variables del lenguaje no representan localidades de memoria, sino señales lógicas (0 ó 1) o grupos de señales en un circuito de conmutación. La salida es el diseño de un circuito en un lenguaje apropiado.
- **Intérpretes de consultas.** Un intérprete de consultas traduce un predicado que contiene operadores racionales y boléanos a órdenes para buscar en una base de datos registros que satisfagan ese predicado.

I.1.2. Fases de un compilador

Conceptualmente, un compilador opera en fases, cada una de las cuales transforma al programa fuente de una representación en otra. De la Fig. I.1.2 se muestra una descomposición típica de un compilador. A continuación se explican brevemente cada una de las fases.

I.1.2.1. Administrador de la tabla de símbolos

Una función esencial de un compilador es registrar los identificadores 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, su tipo, su ámbito (la parte del programa donde tiene validez), y en el caso de nombres de procedimientos, cosas como el número y tipos de sus argumentos, el método de pasar cada argumento.

Una tabla de símbolos es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos del identificador. La estructura de datos permite encontrar rápidamente el registro de cada identificador y almacenar o consultar rápidamente datos de ese registro.

TESIS CON
FALLA DE ORIGEN

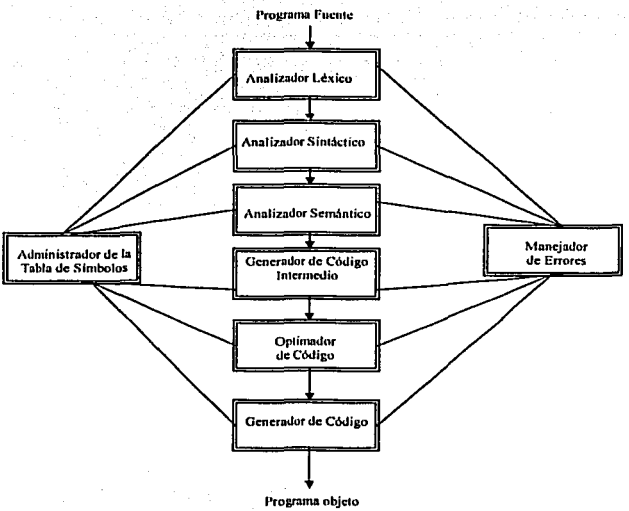


Fig. I.1.2 Fases de un Compilador

I.1.2.2. Programa Fuente

La representación interna del programa fuente dependerá en gran parte de cómo se va a manejar después. Puede ser un árbol representando la sintaxis del programa fuente. O puede ser el programa fuente en la llamada notación polaca.

I.1.2.3 Analizador Léxico ("The scanner")

En un compilador, el análisis lineal se llama análisis léxico o exploración. También llamado analizador lexicográfico es la parte más simple de un compilador y va explorando los caracteres del programa fuente de izquierda a derecha y construye los símbolos del programa (enteros,

identificadores, palabras reservadas, etc). El explorador puede también colocar los identificadores en la tabla de símbolos y realizar otras tareas simples que pueden efectuarse sin analizar a fondo el programa fuente.

Por ejemplo, en la siguiente sentencia los componentes léxicos quedarían de la siguiente manera:

Posición := inicial + velocidad * 30

Se agruparían en los componentes léxicos siguientes:

1. El identificador: posición
2. El símbolo de asignación: :=
3. El identificador: inicial
4. El signo de: suma
5. El identificador: velocidad
6. El signo de: multiplicación
7. El número: 30

Los espacios en blanco que separan los caracteres de estos componentes léxicos normalmente se eliminan durante el análisis léxico

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida unas secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción se demuestra en la Fig. I.1.2.3. suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico, al analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

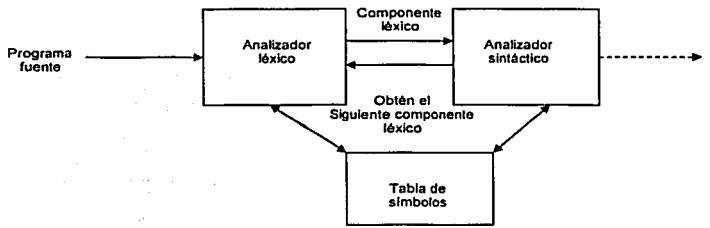


Fig. I.1.2.3 Interacción de un analizador léxico con el analizador sintáctico.

Como el analizador léxico es la parte del compilador que lee el texto fuente, también puede realizar ciertas funciones secundarias en la interfaz del usuario, como eliminar del programa fuente

TESIS CON FALLA DE ORIGEN

comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva.

Otra función es relacionar los mensajes de error del compilador con el programa fuente. Por ejemplo el analizador léxico puede tener localizado el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea con un mensaje de error. En algunos compiladores, el analizador léxico se encarga de hacer una copia del programa fuente en el que están marcados los mensajes de error. Si el lenguaje fuente es la base de algunas funciones de pre-procesamiento de macros, entonces esas funciones del pre-procesador también se pueden aplicar al hacer el análisis léxico.

En algunas ocasiones, los analizadores léxicos se dividen en una cascada de dos fases; la primera, llamada "examen", y la segunda, "análisis léxico", el examinador se encarga de realizar tareas sencillas, mientras que el analizador léxico es el que realiza las operaciones más complejas.

I.1.2.3.1. Aspectos del análisis léxico

Hay varias razones para dividir la fase de análisis de la compilación en análisis léxico y análisis sintáctico.

- Un diseño sencillo es quizá la consideración más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una u otra de dichas fases. Por ejemplo, un analizador sintáctico que incluya las convenciones de los comentarios y espacios en blanco es bastante más complejo que uno que pueda comprobar si los comentarios y espacios en blanco ya han sido eliminados por el analizador léxico. Si está diseñando un lenguaje nuevo, la separación de las convenciones léxicas de las sintácticas puede dar origen a un diseño del lenguaje más claro.

- Se mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente mas eficiente para esta función. Gran parte de tiempo se consume en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de buffers para la lectura de caracteres de entrada y procesamiento de componentes léxicos se puede mejorar significativamente el rendimiento de un compilador.
- Se mejora la transportabilidad del compilador. Las peculiaridades del alfabeto de entrada y otras anomalías propias de dispositivos pueden limitarse al analizador léxico. La representación de símbolos especiales o no estándar, pueden ser aisladas en el analizador léxico.

Se han diseñado herramientas especializadas que ayudan a automatizar la construcción de analizadores léxicos y analizadores sintácticos cuando están separados.

I.1.2.3.2. Componentes léxicos, patrones y lexemas

Cuando se menciona el análisis sintáctico, los términos "componente léxico" (token), "patrón" y "lexema" se emplean con significados específicos. En general hay un conjunto de cadenas en la entrada para el cual se produce como salida el mismo componente léxico. Este conjunto de cadenas se describe mediante una regla llamada patrón asociado al componente léxico. Se dice que el patrón concuerda con cada cadena del conjunto. Un lexema es una secuencia de caracteres en el programa fuente con la que concuerda el patrón par un componente léxico.

Los componentes léxicos se tratan como símbolos terminales de la gramática del lenguaje fuente, con nombres en negritas para representarlos. Los lexemas para el componente léxico que concuerdan con el patrón representan cadenas de caracteres en el programa fuente que se pueden tratar juntos como una unidad léxica.

En la mayoría de los lenguajes de programación, se consideran componentes léxicos las siguientes construcciones: palabras clave, operadores identificadores, constantes, cadenas literales y signos de puntuación, como paréntesis, coma y punto y coma. Un patrón es una regla que describe el conjunto de lexemas que pueden representar a un determinado componente léxico en los programa fuente.

I.1.2.3.3. Errores léxicos

Son pocos los errores que se pueden detectar simplemente en el nivel léxico porque un analizador léxico tienen una visión muy restringida de un programa fuente. Supóngase que surge una situación en la que el analizador léxico no puede continuar porque ninguno de los patrones concuerda con un prefijo de la entrada restante. Tal vez la estrategia de recuperación más sencilla sea la recuperación en "modo pánico", en donde se borran caracteres sucesivos de la entrada restante hasta que el analizador léxico puede encontrar un componente léxico bien formado. Esta técnica de recuperación puede confundir en ocasiones al analizador sintáctico, pero en un ambiente de computación interactivo puede resultar bastante adecuada. Otras posibles acciones de recuperación de errores son:

- Borrar un carácter extraño
- Insertar un carácter que falta

TESIS CON
FALLA DE ORIGEN

- Reemplazar un carácter incorrecto por otro correcto
- Intercambiar dos caracteres adyacentes

Se puede probar este tipo de transformaciones de error para intentar reparar la entrada. La mas sencilla de tales estrategias consiste en observar si un prefijo de la entrada restante se puede transformar en un lexema válido mediante una sola transformación de error. Esta estrategia da por supuesto que la mayoría de los errores léxico se deben a una sola transformación de error, suposición que normalmente, pero no siempre, se cumple en la práctica.

Una forma de encontrar los errores en un programa consiste en calcular el número mínimos de transformaciones necesarias para transformar el programa erróneo en otro que este sintácticamente bien construido. Se dice que el programa erróneo tiene k errores cuando la secuencia más corta de transformaciones de error que lo transformará en algún programa válido tiene la longitud k. La corrección de errores de distancia mínima es un criterio teórico apropiado, pero no se suele usar en la práctica porque su aplicación es demasiado costosa. Sin embargo, algunos compiladores experimentales han empleado el criterio de la distancia mínima para hacer correcciones locales.

I.1.2.4 Analizador Sintáctico("Parser")

El análisis jerárquico se denomina análisis sintáctico. Este implica agrupar los componentes léxicos del programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico.

Todo lenguaje de programación tiene reglas que prescriben la estructura sintáctica de programas bien formados. En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente. Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los escritores de compiladores.

- Una gramática da una especificación sintáctica precisa y fácil de entender de un lenguaje de programación
- A partir de algunas clases de gramáticas se puede construir automáticamente un analizador sintáctico eficiente que determine si un programa fuente esta sintácticamente bien formado. Otra ventaja es que el proceso de construcción del analizador sintáctico puede revelador ambigüedades sintácticas y otras construcciones difíciles de analizar que

TESIS CON
FALLA DE ORIGEN

de otro modo podrían pasar sin detectar en la fase inicial de diseño de un lenguaje y de su compilador.

- Una gramática diseñada adecuadamente imparte una estructura a un lenguaje de programación útil para la traducción de programas fuente a código objeto correcto y para la detección de errores. Existen herramientas para convertir descripciones de traducciones basadas en gramáticas en programas operativos.

- Los lenguajes evolucionan con el tiempo, adquiriendo nuevas construcciones y realizando tareas adicionales. Estas nuevas construcciones se pueden añadir con más facilidad a un lenguaje cuando existe una aplicación basada en una descripción gramatical del lenguaje.

I.1.2.4.1. El papel del analizador sintáctico

En el modelo de compilador que se muestra en la figura I.1.2.4.1 , el analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, y comprueba si la cadena puede ser generada por la gramática del lenguaje fuente. Se supone que el analizador sintáctico informará de cualquier error de sintaxis de manera inteligible. También debería recuperarse de errores que ocurren frecuentemente para poder continuar procesando el resto de su entrada.

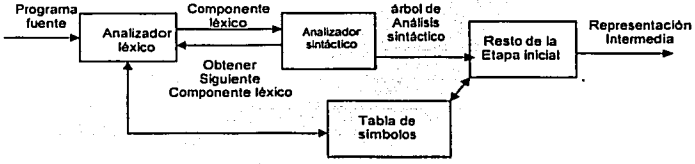


Fig. I.1.2.4.1 Posición del analizador sintáctico en el modelo del compilador

Existen tres tipos generales de analizadores sintácticos para gramáticas. Los métodos universales de análisis sintáctico, como el algoritmo de Cocke-Younger-Kasami y el de Earley, pueden analizar cualquier gramática. Estos métodos, sin embargo, son demasiado ineficientes para usarlos en la producción de compiladores. Los métodos empleados generalmente en los compiladores se clasifican como descendentes o ascendentes. Como sus nombres indican los analizadores

TESIS CON
FALLA DE ORIGEN

sintácticos descendentes construyen árboles de análisis sintáctico desde arriba (la raíz) hasta abajo (las hojas), mientras que los analizadores sintácticos ascendentes comienzan en las hojas y suben hacia la raíz. En ambos casos se examina la entrada al analizador sintáctico de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes trabajan solo con subclases de gramáticas, pero varias de estas subclases como las gramáticas LL y LR, son lo suficientemente expresivas para describir la mayoría de las construcciones sintácticas de los lenguajes de programación. En la práctica hay varias tareas que se pueden realizar durante el análisis sintáctico, como recoger información sobre distintos componentes léxicos en la tabla de símbolos, realizar la verificación de tipo y otras clases de análisis semántico, y generar código intermedio.

I.1.2.4.2. Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es sorprendente que aunque los errores sean tan frecuentes, pocos lenguajes han sido diseñados teniendo en cuenta el manejo de errores. Esta civilización sería completamente distinta si los lenguajes hablados exigieran tanta exactitud sintáctica como los lenguajes de programación. La mayoría de las especificaciones de los lenguajes de programación no describen como debe responder un compilador a los errores; la respuesta se deja al diseñador del compilador.

Considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar sus respuesta a los errores. Se sabe que los programas pueden contener errores de muy diverso tipo. Por ejemplo, los errores pueden ser:

- Léxicos, como escribir mal un identificador, palabra clave u operador
- Sintácticos, como una expresión aritmética con paréntesis no equilibrados
- Semánticos, como un operador aplicado a un operando incompatible
- Lógicos, como una llamada infinitivamente recursiva.

A menudo, gran parte de la detección y recuperación de errores en un compilador se centra en la fase de análisis sintáctico. Una razón es que muchos errores son de naturaleza sintáctica o se manifiestan cuando la cadena de componentes léxicos que proviene del analizador léxico desobedece las reglas gramaticales que definen al lenguaje de programación. Otra razón es la precisión de los métodos modernos de análisis sintácticos, que pueden detectar la presencia de

errores dentro de los programas de una forma muy eficiente. La detección exacta de la presencia de errores semánticos y lógicos en el momento de la compilación es mucho más difícil.

El manejador de errores en un analizador sintáctico tiene objetivos fáciles de establecer:

- Debe informar de la presencia de errores con claridad y exactitud.
- Se debe recuperar de cada error con la suficiente rapidez como para detectar errores posteriores.
- No debe retrasar de manera significativa el procesamiento de programas correctos.

La realización efectiva de estos objetivos plantea desafíos importantes. Afortunadamente, los errores más comunes son simples y a menudo basta con un mecanismo sencillo de manejo de errores. Sin embargo, en algunos casos un error pudo haber ocurrido mucho antes de la posición en que se detectó su presencia, y puede ser muy difícil deducir la naturaleza precisa del error. En los casos difíciles, el manejador de errores quizá tenga que adivinar que tenía en mente el programador cuando escribió el programa.

Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan un error lo antes posible. Es decir, tienen la propiedad del prefijo viable, lo cual quiere decir que detectan la presencia de un error nada más ver un prefijo de la entrada que no es prefijo, de ninguna cadena de lenguaje.

I.1.2.5. Analizador Semántico

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente.

I.1.2.6. Comprobación de tipos

Un compilador debe comprobar si el programa fuente sigue tanto las convenciones sintácticas como las semánticas del lenguaje fuente. Esta comprobación llamada comprobación estática (para distinguirla de la comprobación dinámica que se realiza durante la ejecución del programa objeto), garantiza la detección y comunicación de algunas clases de errores de programación. Los ejemplos de comprobación estática incluyen:

TESIS CON
FALLA DE ORIGEN

- Comprobaciones de tipos. Un compilador debe informar de un error si se aplica un operador a un operando incompatible; por ejemplo, si se suman una variable tipo matriz y una variable de función.

- Comprobaciones del flujo de control. Las proposiciones que hacen que el flujo del control abandone una construcción deben tener algún lugar a donde transferir ese flujo, si dicha posición no existe, ocurre un error.

- Comprobaciones de unicidad. Hay situaciones en que se debe definir un objeto una vez exactamente.

- Comprobaciones relacionadas con nombres. En ocasiones, el mismo nombre debe aparecer dos o más veces.

Un comprobador de tipos se asegura de que el tipo de una construcción coincida con el previsto en su contexto.

I.1.2.7. Generador de código intermedio

Antes que se pueda generar el código, es necesario generalmente el manipular y cambiar el programa interno de algún modo. Se tiene que asignar memoria a las variables para el tiempo de ejecución. Un tema importante aquí es la optimización del programa para reducir el tiempo de ejecución del programa objeto.

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.

En el método de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto. Los detalles del lenguaje objeto se confinan en la etapa final, si esto es posible. Aunque un programa fuente se puede traducir directamente al lenguaje objeto, algunas ventajas de utilizar una forma intermedia independiente de la máquina son:

- Se facilita la redestinación; se puede crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.

- Se puede aplicar a la representación intermedia un optimador de código independiente de la máquina.

Simplificando, se supone que el programa fuente ya ha sido analizado sintácticamente y comprobado estáticamente como se representa en la Figura I.1.2.7.

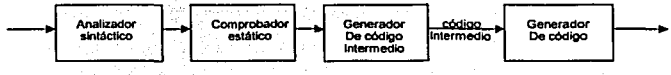


Fig. I.1.2.7 Ubicación del generador de código intermedio

I.1.2.8. Optimador de código

La fase de optimación de código trata de mejorar el código intermedio de modo que resulte un código de máquina más rápido de ejecutar. Hay mucha variedad en la cantidad de optimación de código que ejecutan los distintos compiladores. En los que hacen mucha optimación, llamados "compiladores optimadores", una parte significativa del tiempo del compilador se ocupa en esta fase. Sin embargo, hay optimaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto sin retardar demasiado la compilación.

Idealmente, los compiladores deberían producir código objeto que fuera tan bueno como para ser escrito a mano. La realidad es que este objetivo solo se alcanza en pocos casos y difícilmente. Sin embargo, a menudo se puede lograr que el código directamente producido por los algoritmos de compilación se ejecute más rápidamente o que ocupe menos espacio, o ambas cosas. Esta mejora se consigue mediante transformaciones de programas que tradicionalmente se denominan optimaciones, aunque el término "optimación" no es adecuado porque rara vez existe la garantía de que el código resultante sea el mejor posible. Los compiladores que aplican transformaciones para mejorar el código se denominan compiladores optimadores.

Para crear un programa en lenguaje objeto eficiente, un programador necesita más que un compilador optimador.

TESIS CON
FALLA DE ORIGEN

I.1.2.8.1. Criterios de transformación para mejorar el código

Dicho de una manera sencilla, las mejores transformaciones de programas son las que producen el mayor beneficio con el menor esfuerzo. Las transformaciones realizadas por un compilador optimador debe tener varias propiedades que son las siguientes:

- Primero, una transformación debe preservar el significado de los programas. Es decir, una "optimación" no debe cambiar el resultado producido por un programa para una entrada dada, o causar un error, como una división por cero, que no estuviera presente en la versión original del programa fuente. En todo momento se toma el enfoque "seguro" de desaprovechar la oportunidad de aplicar una transformación en lugar de arriesgarse a cambiar lo que hace el programa.

- Segundo, una transformación debe, como promedio, acelerar los programas en una cantidad mensurable. En ocasiones interesa reducir el espacio que ocupa el código compilado, aunque el tamaño del código tiene menos importancia que la que tenía antes. Por supuesto, no toda transformación consigue mejorar todo programa y, ocasionalmente, una "optimación" puede aligerar un programa en general mientras mejore las cosas.

- Tercero, una transformación debe valer la pena. No tiene sentido que el escritor de un compilador haga el esfuerzo intelectual de aplicar una transformación que mejore el código y que el compilador gaste el tiempo adicional compilando programas fuente si este esfuerzo no es recompensado cuando se ejecutan los programas objeto.

Algunas transformaciones solo se pueden aplicar después de un análisis detallado y que lleva su tiempo del programa fuente, de modo que tiene poco sentido aplicarlas a programas que solo se ejecutaran pocas veces.

I.1.2.8.2. Obtención de un mayor rendimiento

Generalmente se obtienen mejoras espectaculares en el tiempo de ejecución de un programa, como reducir el tiempo de ejecución de unas horas a unos segundos y mejorando el programa a todos niveles, desde el nivel fuente hasta el nivel objeto, como se sugiere en la figura I.1.2.8.2 . En cada nivel las opciones disponibles están entre los dos extremos de encontrar un algoritmo mejor y de implantar un algoritmo dado, así que se realizan menos operaciones.

TESIS CON
FALLA DE ORIGEN

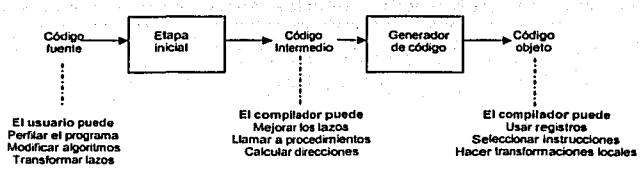


Fig I.1.2.8.2 Lugares en que el usuario y el compilador pueden hacer mejoras potenciales

Lamentablemente, ningún compilador puede encontrar el mejor algoritmo para un problema dado. Sin embargo, a veces un compilador puede sustituir una secuencia de operaciones por una secuencia algebraicamente equivalente, y con ello reducir significativamente el tiempo de ejecución de un programa. Dichos ahorros son mas habituales cuando se aplican transformaciones algebraicas a los programas en lenguajes de muy alto nivel, por ejemplo, lenguajes de consulta para bases de datos.

Aunque es posible que el programador mejore el código, puede ser más conveniente que el compilador realice algunas mejoras. Si se puede confiar en que el compilador genere código eficiente, entonces el usuario puede concentrarse en escribir código claro.

I.1.2.8.3. Una organización para un compilador optimador

Como ya se ha mencionado, existen varios niveles en los que se puede mejorar un programa. Como las técnicas necesarias para analizar y transformar un programa no cambian significativamente con el nivel, esto se puede observar en la Figura I.1.2.8.3. La fase de mejora del código consta del análisis de flujo de control y el análisis de flujo de datos seguidos de la aplicación de transformaciones.

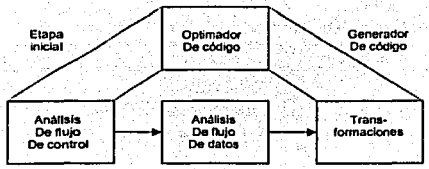


Fig. I.1.2.8.3 Organización del optimador de código

I.1.2.9. Generador de código

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código máquina relocalizable o código ensamblador, como se muestra en la figura I.1.2.9. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a registros.

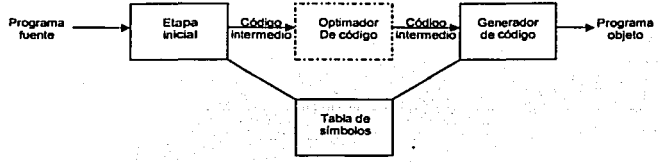


Fig. I.1.2.9 Posición del generador de código

Las exigencias tradicionalmente impuestas a un compilador son duras. El código de salida debe ser correcto y de gran calidad, lo que significa que debe utilizar de forma eficaz los recursos de la máquina objeto. Además , el generador de código mismo debe ejecutarse eficientemente.

Matemáticamente, el problema de generar código óptimo es indecible. En la práctica, hay que conformarse con técnicas heurísticas que generan código bueno pero no siempre óptimo. La elección de las heurísticas es importante, ya que un algoritmo de generación de código cuidadosamente diseñado puede producir fácilmente código que sea varias veces más rápido que el producido por un algoritmo diseñado precipitadamente.

TESIS CON
FALLA DE ORIGEN

En tanto que los detalles dependen de la máquina objeto y del sistema operativo, aspectos como el manejo de la memoria, la selección de instrucciones, la asignación de registros y el orden de evaluación son inherentes en casi todos los problemas de generación de código.

I.1.2.9.1 Entrada al generador de código

La entrada para el generador de código consta de la representación intermedia del programa fuente producida por la etapa inicial, junto con información de la tabla de símbolos que se utiliza para determinar las direcciones durante la ejecución de los objetos de datos denotados por los nombres de la representación intermedia.

Se asume que antes de la generación de código, la etapa inicial ha hecho los análisis léxico y sintáctico, y traducido el programa fuente a una representación intermedia razonablemente detallada, así que los valores de los nombres que aparecen en el lenguaje intermedio pueden ser representados por cantidades que la máquina objeto puede manipular directamente (bits, enteros, reales, apuntadores, etc). También se supone que ya ha tenido lugar la comprobación de tipos necesaria, de modo que los operadores de conversión de tipos ya se han insertado donde fuera necesario y ya se han detectado los errores semánticos obvios. Por tanto, la fase de generación de código puede proseguir con la hipótesis de que su entrada no contiene errores. En algunos compiladores, esta clase de comprobación semántica se realiza junto con la generación de código.

I.1.2.10. Programas objeto

La salida del generador de código es el programa objeto. Al igual que el código intermedio, esta salida puede adoptar una variedad de formas: lenguaje de máquina absoluto, lenguaje de máquina relocalizable o lenguaje ensamblador.

Producir como salida un programa en lenguaje de máquina absoluto tiene la ventaja de que se puede colocar en una posición fija de memoria y ejecutarse inmediatamente. Un programa pequeño se puede compilar y ejecutar rápidamente. Producir como salida un programa de máquina relocalizable (módulo-objeto) permite que los subprogramas se compilen por separado. Un conjunto de módulos objeto relocalizables se puede enlazar y cargar para su ejecución mediante un cargado enlazador.

Aunque se tenga que pagar el costo añadido de enlazar y cargar si se producen módulos objeto relocalizables, se gana mucha flexibilidad al poder compilar subrutinas por separado y llamar desde un módulo objeto a otros programas previamente compilados. Si la máquina objeto no maneja relocalización automáticamente, el compilador debe proporcionar al cargador información de relocalización explícita para que enlace los segmentos de programa compilados por separado.

TESIS CON
FALLA DE ORIGEN

Producir como salida un programa en lenguaje ensamblador facilita el proceso de generación de código. Se pueden generar instrucciones simbólicas y utilizar las macros del ensamblador para ayudar a generar el código. El precio que se paga es el paso de ensamble después de la generación de código. Producir código ensamblador no duplica la tarea completa del compilador, esta elección es otra alternativa razonable, especialmente para una máquina de memoria pequeña, donde un compilador debe utilizar varias pasadas.

Sin embargo, se debe insistir en que mientras las direcciones se puedan calcular según los desplazamientos y otra información almacenada en la tabla de símbolos, el generador de código puede producir direcciones relocalizables o absolutas para nombres al igual que direcciones simbólicas.

I.1.2.11. Manejador de errores

Cada fase puede encontrar errores. Sin embargo, después de detectar un error, cada fase debe tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de más errores en el programa fuente. Un compilador que se detiene cuando encuentra el primer error, no resulta tan útil como debiera.

Las fases de análisis sintáctico y semántico por lo general manejan una gran porción de los errores detectables por el compilador. La fase léxica puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje.

Los errores donde la cadena de componentes léxico violan las reglas de estructura (sintaxis) del lenguaje son determinados por la fase de análisis sintáctico. Durante el análisis semántico el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta, pero que no tengan significado para la operación implicada.

Así podemos decir que en realidad, un compilador sólo es un programa. El entorno en que se desarrolle este programa puede afectar la velocidad y la fiabilidad de la implantación del compilador. El lenguaje en el que se implante el compilador es igualmente importante, ya que la mayoría de las personas que escriben compiladores elegirán un lenguaje orientado a sistemas como C, Visual Basic, Visual Java, etc, esto debido a la facilidad que nos presentan estos lenguajes comparados con UNIX por ejemplo.

CAPÍTULO I. FUNDAMENTOS TEÓRICOS

I.2 Lenguajes Formales y Autómatas

Autómatas finitos y ciertas clases de **gramáticas formales** son usadas en el diseño y construcción de software.Los lenguajes nos permiten comunicarnos con la máquina, parte de lo que puede hacer la máquina depende del poder descriptivo del lenguaje.

- Compiladores.
- Traductores.
- Diseño de lenguajes de alto nivel.

I.2.1. Referencias Históricas

En 1937 **Alan Turing** desarrolló una máquina abstracta denominada **Máquina de Turing** para el estudio de la computabilidad, llegando a asegurar que: "si algún problema se puede resolver algoritmicamente, entonces existe una máquina de Turing capaz de resolverlo".

Entre 1940 y 1950, se desarrollan unas máquinas simples, en cuanto a su funcionamiento, que fueron conocidas como autómatas finitos, para modelar el funcionamiento del cerebro. También en los 50's, **N. Chomsky** comienza el estudio formal de las gramáticas (generadoras de lenguajes).

En 1969, S. Cook extiende el estudio de Turing. Cook separa aquellos problemas que pueden ser solucionados de aquellos que en principio pueden ser solucionados pero que en la práctica toman demasiados recursos.

I.2.2. Conceptos básicos

Alfabeto: Conjunto finito, a sus elementos se les llaman símbolos o letras.

Palabra: Sobre un alfabeto A es una sucesión finita de elementos de A, es decir u es una palabra sobre A si y solo si $u = a_1...a_n$ donde $a_i \in A$, $\forall\ i=1..n$.

Longitud : De una palabra u, definida sobre el alfabeto A ($u \in A^*$) es el número de símbolos de A que contiene. La longitud se nota como $|u|$.

Palabra vacía: Es la palabra de longitud cero. Es la misma para todos los alfabetos y se nota como (ε o λ).

TESIS CON
FALLA DE ORIGEN

Conjunto: De todas las palabras formadas sobre un alfabeto se nota como A^*. El conjunto de todas las palabras formadas sobre un alfabeto **excluyendo** la cadena vacía se nota como A^+.

Lenguaje : Sobre el alfabeto A es un subconjunto de las cadenas sobre A ($L \subseteq A^*$)

I.2.3. Operaciones

La concatenación de dos palabras u y v, $u = a_1a_2...a_n$ y $v = b_1b_2...b_m$, ambas definidas sobre A^* ($u,v \in A^*$), es la cadena $a_1a_2...a_nb_1b_2...b_m$. Se nota como u.v o simplemente como uv.

Propiedades de la operación concatenación:

– $|u.v| = |u|+|v|$, $\forall u,v \in A^*$.

– **Asociativa:** $u.(v.w) = (u.v).w$, $\forall u,v,w \in A^*$.

– **Elemento neutro.** $u.\varepsilon = \varepsilon.u=u$

Si $u \in A^*$ entonces:

$-u^0 = \varepsilon$.

$-u^{i+1} = u^i.u$, $\forall i \leq 0$.

Si $u = a_1a_2...a_n \in A^*$, entonces la cadena inversa de u es la cadena $u^{-1} = a_na_{n-1}...a_1$..

La concatenación de dos lenguajes L_1 y L_2 definidos sobre el alfabeto A, se obtiene conforme a la siguiente expresión:

$L_1L_2 = \{u_1u_2 \mid u_1 \in L_1 \text{ y } u_2 \in L_2\}$..

Propiedades de la concatenación:

$L\varnothing = \varnothing L = \varnothing$ ($\varnothing$ es el lenguaje que contiene 0 palabras).

– **Elemento neutro.** $\{\varepsilon\}L = L\{\varepsilon\} = L$.

Asociativa. $L_1(L_2L_3) = (L_1L_2)L_3$.

La conmutativa no aplica

Si L es un lenguaje sobre el alfabeto A, entonces la iteración de este lenguaje se define de acuerdo con las siguientes expresiones:

- $L^0 = \{\varepsilon\}$
- $L^{i+1} = L^i L$ Sean dos lenguajes L_1 y L_2,

La unión de L_1 y L_2: $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ o } x \in L_2\}$La intersección L_1 y L_2:$L_1 \cap L_2 = \{x \mid x \in L_1 \text{ y } x \in L_2\}$
La diferencia de L_1 y L_2:$L_1 - L_2 = \{x \mid x \in L_1 \text{ y } x \in L_2\}$
$L_1 \cup \varnothing = L_1$; $L_1 \cap \varnothing = \varnothing$.

I.2.4. Lenguajes naturales y Lenguajes formales

Lenguajes Naturales.
Las reglas gramaticales se desarrollan para reglamentar de alguna forma la propia evolución del lenguaje.
Ejemplos: Inglés, español, alemán, ruso, chino...

Lenguajes Formales.
Las reglas gramaticales definen y determinan claramente las estructuras del lenguaje.
Ejemplos: Lenguajes de programación, lenguajes matemáticos...

I.2.5. Expresiones regulares

Las expresiones regulares son otro tipo de notación para la definición de lenguajes. Podemos pensar en las expresiones regulares como lenguajes de programación mediante los cuales expresamos algunas aplicaciones importantes, como búsqueda de textos o componentes de un compilador.

Las expresiones regulares también pueden ser vistas como una alternativa a la notación de los Autómatas Finitos No Deterministicos para describir componentes Las expresiones regulares son descripciones algebraicas de lenguajes. Pueden definir exactamente los mismos lenguajes que los autómatas finitos, y estos son los lenguajes regulares. Ofrecen un método declarativo para expresar las cadenas válidas de un lenguaje.

I.2.5.1. Precedencia de operadores

Los operadores son asociados con sus operandos en un determinado orden, que se establece por la precedencia de los operadores.

TESIS CON
FALLA DE ORIGEN

El orden de precedencia para los operadores es el siguiente:

El operador * ó cerradura, se aplica a la expresión regular más pequeña bien formada que quede a su derecha.
El operador . ó concatenación.
El operador + ó unión.

I.2.6. Gramáticas

Es el mecanismo empleado para establecer la estructura de un lenguaje, es decir, las sentencias que lo forman. Consiste de un conjunto de reglas sintácticas que establecen la forma en la que se pueden combinar los símbolos del alfabeto:

- **ORACION** es un SUJETO y un PREDICADO.
- **SUJETO** es una FRASE NOMINAL.
- **FRASE NOMINAL** es un GRUPO NOMINAL y un CALIFICATIVO que puede o no estar.
- **GRUPO NOMINAL** es un ARTICULO que puede no estar y un NOMBRE.
- **CALIFICATIVO** es un ADJETIVO o una CONJUNCIÓN y una ORACION.

Gramática Informal: es un conjunto finito de reglas para describir y/o generar las sentencias que forman un lenguaje.

I.2.6.1. Ambigüedad

La ambigüedad puede aparecer a varios niveles (sentencias, lenguajes y gramáticas).
Una **sentencia** es ambigua si tiene más de una derivación o árbol de derivación.
Una **gramática** es ambigua si tiene al menos una sentencia ambigua.
Un **lenguaje** es ambiguo si es generado por una gramática ambigua.
Un lenguaje es inherentemente ambiguo si todas las gramáticas que lo generan son ambiguas.

I.2.7. Autómatas

Son sistemas que en todo momento se encuentran en uno de un conjunto finito de estados. El propósito de un estado es recordar la historia del sistema.
Puesto que el número de estados es finito, el sistema debe ser diseñado para recordar aquello que es importante y olvidar lo que no. La ventaja de tener un número finito de estados es que el sistema podrá ser implementado con un fijo conjunto de recursos. Los autómatas vienen a ser

mecanismos formales que "realizan" derivaciones en gramáticas formales. La manera en que las realizan es mediante la noción de reconocimiento. Una palabra será generada en una gramática si y sólo si la palabra hace transitar al autómata correspondiente a sus condiciones terminales. Por esto es que los autómatas son analizadores léxicos (llamados en inglés "parsers") de las gramáticas a que corresponden.

I.2.7.1. Autómatas regulares

Estos son los autómatas finitos más sencillos. Se construyen a partir de un conjunto de estados Q y de un conjunto de símbolos de entrada T. Su funcionamiento queda determinado por una función de transición. Si

$$t : Q \times T \rightarrow Q$$

t(q,s)=p esto se interpreta como que el autómata transita del estado q al estado p cuando arriba el símbolo s.

$$q_0 \in Q$$

En todo autómata finito se cuenta con un estado inicial, y un conjunto de estados finales. Con todo esto definido, la estructura es un autómata regular.

$$\mathbf{AutoReg} = (Q, T, t, q_0, F)$$

$$T^* \rightarrow Q$$

De manera natural, t se extiende a una función de transición: Toda palabra se aplica al autómata y éste, partiendo del estado inicial, transita con cada símbolo de la palabra dada según lo especifique t, correspondiendo a ese símbolo y al estado actual en el autómata. Una palabra es reconocida por el autómata si lo hace arribar a un estado final. El lenguaje del autómata consta de todas las palabras reconocidas.

Ejemplo: Sea $\mathbf{AutoReg} = (Q, T, t, q_0, F)$ el autómata cuyo conjunto de estados es $Q = \{a, b, c\}$, el de símbolos de entrada es $T = \{0, 1\}$, su estado inicial es q_0 = a y el conjunto de estados finales es $F = \{a\}$. Su transición queda determinada por la tabla

t	0	1
a	b	a
b	c	a
c	c	c

Observamos que, partiendo del estado a, mientras lleguen 1's se está en el estado inicial, con un 0 se pasa a b, con un segundo 0 se pasa a c y de ahí no se sale más. En b, al llegar un 1 se regresa

TESIS CON
FALLA DE ORIGEN

al estado inicial. Así pues, para arribar al estado a desde a mismo la cadena de entrada ha de ser una sarta de varias de 1's separadas éstas por únicos 0's. En otras palabras, el autómata reconoce al lenguaje $(1^*0)^*1^*$.

I.2.7.2. Autómatas de pila

Estos autómatas finitos cuentan con un dispositivo de memoria muy elemental, del tipo pila, el cual es un almacenamiento lineal que funciona bajo el principio PEUS: Primero en Entrar, Ultimo en Salir. Sea Q un conjunto de estados, sea T el alfabeto de entrada y sea V un alfabeto de pila.

La función de transición es de la forma $t : Q \times T \times V \to Q \times V^*$, donde la relación $t(q, a, v) = (p, \mathbf{v})$ se interpreta como sigue: "Si se está en el estado q, arriba el símbolo a y en el tope de la pila está el símbolo b entonces se pasa al estado p y se empila la palabra $\mathbf{v}$". Un autómata de pila reconoce a una palabra si, tras haberla leído, termina con su pila vacía. Ejemplo: Las cadenas equilibradas de paréntesis son reconocidas por un autómata de pila determinista. Recordamos que

1. () es una cadena equilibrada de paréntesis, (CEP).
2. Si σ es una CEP entonces (σ) es una CEP.
3. La concatenación de dos CEP's es una CEP.

Para describir a un autómata que reconozca CEP's, representemos al paréntesis que abre "(" con el símbolo a, al paréntesis que cierra ")" con c, y con b al "blanco", es decir, al fin de la cadena de entrada. Consideremos el autómata de pila cuyas componentes son las siguientes:

$$
\begin{array}{lll}
Q = \{\text{Seguir}, \text{Exito}, \text{Fracaso}\} & : & \textbf{estados,} \\
T = \{a, b, c\} & : & \textbf{símbolos de entrada,} \\
V = \{A, C\} & : & \textbf{símbolos de pila,} \\
q_0 = \text{Seguir} & : & \textbf{símbolo inicial,}
\end{array}
$$

y cuya función de transición actúa como sigue,

$$
\begin{array}{lllll}
 & (\text{Seguir}, a, y) & \mapsto & (\text{Seguir}, Ay) & \textbf{empila paréntesis que abren,} \\
 & (\text{Seguir}, c, A) & \mapsto & (\text{Seguir}, nil) & \textbf{suprime paréntesis empatados,} \\
t: & (\text{Seguir}, c, nil) & \mapsto & (\text{Fracaso}, C) & \textbf{no hay equilibrio,} \\
 & (\text{Seguir}, b, A) & \mapsto & (\text{Fracaso}, A) & \textbf{no hay equilibrio,} \\
 & (\text{Seguir}, b, nil) & \mapsto & (\text{Exito}, nil) & \textbf{equilibrio verificado.}
\end{array}
$$

Es claro que este autómata de pila reconoce al lenguaje CEP.

I.2.7.3. Autómatas lineales

Los autómatas lineales son autómatas de pila deterministas que a lo largo de su computación sólo hacen un "cambio de turno". A grandes rasgos, esto significa que toda computación consiste de un procedimiento de empilar consecutivamente para después pasar a desempilar.

Ejemplo: Consideremos el lenguaje de palíndromas con una marca central:

$$L = \{y M x | x \in (0+1)^*, y = reverso(x)\}.$$

Consideremos el autómata de pila cuyas componentes son las siguientes:

$Q = \{\text{Meter}, \text{Sacar}, \text{Exito}, \text{Fracaso}\}$: estados,
$T = \{0, 1, M\}$: símbolos de entrada,
$V = \{C, U\}$: símbolos de pila,
$q_0 = \text{Meter}$: símbolo inicial,

y cuya función de transición actúa como sigue,

$$t: \begin{array}{lcll} (\text{Meter}, 0, y) & \mapsto & (\text{Meter}, Cy) & \text{empila 0's,} \\ (\text{Meter}, 1, y) & \mapsto & (\text{Meter}, Uy) & \text{empila 1's,} \\ (\text{Meter}, M, y) & \mapsto & (\text{Sacar}, y) & \text{con } M \text{ pasa a desempilar,} \\ (\text{Sacar}, 0, C) & \mapsto & (\text{Sacar}, nil) & \text{desempila si hay empatamiento de 0's,} \\ (\text{Sacar}, 1, U) & \mapsto & (\text{Sacar}, nil) & \text{desempila si hay empatamiento de 1's,} \\ (\text{Sacar}, b, nil) & \mapsto & (\text{Exito}, nil) & \text{estado de éxito,} \end{array}$$

donde $y \in V$ y b es el símbolo "blanco". En cualquier otra instancia de t, ésta transitará al estado de fracaso. Es claro que este autómata de pila reconoce al lenguaje L.

I.2.7.4. Autómatas de pila no-deterministas

Los autómatas de pila no-deterministas coinciden con sus homólogos de pila salvo en que su transición no es propiamente una función. Aquí se tiene que la transición es un subconjunto $t \subset (Q \times T \times V) \times (Q \times V^*)$

TESIS CON
FALLA DE ORIGEN

Ejemplo: El lenguaje de palíndromas sin marca central $L = \{x \in (0+1)^* | x = \text{reverso}(x)\}$ es reconocido por un autómatas de pila no-determinista que funciona de acuerdo con el siguiente procedimiento:
Avanzando a la derecha, se almacena primeramente cada símbolo y cuando se "cree" estar a la mitad, se compara cada símbolo leído con el tope de la pila. Si coinciden, se continúa. En otro caso se marca un error. El no-determinismo del autómata está en que no se precisa en qué momento pasará al estado de desempilar. Transita a ése de manera indeterminada.

I.2.7.5. Autómata finito determinístico

Un autómata finito determinista consiste en un dispositivo que puede estar en un estado de entre un número finito de los mismos; uno de ellos será el estado inicial y por lo menos uno será estado de aceptación. Tiene un flujo de entrada por el cual llegan los símbolos de una cadena que pertenecen a un alfabeto determinado. Se detecta el símbolo y dependiendo de este y del estado en que se encuentre hará una transición a otro estado o permanece en el mismo. El mecanismo de control (programa) es que determina cual es la transición a realizar. La palabra finito se refiere a que hay un número finito de estados.

La palabra determinista es porque el mecanismo de control (programa) no debe tener ambigüedades, es decir, en cada estado solo se puede dar una y solo una (ni dos ni ninguna) transición para cada símbolo posible (en el ejemplo anterior, la tabla de transiciones era determinista en ese caso, no así el diagrama, aunque podría serlo como veremos mas tarde).

El autómata acepta la cadena de entrada si la máquina cambia a un estado de aceptación después de leer el último símbolo de la cadena. Si después del último símbolo la máquina no queda en estado de aceptación, se ha rechazado la cadena.

Si la máquina llega al final de su entrada antes de leer algún símbolo la entrada es una cadena vacía (cadena que no contiene símbolos) y la representaremos con λ. Solo aceptará λ si su estado inicial es de aceptación.

I.3 Estructuras de Datos

Para procesar información en un computador es necesario hacer una abstracción de los datos que tomamos del mundo real. Se hace una selección de los datos más representativos de la realidad a partir de los cuales pueda trabajar la computadora para obtener resultados.
Cualquier lenguaje suministra un subconjunto de tipos de datos simples, como son los números enteros, caracteres, números reales, ya que la memoria del ordenador es finita. El tamaño de todos los tipos de datos depende de la máquina y del compilador sobre los que se trabaja.

Una ESTRUCTURA DE DATOS es un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se busca es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Podemos clasificarlas como:

Elementales: aquellas cuya manipulación y representación se ha estandarizado en los lenguajes de programación, como son los números enteros, reales, boléanos, caracteres, arreglos.
Compuestas: aquellas cuya manipulación y representación requiere del ingenio del programador del tipo lineales pilas, colas, cola doble, lista doblemente ligada, lista circular o bien las no lineales graficas, árboles, árboles binarios.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:
Estáticas: su tamaño en memoria es fijo. Ejemplo: arrays.
Dinámicas: su tamaño en memoria es variable. Ejemplo: listas enlazadas con punteros, ficheros, etc.
Las denominaciones usadas son: Arrays, Listas Enlazadas, Árboles, Conjuntos, Pilas, Colas y Grafos

I.3.1. Tipos Abstractos de Datos

Los tipos abstractos de datos (TAD) permiten describir una estructura de datos en función de las operaciones que pueden efectuar, dejando a un lado su implementación. Además mezclan estructuras de datos junto a una serie de operaciones de manipulación. Incluyen una especificación, que es lo que verá el usuario, y una implementación (algoritmos de operaciones sobre las estructuras de datos y su representación en un lenguaje de programación), que el usuario no tiene necesariamente que conocer para manipular correctamente los tipos abstractos de datos. Se caracterizan por el encapsulamiento, lo que permite aumentar la complejidad de los programas pero manteniendo una claridad suficiente que no desborde a los desarrolladores. Un TAD puede definir a otro TAD. Como, construir pilas, a partir de arrays y listas enlazadas, etc.

I.3.2. Recursividad

Se dice que algo es recursivo si se define en función de sí mismo o a sí mismo.
Un programa puede definirse en términos recursivos, como una serie de pasos básicos, o paso base (también conocido como condición de parada), y un paso recursivo, donde vuelve a llamarse al programa. En un computador, esta serie de pasos recursivos debe ser finita, terminando con un paso base. Es decir, a cada paso recursivo se reduce el número de pasos que hay que dar para terminar. La recursividad también puede ser indirecta, si tenemos un procedimiento P que llama a otro Q y éste a su vez llama a P. También en estos casos debe haber una condición de parada.
Si se produce una llamada recursiva infinita, llega un momento en el que no quedará memoria para almacenar más datos, y en ese momento se abortará la ejecución del programa.

Algunos lenguajes de programación no admiten el uso de recursividad, como por ejemplo el ensamblador; no se debe utilizar cuando la solución iterativa sea clara a simple vista
El compilador transformará la solución recursiva en una iterativa, utilizando una pila, para cuando compile al código del computador.

Si no se conoce el número de elementos, se introduce un centinela o la constante NULO para punteros, u otros valores como el mayor o menor entero que la máquina pueda representar, para indicar el fin de la estructura.
La recursividad es una herramienta potente para resolver múltiples problemas. Es más, todo programa iterativo puede realizarse empleando expresiones recursivas y viceversa.

I.3.3. Matrices [Arrays]

Un array es un tipo de estructura de datos que consta de un número fijo de elementos del mismo tipo. Estos elementos se almacenan en posiciones contiguas de memoria. Estos elementos pueden ser variables o estructuras. Hay que tener cuidado con no utilizar un índice fuera de los límites, porque dará resultados inesperados (tales como cambio del valor de otras variables o finalización del programa, con error "invalid memory reference".

I.3.4. Listas

Una lista es una estructura de datos secuencial. Una manera de clasificarlas es por la forma de acceder al siguiente elemento:
Lista densa: la propia estructura determina cuál es el siguiente elemento de la lista.
Lista enlazada: la posición del siguiente elemento de la estructura la determina el elemento actual.

Es necesario almacenar al menos la posición de memoria del primer elemento. Además es dinámica, es decir, su tamaño cambia durante la ejecución del programa.

TESIS CON
FALLA DE ORIGEN

Una lista enlazada se puede definir recursivamente de la siguiente manera:
Una lista enlazada es una estructura vacía o un elemento de información y un enlace hacia una lista (un nodo).Gráficamente se suele representar así:

Pueden cambiar de tamaño, y ser flexibles a la hora de reorganizar sus elementos; a cambio se ha de pagar una mayor lentitud a la hora de acceder a cualquier elemento. Para añadir un nuevo nodo, con la información p, al inicio, basta con crear ese nodo, introducir la información p, y hacer un enlace hacia el siguiente nodo, que en este caso contiene la información x. Superando al array ya que no es necesario desplazar la información a la derecha. Al crear una lista debe estar vacía.

Operaciones básicas sobre listas

- Inserción al comienzo de una lista
- Recorrido de una lista.

I.3.4.1. Listas ordenadas

Las listas ordenadas son aquellas en las que la posición de cada elemento depende de su contenido. Cuando haya que insertar un nuevo elemento en la lista ordenada hay que hacerlo en el lugar que le corresponda, y esto depende del orden y de la clave escogidos. Este proceso se realiza en tres pasos:

1.- Localizar el lugar correspondiente al elemento a insertar. Se utilizan dos punteros: *anterior* y *actual*, que garanticen la correcta posición de cada enlace.

2.- Reservar memoria para él (puede hacerse como primer paso). Se usa un puntero auxiliar (*nuevo*) para reservar memoria.

3.- Enlazarlo. Esta es la parte más complicada, porque hay que considerar la diferencia de insertar al principio, no importa si la lista está vacía, o insertar en otra posición. Se utilizan los tres punteros antes definidos para actualizar los enlaces.

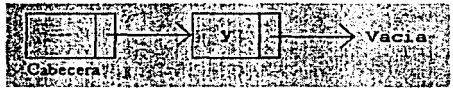
I.3.4.2. Listas reorganizables

Son aquellas en las que cada vez que se accede a un elemento éste se coloca al comienzo de la lista. Si el elemento al que se accede no está en la lista entonces se añade al comienzo de la misma. Cuando se trata de borrar un elemento se procede de la misma manera que en la operación de borrado de la lista ordenada. El orden en una lista reorganizable depende del acceso a un elemento, y no de los valores de las claves.

Cabecera ficticia y centinela

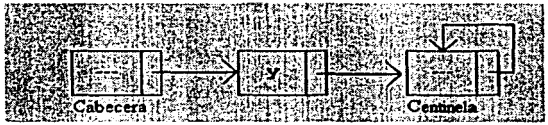
Al insertar o actualizar elementos en una lista ordenada o reorganizable es fundamental actualizar el primer elemento de la lista cuando sea necesario. Esto lleva un coste de tiempo, aunque sea pequeño salvo en el caso de numerosas inserciones y borrados. Para subsanar este problema se utiliza la cabecera ficticia.

La cabecera ficticia añade un elemento (sin clave, por eso es ficticia) a la estructura delante del primer elemento. Evitará el caso especial de insertar delante del primer elemento. Gráficamente se puede ver así:



El centinela es un elemento que se añade al final de la estructura, sirve para acotar los elementos de información que forman la lista, al buscar un elemento comprueba que no se está en una posición de información vacía, al tiempo que acelera la búsqueda.

En la búsqueda primero se copia la clave que buscamos en el centinela, y a continuación se hace una búsqueda por toda la lista hasta encontrar el elemento que se busca. Este se encontrará en cualquier posición de la lista, o bien en el centinela en el caso de que no estuviera en la lista. Cuando la lista está vacía la cabecera apunta al centinela. El centinela siempre se apunta a si mismo. Esto se hace así por convenio. Gráficamente se puede representar así:

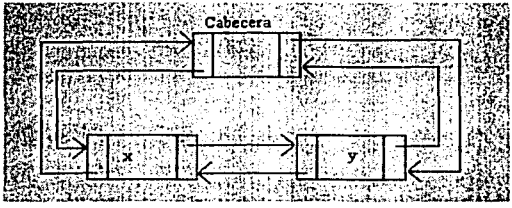


TESIS CON
FALLA DE ORIGEN

I.3.4.3. Listas doblemente enlazadas

Son listas que tienen un enlace con el elemento siguiente y con el anterior. Una ventaja es que pueden recorrerse en ambos sentidos, ya sea para efectuar una operación con cada elemento o para insertar / actualizar y borrar. Otra ventaja es que las búsquedas son algo más rápidas puesto que no hace falta hacer referencia al elemento anterior. Su inconveniente es que ocupan más memoria por nodo que una lista simple.

Es posible implementar una lista ordenada con doble enlace, enlazada con cabecera y centinela, para lo que se utiliza un único nodo que haga las veces de cabecera y centinela.



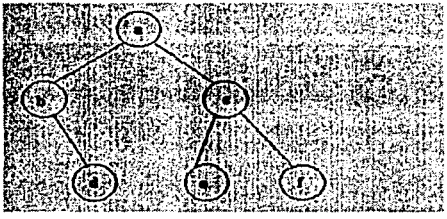
I.3.4.4. Listas circulares

Las listas circulares son aquellas en las que el último elemento tiene un enlace con el primero, pueden definir estructuras más complejas a partir de las listas, como arrays de listas. Ocasionalmente los grafos se definen como listas de adyacencia. Son eficaces para diseñar colas de prioridad, pilas y colas sin prioridad, y en general cualquier estructura cuyo acceso a sus elementos se realice de manera secuencial.

I.3.5. Árboles

Un árbol es una estructura de datos, que puede definirse de forma recursiva y no secuencial. O bien es un grafo acíclico, conexo y no dirigido. Es decir, es un grafo no dirigido en el que existe exactamente un camino entre todo par de nodos. Si dicho número de estructuras es inferior o igual a 2, se tiene un árbol binario con 0, 1 o 2 descendientes como máximo
Representación mediante un grafo:

TESIS CON
FALLA DE ORIGEN



I.3.5.1. Nomenclatura sobre árboles

- Raíz: es aquel elemento que no tiene antecesor; ejemplo: *a*.
- Rama: arista entre dos nodos.
- Antecesor: un nodo X es antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.
- Sucesor: un nodo X es sucesor de un nodo Y si por alguna de las ramas de Y se puede llegar a X.
- Grado de un nodo: el número de descendientes directos que tiene. Ejemplo: *c* tiene grado 2, *d* tiene grado 0, *a* tiene grado 2.
- Hoja: nodo que no tiene descendientes; grado 0. Ejemplo: *d*
- Nodo interno: aquel que tiene al menos un descendiente.
- Nivel: número de ramas que hay que recorrer para llegar de la raíz a un nodo. Ejemplo: el nivel del nodo *a* es 1 (es un convenio), el nivel del nodo *e* es 3.
- Altura: el nivel más alto del árbol. En el ejemplo de la figura 1 la altura es 3.
- Anchura: es el mayor valor del número de nodos que hay en un nivel. En la Fig.anterior, la anchura es 3.
- Es necesaria una jerarquía, es decir, que haya una única raíz.

I.3.6. Pilas

Una pila es una estructura de datos de acceso restrictivo a sus elementos. Se puede entender como una pila de libros que se amontonan de abajo hacia arriba. En principio no hay libros; después ponemos uno, y otro encima de éste, y así sucesivamente. Posteriormente los solemos retirar empezando desde la cima de la pila de libros, es decir, desde el último que pusimos, y terminaríamos por retirar el primero que pusimos.

TESIS CON
FALLA DE ORIGEN

La recursividad se simula en una computadora con la ayuda de una pila. Asimismo muchos algoritmos emplean las pilas como estructura de datos fundamental, por ejemplo para mantener una lista de tareas pendientes que se van acumulando.

Las pilas ofrecen dos operaciones fundamentales, que son apilar y desapilar sobre la cima. El uso que se les de a las pilas es independiente de su implementación interna. Es decir, se hace un encapsulamiento. Por eso se considera a la pila como un tipo abstracto de datos.
Es una estructura de tipo LIFO (Last In First Out), es decir, último en entrar, primero en salir.

La implementación de pilas, puede ser mediante arrays y mediante listas enlazadas. En ambos casos se cubren cuatro operaciones básicas: Inicializar, Apilar, Desapilar, y Vacía (nos indica si la pila está vacía).

El uso del array es idóneo cuando se conoce de antemano el número máximo de elementos que van a ser apilados y el compilador admite una región contigua de memoria para el array. En otro caso sería más recomendable usar la implementación por listas enlazadas, también si el número de elementos llegase a ser excesivamente grande.

La implementación por array es ligeramente más rápida. Si se implementa la pila mediante una lista enlazada entonces quedarían en memoria una serie de elementos que es necesario borrar. La única manera de borrarlos es liberar todas las posiciones de memoria que le han sido asignadas a cada elemento, esto es, desapilar todos los elementos. En el caso de una implementación con array esto no es necesario, salvo que quiera liberarse la región de memoria ocupada por éste.

TESIS CON
FALLA DE ORIGEN

CAPÍTULO II

Análisis, Definición y Diseño del Compilador

33-A
TESIS CON
FALLA DE ORIGEN

CAPÍTULO II. ANÁLISIS, DEFINICIÓN Y DISEÑO DEL COMPILADOR

II.1. Definición de Lenguaje Simbólico

El lenguaje coloquial es el lenguaje que utilizamos cotidianamente, para definir el lenguaje simbólico utilizado en Pathfinder se puede hacer una analogía representada en la siguiente tabla, entre el lenguaje coloquial y el lenguaje simbólico.

Lenguaje coloquial	Lenguaje simbólico (TOKEN)
"Avanza"	AV
"Vuelta Izquierda"	VI
"Vuelta Derecha"	VD
"Recoge Trompo"	RT
"Deja Trompo"	DT
"Pide Trompo"	PT

Definiendo como un lenguaje simbólico el conjunto de palabras y reglas que tienen su representación mediante símbolos. Símbolos que en nuestro caso llamaremos tokens.

II.1.1. Instrucciones Primitivas

Se iniciará explicando las seis instrucciones primitivas que componen el vocabulario de Pathfinder y que son: AVANZA (AV), VUELTA DERECHA (VD), VUELTA IZQUIERDA (VI), RECOGE TROMPO (RT), DEJA TROMPO (DT) Y PIDE TROMPO (PT).

Usando éstas instrucciones Pathfinder puede realizar diferentes tareas y moverse a través de su mundo, recoger o dejar trompos ya que cuentan con una acción predeterminada, por instrucción.

II.1.1.1. Cambio de Posición

Pathfinder entiende tres instrucciones primitivas que cambian su posición, estas instrucciones se definen a continuación:

Avanza (AV): Pathfinder avanza una celda.

Vuelta Izquierda (VI): Gira 90° sobre su posición a la izquierda.

El usuario es quien define la posición inicial del Pathfinder pudiendo ser esta norte, sur, este u oeste.

Vuelta Derecha (VD): Pathfinder ejecuta una instrucción Vuelta Derecha ejecutando tres veces la instrucción Vuelta Izquierda.

II.1.1.2. Manejando Trompos

Pathfinder cuenta con la posibilidad de manipular objetos llamados trompos y con una "bolsa virtual" que le permite guardar los mismos. Así como Pathfinder entiende tres cambios de posición, hay tres instrucciones primitivas que permiten manipular trompos. Dos de esas tres instrucciones son acciones opuestas.

Recoge Trompo (RT): Cuando Pathfinder ejecuta una instrucción Recoge Trompo, levanta uno desde la posición en la cual se encuentra y entonces lo deposita en su bolsa.

Deja Trompo (DT): Pathfinder ejecuta una instrucción Deja Trompo cuando extrae un trompo desde su bolsa y lo deja en la posición actual.

Pide Trompo (PT): Cuando se ejecuta una instrucción Pide Trompo, este colocará un trompo en la posición en donde se le indique.

II.1.1.3. Extensión del Vocabulario de Pathfinder

Con las seis instrucciones básicas anteriormente mencionadas se puede hacer un vocabulario extenso de Pathfinder que le permite realizar más acciones con menos instrucciones. También se pueden realizar ciclos iterativos, asignar tareas y censar el terreno. Estas instrucciones son las siguientes:

Instrucciones Condicionales: En Lenguaje ULP una condicional es una instrucción que debe cumplirse para poder una acción determinada. Las instrucciones condicionales de Pathfinder son las siguientes:

Algo Enfrente (AE): Instrucción que valida que la celda que se encuentre frente a Pathfinder este ocupada por una roca.

Nada Enfrente (NE): Instrucción que valida que la celda que se encuentre frente a Pathfinder este vacía.

Algo Izquierda (AI): Instrucción que valida que la celda que se encuentre a la izquierda de Pathfinder este ocupada.

TESIS CON
FALLA DE ORIGEN

Nada Izquierda (NI): Instrucción que valida que la celda que se encuentra a la derecha de Pathfinder este vacía.

Algo Derecha (AD): Instrucción que valida que la celda que se encuentra a la derecha de Pathfinder este ocupada.

Nada Derecha (ND): Instrucción que valida que la celda que se encuentra a la derecha de Pathfinder este vacía.

Algo Suena (AS): Instrucción que valida que el Pathfinder este encima de un trompo.

Nada Suena (NS): Instrucción que valida el que Pathfinder tenga algún trompo en su bolsa

Algún Trompo (AT): Instrucción que valida el que Pathfinder no tenga ningún trompo en su bolsa.

Dirección Norte (DN): Instrucción que condiciona al robot a dirigirse a la dirección norte.

Instrucciones Iterativas: En lenguaje ULP este tipo de instrucciones son capaces de realizar una instrucción ó bien una serie de instrucciones que además ofrecen la posibilidad de alternar, si una condición se cumple es posible enlazar otras iguales para que dentro de una iteración, de ser necesario se cumpla otra serie de instrucciones u otra iteración hasta que la condición se cumpla ó sea distinta.

SI (SI): Instrucción que se utiliza con las funciones primitivas y las condiciona para hacer algo en específico.

Luego (LG): Instrucción que condiciona que después de una instrucción primitiva o condicional sigue una instrucción primitiva ó también indica el fin de un ciclo donde se utilizó (SI).

Repite (RP): Instrucción que permite repetir una serie de instrucciones anteriormente dadas.

Hasta Que (HQ): Instrucción que detiene la instrucción repite, al poner una restricción ó condición.

Mientras (MI): Esta instrucción aplica cuando se requiere que Pathfinder realice una tarea cumpliendo cierta condición.

Instrucciones de Asignación: Estas tienen como finalidad el que Pathfinder aprenda, ó ejecute ciertas tareas.

Aprende Que (AQ): Instrucción que determina que Pathfinder guarde en su memoria ciertas instrucciones ó tareas que utilizará después.

Ejecuta (EJ): Comando que indica a Pathfinder que realice tareas previamente aprendidas, esta instrucción puede ir seguida por una serie de n tareas aprendidas.

Asigna (=): Este símbolo siempre va después de un (AQ) ya que permite asignar un identificador a una serie de instrucciones.

Instrucciones de Fin de Programa: Estas instrucciones sirven para indicar que un programa hecho en ULP finaliza.

Fin de programa (FI): Instrucción va al último de cualquier programa en ULP.

Fin de instrucción (.): Este símbolo siempre va al final de cada línea de código en ULP.

A continuación se presenta un resumen de las instrucciones arriba mencionadas.

Lenguaje Simbólico

Avanza	: AV	Algún Trompo	: AT
Vuelta Izquierda	: VI	Ningún Trompo	: NT
Vuelta Derecha	: VD	Dirección Norte	: DN
Recoge Trompo	: RT	Aprende Que	: AQ
Deja Trompo	: DT	Ejecuta	: EJ
Pide Trompo	: PT	Asigna	: =
Algo Enfrente	: AE	Condicional	: SI
Nada Enfrente	: NE	Dame Más (instrucción)	: DM
Algo Izquierda	: AI	Luego (condicional)	: LG
Nada Izquierda	: NI	Repite (Iteración)	: RP
Algo Derecha	: AD	Hasta Que (condicional	: HQ
Nada Derecha	: ND	Mientras (iteración)	: MI
Algo Suena	: AS	Nada Suena	: NS
Fin de Instrucción	: .	FI	: Final condicional

TESIS CON
FALLA DE ORIGEN

II.2. ANÁLISIS LÉXICO

El proceso de análisis léxico o de reconocimiento puede entenderse como la transformación de un flujo de caracteres en un flujo de símbolos "reducidos". En este caso "reducido" significa que la entrada es filtrada para eliminar aquellos elementos que solo sirven para hacer legible el programa. Por que de otro modo cualquier otra combinación no tiene sentido para el análisis.Entre las funciones características de un analizador léxico están:

- Eliminar espacios y comentarios.
- Reconocer identificadores y palabras clave.
- Reconocer constantes y números
- Generar un listado para el compilador.

El primer paso que se tomo para hacer el análisis fue que el compilador leyera las líneas de código

```
Sub Scanner()
   Dim Simbolo$
Inicio:
   If F1 Then                    '*** Scaneando una nueva línea
      F1 = 0: EN$ = FU$(DX): DX = DX + 1
      Avisa ("Scanning " + EN$)
      JJ = 0: LU = Len(EN$)
   End If
   Do                                '*** Ignora los espacios en blanco
     JJ = JJ + 1                     '*** y avanza en la lectura de la línea
     If JJ > Len(EN$) Then Exit Do   '*** hasta hallar algo que sea
   Loop Until Mid$(EN$, JJ, 1) <> " "  '*** diferente de un espacio.
   If JJ > Len(EN$) Then F1 = 1: GoTo Inicio
```

Ahora bien conforme va leyendo las líneas de código, se pueden empezar a hacer las funciones que hace el analizador que son:

Salto de operadores: Los espacios, tabuladores, comentarios y saltos de línea se denominan separadores y pueden ser muy numerosos en un código fuente, pero no proporcionan información para la traducción, por lo tanto deben ser eliminados del código entregado al analizador sintáctico.

Reconocimiento de operadores y nombres: Es fácil reconocer símbolos de operadores como +,-,*,/, porque constan de un solo carácter. Sin embargo hay otros operadores como <, <=,<>, := que comienzan con el mismo carácter y tienen significado diferente. En estos casos el analizador léxico tiene que hacer examinar el carácter siguiente por anticipado ó pre-análisis, para determinar el operador correcto.
Estas funciones específicas las realiza en las siguientes líneas de código.

BLOQUE A

```
    Do                              '*** Ignora los espacios en blanco
   JJ = JJ + 1                      '*** y avanza en la lectura de la línea
   If JJ > Len(EN$) Then Exit Do    '*** hasta hallar algo que sea
  Loop Until Mid$(EN$, JJ, 1) <> " "  '*** diferente de un espacio.
  If JJ > Len(EN$) Then F1 = 1: GoTo Inicio
```

BLOQUE B

```
Simbolo$ = ""
  Do While JJ <= Len(EN$)        '*** Lee caracteres mientras que no se acabe
   C$ = Mid$(EN$, JJ, 1)         '*** la línea de entrada. Los acumula en
   If C$ = " " Then Exit Do
   Simbolo$ = Simbolo$ & C$      '*** la variable local Simbolo
   JJ = JJ + 1
  Loop
  KK = 1: TX%(KK) = Asc(Left$(Simbolo$, 1))
  X2 = JJ: B1 = 0: B2 = 0
  If Left$(Simbolo$, 1) = "=" Then
```

En este bloque lo que se hace es que verifica si el primer carácter leído es un signo de igual, si es así entonces le agrega otro signo para hacer válida la regla de que se están manejando símbolos pares.

El reconocimiento de palabras clave (o reservadas) e identificadores en un flujo de caracteres, se basa en un autómata finito, que reconoce identificadores.

BLOQUE C

```
  TX%(2) = Asc(Mid$(Simbolo$, 2, 1))
```

Por medio de esta fórmula que llamaremos fórmula de dispersión se le asigna a un par de caracteres leídos un lugar en el vector de dispersión. Y guarda en X3 el valor del subíndice en donde debe estar el vector de nombres.

```
  If VD%(DD) = 0 Then
    GoTo 720
  End If
```

En el vector de dispersión se encuentran ubicados todos los tokens que componen el lenguaje, y existen casillas que se encuentran ubicadas a lo largo del vector de 200 lugares con un valor de cero es decir que esas casillas pueden ser ocupadas por un token leído por el programa.

```
  If VD%(DD) = 0 Then
    BE = 28
    BX$ = Simbolo$
    UX%(1) = TX%(1): UX%(2) = TX%(2)
    PA = 0
  Else
X3 = VD%(DD)
```

Asigna a X3 el valor numérico del token leído. Y sigue con el siguiente bloque.

TESIS CON
FALLA DE ORIGEN

BLOQUE D

```
  Ciclo:  If B1 = 1 Or B2 = 1 Or DN$(X3) = Simbolo$ Then
      BE = X3: B1 = 0
      If BE <= 31 Then PA = 0: Exit Sub
      If PA Then
        XF = XE: XE = BE: BE = 27: B3 = X3: Exit Sub
      Else
        XE = BE: BE = 27: B3 = X3: PA = 1
      End If
    Else
      If DL%(X3) = 0 Then
        BE = 28
        BX$ = Simbolo$
        UX%(1) = TX%(1): UX%(2) = TX%(2)
        PA = 0
      Else
        X3 = DL%(X3): GoTo Ciclo
      End If
    End If
  End If
End Sub
```

Se usan 2 vectores llamados TX%(1) y TX%(2) en donde a cada vector se le asigna una letra y lo que hace es aplicarles la fórmula de dispersión vista anteriormente para después asignarles un lugar en el vector de dispersión este valor debe ser menor o igual a 31 ya que los tokens que conforman el vocabulario de Pathfinder son 31.

La tarea más difícil para el analizador léxico es distinguir entre identificadores o palabras clave, lo cual puede hacerse con tablas de símbolos ó con una estructura de datos adicional que contenga todas las palabras reservadas.

Se puede decir que este bloque de código lo que hace básicamente es dejar en un vector los tokens leídos en el programa asignándoles a cada uno un valor y si encuentra un nuevo identificador le asignara el valor 28 en el vector, esto le sirve al compilador para tener una referencia hacia donde dirigirse cuando se hace el análisis sintáctico y semántico. También trata casos especiales como los espacios y el signo igual.

TESIS CON
FALLA DE ORIGEN

II.3. Determinación Sintáctica

Una vez definidos los símbolos validos que conformaran el lenguaje, es necesario determinar las reglas de conexión o precedencia para el uso de dichos símbolos, de ahí la necesidad de establecer la sintaxis que aplicara para los mismos.
Tomando como premisas las siguientes definiciones:

```
Instruc  = { AV
             VI
             VD
             RT
             DT
             PT
             SI cond listai LG
             SI cond listai DM listai LG
             MI cond listai LG
             RP listai HQ cond LG }

Listai   = listai + instruc

AQ       = listai
Id       = <letra> < letra/ dígito >
Cond       { AE, NE, AI, NI, AD, ND, AS, NS, AT, NT, DN }
```

La instrucción AV puede ser precedida por una o más de las siguientes instrucciones o identificador/es o fin de sentencia

```
AV -> AV
AV -> VI
AV -> VD
AV -> RT
AV -> DT
AV -> PT
AV -> SI
AV -> DM
AV -> LG
AV -> RP
AV -> HQ
AV -> MI
AV -> id
AV -> .

AV -> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / id / .
AV -> Instruc / id / .
```


TESIS CON
FALLA DE ORIGEN

La instrucción VI puede ser precedida por una o más de las siguientes instrucciones o identificador/es o fin de sentencia

VI --> AV
VI --> VI
VI --> VD
VI --> RT
VI --> DT
VI --> PT
VI --> SI
VI --> DM
VI --> LG
VI --> RP
VI --> HQ
VI --> MI
VI --> Id
VI --> .

VI --> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / Id / .
VI --> Instruc / id / .

La instrucción VD puede ser precedida por una o más de las siguientes instrucciones o identificador/es o fin de sentencia

VD --> AV
VD --> VI
VD --> VD
VD --> RT
VD --> DT
VD --> PT
VD --> SI
VD --> DM
VD --> LG
VD --> RP
VD --> HQ
VD --> MI
VD --> id
VD --> .

VD --> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / Id / .
VD --> Instruc / id / .

La instrucción RT puede ser precedida por una o más de las siguientes instrucciones o identificador/es o fin de sentencia

RT --> AV
RT --> VI
RT --> VD
RT --> RT
RT --> DT
RT --> PT
RT --> SI

RT --> DM
RT --> LG
RT --> RP
RT --> HQ
RT --> MI
RT --> id
RT --> .

RT --> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / id / .
RT--> Instruc / id / .

La instrucción DT puede ser precedida por una o más de las siguientes instrucciones o identificador/es o fin de sentencia

DT --> AV
DT --> VI
DT --> VD
DT --> RT
DT --> DT
DT --> PT
DT --> SI
DT --> DM
DT --> LG
DT --> RP
DT --> HQ
DT --> MI
DT --> id
DT --> .

DT --> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / id / .
DT --> Instruc / id / .

La instrucción PT puede ser precedida por una o más de las siguientes instrucciones o identificador/es o fin de sentencia

PT --> AV
PT --> VI
PT --> VD
PT --> RT
PT --> DT
PT --> PT
PT --> SI
PT --> DM
PT --> LG
PT --> RP
PT --> HQ
PT --> MI
PT --> id
PT --> .

PT --> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / id / .
PT --> Instruc / id / .

TESIS CON
FALLA DE ORIGEN

La condicional AE puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

AE –> AV
AE –> VI
AE –> VD
AE –> RT
AE –> DT
AE –> PT
AE –> SI
AE –> LG
AE –> RP
AE –> MI
AE –> id

AE –> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / id
AE –> instruc / id

La condicional NE puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

NE –> AV
NE –> VI
NE –> VD
NE –> RT
NE –> DT
NE –> PT
NE –> SI
NE –> LG
NE –> RP
NE –> MI
NE –> id

NE –> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / id
NE –> instruc / id

La condicional AI puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

AI –> AV
AI –> VI
AI –> VD
AI –> RT
AI –> DT
AI –> PT
AI –> SI
AI –> LG
AI –> RP
AI –> MI
AI –> id

AI –> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / id

TESIS CON
FALLA DE ORIGEN

AI --> instruc / id

La condicional NI puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

NI --> AV
NI --> VI
NI --> VD
NI --> RT
NI --> DT
NI --> PT
NI --> SI
NI --> LG
NI --> RP
NI --> MI
NI --> id

NI --> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / Id
NI --> instruc / id

La condicional AD puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

AD --> AV
AD --> VI
AD --> VD
AD --> RT
AD --> DT
AD --> PT
AD --> SI
AD --> LG
AD --> RP
AD --> MI
AD --> id

AD --> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / Id
AD --> instruc / Id

La condicional ND puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

ND --> AV
ND --> VI
ND --> VD
ND --> RT
ND --> DT
ND --> PT
ND --> SI
ND --> LG
ND --> RP
ND --> MI

ND --> id

ND --> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / id
ND --> instruc / id

La condicional AS puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

AS --> AV
AS --> VI
AS --> VD
AS --> RT
AS --> DT
AS --> PT
AS --> SI
AS --> LG
AS --> RP
AS --> MI
AS --> id

AS --> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / id
AS --> instruc / id

La condicional NS puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

NS --> AV
NS --> VI
NS --> VD
NS --> RT
NS --> DT
NS --> PT
NS --> SI
NS --> LG
NS --> RP
NS --> MI
NS --> id

NS --> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / id
NS --> instruc / id

La condicional AT puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

AT --> AV
AT --> VI
AT --> VD
AT --> RT
AT --> DT
AT --> PT
AT --> SI
AT --> LG

TESIS CON
FALLA DE ORIGEN

AT –> RP
AT –> MI
AT –> id

AT –> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / Id
AT –> instruc / id

La condicional NT puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

NT –> AV
NT –> VI
NT –> VD
NT –> RT
NT –> DT
NT –> PT
NT –> SI
NT –> LG
NT –> RP
NT –> MI
NT –> id

NT –> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / Id
NT –> instruc / id

La condicional DN puede estar precedida por una o más de las siguientes instrucciones o por identificador/es

DN –> AV
DN –> VI
DN –> VD
DN –> RT
DN –> DT
DN –> PT
DN –> SI
DN –> LG
DN –> RP
DN –> MI
DN –> id

DN –> AV / VI / VD / RT / DT / PT / SI / LG / RP / MI / Id
DN –> instruc / id

La instrucción " AQ solo puede ser precedida por un nuevo identificador

AQ –> nid

EJ puede ser precedida por cualquiera de las siguientes instrucciones o identificador/es

EJ –> AV
EJ –> VI
EJ –> VD
EJ –> RT

EJ –> DT
EJ –> PT
EJ –> SI
EJ –> RP
EJ –> MI
EJ –> id

EJ –> AV / VI / VD / RT / DT / PT / SI / RP / MI / id
EJ –> instruc / id

= puede ser precedida por cualquiera de las siguientes instrucciones o identificador/es

= –> AV
= –> VI
= –> VD
= –> RT
= –> DT
= –> PT
= –> SI
= –> RP
= –> MI
= –> id

= -> AV / VI / VD / RT / DT / PT / SI / RP / MI / id
= -> instruc / id

La instrucción SI puede ser precedida por solo una de las siguientes condicionales

SI –> AE
SI –> NE
SI –> AI
SI –> NI
SI –> AD
SI –> ND
SI –>AS
SI –> NS
SI –> AT
SI –> NT
SI –> DN

SI –> AE / NE / AI / NI / AD / ND / AS / NS / AT / NT / DN

SI –> cond

La instrucción DM puede ser precedida por una o más de las siguientes instrucciones o identificador/es

DM –> AV
DM –> VI
DM –> VD
DM –> RT

TESIS CON
FALLA DE ORIGEN

DM --> DT
DM --> PT
DM --> SI
DM --> RP
DM --> MI
DM --> id

DM --> AV / VI / VD / RT / DT / PT / SI / RP / MI / Id
DM --> instruc / id

La instrucción LG puede ser precedida por una o más de las siguientes instrucciones o identificador/es o terminador de sentencia

LG --> AV
LG --> VI
LG --> VD
LG --> RT
LG --> DT
LG --> PT
LG --> SI
LG --> DM
LG --> LG
LG --> RP
LG --> HQ
LG --> MI
LG --> id
LG --> .

LG --> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / id / .
LG --> instruc / id / .

La instrucción RP puede ser precedida por una o más de las siguientes instrucciones o identificador/es

RP --> AV
RP --> VI
RP --> VD
RP --> RT
RP --> DT
RP --> PT
RP --> SI
RP --> RP
RP --> MI
RP --> id

RP --> AV / VI / VD / RT / DT / PT / SI / RP / MI / id
RP --> instruc / id

TESIS CON
FALLA DE ORIGEN

La instrucción HQ puede ser precedida por solo una de las siguientes condicionales

HQ –> AE
HQ –> NE
HQ –> AI
HQ –> NI
HQ –> AD
HQ –> ND
HQ –> AS
HQ –> NS
HQ –> AT
HQ –> NT
HQ –> DN

HQ –> AE / NE / AI / NI / AD / ND / AS / NS / AT / NT / DN
HQ –> cond

La instrucción MI puede ser precedida por solo una de las siguientes condicionales

MI –> AE
MI –> NE
MI –> AI
MI –> NI
MI –> AD
MI –> ND
MI –> AS
MI –> NS
MI –> AT
MI –> NT
MI –> DN

MI –> AE / NE / AI / NI / AD / ND / AS / NS / AT / NT / DN
MI –> cond

Identificador id puede ser precedido por una o más de las siguientes instrucciones o identificador/es o terminador de sentencia

id –> AV
id –> VI
id –> VD
id –> RT
id –> DT
id –> PT
id –> SI
id –> DM
id –> LG
id –> RP
id –> HQ
id –> MI
id –> id
id –> .

TESIS CON
FALLA DE ORIGEN

id –> AV / VI / VD / RT / DT / PT / SI / DM / LG / RP / HQ / MI / id / ..
id –> instruc / id / .

Nuevo identificador nid puede ser precedida solo por "asignacion" =

nid –> =

Fin de sentencia puede ser precedido por aprende que o por ejecuta o por fin de programa

. –> AQ

. –> EJ

. –> FI

. –> AQ / EJ / FI

Fin de programa

FI –> .

Lo anterior podemos expresarlo a través de una matriz de precedencia que sea utilizada por el compilador para el análisis sintáctico de los programas realizados por el usuario . **Fig 2.3.1.**

Una vez que se definió la gramática estructural del lenguaje el compilador lo que hace es que pasando por el rastreador y luego por la máquina Q verifica que la secuencia leída tenga una lógica aceptable y la valida a través de precedencia arriba ilustrada. El código que hace posible esto es la Subrutina llamada Máquina Q.

```
Sub MAQUINAQ()
  '*** La Maquina-Q obtiene un Token proporcionado por el RASTREADOR (Scanner)
  '*** y verifica que sea procedente la secuencia con los Tokens anteriores.
  '*** Para ello, ocupa la TABLAQ(). La variable TY guarda el resultado de la
  '*** evaluacion. Si TY=0 => Error. Si TY=1 => Continuar. Si TY=2 => Reducir
  If Q1 <> 1 Then
    RQ = SX%(TEX)
  Else
    F1 = 1: Q1 = 0: Scanner
    RQ = BE: TEX = 1: SX%(TEX) = BE
    Scanner
  End If
  Do
    TY = TABLAQ(RQ, BE)
    If (RQ = 28) And (BE = 20) Then
      YU$ = BX$
    End If
    If TY <> 1 Then
      Exit Do
```


TESIS CON
FALLA DE ORIGEN

```
    End If
    RQ = BE: TEX = TEX + 1
    SX%(TEX) = BE
    Scanner
  Loop Until False
End Sub
```

	AV	VI	VD	RT	DT	PT	AE	NE	AI	NI	AD	ND	AS	NS	AT	NT	DN	AQ	EJ	**	SI	DM	LG	RP	HQ	MI	id	nid	.	FI
AV	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
VI	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
VD	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
RT	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
DT	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
PT	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
AE	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
NE	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
AI	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
NI	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
AD	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
ND	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
AS	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
NS	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
AT	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
NT	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
DN	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	2	2	2	2	0	0	0
AQ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
EJ	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	0
**	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	0
SI	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
DM	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	0
LG	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
RP	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	0
HQ	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
MI	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
id	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
nid	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1
FI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
prog-lsidef-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
def-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
ejec-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
lstai-	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	2	0
instruc-	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	0
cond-	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	1	0	0

Fig 2.3.1 Matriz de Procedencia

II.4. Análisis Semántico: Gramática del lenguaje

Ha sido expuesto anteriormente el vocabulario de Pathfinder. En él se han presentado todos los símbolos (tokens) que entiende Pathfinder. Sin embargo, la unión de esos tokens (sentencias) de acuerdo a una cierta gramática, es la que le dará significado (semántica) a esas sentencias.
La gramática de Pathfinder es muy sencilla: consiste en realidad de pocos símbolos. Estos símbolos se enlistan a continuación:

- `id-`
- `nid-`
- `instruc-`
- `listai-`
- `def-`
- `listdef-`
- `cond-`
- `ejec-`
- `prog-`

Cada uno de estos símbolos tienen su equivalencia tanto en tokens del vocabulario de Pathfinder, como en los propios símbolos entre sí. Será explicado a continuación el significado y equivalencias de cada uno de estos símbolos.

`id-`

Una `id-` (abreviatura de *identificador*) es un nombre que equivale a una definición de instrucciones de Pathfinder. Dentro del lenguaje de Pathfinder es posible encapsular una o más instrucciones ya conocidas por Pathfinder y asignarles un nombre (identificador) único e irrepetible. De esta manera, es posible "enseñarle" nuevas instrucciones al robot. Desde el punto de vista del robot, será exactamente igual invocar este identificador o invocar la secuencia de instrucciones que definen a este identificador.

`nid-`

Un `nid-` (abreviatura de *nuevo identificador*) es un `id-` nuevo dentro del vocabulario de Pathfinder. Los `nid-` son empleados dentro de la estructura AQ ("Aprende Que"). La estructura AQ permite definir que un `nid-` equivale a una secuencia de instrucciones dada; una vez que Pathfinder registra satisfactoriamente la equivalencia de un `nid-` dentro de su vocabulario, es posible invocarlo en cualquier porción de programa donde sea aceptable un `id-`. Desde un punto de vista léxico, un `nid-` se define como la secuencia de dos (y sólo dos) letras o dígitos, y cuya combinación no se encuentre registrada en el diccionario de Pathfinder. Ejemplos de `nid-` válidos son los siguientes: MV, T1, PP, 12, 1B, B1, BB, 75, etc.

TESIS CON
FALLA DE ORIGEN

instruc-

Una **instruc-** (abreviatura de *instrucción*) es el más elemental de los símbolos en los que se reducen los tokens del lenguaje de Pathfinder. En la siguiente tabla se define el equivalente de un **instruc-**.

AV		**id-**
VD		MI **cond- listai-** LG
VI		RP **listai-** HQ **cond-** LG
RT		SI **cond- listai-** DM **listai-** LG
DT		SI **cond- listai-** LG
PT		

De esta tabla se desprende que una **instruc-** es cualquiera de las seis instrucciones primitivas de Pathfinder (AV, VD, VI, RT, DT, PT). Cada vez que el Autómata de Pathfinder se encuentre con un AV, por ejemplo, sabrá que debe de reducirlo a una **instruc-** (para propósitos de reconocimiento de la validez gramatical de la sentencia). Se observa también que un **id-** es un token que puede ser reducido a una **instruc-**: de hecho, al ser una **id-** una invocación a una definición ya existente dentro del vocabulario de Pathfinder, ésta definición debe ser considerada como una más de las instrucciones que "entiende" Pathfinder. Adicionalmente, una estructura condicional bien formada, o una estructura de control iterativa, se reducen también a **instruc-**.

listai-

Una **listai-** (abreviatura de *lista de instrucciones*) es precisamente el símbolo en el que se reduce una **instruc-**. Cada vez que el Autómata encuentra una **instruc-**, la reduce inmediatamente a una **listai-**. Y cada vez que halla la secuencia: **listai- instruc-** , esta secuencia es reducida a un **listai-**. Esto posibilita que Pathfinder vaya aceptando una secuencia de instrucciones como válida.

def-

Una **def-** (abreviatura de *definición*) es la manera en la que se reduce gramaticalmente una estructura de definición de nuevas tareas en Pathfinder. La definición gramatical de una **def-** es la que se muestra a continuación:

AQ **nid- = listai-** ..

TESIS CON
FALLA DE ORIGEN

Note usted el terminador de instrucción (**.**) que aparece al final de la definición anterior. Obsérvese que una **def-** es la manera de reducir una sentencia que cumple con la sintaxis de una estructura AQ. De igual manera, obsérvese que la definición de un **def-** implica a otros dos símbolos, que son el **nid-** y la **listai-**.

listdef-

Una **listdef-** (abreviatura de *lista de definiciones*) es la manera en la que se reduce gramaticalmente una **def-**. Cada vez que el Autómata encuentra una **def-**, la reduce inmediatamente a una **listdef-**. Y cada vez que halla la secuencia: **listdef- def-** , esta secuencia es reducida a un **def-**. Esto posibilita que Pathfinder vaya aceptando una secuencia de definiciones como válida.

cond-

Una **cond-** (abreviatura de *condicional*) es la manera en la que gramaticalmente se reducen los tokens condicionales del lenguaje de Pathfinder. Hay que recordar que los tokens condicionales del lenguaje aparecen en las estructuras SI, RP y MI. En la siguiente tabla se enlistan los tokens condicionales que son reducidos a un **cond-**.

AE	Algo Enfrente	AS	Algo Suena
NE	Nada Enfrente	NS	Nada Suena
AI	Algo a la Izquierda	AT	Algún Trompo
NI	Nada a la Izquierda	NT	Ningún Trompo
AD	Algo a la Derecha	DN	Dirección Norte
ND	Nada a la Derecha		

De esta tabla se desprende que una **cond-** es cualquiera de las once palabras reservadas de Pathfinder que expresan un condicional. Cada vez que el Autómata de Pathfinder se encuentre con un NE, por ejemplo, sabrá que debe de reducirlo a una **cond-** (para propósitos de reconocimiento de la validez gramatical de la sentencia).

ejec-

Una **ejec-** (abreviatura de *ejecutar*) simboliza la porción de un programa en Pathfinder que es "ejecutable". La definición gramatical de un **ejec-** es la que se muestra a continuación:

EJ **listai-** **.**

TESIS CON
FALLA DE ORIGEN

Note usted el terminador de instrucción (`..`) que aparece al final de la definición anterior. Se observa de esta definición, que una instrucción EJ sólo puede ejecutar una `listai-` (recurra a la definición de `listai-` proporcionada más arriba). Si usted aplica recursivamente la definición de `listai-`, se podrá dar cuenta de que lo único ejecutable en un programa escrito en Pathfinder es una `listai-`.

`prog-`

Un `prog-` (abreviatura de *programa*) es el símbolo que define lo que es un programa, gramaticalmente hablando. Este es el símbolo al que debe ser reducido todo programa escrito en Pathfinder. Si esta reducción es exitosa, entonces la compilación será exitosa. La definición de un `prog-` se muestra a continuación:

`listdef-` `ejec-`

Un `prog-` se define como la secuencia de una `listdef-` seguida de un `ejec-`. De acuerdo a esta definición, siempre tendríamos que introducir una `listdef-` en todo programa escrito en Pathfinder. Esto parece un poco restrictivo y le resta flexibilidad al lenguaje. Sería deseable que un programa válido en Pathfinder fuera simplemente:

EJ AV AV AV VD AV RT AV AV DT AV AV .

Note usted que esta propuesta de programa no contempla ninguna `def-`. Dado que no podemos tener dos definiciones de `prog-` en la gramática del lenguaje de Pathfinder, será manejado este caso, mediante una excepción en el proceso de compilación (será insertado un `listdef-` ficticio cuando sea detectado que el programa cae en el caso descrito anteriormente). La razón por la que no podemos tener dos definiciones de `prog-` en la gramática, es porque esto causaría que la gramática de Pathfinder fuera ambigua.

TESIS CON
FALLA DE ORIGEN

II.5 Generación de Código Ejecutable

Una vez que la Máquina-Q ha procesado la validez gramatical de una sentencia, es necesario generar el código ejecutable que le corresponde. Pathfinder reconoce únicamente 22 instrucciones ejecutables como tales; esto es, el set de instrucciones de la Máquina-P (máquina ejecutora de Código-P) consiste únicamente de 22 instrucciones. Haciendo una analogía con los procesadores de una computadora, podríamos hablar del set de instrucciones del procesador. En la siguiente tabla se muestran las 22 instrucciones que reconoce la Máquina-P:

Instrucción	Significado
1	Avanza. Desplaza a Pathfinder una posición. Actualiza registros.
2	Vuelta Izquierda. Giro de 90 grados a la izquierda.
3	Vuelta Derecha. Giro de 90 grados a la derecha.
4	Recoger Trompo. Incrementa en uno el Contador de Trompos. Borra el trompo del Terreno (planeta).
5	Dejar Trompo. Decrementa en uno (si es posible) el Contador de Trompos. Pinta un trompo en el Terreno (planeta).
6	Pedir Trompo. Incrementa en uno el Contador de Trompos.
7	Algo Enfrente. Actualiza KL con el valor de AE.
8	Nada Enfrente. Actualiza KL con el valor de NE.
9	Algo a la Izquierda. Actualiza KL con el valor de AI.
10	Nada a la Izquierda. Actualiza KL con el valor de NI.
11	Algo a la Derecha. Actualiza KL con el valor de AD.
12	Nada a la derecha. Actualiza KL con el valor de ND.
13	Algo Suena. Actualiza KL con el valor de AS.
14	Nada Suena. Actualiza KL con el valor de NS.
15	Algun Trompo. KL=1 si Pathfinder trae trompos.
16	Ningún Trompo. KL=1 si Pathfinder NO trae trompos.
17	Dirección Norte. KL=1 si Pathfinder mira al Norte.
18	Salta en Falso. Salta el número de posiciones en memoria indicadas por el PC+1
19	Marca Snack. Guarda en el Stack la posición PC-1
20	Entra a Subrutina. Guarda en el Stack la posición PC+1. Carga el PC con el valor de la posición de memoria indicado por PC.
21	Regreso (Desmarca Stack). Obtiene del Stack el valor del PC.
22	Bota Stack. Decrementa en uno el valor del SP.

TESIS CON
FALLA DE ORIGEN

Antes de continuar, debe observarse lo siguiente:

a) Las primeras seis instrucciones corresponden a los únicos verbos de ULP. Esto es, las únicas instrucciones que generan acciones de Pathfinder son las primeras seis.

b) ULP cuenta con once instrucciones condicionales. Cada una de estas instrucciones condicionales devuelven un valor lógico de *verdadero* o *falso*, dependiendo del estado del registro lógico KL. El invocar la instrucción 13 implica que Pathfinder activa el valor de KL con el resultado de evaluar si Algo Suena; el invocar la instrucción 10 implica que Pathfinder actualiza el valor del registro lógico KL con el resultado de evaluar si Nada a la Izquierda, y así sucesivamente.

c) ULP cuenta con cinco instrucciones que le permiten manejar el stack de posiciones dentro de la memoria de Pathfinder. Con esas cinco instrucciones la Máquina-P es capaz de ingresar y regresar de subrutinas, saltar en falso, marcar y desmarcar el stack de posiciones.

d) Es importante recordar que el equivalente de los registros de un procesador son los registros KL, RS, KN. KL es el registro lógico que almacena el resultado de una evaluación lógica. RS es el audífono de Pathfinder y se activa con un valor de verdadero siempre que Pathfinder está parado encima de un trompo, y se activa con un valor de falso cuando Pathfinder no está parado encima de un trompo. Finalmente, KN es el contador de trompos que tiene Pathfinder; este registro almacena la cantidad de trompos que almacena Pathfinder en su bodega de carga.

La manera en la que Maquina-P ejecuta el código cargado en la memoria de Pathfinder es la siguiente:

a) Maquina-P cuenta con una memoria consistente de un vector de casillas; cada una de ellas es capaz de alojar una y solo una de las instrucciones mostradas en la tabla anterior.

b) Maquina-P cuenta con un Indice de Programa (IP, o *program-counter*) que se encarga de apuntar a la siguiente localidad de memoria de la cual se hará la extracción (*fetch*) de la siguiente instrucción a ejecutar. En todo momento, IP apunta a la siguiente casilla de memoria a leer.

c) En esa posición de memoria, Máquina-P puede hallar una de tres posibles clases de instrucción: una acción, una evaluación o una instrucción de manejo del stack.

d) Maquina-P inicia su ejecución en la posición indicada por el IP. Cada vez que Máquina-P hace una extracción de una instrucción de memoria, incrementa en uno el valor de IP.

e) Cada instrucción extraída de memoria de esta forma es ejecutada por la Máquina-P.

TESIS CON
FALLA DE ORIGEN

f) Maquina-P detiene la ejecución del programa cuando halla una instrucción nula, es decir, cuando halla una instrucción igual a cero.

II.5.1. Patrones de Código Generado.

En este apartado serán ilustrados los patrones a los que el compilador traduce las posibles instrucciones halladas en un programa fuente.

II.5.1.1. Acciones

Como fue expuesto anteriormente, las instrucciones que causan una acción en Pathfinder son seis y corresponden a: AV, VI, VD, RT, DT, PT. Estas instrucciones se traducen de manera directa por el Generador de Código a las siguientes equivalencias: AV=1, VI=2, VD=3, RT=4, DT=5 y PT=6. Estos valores serán almacenados directamente en memoria.

II.5.1.2. Condicionales

Los condicionales son instrucciones que causan una toma de decisión por parte de Pathfinder durante el tiempo de ejecución. Dependiendo del valor que adopte KL causado por el condicional a evaluar, deberá ser tomada una decisión. De acuerdo al entorno en el que sean empleadas, se cuentan con dos grandes grupos de condicionales: los condicionales utilizados en una estructura de toma de decisión de la forma SI cond listai- LG, o bien SI cond listai- DM listai- LG. Adicionalmente se cuenta con el empleo de condicionales en estructuras iterativas, las cuales serán explicadas en el siguiente apartado.

Una estructura condicional del tipo **SI cond listai- LG** es traducida por el compilador de la siguiente manera:

...	cond-	18	*offset*	instruc1	instruc2	....	instrucN	...

Por ejemplo, si la sentencia a traducir es: **SI NI VI AV AV LG** , ésta sería traducida de la siguiente manera:

...	10	18	4	2	1	1	...

Obsérvese que el valor 10 significa que va a ser evaluado el condicional "NI". En el caso de que esta evaluación active el registro KL con un valor de verdadero, Maquina-P "sabe" que debe de "saltar" dos posiciones de memoria, por lo que la siguiente instrucción a ejecutar sería la

TESIS CON
FALLA DE ORIGEN

instrucción 2 (que es un VI) y después dos instrucciones 1 (AV). Sin embargo, en el caso de que la evaluación de la instrucción 10 active el registro KL con un valor de falso, Máquina-P sabe que debe de "saltar" el número de posiciones de memoria indicado en el *offset* (en este caso, 4) señalado por el IP (es importante recordar que el IP apunta a la siguiente posición de memoria de la cual será extraída la instrucción a ejecutar); de esta manera, si la evaluación de condicional es falsa, Maquina-P salta las instrucciones 2,1,1 y continúa con la ejecución de las siguientes instrucciones almacenadas en memoria.

Siempre que la evaluación de un condicional cause que KL contenga el valor de verdadero, Máquina-P "sabe" que debe saltar dos posiciones de memoria para que de esta manera "salte" la instrucción 18 y el valor del *offset*. Esta es la razón por la que salte el número fijo de dos casillas.

Una estructura condicional del tipo **SI cond listal- DM listal- LG** es traducida por el compilador de la siguiente manera:

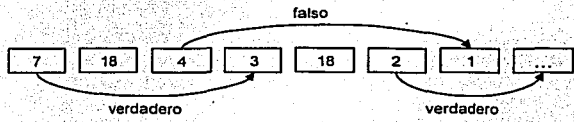
...	cond-	18	*offset1*	instruc1	18	*offset2*	instruc2	...

Por ejemplo, si la sentencia a traducir es: **SI AE VD DM AV LG** , ésta sería traducida de la siguiente manera:

...	7	18	4	3	18	2	1	...

Obsérvese que el valor 7 significa que va a ser evaluado el condicional "AE". En el caso de que esta evaluación active el registro KL con un valor de verdadero, Maquina-P "sabe" que debe de "saltar" dos posiciones de memoria, por lo que la siguiente instrucción a ejecutar sería la instrucción 3 (que es un VD); posteriormente ejecutaría la instrucción 18 y la instrucción 2, con lo que saltaría dos casillas, omitiendo de esta manera la ejecución de la instrucción 1, que constituye el "else" de la estructura condicional. Sin embargo, en el caso de que la evaluación de la instrucción 7 active el registro KL con un valor de falso, Máquina-P sabe que debe de "saltar" el número de posiciones de memoria indicado en el *offset* señalado por el IP (es importante recordar que el IP apunta a la siguiente posición de memoria de la cual será extraída la instrucción a ejecutar); de esta manera, si la evaluación de condicional es falsa, Maquina-P salta las instrucciones 3, 18 y 2 y continúa con la ejecución de la instrucción 1, que es la parte "else" de la estructura condicional.

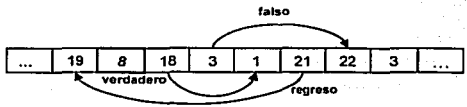
El diagrama que se muestra a continuación permite comprender más fácilmente el flujo que sigue la ejecución del programa dependiendo del valor que arroje la evaluación del condicional:



II.5.1.3. Ciclos Iterativos

Los ciclos iterativos son aquellos que causan la repetición de un bloque de instrucciones (una listal-) dependiendo del valor que adopte la evaluación de un condicional. ULP cuenta con dos tipos de estructuras iterativas: **MI cond listal- LG** (la cual es muy similar a una estructura DO-WHILE de un lenguaje estructurado) y la estructura **RP listal- HQ cond LG** (la cual se asemeja a la estructura REPEAT-UNTIL de un lenguaje estructurado).

Por ejemplo, la sentencia: **MI NE AV LG VD** sería traducida de la siguiente manera:



La instrucción 19 significa *marcar stack*, con lo cual es almacenada la posición del IP-1 (Índice de Programa). A continuación viene la evaluación del condicional 8 (que es un NE): si esta evaluación arroja un valor de verdadero, la Maquina-P sabe que debe de saltar dos posiciones de memoria, llegando a la instrucción 1 (AV). Una vez ejecutada la instrucción 1, se ejecuta la instrucción 21, que significa obtener del stack el valor del IP (Índice de Programa); hay que recordar que este valor es el que fue guardado con la instrucción 19.

Este ciclo es repetido mientras que la instrucción 8 (NE) arroje un valor verdadero. Cuando la evaluación de la instrucción 8 arroja un valor falso, la Máquina-P continua con la lectura de la información contenida en memoria, obteniendo una instrucción 18 (salta en falso) que causa que

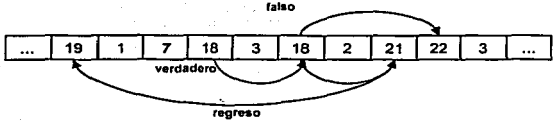
TESIS CON
FALLA DE ORIGEN

el programa salte 3 posiciones de memoria, con lo cual la ejecución del programa se traslada a la instrucción 22 (Bota Stack) y continua adelante con la instrucción 3 (VD). Cabe observar que la ejecución de la instrucción 22 tiene como objetivo el "limpiar" el stack, decrementando en uno el apuntador al tope del stack (*stack pointer*).

La estructura anterior realiza la evaluación del condicional antes de ejecutar ninguna acción, decidiendo si estas acciones deben ser ejecutadas o no, dependiendo del resultado de la evaluación del condicional.

ULP cuenta con otro tipo de estructura iterativa que, a diferencia de la anterior, realiza la evaluación del condicional una vez que ha hecho una pasada del bloque de instrucciones a repetir. Esta estructura es: **RP listal- HQ cond LG** (la cual es muy similar a una estructura REPEAT-UNTIL de un lenguaje estructurado).

Por ejemplo, la sentencia: **RP AV HQ AE LG VD** sería traducida por el compilador de la siguiente manera:



La instrucción 19 significa *marcar stack*, con lo cual es almacenada la posición del IP-1 (Indice de Programa). A continuación viene el bloque de instrucciones, que en este caso es solamente la instrucción 1 (AV). Inmediatamente después aparece la evaluación del condicional 7 (que es un AE): si esta evaluación arroja un valor de verdadero, la Maquina-P sabe que debe de saltar dos posiciones de memoria, llegando a una nueva instrucción 18, y dado que la ultima evaluación del ultimo condicional arrojó un verdadero, vuelve a saltar dos instrucciones, posicionando el IP en la instrucción 21 que significa obtener del stack el valor del IP (Indice de Programa).

Ahora bien, si la evaluación del condicional fue falsa, entonces salta tres instrucciones, posicionando el IP en la casilla que contiene una instrucción 22 que (Bota Stack) y continúa adelante con la siguiente instrucción, que es un 3 (VD). Nuevamente, cabe observar que la ejecución de la instrucción 22 tiene como objetivo el "limpiar" el stack, decrementando en uno el apuntador al tope del stack (*stack pointer*).

TESIS CON
FALLA DE ORIGEN

II.5.1.4. Llamadas a Subrutinas

Las llamadas a subrutinas son logradas mediante el uso de la instrucción 20 y de la instrucción 21. La instrucción 20 causa dos acciones: la primera es guardar en el stack el valor del IP+1y la segunda acción es cargar el IP con el valor de la posición de memoria indicado por PC. La combinación de estas dos acciones causa el "salto a una subrutina". La instrucción 21 tiene como objetivo el recargar el IP con el valor almacenado en el Stack, con lo que se logra el equivalente de una instrucción "*return*" de un lenguaje estructurado convencional.

II.6. Rutinas Funcionales de Pathfinder

En este apartado se muestra una relación de las principales rutinas funcionales que son necesarias para el funcionamiento de la aplicación. Se presentan, el listado de dichas rutinas, sus parámetros de entrada (si es que requieren de algún parámetro de entrada) y los valores de salida o las acciones que deben realizar. Este listado servirá como una referencia para el desarrollo de la aplicación misma.

II.6.1. Rutinas del proceso de Compilación

Las principales rutinas que intervendrán en el proceso de compilación son las siguientes:

- **AUTOMATA()** Autómata "A" (Árbol Gramatical). Realiza la reducción de los tokens adquiridos para convertirlos en un símbolo aceptable por la gramática del lenguaje. Emplea el vector SX%() donde se hallan almacenados los tokens recabados por el scanner.

- **LOADGRAMTREE()** Esta rutina se encarga de cargar en los arreglos AC%(), AT%(), AP%() y NR%() el árbol gramatical contenido en un archivo de datos externo denominado ARBOL.DAT.

- **LOADMAINDIR()** Se encarga de cargar en los arreglos DN$() y T$() la definición del vocabulario básico de Pathfinder. Este vocabulario básico se halla almacenado en un archivo de datos externo denominado CMD.DAT. Adicionalmente, inicializará apuntadores de trabajo.

- **LOADMATRIX()** Carga en la matriz TABLAQ() la Matriz de Precedencia del lenguaje. Esta matriz se almacena en un archivo de datos externo denominado MATRIZP.DAT.

- **LoadVDLink()** Esta rutina se encargará de pre-cargar el Vector de Dispersión VD%() a partir de DN$(), utilizando una función heurística simple. Esto posibilitará una rápida búsqueda de identificadores dentro del vocabulario de Pathfinder.

- **Rastreador()** Esta es una de las rutinas principales del proceso de compilación. Su función será de revisar, carácter por carácter, cada una de las líneas del programa fuente almacenado en FU$() y separar cada uno de los tokens hallados. **Rastreador()** asume que los tokens están delimitados por el símbolo especial "espacio" (código ASCII 32). También, y por simpleza, asumirá que los tokens estarán formados por dos caracteres, con excepción de los tokens "=" y ".", los cuales al estar formados por un solo carácter recibirán un tratamiento especial para ajustarnos a la lógica de dos caracteres. Esta rutina dejará en el vector TX%(1) y TX%(2) los valores ASCII de los caracteres que conforman el token hallado. En la variable BE dejará el valor cardinal del token hallado, de acuerdo al

TESIS CON
FALLA DE ORIGEN

vocabulario del robot. En caso de que Rastreador() localice un nuevo identificador, dejará en BE el valor 28.

- **MÁQUINAQ()** Verifica que la lista de tokens proporcionada por Rastreador() tenga una secuencia aceptable. Para ello, invocará a Rastreador() y consultará la Matriz de Precedencia TABLAQ() para verificar que éste ultimo token pueda seguir al token anteriormente adquirido. La variable TY guardará el resultado de la evaluación, de acuerdo a la siguiente tabla:

TY=0	La secuencia de tokens es inválida y ello causa un error.
TY=1	La secuencia de tokens es válida; es posible continuar el proceso de adquisición de nuevos tokens.
TY=2	La secuencia de tokens adquiridos contiene información suficiente para intentar un proceso de reducción de símbolos para identificar la estructura gramatical que le corresponde.

Fig. II.4.1 Valores de TY

Para realizar la tarea de reducción, la aplicación empleará la rutina **Automata()**.

- **Compilar()** Esta rutina será el bucle principal del proceso de compilación. Se encargará de invocar a las rutinas restantes (MaquinaQ, Autómata, Rastreador, etc.). Su función es comandar el proceso de compilación y generación de código ejecutable. Utilizará SL%() y SE%() como buffers para generar parcialidades de código ejecutable, vaciándolo finalmente en PR%(), que es el vector que funge como la RAM de Pathfinder.

II.6.2. Rutinas del proceso de Ejecución

Las principales rutinas que intervendrán en el proceso de ejecución del código generado por el proceso de compilación son las siguientes:

- **ExecPRG()** Esta rutina se encargará de redibujar "el planeta" e invocar el ciclo central de ejecución de código. Este ciclo central realizará sucesivos *fetchs* a la RAM de Pathfinder (PR%) para extraer de ella instrucciones ejecutables. Por cada byte extraído, lo cotejará con una lista de códigos ejecutables para decidir la acción a realizar. El ciclo central finalizará una vez que se encuentre un byte cero, el cual es señal de fin de programa. Es posible conceptuar esta rutina como el "procesador" central de Pathfinder.

- **Avanza()** La función Avanza dibujará una nueva posición de Pathfinder dentro de "el planeta". Al realizar el desplazamiento de Pathfinder dentro de "el planeta", actualizará los registros sensoriales del Robot; de esta manera, la función Avanza() devolverá un valor "verdadero" si es que el avance del robot fue exitoso; devolverá un valor de "falso" si es que el avance del robot causó una catástrofe (es decir, que Pathfinder se estrelló contra un roca). Hay que recordar que Avanzar significa que Pathfinder se desplace un bloque en la dirección que señala su registro KDir; es posible que Pathfinder no pueda avanzar debido a que hay un objeto (roca) que le impida el movimiento. En este caso se produce una catástrofe. La función Avanza invoca la función BitBlt() del API de Windows para dibujar la nueva posición de Pathfinder dentro del tablero.

- **Vuelta()** Esta rutina tiene como objetivo lograr que Pathfinder dé un giro ya sea a la izquierda o a la derecha. Hay que recordar que una vuelta de Pathfinder es un giro de 90 grados en la dirección indicada. De esta manera, la siguiente tabla muestra las posibles combinaciones de giros que Pathfinder puede dar:

Dirección de Pathfinder	Vuelta Derecha	Vuelta Izquierda
KNorte	• KEste	• KOeste
KSur	• KOeste	• KEste
KEste	• KSur	• KNorte
KOeste	• KNorte	• KSur

Fig. II.6.2 Tabla de direcciones.

La forma de lograr que Pathfinder dé un giro en alguna de las dos direcciones es modificando el contenido del registro PATHFINDERDIR, para que contenga el valor de la nueva dirección, y dibujar a Pathfinder con el nuevo aspecto que tiene, es decir, dibujar la imagen de Pathfinder que corresponde a su nueva orientación. Para lograr esto, se emplea la función BitBlt() del API de Windows.

II.6.2.1 Detección de errores

Una parte importante del proceso de compilación es la **detección de errores** y la recuperación del compilador ante un error. Es necesario recordar que un compilador no es un programa detector de errores, y que en consecuencia, la localización de un error así como la recuperación misma del compilador ante un error son servicios de valor agregado de un proceso de compilación. En este sentido, y estrictamente hablando, un compilador podría cumplir perfectamente su tarea sin contar con la capacidad de identificar errores e interrumpiendo el proceso de compilación mismo ante la

TESIS CON
FALLA DE ORIGEN

presencia de un error. Sin embargo, sería frustrante para cualquier persona que elabore un programa, que el compilador se detenga indicando solamente "hubo un error", sin indicar cual fue ni dónde estuvo. Por ello el tema de la detección e identificación de errores, así como la recuperación del compilador ante los mismos resulta crucial para cualquier compilador.

- **TipoError()** Determina la clase de error ocurrida durante el proceso de compilación. Para ello, se basará en los valores de BE y RQ arrojados por la rutina MAQUINAQ() y con ese contexto podrá determinar que clase de error ocurrió. Los errores que el compilador es capaz de reconocer son los siguientes:

Texto del Error	Descripción del error
Falta condicional	La estructura formada carece de un condicional cuando así lo requería. Aplica para estructuras MI, RP y SI.
Estructura Ilegal	La estructura formada es ilegal. Ocurre cuando hay dos condicionales, un SI sin DM o LG, un MI sin condicional o sin LG, etc.
Esta es palabra reservada	Emplea una de las palabras reservadas del lenguaje de Pathfinder como un nuevo identificador.
Nombre repetido	Emplear un identificador ya utilizado anteriormente como un nuevo identificador.
No existe este nombre	Invoca un nombre de tarea que no ha sido previamente definido
Falta un '=' en tu Tarea	Falta un signo "=" en la estructura de definición de una nueva tarea.
No hay definición	La estructura no forma una definición correcta.
Te faltó el punto	Falta un punto como terminador de instrucción.
Error indefinido	Un error cuyo contexto impide identificar el tipo de error.

- **CompError()** Esta rutina tiene como objetivo el detener el proceso de compilación cuando se encuentra un entorno que prefigure un error. No determinará el tipo de error incurrido; para ello, invocará a la rutina **TipoError()**. Una vez que regresa del proceso de identificación de un error, se encargará de recuperar al compilador de esta situación, a fin de que pueda continuar adelante con el proceso de compilación.

CAPÍTULO III

Plataforma

de

Desarrollo

TESIS CON
FALLA DE ORIGEN

67-A

CAPÍTULO III. PLATAFORMA DE DESARROLLO

III.1. Visual Basic.

El objetivo principal de presentar los movimientos del robot de forma gráfica es el poder visualizar de una manera más clara y concisa el funcionamiento de cada uno de los programas que el usuario cree. De esta forma existirá una mejor comunicación entre el usuario y su forma de programar al robot. Cada persona va a tener cualquier cantidad de programas para resolver un mismo problema.

Debido a que el lenguaje simbólico y el compilador necesitan ser programados en un ambiente gráfico es necesario un lenguaje de alto nivel, el cual la gran mayoría de los usuarios expertos conocen y que resulta ser un lenguaje común entre los estudiantes, así como un producto sencillo en el cual se pueda programar sin muchas complicaciones. Además de ser un producto que se consigue fácilmente en el mercado y que se usa en muchas empresas. Se ha determinado el uso, como mejor alternativa para dar solución a nuestro lenguaje y compilador a Visual Basic.

Visual Basic es hoy el lenguaje de programación más popular del mundo. Es el sueño del programador de aplicaciones, ya que es un producto con una interfaz gráfica de usuario para crear aplicaciones para Windows basado en el lenguaje Basic, Qbasic o QuickBasic, y en la programación orientada a objetos.

La palabra "Visual" hace referencia al método que se utiliza para crear la interfaz gráfica de usuario. En lugar de escribir numerosas líneas de código para implementar una interfaz, se utiliza el ratón para arrastrar y colocar los objetos prefabricados al lugar deseado dentro de un formulario.

La palabra "Basic" hace referencia al lenguaje BASIC (Beginners All Purpose Symbolic Instruction Code), un leguaje utilizado por más programadores que ningún otro lenguaje en la historia de la informática. Visual Basic ha evolucionado a partir del lenguaje BASIC original y ahora contiene centenares de instrucciones, funciones y palabras clave, muchas de las cuales están directamente relacionadas con la interfaz gráfica de Windows.

Este lenguaje es utilizado también por Microsof Excel, Microsoft Access y muchas otras aplicaciones en Windows. El sistema de programación de Visual Basic Script para programar en Internet, también es un subconjunto del lenguaje Visual Basic.

III.1.1. Ventajas principales del uso de VB.

La ventaja principal de este lenguaje de programación es su sencillez para programar aplicaciones de cierta complejidad para Windows. Visual Basic es un entorno de desarrollo diseñado para la creación de aplicaciones para las plataformas de trabajo Microsoft Windows 95, 98 y NT. Este lenguaje, conjunta las posibilidades de un lenguaje de alto nivel con las herramientas de diseño

TESIS CON
FALLA DE ORIGEN

gráfico, y se decidió usar VB porque contiene herramientas que nos ayudaron a realizar el proyecto de tesis de una manera más fácil.

Entre las principales ventajas que se tienen con este producto son:

- Visual Basic soporta la abstracción, la encapsulación, el polimorfismo y la reutilización del código.
- Los objetos de VB tienen propiedades, métodos y eventos. Las propiedades son los datos que describen un objeto. Los eventos son hechos que pueden ocurrir sobre un objeto. Un método agrupa el código que se ejecuta en respuesta a un evento
- Al conjunto de propiedades y métodos se le llama interfaz. Además de su interfaz predeterminada, los objetos pueden implementar interfaces adicionales para proporcionar polimorfismo. El polimorfismo le permite manipular muchos tipos diferentes de objetos sin preocuparse de su tipo.
- Las interfaces múltiples son una característica del modelo de objetos componente y permiten que los programas evolucionen con el tiempo, agregando nueva funcionalidad sin afectar al código existente.
- Facilita la creación de aplicaciones para el sistema operativo Microsoft Windows. Antes de Visual Basic, crear una aplicación para este ambiente habría exigido incontables horas de programación, usando las herramientas que en ese momento existían.
- Es posible crear una interfaz visual que se parece y se siente como las que se observan en las aplicaciones para Windows. Al escribir un código que haga que la interfaz visual responda a los usuarios correctamente. Crear un software amistoso que interactúe de forma dinámica con los usuarios.
- Es una herramienta que puede usarse para hacer que la computadora haga lo que el desarrollador desee.
- Los controles comunes de Windows se actualizan para incluir nuevos controles y funciones y usar nombres de archivo nuevos.
- Pueden crearse de la manera más rápida y más fácil, aplicaciones para Microsoft Windows. Visual Basic proporciona un juego completo de herramientas para simplificar el desarrollo de aplicaciones rápidamente.
- La parte "Visual" se refiere al método usado para crear interfaces gráficas para usuario (GUI) en lugar de escribir numerosas líneas de codificación, describir la apariencia y situación de elementos de la interfaz, simplemente agregar objetos del pre-constructor en lugar de pantallas.

TESIS CON
FALLA DE ORIGEN

- El "Basic" parte referente al lenguaje BASIC ("Beginners All-Purpose Symbolic Instruction Code"), el lenguaje usado por muchos programadores más que cualquier otro lenguaje en la historia de la computación. Visual Basic está desenvuelto desde el lenguaje original BASIC y ahora contiene varias declaraciones, funciones, y palabras clave, muchas de las cuales se relacionan directamente al Windows GUI.
- El lenguaje de programación Visual Basic no es lo único de Visual Basic. El sistema de programación Visual Basic, la edición de las aplicaciones, incluido en Microsoft Access, Microsoft Excel, y muchas otras aplicaciones de Windows usan el mismo lenguaje.
- Si la meta es crear una pequeña utilería para el usuario mismo o grupo de trabajo, o aplicaciones distribuidas utilizando Internet, Visual Basic tiene las herramientas que el usuario necesita.
- Funciones de acceso a los datos que permitan crear una base de datos, aplicaciones front-end que incluye a SQL server.
- La tecnología ActiveX permite el uso de funciones que se encuentran en otras aplicaciones, como por ejemplo Microsoft Word como procesador de textos, Microsoft Excel como hoja de cálculo, y otras aplicaciones de Windows.

Existen tres ediciones de Visual Basic: la estándar o de aprendizaje, la profesional y la empresarial. La edición estándar permite crear aplicaciones robustas para Microsoft Windows 95 y Windows NT; incluye todos los controles intrínsecos además de los controles rejilla, cuadro de diálogo estándar y los controles enlazados a datos.

La edición profesional incluye todas las características de la edición estándar, así como controles activos (ActiveX) adicionales, incluidos controles para Internet y el generador de informes Crystal Reports. La edición empresarial incluye todas las características de la edición profesional, así como el administrador de bases de datos, el sistema de control de versiones orientado a proyectos Microsoft Visual SourceSafe, etc.

En nuestro caso usamos la edición profesional debido a que contiene los controles ActiveX necesarios para soportar el ambiente gráfico del proyecto.

TESIS CON
FALLA DE ORIGEN

III.1.2. Requerimientos de Hardware y Software para la Instalación de Visual Basic

Para instalar VB se deben de contar con los requisitos mínimos que se indican a continuación.

- Microprocesador 80486 ó superior.
- Un disco duro con un espacio mínimo disponible de 50 Mb para poder realizar una instalación completa.
- Una unidad de CD-ROM
- Un ratón
- Una placa de vídeo soportada por Windows
- 16 Mb de memoria ó más
- Microsoft Windows 95 ó posterior, ó Windows NT 3.51 ó posterior.
- Pentium® 90MHz o microprocesador posterior.
- VGA 640x480 o monitor de alta resolución soportado por Microsoft Windows.
- 24MB RAM para Windows 95, 32MB para Windows NT, XP, W2K
- Microsoft Windows NT 3,51 o posterior o Microsoft Windows 95 o posterior.
- Requisitos especiales para Discos duros:
- Edición profesional: instalación típica 48MB, instalación completa 80MB.

III.2. Rutinas Gráficas Utilizadas

Para poder presentar a Pathfinder realizando las tareas definidas en los programas que procesa, es necesario contar con ambiente gráfico versátil y con un considerable conjunto de rutinas gráficas que permitan observar a Pathfinder desplazarse por la pantalla de la computadora. Se ha optado por el ambiente gráfico del Sistema Operativo Windows, debido a las ventajas que éste ofrece al contar con un API (Application Programming Interface) accesible a la comunidad de desarrolladores de software. Combinando las facilidades de Visual Basic con el API de Windows será posible crear una aplicación gráfica funcional y atractiva.

III.2.1. Elementos Gráficos.

Si bien el planeta consiste en una región de dimensiones infinitas formada por renglones y columnas, la implementación del planeta será una imagen (bitmap) de dimensiones finitas. Se definen a cada una de las intersecciones renglón-columna del planeta como "celdas", las cuales se tratan de un bitmap de 15 pixeles de ancho por 13 pixeles de alto. Planeta es, entonces, un tablero formado por 40 celdas de ancho y 30 celdas de alto. Estas dimensiones permiten contar con un tablero lo suficientemente grande (1,200 celdas) como para lograr configuraciones de rocas y trompos versátiles e interesantes. Un ejemplo de este tablero se muestra en la Figura III.2.1:

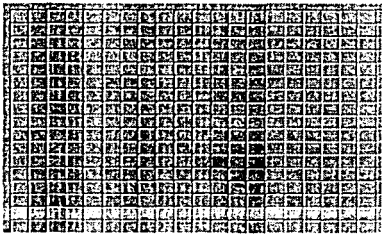


Fig. III.2.1 Una muestra del Planeta

Para representar a Pathfinder y a los objetos que residen dentro del planeta (trompos y rocas) serán necesarios diferentes bitmaps: cada uno de ellos deberá ser de la misma dimensión de una celda del Planeta (15x13 pixeles) a fin de poderlos ubicar en alguna de las celdas y dar así la impresión de que "hay un objeto" en dicha celda. Será necesario contar con cuatro versiones diferentes de de la imagen de Pathfinder: mirando al norte, sur, este y oeste. Además, será necesario contar con una imagen que ilustre a Pathfinder cuando haya chocado contra una roca o contra uno de los bordes del planeta. También será necesario contar con un bitmap que represente

TESIS CON
FALLA DE ORIGEN

una roca y otro más que represente una celda vacía. Finalmente, será importante contar con un bitmap que represente un trompo. Los trompos que sean colocados por el usuario serán trompos rojos y los que sean colocados por Pathfinder durante el desenvolvimiento de sus tareas serán de color verde. En la siguiente figura aparece un mosaico con estas imágenes:

El movimiento de Pathfinder se logrará mediante el dibujado del robot virtual en las sucesivas posiciones que éste adopte durante el desenvolvimiento de sus tareas.

Ahora bien, un elemento importante que será incluido es la representación gráfica del "radar" del robot. Este radar consiste en la representación gráfica de aquello que "ve" el robot con sus sensores: Pathfinder tiene un sensor frontal y dos laterales. El radar de Pathfinder mostrará lo que Pathfinder "ve" con estos tres sensores en todo momento. En la Figura III.2.2 se muestra el "radar" de Pathfinder:

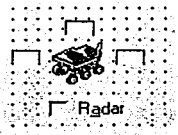


Fig. III.2.2 El Radar de Pathfinder.

La forma en que será lograda la animación de estos objetos gráficos es presentada en el siguiente apartado.

III.2.2. Rutinas gráficas del API de Windows.

Windows brinda un API de rutinas con las cuales un desarrollador de software puede hacer uso de las facilidades gráficas de Windows con poco esfuerzo. Una de las ventajas de emplear el API de Windows es que se trata de rutinas previamente compiladas y que residen en archivos DLL (Dynamic Linking Libraries). Estas rutinas pueden ser invocadas mediante una sintaxis de uso muy cómoda y conveniente.

Para hacer uso de una rutina del API de Windows empleando Visual Basic, uno debe hacer la declaración de uso de la rutina desde un módulo de uso global de la aplicación (archivo con extensión **.bas**). Los diferentes tipos y clases de declaración que se pueden utilizar son obtenidas

mediante el API Text Viewer de Visual Basic, el cual es una herramienta para obtener las declaraciones de rutinas del API de Windows..

Una rutina que será de mucha utilidad es la función **BitBlt()**. Con la ayuda de esta función del API de Windows, uno puede copiar un bitmap encima de otro bitmap. La declaración de uso del **BitBlt()** es la siguiente:

```
Declare Function BitBlt Lib "gdi32" (ByVal hDestDC As Long,
ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal
nHeight As Long, ByVal hSrcDC As Long, ByVal xSrc As Long,
ByVal ySrc As Long, ByVal dwRop As Long) As Long
```

En el texto de la declaración es posible notar que **BitBlt()** reside en el archivo **GDI32.DLL**. Los parámetros que recibe **BitBlt()** son los siguientes:

hDestDC.- Apuntador al objeto gráfico destino que es el receptor de las operaciones de dibujo que realizará la función BitBlt().

x, y.- Coordenadas xy en el objeto gráfico *hDestDC* a partir del cual será realizada la operación de dibujo.

nWidth, nHeight.- Ancho y alto del objeto gráfico que será dibujado en el interior de *hDestDC*.

hSrcDC.- Objeto gráfico fuente que será dibujado encima del objeto gráfico destino (hDestDC).

xSrc, ySrc.- Coordenadas xy a partir de las cuales será "leído" el objeto gráfico fuente. Será leído un rectángulo de x pixeles de ancho por y pixeles de alto.

dwRop.- Tipo de operación gráfica a efectuar. Para este parámetro son posibles cualquiera de los siguientes valores:

- `SRCCOPY = &HCC0020` .- En caso de utilizar este valor, la operación a realizar es un copiado de bitmaps. Este es el valor que será empleado en la aplicación.
- `SRCAND = &H8800C6` .- Este valor indica una operación AND de los bits del bitmap destino sobre los bits del bitmap origen.
- `SRCINVERT = &H660046`.- El uso de este valor causa el realizar una inversión de los bits que componen el bitmap origen.
- `BLACKNESS = &H42&` .- Este valor indica que deben ser convertidos a negro los bits del bitmap destino.

TESIS CON
FALLA DE ORIGEN

- FLOODFILLSURFACE = 1 .- Este valor indica que hay que reproducir el bitmap origen sobre el bitmap destino, en forma de mosaico.

Un ejemplo de uso de esta función es el siguiente:

```
RF = BitBlt(Principal.Cuadro.hdc, OldKX, OldKY, PicAncho,
PicAlto, Principal.Nulo.hdc, 0, 0, SRCCOPY)
```

Para lograr calcular la posición xy en la cual será dibujado un bitmap específico, será empleada una función que convierta coordenadas renglón-columna a valores x-y y viceversa. La función que realliza esta conversión es la siguiente:

```
Function PathfinderXY(ByVal XY$, ByVal RC) As Long
  '*** Una imagen del cuadro mide 15 pixeles de ancho por 13 de alto.
  '*** En ambos casos, el offset X o Y es de 8 pixeles.
  If UCase(XY$) = "Y" Then
    '*** Devuelve la posición Y en Cuadro, dado un valor de renglon en Planeta
    PathfinderXY = ((RC - 1) * 13 + 8)
  ElseIf UCase(XY$) = "X" Then
    '*** Devuelve la posición X en Cuadro, dado un valor de columna en Planeta
    PathfinderXY = ((RC - 1) * 15 + 8)
  End If
End Function
```

Para calcular la posición Renglón-Columna correspondiente a una posición XY de Pathfinder, será empleada una función que tome como parámetros de entrada una posición X ó Y y devolverá un valor de Renglón o Columna respectivos. Esta función es la que se muestra a continuación:

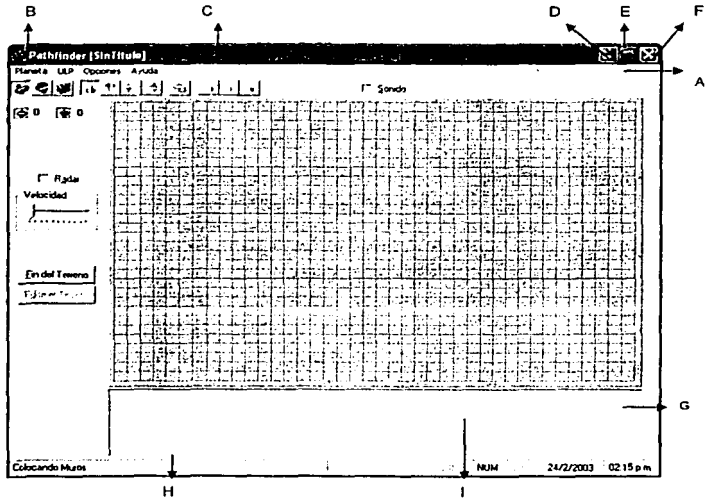
```
Function PathfinderRowCol(ByVal RowCol$, ByVal PosiCuadro) As Long
  '*** Una imagen del cuadro mide 15 pixeles de ancho por 13 de alto.
  '*** En ambos casos, el offset X o Y es de 8 pixeles.
  If UCase(RowCol$) = "R" Then
    '*** Devuelve el renglón en que Pathfinder esta parado, dada Y en Cuadro.
    PathfinderRowCol = ((PosiCuadro - 8) / 13) + 1
  ElseIf UCase(RowCol$) = "C" Then
    '*** Devuelve la columna en que Pathfinder esta parado, dada X en Cuadro.
    PathfinderRowCol = ((PosiCuadro - 8) / 15) + 1
  End If
End Function
```

Estas dos funciones realizan conjuntamente el mapeo de posiciones XY gráficas, a posiciones Renglón-Columna lógicas y viceversa. Con estas dos funciones es posible mapear la ubicación física de Pathfinder hacia su posición lógica y viceversa. Estas dos funciones son invocadas frecuentemente debido a que es mantenida tanto la posición física del robot en el área gráfica a desplegar, como la posición lógica del robot dentro de la matriz KR() que hemos denominado Planeta.

II.2.3 INTERFAZ GRAFICA DE LA APLICACION

Una de las grandes ventajas de trabajar con Windows es que todas las ventanas se comportan de la misma forma y todas las aplicaciones utilizan los mismos métodos básicos (menús desplegables, botones) para introducir órdenes.

Por lo que una ventana típica de Pathfinder contiene las siguientes partes:



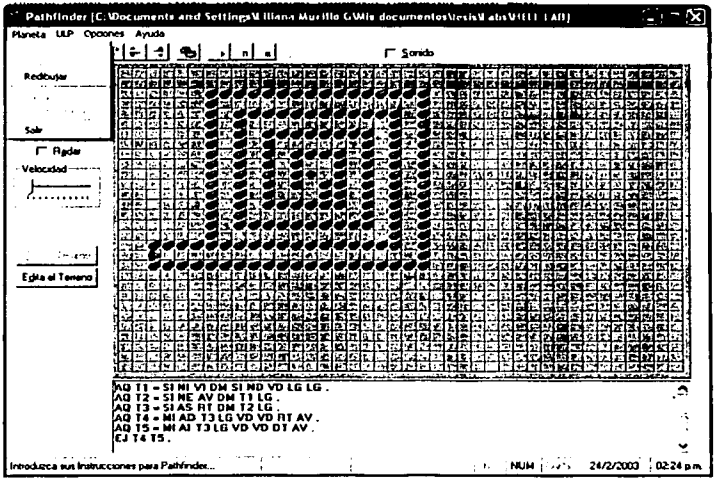
TESIS CON
FALLA DE ORIGEN

CAPÍTULO III. PLATAFORMA DE DESARROLLO

A. BARRA DE MENUS

Visualiza el conjunto de menús disponibles para esa aplicación. Cuando alguno de los menús se activa haciendo clic con el ratón sobre su título se visualiza el conjunto de órdenes que lo forman.

A continuación se describen las aplicaciones que contiene cada menú y su aplicación correspondiente.



MENÚ PLANETA:

Este menú se refiere a la parte del laberinto y exclusivamente a este, es decir todo lo que tenga que ver con la construcción de laberintos y trompos.

Nuevo : Esta opción lo que hace es que crea un nuevo y limpio mundo del planeta, preguntando primero si se quiere salvar el anterior.

```
Private Sub NewLab_Click()
  Beep
```

```
    If MsgBox("¿ Desea salvar el Planeta actual ?", 308, "Nuevo Planeta") = 6 Then
       LabSaveAs_Click
    End If
    MousePointer = 11
    CurrentTool = 0        ' Herramienta en Uso: Muros
    'Tools.ScaleMode = 3   ' Escalamiento en Pixeles
    KTool.Buttons(5).Value = tbrPressed    ' Se asume Orientación al Norte
    KTool.Buttons(6).Value = tbrUnpressed
    KTool.Buttons(7).Value = tbrUnpressed
    KTool.Buttons(8).Value = tbrUnpressed
    PDir = -1              ' Dirección de Pathfinder=Norte
    Figurita.Picture = PicClip1.GraphicCell(2)
    PXPathfinder = -1      ' Posición Inicial X de Pathfinder=Nula
    PYPathfinder = -1      ' Posición Inicial Y de Pathfinder=Nula
    NTRojosN = 0       ' Cantidad de Trompos Rojos
    NTVerdesN = 0      ' Cantidad de Trompos Verdes
    For I = 1 To MaxRows              ' de MaxRows renglones
       For J = 1 To MaxCols           ' por MaxCols columnas
          Planeta(I, J) = 0     ' Inicialmente con NULO
       Next
    Next
    MousePointer = 0
    RedrawLab_Click
    NTRojosN = 0
    NTVerdesN = 0
    NTRojos.Caption = Format$(NTRojosN)
    NTVerdes.Caption = Format$(NTVerdesN)
    NombrePlaneta$ = ""
    Principal.Caption = "Pathfinder [SinTitulo]"
End Sub
```

Leer: Despliega una ventana en donde se puede buscar un archivo guardado anteriormente con extensión *.lab en donde se guardo un terreno ya construido.

```
Private Sub LeerLab_Click()
  Dialogos.DefaultExt = "LAB"
  Dialogos.DialogTitle = "Planeta a ser leído"
  Dialogos.FileName = ""
  Dialogos.Filter = "Planetas (*.lab)|*.lab|Todos los archivos (*.*)|*.*"
  Dialogos.ShowOpen
  If Dialogos.FileName <> "" Then
    NombrePlaneta$ = Dialogos.FileName
    Identifica% = FreeFile
    MousePointer = 11
    Open NombrePlaneta$ For Input As Identifica%
    NTRojosN = 0
```

```
  PXPathfinder = -1
  PYPathfinder = -1
  PDir = 0
  For I = 1 To MaxRows              ' de MaxRows renglones
    For J = 1 To MaxCols            ' por MaxCols columnas
      Input #Identifica%, Planeta(I, J)
      If Planeta(I, J) = 2 Then     ' Trompo
        NTRojosN = NTRojosN + 1
      ElseIf Planeta(I, J) < 0 Then
        PDir = Planeta(I, J)
        PosiX = PathfinderXY("X", J)
        PosiY = PathfinderXY("Y", I)
        IColumna = J
        IRenglon = I
        PXPathfinder = PosiX
        PYPathfinder = PosiY
        KREN = IRenglon
        KCOL = IColumna
'PosiX = ((x - 8) \ 15) * 15 + 8
'PosiY = ((y - 8) \ 13) * 13 + 8
'IColumna = (x - 8) \ 15 + 1
'IRenglon = (y - 8) \ 13 + 1

        'PXPathfinder = (J * 15 + 8)
        'PYPathfinder = (I * 13 + 8)
      End If
    Next
  Next
  Principal.NTRojos.Caption = Format$(NTRojosN)
  Close Identifica%
  Principal.Caption = "Pathfinder [" + NombrePlaneta$ + "]"
  RedrawLab_Click
  MousePointer = 0
 End If
End Sub
```

Redibujar: Permite al usuario crear sobre un terreno ya establecido, nuevos rocas o poner más trompos o quitarlos también. Es decir nos permite modificar un entorno ya guardado anteriormente.

```
Private Sub RedrawLab_Click()
  Cuadro_Paint
End Sub
```

TESIS CON
FALLA DE ORIGEN

ESTA TESIS NO SALE
DE LA BIBLIOTECA

Salvar: Nos permite guardar el terreno.

```
Private Sub SaveLab_Click()
   If Len(NombrePlaneta$) > 0 Then
      PP$ = Principal.Status.Panels(1).Text
      Avisa ("Salvando " + NombrePlaneta$)
      Identifica% = FreeFile
      MousePointer = 11
      Open NombrePlaneta$ For Output As Identifica%
      For I = 1 To MaxRows              ' de MaxRows renglones
         For J = 1 To MaxCols           ' por MaxCols columnas
            Print #Identifica%, Planeta(I, J);
         Next
         Write #Identifica%,
      Next
      Close Identifica%
      MousePointer = 0
      Avisa (PP$)
   Else
      LabSaveAs_Click
   End If
End Sub
```

Salvar Como: Permite guardar el terreno con otro nombre.

```
Private Sub LabSaveAs_Click()
   Dialogos.DialogTitle = "Salvar laberinto"
   Dialogos.Filter = "Planetas de Pathfinder (*.lab)|*.lab|Todos los Archivos (*.*)|*.*"
   Dialogos.DefaultExt = "LAB"
   Dialogos.Flags = &H2& Or &H800& Or &H200000
   Dialogos.ShowSave
   If Len(Dialogos.FileName) > 0 Then
      NombrePlaneta$ = Dialogos.FileName
      Avisa ("Salvando " + NombrePlaneta$)
      Identifica% = FreeFile
      MousePointer = 11
      Open NombrePlaneta$ For Output As Identifica%
      For I = 1 To MaxRows              ' de MaxRows renglones
         For J = 1 To MaxCols           ' por MaxCols columnas
            Print #Identifica%, Planeta(I, J);
         Next
         Write #Identifica%,
      Next
      Close Identifica%
```

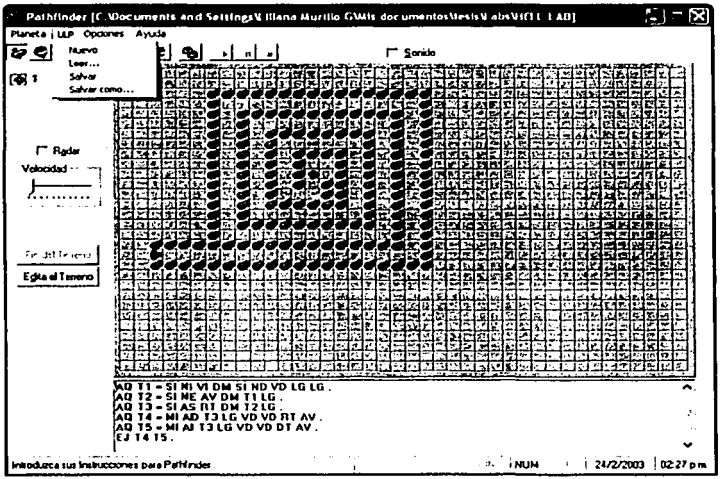
```
        MousePointer = 0
        Avisa ("Planeta Salvado")
        Principal.Caption = "Pathfinder [" + NombrePlaneta$ + "]"
    End If
End Sub
```

Salir: Sale de la aplicación.

```
Private Sub Salir_Click()
    Unload Principal
End Sub
```

MENÚ ULP

Cabe destacar que para hacer uso de este menú se debe de haber dado fin al laberinto, de otra forma no se podrá acceder a este menú.



TESIS CON
FALLA DE ORIGEN

Nuevo: Limpia el área de trabajo, para hacer una nueva programación de Pathfinder.

```
Private Sub NewProgram_Click()
   Beep
   If MsgBox("¿ Desea salvar el programa actual ?", vbQuestion + vbYesNo, "Nuevo Programa") =
vbYes Then
      PrgSaveAs_Click
   End If
   MousePointer = 11
   FWin.Text = ""
End Sub
```

Leer: Despliega una ventana de búsqueda de archivos en donde se puede escoger un programa ya hecho con anterioridad. Los programas que serán aceptados son los que contienen *.prg como extensión.

```
Private Sub LeePrg_Click()
   Dialogos.DefaultExt = "PRG"
   Dialogos.DialogTitle = "Elija un programa para Pathfinder"
   Dialogos.FileName = ""
   Dialogos.Filter = "Programas para Pathfinder (*.prg)|*.prg|Todos los archivos (*.*)|*.*"
   Dialogos.ShowOpen
   If Dialogos.FileName <> "" Then
     Principal.FWin.Text = ""
     PrgName$ = Dialogos.FileName
     Identifica% = FreeFile
     MousePointer = 11
     Open PrgName$ For Input As Identifica%
     Do While Not EOF(Identifica%)
       Hola$ = ""
       Input #Identifica%, Hola$
       If (Hola$ <> "") And (Hola$ <> "Fl .") Then
         Principal.FWin.Text = Principal.FWin.Text + Hola$ + Chr$(13) + Chr$(10)
       End If
     Loop
     Close Identifica%
     MousePointer = 0
   End If
End Sub
```

Salvar: Guarda el programa.

```
Private Sub SaveProgram_Click()
   If Len(PrgName$) > 0 Then
```


TESIS CON
FALLA DE ORIGEN

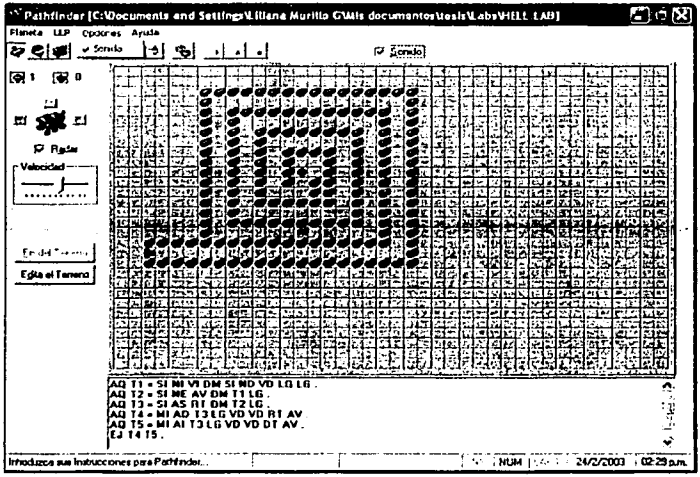
```
    PP$ = Principal.Status.Panels(1).Text
    Avisa ("Salvando " + PrgName$)
    Identifica% = FreeFile
    MousePointer = 11
    Open PrgName$ For Output As Identifica%
    Print #Identifica%, FWin.Text
    Close Identifica%
    MousePointer = 0
    Avisa (PP$)
  Else
    PrgSaveAs_Click
  End If
End Sub
```

Salvar como: Guarda el programa con otro nombre, en otra ubicación diferente.

```
Private Sub PrgSaveAs_Click()
  Dialogos.DialogTitle = "Salvar Programa"
  Dialogos.Filter = "Programas para Pathfinder (*.prg)|*.prg|Todos los Archivos (*.*)|*.*"
  Dialogos.DefaultExt = "PRG"
  Dialogos.Flags = &H12& Or &H800& Or &H200000
  Dialogos.ShowSave
  If Len(Dialogos.FileName) > 0 Then
    PrgName$ = Dialogos.FileName
    Avisa ("Salvando " + PrgName$)
    Identifica% = FreeFile
    MousePointer = 11
    Open PrgName$ For Output As Identifica%
    Print #Identifica%, Principal.FWin.Text
    Close Identifica%
    MousePointer = 0
    Avisa ("Programa Salvado")
  End If
  'SalvaPrg.Show 1
End Sub
```


TESIS CON
FALLA DE ORIGEN

MENÚ OPCIONES



Sonido: esta opción nos permite habilitar ó deshabilitar sonido en la aplicación.

```
Private Sub Sonido_Click()
   Sonido.CHECKED = Not Sonido.CHECKED
End Sub
```

B. ÍCONO DE LA APLICACIÓN Y MENÚ DE CONTROL

El menú de control proporciona órdenes para: restaurar su tamaño, mover, minimizar y cerrar la ventana.

C. BARRA DE TÍTULO.

Contiene el nombre de la ventana y del documento. Para mover la ventana a otro lugar, se tiene que apuntar con el ratón a esta barra, se hace click utilizando el botón izquierdo del ratón y

manteniendo pulsado el botón, se arrastra a la dirección deseada. Un doble click maximiza o retorna a tamaño normal la ventana, dependiendo del estado actual.

D. BOTON PARA MINIMIZAR LA VENTANA

Cuando se pulsa este botón la ventana se reduce a un icono.

E. BOTÓN PARA MAXIMIZAR LA VENTANA

Cuando se pulsa este botón, la ventana se amplia al máximo y el botón se transforma. Si éste se pulsa, la ventana se reduce al tamaño anterior

F. BOTÓN PARA CERRAR LA VENTANA.

Cuando se pulsa este botón, se cierra la ventana y la aplicación se la ventana es la principal.

G. BARRA DE DESPLAZAMIENTO VERTICAL.

Cuando la información no entra verticalmente en una ventana, el programa añade una barra de desplazamiento vertical a la derecha de la ventana.

H. MARCO DE LA VENTANA.

Permite modificar el tamaño de la ventana. Para cambiar el tamaño se debe de apuntar con el mouse a la esquina o aun lado del marco, cuando el puntero cambie a una flecha doble, con el botón izquierdo del mouse pulsado se debe de arrastrar en el sentido adecuado para conseguir el tamaño deseado.

I. ÁREA DE TRABAJO.

Es la parte de la ventana en la que el usuario coloca el texto de programación de Pathfinder.

BOTONES

I.1. FIN DEL TERRENO

Lo que hace esta rutina es que guarda el laberinto actual y ya no deja hacer modificaciones, y procede a salvarlo.

```
Private Sub Command1_Click()
  If PYPathfinder = -1 And PXPathfinder = -1 Then
     Beep
     If 6 = MsgBox("Aún no há colocado a Pathfinder. ¿Terminar de todas maneras?", 308, "Fin de
Laberinto") Then
```


TESIS CON
FALLA DE ORIGEN

```
        IsLabOK = False
        FWin.Enabled = False
        End
      End If
    Else
      Avisa ("Introduzca sus Instrucciones para Pathfinder...")
      IsLabOK = True
      FWin.Enabled = True
      Command1.Enabled = False
      EditLab.Enabled = True
      KTool.Buttons(10).Enabled = True    '*** Habilita Compilar
      KTool.Buttons(12).Enabled = True    '*** Habilita Ejecutar
      Cuadro.Enabled = False
' Activación de Menús
      NewProgram.Enabled = True
      LeePrg.Enabled = True
      SaveProgram.Enabled = True
      PrgSaveAs.Enabled = True
      NewLab.Enabled = False
      LeerLab.Enabled = False
      RedrawLab.Enabled = True
      SaveLab.Enabled = False
      LabSaveAs.Enabled = False
' Deshabilita las opciones de Dirección de Pathfinder
      KTool.Buttons(5).Enabled = False
      KTool.Buttons(6).Enabled = False
      KTool.Buttons(7).Enabled = False
      KTool.Buttons(8).Enabled = False
      For I = 1 To MaxRows
        For J = 1 To MaxCols
          RPathfinderia(I, J) = Pathfinderia(I, J)
        Next
      Next
    End If
End Sub
```

I.2. EDITA EL TERRENO

Con este botón lo que se hace es poder acceder nuevamente a un laberinto ya hecho y modificarlo.

```
Private Sub EditLab_Click()
    Avisa ("Editando el Laberinto...")
    FWin.Enabled = False
    EditLab.Enabled = False
    Command1.Enabled = True
    KTool.Buttons(10).Enabled = False    '*** Habilita Compilar
    KTool.Buttons(12).Enabled = False    '*** Habilita Ejecutar
' Activación de Menús
    NewProgram.Enabled = False
    LeePrg.Enabled = False
    SaveProgram.Enabled = False
    PrgSaveAs.Enabled = False
    NewLab.Enabled = True
    LeerLab.Enabled = True
```


TESIS CON
FALLA DE ORIGEN

```
    RedrawLab.Enabled = True
    SaveLab.Enabled = True
    LabSaveAs.Enabled = True
    Cuadro.Enabled = True
    RedrawLab_Click
    If CurrentTool = 2 Then     ' Estaba colocando Robot
      KTool.Buttons(5).Enabled = True
      KTool.Buttons(6).Enabled = True
      KTool.Buttons(7).Enabled = True
      KTool.Buttons(8).Enabled = True
      If KTool.Buttons(5).Value = tbrPressed Then
        Avisa ("Colocando a Pathfinder mirando al Norte")
        Figurita.Picture = PicClip1.GraphicCell(6)
      ElseIf KTool.Buttons(6).Value = tbrPressed Then
        Avisa ("Colocando a Pathfinder mirando al Sur")
        Figurita.Picture = PicClip1.GraphicCell(7)
      ElseIf KTool.Buttons(7).Value = tbrPressed Then
        Avisa ("Colocando a Pathfinder mirando al Este")
        Figurita.Picture = PicClip1.GraphicCell(8)
      ElseIf KTool.Buttons(8).Value = tbrPressed Then
        Avisa ("Colocando a Pathfinder mirando al Oeste")
        Figurita.Picture = PicClip1.GraphicCell(9)
      End If
    ElseIf CurrentTool = 0 Or CurrentTool = 1 Then  ' Colocando Muros o Trompos
      KTool.Buttons(5).Enabled = False
      KTool.Buttons(6).Enabled = False
      KTool.Buttons(7).Enabled = False
      KTool.Buttons(8).Enabled = False
      If CurrentTool = 0 Then
        Avisa ("Colocando Muros")
        Figurita.Picture = PicClip1.GraphicCell(2)
      ElseIf CurrentTool = 1 Then
        Avisa ("Colocando Trompos")
        Figurita.Picture = PicClip1.GraphicCell(0)
      End If
    End If
End Sub
```

I.3. COMPILAR

Este botón realiza la función de compilación del programa escrito, realiza la primera y segunda pasada del compilador, verifica sintaxis y errores ortográficos. En caso de que haya algún error mandará el mensaje correspondiente. Este botón llama a la función precompila. Que se encarga de realizar la compilación del programa

```
ElseIf (Button.Index = 10) Then     '*** Compilar
    Avisa ("Preparando la Compilacion")
    ReadProgram
    Precompila
    Avisa ("Compilando...")
    Compilar
    IsCompOk = (Not (ER > 0))
    KTool.Buttons(12).Enabled = IsCompOk
    Avisa ("Compilación Finalizada")
```

```
Sub Precompila()
   '*** Inicializa variables para el proceso de compilacion
   RS = 0: PA = 0: ID = 31: IA = 0: IC = 0: IX% = 0: JX% = 0: TEX = 0
   TE = 0: BE = 0: BX$ = "": FI% = 0: JF% = 0: RQ = 0: RA = 0: DX = 1
   ER = 0: BJ = 0: YU$ = "": IP = 1: RIP = 1
   '*** Carga el Diccionario de Nombres (DN$) donde esta el vocabulario de Pathfinder
   For I = 1 To 37
      T$(I) = DN$(I)
   Next
   '*** El resto del diccionario de nombres es blanqueado
   For I = 38 To 100: T$(I) = "": DN$(I) = "": Next
   DB = 37
   DLW = 37
   ID = 37
   '*** Limpio vectores de trabajo
   For I = 0 To 50: OG%(I) = 0: LN%(I) = 0: Next
   For I = 0 To 100: DL%(I) = 0: Next
   For I = 0 To 200: VD%(I) = 0: Next
   For I = 0 To 24: For II = 0 To 85: SE%(I, II) = 0: Next: Next
   For I = 0 To 30: SX%(I) = 0: Next
   For I = 0 To 24: SL%(I) = 0: Next
   For I = 0 To 100: ST(I) = 0: Next
   '*** Carga el Vector de Dispersion VD%() con la informacion del DN$()
   '*** La funcion del vector de dispersion es la de cargar tanto los IDs como los NIDs
   '*** para poderlos localizar con rapidez.
   For I = 1 To 37
      X1 = Asc(Mid$(DN$(I), 1, 1))
      X2 = Asc(Mid$(DN$(I), 2, 1))
      X3 = X1 * X2
      DD = X3 - Int(X3 / 200) * 200
      If VD%(DD) = 0 Then
         VD%(DD) = I
      Else
         J = VD%(DD)
         Do
            If DL%(J) = 0 Then
               DL%(J) = I
               Exit Do
            Else
               J = DL%(J)
            End If
         Loop Until (2 <> 3)   ' Loop Infinito
      End If
   Next
   EraseRAM
End Sub
```


TESIS CON
FALLA DE ORIGEN

I.4. EJECUTAR

Este botón tiene la función de ejecutar el programa después de que ha sido compilado, en el caso de que se encontraran errores de compilación este botón no se podrá pulsar.

```
Elself (Button.Index = 12) Then     '*** Ejecutar
   KTool.Buttons(13).Enabled = True      '*** Habilita boton Pausa
   KTool.Buttons(14).Enabled = True      '*** Habilita boton Stop
   Stat1 = Principal.Command1.Enabled    '*** Respalda status boton FinLab
   Stat2 = Principal.EditLab.Enabled     '*** Respalda status boton EditLab
   Stat3 = KTool.Buttons(10).Enabled     '*** Respalda status boton Compila
   Stat4 = KTool.Buttons(12).Enabled     '*** Respalda status boton Ejecutar
   Command1.Enabled = False
   EditLab.Enabled = False
   KTool.Buttons(10).Enabled = False
   KTool.Buttons(12).Enabled = False
   Detente = False
   Ejecuta
   RedrawLab_Click
   ExecPrg
   KTool.Buttons(13).Enabled = False
   KTool.Buttons(14).Enabled = False
   Command1.Enabled = Stat1
   EditLab.Enabled = Stat2
   KTool.Buttons(10).Enabled = Stat3
   KTool.Buttons(12).Enabled = Stat4
```

I.5. PAUSA

Detiene la acción de ejecutar y esta se puede volver a ejecutar pulsando reanuda.

```
Elself (Button.Index = 13) Then     '*** Pausar
   If KTool.Buttons(13).Tag = "Pausa" Then
     KTool.Buttons(13).Tag = "Reanuda"
     Avisa ("Pathfinder está pausado. Pulse 'Reanuda' para seguir")
   Elself KTool.Buttons(13).Tag = "Reanuda" Then
     KTool.Buttons(13).Tag = "Pausa"
     Avisa ("Pathfinder está trabajando")
   End If
```

I.6. ABORTA

Detiene todo el proceso de ejecutar y no se puede volver a reanudar, solo volviendo a compilar y ejecutar el programa se puede hacer esto.

```
 Elself (Button.Index = 14) Then     '*** Detener
   Detente = True
 End If
```


TESIS CON
FALLA DE ORIGEN

CAPÍTULO IV

Costo Estimado

del

Proyecto

CAPÍTULO IV: ESTIMACIÓN DEL COSTO DEL PROYECTO

IV.1 Introducción

Hoy día, la mayoría de las compañías demandan sistemas de información cada vez más confiables, es decir, que su realización se lleve a cabo de forma correcta conforme a: primero, estándares de calidad y segundo, que el desarrollo se realice en tiempo y costos establecidos.
La estimación de costos de un producto de programación es una de las tareas más difíciles y erráticas de la ingeniería de software; es difícil hacer estimaciones exactas durante la fase de planeación de un desarrollo debido a la gran cantidad de factores desconocidos en ese momento.
De acuerdo a esta situación se utiliza una serie de estimadores de costos de tal forma que se prepara un estudio preliminar durante la fase de planeación y se presenta en la revisión de la factibilidad del proyecto. La estimación mejorada se muestra en las revisiones de los requisitos de programación y la estimación final se presenta durante la revisión preliminar del diseño.
Cada estimación es un refinamiento obtenido como resultado de las actividades de trabajo desarrollado adicionalmente; algunas veces, varias opciones del producto, con sus respectivos costos, se exhiben en las revisiones; lo anterior permite al usuario final escoger una solución adecuada dentro de las posibles soluciones.
A continuación se presentan los principales factores que influyen en los costos de un producto de programación.

IV.2. Factores en el costo del Software

Como se mencionó anteriormente, existen muchos factores que influyen en el costo de un producto de programación. El efecto de estos factores es difícil de estimar y, por ende también lo es el costo del esfuerzo en el desarrollo o en el mantenimiento.

IV.2.1. Capacidad del Programador

La experiencia, conocimiento del tema así como la manera de conceptualizar las cosas hacen que los programadores tengan un desempeño variable en el desarrollo de un producto. Esta variación en la productividad de la programación es un factor significativo para la estimación de los costos. En proyectos muy grandes, las diferencias individuales tienden a compensarse, pero en proyectos de cinco programadores o menos la diferencia puede ser muy importante.

IV.2.2. Complejidad del Producto

Existen tres categorías para los productos de programación:

Programas de Aplicación: generalmente se desarrollan bajo el ambiente proporcionado por un compilador. Las interacciones con el sistema operativo se limitan a las instrucciones de control del trabajo y al llamado a las facilidades del lenguaje durante el tiempo de ejecución. Se incluyen procesamiento de datos y programas científicos.

Programas de Apoyo: se escriben con el fin de permitir al usuario ambientes de programación complicando el empleo del sistema operativo. Compiladores, ligadores y sistemas de inventarios.
Programas de Sistema: interactúan directamente con el equipo, éstos suelen utilizar un proceso concurrente y trabajan bajo ciertas limitantes de tiempo de ejecución. Sistema de bases de datos, sistemas operativos y sistemas para tiempo real.

IV.2.3. Tamaño del Producto

Un proyecto grande de programación es obviamente más caro en su desarrollo que uno pequeño. La tasa de crecimiento en cuanto al esfuerzo requerido aumenta con el número de instrucciones de código fuente.

IV.2.4. Tiempo Disponible

El esfuerzo total del proyecto se relaciona con el calendario de trabajo asignado para la terminación del proyecto. Los proyectos de programación requieren más esfuerzo si el tiempo de desarrollo se reduce o incrementa más de su valor óptimo.

IV.2.5. Nivel de Confiabilidad Requerido

La confiabilidad de un producto de programación puede definirse como la probabilidad de que un programa desempeñe una función requerida bajo ciertas condiciones especificadas y durante cierto tiempo. La confiabilidad puede expresarse en términos de exactitud, firmeza, cobertura y consistencia del código fuente. Las características de la confiabilidad pueden instrumentarse en un producto de programación, pero existe un costo asociado con el aumento del nivel de análisis, diseño, instrumentación y esfuerzo de verificación y validación que debe aportarse para asegurar alta confiabilidad.

El nivel de confiabilidad deseado debe establecerse durante la fase de planeación al considerar el costo de las fallas del programa, en algunos casos, las fallas pueden causar al usuario pequeñas inconveniencias, mientras que en otros tipos de productos puede generarse gran pérdida financiera.

IV.2.6. Nivel Tecnológico

El nivel de tecnología empleado en un proyecto de programación se refleja en el lenguaje utilizado, la máquina abstracta (tanto el equipo como los programas de apoyo), las prácticas y las herramientas de programación utilizadas. El número de líneas de código fuente escritas por día es independiente del lenguaje utilizado y que las proposiciones escritas en un lenguaje de alto nivel suelen generar varias instrucciones a nivel de máquina. El uso de un lenguaje de alto nivel, en vez de un ensamblador, aumenta la productividad. Las reglas de verificación de tipos de datos y los aspectos de la documentación de estos lenguajes mejoran la confiabilidad y la capacidad de modificación de los programas. Los lenguajes modernos de programación

brindan características adicionales para mejorar la productividad y confiabilidad del producto de programación.
La abstracción de la máquina en cuestión se refiere al conjunto de facilidades del equipo y de los paquetes del sistema utilizado durante el proceso de desarrollo. La familiaridad, estabilidad y facilidad de acceso a dicha abstracción influyen en la productividad del programador y, por ende, en el costo del proyecto. La productividad se afectará si los programadores tienen que aprender a usar un nuevo ambiente de programación como parte del proceso de desarrollo del producto.

IV.3. Técnicas de Estimación de Costos del Software

Dentro de la mayor parte de las empresas, la estimación de costos de la programación se basa en las experiencias pasadas. Los datos históricos se usan para identificar los factores de costo y determinar la importancia relativa de los diversos factores dentro de la compañía. Esto significa, que los datos de costos y productividad de los proyectos actuales deben ser centralizados y almacenados para un empleo posterior.

La estimación de costos puede llevarse a cabo en forma jerárquica hacia abajo o en forma jerárquica hacia arriba. La estimación jerárquica hacia abajo se enfoca primero a los costos del nivel del sistema, así como a los costos de manejo de la configuración, del control de calidad, de la integración del sistema, del entrenamiento y de las publicaciones de documentación.
Los costos del personal relacionado se estiman mediante el examen del costo de proyectos anteriores que resulten similares.

En la estimación jerárquica hacia arriba, primero se estima el costo del desarrollo de cada módulo o subsistema; tales costos se integran para obtener un costo total. Esta técnica tiene la ventaja de enfocarse directamente a los costos del sistema, pero se corre el riesgo de despreciar diversos factores técnicos relacionados con algunos módulos que se desarrollarán. La técnica subraya los costos asociados con el desarrollo independiente de cada módulo o componente individual del sistema, aunque puede fallar al no considerar los costos del manejo de la configuración o del control de calidad.

IV.3.1. Juicio Experto

La técnica más utilizada para la estimación de costos es el uso del juicio experto, que además es una técnica de tipo jerárquica hacia abajo. El juicio experto se basa en la experiencia, en el conocimiento anterior y en el sentido comercial de uno o más individuos.
La mayor ventaja del juicio experto, que es la experiencia, puede llegar a ser su debilidad; el experto puede confiarse de que el proyecto sea similar al anterior pero bien puede suceder que haya omitido algunos factores que ocasionan que el sistema nuevo sea significativamente diferente; o que el experto que realiza la estimación no tenga experiencia en ese tipo de proyecto.

TESIS CON
FALLA DE ORIGEN

IV.3.2. Estimación del Costo por la Técnica DELPHI

El nombre Delphi proviene de la Antigua Grecia. Delphos fue la localidad donde estuvo el más famoso santuario panhelénico, centrado en el oráculo de Apolo, donde según la leyenda, el oráculo de Apolo manifestaba la voluntad de Zeus a través de una sacerdotisa ('la pitonisa), cuyas ambiguas palabras interpretaban los sacerdotes. Este oráculo alcanzó prestigio en los siglos V, VI y VII antes de J.C.

El primer estudio Delphi fue realizado en 1950 por la Rand Corporation para la fuerza aérea de EE.UU. y se le dio el nombre de "Proyecto Delphi". El objetivo de este estudio fue obtener el mayor consenso posible en la opinión de un grupo de expertos por medio de una serie de cuestionados intensivos, a los cuales se les intercalaba una retroalimentación controlada.

El propósito de este estudio fue la aplicación de la opinión de expertos a la selección -desde el punto de vista de una planificación de la estrategia soviética- de un sistema industrial norteamericano óptimo y la estimación del número de "bombas A" requeridas para reducir la producción de municiones hasta un cierto monto. Es importante recalcar que los métodos alternativos de manejar este problema habría involucrado un proceso prácticamente prohibitivo, en términos de costo y de tiempo, de recolección y procesamiento de la información.

Es así, como las justificaciones originales para este primer estudio Delphi aún son válidos para muchas aplicaciones, cuando no se dispone de la información precisa, es muy costoso conseguirla o la evaluación requiere de datos subjetivos en los principales parámetros.

La técnica Delphi se ha convertido en una herramienta fundamental en el área de las proyecciones tecnológicas, incluso en el área de la Administración clásica y operaciones de investigación. Existe una creciente necesidad de incorporar información subjetiva (por ejemplo análisis de riesgo) directamente en la evaluación de los modelos que tratan con problemas complejos que enfrente la sociedad, tales como, medio ambiente, salud, transporte, comunicaciones, economía, sociología, educación y otros.

La técnica DELPHI se desarrolló en la corporación RAND en 1948 con el fin de obtener el consenso de un grupo de expertos sin contar con los efectos negativos de las reuniones de grupos. La técnica puede adaptarse a la estimación de costos de la siguiente forma:

1) Delimitar el contexto y el horizonte temporal en el que se desea realizar la provisión sobre el tema en estudio.

2) Seleccionar el panel de expertos y conseguir su compromiso de colaboración.

TESIS CON
FALLA DE ORIGEN

3) Un coordinador proporciona a cada experto la documentación con la Definición del sistema y una papeleta para que escriba su estimación.

Un formato aplicable para el uso de esta técnica se presenta en la siguiente Figura IV.3.2.

Proyecto:_____ Fecha: _____
Estimación: en la __ vuelta.
Su estimación de tiempo es: .
Tiempo: 0---1---2---3---4---5---6---7---8---9---10---11---12 Meses de:
Su estimación de Costo es:
$(Costo): _____ Pesos
Razones de su estimación (Justificación del costo estimado):
1.- _____
2.- _____
3.- _____
Su estimación para la siguiente vuelta: ____

Figura IV.3.2. Formato Técnica DELPHI para Estimación de Costos

4) Cada experto estudia la definición y determina su estimación en forma anónima; los expertos pueden consultar con el coordinador, pero no entre ellos.

5) El coordinador prepara y distribuye un resumen de las estimaciones efectuadas, incluyendo cualquier razonamiento extraño efectuado por alguno de los expertos.

6) Los expertos realizan una segunda ronda de estimaciones, otra vez anónimamente, utilizando los resultados de la estimación anterior. En los casos que una estimación difiera mucho de las demás, se podrá solicitar que también en forma anónima el experto justifique su estimación. El proceso se repite tantas veces como se juzgue necesario, impidiendo una discusión grupal durante el proceso.

7) De la información arrojada por los expertos será procesada estadísticamente hasta encontrar un consenso.

TESIS CON
FALLA DE ORIGEN

IV.3.2.1. Justificación del Método DELPHI

Es una técnica cualitativa, es decir, subjetiva de juicio. Basada en cálculos y opiniones para establecer un pronóstico.

El propósito principal de un pronóstico es el de producir información sobre posibles comportamientos futuros de ciertos factores o variables comprendidos en el área de interés. El pronóstico debe contribuir a una mínima comprensión de las incertidumbres del futuro, de manera tal que quienes toman decisiones de impacto e implementan, puedan hacerlo a sabiendas del nivel de riesgo y oportunidad.

La iteración del cuestionario permite la retroalimentación de las opiniones y facilita la interacción entre los participantes. Así se llega a la posición general del grupo ante el tema que se analiza.

Es posible reconsiderar su posición anterior, hasta llegar a consensos que hacen más nítidos los escenarios emergentes.

La información se recoge mediante un cuestionario cuyo análisis permite conocer la respuesta estadística, es decir aunque el cuestionario tenga un carácter cualitativo se realiza una medición cuantitativa del resultado. Se tiene así una respuesta mayoritaria del grupo y el grado de consenso o dispersión que existe en la respuesta.

IV.4. Aplicación del Método DELPHI al Proyecto

1) Diseñar y elaborar una herramienta grafica de apoyo didáctico en la inducción a la programación en un lapso de tres meses, como proyecto de tesis.

2) Se asumen como expertos desarrolladores de software cada uno de los integrantes de este proyecto, quienes a través de su amplia experiencia, completan las papeletas para el pronostico de costos y tiempo.

3) El que funge como coordinador es la Ing. Norma Elva Chávez.

4) Los expertos proporcionaron las siguientes justificaciones para determinar el costo:

Experto No.1
Costo de programación
Tiempo de programación
Rentabilidad

TESIS CON
FALLA DE ORIGEN

Experto No. 2
Pago de programadores
Desarrollo y planteamiento del proyecto
Tiempo de investigación, desarrollo y pruebas

Experto No. 3
Tiempo de programación
Costo por hora de programación
Recursos de la empresa

Experto No. 4
Pago de 3 meses por 2 desarrolladores $25 000 c/u
Requiere de investigación teórica sobre el tema
Énfasis en la interfaz del usuario

Experto No. 5
El grupo tiene la experiencia necesaria para el proyecto
Es un sistema que requiere de velocidad para terminar el proyecto
El software a utilizar es suficiente y necesario para que la aplicación sea desarrollada

FACTORES CONSIDERADOS POR LOS EXPERTOS	Experto 1	Experto 2	Experto 3	Experto 4	Experto 5	% DESVIACIONES
Costo de programación	SI	SI	SI	SI	SI	0
Tiempo de programación	SI	NO	SI	NO	SI	0.4
Rentabilidad	SI	SI	NO	SI	SI	0.2

5) Estimaciones efectuadas

FACTORES EVALUADOS POR LOS EXPERTOS	Experto 1	Experto 2	Experto 3	Experto 4	Experto 5	MEDIA
Estimación de tiempo	3 m	4 m	3 m	3 m	3 m	3.2

TESIS CON FALLA DE ORIGEN

Costo Estimado	70,000	150,000	60,000	150,000	98,000	105,600

La media en el tiempo Tmedia

Tmedia = (texp1+ texp2+ texp3 + texp4 + texp5) / num de exps
Tmedia = (3 + 4+ 3 + 3 + 3) / 5
Tmedia = 3.2 meses

La desviación con respecto al tiempo medio Desvt

Desvtexps – Tmedia

Desvtexp1 = 3 - 3.2 = 0.2 m
Desvtexp2 = 4 - 3.2 = 0.8 m
Desvtexp3 = 3 - 3.2 = 0.2 m
Desvtexp4 = 3 - 3.2 = 0.2 m
Desvtexp5 = 3 - 3.2 = 0.2 m

La media de los costos estimados Mcosto

Mcosto = (70,000 + 150,000 + 60,000 + 150,000 + 98,000) / 5
Mcosto = 528,000 / 5
Mcosto = 105,600

La desviación del costo estimado entre la media de los costos Desvc

Desvc exp1 = 70,000 – 105,600 = - 35,600
Desvc exp2 = 150,000 – 105,600 = 44,400
Desvc exp3 = 60,000 – 105,600 = - 45,600
Desvc exp4 = 150,000 – 105,600 = 44,400
Desvc exp5 = 98,000 – 105,600 = - 7,600

Por las desviaciones encontradas se propuso una segunda ronda

6) El formato de las papeletas permanece, esta es la información arrojada:

Experto No.1

TESIS CON FALLA DE ORIGEN

Salarios a programadores
Licencias
Consumibles, recursos materiales

Experto No. 2
Licencias de Software
Compra de consumibles y papelería

Experto No. 3
Programadores expertos
Costos de operación y sueldos
Licencias de software

Experto No. 4
Pago a programadores (3)
Computadoras (3), consumibles etc.
Licencias de software (3)

Experto No. 5
Costo de los desarrolladores de software
Costo de las licencias del software de desarrollo
Costo de la infraestructura (consumibles periféricos)

FACTORES CONSIDERADOS POR LOS EXPERTOS	Experto 1	Experto 2	Experto 3	Experto 4	Experto 5	% DESVIACIONES
Salarios a programadores	SI	–	SI	SI	SI	0
Licencias	SI	SI	SI	SI	SI	0
Recursos materiales, consumibles	SI	SI	SI	SI	SI	0

TESIS CON
FALLA DE ORIGEN

7) Determinación del consenso

FACTORES EVALUADOS POR LOS EXPERTOS	Expert o1	Expert o2	Expert o3	Expert o4	Expert o5	MEDIA
Estimación de tiempo	3 m	3 m	3 m	3 m	3 m	0
Costo Estimado	170,000	160,000	178,000	165,000	180,000	170,600

Tiempo sin desviación

La media del costo en la segunda vuelta Mcosto

Mcosto = (170,000 + 160,000 + 178,000 + 165,000 +180,000) / 5
Mcosto = 170,600

Desviación referente a la media del costo Desvc

Desvc exp1 = 170,000 – 170,600 = 600
Desvc exp2 = 160,000 – 170,600 = 10,600
Desvc exp3 = 178,000 – 170,600 = 7,400
Desvc exp4 = 165,000 – 170,600 = 5,600
Desvc exp5 = 180,000 – 170,600 = 9,400

Ahora los parámetros son homogéneos, la desviación presentada es mucho menor, no se considera necesaria una tercera vuelta y por tal se establece la estimación del costo de este proyecto en $ 170,600

TESIS CON
FALLA DE ORIGEN

CAPÍTULO V

Pruebas del uso del Sistema

99-A

TESIS CON
FALLA DE ORIGEN

CAPÍTULO V. PRUEBAS DEL USO DEL SISTEMA

V.1 Pruebas del sistema

Al termino de cualquier sistema, se deben aplicar pruebas al mismo con la intención de descubrir y corregir posibles errores, así estas pruebas indicaran si las funciones del software funcionan de acuerdo con las especificaciones y si alcanzan los requisitos de rendimiento. Se deben diseñar pruebas que encuentren el mayor número de errores en el menor tiempo y con el menor esfuerzo. Una prueba tiene éxito si se descubre un error no detectado hasta entonces.

En nuestro caso los programadores que desarrollamos el sistema son los que llevan a cabo la prueba, y son responsables de probar los módulos del programa, incluyendo la prueba de integración que es la que lleva a la construcción de la estructura total del sistema.

A medida que se van recopilando y evaluando los resultados de la prueba, se da uno cuenta si la calidad y la fiabilidad del software son aceptables o las pruebas no son adecuadas para descubrir errores serios.

Cualquier sistema puede ser probado por medio de las dos siguientes formas:

- **Prueba de la caja blanca:** se enfoca en la estructura de control del programa. Se realizan pruebas para asegurarse que se ejecutan al menos una vez todas las líneas de código correctamente según lo diseñado.

- **Prueba de la caja negra:** Se enfoca para validar los requisitos funcionales sin tomar en cuenta el funcionamiento interno del programa. Se realizan pruebas para encontrar funciones incorrectas o ausentes, errores de interfaz, errores en estructuras de datos, errores de rendimiento y errores de inicio y término en la programación. Esta prueba intenta descubrir diferentes tipos de errores que la prueba de la caja blanca.

En la fase de prueba de un sistema se realizan los siguientes pasos:

1. **Prueba del módulo o prueba individual.** Las unidades individuales o módulos del programa se prueban para verificar que cada una lleva a cabo la función para la cual fue diseñada. Esta prueba se lleva a cabo proporcionando un conjunto de datos predeterminados al módulo y se observan los resultados de la prueba que son los datos de salida. La prueba final verifica la lógica, la estructura interna de los datos y las condiciones de entrada y salida de los datos.
2. **Prueba de integración.** Se incorporan los módulos probados de la unidad y se construye una estructura del programa que esté de acuerdo con las especificaciones del diseño del

TESIS CON
FALLA DE ORIGEN

sistema. Se llevan a cabo pruebas para detectar errores asociados a la interacción entre los módulos. Esta prueba se enfoca al diseño y a la construcción de la arquitectura del software.

3. **Prueba de validación.** Después de realizar la prueba de integración, el software esta ensamblado como un paquete, se han detectado y corregido los errores de interfaz. La validación se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del usuario. Se deben comprobar los criterios de validación establecidos de acuerdo a las especificaciones del sistema. Este tipo de prueba se consigue mediante una serie de pruebas de la caja negra que demuestran a conformidad con los requisitos.
4. **Prueba del sistema.** Esta constituida por una serie de pruebas diferentes cuyo objetivo principal es verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.
    - Pruebas de recuperación. Es una prueba para verificar que la recuperación del sistema por diversos tipos de fallas se lleve a cabo apropiadamente.
    - Prueba de seguridad. Esta prueba verifica que los mecanismos de protección incorporados en el sistema lo protegerán del acceso no permitido. Comprueba que se cumplan los requerimientos de calidad.
    - Pruebas de resistencia. Esta prueba ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales.
    - Prueba de rendimiento. Esta prueba esta diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado.
    - Prueba de desempeño. Se validan los requisitos establecidos como parte del análisis de requisitos de software, comparándolos con el sistema que ha sido desarrollado.

El objetivo de la prueba de software es descubrir errores, par lo cual se recurre a la verificación y validación de datos.

La verificación se refiere al conjunto de actividades que aseguran que el software se implementa correctamente una función específica.

La validación es un conjunto diferente de actividades que aseguran que el software desarrollado se ajusta a las necesidades del usuario.

TESIS CON
FALLA DE ORIGEN

CAPÍTULO V. PRUEBAS DEL USO DEL SISTEMA

V.1.1. Pruebas a Pathfinder

El sistema propuesto se sometió a cada una de las pruebas mencionadas anteriormente.

La prueba de módulo o prueba individual. Se llevo a cabo desde el inicio de la programación del sistema. El lenguaje de programación en este caso Visual Basic permitió probar que cada procedimiento funcionara correctamente. Se verificó paso a paso el código del mismo.

La prueba de integración. Se efectuó durante la programación del sistema. Después de que se verificó que funcionara adecuadamente cada procedimiento de evento de los diferentes formularios, se integraron para formar un módulo esto de acuerdo a las especificaciones del diseño del sistema.

La prueba de validación. Se efectuó después de la programación del sistema. Se deben comprobar los criterios de validación establecidos de acuerdo a las especificaciones del sistema. Como por ejemplo se validaron que todos los formularios funcionaran de una manera correcta y llamaran a los procesos adecuados.

La prueba del sistema. Se llevo a cabo al poner en marcha el sistema primero con los programadores y después al realizar las pruebas con un grupo de niños en donde se realizaron las pruebas de resistencia, rendimiento y desempeño.

V.2 Pruebas del uso del Sistema

Una vez concluido el desarrollo de la aplicación, ésta le fue ofrecida a un grupo de usuarios para que la evaluaran y se comprobaran las hipótesis iniciales. El grupo de usuarios elegido fueron niños y niñas de corta edad (de 8 a 12 años de edad), que supieran leer y escribir y que no necesariamente hubieran tenido contacto con computadoras previamente. El grupo estuvo integrado por doce individuos con equilibrio de edades y sexo; el nivel socioeconómico fue irrelevante para la selección de candidatos. Las edades y sexo de los niños fueron como se ilustra en la siguiente tabla:

Niños		Niñas	
Edad	**Cantidad**	**Edad**	**Cantidad**
8	2	8	1
9	1	9	2
10	2	10	1
11		11	2
12	1	12	

TESIS CON
FALLA DE ORIGEN

CAPÍTULO V. PRUEBAS DEL USO DEL SISTEMA

La razón de haber elegido niños fue porque una de las metas iniciales del trabajo era la de crear un mecanismo fácil para que sujetos de corta edad fueran capaces de desarrollar su pensamiento creativo y su capacidad de deducción e inferencia, en el diseño y desarrollo de programas sencillo.

V.3 Naturaleza de las pruebas

Con el fin de evaluar la aceptación del sistema, les fueron ofrecidos a todos los niños una serie de antecedentes similares y un conjunto de ejercicios similares. Fueron creados dos grupos de niños, cada uno de seis individuos y se les dio un "curso" de computación utilizando a Pathfinder como herramienta de trabajo. Dichos cursos de computación fueron impartidos en cinco sesiones consecutivas de una hora diaria cada una de las sesiones. El temario de las sesiones fue esencialmente:

a) Introducción a Pathfinder: qué es, donde "vive", como funciona.
b) El entorno de Pathfinder: cómo son los planetas que visita Pathfinder; las misiones que debe cumplir.
c) Las instrucciones básicas de Pathfinder (el vocabulario de Pathfinder).
d) La primera misión de Pathfinder: recoger un trompo y dejarlo en otro sitio.
e) Enseñándole a Pathfinder a que aprenda nuevos trucos. Uso de la instrucción AQ.
f) Uso de la instrucción MI ...LG. Uso de la instrucción RP ... HQ.
g) Pathfinder toma decisiones: uso de la instrucción SI ... DM ... LG y de la instrucción SI ... LG.
h) Solución de misiones más complejas: combinando instrucciones.
i) Diseñando misiones propias.

En estas sesiones, los niños fueron capaces de aprender y comprender la fantasía que rodea a Pathfinder. Posteriormente, se les indujo a que resolvieran ciertos "planetas" sencillos (misiones) que les eran planteados por el instructor. Finalmente, se les motivó a que crearan "planetas" propios y a que los resolvieran ellos mismos. En este apartado son mostrados los resultados de tal experiencia.

Aun cuando fueron guardados los planetas y programas que eran capaces de hacer los niños, en este trabajo se muestran las conclusiones que se desprendieron de la experiencia educativa.

Hemos dividido los resultados de los ejercicios en los apartados: aceptación, comprensión, logro de objetivos, trabajo en equipo, creatividad y complejidad.

CAPÍTULO V. PRUEBAS DEL USO DEL SISTEMA

Aceptación.

La fantasía que se les enseño a los niños era la siguiente:

> Pathfinder es su nave de exploración personal. Esta nave es capaz de viajar a cualquier planeta que a ustedes se les ocurra. Ustedes pueden diseñar los planetas, ponerles nombre y también pueden diseñar y ponerle nombre a la misión que Pathfinder va a realizar en ese planeta. Llamaremos "programas" a las misiones. Las misiones de Pathfinder usualmente consisten en recoger y transportar objetos llamados trompos, para lo cual debe de esquivar muros de rocas. Pathfinder posee sensores que le ayudan a reconocer el terreno del planeta, y que le ayudan a escuchar los trompos. También posee una tenaza para recoger y depositar trompos y un receptáculo en el cual transportarlos. La recompensa que recibimos cuando una misión es cumplida exitosamente es la satisfacción de saber comandar cada vez mejor a nuestra nave. Para poder programar a Pathfinder, necesitamos aprender el lenguaje con el cual le indicaremos en qué consiste la misión que debe desarrollar.

En términos generales, esta fantasía fue plenamente aceptada por los niños. Varios de ellos se entusiasmaron con el hecho de saber que "tenían" una nave de exploración de otros planetas. Algunos de ellos, los niños más grandes, pidieron ver físicamente la nave, ante lo que se les respondía que se trataba de una nave de pruebas y que era necesario que se entrenaran con ella antes de poder comandar una nave real. Aun cuando a las niñas no les entusiasmó particularmente la idea de tener una nave espacial, aceptaron fácilmente este hecho y mostraron interés en saber como se podían comunicar con Pathfinder.

Comprensión.

Una vez hecha la introducción de la fantasía de Pathfinder, se procedió a enseñarles a utilizarlo. Para ello, se les mostró la manera de dibujar, con ayuda del *mouse*, los planetas que ellos deseaban. Una vez que los dibujaban, podían elegir salvarlos en disco duro o bien borrarlos y comenzar nuevos planetas. Se les mostró que podían dibujar muros de rocas y colocar trompos en el lugar que desearan. Finalmente, se les mostró que podían colocar a Pathfinder en cualquier sitio y que podían orientarlo de la manera que ellos desearan.

Dado que la manera de dibujar objetos en la pantalla era manteniendo presionado el botón izquierdo del *mouse*, y para borrarlos se mantenía presionado el botón derecho del *mouse*, pudimos observar que no les causó ningún trabajo la tarea de coordinación neuromuscular que ello implicaba. A los pocos minutos de haberles enseñado la manera de utilizar el mouse y colocar y

TESIS CON
FALLA DE ORIGEN

borrar objetos, ellos ya se habían habituado y procedían a hacer trazos cada vez más complicados con suficiente soltura.

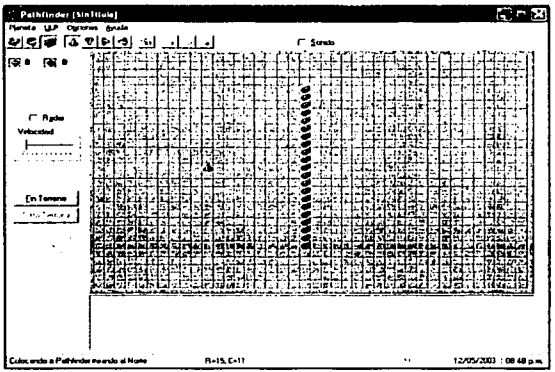
Una parte interesante fue el hecho de que los programas que Pathfinder recibía debían ser tecleados empleando únicamente mayúsculas, debido a que el scanner necesitaba recibir los tokens en mayúsculas. Esto los desconcertó un poco y en algunos de ellos les causó una cierta molestia, la cual dejaron de lado una vez que descubrieron la tecla de bloqueo de mayúsculas.

Una vez que ellos introdujeron su primer programa, se les explicó que debían de compilarlo. La explicación que recibieron era que el programa estaba escrito en un lenguaje que nosotros los seres humanos éramos capaces de entender, pero no así para Pathfinder. El compilador era el proceso mediante el cual eran traducidas las instrucciones al lenguaje que Pathfinder era capaz de entender. Esta explicación fue tan clara para ellos que no volvieron a reflexionar en la función del compilador: ellos simplemente pulsaban el botón "compilar" inmediatamente antes del botón "ejecutar".

Para los niños, el ver ejecutarse su primer programa fue de una gran excitación. El instructor disminuyó intencionalmente la velocidad al máximo de tal manera que ellos pudieran observar el desenvolvimiento de la nave al realizar sus tareas y que pudieran reflexionar sobre las cosas que Pathfinder hacía. Fue un poco difícil lograr contenerlos para que recibieran nuevas explicaciones, pues ellos ya querían programar a Pathfinder sin ayuda.

V.4 Logro de Objetivos

Una vez que se les permitió, procedieron a crear su primer programa por sí mismos. El planeta que debían resolver era el que se muestra en la siguiente figura:



El objetivo que se les planteó era que debían hacer que Pathfinder se dirigiera hacia el muro de rocas, lo rodeara y colocaran a la nave en una posición simétrica del otro lado del muro. Cabe hacer notar que en estos momentos las únicas instrucciones que ellos conocían eran AV, VI y VD. Con esas instrucciones procedieron a hacer su programa.

Curiosamente, los niños más jóvenes se abocaron a resolver la misión planteada contando cuidadosamente las casillas que Pathfinder debía de recorrer para lograr el objetivo. Los niños mas grandes se comportaron un poco más lúdicamente, con lo que lograron, eventualmente, estrellar a la nave contra el muro de rocas. Esto los divirtió mucho y transformaron la misión "pacífica" que Pathfinder debía realizar, en una auténtica misión kamikaze, buscando estrellar a la nave contra las rocas de la manera más escandalosa posible. Las niñas y los niños de corta edad lograron cumplir la misión en el tiempo mas breve que sus compañeros mas avanzados. De hecho, entre los más grandes se divertían compitiendo por hacer que Pathfinder hiciera la mayor cantidad de piruetas posibles antes de estrellarlo.

Nuestra reflexión fue que, a final de cuentas, ambos grupos de niños se habían planteado una misión a realizar y habían demostrado habilidad en el manejo del lenguaje de Pathfinder para que éste hiciera lo que ellos deseaban, por lo cual resultó válida la experiencia. Comprendimos que ellos habían comprendido como usar a Pathfinder incluso mejor que nosotros mismos.

TESIS CON
FALLA DE ORIGEN

V.4.1 Trabajo en Equipo

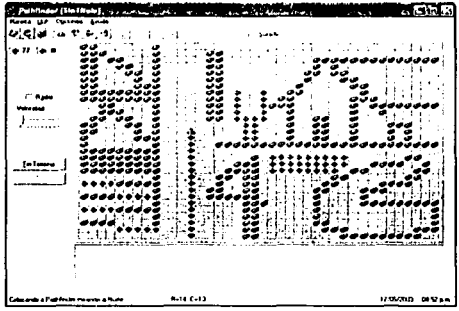
Una vez que fueron presentadas a los niños las estructuras de control MI ... LG y RP ... HQ ... LG, ellos contaron con los elementos para poder realizar programas cada vez más complejos. Inicialmente, cuando la misión a resolver consistía en elaborar programas sencillos, su actitud tendía a ser individualista y focalizada; es decir, ellos se concentraban por separado en la resolución del problema y no buscaban o permitían la participación de sus compañeros. Esta actitud cambia radicalmente cuando cuentan con las estructuras de control mencionadas arriba. Es a partir de este momento en que ellos comienzan a interactuar con el resto de sus compañeros, logrando una división del trabajo.

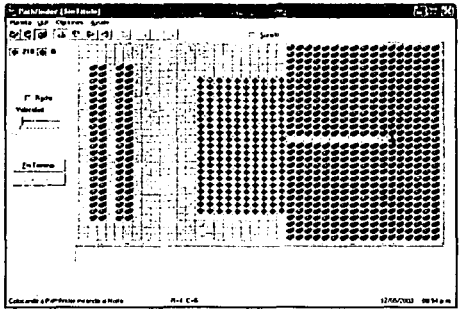
Transformando el trabajo en una diversión, ellos comenzaron a dividir la solución del problema en partes que se repartían "equitativamente" entre ellos: "tu haces este muro y yo hago el otro", "tu recoges los trompos de esa pared y yo los pongo junto a esta pared". Fue muy ilustrativo el ver como el trabajo en equipo fortalecía los lazos humanos entre los involucrados.

V.4.2. Creatividad

Los niños comenzaron a crear planetas utilizando todos los recursos a su alcance (muros y trompos) y formando las figuras mas extrañas y complejas en el terreno de Pathfinder, sin preocuparse por la manera en que podían ser recorridos estos planetas por la nave. Lo divertido inicialmente era dibujar nuevos mundos, y no preocuparse por otra cosa. Durante esta etapa surgieron planetas irregulares, intrincados y asimétricos cuya resolución prometía ser una labor ardua y altamente compleja. Hubieron algunos niños que comenzaron a utilizar las capacidades de trazado de planetas de Pathfinder como si éste fuera un programa de dibujo similar a Paintbrush o a Paint.

Producto de esta etapa en el aprendizaje de los niños fueron, entre otros, los dos planetas que se muestran en la figura siguiente:





Entre algunos de los niños fue escuchada la expresión "a ver como sale Pathfinder de ésta" o "nunca va a poder salir de aquí". Hasta ese momento veían que la solución de las misiones recala en "alguien más" que no eran ellos mismos. Sin embargo, cuando les fue explicado que ellos mismos eran los que debían de resolver el planeta que habían diseñado, su actitud cambió mucho. Algunos rediseñaron su planeta inicial e hicieron uno más sencillo de resolver. Lo que ellos mismos se dieron cuenta fue que eran más fáciles de resolver los planetas que eran simétricos o que guardaban cierta particularidad en su diseño.

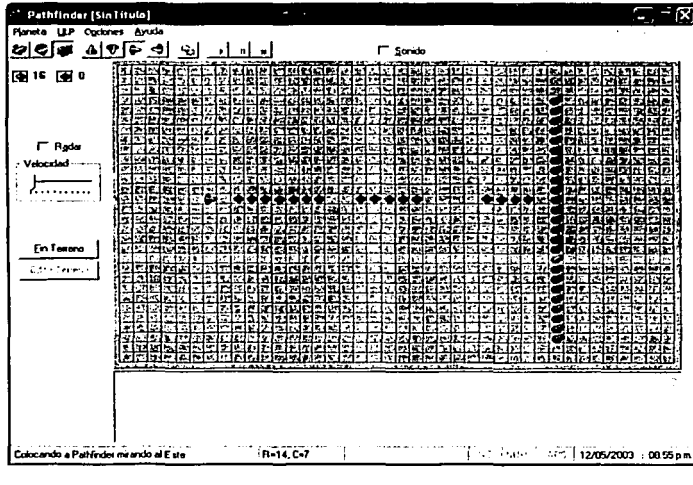
TESIS CON
FALLA DE ORIGEN

Esta circunstancia fue aprovechada para hablarles acerca de que en Pathfinder ellos mismos se planteaban un problema a resolver y ellos mismos lo resolvían.

Posteriormente el instructor cambió la dinámica del grupo, siendo él quien diseñaba el problema y los niños quienes lo resolvían. Esta fue la parte más emocionante de toda la experiencia educativa, pues los niños comenzaron a esforzarse de manera entusiasta en resolver cada nuevo planeta que planteaba el instructor.

V.4.3 Complejidad

Pathfinder resultó ser una herramienta que ofrecía retos continuos a los niños que lo utilizaron. A medida que el instructor mostraba planetas cada vez mas complicados, los niños enfrentaban el reto que implicaba el resolver la misión y, no contentos con resolver satisfactoriamente el problema, le agregaban niveles de complejidad particulares.
Un caso interesante fue la resolución del planeta que se muestra a continuación:



TESIS CON
FALLA DE ORIGEN

La misión planteada era: Pathfinder debe recorrer la distancia que lo separa del muro y recolectar todos los trompos que halle en su camino. La secuencia de instrucciones más corta que resuelve esta misión es la siguiente:

```
AQ T1 = SI AS RT DM AV LG .
AQ T2 = MI NE T1 LG VD VD .
EJ T2 .
```

Si bien esta secuencia de instrucciones resuelve correctamente la misión, lo sorprendente fue el que un par de niños le añadieron una complejidad a la solución. Ellos quisieron que Pathfinder hiciera algunas piruetas adicionales en el camino, por lo que hicieron el siguiente programa:

```
AQ T0 = VI AV AV AV VI AV AV AV VI AV AV AV VI AV AV AV AV .
AQ T1 = SI AS RT DM T0 LG .
AQ T2 = MI NE T1 LG VD VD .
EJ T2 .
```

Como se puede ver, ellos cambiaron la T1 reemplazando el AV sencillo que había, con una llamada a la tarea T0. Dicha tarea hace que Pathfinder realice, literalmente, una pirueta por cada uno de los pasos que dá. El efecto visual de esto es que Pathfinder hace piruetas cada vez que avanza un paso para recorrer la distancia que lo separa del muro.

TESIS CON
FALLA DE ORIGEN

APÉNDICE A

Manual de Usuario

TESIS CON
FALLA DE ORIGEN

APÉNDICE A

MANUAL DE USUARIO DE PATHFINDER

Este manual tiene como objetivo guiarle en el manejo del Robot Pathfinder, el cual, simplifica el manejo, aprendizaje y ejecución de programas hechos en UPL (Un Lenguaje de Programación).

INTRODUCCIÓN

Pathfinder, fue hecho para contribuir en el aprendizaje de la programación estructurada, y como apoyo a la enseñanza de los compiladores, dándole a usted, la oportunidad de adentrarse al fascinante mundo de la programación de una manera sencilla, rápida y fácil de entender. Le ofrece, también la posibilidad de innovar, creando sus propios programas así como poder guardar la información de otros programas hechos bajo ULP y poderlos ejecutar a través de la interfaz gráfica con la que éste cuenta. Dicha interfaz le proporciona herramientas al robot virtual para que lleve a cabo determinadas tareas que se le enseñan mediante la programación del mismo.

Se le dio el nombre de Pathfinder debido a la similitud que presenta (en cuanto a su operación básica) con el programa desarrollado por la NASA para la expedición al planeta Marte, en donde el Pathfinder tendría que recolectar muestras marcianas y en su viaje tendría que salvar varios obstáculos como rocas.

OBJETIVO

El objetivo de Pathfinder es el de crear ambientes limitados por rocas en el plano especialmente creado para ello, en estos ambientes pueden encontrarse dispersas varias muestras, colocadas intencionalmente por el usuario a los que llamaremos trompos.

Pathfinder, que en realidad es un robot virtual pequeño también puede dejar o recolectar trompos. Estos trompos pueden ser de color rojo (si Pathfinder los coloca) o de color verde, (si Pathfinder los recoge).Dentro del programa se cuentan con diversas herramientas que se pueden manipular como la velocidad del robot y orientación del mismo para hacer de este programa un método de enseñanza ameno y didáctico. Además de que cuenta con una ventana de programación que permite al usuario, ver o corregir lo que esta programando en tiempo real. Así como efectos de sonido que aunado al entorno amigable de la aplicación lo convierten en algo atractivo y novedoso.

A diferencia de otros lenguajes de programación Pathfinder se diseñó para programar mediante un lenguaje simbólico y fácil de entender y usar, no cuenta con signos de puntuación y se encuentra en español, por lo que es de fácil entendimiento.

TESIS CON
FALLA DE ORIGEN

APÉNDICE A

REQUERIMIENTOS DE INSTALACIÓN

- Espacio en disco duro de 3 MB mínimo
- Windows 95 o superior

PROGRAMA DE INSTALACIÓN

Pathfinder cuenta con un programa de instalación llamado Pathfindersetup, el cual instalará el programa en el siguiente directorio por default :

c:\ Archivosdeprograma\Pathfinder

También existe la opción de poder direccionar la instalación a otra carpeta. Después de haber instalado el programa, este se ejecuta con el archivo **Pathfinder.exe**

LENGUAJE DE PROGRAMACIÓN.

Pathfinder cuenta con un lenguaje básico de seis instrucciones de programación, al que de ahora en adelante llamaremos tokens:

TOKEN	DEFINICION
Avanza	AV
Vuelta Izquierda	VI
Vuelta Derecha	VD
Recoge Trompo	RT
Deja Trompo	DT
Pide Trompo	PT

Éstas seis instrucciones básicas, facilitan el empleo del sentido común, para lograr que se utilicen, como se habla coloquialmente, es decir, si se quiere que gire a la derecha; se programara con la instrucción VD VUELTA DERECHA que implicara que en sus coordenadas cartesianas el rastreador gire 90° a la derecha; un avanza implica el desplazamiento de una posición; un recoge trompo implicará que incremente su contador de trompos, y recoja el trompo en cuestión, cada una de estas instrucciones podrá realizarse tantas veces sea necesario para los fines del usuario
Cada una de estas seis instrucciones básicas AV, VI, VD, RT, PT, DT puede ser precedida:

- de cualquier instrucción o una serie de estas
- por una instrucción iterativa o una serie de estas
- por un identificador o identificadores
- o bien por un fin de instrucción.

TESIS CON
FALLA DE ORIGEN

Y entonces cualquiera de las siguientes construcciones sintácticas es valida:

1) AV AV VI RT
2) AV
3) VI DM
4) DT T1
5) PT.

Las instrucciones condicionales son las siguientes:

TOKEN	DEFINICIÓN
AE	Algo Enfrente.
NE	Nada Enfrente.
AI	Algo a la Izquierda.
NI	Nada a la Izquierda.
AD	Algo a la Derecha.
ND	Nada a la derecha.
AS	Algo Suena.
NS	Nada Suena.
AT	Algún Trompo.
NT	Ningún Trompo.
DN	Dirección Norte.

Las instrucciones condicionales, sugieren una condición, que debe cumplirse para poder ejecutar una instrucción, es decir, AE ALGO ENFRENTE, puede ser una roca, que impida al robot continuar en la misma dirección, o bien AT ALGUN TROMPO que identifique si se trata de un trompo, etc.
Cada una de estas instrucciones condicionales puede estar precedida

- por una o una combinación de instrucciones básicas.
- de instrucciones iterativas
- o identificadores

Y son construcciones sintácticas correctas las siguientes:

1) AD VD AV AV
2) AT HQ T6
3) DN T1

APÉNDICE A

Las instrucciones iterativas son:

TOKEN	DEFINICION
SI	Si
DM	De otro Modo
LG	Luego
RP	Repite
HQ	Hasta Que
MI	Mientras

El lenguaje se auxilia de instrucciones iterativas que realizarán una instrucción o bien una serie de instrucciones, que además ofrecen la posibilidad de alternar, es decir, si una condición se cumple el robot realizará cierto número de acciones de lo contrario realizará otras. Para estas instrucciones es posible enlazar otras iguales para que dentro de una iteración, de ser necesario se cumpla otra iteración hasta que la condición sea distinta.

SI solo puede ser precedida:

- de una instrucción condicional

DM solo puede ser precedida:

- de una a mas instrucciones básicas
- de una o mas instrucciones iterativas
- o de uno o mas identificadores

LG solo puede ser precedida por:

- una o mas instrucciones básicas
- una o mas instrucciones iterativas
- uno o mas identificadores
- fin de instrucción

RP solo puede ser precedida por:

- una o mas instrucciones básicas
- una o mas instrucciones iterativas
- uno o mas identificadores

TESIS CON
FALLA DE ORIGEN

APÉNDICE A

HQ solo puede ser precedida por

- instrucciones condicionales

MI solo puede ser precedida por:

- instrucciones condicionales

Para estas instrucciones iterativas las sintaxis válidas son las siguientes

SI <instrucción condicional><instrucciones básicas o iterativas> LG

SI <instrucción condicional><instrucciones básicas o iterativas> DM <instrucciones básicas o iterativas> LG

RP <instrucciones básicas o iterativas> HQ <instrucción condicional> LG

MI <instrucción condicional><instrucciones básicas o iterativas> LG

Ejemplos:

SI hay algo enfrente vuelta Izquierda; SI AE VI

Mientras no haya una roca avanza MI NE AV

Es importante considerar, que el robot, divide su desempeño en dos etapas; primero, aprende un conjunto de acciones sugeridas por el usuario, a las que llamaremos tareas; posteriormente las ejecuta, al ser invocadas, una a una ó dentro de tareas mas grandes.

Existen otras cuatro instrucciones, para que Pathfinder realice cualquier acción, referidas esencialmente, a la forma como se le debe enseñar al robot. Y otras dos que son utilizadas internamente que el usuario no usa como tal pero que son invocadas al "enseñar" o programar cada tarea

TOKEN	DEFINICION
AQ	Aprende Que
EJ	Ejecuta
=	Igual
.	Fin de instrucción

APÉNDICE A

Siempre que se quiera programar, para que realice cualquier numero de acciones, lo primero que se debe digitar, es AQ APRENDE QUE; es decir, el robot "entenderá" que se trata de una lista de instrucciones, que deberá memorizar para su ejecución.

Para poder identificar, diferenciar e invocar, para la realización de dichas tareas, será necesario, digitar, por ejemplo: T1, es decir, una letra seguida de un número, 2 números ó 2 letras, a lo que llamaremos IDENTIFICADOR. Estos IDENTIFICADORES serán nuevos identificadores, en el momento que aprende las tareas, e IDENTIFICADORES; para su realización, pero, no son parte del lenguaje, es el usuario, quien define su identificador y su consecutivo.

Para aprender

nId, Se trata internamente como un nuevo identificador y solo puede ser precedido de un símbolo de asignación.

Para ejecutar

Id se trata de identificador de alguna tarea aprendida, precedido por una combinación de instrucciones básicas o instrucciones iterativas o identificadores y finalmente por terminador de instrucción

El lenguaje cuenta con el símbolo (=) ASIGNACION referente exclusivamente a la asignación que relaciona a un nuevo identificador con la serie de instrucciones que habrá de realizar

Y con el símbolo (.) FIN DE INSTRUCCION que da por terminada la lista de instrucciones sugerida

Con lo anterior es posible construir

AQ T1 = AV AV VD
T3 = T2 AV VI RT

Finalmente EJ, EJECUTAR que implica, la realización de las acciones sugeridas, con la invocación, de cada una de las tareas aprendidas anteriormente.
El siguiente Programa ejemplificará y reafirmará lo antes mencionado:

TESIS CON
FALLA DE ORIGEN

APÉNDICE A

AQ T1 = SI NI VI DM SI ND VD LG. Aprende que, tarea uno es igual si no hay nada a la izquierda, gira a la izquierda de otro modo si no hay nada a la derecha, vuelta a la derecha luego (fin de instrucción).

AQ T2 = SI NE AV DM T1 LG. Aprende que tarea dos es igual si nada enfrente avanza de otro modo invoca la tarea uno, luego (fin de instrucción)

AQ T3 = SI AS RT DM T2 LG. Aprende que tarea tres es igual a si algo suena recoge trompo de otra manera invoca tarea dos, luego (fin de instrucción)

AQ T4 = MI AD T3 LG VD VD RT AV. Aprende que tarea cuatro es igual a mientras algo a la derecha invoca tarea tres luego vuelta derecha y vuelta derecha y recoge trompo y avanza

AQ T5 = MI AI T3 LG VD DT AV. Aprende que tarea cinco es igual a mientras algo a la izquierda invoca tarea tres luego vuelta a la derecha, deja trompo y avanza

EJ T4 T5.

Ejecuta
La tarea cuatro y la tarea cinco

Errores

Existe la posibilidad de que ocurran errores durante la compilación de cualquier programa, estos errores se pueden catalogar de la siguiente manera:

- **TipoError()** Determina la clase de error ocurrida durante el proceso de compilación. Para ello, se basará en los valores de BE y RQ arrojados por la rutina MAQUINAQ() y con ese contexto podrá determinar que clase de error ocurrió. Los errores que el compilador es capaz de reconocer son los siguientes:

Texto del Error	Descripción del error
Falta condicional	La estructura formada carece de un condicional cuando así lo requería. Aplica para estructuras MI, RP y SI.
Estructura Ilegal	La estructura formada es ilegal. Ocurre cuando hay dos condicionales, un SI sin DM o LG, un MI sin condicional o sin LG, etc.
Esta es palabra reservada	Emplea una de las palabras reservadas del lenguaje de Pathfinder como un nuevo identificador.
Nombre repetido	Emplear un identificador ya utilizado anteriormente como un nuevo identificador.
No existe este nombre	Invoca un nombre de tarea que no ha sido previamente definido
Falta un "=" en tu Tarea	Falta un signo "=" en la estructura de definición de una nueva tarea
No hay definición	La estructura no forma una definición correcta.
Te falto el punto	Falta un punto como terminador de instrucción.
Error indefinido	Un error cuyo contexto impide identificar el tipo de error.

- **CompError()** Esta rutina tiene como objetivo el detener el proceso de compilación cuando se encuentra un entorno que prefigure un error. No determinará el tipo de error incurrido; para ello, invocará a la rutina **TipoError()**. Una vez que regresa del proceso de identificación de un error, se encargará de recuperar al compilador de esta situación, a fin de que pueda continuar adelante con el proceso de compilación.

ENTORNO GRAFICO

Pathfinder cuenta con una interfaz gráfica que le permite al usuario, interactuar con el robot virtual esta interface fue desarrollada en Visual Basic y cuenta con algunas herramientas que hacen más amable y ameno el uso de este programa. A continuación se describirá el entorno de Pathfinder Fig. B. Esta es la pantalla de inicio del programa. En este ambiente es en donde se pueden desarrollar las aplicaciones del programa.

TESIS CON
FALLA DE ORIGEN

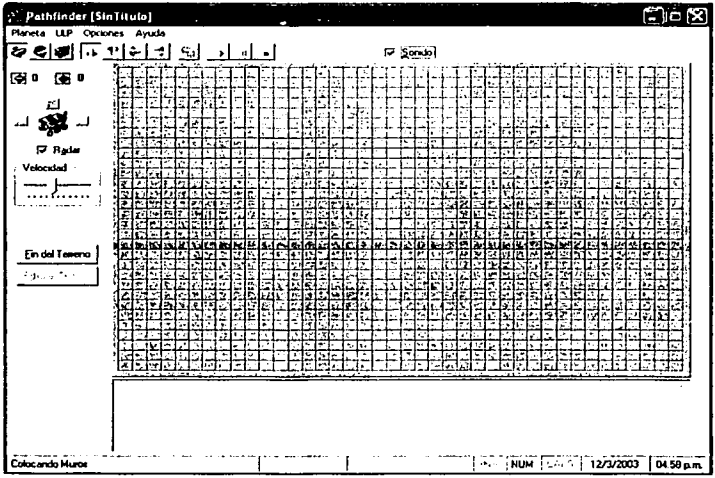


Fig B. Entorno de Pathfinder

Esta pantalla esta formada por varios elementos que se describen a continuación:

Roca

Trompo

Pathfinder

Pathfinder orientado al norte

APÉNDICE A

 Pathfinder orientado al sur

 Pathfinder orientado al este

 Pathfinder orientado al oeste

 Compila programa

 Corre el programa

 Pausa el programa

 Detiene el programa

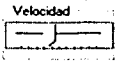
Habilita el sonido

Tablero que muestra cuantos trompos ha dejado el usuario (rojos) y cuantos ha dejado Pathfinder (verde)

Radar de Pathfinder que indica los elementos que lo rodean

TESIS CON
FALLA DE ORIGEN

APÉNDICE A



Por medio de este dispositivo se puede variar la velocidad de Pathfinder que va de menos a mas.

Fin del Terreno Da por terminado el planeta y lo bloquea, es decir no se permiten hacer correcciones.

Edita el Terreno Edita el planeta o terreno que se formó, es decir, se permiten hacer modificaciones a algún terreno existente.

A continuación se dará un ejemplo real paso a paso para el mejor entendimiento del programa. Cuando se trata de crear un nuevo ambiente Pathfinder despliega la pantalla de la Fig B.

PASO 1

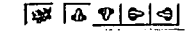
Para crear un nuevo planeta, se logra de la siguiente manera. En la barra de herramientas se encuentran las siguientes opciones:

El primer botón es una roca que sirve para delimitar el mundo de Pathfinder, para poder usarla basta simplemente con dar un clic y mover el mouse hacia donde se quiere poner dentro del grid, mantener oprimido el mouse y arrastrar para crear una barrera ó darle forma a un espacio, estas rocas Pathfinder no será capaz de atravesarlas, para borrar alguna roca basta con oprimir el botón derecho y desplazar el mouse por donde se quiera borrar la roca.

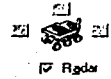
El segundo botón es un trompo, que se puede dejar en cualquier lugar del grid, y funciona de la misma manera que la roca. Y se pueden dejar tantos como se quiera dentro del planeta. El número de trompos que se dejan en el planeta se registran en las casillas marcadas con trompos de color rojo y los que recoge el robot también quedan registrados en las casillas de color verde, estas casillas se muestran a continuación

TESIS CON
FALLA DE ORIGEN

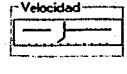
El tercer botón es Pathfinder (en miniatura), se tiene acceso a él de la misma forma que los otros dos botones con la única diferencia que sólo puede haber un robot por planeta. Y al seleccionar el botón del robot quedan habilitadas las opciones de posición del mismo.
Ahora bien el robot se puede orientar de cuatro formas, que a continuación se describen.



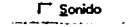
El robot puede quedar en la posición norte, sur este u oeste, para posicionar al robot basta con oprimir el botón en la posición deseada y este cambiara inmediatamente de dirección.
También se puede habilitar la opción de radar para el robot, esta opción permite saber en tiempo real en que dirección esta caminando el robot. Cuando se habilita se muestra el robot de la siguiente manera.



Existe la opción de que se pueda variar la velocidad con la que se quiere ver el movimiento del robot. Basta con desplazar la barra y ajustarla a la velocidad requerida.



Ahora bien Pathfinder cuenta con sonido y esta opción se puede habilitar mediante la siguiente casilla.



TESIS CON
FALLA DE ORIGEN

Mediante estas herramientas se podría crear un planeta como el que a continuación se describe Fig. C:

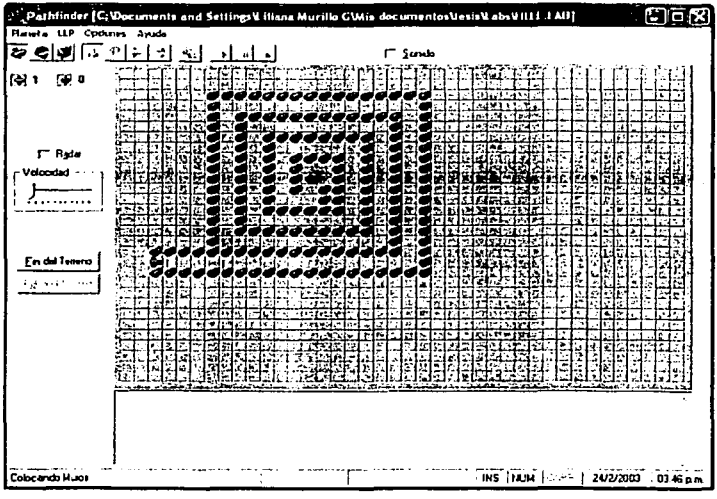
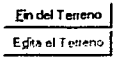


Fig. C.

Después de que se ha creado el planeta , se tiene que dar fin al Terreno. Cuando se da fin al Terreno este no se puede modificar para eso existen las opciones de Edita Terreno, el cual nos permite editarlo, también se puede editar en el menú Planeta en la opción de Redibujar.



TESIS CON
FALLA DE ORIGEN

PASO 2

Existe una ventana en donde se programa al robot virtual. Cuando se termina el programa este aparece como se muestra en la Fig. D.

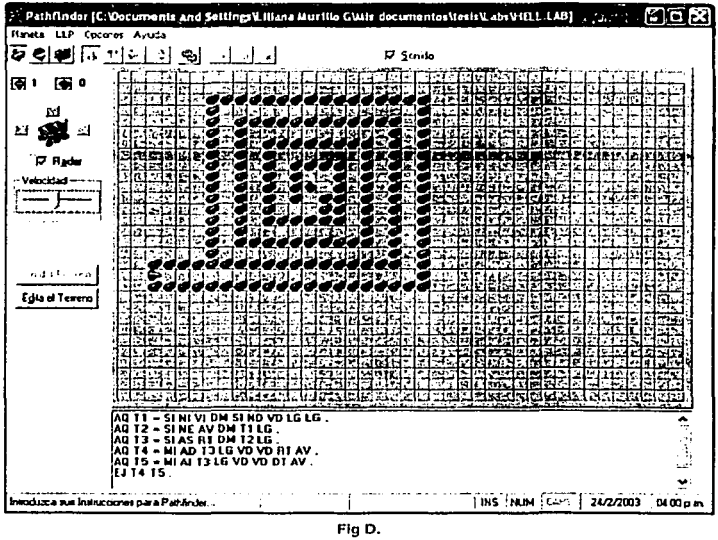


Fig D.

PASO 3

El programa desarrollado se tiene que compilar y correr para eso se tienen las siguientes herramientas:

El primer botón sirve para compilar el programa. Una vez compilado y que el programa de compilación no ha encontrado errores se habilitan los tres botones que le siguen.

TESIS CON
FALLA DE ORIGEN

El segundo botón sirve para que el programa inicie, se pause ó se detenga. Durante este proceso se pueden habilitar las opciones de radar, sonido y velocidad..

PASO 4

Ahora que ya se ha hecho tanto el Terreno como el programa se tienen que guardar en algún lugar para esto existen opciones que se encuentran en la barra de menús. A continuación se desarrollan y explican.

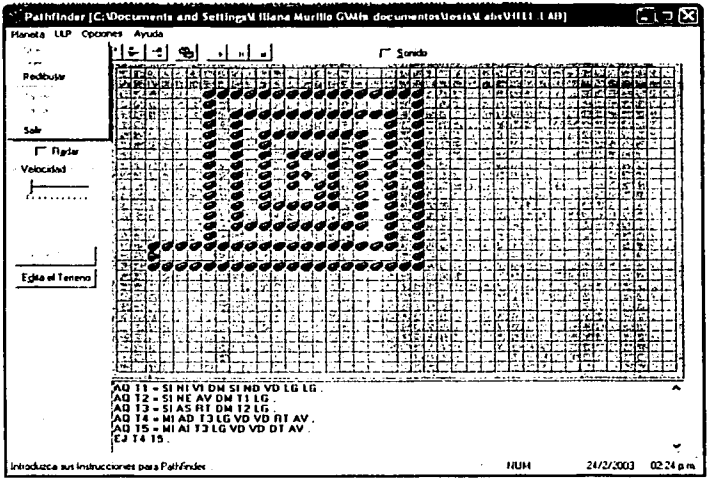


Fig. E.

MENU PLANETA

Fig. E.

Nuevo: El cual limpia el plano y nos permite crear un nuevo Terreno. Pero antes pregunta si deseamos salvar el planeta actual.

Leer: Despliega una pantalla de búsqueda en donde se localizará un Terreno ya existente, estos Terrenos tienen la terminación *.lab. Si se desea extraer uno, solo se tendrá que seleccionar de la ventana.

TESIS CON
FALLA DE ORIGEN

Redibujar: Permite hacer modificaciones a un Terreno ya existente, tales como agregar o quitar trompos, modificar la imagen del Terreno actual, etc.
Salvar: Guarda el Terreno actual con un nombre ya definido.
Salvar Como: Permite guardar un Terreno con otro nombre ó en otra ubicación.

MENU ULP

Fig. F

Nuevo: Limpia la pantalla de programación
Leer: Despliega una pantalla de búsqueda para localizar un programa ya hecho.
Salvar: Guarda el programa actual con un nombre ya definido.
Salvar Como: Permite guardar un programa con otro nombre ó en otra ubicación.

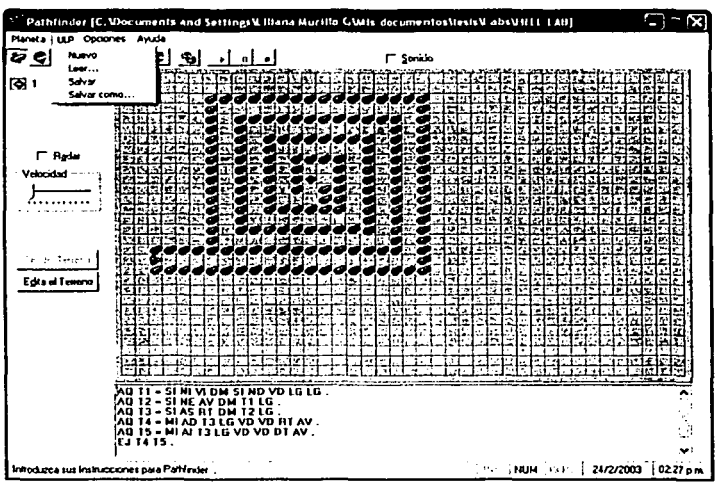


Fig F

Existen otros menús como son el de Opciones en donde se puede habilitar el sonido ó el menú de Ayuda en donde se encuentran los créditos del programa.

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

Programas en Pathfinder

TESIS CON
FALLA DE ORIGEN

126-A

APÉNDICE B

Programas en Pathfinder

En este capítulo se presentan varios programas que puede ejecutar Pathfinder.

PROGRAMA 1 → LA PIRAMIDE

En este programa lo que hace Pathfinder es que conforme va subiendo por la pirámide hecha de rocas, Fig. A va recogiendo los trompos dejados en los escalones y los guarda Fig. B
Al final deja los trompos recogidos en la base de la pirámide Fig. C.

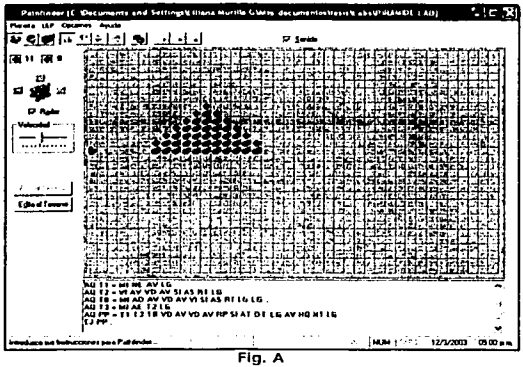


Fig. A

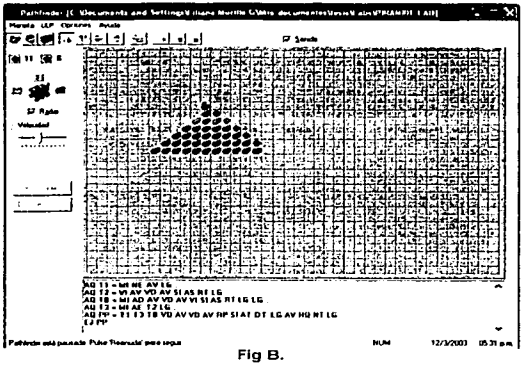


Fig B.

TESIS CON
FALLA DE ORIGEN

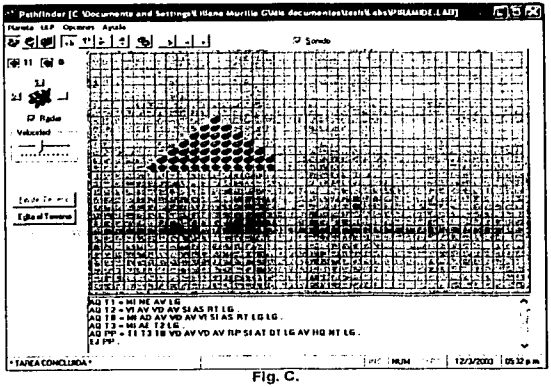


Fig. C.

PROGRAMA 2 → SUMA

Este programa realiza la suma de dos hileras de trompos colocadas al inicio del planeta, Fig. D. Después de que realiza el conteo de los trompos, coloca el total de trompos en la parte inferior del planeta, realizando así la suma de las dos hileras Fig. E.

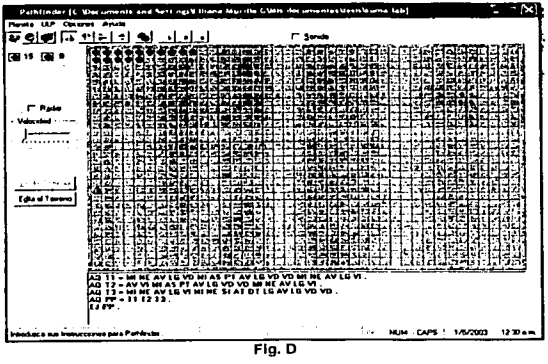


Fig. D

TESIS CON
FALLA DE ORIGEN

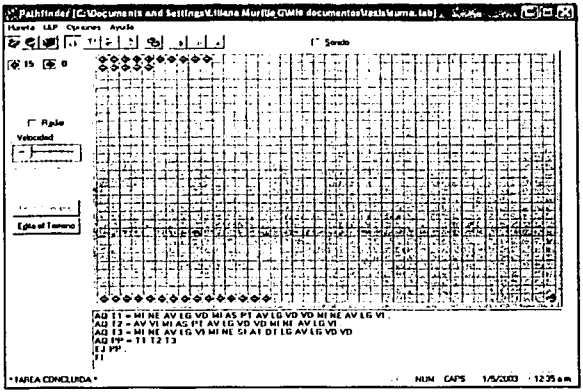


Fig E.

PROGRAMA 3 → LABERINTO

El objetivo principal de este programa es que Pathfinder recorra el laberinto mostrado en la Fig. F. Para encontrar el trompo que se localiza al centro del mismo y recogerlo Fig. G para posteriormente dejarlo a la entrada del laberinto, Fig. H.

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

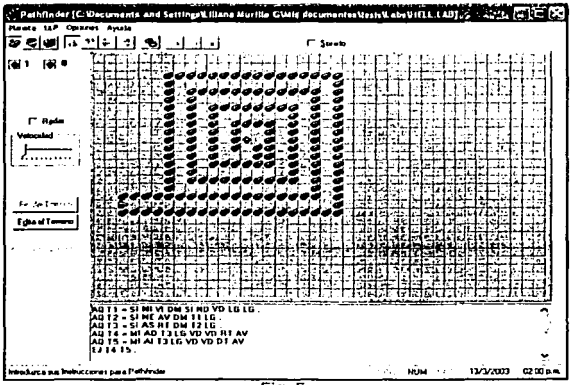


Fig. F.

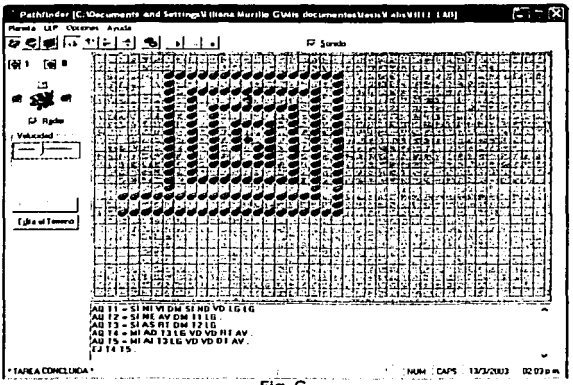


Fig. G.

TESIS CON
FALLA DE ORIGEN

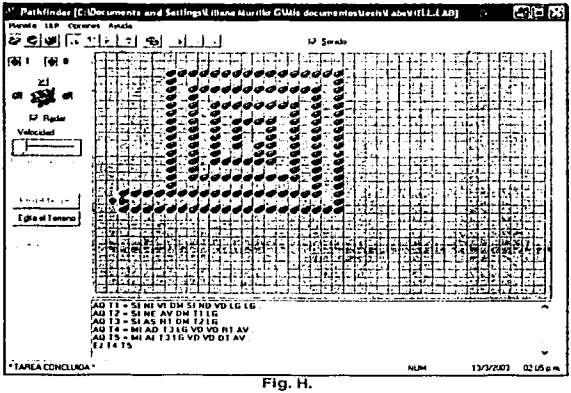


Fig. H.

PROGRAMA 4 → RESTA

Este programa realiza la operación aritmética llamada resta en donde la primer línea de trompos que se coloca al inicio del planeta es el minuendo y la segunda línea de trompos es el sustraendo, Fig. I .Lo que realiza Pathfinder es un conteo total para después ejercer la resta de trompos, Fig. J. El resultado de esta resta es colocado en la parte inferior del planeta. Fig K.

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

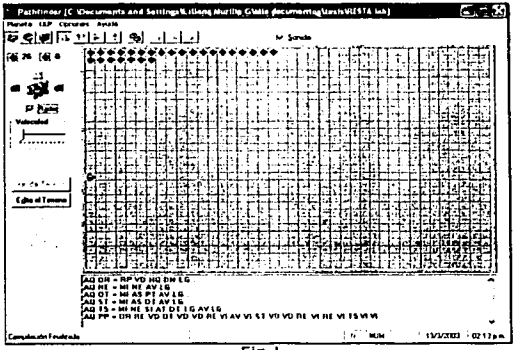


Fig. I

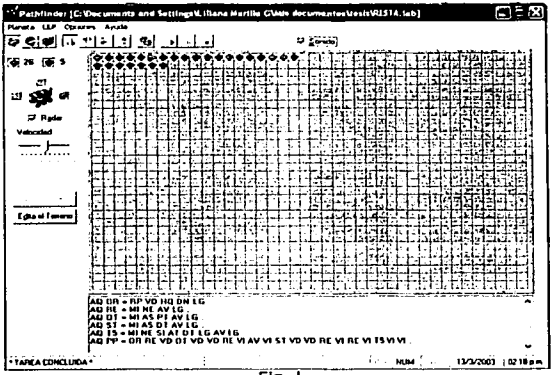


Fig. J

TESIS CON
FALLA DE ORIGEN

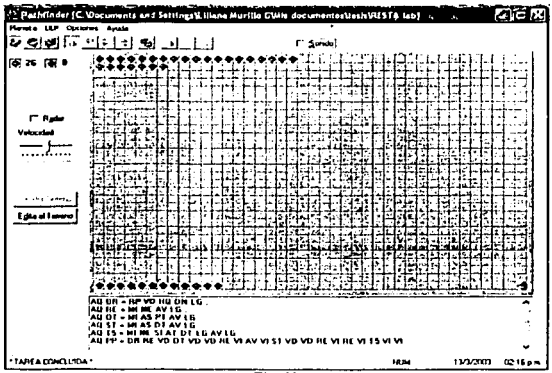


Fig. K

PROGRAMA 5 → ESPIRAL

Este programa tiene la finalidad de que Pathfinder recorra la espiral creada por rocas. **Fig. L, Fig. M.** Este programa es un claro ejemplo de que con pocas instrucciones Pathfinder puede realizar tareas complicadas **Fig. N.**

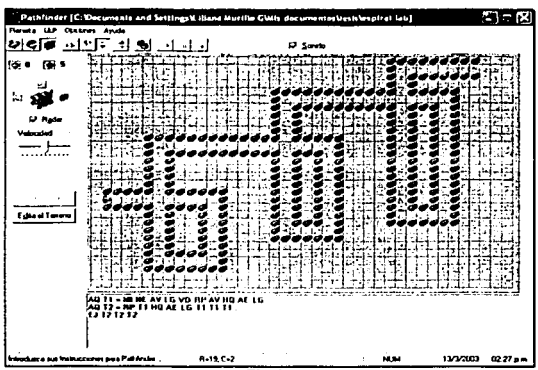


Fig. L.

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

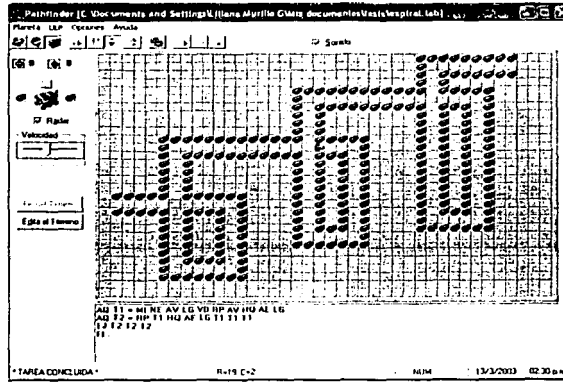


Fig. M

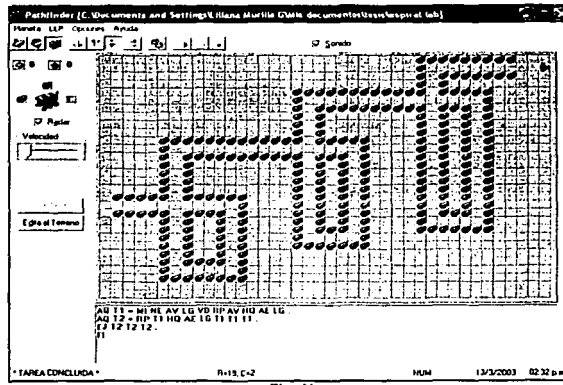


Fig. N

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

PROGRAMA 6 → MULTIPLICACIÓN

El Planeta de este programa es muy sencillo: en el primero renglón del tablero se coloca horizontalmente, alineado a la izquierda y de manera continua un conjunto de trompos que representen el primer factor.

En el segundo renglón, se coloca de la misma manera el segundo factor. Se coloca a Pathfinder en cualquier renglón por debajo de los conjuntos de trompos, y alineado a la izquierda. la orientación no importa, porque Pathfinder se orienta hacia el norte, enfilado hacia los trompos. Fig. O

Pathfinder recaba el primer conjunto de trompos el número de veces indicado por el segundo conjunto de trompos y el resultado lo coloca en la parte inferior del planeta. Fig. P.

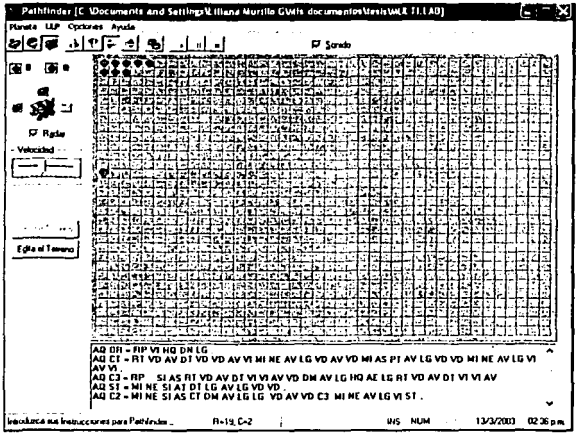


Fig O.

TESIS CON
FALLA DE ORIGEN

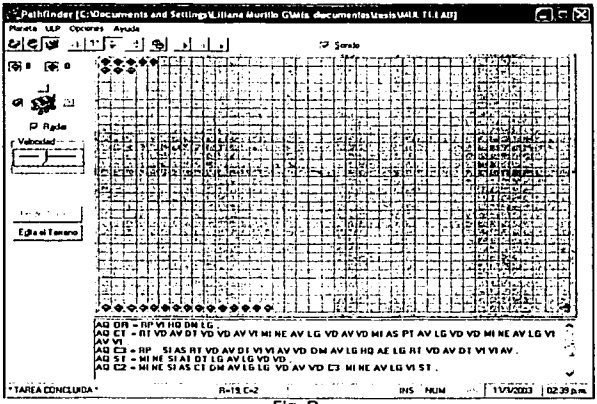


Fig. P

PROGRAMA 8 → RECOLECCIÓN DE TROMPOS
Este programa realiza una búsqueda de trompos en un planeta delimitado por rocas. **Fig. Q**, para después recolectarlos y llevarlos fuera de ese planeta **Fig. R.**

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

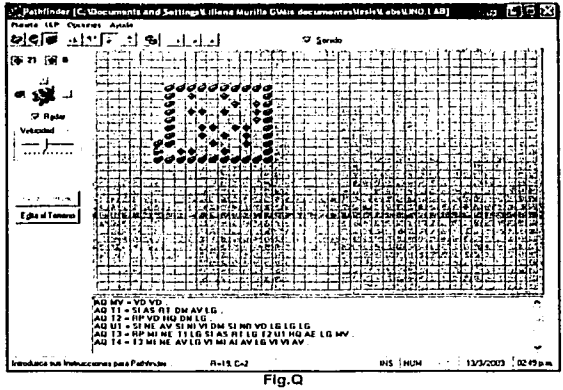


Fig.Q

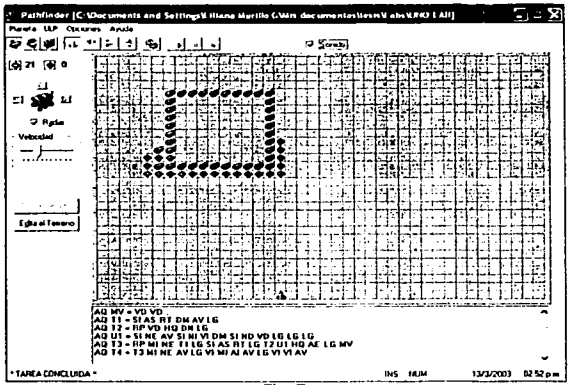


Fig. R

TESIS CON
FALLA DE ORIGEN

CONCLUSIONES

TESIS CON
FALLA DE ORIGEN

CONCLUSIONES

La importancia y relevancia que tiene el presentar un trabajo final, en especial un trabajo de tesis, se ve reflejada a lo largo de todas estas páginas, donde se ha expuesto el desarrollo y diseño de un lenguaje propio para la operación de un sencillo compilador que gráficamente se ve representado por un robot virtual.

El tiempo que nos llevó la elaboración de este proyecto fue un aproximado de tres meses aunque al final casi fueron cuatro, debido a que hubo factores muy importantes que influyeron para que el proyecto quedara finalmente como se muestra en este trabajo de tesis.

Los factores predominantes fueron el tiempo de programación y las pruebas del sistema, ya que esa parte fue complicada debido a que se tuvieron que reconocer errores específicos en tiempo de ejecución en el programa, y en la sintaxis del mismo. Este tipo de problemas se lograron resolver satisfactoriamente debido a los conocimientos adquiridos en la carrera y al trabajo en equipo.

También nos llevo tiempo el probar el sistema y ver que realmente era fácil de entender y usar. Esto lo conseguimos gracias a la ayuda de un grupo de niños que pudimos reunir, ellos fueron de gran ayuda ya que realmente nos ayudaron a ver si la metodología que estábamos usando para interesarlos en la programación era realmente la adecuada, y si el programa cubría las expectativas que esperábamos. Y para fortuna de nosotros resulto que si cubría dichas expectativas.

Este trabajo nos ha hecho retomar conocimientos que fueron adquiridos en el pasado como las materias de introducción a las computadoras, compiladores, estructuras de datos y ponerlos en práctica en un proyecto pensado en el beneficio no solo de unos cuantos, sino de todo aquel que sienta curiosidad por aprender a hacer algo que, de entrada, pareciera imposible... programar.

Nuestro principales objetivos al diseñar esta tésis fueron los siguientes:

- Proporcionar un medio que pueda estimular la imaginación y la creatividad de la gente en el desarrollo de programas

- Mediante este proyecto poder ofrecer una herramienta rentable y práctica para la enseñanza de la programación.

TESIS CON
FALLA DE ORIGEN

- La creación de un lenguaje destinado a la creación de un compilador, muy sencillo en su manejo y operación, mediante un lenguaje simbólico

- Poder facilitar este programa a diversos niveles escolares para que desde niños hasta gente adulta puedan interesarse en el desarrollo de programas de una manera rápida y sencilla.

- Proponer este proyecto como una especie de introducción a la programación en la facultad de ingeniería.

Conforme se fue desarrollando el trabajo se fueron logrando los objetivos trazados, estos fueron los siguientes:

- Al desarrollar el entorno gráfico del sistema se logró proporcionar al usuario un medio amigable y sencillo para que se interesara en el programa y sintieran curiosidad hacia el programa.

- Al ser una herramienta desarrollada por alumnos de la facultad y sin fines de lucro sino como parte de un proyecto de tesis el programa se hace altamente rentable.

- En cuanto se le empezó a enseñar a niños por ejemplo el "lenguaje" de Pathfinder se hizo de tal manera que resultara muy sencillo, y divertido debido a la explicación que se les daba de donde vivía ese robot y que tareas realizaba se logro que niños de corta edad aprendieran a programar

- También se cumplió el objetivo de enseñarles lo que es un lenguaje simbólico en nuestro caso que es un lenguaje formado por dos letras o dos números o la combinación de estos elementos.

- Al final de la experiencia de enseñarles a programar a niños de corta edad con la ayuda de Pathfinder, nos dimos cuenta que el pensamiento analítico que es la base necesaria para poder programar, está latente en todos nosotros. Solo hace falta que recibamos la instrucción adecuada para saberlo desarrollar.

Fue muy interesante el darnos cuenta que todos los niños fueron capaces de enseñarle a Pathfinder a que resolviera tareas cada vez mas complejas de una manera u otra. Es decir los niños de 8 años tardaban un poco más en resolver los problemas, en cambio los niños más

TESIS CON
FALLA DE ORIGEN

grandes resolvían el problema de una manera más rápida, aunque a veces no era la mejor solución, esto permitía que se fueran interesando más en el sistema y que aprendieran más rápido que los chicos de corta edad.

Esto nos refuerza la idea de que programar computadoras no es labor reservada a unos cuantos iniciados en esto, sino que cualquier persona es capaz de desarrollar esta disciplina. Siempre y cuando exista un interés y un buen motivador.

Finalmente ese es el objetivo de esta tesis, acercar a todo tipo de personas a este mundo tan fascinante y a la vez tan misterioso de la computación, y que este trabajo de tesis sirva como una base para que las personas se interesen más en programar en especial al desarrollar programas para la operación de equipos de cómputo o para el desarrollo de sistemas que permitan hacer de tareas difíciles cosas fáciles, de entender y rápidas de resolver.

Creemos firmemente haber logrado nuestros objetivos al demostrar lo sencillo y divertido que puede ser programar al realizar las pruebas del sistema con el grupo de niños.

Otro objetivo que perseguimos fue el de presentar este proyecto a la coordinación de Ingeniería en Computación sea una herramienta para el aprendizaje de programación de computadoras ó que sirva como introducción ó ejemplo para esta materia. Este objetivo esperamos se cumpla al término de la presentación de este trabajo de tesis.

Sentimos que el tiempo y el esfuerzo dedicado a este trabajo ha sido fructífero y que el resultado final cumple con lo que esperábamos.

TESIS CON
FALLA DE ORIGEN

REFERENCIAS

- Pattis, Richard E, Karel the Robot: a gentle introduction to the art of programming, 2nd edition / revision by Jim Roberts, Mark StehlikJohn Wiley & Sons,' Inc., printed in the United States of America, 1995.
- Friedman, D.P.; Wand, M.; Heynes, C.T., Essentials of programming Languajes, The Mit Press, 1992.
- Sethi, R. Programming Languajes, Concepts and Constructs, Addison-Wesley Publishing Company, 1989.
- Scragg, G.W., Computer Organization, A Top Down Approach, McGraw-Hill Publishing Company, Inc., 1992.
- Budd, T., An Introduction to Object-Oriented Programming, Adisson Wesley Publishing Company, 1991.
- Field, A.J.; Harrison, P.G., Funtional Programmng, Addison-Wesley Publishing Company, 1989.
- Friedman, L.W., Comparative Programming Languajes, Generalizing the Programming Function, Prentice Hall, Inc., 1991.
- Kogge, P.M., The Architecture of Simbolic Computer, McGraw-Hill Incorporated, 1991.
- Tucker, A.B., Jr., Lenguajes de Programación, Segunda Edición, McGraw-Hill, España, 1987.
- Joyanes, Aguilar Luis.,Metodología de la programación, Mc Graw-Hill, Mexico, 1990.
- Fairley, Richard. Ingenieria de software, Mc Graw-Hill, Mexico,1990.
- Aho Alfred V, Jeffrey D. Ullman, Ravi Sethi.Compiladores Principios Técnicas y Herramientas Ed. Addison-Wesley Longman Incorporated 1986
- David Gries Construcción de Compiladores Ed. Paraninfo Madrid, España 1975
- Lemone, A Karen, Fundamentos de Compiladores: Cómo traducir al lenguaje de computadora", Ed. CECSA, 1ª. edición en México, 1999.
- Albrecht, Bob , Visual Basic 4 Guía de Autoenseñanza, Ed. McGraw Hill, España, 1996.
- Cevallos Sierra Javier Visual Basic ver. 5.0 Curso de programación Ed. Alfa-Omega México 1998

TESIS CON
FALLA DE ORIGEN

http://www.canalvisualbasic.net
http://www.ciberteca.net
http://quille.costasol.net
http://www.eidos.es
http://www.mundovb.net
http://www.members.tripod.com/
http://www.gestiopolis.com
http://www8.informatik.uni-erlangen.de
http://www.geocities.com
http://www.ciemat.es.com
http://www.upaz.edu.uy
http://www.uv.cl
http://sigma.poligran.edu.co
http://www.terra.com.pe
http://ute.edu.ec
http://www.cs.buap.mx
http://www.infor.uva.es
http://www.algoritmia.net