# PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

## FACULTAD DE CIENCIAS E INGENIERÍA


# INTÉRPRETE Y ENTORNO DE DESARROLLO PARA EL APRENDIZAJE DE LENGUAJES DE PROGRAMACION ESTRUCTURADA

**Tesis para optar por el título de Ingeniero Informático**

**Presentado por:**
**Layla Hirsh Martínez**

**Lima – Perú**
**2007**

**RESUMEN**

Este proyecto tiene como objetivo principal el diseño, desarrollo e implementación de un intérprete de un lenguaje de programación que pueda ser usado en los primeros cursos de introducción a la computación. El trabajo muestra cómo se pueden crear intérpretes, lo que en nuestro país tiene escasa tradición, a diferencia de lo que ocurre en los países más desarrollados. Además, presenta un entorno de desarrollo integrado para facilitar la introducción a la programación, ofreciendo un ambiente amigable y un lenguaje de programación totalmente basado en el idioma español. En opinión de la autora esta segunda característica favorecerá a que el alumno entienda mejor el lenguaje y los procesos de computación.

En el capítulo 1 del presente documento se presenta la descripción del problema de escoger un lenguaje adecuado para la enseñanza de los primeros cursos de programación, las opciones que tenemos en nuestra actualidad y una posible solución a este problema.

En el capítulo 2 se formula una propuesta que resuelve el problema planteado en el capítulo 1 que permite definir el lenguaje, su funcionamiento y el entorno en el que se ha de ejecutar.

El capítulo 3 presenta la implementación del intérprete y la del entorno, propuestos anteriormente.

En el capítulo 4 se exponen las observaciones, conclusiones, recomendaciones y trabajos futuros, tanto del intérprete como del entorno.

# ÍNDICE GENERAL

# GENERALIDADES

## 1.1. Definición del problema

Como se ha indicado el problema consiste en desarrollar un intérprete mostrando todas las etapas que este proceso involucra y las técnicas que se requieren, de manera que además de servir para facilitar la programación inicial, pueda ser usado adicionalmente como un modelo de estudio para generar cualquier otro intérprete.

El primer paso es precisar el lenguaje de programación del intérprete, para lo cual se tuvo en cuenta el análisis que sigue.

Hoy en día, la enseñanza de lenguajes de programación se considera como un aspecto fundamental en la formación de los estudiantes de Ciencias y de Humanidades, ya que dicho aprendizaje aporta habilidades tan importantes como el razonamiento lógico, lectura crítica, atención a detalles, y el nivel de abstracción, tanto como cualquier curso de cálculo o de lengua.

Muchas instituciones educativas utilizan como primer lenguaje de programación C, Pascal, Java, Visual Basic, Scheme, inclusive C++. Las discusiones son muchas y las razones del por qué escoger uno en lugar de otro también.

Existen muchos tipos o técnicas de programación* tales como la Programación Estructurada (PE), la Programación Orientada a Objetos (POO), la Programación

* Ver anexo: Tipos o Técnicas de Programación

Modular (PM), la Programación Concurrente (PC), la Programación Funcional (PF) y la Programación Lógica (PL) y a la vez existen muchos lenguajes de programación[†] de donde escoger, de Primera Generación (assembler), de Segunda Generación, de Tercera Generación (C, Fortran, ADA, C++, C#, Cobol, Delphi), o de Cuarta Generación.

La pregunta que nos planteamos es: ¿Cómo debería ser el lenguaje, y el entorno de desarrollo, para facilitar el aprendizaje de los elementos de programación en los cursos iniciales?

La respuesta es que el lenguaje debe propiciar la asimilación de conceptos, la adquisición de buenos hábitos de programación y el mejoramiento de las capacidades de entendimiento y de abstracción. Así, se propone finalmente un lenguaje que trata de cumplir estos objetivos, propuesta que inevitablemente ha de presentar una dosis de subjetividad pues la creación de un nuevo lenguaje siempre depende de una apreciación personal.

[†] Ver anexo: Generaciones de Lenguajes de programación

## 1.2. Marco Conceptual del problema

### 1.2.1. Lenguajes de Programación existentes

#### 1.2.1.1. Scheme

El lenguaje de programación Scheme es un lenguaje funcional y un dialecto de Lisp. Fue desarrollado por Guy L. Steele y Gerald Jay Sussman. La filosofía de Scheme es decididamente minimalista. Su objetivo no es acumular un gran número de funcionalidades, sino evitar las debilidades y restricciones que hacen necesaria su adición.

Ventajas de Scheme

Scheme, como todos los dialectos de Lisp, tiene una sintaxis muy reducida, comparado con muchos otros lenguajes. No necesita reglas de precedencia, ya que carece de operadores. Los procedimientos son objetos de primera clase, ello permite la definición de funciones de orden superior, que facilitan un mayor grado de abstracción en los programas.

Desventajas de Scheme

El estándar de Scheme es realmente minimalista y específico en sí. Ello provoca que existan multitud de implementaciones diferentes, cada una de las cuales introduce extensiones y bibliotecas propias que las hace incompatibles entre sí. Los procedimientos y variables comparten el mismo espacio de nombres. El espacio de nombres es único (Scheme es lo que se conoce como un LISP-1) y, por tanto, también incluye a las

macros, lo que hace imposible distinguir el uso de una macro del de una función [WikiScheme].

Ejemplo:

```
(define (factorial n)
  (cond ((= n 0) 1)
        (else (* n (factorial (- n ))))))
```

La llamada seria:

```
(factorial 5)
```

#### 1.2.1.2. Pascal

Fue desarrollado por el profesor Niklaus Wirth a finales de los años 60. Su objetivo era crear un lenguaje que facilitara el aprendizaje de la programación a sus alumnos. Sin embargo, con el tiempo, su utilización excedió el ámbito académico para convertirse en una herramienta para la creación de aplicaciones de todo tipo.

El lenguaje de programación en Pascal es un lenguaje de alto nivel, y se puede utilizar para cualquier tipo de propósitos. Se considera un lenguaje estructurado y práctico para los usuarios que se inician en el mundo de la programación, ya que fue creado con fines de aprendizaje. Por estas características, es utilizado en las universidades e institutos de educación para inicializar a los futuros ingenieros en sistemas o informática.

Este lenguaje es idóneo en el estudio y en la definición de las estructuras de datos. Es de sintaxis pesada, estructurado y comprueba exhaustivamente todo tipo de datos.

El código está dividido en porciones fácilmente legibles llamadas funciones o procedimientos. De esta forma, Pascal facilita la utilización de la programación estructurada.

El tipo de dato de todas las variables debe ser declarado previamente para que su uso quede habilitado, el tipo de una variable se fija en su definición; la asignación a variables de valores de tipo incompatible no están autorizadas (En C, en cambio, el compilador hace el mejor esfuerzo para dar una interpretación a casi todo tipo de asignaciones). Esto previene errores comunes donde las variables son usadas incorrectamente, porque el tipo es desconocido.

Todos los programas comienzan con la palabra clave ***program***, y un bloque de código es indicado con la pareja de delimitadores ***begin***/***end***. No hace diferenciación entre instrucciones o variables escritas en mayúsculas o minúsculas, como hace el C. El carácter punto y coma **( ; )** separa las declaraciones, y el punto **(.)** sirve para indicar el final del programa o unidad. Para algunos compiladores la línea de ***program*** es opcional [WikiPascal].

Ejemplo:

```
Program HolaMundo;
begin
   Writeln('¡Hola, mundo!');
end.
```

#### 1.2.1.3. C

El lenguaje de programación C fue ideado e implementado por Dennis Ritchie en 1972 en un DEC PDP-11[‡] usando UNIX como sistema operativo. Inicialmente, el estándar de C fue realmente la versión proporcionada por la implementación de la versión V del sistema operativo UNIX. Su rápida expansión lleva a la aparición de varias variantes y problemas de compatibilidad, por lo que en verano del 1983 se estableció un comité con el fin de crear el estándar ANSI para C.

En 1989 se adoptó finalmente el estándar y poco después aparecieron los primeros compiladores conformes a este estándar. En 1995 se adoptó la primera enmienda con algunos cambios de la biblioteca de funciones, y fue la base para desarrollar C++.

Finalmente, en 1999 se adoptó el estándar C99 con algunas mejoras e ideas prestadas de C++. Actualmente cohexisten las dos versiones, mientras los programas migran a C99.

[‡] La **PDP-11** fabricado por la empresa Digital Equipment Corp. en las décadas de 1970 y 1980. Fue la primera minicomputadora para interconectar todos los elementos del sistema (procesador, memoria y periférico) a un único bus de comunicación, bidireccional y asíncrono.

C es un lenguaje estructurado de nivel medio, ni de bajo nivel como ensamblador, ni de alto nivel como Ada. Esto permite una mayor flexibilidad y potencia, a cambio de menor abstracción.

No se trata de un lenguaje fuertemente tipado, lo que significa que se permite casi cualquier conversión de tipos. No es necesario que los tipos sean exactamente iguales para poder hacer conversiones, basta con que sean parecidos.

No lleva a cabo comprobación de errores en tiempo de ejecución, por ejemplo no se comprueba que no se sobrepasen los límites de los arreglos. El programador es el único responsable de llevar a cabo esas comprobaciones.

Tiene un reducido número de palabras clave, unas 32 en C89 y 37 en C99.

C dispone de una *biblioteca estándar* que contiene numerosas funciones y que siempre está disponible, además de las extensiones que proporcione cada compilador o entorno de desarrollo.

En resumen, es un lenguaje muy flexible, muy potente, muy popular, pero que no protege al programador de sus errores [UPM-ACM].

Ejemplo:

```
#include <stdio.h> /*Incluimos la librería "standard input output"*/
void main (int argc,char **argv {
```

```
    printf("Hola mundo\n");
}
```

#### 1.2.1.4. C++

C++ es un lenguaje imperativo orientado a objetos derivado del C. En realidad un superconjunto de C, que nació para añadirle cualidades y características de las que carecía. El resultado es que como su ancestro, sigue muy ligado al hardware, manteniendo una considerable potencia para programación a bajo nivel, pero se le han añadido elementos que le permiten también un estilo de programación con alto nivel de abstracción.

C++ no es un lenguaje orientado a objetos puro (en el sentido en que puede serlo Java por ejemplo), además no nació como un ejercicio académico de diseño. Se trata simplemente del sucesor de un lenguaje de programación hecho por programadores (de alto nivel) para programadores, lo que se traduce en un diseño pragmático al que se le han ido añadiendo todos los elementos que la práctica aconsejaba como necesarios.

C frente a C++

No podemos hablar de C sin mencionar a C++, dado que es habitual no tener muy claro cuales son sus diferencias. En pocas palabras, C++ es un lenguaje para programación orientada a objetos que toma como base el lenguaje C. La programación orientada a objetos es otra filosofía de programación distinta a la de C (programación estructurada) aunque con

C++ es posible (aunque no especialmente recomendable) mezclar ambos estilos de programación. En términos generales, podemos ver a C++ como una extensión de C. Sin embargo, como se definió a partir del estándar de C de 1989, y habiendo evolucionado por separado, hay pequeñas divergencias entre ellos.

Por lo general, se puede utilizar un compilador de C++ para compilar un programa escrito en C, inversamente la gran mayoría de los programas escritos en C son válidos en C++. De hecho actualmente la mayoría de los compiladores admiten tanto código en C como en C++ [UPM-ACM].

Ejemplo:

```
using System;

class MainClass
{
    public static void Main()
    {
        Console.WriteLine("¡Hola, mundo!");
    }
```

1.2.1.5. Java

En un sentido estricto, Java no es un lenguaje absolutamente orientado a objetos, a diferencia de, por ejemplo, Ruby o Smalltalk. Por motivos de eficiencia, Java ha relajado en cierta medida el paradigma de orientación a objetos, y así por ejemplo, no todos los valores son objetos. El código Java puede ser a veces redundante en comparación con otros lenguajes. Esto es en parte debido a las frecuentes declaraciones de tipos y conversiones de tipo manual (casting). También se debe a que no

se dispone de operadores sobrecargados, y a una sintaxis relativamente simple. Sin embargo, J2SE5.0 introduce elementos para tratar de reducir la redundancia, como una nueva construcción para los bucles ‘’’foreach’’’. A diferencia de C++ , Java no dispone de operadores de sobrecarga definidos por el usuario. Sin embargo esta fue una decisión de diseño que puede verse como una ventaja, ya que esta característica puede hacer los programas difíciles de leer y mantener. Java es un lenguaje basado en un solo paradigma. Java no permite herencia múltiple como otros lenguajes. Sin embargo el mecanismo de los interfaces de Java permite herencia múltiple de tipos y métodos abstractos. El soporte de Java para patrones de texto y la manipulación de éste no es tan potente como en lenguajes como Perl, Ruby o PHP, aunque J2SE1.4 introdujo las expresiones regulares.

Para poder trabajar con java es necesario emplear un software que permita desarrollar en java. Existen varias alternativas comerciales en el mercado: JBuilder, Visual Age, Visual Café, NetBeans, etc y un conjunto de herramientas shareware, e incluso freeware, que permiten trabajar con java. Pero todas estas herramientas en realidad se basan en el uso del Java Development Kit, creado por la Sun[§].

Desventajas de Java sobre C

Una de las características más importantes de Java es indudablemente el soporte para POO[**]. Sin embargo la POO puede ser un arma de doble

[§] Sun MicroSystems Inc. http://www.sun.com/

[**] POO: Programación Orientada a Objetos

filo desde que ella provee a los programadores de otra capa de abstracción para escudarse de las partes complejas de la programación. La POO debe ayudar a los programadores a modelar problemas de una manera más natural que la orientada a la máquina como en la programación estructurada.

Por otro lado, esta capa de abstracción extra puede alejar a los programadores del concepto de cómo trabaja en realidad la máquina. Aquellos que no fueron iniciados usando la programación estructurada podrían encarar muchas dificultades cuando necesiten hacer uso de lenguajes de menor nivel o más cerca de la máquina, como el lenguaje ensamblador usado para programar microcontroladores o de los lenguajes de definición de hardware (HDL) que se utilizan para programar dispositivos electrónicos. Por esta razón los lenguajes que utilizan el paradigma de la programación estructurada, tales como C y Pascal, son mucho más cercanos al lenguaje máquina que los lenguajes que usan la POO [WikiJava].

Ejemplo:

```
public class HolaMundo{
    public static void main(String[] args){
        System.out.println("¡Hola, mundo!");  }
}
```

### 1.2.2. Comparación entre entornos de desarrollo

En los últimos años, se han utilizado en nuestro centro de estudios dos entornos de desarrollo en los cursos introductorios de programación: Borland Turbo Pascal 7.0 y Dr. Scheme versión 301 por PLT.

A continuación, se hace una breve presentación de estos dos entornos junto con otros comúnmente usados como DevC++, TextPad y Kate .

#### 1.2.2.1. Borland Turbo Pascal 7.0

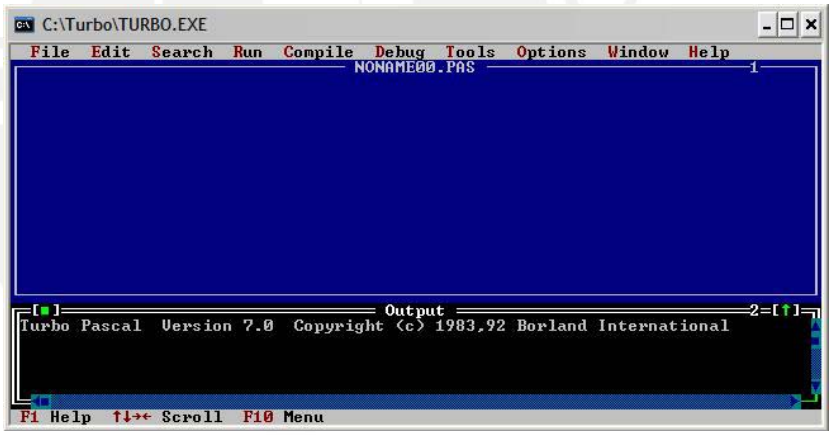



**Figura 1.**
**El entorno de Turbo Pascal 7.0**

Turbo Pascal fue el compilador Pascal dominante para PCs durante los años 1980 y hasta principios de los años 1990; fue muy popular debido a sus magníficas extensiones y tiempos de compilación sumamente cortos.

1.2.2.2. DrScheme

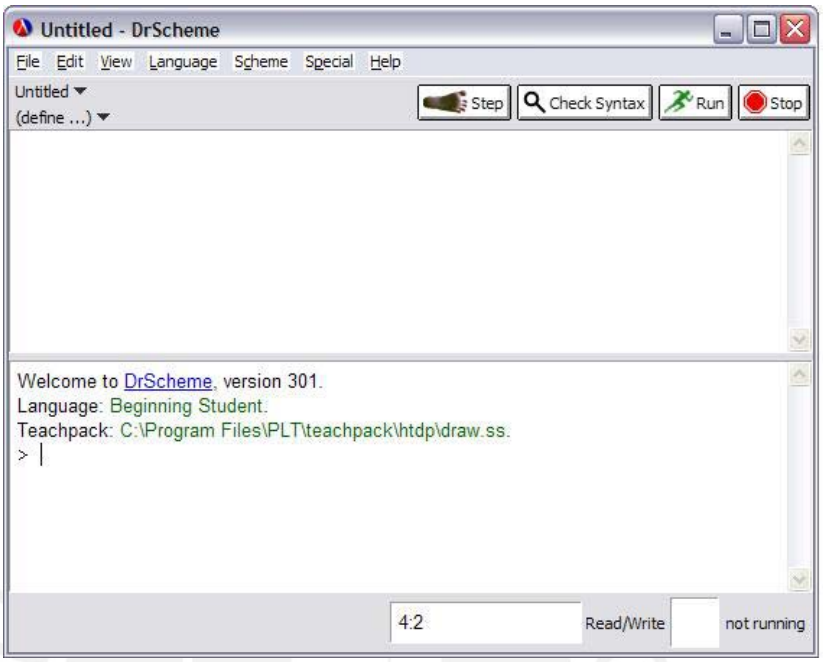



**Figura 2.**

**Dr. Scheme**

DrScheme (Findler et al, 2001) es una propuesta pedagógica basada en el lenguaje Scheme. Según las mismas palabras de sus creadores, “DrScheme es un ambiente de programación integrado e interactivo diseñado específicamente teniendo en mente las necesidades de los principiantes". Ofrece un muy sofisticado entorno de programación en el que se incluyen diferentes modos operativos de acuerdo con el nivel del estudiante (principiante, intermedio, avanzado). También incluye el manejo de paquetes de enseñanza (teachpacks), que vienen a ser proyectos en áreas específicas con el fin de conectar el código de los estudiantes con paquetes de rutinas avanzadas en gráficas, redes y otros.

1.2.2.3. DevC++

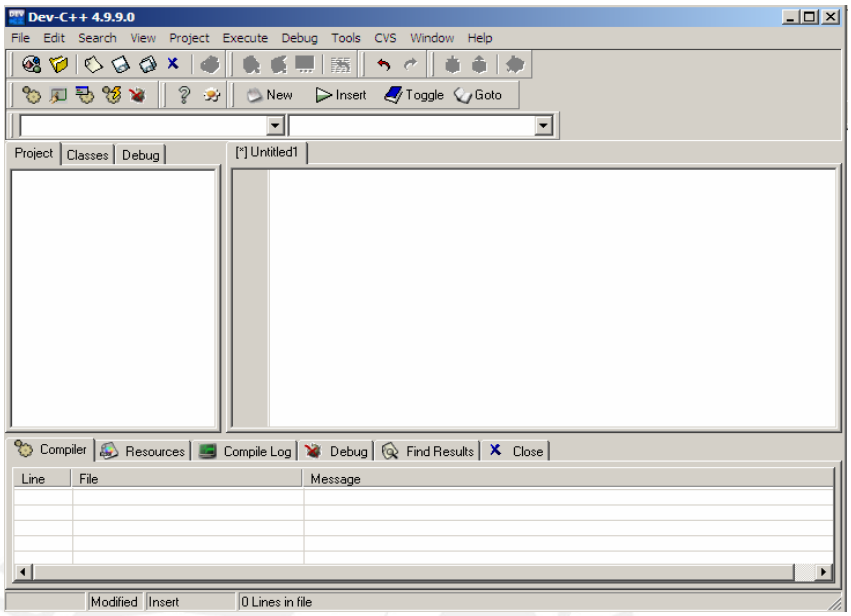



**Figura 3.**

**Dev-C++**

Dev-C++ es un entorno integrado de desarrollo libre, distribuido bajo la Licencia General Pública de GNU para programar en C y C++. El ambiente en sí tiene una apariencia similar al más comúnmente usado Microsoft Visual Studio. Un aspecto importante de Dev-C++ es la presencia de DevPaks, extensiones empaquetadas para el entorno integrado que incluyen librerías, plantillas y algunos utilitarios.

1.2.2.4. TextPad

TextPad es un editor de texto para la familia de sistemas operativos Windows que ofrece varias funciones de edición.

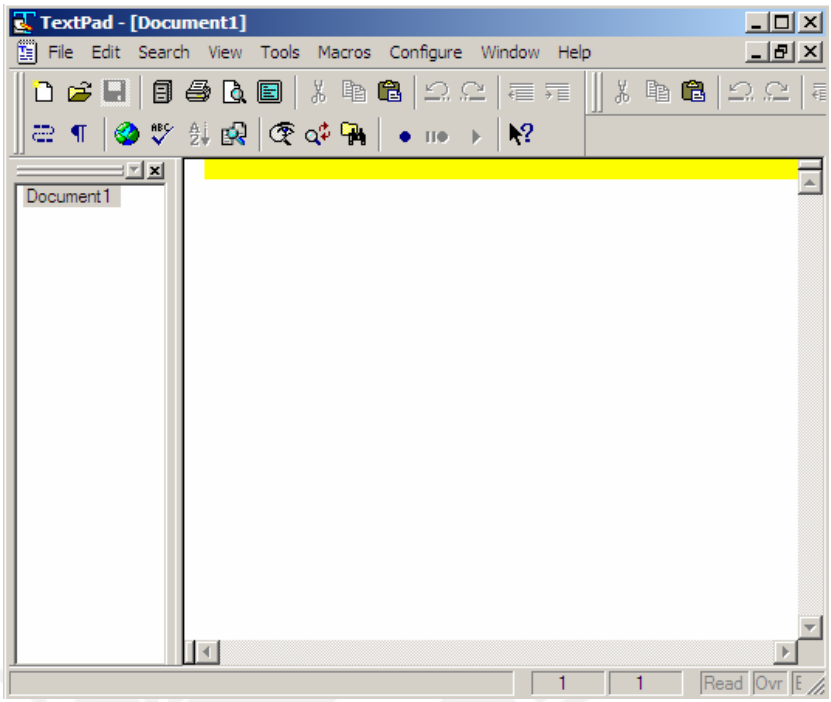



**Figura 4.**

**TextPad.**

Entre ellas tenemos indentación automática de código, búsqueda y reemplazo de texto con expresiones regulares, soporte de macros, resaltador sintáctico, la facultad de llamar programas externos tales como compiladores y depuradores (JavaC y Java, por ejemplo), movimiento sincronizado en múltiples archivos y soporte de archivos grandes.

### 1.2.2.5. Visual C++

Microsoft Visual Studio es el principal producto que ofrece Microsoft para los desarrolladores de programas.

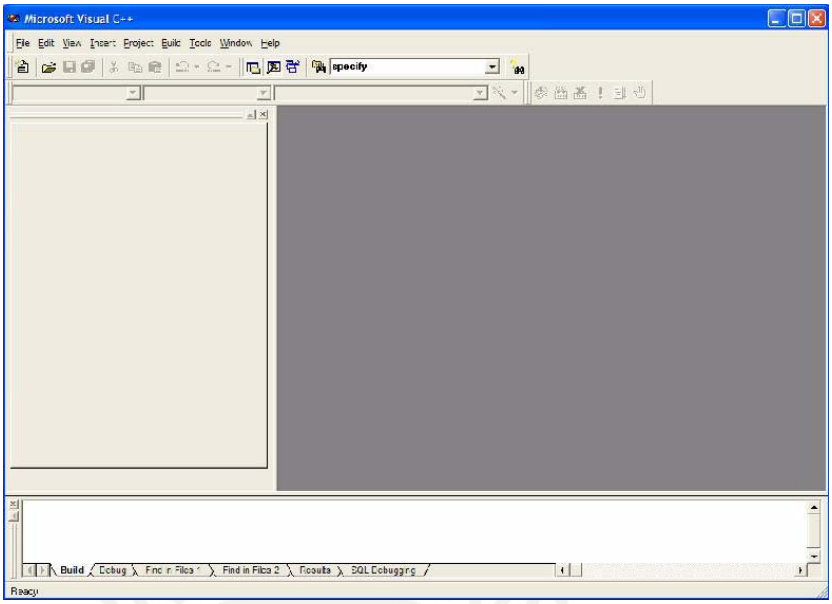



**Figura 5.**

**El IDE Microsoft Visual Studio.**

Es un IDE que permite crear aplicaciones independientes, sitios web, aplicaciones web y otros servicios que pueden usarse en cualquier plataforma soportada por el marco de trabajo .Net. Microsoft Visual Studio permite desarrollos en lenguajes como Basic, C++, C# y J#.

### 1.2.2.6. Kate

Kate es un editor de texto para el ambiente KDE de Linux. “Kate” es el acrónimo para “**K**DE **a**dvanced **t**ext **e**ditor”.

Algunas de las funciones que ofrece Kate son: motor de reconocimiento sintáctico extensible mediante archivos XML, búsqueda y reemplazo de de texto usando expresiones regulares, capacidad de “doblaje de código”, consola integrada al sistema operativo, multi-documento y soporte de sesiones.

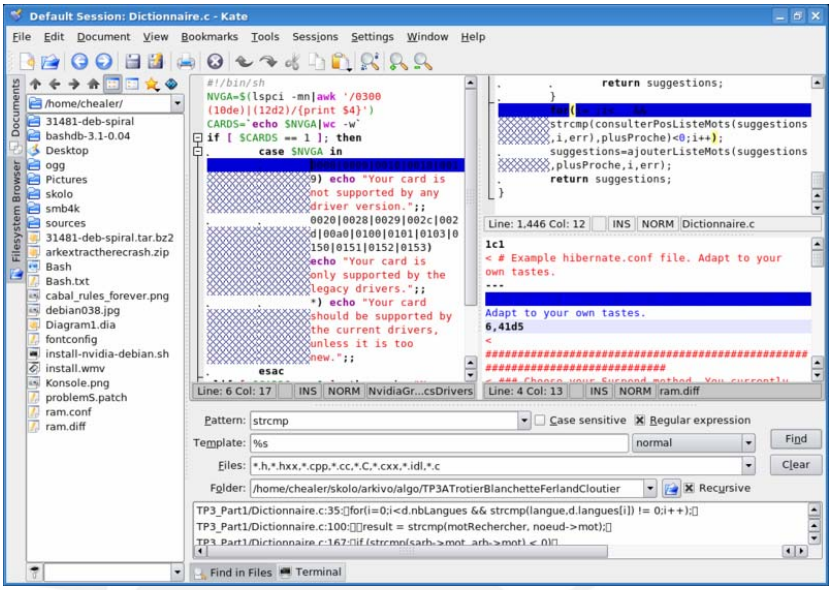



**Figura 6.**

**El editor de texto Kate.**

Es importante resaltar que de los programas mencionados, TextPad y Kate en realidad son editores de texto y no entornos integrados de desarrollo, como sí lo son DrScheme, Turbo Pascal y MS Visual Studio. TextPad y Kate no son distribuidos junto con compiladores o intérpretes de algún lenguaje específico, pero si proveen mecanismos para interactuar con programas externos, en el caso de Textpad, o con el sistema operativo por medio de una consola interna, en el caso de Kate. Así, estos editores pueden invocar compiladores como el GCC o Javac, intérpretes como Java, Scheme o Lisp o depuradores como GDB (GNU Debugger), igualando de esta manera las funciones ofrecidas por otros IDEs.

| FUNCIONALIDADES DE LOS ENTORNOS MENCIONADOS | DrScheme | Turbo Pascal | Dev-C++ | TextPad | MS Visual Studio | Kate |
|---|---|---|---|---|---|---|
| Barra de herramientas | SI | NO | SI | SI | SI | SI |
| Menús | SI | SI | SI | SI | SI | SI |
| Ventana de salida | SI | SI | SI | SI | SI | SI |
| Ventana de Errores | SI | SI | SI | SI | SI | SI |
| Manejo de archivos de programas | SI | SI | SI | SI | SI | SI |
| Edicion de texto | SI | SI | SI | SI | SI | SI |
| Compilacion / ejecucion | SI | SI | SI | SI | SI | NO |
| Ventana de comentarios | NO | NO | NO | NO | NO | NO |
| Ventana de variables | NO | SI | SI | NO | SI | NO |
| Autocompletar | NO | NO | SI | SI | SI | SI |
| Reconocimiento de palabras reservadas | SI | SI | SI | SI | SI | SI |
| Seguimiento, Punto de quiebre | NO | SI | SI | NO | SI | NO |
| Registro del usuario (ingreso) | NO | NO | NO | NO | NO | SI |
| Configuracion de usuario | NO | NO | NO | NO | NO | SI |
| Plantillas de palabras reservadas | NO | NO | NO | SI | SI | SI |
| Conversion de diagramas de flujo a codigo fuente | NO | NO | NO | NO | NO | NO |

**Figura 7.**

**Cuadro Comparativo de funcionalidades entre los ambientes y editores DrScheme, Borland Turbo Pascal, TextPad, MS Visual Studio y Kate.**

### 1.3. Estado del arte

¿Qué características debe ofrecer una herramienta computacional de programación (tanto lenguaje como entorno de desarrollo) diseñada específicamente para fines didácticos y que Tipo de programación usar?

La siguiente es una lista seleccionada de características deseables para los lenguajes de programación que favorecen fines didácticos [Kolling] según los propósitos de este trabajo:

- Conceptos bien definidos. Es importante que el lenguaje refleje un compromiso directo con los conceptos básicos que se desean enseñar. Esto se logra, en parte, al hacer que el lenguaje involucre construcciones sintácticas que representen, de manera directa y expresa, los conceptos correspondientes.
- Diseño por contrato. El lenguaje debe promover la documentación de los procesos (sus entradas y salidas), así como el uso de precondiciones, poscondiciones y afirmaciones dentro del código, de tal manera que se propicie la corrección de los programas y se enfaticen los aspectos semánticos de la programación.
- Consistencia sintáctico-semántica. Es importante que cada construcción sintáctica se asocie con un sentido específico para evitar confusiones y redundancias. Esto permite que el lenguaje sea más compacto, claro y fácil de aprender.
- Alto nivel. Esta es una característica importante que permite concentrar esfuerzos en los conceptos fundamentales, y no distraerse en aspectos internos demasiado complejos relacionados con la máquina subyacente o con la organización y con la manipulación de la memoria.

- Seguridad. El lenguaje debe contar con un fuerte sistema de tipos en tiempo de compilación, detección de variables no inicializadas y no incluir operaciones sobre apuntadores (estilo C) reconocidas como peligrosas. Cualquier tipo de error, sea en compilación o en ejecución, debe ser detectado y reportado con el mayor detalle posible tan pronto se presente.

También, es fundamental el disponer de un sistema de trabajo integrado apto para el perfil de un principiante. Las características deseables en este caso son las siguientes:

- Facilidad de uso. El entorno debe permitir una interacción inmediata y transparente con el lenguaje, lo que reducirá, en lo posible, cualquier distracción innecesaria con respecto al manejo del entorno como tal.
- Integración. El entorno debe integrar las diversas actividades típicamente asociadas con el desarrollo de programas como la edición, la compilación, la ejecución, la depuración y la visualización de documentación.
- Reutilización de código. Se debe permitir la visualización cómoda de los elementos de código disponibles en bancos existentes para su eventual referencia y reutilización en programas nuevos, así como la creación y actualización de nuevos bancos de código, buscando que el estudiante se familiarice, prontamente, con este importante rasgo de la práctica de la programación.
- Fácil transición. Es importante que los diferentes componentes y terminología utilizados en el entorno de desarrollo tengan una manifestación similar y coherente en sistemas subsecuentes.

Robinson [Robinson] presenta una serie de criterios para la elección del lenguaje a ser impartido inicialmente a estudiantes de ciencias de la computación.

1. Base matemática: Los estudiantes necesitan ver los programas como descripciones formales de algoritmos abstractos.
2. Uso estricto de tipos: El gran valor del manejo de tipos correctos y mantenibles está bien establecido. Esto es particularmente importante en la evolución de grandes sistemas en donde un equipo de programadores puede estar trabajando por varios años.
3. Énfasis funcional: Un estilo funcional conduce a una programación correcta; además, lleva así mismo a un análisis matemático de algoritmos.

Con respecto al tipo de lenguaje que se debe usar, la discusión está entre POO o PE[††], la POO es considerada por los profesionales del desarrollo de software como el mejor paradigma disponible, sin embargo esta no puede resolver todos los problemas de la programación, además de ser un paradigma aun en evolución, los lenguajes disponibles no son los mejores lenguajes Orientados a Objetos posible.

Si nuestro objetivo fuera el de dominar la POO deberíamos comenzar por dicho tipo de programación, pero nuestro objetivo es otro.

No hay duda de que las características principales de la POO (llámese encapsulación de datos relacionados y código en objetos, herencia en una jerarquía de clases de objetos) son un gran beneficio cuando se trata de construir programas grandes orientados hacia las corporaciones (sistemas de contabilidad, manejo de recursos, etc). Sin embargo el viaje es largo desde el primer curso de programación hasta llegar a la construcción de sistemas grandes (corporativos).

[††] PE: Programación estructurada

Las aproximaciones a la enseñanza de la programación se dividen generalmente en dos estilos. Un estilo comienza con un alto nivel de abstracción, usando típicamente un lenguaje puramente funcional para el primer curso de programación. Esto tiene la ventaja de enseñar buenos hábitos al estudiante, pero tiene la desventaja de no ilustrar el por qué esos hábitos son buenos. Esto es mucho como una aproximación al "como" programar, mostrando a los estudiantes la mejor práctica desde el comienzo.

El otro estilo comienza a un nivel de abstracción bajo, típicamente con una introducción a la arquitectura de los computadores, esto hace que se haga una breve introducción a los lenguajes de máquina y a el ensamblador antes de trabajar en un lenguaje de tercera generación y mas allá. Esta aproximación tiene la desventaja de exponer a los estudiantes a todo tipo de detalles grotescos, pero por otro lado los estudiantes ganan aprecio por los beneficios de las muchas innovaciones en la programación. Se puede llamar esta aproximación la del "por qué" programar, revelando los problemas de estilos de programación antiguos motivándolos a moverse a capas mas altas de abstracción.

En la filosofía del ingeniero creemos que la aproximación del "por qué" es más apropiada para orientar los cursos de programación: son estos estudiantes que tienen la apreciación del por qué se hacen las cosas de la forma en que se hacen, y son estos estudiantes los que validan esta aproximación. En contraste la aproximación del "cómo" es más para la persona que le interesa más la programación como una herramienta que como un fin en si misma y para los que en general no necesitan resolver problemas en ingeniería. Entonces, es mejor introducir a los estudiantes, especialmente a los de Ingeniería a la programación estructurada antes que a la POO.

### 1.4. Descripción y sustentación de la solución

No se cuenta con un panorama de consenso general sobre cómo atacar efectivamente la problemática de la enseñanza de la programación. Por el contrario, se presentan posiciones diversas y hasta encontradas. En este proyecto, se partió de un reconocimiento claro de esta situación y, a pesar de que existen algunas alternativas disponibles en dicho campo, este proyecto se realizó con el propósito de ofrecer un recurso adicional, fundamentalmente de carácter abierto, que apoye al profesor en la enseñanza de la programación introductoria.

Debe quedar claro que el lenguaje presente no fue creado para ser una herramienta de desarrollo a nivel de producción, sino que fue concebido con fines exclusivamente didácticos. Es un lenguaje cuya estructura sintáctica refleja un compromiso explícito con su semántica, de tal manera que reduzca el riesgo de confusión, facilite la gradual introducción de conceptos, y que cree buenos hábitos de programación.

Los lenguajes de tipo industrial comúnmente utilizados para propósitos de enseñanza de la programación (C, C++, Pascal, Scheme) no siempre se constituyen en los recursos adecuados cuando se trata de un nivel de principiante. Y el Lenguaje ICH, junto con su entorno ICHelper es una nueva propuesta para lograr este compromiso.

## 2. APORTE DE LA INVESTIGACION

El presente trabajo de tesis tiene como objetivo diseñar e implementar un lenguaje y un entorno integrado que ayude a que los alumnos entiendan conceptos básicos, que adquieran buenos hábitos de programación y que aprendan a resolver problemas sin que el lenguaje de programación sea un obstáculo, ayudando de esta manera a que mejoren sus capacidades de entendimiento y de abstracción

### 2.1. Objetivos del lenguaje

- Definir un lenguaje cuya estructura refleje un compromiso explícito con su semántica, de tal manera que se reduzca el riesgo de confusión, por lo que el lenguaje estará en español.
- Obligar al alumno a adquirir buenas prácticas de programación, tales como la inicialización de variables, y actualización de las mismas.
- Cubrir aspectos pedagógicos cruciales como palabras reservadas con sentido, relativamente fáciles de aprender y recordar; un conjunto reducido de instrucciones que, sin embargo, cuentan con semánticas flexibles; una sintaxis simple y directa que no sea obstáculo alguno para el alumno.
- Manejar datos sin tipos.

### 2.2. Objetivos del entorno

- Permitir conocer el lenguaje y su estructura.
- Configurar las características del entorno, tales como los colores de las palabras reservadas, el tipo de letra, la carpeta de trabajo, etc.

- Permitir de manera integrada y uniforme la codificación, compilación y ejecución de programas.

### 2.3. Funcionalidades implementadas en el lenguaje:

- Manejo de funciones
- Impresión de valores, ya sean numéricos o cadenas
- Manejo de arreglos
- Manejo de Bucles
- Manejo de estructura condicional
- Asignación de valores
- Manejo de funciones matemáticas

### 2.4. Funcionalidades implementadas en el entorno:

- Barra de herramientas

  Permite un rápido acceso a las funcionalidades más importantes de la barra de menús.

**Figura 8.**

**Barra de herramientas del entorno ICHelper.**

- Menús

  La Barra de Menú Principal contiene accesos a las diferentes funcionalidades del entorno.


**Figura 9.**

**El Menú del entorno ICHelper.**

- Ventana de salida

  En esta ventana se visualizan los resultados del programa al momento de ejecutarlos.

- Ventana de Errores

  En esta ventana se muestran los errores encontrados en tiempo de compilación y ejecución, el error y la línea donde se produjo.

- Ventana de Variables

  En esta ventana se visualiza el valor de las variables del programa al momento de su ejecución.

- Abrir, guardar, crear, cerrar e imprimir archivos de programas

  Estas opciones del menú permiten al usuario administrar los archivos de programas.

- Cortar, pegar, copiar, buscar texto

  Estas opciones del menú permiten al usuario editar los programas.

- Compilación, ejecución del programa desarrollado

  Estas opciones del menú permiten al usuario probar los programas elaborados.

- Ventana de comentarios

  En ella se colocará la descripción del programa, función o procedimiento.

- Reconocimiento de palabras reservadas

  Se usa para colocar las palabras propias del lenguaje en un color diferente.

- Seguimiento

  Sirve para realizar un seguimiento de los valores de las variables del programa, en tiempo de ejecución.

- Configuración de usuario

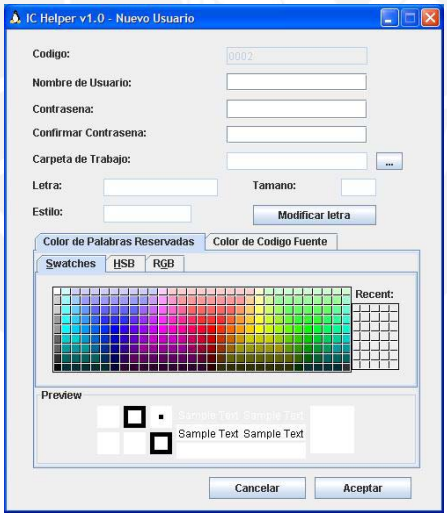



**Figura 10.**

**Ventana de configuración de Usuario.**

Configura lugares de almacenamiento de archivos así como colores a utilizar para las palabras reservadas.

- Registro del usuario (ingreso)

  Cada vez que el programa inicie, el usuario deberá colocar una clave y un identificador para poder utilizar su configuración.


**Figura 11.**

**Registro de usuario.**

- Plantillas de palabras reservadas y de operaciones básicas

  Permiten al usuario elaborar código de una manera más sencilla y directa.

# 3. IMPLEMENTACIÓN DEL APORTE

## 3.1. El intérprete ICH

El intérprete es un programa traductor. La diferencia entre un compilador puro y un intérprete consiste en que el primero de estos no ejecuta el programa que recibe como entrada, el código fuente. En lugar de ello, lo traduce a código máquina ejecutable (generalmente, llamada código objeto), el cual es guardado en un archivo para su posterior ejecución. Es posible ejecutar el mismo código fuente ya sea directamente en un intérprete o compilando y, luego, ejecutando el código máquina obtenido.

En términos de eficiencia, resulta más lento interpretar un programa que correr una versión compilada del mismo. Sin embargo, la ejecución por medio del intérprete alcanza tiempos menores que el tiempo total requerido para compilar y para ejecutar el mismo programa. Los intérpretes generan código intermedio y, generalmente, no lo almacenan. Un caso especial es el de Java, un híbrido que trabaja con código intermedio (aunque ellos lo denominan “código de bytes”), lo guarda en archivos con extensión “.class” para luego interpretar dichos archivos con su máquina virtual.

El intérprete ICH mediante el proceso de compilación genera código intermedio que luego es ejecutado por la máquina virtual de java y haciendo uso del entorno ICHelper, muestra la salida del programa elaborado, que sirvió como entrada al intérprete.

El proceso de compilación se lleva a cabo a lo largo de varias fases. Típicamente estas son: la fase de análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimización del código intermedio y generación de código final.

Algunas de estas fases dependen del tipo de compilador que se desee construir. Dado que ICH es un intérprete con fines didácticos y no es esencial tener código optimizado, la implementación sólo abarcará las cuatro primeras fases del modelo descrito.

### 3.1.1. Análisis Léxico

Para esta primera fase del proceso de compilación se consideraron varias opciones de diseño e implementación. Las principales fueron:

- Implementar un scanner a mano.
- Usar un generador de scanners similar a Lex pero para Java.
- Usar la clase String Tokenizer de Java.

Se optó por usar la clase String Tokenizer de Java ya que provee la funcionalidad necesaria para el reconocimiento de los lexemas del lenguaje. Si el lenguaje de implementación hubiese sido otro, como C, por ejemplo, muy probablemente se hubiera optado por desarrollar un scanner manual, considerando que el número de tokens y patrones léxicos en ICH no es muy extenso. Sin embargo, si éste último hubiera sido bastante mayor, se escogería un generador de Scanners al estilo de Lex para dicha labor.

El constructor de la clase String Tokenizer divide una cadena de entrada en elementos llamados tokens, unidades básicas de la gramática. Los tokens son identificados por un conjunto de caracteres separadores. En el caso de ICH, algunos de los separadores usados son todos los tipos de espacios en blanco, punto y coma (;), dos puntos (:), comillas (“) y operadores aritméticos, entre otros.

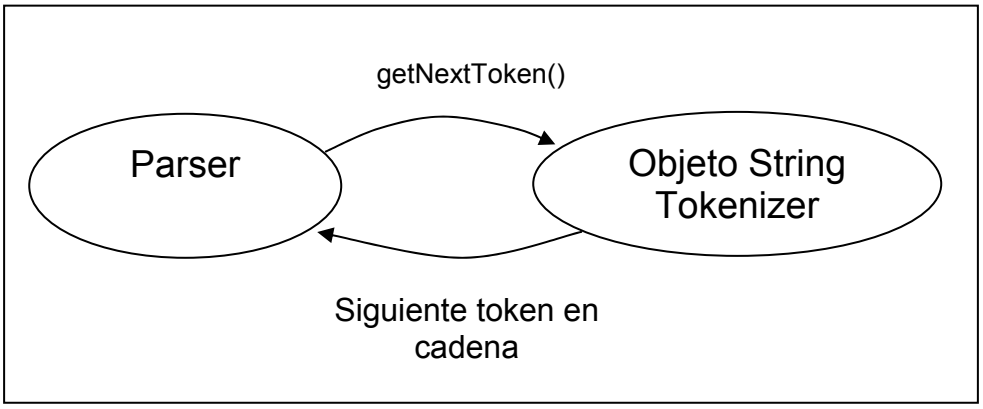



**Figura 12.**

**Interacción entre el parser y la clase String Tokenizer.**

El constructor de esta clase recibe la cadena de entrada a dividir, en nuestro caso un programa fuente en ICH, y el conjunto de caracteres separadores. Luego, cuando el parser necesita el siguiente token se hace un llamado al método getNextToken() de String Tokenizer.

### **3.1.2.** Análisis Sintáctico

Yacc (Yet Another Compiler-Compiler) es una herramienta que sirve para describir la estructura sintáctica de los lenguajes de programación. Yacc recibe como entrada el conjunto de reglas que conforman la gramática del lenguaje. Cada regla (o símbolo no terminal) puede estar compuesta por otras reglas o símbolos terminales (tokens).

Este proyecto fue realizado utilizando el lenguaje java para lo cual se cuenta con herramientas similares a Yacc. existen las herramientas JavaCC (inicialmente Jack,

creada por la Sun), CUP Parser Generator for Java (de la Universidad Técnica de Münich), CoCoJava y otras, que además de ser de fuente libre automáticamente generan parsers por medio de la compilación de especificaciones gramaticales de alto nivel colocadas en un archivo de texto. Cada generador tiene una especificación distinta y toma tiempo aprenderlas para realizar aplicaciones. También contamos con la herramienta llamada Yack-Java creada por el doctor Maynard Kong en el 2004, que utiliza la misma sintaxis del yacc estándar (o bison) del lenguaje C y que se usa para generar compiladores desde 1970 hasta el presente. Este último generador de parsers funciona de la siguiente manera:

Al ejecutar el programa se obtiene un archivo fuente en java (o del lenguaje que soporte) que contiene principalmente una clase pública (public class ytab) la cual contiene 2 métodos importantes:

- yyparse: Función responsable de realiza el análisis sintáctico a los programas, es decir, verificar si el programa escrito en dicho lenguaje coincide con la sintaxis de las reglas que componen la gramática.

- yylex: Función encargada obtener de la tabla creada por el objeto de la clase String Tokenizer los tokens y proveérselos al yyparse para que éste determine qué regla de la gramática debe seguir.

  A continuación se muestra el clásico ejemplo de expresiones aritméticas, el “Hola Mundo” de la familia Yacc, para Yack-Java:

```
YYSTYPE double
%{
        import java.io.*;
        import java.lang.*;
        %token NUM
%}
%%
        lineas   : lineas expr '\n' { System.out.println("==> "+ $2); }
                 | lineas '\n'
                 | /* e */
                 | error '\n'        {  yyerror("reingrese expresion :");  } ;
        expr     : expr '+' term              { $$ = $1 + $3; }
                 | expr '-' term     { $$ = $1 - $3; }
                 | term  ;
        term     : term '*' factor     { $$ = $1 * $3; }
                 | term '/' factor  { $$ = $1 / $3; }
                 | factor  ;
        factor   : '(' expr ')'              { $$ = $2; }
                 | '-' factor          { $$ = -$2; }
                 | NUM  ;
%%
```

### 3.1.3. Análisis Semántico y Generación de Código Intermedio

El análisis semántico se encarga de darle significado al lenguaje. Por ejemplo. El significado semántico de la instrucción WHILE de Pascal, es poder iterar o repetir

una o más instrucciones siempre y cuando una condición o expresión lógica sea verdadera. Tan pronto esta condición sea falsa las instrucciones dejarán de ejecutarse. Además, la condición mencionada se evalúa al inicio de cada vuelta o iteración.

Debido al uso de Yack-Java, el análisis semántico se realiza a la par con la fase de análisis sintáctico. Esto es, conforme el parser va reconociendo la validez sintáctica de las sentencias de un programa escrito en ICH, el intérprete genera código en una representación intermedia para su posterior ejecución.

Durante el desarrollo del intérprete ICH, y en especial en la fase de análisis semántico, se aplicaron técnicas ad hoc comunes en varios compiladores hoy en día. En particular se siguió la evaluación basada en reglas. En este método el autor del compilador especifica una serie de acciones que son asociadas con producciones de la gramática. La observación importante aquí es que las acciones necesarias para el análisis semántico, también conocido como análisis sensible al contexto o dependiente de contexto, pueden ser organizadas alrededor de la estructura de la gramática del lenguaje. Esto lleva a tener un enfoque robusto, pero muy específico y particular a la implementación del análisis semántico en el proceso de parsing del compilador.

Bajo este esquema, el autor del compilador provee extractos de código que se ejecutan en tiempo de parsing. Cada extracto o acción, es directamente ligado a una producción en la gramática. Cada vez que el parser reconoce un lugar determinado en la gramática la acción correspondiente es ejecutada para realizar su tarea.

Para posibilitar el método ad hoc de traducción dirigida por sintaxis, el parser debe incluir mecanismos que hagan posible el paso de valores de la acción donde el valor se define, a otra en donde ésta es usada.

El generador de parsers usado, Yack-Java, sigue las mismas convenciones notacionales que la familia de sistemas Yacc. Estas son:

Comunicación entre acciones

Puesto que Yack-Java genera un parser ascendente de la familia de gramáticas LR, el paso de valores entre acciones se realiza mediante la pila del parser. Aquí, cada elemento de la pila contiene información del estado del parser, del último símbolo analizado y de la acción a la cual se hace referencia.

Nomenclatura de valores

La nomenclatura de Yack-Java establece el uso de una serie de variables $X para manipular la pila. En particular, como toda estructura de tipo pila, solamente se pueden agregar nuevos elementos por la cima. Para ello, Yack-Java permite agregar como máximo un valor por acción mediante la variable $$. Esto representa el resultado del lado izquierdo de alguna regla o símbolo no terminal. Posteriormente, estos valores pueden ser recuperados mediante variables $X, donde X es la cantidad de elementos debajo de la posición actual de la pila. Además X debe tener un valor entre 1 y la cantidad de símbolos en el lado derecho de la producción (incluyendo las acciones anteriores).

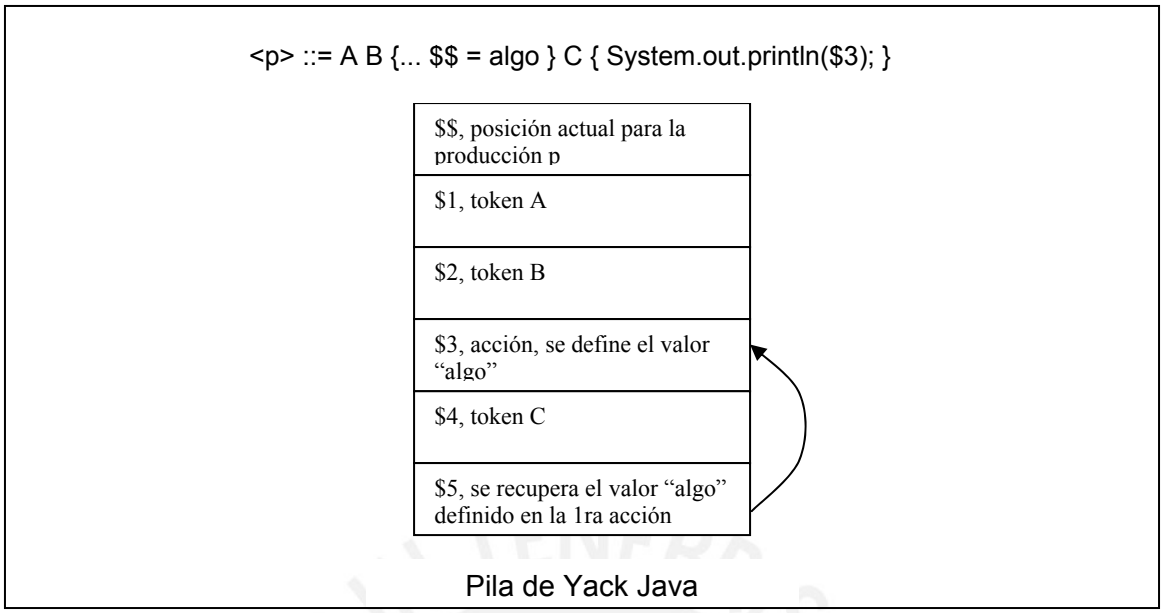



**Figura 13.**

**Configuración de la pila de Yack-Java para una producción genérica p.**

En la figura anterior, se presenta la configuración de la pila de Yack-Java para una pequeña producción p con cinco símbolos en su lado derecho. Los cinco símbolos incluyen las dos acciones definidas, la primera entre los tokens B y C, y la segunda al final de la producción, después del token C.

### 3.2. La gramática del intérprete ICH

Todo lenguaje de programación debe ser definido detalladamente para poder saber qué es lo que se quiere decir o para que el mismo compilador sepa cómo traducir el programa. La definición de un lenguaje debe permitir a una persona o a un computador

determinar si el programa es valido y cuál es su significado (sintaxis y semántica). Para definir la sintaxis del lenguaje, se utiliza una notación BNF[‡‡] extendida.

En esta sección, se describe la gramática diseñada para el intérprete.

```
<eval>::=  <listaFunciones> <program> | <program> ;

<program>::=  PROGRAMA ID INICIO <acciones> FIN  ;

<funcion>::= LLAIZQ ID PARIZQ <listaVariables> PARDER  <listaInstr> <retorno>
                LLADER ;
<listaFunciones>::= <listaFunciones> <funcion> | <funcion>;
<listaVariables>::= <listaVariables> ID          |  ;

<listaInstr>::= INICIO <acciones> FIN
         | <instr> ;
<retorno>::= RESULTADO ':' IGUAL  ID  |  ;
<acciones>::= <acciones> <instr>  | ;
<expr>::= <expr> MAS <term>
             | <expr> MENOS <term>
              | <term>;
<term>::= <term> '*' <factor>
               | <term> '/' <factor>
              | <term> '%' <factor>
               | factor
```

[‡‡] Ver anexo: BNF es un metalenguaje de sintaxis y es usado para describir otro lenguaje.

```
;
<llamaFuncion>::=  ID  PARIZQ <listaArg>  PARDER ;


<factor>::= PARIZQ <expr> PARDER
| MENOS <factor> | NUM
                  | ID
          | ID CORIZQ <expr> CORDER
  | SENO PARIZQ <expr> PARDER
                | COS PARIZQ <expr> PARDER
                | TAN PARIZQ <expr> PARDER
                 | POW PARIZQ <expr> ',' <expr> PARDER
                 | RANDOM PARIZQ PARDER
                 | EXP PARIZQ <expr> PARDER
                 | SQRT PARIZQ  <expr> PARDER
                 | ABS PARIZQ <expr> PARDER
                 | <llamaFuncion>;


<listaArg>::=< listaArg> ';'  CADENA  |
                CADENA  |


                 expr |
                 <listaArg> ';' <expr>   |;
<instr>::= INICIALIZA <listaAsig> CONDFIN <listacond> REPITE <acciones>
            ACTUALIZA <listaAsig> FINBUCLE ;
```

*<listaAsignar>::= <listaAsignar>  <asignado>*

*| <asignado> ;*

*<instr>::= <asignado>;*

*<instr>::= LEER <value>;*

*<instr>::=  MOSTRAR  <texto> ;*

*<instr>::=  MOSTRARLN  <textoLN> ;*

*<instr>::= ASIGNAR ID;*

*<instr>::= ARREGLO ID CORIZQ <expr> ',' <tipo>  CORDER ;*

*<asignado>::= ID  CORIZQ <expr>  CORDER ':' IGUAL  <valor>*

*|  ID ':' IGUAL  <expr>  ;*

*<valor>::= <expr>*

*| CADENA ;*

*<instr>::=  CASO <listaClausulas> <else>  FINCASO;*

*<listaClausulas>::= <listaClausulas> <clausula>*

*| <clausula>  ;*

```
<else> :=: SINO  ':'  acciones   ';'  |  ;


<clausula>::=  <condicion> ':'  <acciones> ';' ;

<exprL> ::=  <expr> IGUAL <expr>

              | <expr> DISTINTO <expr>

              | <expr> MENOR <expr>

      | <expr> MENORIGUAL <expr>

      |<expr> MAYOR <expr>

      | <expr> MAYORIGUAL ;

<condicion>::= PARIZQ <exprL> PARDER;


<texto> ::= <texto> ',' ID

       | ID

              | CADENA

              | <texto> ',' CADENA;


<textoLN> ::= <textoLN> ',' ID

              | ID

              | CADENA

              | <textoLN> ',' CADENA  ;


<tipo> ::= NUMERO

    | PALABRA  ;


<listacond>::= <listacond> <condicion> | ;
```

### 3.3. La Semántica del intérprete ICH

Las reglas semánticas de un lenguaje definen el significado de sentencias válidas en ese lenguaje.

Aspectos léxicos

En general, el conjunto de caracteres es el conjunto Unicode. Las letras mayúsculas y minúsculas se toman como iguales en el lenguaje.

Espacios en blanco

El espacio simple, tabulador, y el cambio de línea corresponden a los espacios en blanco, cuya función es la de decidir la separación de los componentes léxicos del lenguaje. Por lo menos un espacio, es necesario para separar dos palabras clave o identificadores adyacentes, una palabra clave o identificador y un número.

Comentarios

Un comentario es cualquier texto (secuencia de caracteres) en el código fuente del programa que usualmente es incluido con fines aclarativos. El comentario no es una instrucción del lenguaje, simplemente es texto ignorado por el compilador. Par ello se ha seleccionado los caracteres " !- " como el delimitador de inicio y como fin de comentario " -! ".

Los comentarios pueden aparecer en cualquier punto en donde pueda venir un componente léxico y no requieren espacios en blanco a ningún lado para delimitarse.

Componentes léxicos

Los componentes léxicos son las palabras clave, identificadores, operadores y delimitadores.

Palabras Clave

Las palabras claves del lenguaje empiezan con el carácter (#) y son las que se muestran a continuación:

#programa, #inicio, #fin, #mostrar, #mostrarln, #resultado, #a-, #cadena, #numero, #asignar, #caso, #sino, #fincaso, #Inicializa, #CondicionFin, #repite, #Actualiza, #finBucle,

Identificadores

Un identificador es una secuencia de letras y, posiblemente, dígitos comenzando con una letra. Por letra, se entiende cualquiera de los caracteres Unicode correspondientes a letra, no hay una longitud límite preestablecida para los identificadores.

Operadores

Los operadores denotados con símbolos (no palabras clave) son los siguientes:

:=, <, >, <=, >=, +, -, =, *, /

Delimitadores

Los siguientes son componentes léxicos (no palabras clave) utilizados como delimitadores:

(, ), {, }, [, ], ;, , . .

Tipos

Tres tipos primitivos de datos atómicos:

- número
- palabras
- booleano

Acción

Por acción, se entiende, básicamente, una instrucción ejecutable. Una acción puede ser simple (***termine***, ***retorne***, por ejemplo) o puede ser compuesta en el sentido de que incluye, a su vez, otras acciones, lo que se denota dentro de su sintaxis como una o más apariciones de acciones (por ejemplo, la acción si).

Una acción también puede ser una declaración de variable, cuya vigencia queda limitada al segmento de acciones en que aparezca. Por ejemplo, si un segmento de acciones incluye una declaración de una variable x, entonces, la vigencia de x comenzará justo a continuación de su declaración hasta el final del correspondiente segmento.

Decisión Simple y multiple

- Condicional

  #Caso

  ( expresión ) : acciones ;

  ( expresión ) : acciones ;

  ( expresión ) : acciones ;

  …….

  #sino : acciones ;

#finCaso

Si alguna expresión de los casos indicados es igual a la expresión principal, se ejecutan las acciones dadas a continuación de los dos puntos hasta que se encuentre un finCaso. Las expresiones para los casos deben ser diferentes entre sí, de querer tener acciones por defecto solo es necesario utilizar la palabra reservada #sino en lugar de la expresión del caso.

Iteraciones

- Bucle

  #Inicializa

  acciones

  #CondicionFin

  accion

  #repite

  acciones

  #Actualiza

  acciones

  #FinBucle

El conjunto de acciones se ejecutará repetidamente en tanto que la condición de fin, que debe ser de tipo “booleano”, sea falsa. En el área de inicialización irán las variables necesarias para la ejecución del bucle, y luego de que se ejecuten las acciones a repetir se actualizaran dichas variables.

- Retorno de valor

  Esta acción permite retornar un valor de salida en un algoritmo y al mismo tiempo completar su ejecución.

- Asignación

  variable := expresión

  Esta expresión resulta igual a la evaluación de la expresión de la derecha promovida al tipo declarado para la variable, siempre y cuando haya compatibilidad. Una asignación tiene el efecto secundario de asignar a la variable el valor resultante de evaluar la expresión. El operador de asignación := tiene asociatividad por la derecha.

- Condición

  Sintaxis:

  - ( expresión operador expresión )

  Los operadores binarios sobre (expresión, expresión) se muestran a continuación. Las líneas separan las distintas precedencias en orden de menor a mayor. Todos estos operadores asocian por la izquierda excepto el de asignación que asocia por la derecha.

| Operador | Descripción |
|---|---|
| := | Asignación |

| | |
|---|---|
| = | Igualdad |
| /= | No igualdad |
| =/ | No igualdad |
| < | menor que |
| > | mayor que |
| <= | menor que o igual a |
| >= | mayor que o igual a |

### 3.4. Código Intermedio

En el modelo de análisis y síntesis de un compilador, el *front end* se encarga de traducir el código fuente del programa en una representación intermedia a partir del cual el *back end* pueda construir código final. Aquellos aspectos relacionados con la generación de código final se tratan de delegar al *back end* tanto como sea posible.

En el resto de la presente sección describiremos la estructura utilizada para la representación intermedia.

La estructura condicional CASO de ICH.

<instr>::= CASO <listaClausulas> <else> FINCASO;

<listaClausulas>::= <listaClausulas> <clausula>

| <clausula> ;

<else> :=: SI ':' acciones ';' | ;

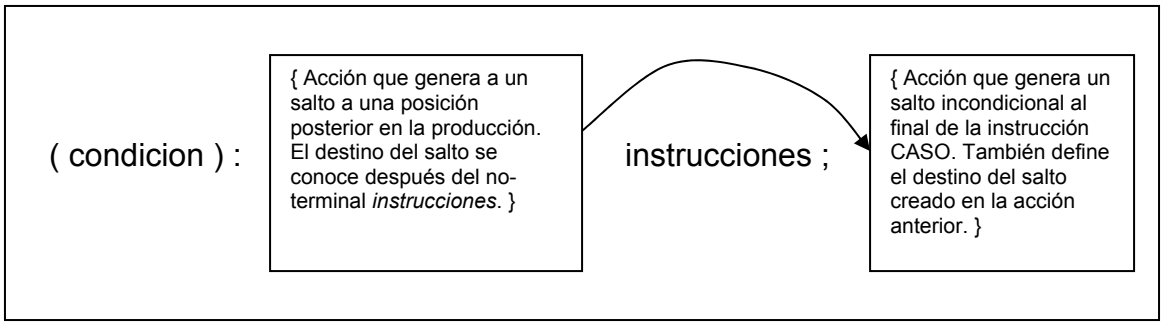



**Figura 14.**

**Acciones semánticas de la regla <clausula>**

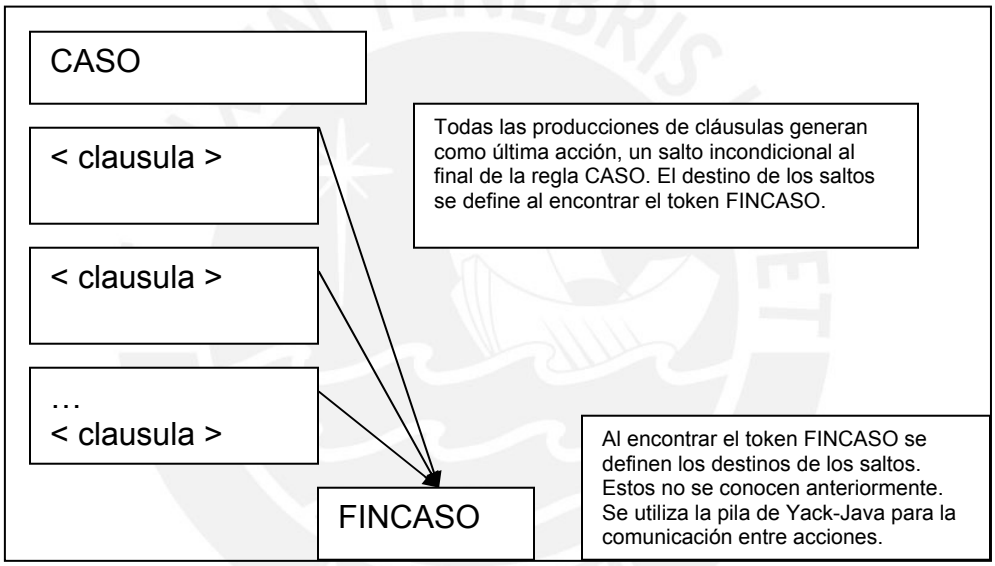



**Figura 15.**

**Acciones semánticas para la estructura CASO**

La estructura iterativa BUCLE de ICH

ICH provee la estructura bucle, una instrucción que realiza una división específica de las fases y tareas presentes en las instrucciones iterativas. A continuación se presenta la producción que permite generar cadenas del tipo BUCLE junto con las acciones involucradas en esta producción.

Producción que genera las cadenas para la estructura BUCLE:

<instr> ::= INICIALIZA <listaAsig>

CONDFIN <listacond>

REPITE <acciones>

ACTUALIZA <listaAsig>

FINBUCLE ;

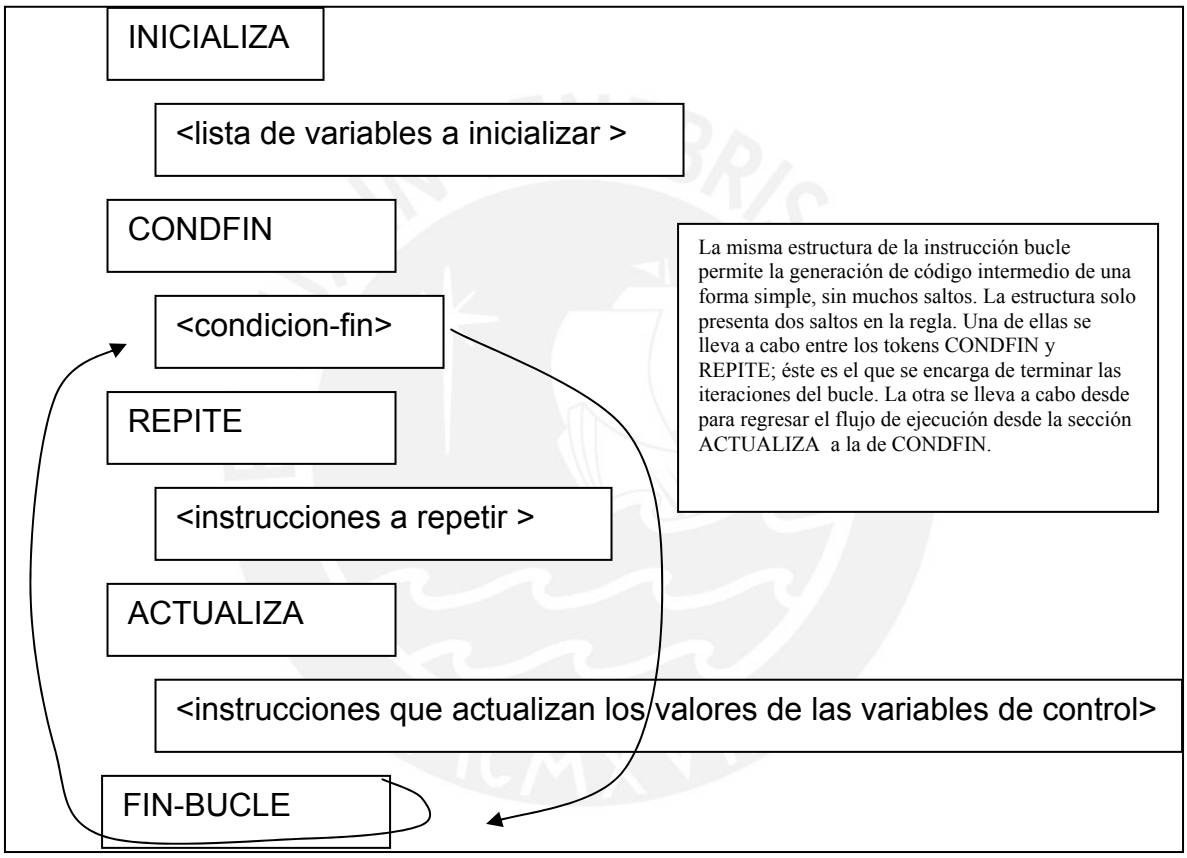



**Figura 16.**

**Acciones semánticas para la instrucción BUCLE**

### 3.4.1. Estructura utilizada

El código intermedio es almacenado en una tabla llamada Tabla de código ( class TtipoTablaCodigos { int op, a1, a2, a3 ; } ). Cada entrada en esta tabla representa una instrucción en la representación intermedia. Nuestra representación estará

basada en cuartetos, donde cada cuarteto se conforma de un operador (código de operación) y tres operandos.

En el siguiente contexto se entiende por “Operación binaria” a la operación que necesite de dos operandos para poder realizarse, mientras que “Operación unaria” es aquella que necesita de un operando. Las operaciones implementadas a través de este código intermedio se pueden clasificar en 5 grupos:

- Operaciones binarias con resultado: Las operaciones de este tipo almacenarán el resultado en el primer operando. La operación en sí será aplicada al segundo y tercer operando. Las operaciones implementadas de este tipo son:

| | |
|---|---|
| MENOR | Operación relacional “menor a”. |
| MENORIGUAL | Operación relacional “menor igual a”.. |
| IGUAL | Operación relacional “igual a”. |
| POTENCIA | Operación aritmética de potencia |
| % | Operación aritmética modular |
| MAYOR | Operación relacional “mayor que”. |
| MAYORIGUAL | Operación relacional “mayor o igual a”. |
| DISTINTO | Operación relacional “distinto a”. |
| ‘+’ | Operación aritmética de adición. |
| ‘-‘ | Operación aritmética de substracción. |
| ‘*’ | Operación aritmética de multiplicación. |
| ‘/’ | Operación aritmética de división. |
| REDONDEAR | Operación aritmética de redondeo |
| ‘%’ | Operación aritmética de residuo |

- Operaciones binarias sin resultado: Son aquellas que necesitan de dos operandos para poder realizarse, a pesar de no contar con un resultado en concreto. Por lo general se usarán el primer y tercer operando. Operaciones implementadas:

| | |
|---|---|
| SALTARV | Operación “saltar si verdadero”. Si el operando 1 es cierto se salta a la dirección indicada por el operando 3. |
| SALTARF | Operación “saltar si falso”. Si el operando 1 es falso se salta a la dirección indicada por el operando 3. |
| ASIGNAR | Operación de asignación. Copia el valor del operando 3 al operando 1. |

- Operaciones unarias con resultado: La operación se aplica al operando 3 y el resultado se guarda en el operando 1. Operaciones implementadas:

| | |
|---|---|
| CAMBIARSIGNO | Operación “menos unario”. |
| VALORABSOLUTO | Operación aritmética de valor absoluto |
| SENO | Operación trigonométrica de seno |
| COSENO | Operación trigonométrica de coseno |
| TANGENTE | Operación trigonométrica de tangente |
| EXPONENCIAL | Operación aritmética de exponencial |
| RAIZCUADRADA | Operación aritmética de raíz cuadrada |

- Operaciones unarias sin resultado: La operación se aplica al único operando válido, el cual es en ocasiones el operando 1 y en otras el operando 3. Operaciones implementadas:

| | |
|---|---|
| SALTAR | Salto incondicional a la dirección indicada por el operando |

| | |
|---|---|
| | 3. |
| INC | Incremento unitario del primer operando. |
| DEC | Decremento unitario del primer operando. |

- Otras: Son operaciones que aceptan un número variable de argumentos. Operaciones implementadas:

| | |
|---|---|
| IMPRIMIR | Escritura con número variable de operandos. |
| IMPRIMIRENTER | Escritura con número variable de operandos. Incluye cambio de línea. |
| LLAMADA | Realiza la llamada a una función. Guarda la información pertinente en un registro de activación y lo empuja dentro de la pila de registros de activación. Véase la sección “Registros de Activación” para los detalles de la implementación. |
| OBTENERARREGLO | Realiza la llamada a un procedimiento. Devuelve el valor correspondiente al índice solicitado del arreglo. |
| CREARARREGLO | Creación de un arreglo con las características especificadas. |
| ASIGNARARREGLO | Escritura en una celda de arreglo. |

### 3.4.2. Tabla de Símbolos

Un compilador utiliza las tablas de símbolos para llevar cuenta del alcance y la asociación de ciertos atributos con nombres para su posterior identificación. Cada vez que se encuentra un nombre se realiza una búsqueda en la tabla de símbolos.

Mientras tanto, la tabla sólo se modifica si se encuentra un nuevo nombre (en tal caso se agrega un nuevo registro o símbolo a la tabla) o si se descubre nueva información acerca de algún símbolo (nombre).

En nuestro caso, cada entrada en la tabla de símbolos consta de siete campos descritos a continuación:

```
class TtipoTablaSimbolos  {
        String nombre;
        int token;
        double valor ;
        int iNumLinea;
        int nivel;
        int direccion;
        int tipoDato; }
```

- Nombre del símbolo: nombre que caracteriza a la variable, función o procedimiento.
- Tipo de dato: determina el tipo de dato correspondiente.
- Código de token: es el token que caracteriza a cada identificador, función o procedimiento.
- Valor: el valor que corresponde a la variable.
- Número de línea: la línea de código donde se encuentra el lexema.
- Nivel: determina si es parte del programa principal o de alguna función o de algún procedimiento.
- Dirección: nos da la referencia al registro de activación.

### 3.4.3. Registros de Activación

Es aquí donde se guarda la información necesaria para la ejecución de un procedimiento o de una función (nos referiremos a ambos en general como "operaciones"). Para realizar una llamada a una operación cualquiera se utilizan Registros de Activación (*Activation Records*) o Marcos (*Frames*) [AHO].

Antes de proceder a explicar la estructura de los registros de activación introduciremos la estructura que guarda la información de las operaciones declaradas, la cual llamaremos TtipoRegistroActivacion.

```
public class TtipoRegistroActivacion{
    TtipoTablaSimbolos  GTablaSimbolos[];
    TtipoTablaCodigos  GTablaCodigos[];
    int cx;
    int nSim;
    double resultado;TOperacion: }
```

- Resultado: valor de retorno de la función
- GTablaSimbolos: tabla de símbolos de la función o procedimientos ( variables locales)
- nSim: cantidad de entradas en la tabla de Símbolos de la función.
- GTablaCodigos: tabla de códigos referente a las acciones realizadas dentro de la función o del procedimiento.
- Cx: cantidad de entradas en la tabla de código de la función o procedimiento.

### 3.4.4. Algunos aspectos de la implementación a resaltar

- Se inicia el proceso de compilación leyendo el archivo, si se encuentra una “{“ , símbolo de que una función empieza, entonces los datos se cargan a un registro de activación y se hace referencia a este registro en la tabla de símbolos. El análisis sigue de la misma manera por cada función que encuentre.
- Al encontrar el token “#PROGRAMA” se cargan los datos en la tabla de símbolos principal, y tabla de códigos según corresponda.
- Por cada vez que se encuentre en el programa principal una llamada a función se hace una copia del registro de activación correspondiente a la función.
- Al iniciar el proceso de ejecución se empieza a ejecutar las instrucciones de la tabla de código del programa principal al encontrar una llamada a función se va al registro de activación correspondiente ( el cual ya tiene los datos) y se procede a ejecutar las instrucciones de dicho registro de activación y así hasta que se termine con las instrucciones, tanto de los registros de activación como del programa principal.

### 3.5. Opciones del Lenguaje ICH

A continuación se exponen las opciones del lenguaje, su manejo y un ejemplo de cada una de ellas.

- Programa

```
#programa !- nombre -!
#inicio
!- instrucciones -!
#fin
```

Un programa siempre debe empezar con la palabra #programa seguido por el nombre del programa y las instrucciones delimitadas por las palabras reservadas #inicio y #fin.

Ejemplo:

#programa ProgramaPrueba

#inicio

!- instrucciones -!

#fin

- Mostrar

  #mostrar !- variable o cadena -!

  La palabra reservada #mostrar imprime en la ventana de salida el valor de la variable o cadena, se pueden mostrar varios valores a la vez si es que dichos valores son separados por comas

  Ejemplo:

  #mostrar “ejemplo de mostrar” , var1 , var2

  #mostrar var1

  #mostrar “ejemplo”

- MostrarLn

  #mostrarln !- variable o cadena -!

  La palabra reservada #mostrarln imprime en la ventana de salida el valor de la variable o cadena junto con un cambio de línea. También puede mostrar varios

valores a la vez separando los valores por medio de comas.

Ejemplo:

#mostrarln “ejemplo de mostrarln” , var1 , var2

#mostrarln var1

#mostrarln “ejemplo”

- Funciones

  { !- nombre de la función -! ( !- Argumentos )

  #inicio

  !-instrucciones-!

  #fin

  #resultado:= !- variable -! }

Todas las funciones deben definirse antes del programa principal y antes de la primera invocación hecha, es decir, en ningún momento se puede llamar una función definida posteriormente en el código.

Toda función está compuesta de una cabecera y un cuerpo (lista de instrucciones). La cabecera consiste en el nombre de la función seguida de la lista de argumentos separados por comas y encerrados entre paréntesis.

Las instrucciones de la función deben delimitarse por las palabras reservadas #inicio y #fin.

Toda función debe devolver un resultado. Este se puede definir asignándoselo a la palabra reservada #resultado, la cual siempre debe ir después del cuerpo de la función.

Ejemplo:

```
{ Restar ( h, g )
  #inicio
        j:=h-g
  #fin
  #resultado:= j
}

#programa PruebaFuncion
#inicio
        a:=4
        b:=5
        c:=Restar( a , b)
        #mostrar “El resultado de restar ” , a , “ y ”, b , “ es ”, c
#fin
```

- Asignar

        #asignar !- variable -!

La palabra reservada #asignar permite leer valores numéricos a partir del teclado para una o más variables. En el caso de ser varias variables éstas se deben separar por comas.

Ejemplo:

```
#asignar  var0, var1 , var2
#asignar  var1
```

- Arreglo

  #a- !-nombre del arreglo -! [ !-longitud-! , !-tipo:  #cadena o #numero -! ]

  La palabra reservada #a- permite crear un arreglo ya sea de palabras, usando la palabra reservada #palabra al momento de crear el arreglo o de números, usando la palabra reservada #numero.

  Ejemplo:

```
!- de tipo numero- !
#a- ArregloPrueba [ 10 , #numero ]
ArregloPrueba[2]:=9
d:=ArregloPrueba[2]
!- de tipo cadena -!
#a- ArregloPrueba [ 10 , #cadena ]
ArregloPrueba[9]:=“hola”
d:=ArregloPrueba[9]
```

- Caso

  #caso

  ( !-condición-! ): !-instrucciones-! ;

  ( !-condición-! ): !-instrucciones-! ;

  ( !-condición-! ): !-instrucciones-! ;

  #fincaso

  La instrucción caso permite dividir el flujo del programa en varias ramas. Las condiciones deben ir encerradas entre paréntesis y las instrucciones asociadas a cada condición deben ir limitadas por el caracter dos puntos (:) y por un punto y coma (;).

  Ejemplo:

  #caso

  ( x > 10) : #mostrarln "mayor que diez";

  ( x = 10) : #mostrarln "igual a diez";

  ( x < 10) : #mostrarln "menor que diez";

  #fincaso

- Caso-Sino

  #caso

  ( !-condición-! ): !-instrucciones-! ;

  ( !-condición-! ): !-instrucciones-! ;

  #sino : !-instrucciones-! ;

#fincaso

Muy similar a la instrucción CASO, pero con la diferencia de la etiqueta #sino. De no cumplirse ninguna condición anterior, se ejecutarán las instrucciones asociadas a esta etiqueta. Las instrucciones también deben ir limitadas por el caracter dos puntos (:) y por un punto y coma (;).

- Ejemplo:

  #caso

  ( x > 10) : #mostrarln "mayor que diez";

  ( x = 10) : #mostrarln "igual a diez";

  #sino : #mostrarln "menor que diez";

  #fincaso

- Bucle

  #Inicializa

  i:=1;  f:=1;  n:=5

  #CondicionFin

  ( i>n )

  #repite

  f:=f*i

  #mostrarln i," - ",f

  #Actualiza

  i:=i+1

  #finBucle

Esta instrucción consta de 4 secciones. La sección de inicialización de variables, la sección CondicionFin, la sección de instrucciones a repetir y la sección de Actualización de índices.

Hola Mundo usando ICH

- #programa HolaMundo

  #inicio

  #mostrarln “Hola Mundo”

  #fin

  La salida seria:

  Hola Mundo

- #programa Sumando

  #inicio

  !- lee dos valores, los suma, guarda el resultado y lo muestra -!

  #asignar a,b

  suma:=a+b

  #mostrarln a,”+”,b,”=“,suma

  #fin

  Con a=2 y b=3 la salida seria:

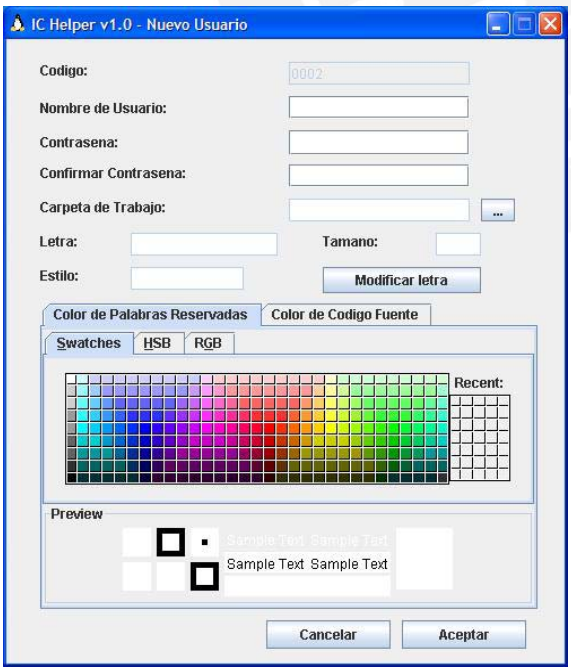
  2+3=5

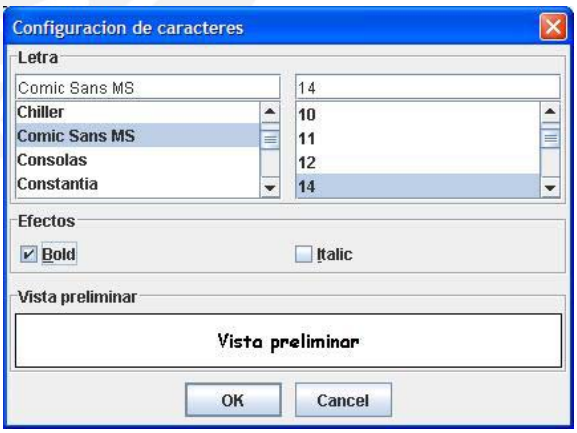
### 3.6. IMPLEMENTACION DEL ENTORNO ICHELPER

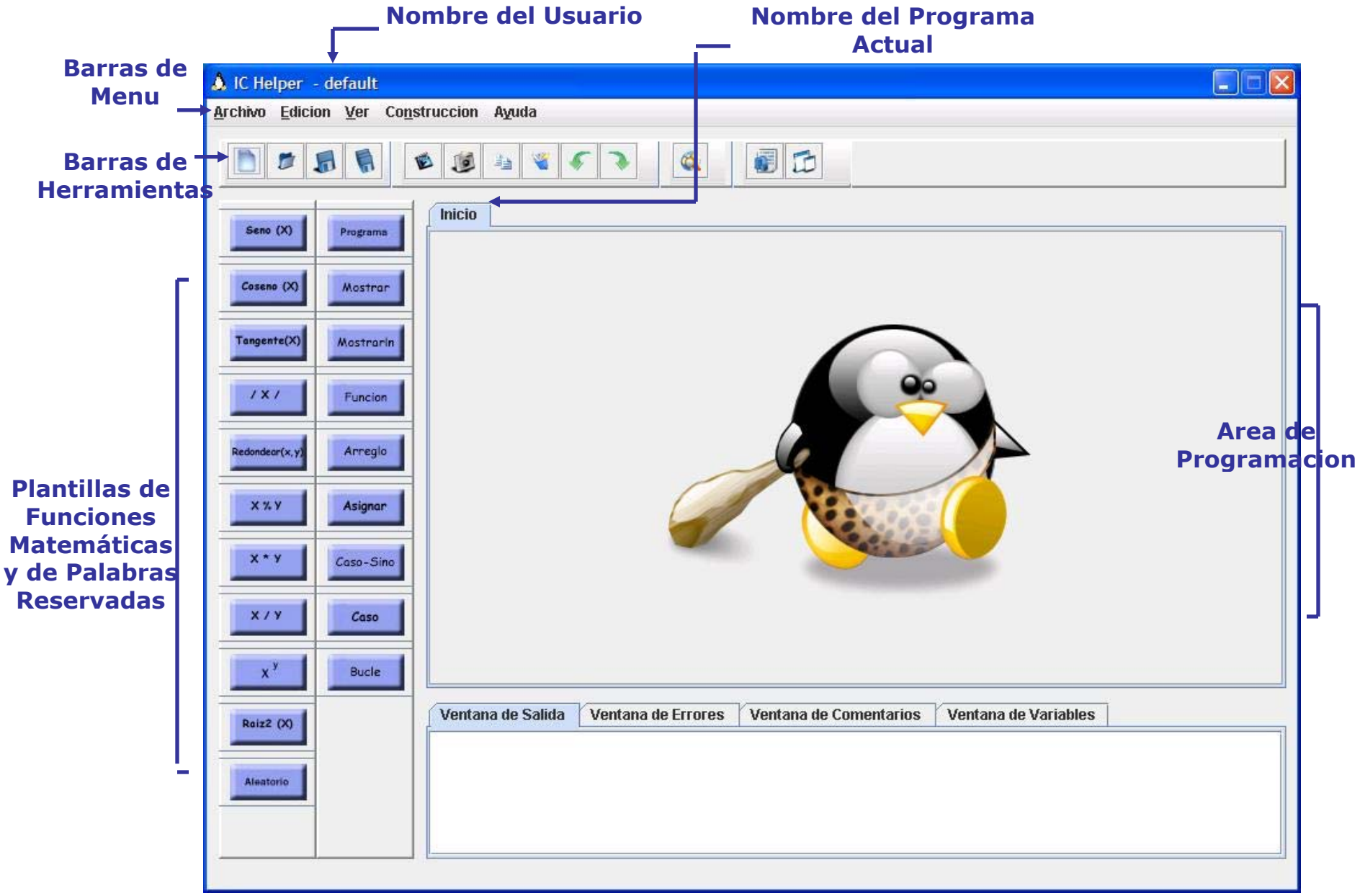
ICHelper se desarrolla utilizando java, como un entorno en el cual se pueda compilar y ejecutar programas escritos en ICH y a su vez proporcione herramientas para que el alumno adquiera buenos hábitos de programación, tales como:

- Siempre comentar el código creado
- Guardar sus programas en una carpeta de trabajo
- Al inicio de la práctica contar con las estructuras del lenguaje y sus características, haciendo uso de las plantillas de palabras reservadas y de funciones matemáticas.
- Reconocer las palabras propias del lenguaje con un color en especial configurado por el mismo usuario.

Pantallas del entorno





## 4. OBSERVACIONES, CONCLUSIONES Y RECOMENDACIONES

El trabajo realizado es un intento por producir un lenguaje en español de alto nivel, estructurado y funcional y que cuente con un entorno de desarrollo amigable pensado exclusivamente para el aprendizaje de lenguajes de programación. Desde la concepción del proyecto se ha tratado de considerar las necesidades más comunes e importantes que tienen las personas con poca o ninguna experiencia en programación, así como los errores más comunes que se cometen, tales como la falta de inicialización y actualización de las variables al momento de la creación de bucles. Además también se han considerado aspectos pedagógicos cruciales como palabras reservadas con sentido, relativamente fáciles de aprender y recordar; un conjunto reducido de instrucciones que, sin embargo, cuentan con semánticas flexibles; una sintaxis simple y directa que no es obstáculo alguno para el alumno; algunas funciones que obligan y guían el proceso de desarrollo como la descripción de programas y subprogramas al momento de editarlos; finalmente, también se tienen otras opciones del entorno que facilitan el ingreso, ejecución y salida de resultados.

Futuros trabajos sobre el presente pueden incluir arreglos bidimensionales, manejos de estructuras, extensiones para depuración de código; paneles para desarrollar diagramas de flujo mediante plantillas; generación automática de código a partir de ésta última; conexión por red para corrección de los trabajos en tiempo real, así el alumno puede avanzar por etapas siempre recibiendo el visto bueno de su profesor o asistente de docencia en cada fase del desarrollo del problema; soporte de librerías estáticas en ICH como por ejemplo librerías gráficas que implementen funciones de dibujo de rectas, círculos y otras funciones matemáticas, etc.

**BIBLIOGRAFIA**

[AHO] Alfred V.Aho, Ravi SEIT y Jeffrey D.Ullman [1988]. “Compilers Principles, Techniques and Tools”. Addison Wesley.

[RITCHIE] Brian Kernighan & Dennos Ritchie [1985]. “El lenguaje de programación C”. Prentice Hall.

[GLENN] J. Glenn Brookshear [1993]. “Teoría de la Computación. Lenguajes formales, autómatas y complejidad”. Addison-Wesley Iberoamericana.

[KELLEY] Dean Kelley [1995]. “Teoría de Autómatas y Lenguajes Formales”. Prentice Hall.

[CooperTorczon] Keith D. Cooper & Linda Torczon, “Engeneering a compiler”, Editorial Elsevier, 2004
[SCHEME] www.drscheme.org

[DIAZJ] Enseñando programación con C++: una propuesta didáctica por Jorge Díaz, Universidad de Oriente, Departamento de Computación, Santiago de Cuba 90500, Cuba http://www.fi.uba.ar/laboratorios/lie/Revista/Articulos/030307/A2Jun2006.pdf

[BAEZA] BAEZA YATES, Ricardo. Diseñemos todo de nuevo: Reflexiones sobre la computación y su enseñanza. En: Revista Colombiana de Computación. Vol. 1. No. 1. Diciembre de 2000. ISSN 1657-2831.
http://www.unab.edu.co/editorialunab/revistas/rcc/pdfs/r11_art1_c.pdf

[KAASBOLL2000], Jens. Learning and Teaching Programming. Department of Informatics. University of Oslo. Norway. 2000.
http://citeseer.ist.psu.edu/cache/papers/cs/16022/http:zSzzSzwww.ifi.uio.nozSz~jensjzSzlearningAndTeaching.pdf/learning-and-teaching-programming.pdf

[KAASBOLL] Exploring didactic models for programming, Jens J. Kaasbøll http://citeseer.ist.psu.edu/cache/papers/cs/16022/http:zSzzSzwww.ifi.uio.nozSz~jensjzSzNIK98.pdf/exploring-didactic-models-for.pdf

[DEITEL] Deitel, Java, How to Program, 6ta edicion, 2006 editorial Pearson Education, Inc.

[Kenneth] Lenguajes de Programación, Principios y practica, Kenneth C. Louden.Editorial thomson

[Beekman] Introducción a la computación, George Beekman, editorial Pearson Education, Inc.

[DICC] Diccionario de Informatica,editado por Cultural S.A.

[Kolling] KOLLING, Michael. The Design of an Object-Oriented Environment and Language for Teaching. PhD. Thesis. Basser Department of Computer Science. University of Sidney. 1999. http://www.cs.kent.ac.uk/pubs/1999/2171/index.html

[Robinson] ROBINSON, Peter. From ML to C via Modula-3 an approach to teaching programming. Dic. 1994.
http://www.cl.cam.ac.uk/~pr10/publications/plc93.pdf

[WikiJava] Lenguaje de programacion Java
http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_Java

[Gaston] Componentes de la PC, Sergio Ernesto Gaston Perez
http://www.mailxmail.com/curso/informatica/componentespcs/capitulo18.htm

[DESWEB] Manual de iniciación a la programación, Desarrolladores web
http://www.desarrolloweb.com/articulos/2477.php

[WikiScheme] Scheme http://es.wikipedia.org/wiki/Scheme

[WikiPascal] Lenguaje de Programación Pascal

http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_Pascal

[UPM-ACM] Introduccion al lenguaje C, Facultad de Informática, Universidad Politécnica de Madrid, Capítulo de Estudiantes ACM
MADRID (SPAIN), Noviembre 2004
http://acm.asoc.fi.upm.es/documentacion/c2005/manual/node4.html

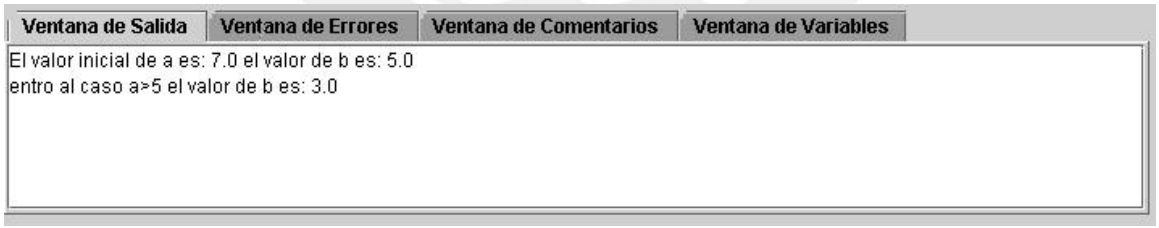
[ZatorC++] Introduccion a la POO, Curso C++, Tutoriales http://www.zator.com

**EJEMPLOS**

1. Prueba de la estructura caso con uso del #sino y sin el:

a. #Caso sin #sino

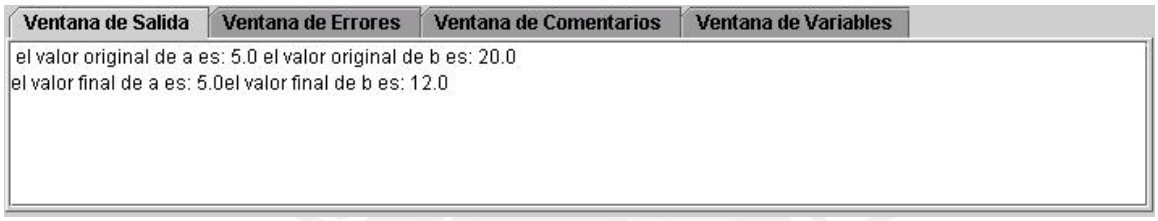
```
#programa PruebaCaso
    #inicio
        a:=7
        b:=5
        #mostrarln "El valor inicial de a es: ",a , " el valor de b es: ",b
        #caso
            ( a>5) :  b:=3 #mostrarln "entro al caso a>5 el valor de b es: ",b;
            ( a<=5) :  b:=15 #mostrarln "entro al caso a>5 el valor de b es: ",b;
        #fincaso
#fin
```




b. #Caso con #sino

```
#programa PruebaCasoConSino
  #Inicio
    #caso
            ( a>5) :  b:=3 ;
```

```
            ( a<5) :  b:=14 ;
            #sino: b:=12;
        #fincaso
        #mostrarln "el valor final de a es: ",a
        #mostrarln "el valor final de b es: ",b
    #fin
```

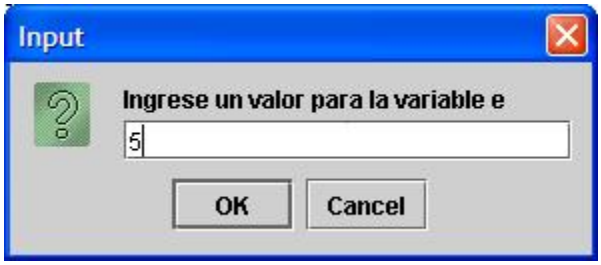


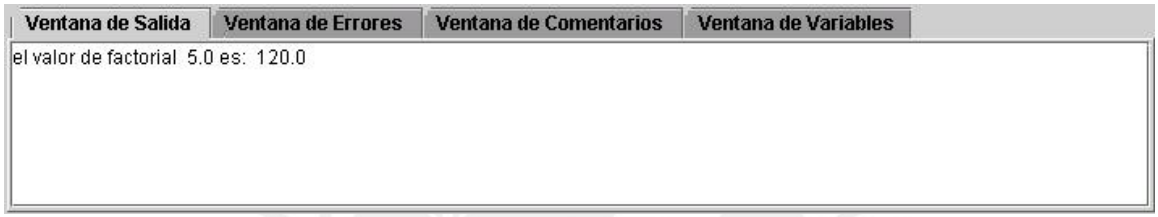

2. Prueba de funciones

    a. Factorial (Factorial.ich)

```
{ factorial(n)
        #caso
            (n>1):  d:= n * factorial ( n-1 );
            (n=1):  d:=1;
        #fincaso
  #resultado:= d
}
#programa ProbandoFactorial
#inicio
        #asignar e
        r:=3
        f:=factorial(e)
```

```
        #mostrar  "el valor de factorial  " , e , " es:  " , f
#fin
```

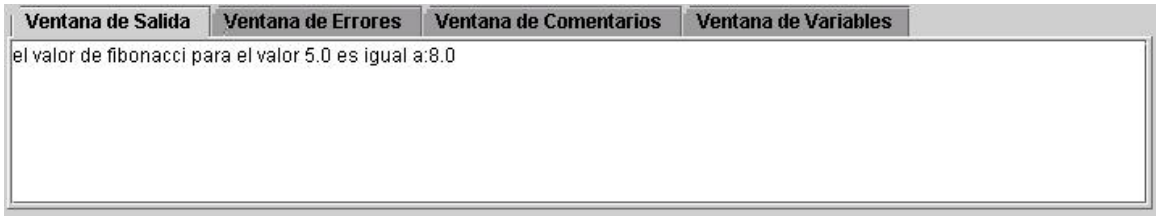



b. Fibonacci  (Fibonacci.ich)

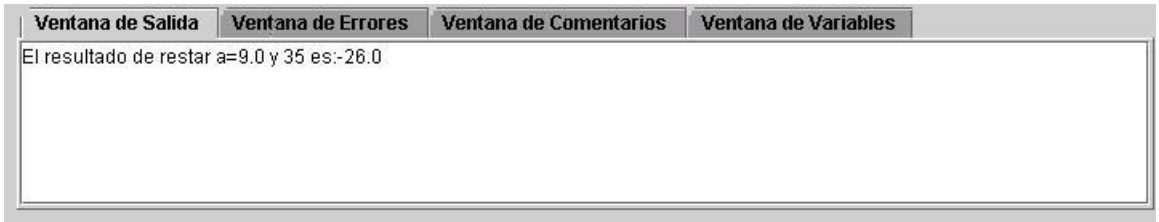
```
{fib(n)
        #caso
                (n>1): d:= fib ( n-1 ) + fib( n-2 );
                (n=1): d:=1;
                (n=0): d:=1;
        #fincaso
  #resultado:= d
}
#programa PruebaFibonacci
  #inicio
        e:=5
        f:= fib(e)
```

```
#mostrar "el valor de fibonacci para el valor " ,  e , " es igual a:", f
#fin
```



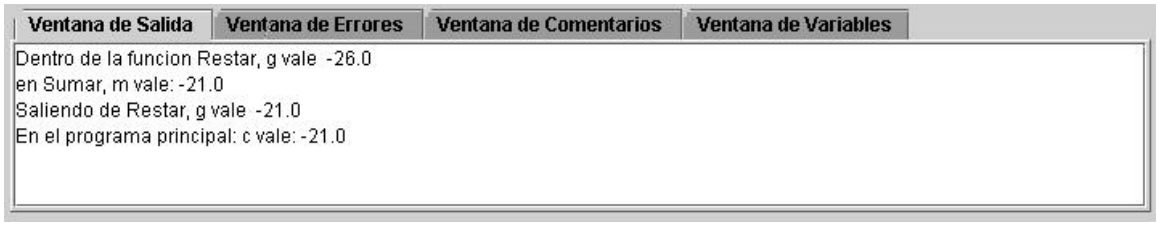

c. Prueba de llamada recursiva (Usando un nivel – una llamada a función desde el programa principal)

```
{ Restar(  h g)
        p:=h-g
    #resultado:= p
}
#programa LlamadaRecSimple
  #inicio
        a:=9
        c:=Restar( a ;35)
        #mostrar "El resultado de restar a=", a , " y 35 es:", c
  #fin
```



d. Usando 2 niveles – una llamada a funcion desde otra funcion llamada desde el programa principal

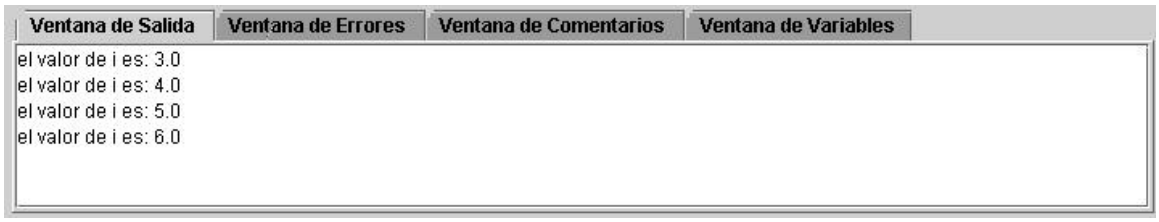
```
{Sumar( n )
  #inicio
        m:=n+5
        #mostrarln "en Sumar, m vale: " , m
  #fin
  #resultado:= m
}
{ Restar(  h g)
  #inicio
        g:=h-g
        #mostrarln "Dentro de la funcion Restar, g vale  " , g
        g:=Sumar( g )
        #mostrarln "Saliendo de Restar, g vale  " , g "
  #fin
  #resultado:= g
}
#programa LlamadaRecursiva
  #inicio
        c:=Restar( 9 ;35)
        #mostrarln "En el programa principal: c vale: ", c
  #fin
```




3. Prueba Bucles

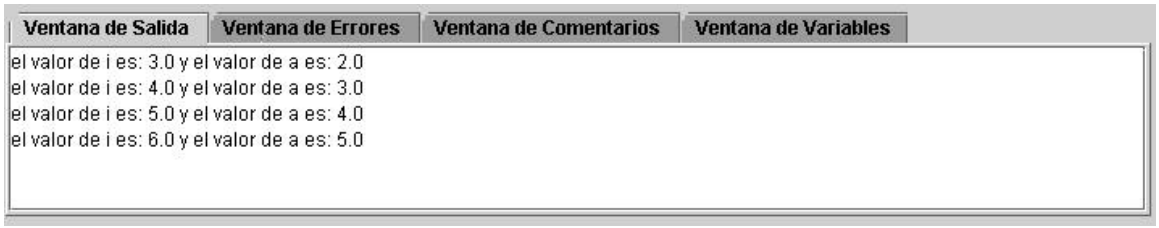
a. Bucle simple

```
#programa BucleSimple
  #inicio
        #inicializa
          i:=3
        #condicionfin
           ( i=7)
        #repite
          #mostrarln "el valor de i es: ", i
          #actualiza
          i:=i+1
        #finbucle
  #fin
```



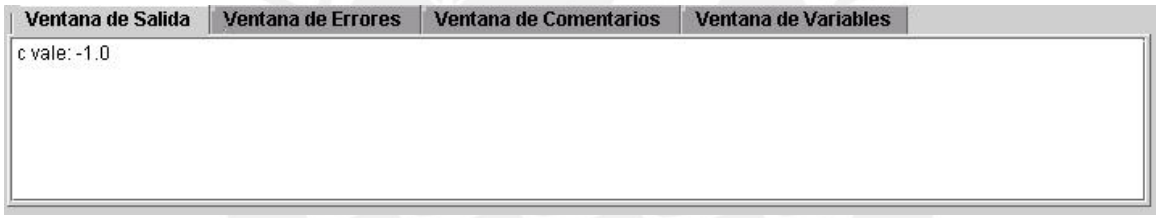
b. Bucle con 2 condiciones

```
#programa PruebaBucleCon2Cond
  #inicio
        #inicializa
          i:=3
          a:=2
        #condicionfin
          ( i=7)
          (a>4)
        #repite
          #mostrar "el valor de i es: "  ,i
          #mostrarln " y el valor de a es: " ,a
        #actualiza
          i:=i+1
           a:=a+1
        #finbucle
 #fin
```



4. Funciones con números por argumentos

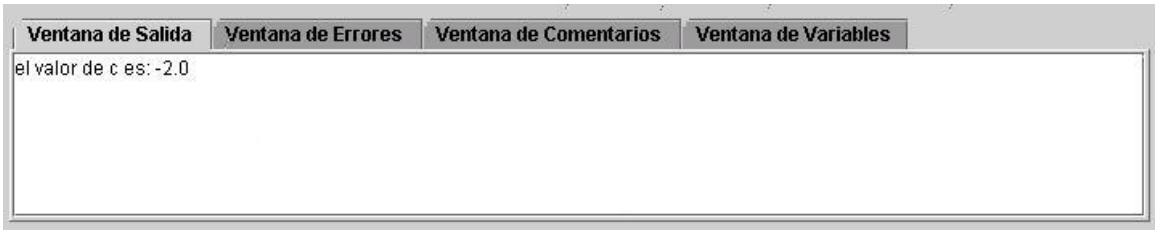
```
{restar( g h )
        g:=g-h
 #resultado:= g
}
#programa FuncionesNumeroxArgumento
 #inicio
        a:=4
        b:=5
        c:=restar( 4 ;5)
        #mostrar “c vale: ”, c
 #fin
```




5. Funciones con números y variables por argumentos además de hacer llamadas a la función definida desde la misma función

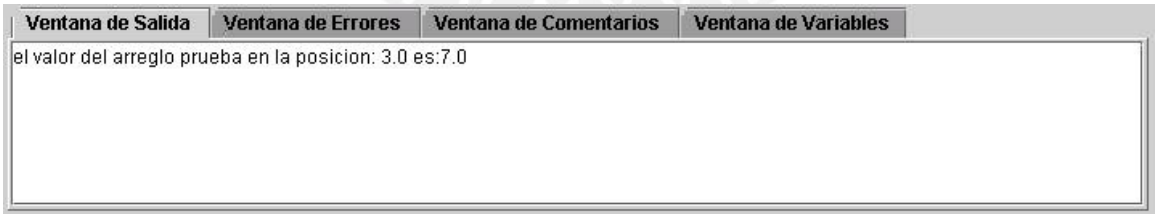
```
{ restar( g h )
 #inicio
        p:=restar(g 2)
        g:=g-h-p
 #fin
 #resultado:= g
}
#programa maranga
 #inicio
        a:=4
        b:=7
```

```
        c:=restar( b 5)
        #mostrar “el valor de c es: ”, c
    #fin
```




6. Arreglos de tipo numero

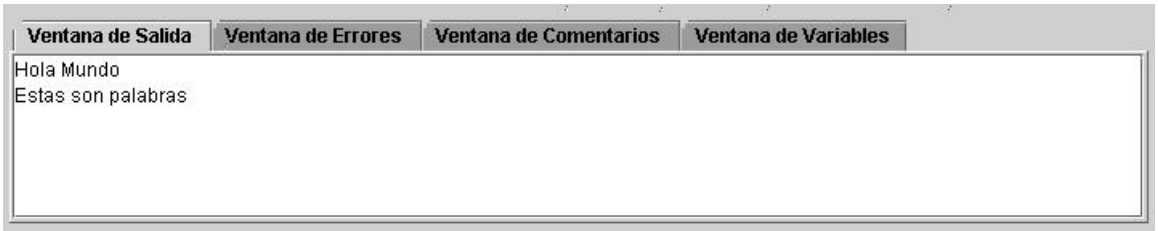
```
    #programa arreglosNumericos
     #inicio
        pos:=3
        #a-prueba[8,#numero]
        prueba[pos]:=7
        b:=prueba[pos]
        #mostrar  "el valor del arreglo prueba en la posicion: ", pos, " es:", b
     #fin
```




7. Arreglos de tipo palabra

```
    #programa arregloPalabras
      #inicio
        #a-prueba[8,#cadena]
        prueba[5]:="Estas son palabras"
        d:=prueba[5]
```

```
    #mostrarln  "Hola Mundo"
    #mostrar  d
#fin
```



**GLOSARIO [DICC]**

**Compilador:**

Es un programa que traduce un lenguaje de alto nivel al lenguaje de máquina de una computadora, tomando como entrada un texto escrito en un lenguaje, llamado fuente, y da como salida otro texto en un lenguaje denominado objeto el cual esta en un lenguaje de bajo nivel (ensamblador o código de máquina).

Según va ejecutando la traducción, coteja los errores hechos por el programador. Traduce un programa una sola vez, y es cinco veces más rápido que los programas intérpretes.


Ejemplos: ALGOL, BASIC, COBOL, FORTRAN, PASCAL y PL/1.

**Intérprete:**

Es un programa que traduce un lenguaje de alto nivel al lenguaje de máquina de una computadora. El programa siempre permanece en su forma original (programa fuente) y traduce cuando está en la fase de ejecución instrucción por instrucción.

Los intérpretes en lugar de producir un Lenguaje objetivo, como en los compiladores, lo que hacen es realizar la operación que debería realizar el Lenguaje origen. Un intérprete lee el código como esta escrito y luego lo convierte en acciones, es decir, lo ejecuta en ese instante.

Existen lenguajes que utilizan un Intérprete, como por ejemplo JAVA, y su intérprete traduce en el instante mismo de lectura, el código en lenguaje máquina para que pueda ser ejecutado.


Ejemplo: BASIC

**Algunas de las ventajas de compilar frente a interpretar son:**

- Se compila una vez; se ejecuta muchas veces.
- La ejecución del programa objeto es mucho más rápida que si se interpreta el programa fuente.
- El compilador tiene una visión global del programa, por lo que la información de mensajes de error es más detallada.

- Un archivo compilado puede ser distribuido fácilmente conociendo la plataforma, mientras que un archivo interpretado no funciona si no se tiene el intérprete.

Por otro lado, algunas de las ventajas de interpretar frente a compilar son:

- Un intérprete necesita menos memoria que un compilador. En las primeras etapas de la informática eran más abundantes dado que los ordenadores tenían poca memoria.
- Permiten una mayor interactividad con el código en tiempo de desarrollo.
- En algunos lenguajes (Smalltalk, Prolog, LISP) está permitido y es frecuente añadir código según se ejecuta otro código, y esta característica solamente es posible implementarla en un intérprete.
- Un intérprete traduce el programa cuando lo lee, convirtiendo el código del programa directamente en acciones.
- Dado cualquier programa se puede interpretarlo en cualquier plataforma (sistema operativo), en cambio el archivo generado por el compilador solo funciona en la plataforma en donde se lo ha creado.

**Yacc:**

Acrónimo de Yet Another Compiler- compiler. En el entorno unix, dícese del comando que se utiliza para generar compiladores. Los Yacc están dotados de una serie de reglas gramaticales LR, que son las que definen la sintaxis de un lenguaje, y con estas reglas se genera un programa fuente de análisis sintáctico en forma de autómata escrito en C, Este programa se puede compilar e incluirse en otros programas para realizar un análisis sintáctico. Con todo ello se ha generado un lenguaje a partir de su propia

gramática. En el desarrollo de esta tesis se utilizó un yacc para el lenguaje de programación java “Yack-Java” desarrollado por el Dr. Maynard Kong.

**Parser:**

Descomponer. Procedimiento de interpretación de un mandato o de una instrucción basándose en la descomposición y el análisis de cada una de sus partes.

**Lenguaje de programación:**

Se define un lenguaje de programación como la aplicación o el programa que facilita al usuario la introducción a la escritura de una serie de mandatos o instrucciones con la que se define una tarea en concreto y que serán posteriormente traducidas a un formato entendible por el microprocesador. Los lenguajes de programación pueden ser de alto nivel, que contienen instrucciones fáciles de entender y de usar y de bajo nivel, que son más complejos y lentos de programar aunque más rápidos de ejecución

**Programa estructurado:**

Dícese del programa que ha sido creado bajo los cánones que marca la programación estructurada, es decir, una programación basada en un conjunto de estructuras y normas fijas que hacen que el programa pueda ser seguido por cualquier programador.

**Programación elegante**

En argot informático, dícese del código fuente que ha sido bien estructurado y se han utilizado el menor número posible de instrucciones. Esto hace que la lectura del programa sea fácil y su ejecución eficiente.