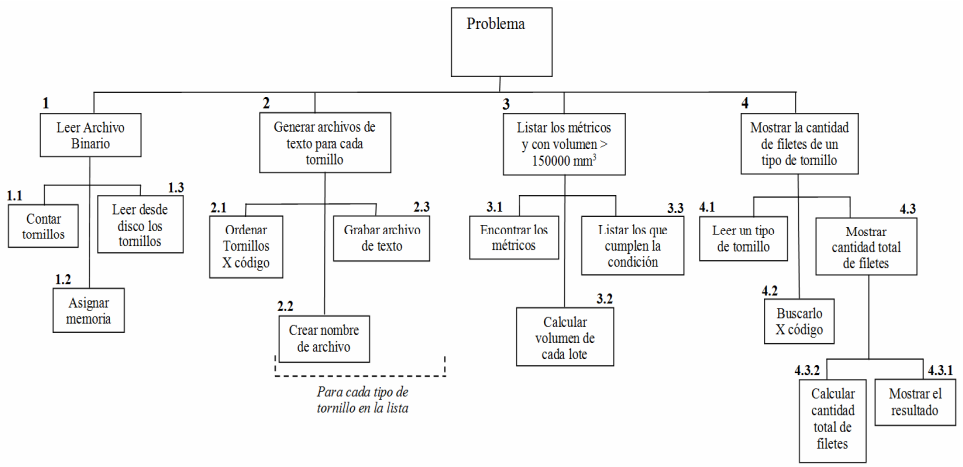
Programación de Computadoras para Estudiantes de Ingeniería



Eduardo M. Zavalla

2020

Programación de Computadoras para Estudiantes de Ingeniería

Eduardo M. Zavalla

Mg. en Ingeniería
Instituto de Automática
Departamento de Electrónica
Facultad de Ingeniería
Universidad Nacional de San Juan

2020

Programación de Computadoras para Estudiantes de Ingeniería. *Eduardo M. Zavalla*

Zavalla, Eduardo Mario
Programación de computadoras para estudiantes de ingeniería / Eduardo Mario Zavalla ; editado por Angel César Veca. - 1a edición especial - San Juan : Angel César Veca, 2020.
Libro digital, PDF/A

Archivo Digital: descarga y online
ISBN 978-987-86-4062-4

1. Lenguajes de Programación. I. Veca, Angel César, ed. II. Título.
CDD 005.13

ISBN 978-987-86-4062-4

Diseño de la Portada: Eduardo M. Zavalla
Imagen: Diagrama descendente de la descomposición Top-Down de un problema.

Libro de edición argentina

INDICE

Prólogo del Editor

Esta obra es el resultado de más de 20 años dedicados por el autor a la docencia, en el ámbito de la Facultad de Ingeniería, Universidad Nacional de San Juan. Está compuesta por siete capítulos que abarcan los temas de: los problemas y su solución, programación de computadoras, estructura de datos, subprogramas y ejercicios.

Por medio de ejemplos cotidianos muy simples, muestra como la aplicación de dos conceptos muy antiguos; Top-Down y Bottom-Up, se puede trabajar, identificando y dividiendo el problema en partes simples o módulos, que permiten llegar a la solución optima del problema a resolver. Además, se logra que al dividir un problema complejo, los pequeños problemas resultantes son analizados en forma individual de manera de buscar similitudes con problemas resueltos anteriormente y de esta forma, reutilizar soluciones ya probadas.

En todos los capítulos se emplean diagramas en bloques que permiten al lector poder visualizar rápidamente el problema en estudio. Desde el principio, se trabaja con la noción de nodo y nivel, a partir de los cuales se arriba finalmente al esquema denominado árbol, que se puede interpretar como el mapa de la resolución del problema. Esta particularidad de trabajo, se mantiene a lo largo de toda la obra, con las correspondientes adaptaciones.

En síntesis, se trata de una obra que encara de forma moderna y sencilla los conceptos básicos que forman el cimiento de la programación de computadoras.

Angel Veca
San Juan, Argentina, marzo de 2020

Prólogo del autor

Durante todo el tiempo que llevo enseñando a programar computadoras a los estudiantes de ingeniería, siempre he observado que las mayores dificultades que encuentran los alumnos no se relacionan con las estructuras propias de los lenguajes de programación ni con los bloques funcionales básicos, sino que se remiten al entendimiento profundo de los problemas a resolver y al análisis e implementación de las etapas necesarias para llegar a soluciones apropiadas.

Quizás los comentarios sobre el *Pensamiento Computacional*, propuesto por Jeannette Wing en 2006, sean los mas relevantes a considerar a la hora de formar alumnos de ingeniería que sean capaces de entender y resolver problemas de diversa naturaleza, y en ese contexto, este libro intenta proporcionar las herramientas básicas necesarias para abrazar este tipo de pensamiento en las etapas mas tempranas de la formación, en el entendimiento que esta habilidad ayudará al alumno a lo largo de toda su carrera académica y profesional.

Y así como es imposible aprender a conducir un automóvil solamente leyendo un libro sobre manejo, también es imposible aprender a programar solo leyendo un libro sobre programación pero sin realizar programas. Por este motivo se utiliza un lenguaje de programación llamado **LPP** (**L**enguaje de **P**rogramación para **P**rincipiantes) que fue creado por el ingeniero Iván Deras como tesis de graduación y actualmente se encuentra en el dominio público. Este lenguaje tiene todas sus directivas en castellano e implementa tipos de datos y estructuras de control compatibles con las que posee cualquier lenguaje de programación convencional. El uso de este lenguaje le permitirá al alumno desarrollar programas, codificarlos en LPP y ensayarlos, a la vez que asocia las estructuras y tipos de datos a su propio lenguaje natural, de manera tal de facilitar el aprendizaje sin tener que girar el pensamiento al idioma inglés.

Por último, quiero agradecer a los docentes que desarrollaron las bases de este libro durante mas de 20 años en sus tareas de enseñanza de programación de computadoras en Ingeniería Electrónica y Bioingeniería de la Universidad Nacional de San Juan: Ing. Carlos G. Gil, Dr. Ing. Oscar H. Nasisi y Lic. Claudia Porres.

1 Los problemas y su solución

1.1 Introducción

La vida, el trabajo y las relaciones humanas están plagadas de dificultades, contratiempos e inconvenientes que alteran el normal desenvolvimiento de las actividades. Estas complicaciones, globalmente llamadas **problemas**, deben ser resueltas si es que se pretende devolver la normalidad a la entidad o proceso afectado.

El significado de la palabra **problema** proporcionado por la *Real Academia Española* es consistente con esta visión, ya que lo define como:

> ***"Un conjunto de hechos o circunstancias que dificultan la consecución de algún fin."***

La naturaleza de la **solución** de un problema siempre es fuertemente dependiente del tipo de problema que se trate, y para lograrla probablemente será necesario apropiarse de nuevos conocimientos y habilidades, ya sea mediante estudios guiados por algún sistema de enseñanza o bien, usando la experiencia personal y la ajena para construirlos.

Dependiendo de la existencia de una solución, los problemas pueden clasificarse en:

- ***Solubles:*** cuando se conoce de antemano que existe una solución para el problema.
- ***Insolubles:*** cuando se conoce de antemano que **no** existe una solución para el problema.
- ***Indecidibles:*** cuando **no** se conoce de antemano si existe o no una solución para el problema.

Los problemas **Solubles** son los mas importantes en este estudio, y a su vez se subdividen en:

- ***Algorítmicos:*** son aquellos en los que existe un *algoritmo* que soluciona el problema.
- ***No-algorítimicos:*** son aquellos en los que **no** existe un *algoritmo* que soluciona el problema.

donde un ***Algoritmo*** es ***una lista ordenada y exhaustiva de las tareas, pasos y procesos que es necesario llevar a cabo para lograr la solución del problema***, y que además cumple con el siguiente conjunto de propiedades:

- ***Precisión:*** el algoritmo debe indicar el orden de realización exacto de cada tarea.
- ***Determinismo:*** múltiples ejecuciones del algoritmo con los mismos datos de entrada deben producir siempre el mismo resultado.

- ***Finitud:*** el algoritmo debe finalizar en algún momento, lo que implica que el mismo debe tener un número **finito** de pasos.

Si se extiende la anterior definición de problema con las condiciones necesarias para una *solución algorítmica* puede entonces lograrse una definición más precisa del problema:

> ***"Un problema es una situación desfavorable que puede especificarse en un número finito de declaraciones, cuyo mejoramiento es posible en un tiempo finito y cuya solución produce algún beneficio"***

En el contexto de esta nueva definición existirán numerosos problemas que cumplen con que:

- Pueden especificarse en forma *clara* y *precisa*.
- Siempre se presentan del mismo modo ante las mismas circunstancias.
- Su solución se puede alcanzar en un tiempo razonable.

Estos problemas se dice que son ***solubles algorítmicamente*** y son de capital importancia en este libro, ya que la programación de computadoras consiste en desarrollar algoritmos que solucionen problemas, pudiéndose enunciar que:

> *"La Programación de Computadoras es una disciplina de carácter intelectual orientada a resolver problemas genéricos que cumplen con una serie de requisitos claramente establecidos"*

1.2 Resolución de Problemas

1.2.1 El "Pensamiento Computacional"

Al intentar resolver problemas usando computadoras surge la necesidad de adoptar una nueva forma de pensar, mucho más útil y estructurada que la usada normalmente en la vida cotidiana. Esta nueva forma de organizar el pensamiento es lo que se ha dado en llamar *Pensamiento Computacional*, y es un proceso mediante el cual una persona, usando sus habilidades y conocimientos, su pensamiento crítico y su pensamiento lateral, logra hacer frente y solucionar problemas de diferente naturaleza. El *Pensamiento Computacional* implica resolver problemas, diseñar sistemas y comprender el comportamiento humano, basándose en conceptos fundamentales de la informática. El *Pensamiento Computacional* incluye una variedad de herramientas mentales que reflejan la amplitud del campo de la informática.

Este tipo de pensamiento incluye habilidades tales como modelar y descomponer un problema, procesar datos, crear algoritmos y generalizarlos. Además, se utiliza para resolver de forma algorítmica problemas de distintas disciplinas incluyendo las matemáticas, la biología y las humanidades. El *Pensamiento Computacional* también se puede utilizar para la resolución de problemáticas cotidianas, ya que permite tener una nueva perspectiva y encontrar diferentes soluciones que no son fácilmente consideradas por otros individuos. En el

proceso de resolución pueden distinguirse distintas fases: Eliminación de información no-relevante, Descomposición, Reconocimiento de patrones, Abstracción y Codificación.

- **Eliminación de información no-relevante:** consiste en la omisión de información irrelevante al problema propuesto.
- **Descomposición:** consiste en el procedimiento por el cual un problema de complejo se descompone en pequeños problemas más fáciles de manejar.
- **Reconocimiento de patrones:** luego de la descomposición de un problema complejo, los pequeños problemas resultantes son analizados en forma individual de manera de buscar similitudes con problemas resueltos anteriormente y de esta forma reutilizar soluciones ya probadas.
- **Abstracción:** el proceso de abstracción permite encontrar soluciones que reúnen varias otras de problemas mas simples, generalizar soluciones a partir de su ocurrencia y variabilidad reiterada, generar simulaciones que permitan reproducir un comportamiento complejo y/o producir modelos que resuman las características del problema.
- **Codificación:** en esta fase se redacta el algoritmo que, cumpliendo con lo descubierto en los procesos anteriores, permite resolver el problema.

Es importante destacar que los pasos anteriores no son decisivos en su resultado, y la metodología a seguir puede variar ya que su estructura es mas bien esquemática. Sin embargo, siempre existen las etapas que a continuación se detallan y que asisten en la resolución de un problema.

1.2.2 Etapas en la Resolución de Problemas.

La búsqueda de la solución de un problema siempre se plantea en etapas o pasos que deben seguirse rigurosamente para alcanzar exitosamente la meta perseguida (lo que a la vista de lo recientemente analizado no es más que otro *algoritmo*). En general, puede considerarse que existen las siguientes ***cinco etapas***, que deben cumplirse en la secuencia dada para lograr encontrar la solución de un problema:

I. Formulación o enunciación del problema.

Todo problema debe ser formulado en forma *correcta*, *completa* y *sin ambigüedades*. En este paso deben definirse todas las circunstancias y restricciones vinculadas al problema para poder tenerlas en cuenta. Es inútil continuar con los pasos siguientes si esta etapa no se ha cumplido con suficiencia.

II. Comprensión y Análisis.

Una vez formulado el problema, se debe realizar una comprensión intelectual del mismo y para ello se procede a **analizarlo**. En este contexto, *analizar el problema* significa "desmenuzarlo" o "particionarlo" en componentes o sub-problemas mas pequeños, ya que de esta manera es factible garantizar un conocimiento profundo y detallado del mismo. Debe observarse que esta etapa es coincidente con la de Descomposición del *Pensamiento Computacional*.

III. Elección del procedimiento o método.

Esta etapa consiste en seleccionar la solución óptima entre todas las posibles, si es que existiera más de una. En esta elección el criterio generalmente empleado es el *económico* aunque en otros casos pueden tener mayor importancia la confiabilidad, la rapidez, etc. Esta etapa está directamente relacionada con la de Reconocimiento de Patrones y la de Abstracción del *Pensamiento Computacional*.

IV. Efectivización de la solución.

Esta etapa, también llamada **codificación**, consiste en expresar el método o procedimiento seleccionado de manera tal que pueda ser entendido por quien llevará a cabo la solución. Esta etapa es coincidente con la de Codificación del *Pensamiento Computacional*

V. Ejecución y evaluación de la solución.

Esta es la última etapa y consiste en la ejecución del procedimiento elegido y la evaluación de los resultados obtenidos. Luego de esta etapa, probablemente se repetirá la secuencia de pasos desde el punto **II** si la evaluación de los resultados no permite garantizar que el procedimiento implementado es el correcto.

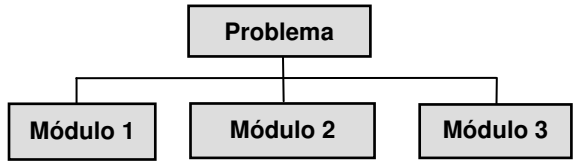
De todos estos pasos, los más importantes son, en primer lugar, el **análisis del problema**, luego la **elección del procedimiento** y por último, la **efectivización de la solución**. Si bien existen diversas técnicas para llevar a cabo la etapa de *análisis*, se estudiará y aplicará intensivamente una metodología simple y efectiva denominada **análisis Top-Down** que permitirá analizar, obtener la comprensión y llegar a la solución *de cualquier tipo de problema* y en particular, los solubles algorítmicamente usando programación.

1.2.3 Técnica de Resolución de problemas

El proceso de resolución de un problema comienza por entender el problema, y para lograrlo, una técnica poderosa es la división del mismo en subproblemas mas pequeños y simples de resolver. La técnica de análisis **Top-Down** es un proceso que permite esta división contribuyendo a lograr el entendimiento buscado, y claramente se corresponde con la etapa de **Descomposición** del *Pensamiento Computacional*.

En esencia, el análisis **Top-Down** consiste en capturar una idea sin especificar detalles (con un alto nivel de abstracción), implementarla partiendo de la misma, e incrementar el nivel de detalle según sea necesario. El sistema inicial se va subdividiendo en módulos cada vez más simples, estableciendo una jerarquía y de esta forma, cada parte del problema se refina diseñando con mayor detalle; y se subdivide cuantas veces sea necesario hasta llegar a los componentes primarios del diseño, es decir, hasta que la especificación completa es lo suficientemente detallada para validar el modelo obtenido.

El modelo resultante del análisis **Top-Down** se representa con la ayuda de "cajas negras" que hacen más fácil cumplir requisitos no expongan en detalle los componentes individuales, tal como muestra el esquema de la figura.



El enfoque **top-down** enfatiza la planificación y conocimiento completo del problema, logrando:

- ⇨ Ubicar la información crítica en una posición de alto nivel.
- ⇨ Comunicar esa información a los niveles mas bajos de la estructura del modelo del problema.
- ⇨ Capturar toda la información del problema en una ubicación centralizada.

y también proporciona las siguientes ventajas:

1. La división de un problema en módulos ayuda a que el trabajo se realice más eficientemente, ya que los desarrolladores pueden trabajar en diferentes módulos al mismo tiempo.
2. Dividir un problema en módulos ayuda al equipo de desarrolladores a trabajar de manera más eficiente, ya que los módulos más sencillos se pueden entregar a los menos experimentados, mientras que los más difíciles se pueden entregar a los más experimentados.
3. Dividir un problema en módulos ayuda a los mecanismos pruebas y de control, porque es más fácil depurar muchos módulos pequeños e independientes que un solo módulo muy grande.
4. Dividir un problema en módulos ayuda a la legibilidad del algoritmo, porque es más fácil seguir lo que sucede en módulos más pequeños que un solo módulo muy grande.
5. La división de un problema en módulos mejora la eficiencia de los desarrolladores porque los módulos independientes pueden almacenarse generando una biblioteca de módulos y reutilizarse cuando sea necesario: por ejemplo, cuando se necesita un módulo para mostrar algunos valores no es necesario que los programadores desarrollen y prueben un módulo para esto nuevamente; solo se reutiliza un módulo existente en la biblioteca. Esto permite ahorros significativos de tiempo y dinero.
6. La división de un problema en módulos mejora la capacidad de los Administradores de Proyectos para monitorear el progreso del desarrollo. Los módulos se pueden registrar en una lista y *"tacharlos"* a medida que se construyen y verifican. Así, la cantidad de módulos *"tachados"* será un indicador del progreso conseguido. Esto es mucho más difícil de hacer si el problema no se ha dividido en módulos.
7. Dividir un problema en módulos es útil para el mantenimiento futuro, por que si alguna funcionalidad necesita ser cambiada por alguna razón, puede ser posible simplemente eliminar un módulo y reemplazarlo por otro.

1.2.4 El Diseño de soluciones a Problemas

Existen dos conceptos claves para entender y aplicar una técnica para el diseño de soluciones:

- Uno de ellos es el **discernimiento** del problema, en el sentido de **comprenderlo en profundidad**. *Discernir* un problema es "descubrir y entender las **partes que componen** el problema". Este es un **camino descendente**, **desde lo complejo a lo sencillo**, desde las unidades compuestas hasta cada una de las partes que las componen, siendo este el proceso de análisis **Top-Down**.
- El otro concepto es la **composición**, que significa *"formar un todo a partir de las partes"*. Se debe entender a la *Composición* como el proceso de "elevar la visión para englobar las partes componentes". A la inversa de lo anterior, es un **camino ascendente**, **desde lo simple a lo complejo**, desde las partes componentes hasta las unidades compuestas, siendo este un proceso llamado **Bottom-Up**.

Ambas etapas son necesarias para encontrar la solución al problema, y así como el **discernimiento**, materializado en el análisis **Top-Down**, ayuda a entender el problema; la **composición**, materializada por el proceso **Bottom-Up**, está asociada a la construcción de la solución.

Cuando se trabaja para lograr el discernimiento y/o la composición del problema usualmente se encuentran diferencias entre los enfoques que diferentes personas hacen del mismo problema, y esto se debe a que en los procesos de discernimiento y composición hay influencia de las características y capacidades propias del ser humano... pero también influyen el entrenamiento y el uso de metodologías adecuadas.

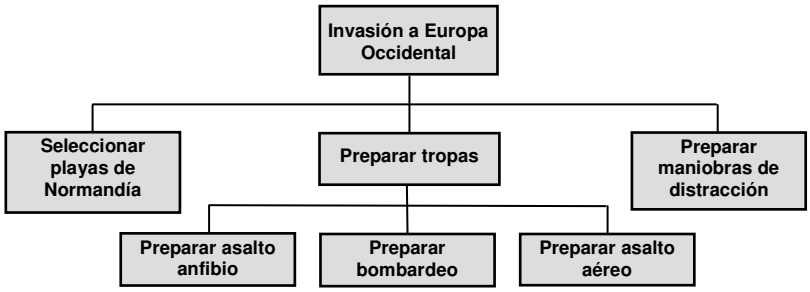
1.2.4.1 Análisis Top-Down

El análisis **Top-Down** de problemas no es otra cosa que la aplicación de *una estrategia de análisis y resolución muy antigua*, basada en las habilidades humanas de *discernimiento* y *composición*. Esta estrategia fue enunciada por el filósofo, matemático y físico francés *René Descartes* en 1637, en su libro *"El Discurso del Método" (Segunda Parte)*, cuando propone los cuatro preceptos que lo guiarán en el entendimiento de las ciencias:

> *"…Fue el primero, no admitir como verdadera cosa alguna, como no supiese con evidencia que lo es; es decir, evitar cuidadosamente la precipitación y la prevención, y no comprender en mis juicios nada más que lo que se presentase tan clara y distintamente a mí espíritu, que no hubiese ninguna ocasión de ponerlo en duda.*
>
> ***El segundo, dividir cada una de las dificultades que examinare, en cuantas partes fuere posible y en cuantas requiriese su mejor solución.***

> ***El tercero, conducir ordenadamente mis pensamientos, empezando por los objetos más simples y más fáciles de conocer, para ir ascendiendo poco a poco, gradualmente, hasta el conocimiento de los más compuestos, e incluso suponiendo un orden entre los que no se preceden naturalmente.***
>
> *Y el último, hacer en todo unos recuentos tan integrales y unas revisiones tan generales, que llegase a estar seguro de no omitir nada…"*

A modo de ejemplo, podría suponerse que los militares de la *Conferencia Trident* (1943) se plantearon un esquema similar al siguiente cuando analizaban la preparación del *desembarco aliado en Normandía* (el famoso *Día D* en junio de 1944):



Estos militares tenían el objetivo específico de *liberar los territorios de Europa Occidental en poder de las tropas alemanas, iniciando la invasión por las playas francesas de Normandía*. Pero siendo seres humanos, su propia naturaleza no les permitía *discernir* el problema en forma completa ya que era demasiado abstracto, extenso y complejo como para conocer en detalle todo lo que este objetivo implicaba. Para poder encararlo, dividieron el problema en partes más pequeñas y simples de resolver, y a medida que realizaban la división encontraban partes que les resultaban fáciles de comprender y de llevar a cabo, mientras otras partes continuaban siendo abstractas y complejas. A estas las volvieron a dividir, y así sucesivamente hasta que el objetivo complejo pudo ser alcanzado a través de muchas **acciones** simples. Construyeron el esquema desde lo general a lo particular, desde lo abstracto a lo concreto (aplicando la técnica Top-Down). Y lo ejecutaron a la inversa, desde lo particular a lo general, construyendo abstracciones complejas por composición de elementos simples (aplicando el proceso Bottom-Up)… y finalmente lograron una invasión exitosa.

Cuando en la sección 1.2.2 se propone el paso de **"Comprender y analizar el problema"**, se plantea en él la necesidad de dividir el problema en extensión y complejidad para comprenderlo y solucionarlo, y el ejemplo del *Día D* es una muestra de como se pueden resolver los problemas que la mente, inicialmente, no puede abarcar, y por ello, la pregunta:

> ***¿Cómo se resuelven los problemas que inicialmente se encuentran más allá de la capacidad de comprensión humana?***

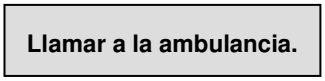
tendrá como respuesta:

> ***Mediante la aplicación organizada y sistemática de los criterios de discernimiento y de composición.***

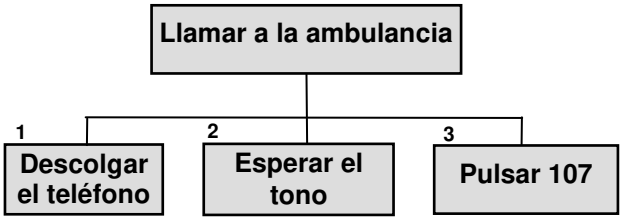
y para diseñar un algoritmo que solucione un problema complejo se aplicará el siguiente postulado:

> ***"Para desarrollar algoritmos que solucionen problemas complejos y difíciles de comprender hay que descomponer los problemas, posiblemente en mediante iteraciones sucesivas hasta llegar a las partes componentes (llamadas módulos o segmentos) que sean posible comprender y resolver. Luego, el algoritmo se construirá componiendo las soluciones individuales de cada uno de estos módulos en el mismo orden indicado en la descomposición realizada".***

Entonces, para encarar el diseño de la solución a un problema mediante el análisis ***Top-Down***, se partirá de un objetivo o problema abstracto y se lo dividirá en las partes que lo componen. Cada una de estas partes se dividirá a su vez en sus propias partes componentes y así se procederá hasta llegar a un nivel final de división en el cual ya no es factible (o conveniente) realizar nuevas descomposiciones. El nivel en el cual se detiene la división depende de quien lleve a cabo este análisis, basándose por ejemplo, en sus conocimientos, en su experiencia, etc. Si se supone que después de subdividir varias veces un problema se llega a que una de las acciones es:



Esta "parte" aparenta ser una acción trivial, y cualquier persona adulta podría llevarla a cabo sin mayores dificultades. Pero si quien debe realizar la llamada es un niño, la tarea deja de ser trivial y requiere una descomposición que permita adecuar su ejecución a las habilidades del niño, tal como la siguiente:



O bien:

1. Descolgar el teléfono

2. Esperar tono
3. Pulsar 107

Evidentemente, la decisión a tomar depende del grado de detalle que se quiera lograr y de lo que *subjetivamente* se considere como "**fácil de hacer**".

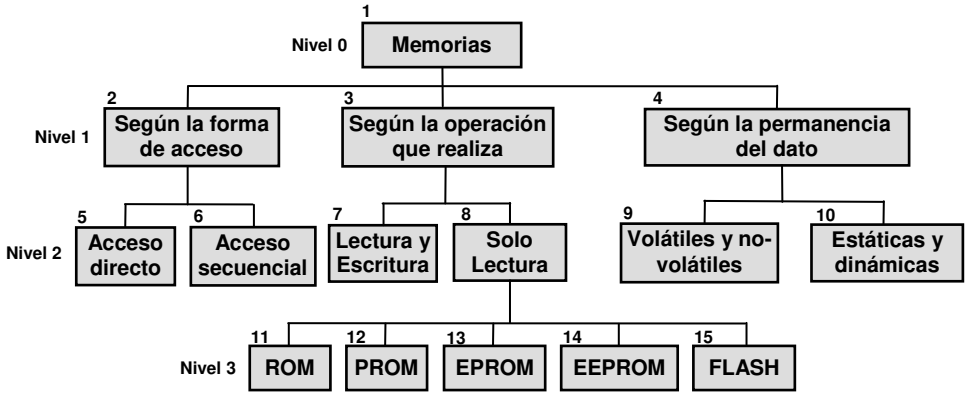
Estas dos formas anteriores de representar la división en las partes componentes se denominan ***diagrama descendente de contenidos***, la gráfica, e ***índice descendente de contenidos*** la lista numerada. Por ejemplo, si el objetivo fuera:

"Analizar la clasificación de las memorias de las computadoras"

se podría entonces generar el siguiente ***índice descendente*** *de contenidos*:

1. Memorias
   1.1. Según la forma de acceso.
      1.1.1. Acceso aleatorio o directo.
      1.1.2. Acceso secuencial
   1.2. Según la operación que realiza.
      1.2.1. Memorias de lectura y escritura.
      1.2.2. Memorias de solo lectura.
         1.2.2.1. ROM
         1.2.2.2. PROM
         1.2.2.3. EPROM
         1.2.2.4. EEPROM
         1.2.2.5. FLASH
   1.3. Según la permanencia del dato.
      1.3.1. Volátiles y no-volátiles
      1.3.2. Estáticas y dinámicas.

Aunque también podría realizarse un ***diagrama descendente*** que represente esta misma división, tal como el que se muestra a continuación:



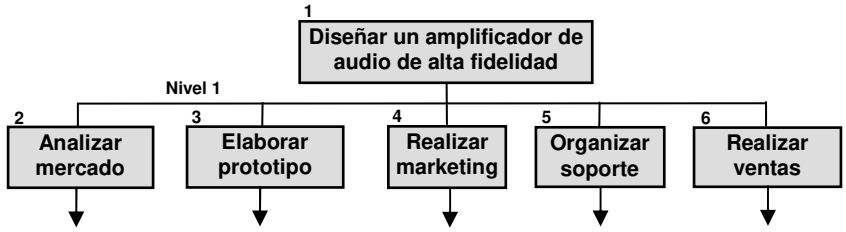
Cada recuadro de este gráfico se denomina **nodo**, y cada *nodo* se encuentra en un cierto nivel de concreción o definición. El *nivel 0* corresponde al problema original, que aún no ha sufrido subdivisiones. El *nivel 1* son los *nodos* que se derivan de la división del nivel 0. El *nivel 2* son los *nodos* que se derivan de la

división de *nodos* del *nivel 1*, el *nivel 3* son los *nodos* que se derivan de la división de *nodos* del *nivel 2*, y así sucesivamente.

Si bien la numeración del diagrama puede ser arbitraria, ya que es la parte gráfica quien muestra la interdependencia entre los nodos, la numeración **debería reflejar cuáles son las partes componentes de cada nodo**.

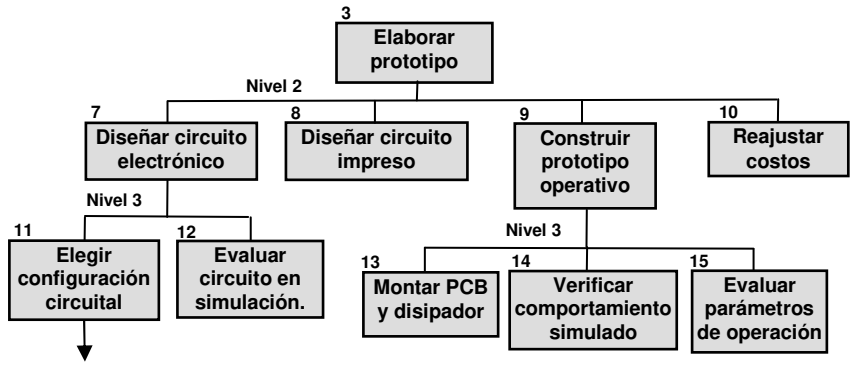
Un *índice descendente de contenidos* y un *diagrama descendente* contienen la misma información, pero siempre se prefiere el uso de los *diagramas descendentes* para determinar los módulos o segmentos del algoritmo, o los módulos y esqueletos de los módulos, sin llegar a una descomposición tan fina que alcance instrucciones primitivas.

Si el problema es grande y complejo, el tamaño de un diagrama descendente de contenidos puede volverse excesivamente grande, y para mitigar este inconveniente se utilizan *herramientas de continuación* (flechas) que indican que aún faltan secciones por explicitar en este diagrama, ya sea por que aún no se han analizado o por que se encuentran en otro esquema aparte. Por ejemplo, en el siguiente caso:

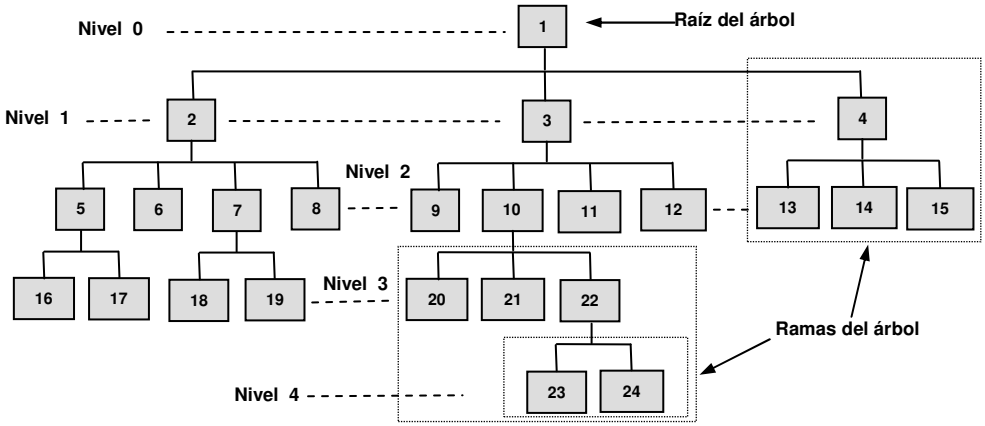


En esta instancia al problema se le ha aplicado una descomposición inicial y se obtiene el *nivel 1* del esquema descendente. Las divisiones siguientes aún no se han realizado, lo cual queda indicado por una *flecha de continuidad*.

El próximo esquema representa el desarrollo del nodo 3 (de *nivel 1*) hasta los nodos del *nivel 3*.

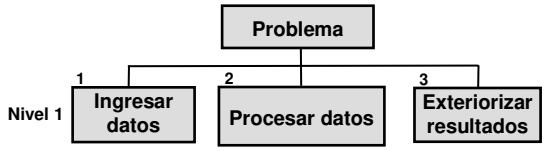


Según este esquema, toda la descomposición del nodo 3 finaliza en el *nivel 3*, excepto el nodo 11. *En otro esquema* se podría tener la descomposición del nodo 11 a *nivel 4*, o a *nivel 5*, o a *nivel 16*, según la complejidad del nodo. Una vez desarrollados todos los nodos hasta el último nivel se obtiene la representación del problema como **"un árbol"**:



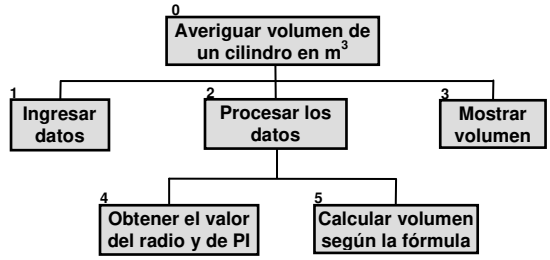
En este esquema la numeración sigue un orden arbitrario, tal vez en el que se fueron construyendo los nodos, pero ***la numeración siempre debe mostrar en forma explícita la secuencia en la que se deben ejecutar los nodos***. A cada *nodo* también se le puede asignar una descripción, así como datos, comentarios, desarrollos, y todo aquello que se considere oportuno.

Cuando se utiliza el diseño ***top-down*** en problemas solubles algorítmicamente mediante programación con frecuencia se encuentra que ***un problema consta de tres partes típicas***:



Estos nodos del *nivel 1* están presentes en muchos diagramas descendentes, y esto se debe a que muchos programas responden naturalmente a esta estructura elemental y *para los problemas más sencillos, ahí finaliza la descomposición*.

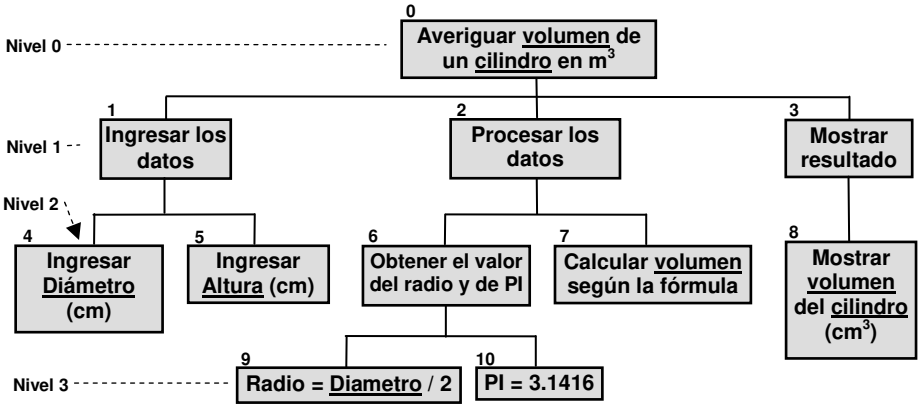
Un ejemplo común de esta estructura básica es el problema de obtener el volumen de un cilindro, del que se conocen su *altura* y su *diámetro*, mediante la fórmula **`Volumen = PI * Radio`^{2} `* Altura`**:



Nuevamente, *el nivel de detalle a generar es completamente dependiente de quien se encargue de llevar a cabo la solución*. Por ejemplo, los **nodos 1** y **3** no se subdividen por que quien realiza el trabajo entiende perfectamente la tarea a realizar y estima innecesario *"obtener mas detalles"* para resolverlos.

Si se los particionara nuevamente, se alcanzaría el nivel de *acción primitiva* (ver 1.3.3), **pero siempre se prefiere el uso de los *esquemas descendentes* para ver la estructura del problema más que llegar a sus detalles**. Sin embargo, *si fuese necesario o conveniente, se puede proseguir el desglose tanto como se quiera*.

En este último ejemplo se pueden observar algunas cosas interesantes. En primer lugar, **se ha tratado de usar un lenguaje coloquial directamente referido al problema**. En vez de "*Ingresar diámetro (m)*", quizás se podría haber escrito *Leer_D*, que es más "estilo programador". Sin embargo, no se debe buscar un enfoque directo a la programación ya que **el objetivo del *diagrama descendente* es entender y clarificar la estructura del problema y de sus partes** y no tener un desarrollo parecido al *código del programa*. Con esto en mente, resulta mucho más claro escribir "*Ingresar diámetro (m)*", que pone de manifiesto que hay que preparar la adquisición de un dato (diámetro) que se espera en metros, antes que escribir *Leer_D* que resulta poco descriptivo.



Este último comentario es particularmente importante ya que cada problema define su propio conjunto de términos que hacen referencia a las *entidades* y *procesos* que participan en él. Este conjunto de entidades y procesos se denomina ***dominio del problema*** y es imprescindible usar el vocabulario que le pertenece, ya que de esta forma los *diagramas* e *índices descendentes* que utilizan tales términos permitirán una comunicación fluida y no-ambigua entre los analistas, los desarrolladores y los "clientes" que requieren el problema resuelto. Básicamente, su uso implica **que todos "hablen el mismo idioma"**.

En este mismo ejemplo del cálculo del volumen del cilindro, analizando el enunciado se determina rápidamente el lenguaje del *dominio del problema*, y así las palabras *diámetro*, *altura*, *volumen* y *cilindro* se evidencian como el conjunto de nombres que deberá utilizarse cuando se haga referencia a cualquiera de estas entidades, y por ende, mas temprano que tarde deberán aparecer en los módulos o segmentos resultantes de la aplicación del método *top-down*.

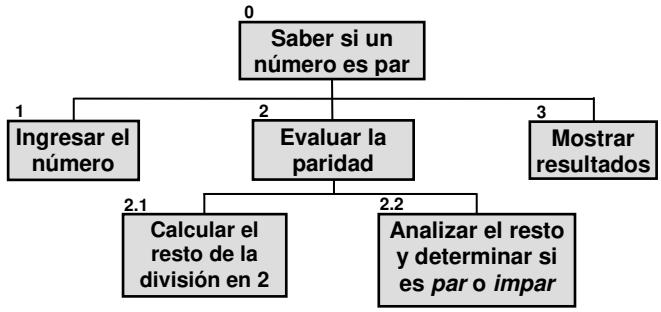
También debe observarse que dentro de los **nodos de un mismo nivel** de descomposición hay nodos que se mantienen relativamente abstractos mientras otros se tornan elementales. Por ejemplo, en el *nivel 2* el nodo "*Preparar datos*

para la fórmula" tiene cierto nivel de abstracción, mientras que el nodo 8, "*Mostrar volumen del cilindro (m^3)*", es bastante elemental o primitivo.

Conclusión: *el nivel en el que se encuentra un nodo no refleja su grado de abstracción, ya que este grado es propio del nodo sin importar su posición relativa en el árbol*.

Así, por ejemplo, en un problema extenso sería factible encontrar nodos de *nivel 2* de carácter elemental y sin subdividir, frente a nodos de *nivel 10* de carácter fuertemente abstracto.

Otro ejemplo interesante es el problema de determinar si un número es par o impar, cuyo esquema descendente es similar a:



En este diagrama ya se dispone *de la estructura* de la solución. Ahora es necesario producir el código que resuelve cada nodo, y luego organizar el mismo en base a las ramificaciones de la descomposición. En este caso se pueden considerar distintas organizaciones posibles:

1. El algoritmo principal engloba a la entrada de datos y la muestra de resultados, y llama, luego de la entrada de datos, al módulo *Evaluar paridad*. El módulo *Evaluar paridad* tiene desarrollo propio y llama a los módulos *Calcular Resto* y *Analizar Resto*.
2. El algoritmo principal tiene en su desarrollo llamadas a los módulos: *Ingresar el número*, *Evaluar paridad* y *Mostrar Resultados*.
3. El algoritmo principal tiene en su desarrollo la llamada a los tres módulos (*Ingresar el número*, *Evaluar paridad* y *Mostrar Resultados*) y a su vez el módulo *Evaluar paridad* llama a otros dos módulos (*Calcular Resto* y *Analizar Resto*).

Todas estas opciones son igualmente válidas, pero ¿existe una que sea la mejor? **Cada analista es quien decide cómo estructurar su algoritmo**, aunque *existen diversos criterios que se consideran "buenas prácticas"*:

- Un módulo no debe ser demasiado extenso ni demasiado breve, pero lo primordial es la correcta organización y estructura del programa y **no se deben agrupar contenidos no-relacionados**, ni dividir un proceso que se entiende mejor estando agrupado.

- Se debe buscar que el módulo sea independiente de otros módulos y que, en lo posible, realice un **único** proceso, acotado y claramente identificable[1].
- La estructura debe ser fácil de entender, y en algunas ocasiones puede ser necesario crear o eliminar módulos al solo fin de obtener un algoritmo con una estructura fácil de comprender.

Siempre es necesario recordar que no existen reglas genéricas para definir el tamaño de los módulos ni el grado de su participación en la solución final. Solo el análisis y la experiencia podrán orientar sobre la mejor decisión.

1.2.4.2 Obtención de soluciones parciales

Una vez elaborado el análisis **Top-Down** del problema hasta el nivel de entendimiento deseado, corresponde ahora encontrar las soluciones a cada uno de los mas pequeños y simples subproblemas resultantes de la descomposición realizada, **que son los que se ubican en el último nivel de cada rama del árbol encontrado**.

Para encontrar estas soluciones debe comenzarse por analizar cada nodo terminal para determinar si la solución del mismo ya es conocida y está disponible (esto es llamado **Reconocimiento de Patrones** en el *Pensamiento Computacional*) tales como las del apartado 2.7, o si se debe proceder a un estudio mas detallado del mismo, en la búsqueda de una nueva solución hasta ahora desconocida.

Evidentemente, a medida que aumenta la cantidad de problemas resueltos aumenta proporcionalmente la experiencia y el conocimiento sobre las pequeñas soluciones que se han desarrollado a lo largo del tiempo, y el Reconocimiento de Patrones se transforma en una tarea mas simple de lograr con una base de soluciones mucho mas amplia.

1.2.4.3 Construcción de la solución final

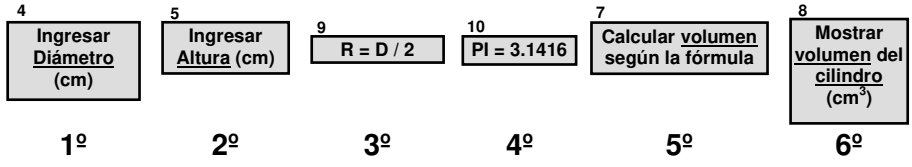
La forma de encontrar la solución a todo el problema es agrupar las soluciones de los subproblemas, ordenándolas secuencialmente según su numeración (que tal como se dijo anteriormente, debe mostrar en forma explícita la secuencia en la que se deben ejecutar los nodos). A fin de ejemplificar esta técnica se puede decir que la solución del problema del cálculo del volumen de un cilindro se constituye en la ejecución del siguiente listado de tareas **exactamente** en el orden detallado:

1º. **Nodo 4:** Ingresar diámetro (cm)
2º. **Nodo 5:** Ingresar altura (cm)
3º. **Nodo 9:** R = D / 2
4º. **Nodo 10:** PI = 3.1416
5º. **Nodo 7:** Calcular volumen según la fórmula

[1] Esto es la base de lo que se conoce como ***Alta Cohesión*** y ***Bajo Acoplamiento*** en el modelo de patrones GRASP del paradigma de la Orientación a Objetos.

6º. **Nodo 8:** Mostrar volumen del cilindro (cm^3)

O gráficamente:



Debe observarse que este ejemplo muestra que el mecanismo elegido para numerar los nodos no es del todo correcto, ya que si bien en el diagrama descendente resulta claro cual es la secuencia de ejecución de nodos descripta, al construir la solución final se intercalan algunos valores numéricos que lucen incorrectos en una secuencia numérica que "**debería ser**" monótona creciente en el *número de nodo*: 4, 5, **9**, **10**, 7, 8. Por este motivo se adopta un **esquema de numeración** que siempre hace referencia a los nodos "padres" de niveles superiores, del tipo **2**, **2.1**, **2.1.1**, etc.

De esta forma y solo a modo de ejemplo, el nodo 9 se hubiera transformado en el **6.1** y el nodo 10 hubiera resultado el **6.2**, por lo que la secuencia final sería: 4, 5, **6.1**, **6.2**, 7, 8; que es totalmente consistente con la numeración correlativa esperada.

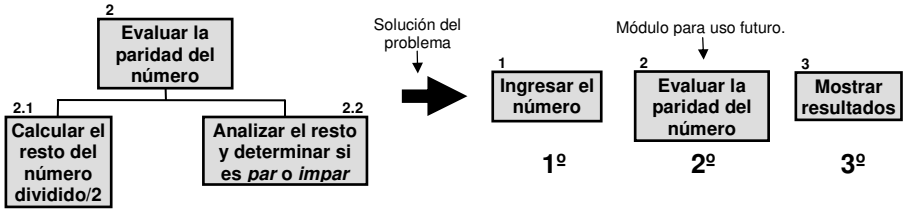
Por supuesto que en un diagrama correctamente diseñado, este esquema de numeración se utiliza desde los niveles iniciales y no solo cuando aparecen inconvenientes con la numeración, por lo que siempre debería aplicarse esa metodología, excepto tal vez en el **Nivel 1** cuyos nodos se convertirán en los super-padres de todas las ramas del árbol obtenido.

Por otra parte, si bien la secuencia expuesta es completamente válida y logra solucionar el problema planteado, en este caso la **composición** no sigue un camino de abstracción controlado e inverso al discernimiento logrado, ya que no se reconstruyen los nodos intermedios del árbol y ello impide obtener la solución de forma incremental. Este tipo de soluciones "*rápidas y sucias*" se aplican comúnmente a problemas cotidianos y no necesariamente relacionados con las computadoras, por que en ellos los nodos intermedios ayudan a comprender el problema pero, generalmente, no aportan mas valor a la solución final.

Sin embargo, en los problemas solubles algorítmicamente usando computadoras, los nodos intermedios si son valiosos por que se suelen constituir en la solución de un subproblema específico pero de complejidad mayor que la línea de nodos que de él se desprende. Puede suceder que en el futuro aparezca el mismo subproblema en la descomposición de un nuevo problema, y si así sucede, ese subproblema (nodo) ya tendrá una solución conocida y probada, por lo que no se deberá invertir tiempo en diseñar y evaluar nuevas alternativas.

Por ejemplo, en el caso del problema de determinar si un número es par, los nodos **2.1** y **2.2** constituyen la solución al **nodo 2**, que resuelve un problema muy útil y concreto: **Evaluar la paridad de un número**. Este nodo es el encargado de determinar si un número es par o impar y una vez **resuelto**, ya es conocido y

puede utilizarse en futuras soluciones que requieran, entre otras cosas, el análisis de la paridad de un número.



Este proceso de construcción de módulos independientes que resuelven subproblemas específicos es de fundamental importancia para la etapa del ***Reconocimiento de Patrones***, ya que le proveen soluciones reutilizables y fácilmente reconocibles.

1.2.5 Un ejemplo de análisis aplicando el método Top-Down

En este ejemplo se realizará el *desmenuzamiento* de un problema muy simple y general, mediante el uso del método *top-down*, tratando de aplicar todas las consideraciones antes enunciadas. El ejemplo a analizar es:

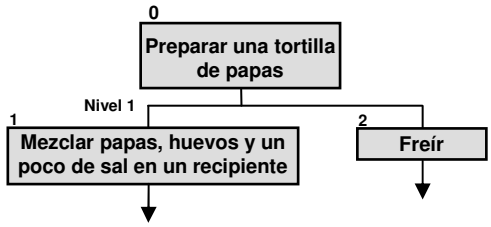
Preparar una tortilla de papas.

En primer lugar, se debe estar al tanto del nivel de conocimiento que posee el encargado de lograr la solución del problema, ya que el grado de *discernimiento* a alcanzar durante el análisis no será igual para un experto en el tema que para un principiante.

De la forma como está expresado, el enunciado de este problema basta para que un *cocinero experto* pueda resolverlo sin información adicional, pero si quien va a preparar la tortilla no es un experto será preciso ahondar en el detalle de los pasos necesarios para llevar a cabo la preparación. Para este caso *se propone* la siguiente descomposición de tareas (*Nivel 1*):

t1: mezclar papas, huevos y un poco de sal en un recipiente.
t2: freír.

O en forma de diagrama descendente:

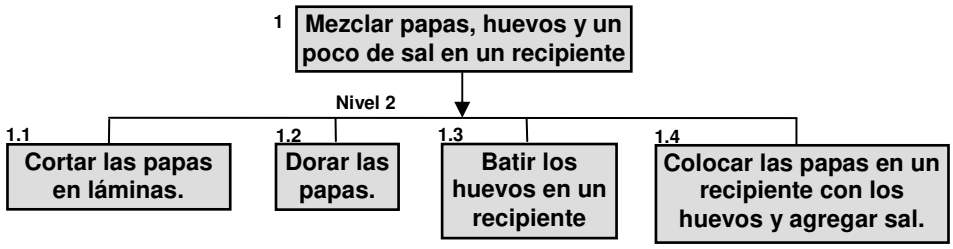


Esto resolvería el problema si el encargado de ejecutar la solución no fuera un principiante o una persona que da sus primeros pasos en tareas relativas a la cocina, ya que esta forma de expresión **presupone** muchas cosas. Si este problema debiera ser resuelto por una persona que no sabe cocinar, sería preciso aumentar el nivel de detalle de ***t1*** y ***t2***, pues estos no son lo suficientemente sencillos para un aprendiz. Por ello, se debe profundizar en los

detalles de cada uno de los módulos anteriores apoyándose en el uso de las *flechas de continuidad* previstas al efecto:

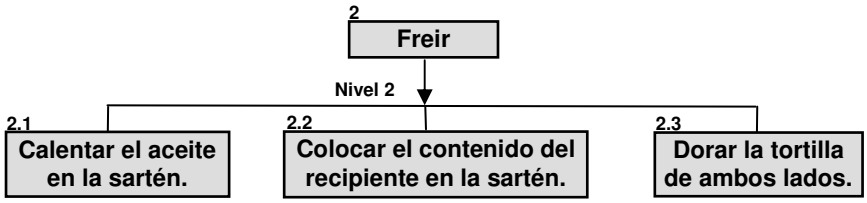
- Descomposición de **t1** (*Nivel 2*):

  t1.1: Cortar las papas en láminas.
  t1.2: Dorar las papas.
  t1.3: Batir los huevos en un recipiente.
  t1.4: Colocar las papas en el recipiente con los huevos y agregar sal.



- Descomposición de **t2** (*Nivel 2*):

  t2.1: Calentar el aceite en la sartén.
  t2.2: Colocar el contenido del recipiente en la sartén.
  t2.3: Dorar la tortilla de ambos lados.



Si este nivel de detalle no fuera suficiente para el aprendiz, se debería explicitar aún más cada módulo y, por ejemplo, la subtarea **t1.3** podría expresarse como:

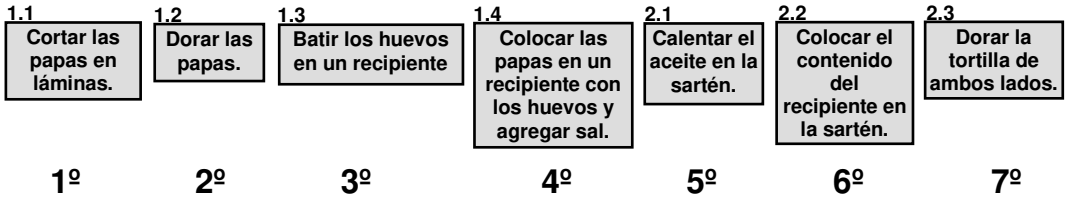
- Descomposición de t1.3:

  t1.3.1: Poner los tres huevos sobre la mesada de la cocina.
  Repita
  t1.3.2: Tomar un huevo de la mesada.
  t1.3.3: Romper el huevo y colocar su contenido en el recipiente.
  t1.3.4: Tirar la cáscara a la basura.
  hasta que no haya mas huevos sobre la mesada.

En esta última descomposición ya se manifiesta un marcado enfoque hacia la programación, dado que la estructura de repetición mostrada es un recurso de la *Programación Estructurada*. ***No es posible incluir este nivel de detalle en los Diagramas Descendentes***, dado que la función principal de estos es mostrar los módulos o segmentos que definen la estructura de la solución del problema. Sin embargo, es perfectamente posible llegar a este nivel de detalle utilizando un *índice descendente* tal como muestra la descomposición de **t1.3**.

También debe observarse que se ha utilizado el **esquema de numeración** antes sugerido, de manera tal de explicitar claramente a cual módulo de nivel superior pertenece la descomposición, y por otra parte, cual es la secuencia de ejecución de los módulos resultantes de la partición. Este proceder es muy importante, ya que simplifica notablemente el proceso de abstracción cuando se construye el algoritmo final.

La solución final al problema de preparar la tortilla de papas es la siguiente secuencia de tareas:



1.3 Participantes en la solución de un problema.

Hasta ahora se han mencionado sin definir algunos de los elementos que intervienen en el problema y en su solución. A continuación se formalizará el significado de cada uno de ellos.

1.3.1 Procesador

Es toda entidad (cosa o persona) capaz de "INTERPRETAR" un enunciado y ejecutar el trabajo indicado por él. En otras palabras, ***el procesador es quien ejecuta las tareas del algoritmo para solucionar el problema****.*

1.3.2 Ambiente

Es el conjunto de recursos y consideraciones necesarias para la ejecución de una tarea. Si el ambiente no provee los elementos necesarios el procesador no podrá ejecutar el trabajo.

Un ejemplo ayuda a ilustrar las afirmaciones realizadas:

Una persona se encuentra en la vía pública y debe efectuar una llamada telefónica utilizando su teléfono celular. Existen tres posibilidades:

1. *No hay cobertura de señal del teléfono celular en la zona.*
2. *Hay cobertura pero la persona no tiene crédito.*
3. *Hay cobertura y la persona tiene crédito.*

- En el primer caso, EL PROCESADOR (la persona) no puede, en función del AMBIENTE (cobertura de la señal celular), efectuar el trabajo propuesto.
- En el segundo caso, EL PROCESADOR decidirá entre no hablar o invertir en comprar crédito.
- En el tercer caso EL PROCESADOR podrá realizar la llamada sin ninguna dificultad.

1.3.3 Acción

Es cada uno de los pasos, etapas o tareas que el procesador ejecuta para llegar a la solución, posiblemente modificando el ambiente. Las Acciones siempre son ***Verbos***.

- **Acción Primitiva o Elemental**

Una acción es primitiva si su enunciado es suficiente para que un procesador pueda realizarla sin necesidad de información adicional.

- **Acción No-Primitiva o Acción Compuesta**

Es aquella cuyo enunciado no puede ser interpretado directamente por el procesador, y por lo tanto debe ser descompuesta en acciones cada vez mas simples hasta que cada una de ellas sea una acción primitiva.

Para aclarar las diferencias entre acciones primitivas y no-primitivas considérese el siguiente problema:

Preparar sopa para cuatro personas

- Si el *procesador* es un cocinero experimentado, el interpretará este enunciado sin necesidad de mayor detalle, y por lo tanto, *para este procesador*, esta es una acción primitiva.
- Si el *procesador* es una persona con escasos conocimientos de cocina, no sabrá interpretar el enunciado y habrá que descomponerlo en una serie de acciones mas simples, las cuales podrá interpretar y por lo tanto estas serán acciones primitivas para *este otro procesador*. Por ejemplo:

  tomar una olla de 1,5 lts.
  verter 1 lt de agua
  poner a calentar la olla
  pelar las verduras
  .....
  etc.

1.3.4 Condición

Una condición es una afirmación lógica sobre el estado del ambiente, que puede ser verdadera o falsa en el momento de la observación.

El procesador determina, en el momento de la ejecución del trabajo, las acciones a seguir dependiendo si la condición es satisfecha o no. No debe confundirse una *condición* con una *acción*, las acciones son ***verbos*** (Leer, Sumar, Repartir, Escribir, etc.) que modifican el ambiente mientras que las condiciones se utilizan para determinar cuando se da por finalizado un determinado proceso (*es una pregunta sobre el estado del ambiente*).

Habiendo realizado estas definiciones, se analizará un nuevo ejemplo para mostrar una aplicación práctica de la metodología de análisis **Top-Down** e introducir un nuevo concepto.

Problema:

Se desea repartir entre cuatro niños, caramelos de frutilla, menta y naranja, contenidos en una bolsa opaca. La persona que debe repartir los caramelos (procesador), para evitar conflictos, dará a cada niño la misma cantidad de caramelos de cada gusto, suponiendo al iniciar que la bolsa contiene por lo menos cuatro del mismo gusto.

a. Para ejecutar el trabajo el procesador dispone del siguiente ambiente:
   - cuatro niños.
   - bolsa conteniendo caramelos de frutilla, menta y naranja.
   - tres cajas, una para cada gusto.

b. Las acciones primitivas que el individuo (procesador) sabe realizar son:
   - tomar un caramelo de la bolsa.
   - poner un caramelo en la caja correspondiente.
   - tomar cuatro caramelos de una caja y distribuirlos.

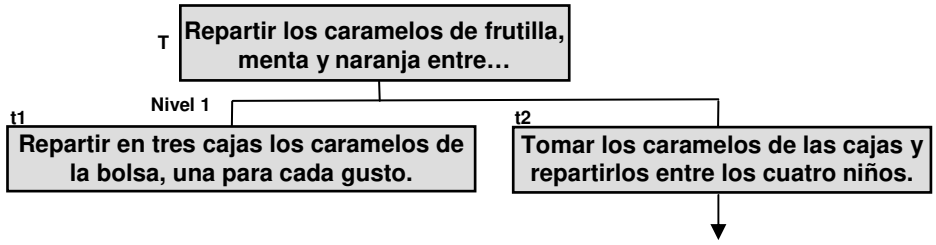
c. Las condiciones que interpreta el procesador son:
   - bolsa vacía.
   - una caja contiene menos de cuatro caramelos.

Este trabajo también puede ser descripto por el siguiente enunciado equivalente:

T. *repartir los caramelos de frutilla, menta y naranja entre cuatro niños de manera tal que a cada uno le corresponda la misma cantidad de caramelos de cada gusto.*

El enunciado anterior no muestra, en principio, acciones primitivas posibles de realizar por parte de el procesador, y por tal motivo la tarea **T** debería descomponerse en otras mas simples. Una primera descomposición podría ser la siguiente:

> ***t1.*** *repartir en tres cajas los caramelos que están en la bolsa, de acuerdo al gusto.*
> ***t2.*** *tomar los caramelos de las cajas y distribuirlos entre los cuatro niños.*



Las acciones ***t1*** y ***t2*** son independientes entre sí y solucionan la tarea **T**, pero aún no constituyen acciones primitivas para el procesador disponible, y por lo tanto se debe continuar la descomposición.

Descomposición de t1:

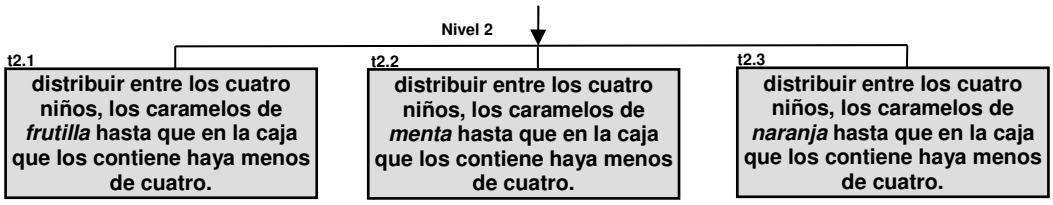
Repita

t1.1 tomar un caramelo de la bolsa
t1.2 poner el caramelo en la caja correspondiente al gusto
hasta que (***t1.3***) la bolsa esté vacía.

Las acciones ***t1.1*** y ***t1.2*** son primitivas y la condición ***t1.3*** puede ser evaluada por el procesador, por lo tanto la descomposición de ***t1*** ha terminado.

Descomposición de t2:

t2.1. *distribuir entre los cuatro niños, los caramelos de frutilla hasta que en la caja que los contiene haya menos de cuatro.*
t2.2. *distribuir entre los cuatro niños, los caramelos de menta hasta que ...*
t2.3. *distribuir entre los cuatro niños, los caramelos de naranja hasta que ...*



Esta descomposición de ***t2*** aún no contempla tareas primitivas, pero la acción ***t2.1*** si puede descomponerse en operaciones primitivas de la siguiente forma:

Repita
t2.1.1 tomar cuatro caramelos de la caja de frutilla y distribuirlos entre los cuatro niños.
hasta que (***t2.1.2***) la caja **de frutilla** contenga menos de cuatro unidades.

De forma similar se deben descomponer las acciones **t2.2** y **t2.3**, y de esta manera el problema **T**, por medio del método **top-down**, se habrá descompuesto en una serie de operaciones primitivas.

La secuencia final de acciones primitivas no es más que un **Algoritmo** según lo definido en 1.1 y para este ejemplo es la composición secuencial de los módulos encontrados, respetando estrictamente el orden definido en la numeración:

Inicio
Repita
t1 tomar un caramelo de la bolsa
poner un caramelo en la caja del gusto correspondiente
hasta que la bolsa esté vacía
Repita
t2.1 tomar 4 caramelos de la caja de frutillas y distribuirlos
hasta que la caja de frutilla contenga menos de 4 caramelos
Repita
t2.2 tomar 4 caramelos de la caja de menta y distribuirlos
hasta que la caja de menta contenga menos de 4 caramelos

t2.3

Repita
    tomar 4 caramelos de la caja de naranja y distribuirlos
hasta que la caja de naranja contenga menos de 4 caramelos

1.3.5 Algoritmo

Adaptando ahora la definición de Algoritmo dada en 1.1 para incluir las entidades descriptas en 1.3.1 a 1.3.4 se puede decir que:

Un algoritmo es una lista ordenada de acciones primitivas que pueden ser ejecutadas por un procesador y que permite alcanzar la solución de un problema.

A continuación se resolverá el mismo problema anterior pero sin tener en cuenta la suposición de que inicialmente la bolsa contiene por lo menos cuatro caramelos de cada gusto. En estas condiciones el esquema de solución adoptado antes:

Repita
    tomar 4 caramelos de la caja de menta y distribuirlos
hasta que la caja de menta contenga menos de 4 caramelos.

fallaría si inicialmente la bolsa contuviera 3 o menos caramelos de menta. Esto indica que se debe buscar otra "forma de repetición", que puede ser la siguiente:

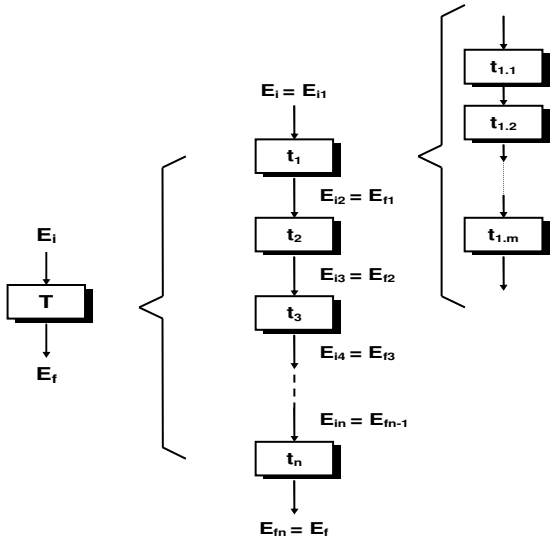
mientras haya por lo menos 4 caramelos en la caja de menta hacer
    tomar 4 caramelos de la caja de menta y distribuirlos
fin mientras

Esta forma alternativa de repetición, aunque similar a la "*Repita*", se diferencia de ella en que la condición se evalúa antes de ejecutar alguna acción. Por otra parte, la condición *"haya por lo menos 4 caramelos en la caja de …"* no es directamente evaluable por el procesador y debe ser obtenida transformando la condición *"que la caja de …. contenga menos de 4 caramelos"* mediante un mecanismo aún desconocido.

La estructuras como las descriptas existen en la *programación de computadoras* y permiten resolver cualquier problema. Sin embargo, las computadoras no pueden ejecutar directamente los algoritmos en forma literal como en el ejemplo, y para que puedan hacerlo es necesario ***codificarlos*** en un lenguaje apropiado que se denomina **"Lenguaje de Programación"**, y el Algoritmo así codificado es lo que se conoce como **"Programa".**

1.3.6 Un Modelo (un poco) más formal del Diseño Top-Down

Con estas nuevas entidades ya definidas, se puede dar una explicación alternativa al método **Top-Down**, que consiste en la propuesta descripta en la siguiente figura:



*Dado un trabajo **T** por medio de un **enunciado no primitivo**, tal que transforme el ambiente desde un estado inicial **E_i** en un estado final **E_f**, es posible encontrar una descomposición **t_1, t_2,..... t_n**, que constituya una secuencia de enunciados que ejecuten el mismo trabajo original **T**.*

Para cada enunciado t_k existen dos posibilidades:

a. t_k es una acción primitiva para el procesador, dando así por terminada la descomposición de t_k.

b. t_k no es una acción primitiva para el procesador y por lo tanto debe descomponerse en una nueva secuencia de acciones (primitivas o nó) t_{k1}, t_{k2},..... t_{kn}.

2 La programación de computadoras

Una vez que se tiene el árbol de módulos que definen las acciones a ejecutar para alcanzar la solución de un problema, y si el mismo cumple con las características mencionadas en el apartado 1.1, entonces será posible llegar a su solución mediante un *programa de computadora*. El diseño de este programa puede lograrse empleando alguna de las diversas metodologías existentes (también llamadas *paradigmas*) entre las cuales existe una particularmente importante denominada *Programación Estructurada*, que será el paradigma a utilizar en este libro.

2.1 La Programación Estructurada

2.1.1 Visión clásica: El Control del Flujo de Ejecución

La visión clásica de la programación estructurada se refiere al *control de ejecución de un algoritmo* y este control del flujo de ejecución es uno de los aspectos más importantes a tener en cuenta al construir un algoritmo, ya que la regla general es que las acciones se ejecutan sucesivamente una tras otra, pero habrán diversas partes del programa que podrán ejecutarse o no, dependiendo de que se cumpla alguna condición. También habrán acciones que deban ejecutarse varias veces, ya sea en un número fijo, o hasta que una condición determine el fin de la repetición, tal como sucedió en la descomposición de la subtarea ***t1.3***.

El concepto de **programación estructurada** fue desarrollado en sus principios por *Edsger W. Dijkstra* en su artículo *"Notes on Structured Programming"* (Academic Press, 1972); y se basa en un teorema elaborado por C. Böhm y G. Jacopini, presentado en mayo de 1966 en las *Communications of the ACM vol. 9 nº 5, pg. 366-371*, en su paper *"Flow Diagrams, Turing Machines and Languages with only two formation rules"*. Este teorema, también llamado *Teorema Fundamental de la Programación Estructurada*, demuestra que todo **programa propio**[2] puede escribirse utilizando únicamente las siguientes tres estructuras básicas de control:

- ⇨ **Secuencia:** el bloque secuencial de acciones, que son ejecutadas sucesivamente una a continuación de la otra.
- ⇨ **Selección:** es una birfucación condicional del flujo de ejecución, con doble alternativa, de la forma ***"SI condición ENTONCES haga-acción-1 SINO haga-acción-2"***.

[2] **Programa propio** es aquel que:
- Tiene un único punto de entrada y un único punto de salida.
- Existen caminos desde la entrada hasta la salida que pasan por todas las partes del programa.
- Todas las instrucciones son ejecutables y no existen bucles sin fin.

- ⇨ **Iteración:** la repetición condicional ***"MIENTRAS condición HAGA acción"***, que ejecuta la acción (o acciones) repetidamente mientras se cumpla la condición.

Los programas que utilizan sólo estas tres instrucciones básicas de control, o sus variantes se denominan **"programas estructurados"**.

Lo expresado es la idea clásica de lo que se entiende por *programación estructurada,* que hasta la aparición de los nuevos paradigmas (como la *programación orientada a objetos*) se convirtió en la forma de programar más extendida.

Una característica importante de un *programa estructurado* es que puede ser leído en secuencia, desde el comienzo hasta el final, sin perder la continuidad de la tarea que cumple el programa, que es lo opuesto a lo que ocurre con otros estilos de programación. Este hecho es importante debido a que es mucho más fácil comprender completamente el trabajo que realiza una acción determinada si todas las otras acciones que influyen en su quehacer están físicamente contiguas y encerradas en un bloque. La facilidad de lectura, de comienzo a fin, es una consecuencia de utilizar solamente las tres estructuras de control y de eliminar la instrucción de transferencia incondicional de control, la ya casi inexistente ***goto***.

2.1.2 Visión moderna: La Segmentación

La realización de un algoritmo sin seguir alguna técnica establecida de desarrollo produce frecuentemente un conjunto de instrucciones cuya ejecución es compleja de seguir y entender, pudiendo hacer casi imposible la depuración de errores y la introducción de mejoras. Se puede incluso llegar al caso de tener que abandonar el código preexistente porque resulta más fácil empezar todo de nuevo.

Cuando se habla de *programación estructurada*, en la actualidad no se hace en referencia al control del flujo de ejecución **sino a la división de un algoritmo en partes más manejables**, usualmente denominadas *segmentos*, *unidades* o *módulos*.

Así, la visión moderna de un algoritmo estructurado es la de una *composición de segmentos o módulos*, los cuales pueden estar constituidos por unas pocas acciones o por muchas de ellas. Cada *segmento* tiene solamente una entrada (comienzo de su ejecución) y una salida (fin de su ejecución), y asumiendo que no existen repeticiones infinitas y que no se tienen acciones que jamás se ejecuten, cada segmento cumple con las propiedades de un *programa propio*[2].

En una segmentación realizada correctamente, cada uno de estos módulos encerrará acciones y datos que están íntimamente relacionados por su significado y/o por su función. En una partición correcta resulta simple e intuitivo comprender lo que debe hacer cada módulo. En este mismo contexto, la *comunicación entre segmentos se llevará a cabo de una manera cuidadosamente controlada*. Así, una correcta partición del problema *producirá una nula o casi nula interdependencia entre los módulos*, pudiéndose entonces trabajar con cada uno de ellos de forma independiente. El carácter autocontenido de los módulos hace que puedan ocultarse las funcionalidades internas comprendidas en cada

uno de ellos y solo exponer su operación mediante sus entradas, sus salidas y los resultados logrados sobre ellas. Esto asegura que si se producen modificaciones o cambios que afecten a una sección del algoritmo, ya sea durante el desarrollo o el mantenimiento, estos no afectarán al resto del algoritmo que no ha sido modificado.

Esta técnica de análisis y diseño tiene varias ventajas:

- ⇨ El costo de resolver varios subproblemas relativamente pequeños de forma aislada es, con frecuencia, menor (y de hecho, mucho menor) que el de abordar el problema en forma global.
- ⇨ Facilita el trabajo simultáneo de distintos grupos.
- ⇨ Posibilita un mayor grado de reutilización de algunos módulos en la solución de futuros problemas.

Aunque no puede definirse anticipadamente la cantidad y tamaño de los módulos, siempre se debe buscar el balance entre ambos factores. Por ejemplo, un módulo de gran tamaño probablemente podrá ser dividido nuevos módulos más fáciles de manejar. Así, esta secuencia de divisiones toma la forma de un árbol cuya raíz es el programa principal que implementa la solución al problema, y que utiliza uno o más módulos que realizan diferentes secciones de la solución por sí solos, o invocando a su vez a otros módulos que resuelven subproblemas más específicos. Esta metodología de trabajo no es otra cosa que el ya conocido ***diseño top-down***, y su aplicación a la programación de computadoras fue manifestada por *Niklaus Wirth* en su artículo *Program Development by Stepwise Refinement* en *Communications of the ACM, Vol. 14 Number 4, 1971*.

Ambas visiones, la clásica y la moderna, confluyen en el hecho de que los segmentos de la visión moderna se combinan utilizando las estructuras básicas de control de flujo presentes en la visión clásica y por lo tanto, ***el resultado también es un algoritmo estructurado***.

2.2 La computadora como Procesador

Habiendo definido en general los conceptos de *procesador*, *ambiente*, *acción*, *condición* y *algoritmo*; se analizarán a continuación las condiciones particulares que se tienen cuando el procesador es una *computadora*. En este caso el ambiente estará formado por los recursos de la misma, por ejemplo: memoria, pantalla, teclado, impresora, unidad de disco, etc.

2.2.1 Constantes y Variables

El elemento mas utilizado por la CPU en la resolución de un algoritmo es la memoria, la que está dividida en lugares donde el procesador deposita o extrae información. El contenido de estos "lugares" puede permanecer inalterable durante la ejecución de un algoritmo o puede ir cambiando su valor a lo largo de la ejecución. A los primeros se les denomina **"constantes"** y a los segundos, **"variables".**

Una constante es un objeto cuyo valor no puede variar.

Una variable es un objeto cuyo valor puede cambiar y además posee los siguientes atributos:

- ***un nombre***
- ***un tipo***
- ***un valor***

Cuando se define una variable es necesario precisar su *nombre* y su *tipo*. **Definir** una variable es reservar una o mas posiciones de memoria para esa variable, donde la cantidad de lugares de memoria a reservar depende del *tipo* de variable.

Los tipos de datos primitivos que pueden manejar la mayoría de los procesadores son:

- numéricos
- lógicos
- carácter

2.2.2 Tipo Numérico

Como su nombre lo indica puede representar el conjunto de los valores numéricos, y en general, en dos formas distintas:

- entero.
- real.

2.2.2.1 Tipos Numéricos Enteros

Este tipo es un subconjunto finito de los números enteros, y pueden representarse todos los números enteros comprendidos entre los extremos máximo y mínimo que acepta la computadora, los cuales son -32768 y 32767. La determinación de estos valores máximos y mínimos se debe a que normalmente la computadora asigna dos bytes de memoria (16 bits) para representar los valores enteros, pero esto puede variar según las características de la CPU utilizada.

2.2.2.2 Tipo Numérico Real

El tipo real es un subconjunto de los números reales, y en este tipo es donde se representan los números con **punto decimal**. Generalmente, las computadoras utilizan la representación de punto flotante según la norma IEEE-754, donde se definen una cierta cantidad de dígitos significativos y un exponente (este es el concepto de notación científica). La computadora almacena los números reales en forma similar a la descripta: la parte fraccionaria (o mantisa) y la parte exponencial.

Por ejemplo el número **0.00543820** con ocho dígitos de precisión en el formato de punto flotante normalizado se expresa como $0.54382000 * 10^{-2}$

2.2.3 Tipo Lógico o Booleano

Este tipo es el conjunto de los dos valores lógicos posibles, ***verdadero*** y ***falso***.

V → verdadero **F** → falso

2.2.4 Tipo Carácter

Cada computadora reconoce un conjunto finito y ordenado de caracteres pero, en general, todas las computadoras pueden manipular:

a) el conjunto de las letras mayúsculas.
b) el conjunto de las letras minúsculas.
c) conjunto de los dígitos decimales.
d) el carácter de espacio en blanco.
e) caracteres especiales como *****, **+**, **-**, **$**, etc.

Una constante literal de tipo carácter se escribe entre apóstrofes para no confundirla con los nombres de variables, operadores, enteros, etc.

Por ejemplo si **X** es una variable entera e **Y** una variable de tipo carácter, se tiene:

X = 8 → a la variable numérica **X** se le asigna el número **8**.

Y = '8' → a la variable carácter **Y** se le asigna el carácter **'8'** (ASCII 56_{10}).

La diferencia reside en que con **X** se pueden realizar todas las operaciones aritméticas mientras que con **Y** sólo las operaciones permitidas con caracteres.

Si se llegara a definir:

Y = 8

Se estaría cometiendo un error ya que a una variable carácter **Y** se le estaría asignando un valor numérico, pero también sería incorrecto si se le asignara una sucesión de caracteres, por ejemplo,

Y = 'abc'

2.2.4.1 Tipo Cadena de Caracteres

Una constante literal de tipo Cadena de Caracteres es una secuencia finita (y posiblemente no-vacía) de caracteres, encerrados entre comillas.

Ejemplo:

"$1255.20" "Informática"

La longitud de la cadena se define como la cantidad de caracteres delimitados por las comillas, incluyendo los espacios en blanco. Una variable capaz de almacenar una cadena de caracteres se denomina *variable cadena de caracteres* o simplemente, ***string***.

2.3 Expresiones

Una expresión describe un cálculo a efectuar por el procesador y cuyo resultado es único.

Una expresión contiene ***operandos*** y ***operadores***, y según el tipo de datos involucrados se clasifican en:

- Expresiones aritméticas
- Expresiones relacionales
- Expresiones lógicas
- Expresiones carácter

El resultado de una expresión aritmética es del tipo numérico, el correspondiente a una expresión relacional o a una expresión lógica es del tipo lógico, y el resultado de una expresión carácter es del tipo carácter o cadena.

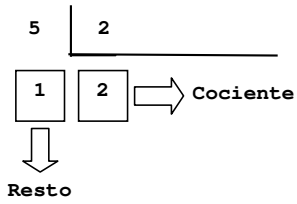
2.3.1 Expresiones Aritméticas

Las expresiones aritméticas se forman utilizando los *operadores* indicados en la Tabla 6.1. Los *operandos* utilizados en estas expresiones pueden ser:

- Constantes numéricas.
- Variables numéricas.
- Expresiones aritméticas entre paréntesis.

Operador Aritmético	Función
+	Suma
-	Resta
*****	Producto
/	División
^	Potenciación
mod	Resto de la división entera
div	División entera
( )	Paréntesis para agrupar expresiones

Los primeros cinco operadores no requieren explicación, pero esta sí es necesaria para el operador **mod** (llamado también operador *Módulo*) el cual permite calcular el resto de la ***división entera***; y también para el operador **div**, quien calcula el resultado de la ***división entera***. Para estos operadores se debe tener en cuenta lo siguiente:



Si se tienen dos valores enteros 5 y 2, la división entera de estos operandos no da como resultado 2.5, valor que se obtendría en una calculadora. Esto es por que este valor 2.5 no es un entero, y por ello la división entera de 5/2 nos entrega dos resultados: el cociente 2 y el resto 1.

Una aplicación trivial de esta operación esta dada en su utilización para determinar si un número entero es par o no, tal como se analizó en el ejemplo anterior:

Si x **mod** 2 es igual a 0 entonces x es par

Si x **mod** 2 es distinto de 0 entonces x es impar

En cuanto al signo, si uno de los operandos es negativo, el signo del resto resultará de aplicar las reglas de los signos del álgebra, teniendo en cuenta que siempre debe cumplirse que, si **A** es el dividendo y **B** el divisor, entonces:

A = (A **div** B) * B + (A **mod** B)

Por ejemplo si **A = -5** y **B = 2**

La operación resto (A **mod** B) dará por resultado **-1** para que se cumpla la expresión anterior.

2.3.2 Reglas con las que se evalúa una expresión

a) todas las cantidades que están encerradas entre paréntesis se evalúan primero, evaluándose primero las más internas en el caso de existir paréntesis anidados.

b) las operaciones aritméticas dentro de una expresión, se ejecutan de acuerdo con el siguiente orden de precedencia:

Orden	Operador	Significado
1º	**^**	Se aplica de Derecha a Izquierda
2º	***, / , div, mod**	Se aplica de Izquierda a Derecha
3º	**+, -**	Se aplica de Izquierda a Derecha

Ejemplo:
Indicar en que orden se realizan las operaciones y ¿cual es el resultado de la siguiente expresión?

$$10 - 7 + 3 * ((\underbrace{52 - 12}_{40}) / 5)$$

$$10 - 7 + 3 * \underbrace{(40 / 5)}_{8}$$

$$10 - 7 + \underbrace{3 * 8}_{24}$$

$$\underbrace{10 - 7}_{3} + 24$$

$$\underbrace{3 + 24}_{27}$$

$$27$$

Además de las operaciones básicas como suma, resta, producto, división y potencia puede existir otro conjunto de operadores especiales que se denominan funciones internas y algunos ejemplos se listan en la siguiente tabla:

Estas operaciones no están disponibles en LPP

Nombre de la Operación	Significado
ABS	Valor Absoluto
RC	Raíz Cuadrada
LOGN	Logaritmo Natural
LOG10	Logaritmo Decimal
EXP	Exponencial (base e)
SEN	Seno
COS	Coseno
TAN	Tangente
TRUNC	Truncamiento
REDON	Redondeo

2.3.3 Expresiones Relacionales

Las expresiones relacionales permiten realizar comparaciones entre valores del mismo tipo, aunque se debe tener en cuenta que ***el resultado de una expresión relacional es siempre de tipo lógico*** (también llamado *Booleano*), es decir, su resultado será Verdadero (**V**) o Falso (**F**). A las funciones preposicionales se les suele denominar ***predicados***.

Operador Relacional	Significado
=	Igual
<	Menor
<=	Menor o igual
>	Mayor
>=	Mayor o igual
< >	Distinto

Ejemplo:
Si A y B son dos variables numéricas cuyos valores son respectivamente, 3 y 15, la siguiente tabla muestra el cálculo de algunos predicados:

Predicado	*Valor*
A > B	**f**
A = B	**f**
A < B	**v**
A < > B	**v**

También es posible la comparación entre cadenas de caracteres, en este caso la comparación se efectúa en orden alfabético, caracter por caracter y de izquierda a derecha. De esta manera la constante "TOMA**S**" es menor que la constante "TOMA**TE**" ya que la primer diferencia aparece en la letra **S** de **TOMAS** versus la segunda letra **T** de **TOMATE**. Como la **S** es menor que la **T** (su valor ASCII es menor), entonces **TOMAS** es menor que **TOMATE**.

Las expresiones relacionales pueden combinarse mediante los llamados conectores u ***operadores lógicos*** para formar predicados compuestos.

2.3.4 Expresiones Lógicas

Los operadores lógicos utilizados en expresiones lógicas son:

Operador Lógico	Equivalente Lógico	Significado
Y	**AND**	Conjunción o Producto Lógico
O	**OR**	Disyunción Inclusiva o Suma Lógica
NE	**XOR**	Disyunción Exclusiva
NO	**NOT**	Negación

pero los operadores **NO** y **NE** no están disponibles en el lenguaje **LPP**.

Los resultados de las operaciones lógicas es un valor **verdadero** o **falso** que depende de las leyes del **Álgebra de Boole**, y pueden evaluarse de manera sencilla utilizando las siguientes *tablas de verdad*, donde **A** y **B** son proposiciones lógicas, y **v**/**f** son verdadero y falso respectivamente.

Suma Lógica

A	B	A **O** B
v	**v**	**v**
v	**f**	**v**
f	**v**	**v**
f	**f**	**f**

Producto Lógico

A	B	A **Y** B
v	**v**	**v**
v	**f**	**f**
f	**v**	**f**
f	**f**	**f**

Suma Exclusiva

A	B	A **NE** B
v	**v**	**f**
v	**f**	**v**
f	**v**	**v**
f	**f**	**f**

Negación Lógica

A	**NO** A
v	**f**
f	**v**

El orden de precedencia de los operadores lógicos es:

Orden	Operador
1º	**NO**
2º	**Y**
3º	**O y NE**

2.3.5 Expresiones Cadena de Caracteres

La operación básica con cadenas de caracteres consiste en tomar dos cadenas y unirlas entre sí transformándolas en una sola. Esta operación se denomina ***concatenación***, y el operador se simboliza con el signo **+**. Este operador es asociativo, y el efecto de una *concatenación* se muestra en el siguiente ejemplo:

"Programa"
"cion Estructurada"

"Programa" **+** "cion Estructurada" da por resultado "Programacion Estructurada"

2.4 Asignación de Variables

Para asignar a una variable un valor que proviene de un ambiente dado, utilizaremos la siguiente notación:

nombre_de_la_variable ← valor

Vn ← E

Donde:

- **Vn** es el nombre de la variable a la que el procesador le va a asignar el valor de **E**.
- **←** es el signo de la asignación formado por el símbolo "**<**" seguido del símbolo "**-**"
- **E** representa el valor que se le va a asignar a la variable.

Y la expresión **Vn ← E** debe leerse como: *"guarde en la variable **Vn** el valor contenido en la constante (o variable o expresión) **E**"*.

Según sean el tipo de variables **Vn** y **E**, la acción de *asignar* se puede clasificar en:

- asignación aritmética
- asignación lógica
- asignación de carácter
- asignación Cadena de Caracteres

2.4.1 Asignación Aritmética

V ← E Será una asignación aritmética si:

V Es una variable de tipo numérico.

E Es una constante de tipo numérico, una variable de tipo numérico o una expresión aritmética.

2.4.2 Asignación Lógica

V ← E Será una asignación lógica si:

V Es una variable lógica.

E Es una constante lógica, una variable lógica, una expresión relacional o una expresión lógica.

2.4.3 Asignación Carácter

V ← E Es una asignación carácter si:

V Es una variable de tipo carácter.

E Es una constante de tipo carácter o una variable de tipo carácter.

2.4.4 Asignación Cadena de Caracteres

V ← E Será una asignación cadena de caracteres si:

V Es una variable tipo cadena de caracteres.
E Es una constante tipo cadena de caracteres, una variable tipo cadena de caracteres, una expresión tipo cadena de caracteres o una expresión tipo carácter.

2.5 Operaciones de Entrada/Salida

2.5.1 Entrada de Datos

Un valor que sea externo al ambiente puede introducirse en éste mediante una acción conocida por el procesador y denominada **Lectura**.

Lectura: **Es toda acción que permite la entrada de un valor al ambiente a través de un dispositivo.**

La lectura de un dato es considerado una asignación encubierta, ya que el valor leído del medio externo es asignado a una de las variables del ambiente. La lectura es una acción primitiva que se escribe de la siguiente forma:

```
Lea V
```

Donde **V** es una variable del ambiente que recibe un valor de un dispositivo, por ejemplo, desde un teclado.

2.5.2 Salida de Datos

Un valor del ambiente puede "salir" del mismo para comunicar alguna información al medio externo, por ejemplo a través de una pantalla o de la impresión sobre un papel, exteriorizando algún resultado.

Escritura: **Es toda operación que permite la salida de un valor del ambiente a través de un dispositivo.**

La escritura también es una acción primitiva y se escribe de la siguiente forma:

```
Escriba V
```

Donde **V** es la variable, cuyo valor se desea transferir al medio externo.

Los siguientes ejemplos muestran su uso:

```
Escriba A,B,C          /*Escribirá el valor de las variables A, B y C en "Pantalla"*/
Escriba "Informática I" /*Escribirá el valor Informática I en "Pantalla" */
Escriba "El Promedio es:",P /*Escribirá el valor constante: El valor del
                              Promedio es: y a continuación el valor
                              numérico correspondiente a la variable P
                              en "Pantalla" */
```

2.6 Estructuras de Control de Flujo

La potencia de un procesador proviene en gran parte de su capacidad para tomar decisiones, estableciendo que acción debe realizar en un determinado paso de la

ejecución de un algoritmo, dependiendo ya sea de los valores de los datos que se leen o bien de los resultados de los cálculos que se han realizado en pasos anteriores. Según lo visto en el apartado 2.1, existen dos conceptos importantes para el control de flujo:

1. ***La selección de una acción entre un conjunto de alternativas específicas.***
2. ***La repetición de un conjunto de acciones.***

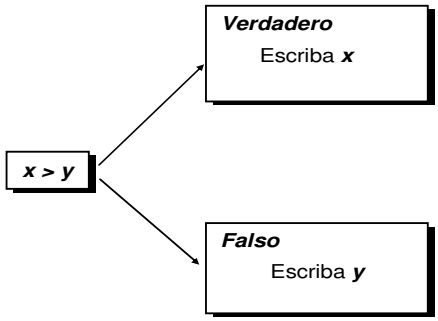
Dentro del flujo de control se tiene el *flujo lineal de control*, es decir, la ejecución secuencial de acciones, desde la primera hasta la última, y las *estructuras de control* que son las que permiten apartarse de este flujo lineal o secuencial. Una de estas estructuras le permite al procesador repetir automáticamente un grupo de acciones, como por ejemplo el esquema "*Repita-hasta que*" visto en ejemplos anteriores, mientras que otras permiten seleccionar una acción entre varias posibles, según se evalúen determinadas condiciones. De esta manera se tienen, en el contexto de la Programación Estructurada, dos tipos de estructuras de control bien diferenciadas:

- **Estructuras de Decisión**
- **Estructuras de Repetición o de Lazo**

2.6.1 Estructuras de Selección

2.6.1.1 Estructura de selección simple: si-fin si

Supongamos tener dos variables numéricas, **x** e **y** cuyos valores son diferentes y se requiere escribir el mayor de ellos. Si **x** es mayor que **y** debe escribirse el valor de **x**, de lo contrario, si **y** es mayor que **x** debe escribirse el valor de **y**. El siguiente gráfico representa esta estructura:



La expresión **x > y** es un *predicado* que describe la condición que se desea evaluar: Si **x > y** es *verdadera* se ejecutará la acción **Escriba x**, pero si **x > y** es *falsa*, se ejecutará la acción **Escriba y**. Este ejemplo muestra una de las estructuras más importantes en el diseño de algoritmos: ***la selección de una acción entre dos alternativas dependiendo de una condición***. Esta estructura algorítmica se denomina: construcción **"si-entonces-sino-fin si"** y su formato es el siguiente:

si (condición) entonces
acciones de la alternativa verdadera
sino
acciones de la alternativa falsa
fin si

La construcción comienza con la palabra **`si`**, seguida de la condición que se debe evaluar. Si la condición es ***verdadera*** la alternativa que debe ejecutarse será la que se encuentra precedida por la palabra **`entonces`**. En caso contrario, si la condición resulta ***falsa***, se ejecutará la alternativa precedida por la palabra **`sino`**. En cualquier caso la estructura siempre debe finalizarse con **`fin si`**. A las palabras **`si`, `entonces`, `sino`, `fin si`**, se les llama ***delimitadores***, y el ejemplo anterior tomará la siguiente forma:

si **(x > y)** ***entonces***
escriba x
sino
escriba y
fin si

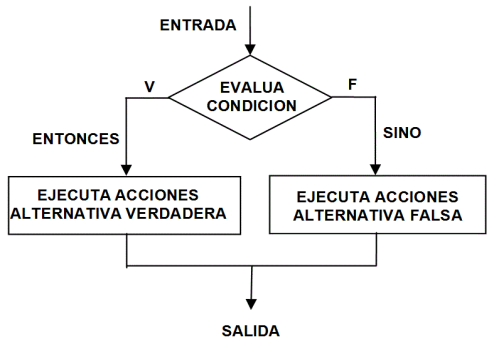
Es importante considerar que tanto la condición a evaluar como cada una de las alternativas pueden ser mucho más complejas: la condición puede estar expresada mediante predicados compuestos, y cada alternativa puede estar constituida por varias acciones primitivas.

A esta estructura de decisión se la considera como una entidad completa, a la que se *"ingresa por arriba"* y de la que se *"egresa por debajo"*, y es importante resaltar este concepto básico de la Programación Estructurada: **la existencia de un único punto de entrada y un único punto de salida**[2].

En algunas oportunidades la sección correspondiente a la alternativa falsa puede **no estar presente**, y este caso particular ocurre si no existen acciones a ejecutar cuando la condición es falsa. En estas condiciones no aparecerá el delimitador **sino** y el formato final quedará:

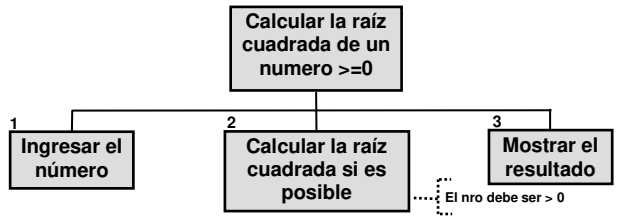
```
si (condición) entonces
     alternativa verdadera
fin si
```

A continuación se muestra un flujograma de esta estructura completa:



y el siguiente ejemplo ilustra el caso de la ausencia de la acción por el camino de la alternativa ***falsa***.

Problema 1: Calcular la raíz cuadrada de un número si éste no es negativo.

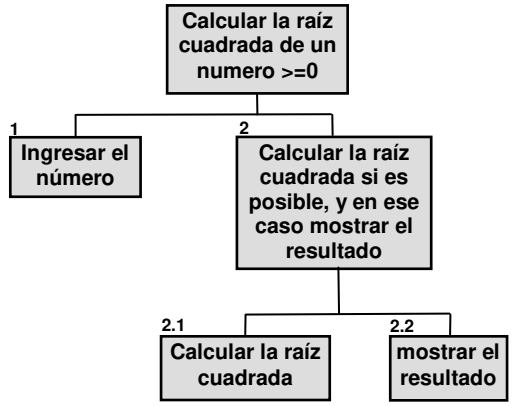


Solución: Calcular la raíz cuadrada de un número si su valor es no-negativo y mostrar el resultado.

```
    real dato, raiz
    Inicio
        escriba "Ingrese el dato a procesar:"
1       lea dato
        si (dato >= 0) entonces
2            raiz <-- RC( dato )
        fin si
3       escriba raiz
    Fin
```

El problema con esta solución ocurre cuando se ingresa un número negativo ya que la estructura **`si`** elegirá el camino del ***falso*** y no se calculará la **`raíz`** pero sí se intentará mostrar su valor, lo que no dará un resultado coherente ya que la variable **`raíz`** no ha sido inicializada.

Esta falla se debe a que en la etapa de análisis no se ha considerado que la tarea de mostrar que el valor de la raíz es dependiente de su cálculo efectivo, por lo que los **nodos 2** y **3** deben reunirse en uno solo y no mantenerse por separado. Un nuevo diagrama descendente permite solucionar este inconveniente:



Lo que conduce al siguiente algoritmo:

```
real dato, raiz
Inicio
    escriba "Ingrese el dato a procesar:"
1   lea dato
    si (dato >= 0) entonces
2.1 raiz <-- RC( dato )
2.2 escriba raiz
    fin si
fin
```

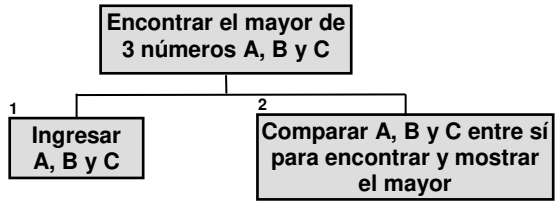
2.6.1.2 Anidamiento de estructuras de selección

Tanto la alternativa verdadera como la falsa pueden contener a su vez estructuras de decisión en su cuerpo. El siguiente problema muestra un ejemplo de tal situación:

Problema 2: Se dispone de tres números **A**, **B** y **C**, y se requiere encontrar el mayor de ellos.

Solución: Se compara **A** con **B**; si **A** es mayor que **B** entonces, se compara **A** con **C**, si **A** es mayor que **C** entonces, el mayor es **A**, sino el mayor es **C**. Si **A** no es mayor que **B** entonces **B** es mayor que **A** y se deben comparar **B** con **C** para determinar el mayor.

El análisis Top-Down resulta:



Ya que el **nodo 2** es un conjunto de comparaciones, es imposible continuar la descomposición en el diagrama y se debe recurrir a un ***índice descendente***.

2.1 Si **A** es mayor que **B** hay que comparar **A** con **C**.
2.2 Entonces, si **A** es mayor que **C,** el mayor es **A** y en caso contrario, el mayor es **C**.
2.3 pero si **A** no es mayor que **B** (**B** es mayor que **A**) hay que comparar **B** con **C**.
2.4 Entonces, si **B** es mayor que **C,** el mayor es **B** y en caso contrario, el mayor es **C**.

Y ahora es posible codificar el algoritmo:

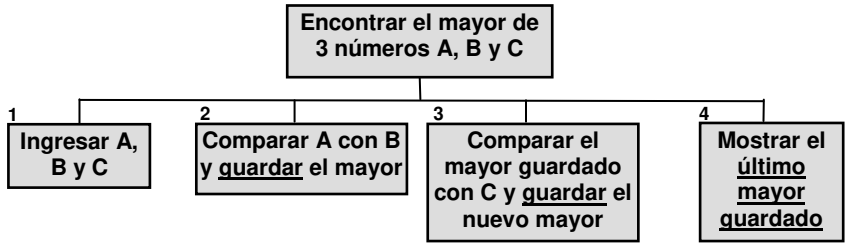
```
real A, B, C
Inicio
    Escriba "Ingrese los datos a procesar:"
    lea A, B, C
    si A > B entonces
         si A > C entonces
              Escriba "el máximo valor es:", A
```

```
            sino
                  Escriba "el máximo valor es:", C
            fin si
      sino
            si C > B entonces
                  Escriba "el máximo valor es:", C
            sino
                  Escriba "el máximo valor es:", B
            fin si
      fin si
Fin
```

Si bien este algoritmo es una solución completamente válida y operativa al problema planteado, resulta bastante visible su complejidad debido a las selecciones simples anidadas, y es posible anticipar la complejidad resultante si fuese necesario encontrar el mayor valor para 4 o más números, además de tener que *modificar el algoritmo completo* cuando se necesite procesar esta mayor cantidad de números.

Para solucionar estos inconvenientes es posible utilizar una técnica alternativa, que aunque también se basa en el mismo principio de comparaciones sucesivas, emplea una estructura diferente. Esta nueva solución puede describirse como:

Solución: Se compara **A** con **B** y se *almacena* el **mayor** para proceder a la próxima comparación. A continuación se compara este **mayor** con **C** y se *almacena* el nuevo **mayor** resultante, que será el mayor definitivo. El análisis Top/Down de esta propuesta resulta:

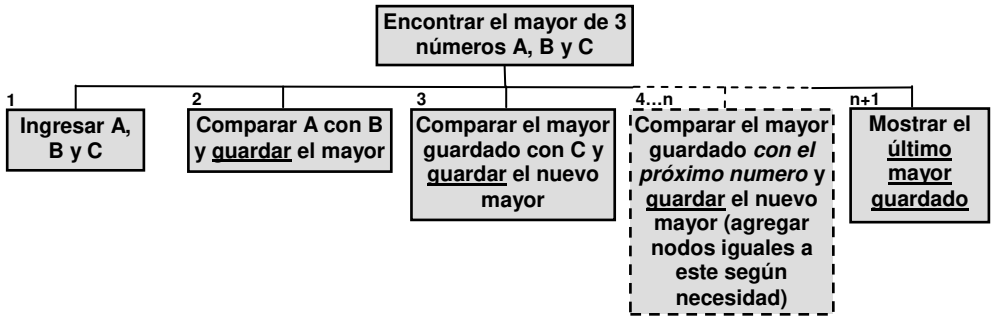


Evidentemente, los **nodos 2** y **3** ahora son *selecciones simples* ***no-anidadas*** lo cual simplifica la estructura final del algoritmo. La acción *guardar* da cuenta de la necesidad de resguardar transitoriamente el mayor valor encontrado. La codificación de este algoritmo sería:

```
      real A, B, C, mayor
      Inicio
1           Escriba "Ingrese los datos a procesar:"
            lea A, B, C
            si A > B entonces
                  mayor ← A
2           sino
                  mayor ← B
            fin si
```

```
        si C > mayor entonces
3            mayor ← C
        fin si
4       Escriba "el máximo valor es:", mayor
    Fin
```

Ahora resulta evidente que si fuera necesario encontrar el mayor entre una cantidad de números más elevada, la complejidad estructural no aumenta como en el caso anterior puesto que ahora solo se requiere agregar módulos iguales al **nodo 3** para comparar cada uno de los números adicionales, lo que resultaría en:



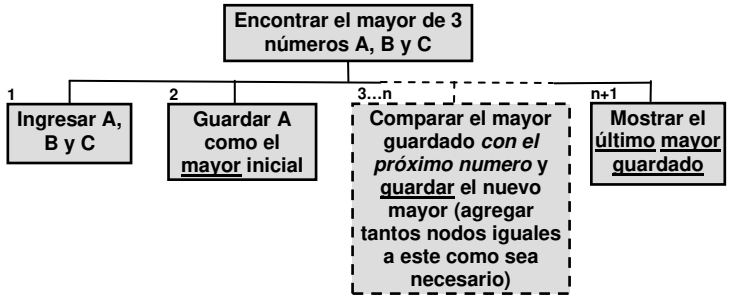
Si se analizan los nodos de los últimos esquemas descendentes se puede apreciar que los **nodos 2** son diferentes de los **nodos 3** (y sus posibles clones **4…n**) ya que el primero es una selección simple con evaluación de la alternativa *falsa* mientras que los segundos no evalúan esta alternativa, lo que se traduce en:

```
si A > B entonces
     mayor ← A
sino
     mayor ← B
fin si
```

versus

```
si C > mayor entonces
     mayor ← C
fin si
```

Aunque esta *asimetría* no constituye un problema, puede eliminarse y así uniformar la estructura de las diferentes selecciones, lo que se logra suponiendo que el *primer dato leído es el* **máximo** *inicial*. Esto lleva a:



y la codificación correspondiente será:

```
real A, B, C, mayor
Inicio
1   Escriba "Ingrese los datos a procesar:"
    lea A, B, C
2   mayor ← A
3   si B > mayor entonces
        mayor ← B
    fin si
4   si C > mayor entonces
        mayor ← C
    fin si
5   Escriba "el máximo valor es:", mayor
Fin
```

← Obsérvese que todas las comparaciones ahora tienen exactamente la misma estructura.

Esta última estructuración de las comparaciones permite generar un algoritmo estándar para encontrar el ***máximo*** (y también el ***mínimo***) de un conjunto de valores con cualquier cantidad de elementos, tal como se describe en 2.7.5

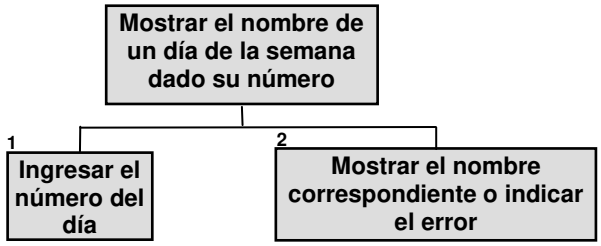
2.6.1.3 Estructura de selección generalizada

Existe una estructura de decisión de uso más cómodo para algunas aplicaciones que se denomina *estructura de decisión generalizada* o también *estructura de decisión múltiple* y su forma es la siguiente:

```
caso E
    E1:   conjunto de acciones 1
    E2:   conjunto de acciones 2
    …
    Ek:   conjunto de acciones k
    sino:     conjunto de acciones B
fin caso
```

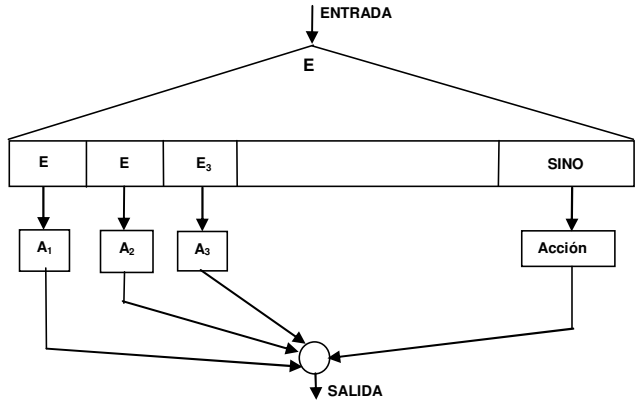
En esta estructura, **E** es una **expresión <u>numérica entera</u>**, cuyo resultado puede ser uno de los valores $\mathbf{E_1, E_2, ...E_k}$, y en función de estos valores **se ejecutará uno y sólo uno** de los conjuntos de acciones indicados. Si por ejemplo **E** tiene el valor $\mathbf{E_i}$, se ejecutará el conjunto **i**. En caso de que el valor de **E** sea distinto de $\mathbf{E_1, E_2, ...E_k}$, se ejecutará el conjunto de acciones **B** (la etiqueta **`sino:`** es opcional y solo debe utilizarse si fuera necesario).

Problema 3: Se necesita diseñar un algoritmo que acepte un número entero comprendido entre 1…7 como entrada y que escriba el nombre del día de la semana correspondiente al valor ingresado (**1**→Domingo, **2**→Lunes, etc.). Si se ingresa un valor fuera de rango se debe escribir un mensaje de error advirtiendo la situación.




```
Entero dia
Inicio
     Escriba "Ingrese el numero del dia:"
     Lea dia
     caso dia
          1:    Escriba "Domingo"
          2:    Escriba "Lunes"
          3:    Escriba "Martes"
          4:    Escriba "Miercoles"
          5:    Escriba "Jueves"
          6:    Escriba "Viernes"
          7:    Escriba "Sabado"
          sino:
                Escriba "ERROR!! Numero no valido."
     fin caso
Fin
```

Se puede graficar la estructura **caso** de forma similar a la siguiente, donde $\mathbf{A_i}$ representa la acción (o conjunto de acciones) correspondiente al valor de la selección $\mathbf{E_i}$:



Dado que la expresión a evaluar para elegir el conjunto de acciones a ejecutar debe ser entera, la estructura **`caso`** no siempre puede utilizarse en todas las circunstancias. En esos casos debe reemplazársela por una cadena de estructuras de selección simple **`si-fin si`** anidadas tal como se detalla a continuación y que permita evaluar la comparación de los tipos de datos involucrados (típicamente reales y cadenas de caracteres).

```
Entero dia
Inicio
     Escriba "Ingrese el numero del dia:"
     Lea dia
     si (dia = 1) entonces
          Escriba "Domingo"
     sino si (dia = 2) entonces
          Escriba "Lunes"
     sino si (dia = 3) entonces
          Escriba "Martes"
     sino si (dia = 4) entonces
          Escriba "Miercoles"
     sino si (dia = 5) entonces
          Escriba "Jueves"
     sino si (dia = 6) entonces
          Escriba "Viernes"
     sino si (dia = 7) entonces
          Escriba "Sabado"
     sino
          Escriba "ERROR!!!!"
     fin si
     fin si
     fin si
     fin si
     fin si
     fin si
     fin si
Fin
```

Resulta obvia la potencia expresiva de la estructura **`caso`** cuando se la compara contra la cadena de estructuras **`si`** anidadas, por lo que solo debería usarse esta última alternativa cuando sea imposible el uso de la estructura **`caso`**.

2.6.2 Estructuras de Repetición o Iteración

Tal como se describió en 2.1.1, no solo se dispone de las estructuras de selección como medio del control de flujo sino también de las ***estructuras de repetición***, que permite ejecutar múltiples veces a una misma acción o a un conjunto de acciones. Estas estructuras son:

2.6.2.1 Estructura Repita-hasta

Esta estructura ya fue introducida anteriormente en el ejemplo de la distribución de caramelos:

repita

t1.1 tomar un caramelo de la bolsa
t1.2 poner el caramelo en la caja correspondiente

hasta (t1.3) la bolsa esté vacía

y en ella se pueden distinguir dos elementos importantes,

- Un conjunto de acciones que se repiten denominado ***rango del ciclo***, que el ejemplo serían ***t1.1*** y ***t1.2***.
- La ***condición de fin de repetición***, expresada en la forma de un predicado tal como ***t1.3*** en este caso.

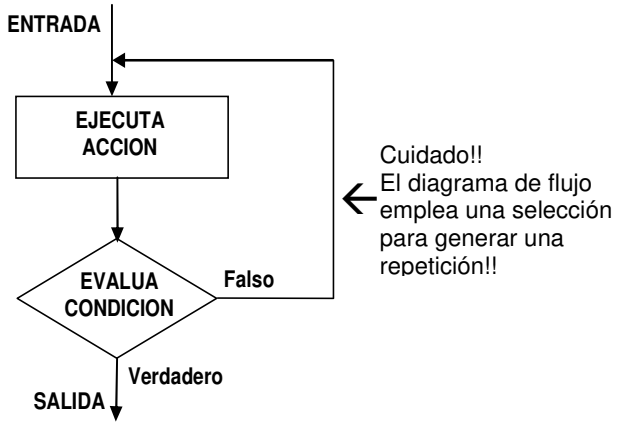
El formato de la estructura es:

```
repita
        acción 1
        acción 2      /* Acciones que forman el cuerpo (rango del ciclo) del repita */
        .....
hasta (condición de salida)
```

El predicado es evaluado ***después*** de cada ejecución del rango y por lo tanto es importante tener en cuenta que ***Todas las acciones del rango del ciclo son ejecutadas por lo menos una vez***.

Las acciones deberán repetirse en tanto el predicado se mantenga falso ya que cuando el predicado resulte verdadero, se dará por finalizada la ejecución de la repetición. Al igual que las anteriores, se considera a esta estructura con un único punto de entrada y una única salida.

La representación gráfica de esta estructura es la siguiente:

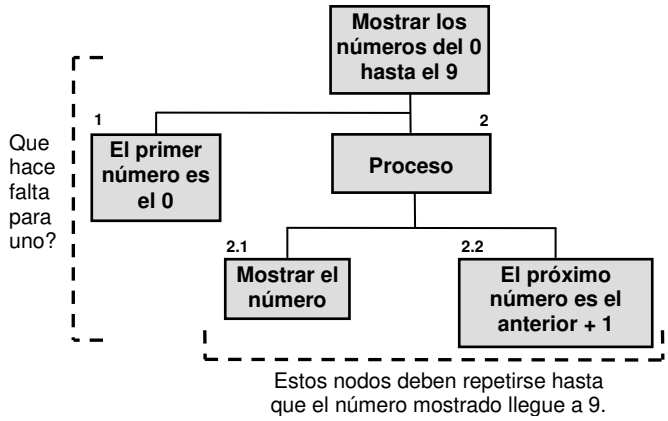


Entre las características particulares de este esquema de repetición podemos citar las siguientes:

- Dentro del conjunto de acciones que forman el rango del lazo debe haber al menos una que esté relacionada con el predicado de tal manera que se pueda modificar su resultado. De lo contrario las acciones se repetirán indefinidamente (*lazo o bucle infinito*).
- El predicado es *"de salida"* ya que en él está expresada ***la condición que hace que el procesador finalice la repetición***.
- En esta estructura de repetición no se conoce anticipadamente el número de repeticiones que se van a llevar a cabo.

Problema 4: Escribir los dígitos del 0 al 9.

Solución: A partir de una variable entera, que inicialmente debe estar en cero, en cada iteración se irá escribiendo el valor actual de la variable y luego se la incrementará. La salida del lazo se produce cuando el valor alcanzado por la variable supere nueve.




```
entero digito
Inicio
     digito ← 0            /* Inicializa la variable*/
     repita
          Escriba digito
          digito ← digito +1
     hasta (digito > 9)     /* condición de salida */
Fin
```

Es importante recordar que el diagrama descendente del análisis top-down **no admite** la presencia de estructuras de control ya que es un **diagrama de tareas** y por ende, es imposible indicar tareas que deben repetirse. Sin embargo, dado que es factible agregar comentarios al diagrama descendente, estos se transforman en una herramienta muy útil para indicar los nodos que deben repetirse en base a alguna condición, tal como se ha hecho en el diagrama anterior.

2.6.2.2 Estructura Mientras-haga-fin mientras

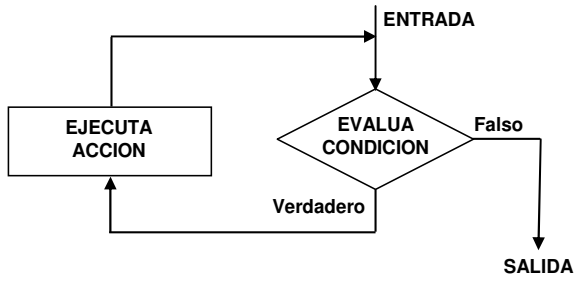
Como se observó en el ejemplo de los caramelos descripto anteriormente, el esquema de repetición **"Repita-hasta"** sólo era aplicable si se tenía la certeza de que la bolsa contuviese cuatro o más caramelos, lo que implicaría ejecutar **al menos una vez** las acciones del lazo. La estructura **"mientras-haga-fin mientras"** es similar a la anterior pero con la diferencia de que *la condición se evalúa antes de ingresar a la estructura de repetición*. Su formato es:

mientras (condición para entrar o quedarse en el lazo) haga

acción 1
acción 2 /* Acciones que forman el rango del ciclo del mientras */

…
acción n
fin mientras

Esta estructura también se considera como un bloque con un único punto de entrada y un único punto de salida.



En este esquema, el delimitador **`fin-mientras`** indica el fin del conjunto de acciones a repetir, y la cantidad de veces que se ejecutarán las acciones *no es conocida de antemano*. El predicado de control siempre es evaluado **antes** de la ejecución de la secuencia de acciones que forman el cuerpo del mientras: Si resulta verdadero la secuencia será ejecutada y si resulta falso se finalizará la repetición. Es evidente **que las acciones del rango del ciclo mientras podrían no ejecutarse** y que la condición expresada es la que *permite mantener la repetición*.

El ejemplo anterior, utilizando ahora la estructura **mientras**, toma la siguiente forma:

Problema 4: Escribir los dígitos del 0 al 9.

Solución: El análisis Top-Down es análogo al antes expresado, y por ende también lo es la solución, pero en este caso el ingreso al lazo para ejecutar las acciones requiere que el valor de la variable sea menor o igual a nueve.

```
Entero digito
Inicio
    digito ← 0            /* Inicializa la variable*/
    mientras (digito <= 9) haga /* entrada al lazo */
         Escriba digito
         digito ← digito +1
    fin mientras
Fin
```

Si se compara este algoritmo con el obtenido para la estructura **`repita`**, se puede observar que se emplean exactamente las mismas acciones, y que la diferencia está dada en las *condiciones*. Esto se debe a que en un caso la condición es para permanecer en el lazo (estructura **`mientras`**) y en el otro es para salir del mismo (estructura **`repita`**).

También puede verse que ante un mismo diseño **Top-Down** se han logrado dos soluciones completamente operativas pero con diferente codificación, lo que

manifiesta una vez más que la codificación es un detalle de implementación y que lo verdaderamente significativo es el análisis y diseño realizado.

Problema 5: Dados dos números naturales que representan el dividendo y el divisor de un cociente se requiere diseñar un algoritmo que calcule el cociente entero y el resto con un procesador para el cual las únicas operaciones aritméticas primitivas son la suma y la resta.

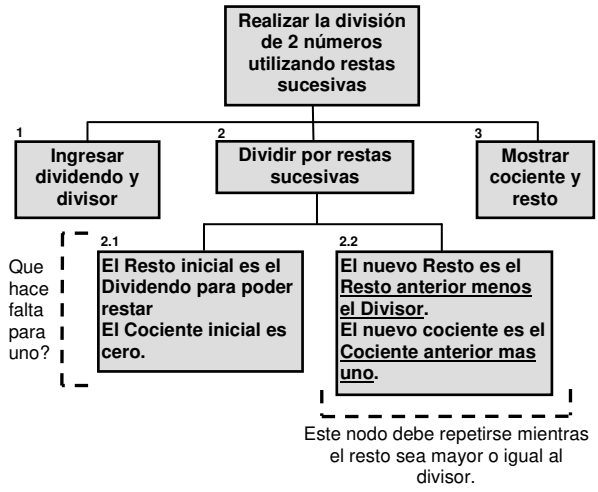
Solución: Para efectuar la división entera se deberá aplicar la técnica de las restas sucesivas, en la que se resta del dividendo el valor del divisor tantas veces como sea posible (el resultado de la resta siempre debe ser >= 0). El **cociente** será la cantidad de restas efectuadas y el **resto** será el resultado de la última diferencia.

DVDO Variable de entrada, tipo numérico entero, cuyo valor representa el **dividendo**.

DIV Variable de entrada, tipo numérico entero, cuyo valor representa el **divisor**.

Q Variable de salida, tipo numérico entero, en la cual se calcula el **cociente**.

R Variable de salida, tipo numérico entero, en la cual se almacena el **resto**.




```
Entero DVDO, DIV, R, Q
Inicio
    escriba "Ingrese dividendo y divisor"
    lea DVDO, DIV
    R ← DVDO
    Q ← 0
    mientras R >= DIV haga
         R ← R - DIV      /* <-- Restador */
         Q ← Q + 1        /* <-- Contador */
    fin mientras
    escriba "cociente", Q
    escriba "resto", R
Fin
```

2.6.2.3 Estructura Para-haga-fin para

Dado que en algunas aplicaciones resulta útil utilizar un conjunto de acciones que se ejecutan un número predefinido de veces, y *cuando el número de repeticiones es conocido de antemano* se plantea una nueva estructura de repetición, denominada **"para-haga-fin para"**

```
para V ← Vi hasta Vf haga
        acción 1
        acción 2        /* Acciones que forman el cuerpo del Para */
        ....
        acción n
fin para
```

V Es una variable tipo numérico llamada variable de control del lazo, o índice del lazo o bucle

Vi, **Vf** son variables de tipo numérico, constantes de tipo numérico o expresiones aritméticas:

Vi Es el valor inicial de la variable **V**

Vf El valor final a alcanzar por la variable **V**

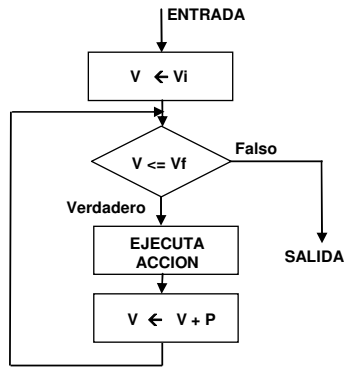
Tal como antes, el conjunto de acciones es el ***rango*** de la estructura de repetición. En este esquema el procesador es el encargado de actualizar la variable de control ***V*** desde su valor inicial ***Vi*** hasta el final ***Vf*** con un incremento ***P***, que en este caso será de uno. El número de veces que el ciclo se repite es $\frac{(V_f - V_i + P)}{P}$ y si este número resultara cero o menor que cero **el ciclo no se ejecutará**.

Es importante resaltar que *en el entorno LPP no existe la posibilidad de utilizar incrementos distintos de 1*, y por lo tanto solamente se puede escribir:

Para V ← V_i hasta V_f haga

siendo $V_f - V_i + 1$ la cantidad total de iteraciones a ejecutar.

La representación gráfica de esta estructura es la de la siguiente figura:



que al igual que las estructuras anteriores, también tiene una sola entrada y una sola salida. Para comprender mejor esta estructura de repetición se puede analizar su implementación utilizando una estructura **mientras**:

```
V ← Vi
mientras ( V <= Vf ) haga
        acción 1
        acción 2        /* Acciones que forman el rango del ciclo para */
        ……..
        V ← V + 1
fin mientras
```

El ejemplo de escribir los diez primeros dígitos utilizando la estructura `para-hacer-fin-para` resultaría en:

```
Entero digito
Inicio
    para digito ← 0 hasta 9 haga
        Escriba digito
    fin para
Fin
```

Como se puede apreciar, la implementación de la solución utilizando la estructura **`para`** es mas sencilla y compacta que las soluciones anteriores, y por lo tanto, cuando se conoce a priori el número de repeticiones su uso es extremadamente conveniente.

2.6.2.4 Ciclos Anidados

Así como dentro de una estructura de decisión se podía incluir otra estructura de decisión, también es posible insertar un ciclo en el interior de otro, y las leyes de anidamiento son similares en ambos casos. ***La estructura interna debe estar totalmente incluida en la externa, no pudiendo existir solapamiento***, y por cada iteración del ciclo **externo** se ejecutarán todas las iteraciones correspondientes del ciclo **interno**.

2.7 Módulos de uso frecuente

Una vez que se conocen las acciones primitivas que puede realizar la computadora cuando se comporta como procesador, es conveniente utilizar algunas de ellas para construir pequeños módulos de uso común en la solución de problemas por programación y que luego servirán como bloques constructivos básicos para resolver numerosos problemas nuevos (**Reconocimiento de Patrones** en el *Pensamiento Computacional*).

2.7.1 Módulo sumador

Este módulo tiene por función realizar la acumulación sucesiva de valores numéricos. Básicamente consta de una variable numérica, de tipo adecuado a los valores numérico a acumular, la que es inicializada en **cero** (***0*** *es el elemento neutro de la suma*) antes de comenzar el proceso. Luego, al valor de esa variable se le sumarán los valores a acumular y al finalizar el proceso, el valor final contenido en esa variable será el resultado de la acumulación de los valores procesados.

```
entero dato, sumador
...
sumador ← 0                          /* Inicializacion */
...
1  sumador ← sumador + dato          /* Posiblemente en una estructura de
                                        repetición.*/
```

Cada vez que se ejecuta la expresión **1**, la variable **sumador** acumula el valor de la variable **dato**, y mantiene así el valor total acumulado hasta el momento.

2.7.2 Módulo multiplicador

Este módulo tiene por función realizar la multiplicación sucesiva de diversos valores numéricos. Básicamente consta de una variable numérica, de tipo adecuado a los valores numéricos a multiplicar, la que es inicializada en **uno** (el ***1*** *es el elemento neutro del producto*) antes de comenzar el proceso. Luego, el valor de esa variable será multiplicado por los valores deseados y el resultado será almacenado también en la misma variable. Al finalizar el proceso, el valor final contenido en esa variable será el resultado de las multiplicaciones sucesivas de los valores procesados.

```
entero dato, multiplicador
...
multiplicador ← 1                    /* Inicializacion */
...
2  multiplicador ← multiplicador * dato   /* Posiblemente en
                                             una estructura de
                                             repetición.*/
```

Cada vez que se ejecuta la expresión **2**, la variable ***multiplicador*** resguarda el resultado de la multiplicación de su propio valor por el contenido de la variable **dato**, y mantiene de esta forma el valor total multiplicado hasta el momento.

2.7.3 Módulo contador o incrementador

La misión de este módulo es realizar el incremento sucesivo, tal como un *contador de personas o ganado*, de una variable generalmente de tipo ***entero***. Este bloque es una variante del *sumador*, con la única diferencia de que siempre acumula **1** (**uno**) en lugar de cualquier otro valor.

```
entero contador
...
contador ← 0                         /* Inicializacion */
...
3  contador ← contador + 1           /* Posiblemente en una estructura de
                                        repeticion.*/
```

Cada vez que se ejecuta la expresión **3**, la variable **contador** aumenta su valor en una unidad, y mantiene de esta forma el valor total *"contado"* hasta el momento.

2.7.4 Módulos restadores, divisores y decrementadores

Estos módulos son complementarios de los tres anteriores, en el sentido de que realizan las operaciones aritméticas *opuestas* a las ya analizadas. De esta forma, un módulo ***restador*** tendrá la siguiente estructura:

```
entero dato, restador
...
restador ← 0                    /* Inicializacion */
...
restador ← restador – dato
```

Un módulo ***divisor*** podrá ser:

```
entero dato, divisor
...
divisor ← 1                     /* Inicializacion */
...
divisor ← divisor / dato
```

Y un módulo ***decrementador*** (o ***descontador***) será:

```
entero descontador
...
descontador ← 0                 /* Inicializacion */
...
descontador ← descontador – 1
```

Es importante observar que si bien se han mantenido las *inicializaciones* de los módulos originales, en estos últimos tres módulos *los valores de inicialización generalmente difieren de los elementos neutros de cada operación a realizar*. Esto se debe a que en la mayoría de los casos, un ***restador*** o un ***decrementador*** inician su vida con algún valor distinto de cero y una vez en operación interesa detectar cuando su valor cruza el umbral del cero. Por ejemplo:

```
entero descontador
...
descontador ← 20        /* <-- Inicializacion en 20!! */
...
4  descontador ← descontador – 1
```

En este caso, cada vez que se ejecuta la acción **4** el valor inicial de la variable ***descontador*** *se reduce en una unidad*. La detección de llegada o cruce del umbral del valor *cero* requerirá el uso de alguna *estructura de control de flujo* que permita evaluar esta condición.

2.7.5 Módulo buscador de máximo o mínimo

En numerosos problemas resulta necesario encontrar el valor *máximo* o *mínimo* que toma una variable a lo largo de su evolución en el algoritmo y a tal fin se han desarrollado los módulos necesarios para lograr esta tarea. El diseño de estos

módulos está fuertemente basado en lo analizado en el **Ejemplo 2** y podrá distinguirse fácilmente la extrema similitud entre ellos.

El principio de operación es sencillo y consiste en *inicializar con un valor muy pequeño a la variable que contendrá el mayor valor* encontrado a lo largo de la búsqueda *(aunque también puede inicializarse con el primer valor del conjunto si este es conocido)*. Luego, a medida que evoluciona el algoritmo, se comparará este mayor valor obtenido con los nuevos valores que tome la variable analizada, y de encontrarse un nuevo mayor, este ocupará el lugar del anterior para continuar la búsqueda. El fragmento algorítmico tendrá una estructura similar a:

```
entero dato, maximo          /* dato contiene los valores a procesar y
                                maximo contendrá al mayor valor al final
                                del proceso */
...
maximo ← -1000               /* Inicializacion: se supone que -1000 es
                                muy pequeño respecto a los valores de
                                los datos*/
...
Si (dato > maximo) entonces  /* Posiblemente en una estructura de
                                repetición.*/
        maximo ← dato        /* Se encontró un nuevo maximo. Se
                                lo resguarda para las futuras
                                comparaciones */
fin-si
```

5

Cada vez que se ejecute la sección marcada como **5**, se contrastará el valor actual de **dato** y el valor máximo almacenado hasta el momento. Si ocurre que este valor máximo es inferior al nuevo valor de la variable **dato**, entonces el valor de **dato** será almacenado como el nuevo máximo para futuras comparaciones.

Si fuese necesario encontrar el mínimo valor en el conjunto de datos solo bastará con inicializar el **minimo** con un valor muy grande e invertir la expresión relacional en la estructura ***si***, quedando de esta forma:

```
entero dato, minimo          /*dato contiene los valores a procesar.
                                minimo contendrá al menor valor al final
                                del proceso */
...
minimo ← 1000                /* Inicializacion: se supone que 1000 es
                                muy grande respecto a los valores de los
                                datos */
...
Si (dato < minimo) entonces
   minimo ← dato             /* Se encontró un nuevo minimo. */
fin-si
```

3 Estructuras de datos

3.1 Introducción

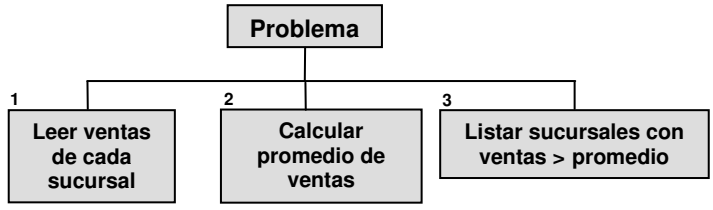
Hasta ahora se ha trabajado con variables que representan tipos de datos simples y de tamaño fijo, tales como entero, real, caracter, etc. Técnicamente, estas variables se denominan ***variables escalares*** y se caracterizan por contener datos **atómicos** y **unidimensionales**, donde el concepto de ***atomicidad*** significa que independientemente del tamaño en bytes que posea la variable, todos ellos se consideran en forma simultánea, o bien, que las variables escalares no pueden dividirse en "partes". El concepto de ***unidimensionalidad*** hace referencia a que las variables escalares contienen un *único valor*.

Sin embargo, el manejo de datos reales no siempre se puede realizar en forma cómoda y eficiente mediante el uso de variables escalares, y en muchos casos resulta casi imposible utilizar este tipo de variables para almacenar y procesar ciertos conjuntos de datos.

El principal problema se plantea cuando el algoritmo debe trabajar sobre una gran cantidad de datos que guardan entre si alguna relación. En estos casos, para cada uno de estos datos se debería utilizar una variable escalar diferente, lo que ocasiona complicaciones en el algoritmo a desarrollar y también en la cantidad de variables a utilizar.

Para resolver estas dificultades se pueden agrupar los datos en un ***conjunto de variables escalares***, bajo un nombre común, lo que permite tratarlos como una unidad. Estos conjuntos **multidimensionales** (por que contienen mas de un valor) reciben el nombre de ***Estructuras de Datos***, y para comprender la verdadera utilidad que se obtiene utilizándolas, se analizarán las posibles soluciones del siguiente problema:

Problema 6: Una empresa recibe mensualmente información sobre las ventas de cada una de sus tres sucursales y desea obtener un listado de aquellas sucursales cuyas ventas superan el valor promedio de el conjunto.



Aplicando lo expuesto en el diseño ***Top-Down***, el algoritmo completo resulta:

```
real  venta1, venta2, venta3, promedio
Inicio
```

1

```
    lea  venta1
    lea  venta2
    lea  venta3
```

```
2 → promedio ← (venta1 + venta2 + venta3) / 3
    si  venta1 >= promedio entonces
          escriba  "Sucursal 1:", venta1
    fin si
    si  venta2 >= promedio entonces
3         escriba  "Sucursal 2:", venta2
    fin si
    si  venta3 >= promedio entonces
          escriba  "Sucursal 3:", venta3
    fin si
  Fin
```

Lamentablemente, esta solución *solo es válida para tres sucursales*. Pero si ahora fuese necesario procesar 100 sucursales ¿como se plantearía la solución para estas 100 sucursales? Empleando el mismo enfoque, sería necesario definir 100 variables *venta1*, *venta2*,... hasta *venta100*, con lo que el nuevo algoritmo para 100 sucursales sería algo similar a:

```
real venta1, venta2,... , venta100, promedio
Inicio
    lea  venta1, venta2,... , venta100
    promedio ← (venta1 + venta2 + ... + venta100)/100
    si venta1 >= promedio entonces
    escriba "1:", venta1
    fin si
    si venta2 >= promedio entonces
    escriba  "2:", venta2
    fin si
    ...
    si venta100 >= promedio entonces
    escriba "100:", venta100
    fin si
Fin
```

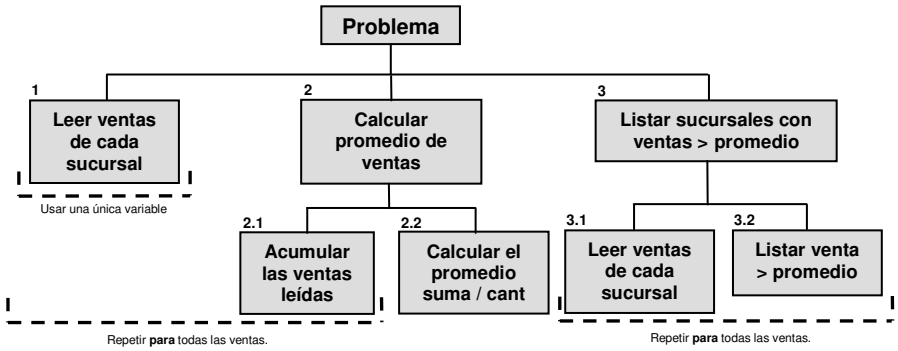
En este ejemplo se ha utilizado el símbolo **"..."** que nos permite "resumir" la escritura de un gran número de instrucciones similares. La primera vez lo hemos utilizado para denotar la declaración, lectura y cálculo del promedio de las 100 variables *ventaXXX*, y la segunda para indicar que la comprobación de las muestras continúa desde la sucursal 3 hasta la 99. El inconveniente es que **este símbolo no es comprendido por ningún procesador** y la única solución viable es escribir explícitamente las 100 variables y las 100 estructuras de decisión, lo cual es engorroso, poco versátil y muy propenso a errores.

Una solución alternativa, que minimiza el uso de variables y vuelve al sistema apto para procesar cualquier cantidad de sucursales es realizar la lectura sobre una única variable pero, como contrapartida, esta propuesta **requiere leer dos veces** la información de cada sucursal:

- Con la primer lectura de datos se determinará el promedio de las ventas.

- Con la segunda se determinará cual de las sucursales tienen ventas superiores al promedio.

El nuevo diseño **Top-Down** sería similar a:




```
entero i
real  venta, promedio, suma
Inicio
     suma ← 0
     para i ← 1 hasta 3 haga
➔         lea venta
          suma ← suma + venta
     fin para
     promedio ← suma / 3
     para i ← 1 hasta 3 haga
➔         lea  venta
          si venta >= promedio  entonces
               escriba  i, venta
          fin si
     fin para
fin
```

Este algoritmo es extremadamente ineficiente por el hecho de leer dos veces el mismo conjunto de datos, sobrecargando la tarea del usuario, y desde este punto de vista, la primer solución es la más cercana a lo óptimo.

Cuando se resuelve un problema en particular, se debe tratar de que la organización de sus datos se realice en forma estructurada; ***y por lo tanto, seleccionar adecuadamente los tipos de datos es un paso necesario y fundamental al definir y resolver el problema***. Todas las formas posibles en que los datos primitivos se relacionan *lógicamente*, definen distintos tipos de *estructuras de datos*.

En este problema, una forma más adecuada de resolverlo considerando la primer solución, consiste en reunir las variables que contienen las ventas de las 100 sucursales *bajo un único nombre*, por ejemplo ***venta***. Este conjunto de valores reales podría estar representado, esquemáticamente, en la forma de una *tabla*, tal como muestra la siguiente figura:

	1	2	3	4	...	100
venta	120	578	115	654	...	121

Para ***individualizar*** a cada una de las 100 variables se utiliza **su ubicación** en la *tabla* colocándola entre corchetes `[]` a la derecha del nombre de la tabla. Este valor entre corchetes es la posición que ocupa la variable en el conjunto y se denomina ***índice***. De esta manera `venta[3]` representa la variable ubicada en la tercera posición (de valor 115), que contiene la venta de la tercera sucursal. Esta "tabla" es lo que se denomina ***arreglo lineal*** de nombre **venta** cuyos 100 elementos son `venta[1]`, `venta[2]`,..., `venta[100]`.

3.2 Definición y Características de los Arreglos

Definición: ***Un arreglo (o array) es un conjunto ordenado de variables del mismo tipo, identificadas por un mismo nombre y que se distinguen entre sí por su posición dentro del conjunto.***

Según el tipo de datos almacenados, los arreglos se pueden clasificar en:

- Arreglos Numéricos (enteros o reales)
- Arreglos Alfanuméricos (caracteres o *string*)

y según la cantidad de ***"dimensiones"*** de un arreglo, este se pueden clasificar en:

- Unidimensional.
- Bidimensional.
- Multidimensional.

Cada *variable componente* de un arreglo se denomina ***elemento del arreglo*** y se denota y accede escribiendo el nombre del arreglo seguido de *una* ***expresión entera positiva*** *encerrada entre corchetes*, llamada ***índice del elemento del arreglo***.

El número o cantidad total de elementos se indica cuando se declara el arreglo y desde entonces queda invariable. A este número se lo denomina ***tamaño*** del arreglo y, por ejemplo, el *tamaño* del arreglo ***venta*** es de 100 elementos.

Para entenderlo mejor es posible hacer la siguiente analogía: suponer que la zona de memoria donde está almacenado el arreglo es un conjunto de casilleros, uno contiguo al siguiente, y para diferenciar un casillero del otro ***se debe especificar su ubicación o índice***, ya que el nombre es común a todos los casilleros. De esta manera se puede modificar un dato de una determinada posición, se puede borrar, se puede operar con el, etc. ***Todas las operaciones y accesos que se pueden realizar sobre variables escalares también están permitidos sobre cada elemento de un arreglo.***

3.2.1 Arreglos Unidimensionales

Un arreglo unidimensional es un conjunto formado por un número determinado de variables escalares, que comparten un nombre y está ordenado de forma tal que se

puede acceder a ellas mediante su posición, la que está simbolizada por un ***unico*** **índice**. En el **Problema 6** los elementos del arreglo (unidimensional) son:

`venta[1], venta[2],....,venta[i], ......,venta[n]`

Donde **n** es la cantidad de elementos que forman el arreglo y además determina la ***longitud*** del mismo. El nombre del arreglo es **`venta`**, y por lo tanto **`venta[1]`** es el primer elemento del arreglo y **`venta[n]`** es el último.

Los elementos del arreglo están ordenados:

- **`venta[ i-1 ]`** antecede a **`venta[ i ]`** para *i = 2, 3,....n*
- **`venta[ i+1 ]`** sucede a **`venta[ i ]`** para *i = 1,....n-1*

Los siguientes son otros posibles ejemplos de *arreglos unidimensionales*:

- El conjunto de las notas de un parcial correspondientes a los alumnos de un curso.
- Los números de día del mes.
- Las edades de los concurrentes a un centro de salud, etc.

y las operaciones más habituales que pueden realizarse sobre arreglos son:

- Acceder a un elemento para examinar o modificar su contenido.
- Operar con un elemento del arreglo
- Ordenar los elementos de un arreglo según un determinado criterio.
- Explorar el arreglo en busca de los elementos que cumplan con determinada condición.

En el entorno **LPP** los arreglos unidimensionales se declaran según la siguiente sintaxis:

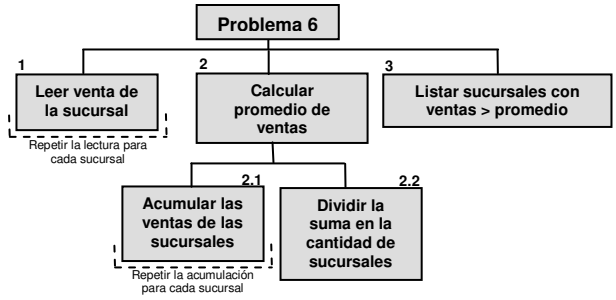
`arreglo[cantidad_de_elementos] de tipo_dato  nombre_arreglo`

Por ejemplo:

`arreglo[100] de entero venta`

3.2.2 Uso de Arreglos Unidimensionales

Ahora se resolverán problemas utilizando arreglos en lugar de variables escalares, comenzando por el **Problema 6** ya analizado:




```
arreglo[100] de real venta
real promedio
entero i
Inicio
```

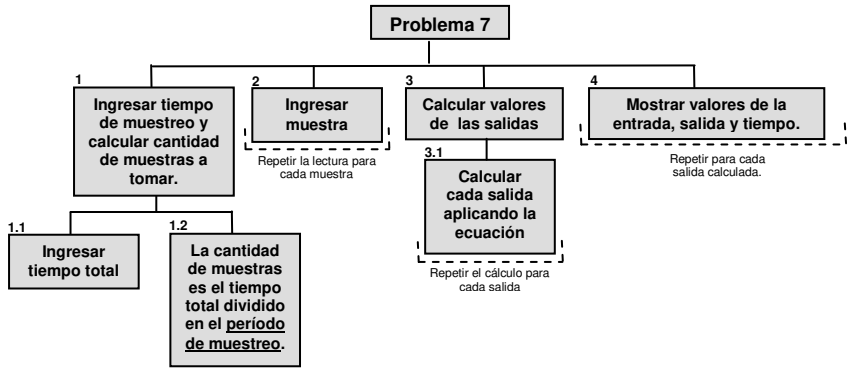
```
        promedio ← 0
        /* 1-Se leen los montos de las 100 sucursales */
1       para i ← 1 hasta 100 haga
              lea  venta[ i ]
        fin para
        /* 2.1-Se calcula el promedio de las 100 sucursales */
2.1     para i ← 1 hasta 100 haga
              promedio  ← promedio + venta[ i ]
        fin para
        /* 2.2 */
2.2     promedio ← promedio / 100
        /* 3-Se muestran las sucursales con venta>promedio */
        para i ← 1 hasta 100 haga
3             si venta[ i ] >= promedio entonces
                    escriba "Suc", i, "=", venta[ i ]
              fin si
        fin para
Fin
```

Problema 7: En un experimento se deben almacenar los valores de tensión eléctrica sobre una determinada resistencia medidos cada 100 milisegundos, realizando esta tarea durante el lapso de tiempo que indique un operador. Los datos de entrada almacenados se deben procesar aplicando la siguiente expresión para generar un nuevo conjunto de datos (salida):

$salida[t_i] = 2.5 * entrada[t_i] + 0.25 * entrada[t_i - 1]$ para $t_i > 0.2s$
$salida[1] = salida[2] = 0.0$

$entrada[t_i]$ y **$salida[t_i]$**, son los valores de tensión de entrada y salida correspondientes al instante de tiempo t_i. Se deben mostrar los valores de las tensiones de entrada y salida correspondientes a todos los instantes de tiempo t_i, en que se realizaron las mediciones.

Solución: Si se emplean un par de arreglos como estructuras de datos, **`salida[k]`** y **`entrada[k]`**, donde **`salida`** y **`entrada`** son los nombres de los arreglos de datos y **`k`** un índice que indica a qué elemento del arreglo se está haciendo referencia, se podría disponer la adquisición de datos dentro de una estructura de iteración la que finalizaría cuando se supere el tiempo determinado por el operador (mientras **`k<(tiempo/0.1)`**), y del mismo modo se podría implementar el cálculo de la tensión de salida en otra iteración:



Y el algoritmo resultará:

```
arreglo[500] de real entrada
arreglo[500] de real salida
real tiempo, reloj
entero i, f
Inicio
1.1   escriba "Ingrese tiempo de toma de muestras en seg."
      lea  tiempo
1.2   f ← (tiempo / 0.1)
      para i ← 1 hasta f haga
2           lea entrada[i]
      fin para
      salida[1] ← 0
      salida[2] ← 0
3.1   para i ← 3 hasta f haga
            salida[i] ← entrada[i]*2.5+entrada[i-1]*0.25
      fin para
      escriba "SALIDA  ENTRADA  TIEMPO"
4     para i ← 1 hasta f haga
            escriba salida[i], " ", entrada[i], " ", i * 0.1
      fin para
Fin
```

Ahora, las implementaciones son mucho más sencillas, compactas y ordenadas cuando se utilizan arreglos de datos que cuando se emplean variables escalares.

3.2.3 Arreglos Multidimensionales

Así como se han descripto los arreglos unidimensionales, también pueden implementarse **arreglos multidimensionales** del siguiente modo:

`a[20][20]`	donde **a** es una matriz cuadrada de 20 x 20.
`a[10][20][10]`	donde **a** es un arreglo de tres dimensiones, de 10 x 20 x 10 elementos.

A título meramente ilustrativo, si en el ejemplo previo se desea almacenar también la corriente que circula por la resistencia, se podría definir un arreglo bidimensional con la misma cantidad de elementos que el arreglo unidimensional (como si se tuvieran dos arreglos unidimensionales uno junto al otro, del mismo tipo que el empleado anteriormente para almacenar la tensión). A la primera dimensión se le podría asignar la tarea de almacenar la tensión y a la segunda, la de almacenar la corriente.

4 Subprogramas

El análisis por medio de la metodología ***Top-Down*** implica la división de un problema en sub-problemas mas pequeños (módulos), que pueden ser entendidos y resueltos de manera simple. Sin embargo, hasta ahora se ha escrito el algoritmo que resuelve cada uno de estos módulos en forma independiente y *se lo ha integrado como un bloque de código* más a la solución final del problema, tal como sucede con cada tarea en el ejemplo del punto 1.3.4, que se repite a continuación:

```
        Inicio
        Repita
t1          tomar un caramelo de la bolsa
            poner un caramelo en la caja del gusto correspondiente
        hasta que la bolsa esté vacía

        Repita
t2.1        tomar 4 caramelos de la caja de frutillas y
            distribuirlos a los 4 niños.
        hasta que la caja de frutilla contenga menos de 4 caramelos
        Repita
t2.2        tomar 4 caramelos de la caja de menta y distribuirlos
            a los 4 niños.
        hasta que la caja de menta contenga menos de 4 caramelos
        Repita
t2.3        tomar 4 caramelos de la caja de naranja y
            distribuirlos a los 4 niños.
        hasta que la caja de naranja contenga menos de 4 caramelos
```

Esta forma de desarrollo de los *bloques de código* trae aparejada una revisión de los bloques ya resueltos anteriormente, y también fija restricciones a los próximos a resolver, ya que la mayoría de las veces los diferentes bloques comparten, total o parcialmente, los recursos sobre los que operan (los ***4 niños*** del ejemplo son compartidos por todas las tareas **t2.x**).

Esto puede violar el precepto de la ***independencia entre módulos*** y además, puede forzar la necesidad de acuerdos adicionales entre los integrantes de los distintos grupos de desarrollo. La solución a este inconveniente consiste en aprovechar la división del algoritmo en tantas partes como sub-problemas hayan resultado del análisis, y encerrar cada uno de estos mini-algoritmos en una suerte de *"cápsula"* que lo aísla del resto de los módulos a la vez que le provee los recursos necesarios para llevar a cabo su trabajo y define las vías de comunicación que la cápsula tendrá con el exterior. De esta manera, el procedimiento de análisis y diseño de la solución es idéntico a lo ya estudiado, pero se agrega una "envoltura" adicional que permite ocultar los detalles de cada mini-algoritmo creando entonces ***una nueva acción primitiva*** con un *nombre*, *comportamiento* y *comunicación* perfectamente definidos.

Cada uno de estos *algoritmos encapsulados* constituye lo que antiguamente se denominaba *subprograma* y que hoy se conoce como ***procedimiento***, ***función*** o ***método***. La *"forma"* de cada uno de estos subprogramas es análoga a la de un

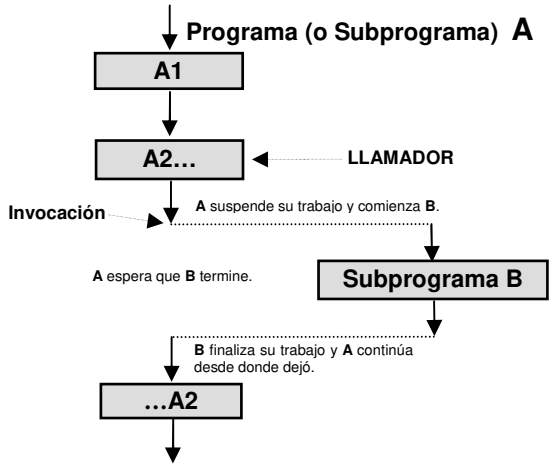
programa convencional en el sentido de que consta de su propio ambiente y sus propio conjunto de acciones primitivas que definen su comportamiento. Los subprogramas poseen un ***nombre*** que los identifica en el contexto de aplicación y también declaran un conjunto de variables, llamadas ***parámetros*** o ***argumentos***, que operan como canales de comunicación con el ambiente exterior.

> Así, la idea detrás de los subprogramas es la de construir *bloques de código* ***opacos*** cuando se los observa desde el exterior, pero que pueden ser caracterizados por el comportamiento sobre la información que ***entra*** y ***sale*** de los mismos, y por ello favorecen la *modularización* y *reutilización* del código.

Una característica importante es que ***los subprogramas no realizan su trabajo a menos que alguien o algo se los demande*** (el programa principal, otro subprograma o la señal eléctrica de algún dispositivo de hardware). El proceso de solicitar que un subprograma realice su trabajo se denomina ***invocación del subprograma*** (o ***llamar al subprograma*** en la jerga informática), y la secuencia de operación es:

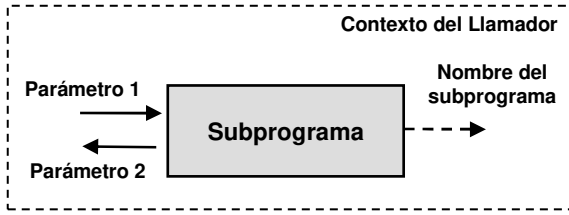
1. Un programa o subprograma ***A*** requiere los servicios de un subprograma ***B***, y para ello ***A invoca a B***.
2. El programa ***A*** *suspende su trabajo en el momento de la invocación* y comienza la operación del subprograma ***B***.
3. Cuando el subprograma ***B*** finaliza su trabajo, el programa ***A*** *continúa su operación desde el punto donde la suspendió*, posiblemente utilizando los resultados del subprograma **B**.

El siguiente diagrama muestra el proceso de ***invocación***, donde el programa, subprograma o sección que realiza la invocación está señalado como **LLAMADOR**.



Un subprograma, a menudo, necesita comunicarse con el módulo que lo invoca a fin de recibir información para realizar su trabajo o para devolverle los resultados del procesamiento realizado. Esta comunicación se lleva a cabo por medio de los ***parámetros*** o ***argumentos***, que no son mas que variables que el subprograma

declara a tal efecto y que forman parte de su estructura. En otras ocasiones, *el nombre del subprograma opera como una variable* cuyo valor, asignado por el subprograma, puede ser utilizado por el *módulo invocador*.



En el entorno **LPP** existen dos tipos de subprogramas diferentes: ***los procedimientos*** y ***las funciones*** cuyas diferencias se analizarán a continuación.

4.1 Procedimientos

Un ***procedimiento*** es un subprograma con un comportamiento específico asociado a uno o más módulos o segmentos del análisis, y que tiene la particularidad de utilizar únicamente los *parámetros* como medio de intercambio de datos con el módulo *llamador*. De esta manera, algunos parámetros serán ***entradas*** de datos, otros serán ***salidas*** y otros podrán ser *entradas* y *salidas* a la vez, todo ello de acuerdo a los requerimientos de la funcionalidad encerrada en la "cápsula" del *procedimiento*. Los procedimientos se declaran de la siguiente forma:

Procedimiento nombre_del_procedimiento [(lista_de_parámetros)]
[variables_locales_del_procedimiento ***]***
inicio
    acciones_propias_del_procedimiento
fin

La palabra clave ***Procedimiento*** indica que la secuencia de acciones a continuación es el "cuerpo" de un subprograma

Tal como ya se dijo, cada *procedimiento* requiere un ***nombre*** que lo identifica de forma única en el contexto del problema y que permitirá su invocación de manera simple. *Se recomienda muy seriamente que este nombre sea representativo de la tarea desarrollada por el subprograma* y que se eviten las abreviaturas o siglas extrañas que puedan llevar a confusión.

La ***lista_de_parámetros*** no es mas que una secuencia de declaraciones de variables encerrada entre paréntesis y separadas por comas, de la forma:

tipo nombre_param_1, tipo nombre_param_2, . . . , tipo nombre_param_k

La cantidad de parámetros a declarar depende de las necesidades del subprograma y el **módulo invocador** completará los valores de los *parámetros* en el momento de la invocación del mismo, *respetando estrictamente la secuencia y tipos declarados*. Los corchetes que encierran a la lista de parámetros en la declaración del procedimiento manifiestan que la presencia de la lista es ***opcional***. Esto es así por cuanto puede darse el caso de subprogramas que no requieran

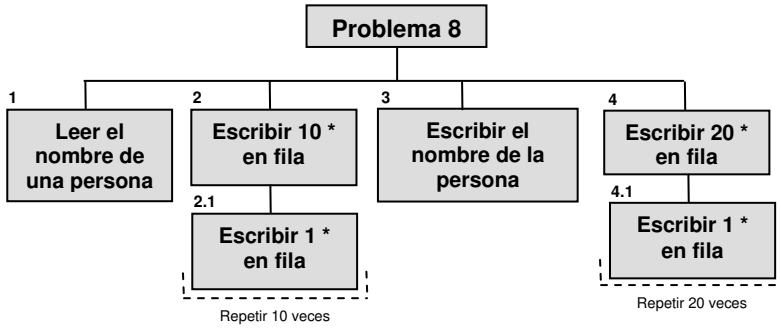
comunicación alguna con su llamador, y en este caso puede obviarse la lista por completo.

Las ***variables_locales_del_procedimiento*** son un conjunto de declaraciones de variables que brindan servicio al subprograma. *Esta variables solo son accesibles en el contexto propio del subprograma y no existen fuera de él*, por lo que también se las conoce como ***variables locales***. También aparecen encerradas entre corchetes, lo que una vez más significa que su uso es ***opcional***, y si el *procedimiento* no requiere de *variables locales*, estas pueden obviarse por completo.

Entre las etiquetas ***inicio*** y ***fin*** se escribe la secuencia de acciones que definen el comportamiento del subprograma, de forma análoga a lo que sucede en un programa convencional.

Por último, para invocar un procedimiento se utiliza la acción primitiva ***Llamar*** seguida del nombre del mismo y de la lista de parámetros encerrada entre paréntesis.

Problema 8: Diseñar un algoritmo que permita la lectura del nombre de una persona y lo muestre encerrado entre dos líneas de asteriscos: la superior con 10 asteriscos y la inferior de 20.



En este diseño Top-Down se aprecia que los nodos 2 y 4 realizan la misma tarea modificando solo la cantidad de asteriscos impresa, y de hecho, los nodos descendientes 2.1 y 4.1 son exactamente iguales, cambiando entre ellos la cantidad de repeticiones necesarias.

Si se procede según la metodología utilizada hasta ahora, la solución final sería similar a:

```
cadena[25]nombre
entero i
Inicio
      escriba "Ingresar el nombre..:"
      lea nombre
      para i←1 hasta 10 haga
            escriba "*"
      fin para
      escriba nombre
      para i←1 hasta 20 haga
```

```
            escriba "*"
      fin para
Fin
```

Aunque el algoritmo que soluciona este problema es muy simple, es fácil encontrar que los fragmentos remarcados constituyen una duplicación de código, que aparte de "*ocultar*" el propósito de esas iteraciones, podría complicar el mantenimiento a futuro. Esto es un caso típico donde la utilización de un subprograma puede mejorar la solución lograda, aún a expensas de escritura adicional.

Para lograr esta mejora se encapsularán las iteraciones en un procedimiento que podrá ser invocado cada vez que se requiera escribir una fila de asteriscos cuya longitud se determinará en el momento de la invocación, disponiendo para ello de un argumento entero que indicará al procedimiento la cantidad de asteriscos a escribir en la fila:

```
procedimiento escribe_asteriscos(entero cuantos)
entero i
inicio
      para i←1 hasta cuantos haga
            escriba "*"
      fin para
fin
```

El nombre del procedimiento (***escribe_asteriscos***) es totalmente descriptivo de la tarea que lleva a cabo, como también lo es el nombre del parámetro ***cuantos***, que hace referencia a la cantidad de asteriscos a escribir.

Si ahora se reemplazan las iteraciones iniciales por las invocaciones al procedimiento, el código resultará sin duplicaciones y será mas fácilmente legible:

```
cadena[25]nombre
Inicio
      escriba "Ingresar el nombre..:"
      lea nombre
      llamar escribe_asteriscos( 10 )
      escriba nombre
      llamar escribe_asteriscos( 20 )
Fin
```

El lenguaje LPP requiere que las declaraciones de los subprogramas estén contenidas en el mismo archivo que el algoritmo solución, y además deben precederlo, por lo que la solución final quedará:

```
cadena[25]nombre
procedimiento escribe_asteriscos(entero cuantos)
entero i
inicio
      para i←1 hasta cuantos haga
            escriba "*"
      fin para
fin
Inicio
      escriba "Ingresar el nombre..:"
```

```
        lea nombre
        llamar escribe_asteriscos( 10 )
        escriba nombre
        llamar escribe_asteriscos( 20 )
Fin
```

4.2 Funciones

Una ***función*** es un subprograma con un comportamiento específico asociado a uno o mas módulos o segmentos del análisis, y que aparte de utilizar los *parámetros* como medio de intercambio de datos con el módulo *llamador* también provee un camino directo para retornarle un único valor. De esta manera, y al igual que en los **procedimientos**, algunos parámetros serán ***entradas*** de datos, otros serán ***salidas*** y otros podrán ser *entradas* y *salidas* a la vez, todo ello de acuerdo a los requerimientos de la funcionalidad encerrada en la "cápsula" de la *función*. En este caso el **nombre_de_la_funcion** se comporta como "*una variable*" que contiene el valor que la función *retorna* al llamador, y como tal, este nombre puede participar en alguna expresión compatible con su tipo de dato.

Las funciones se declaran de forma casi análoga a los procedimientos:

Funcion nombre_de_la_funcion [(lista_de_parámetros)] : tipo_de_retorno
[variables_locales_de_la_función **]**
inicio
    acciones_propias_de_la_función
    retorne variable_o_valor
fin

La palabra clave ***Funcion*** indica que la secuencia de acciones a continuación es el "cuerpo" de un subprograma, aunque ahora se incluyen ***dos puntos*** al final de la lista de parámetros seguidos por un tipo de dato (***tipo_de_retorno***). Este tipo representa el tipo de dato del valor retornado por la función al programa o subprograma llamador.

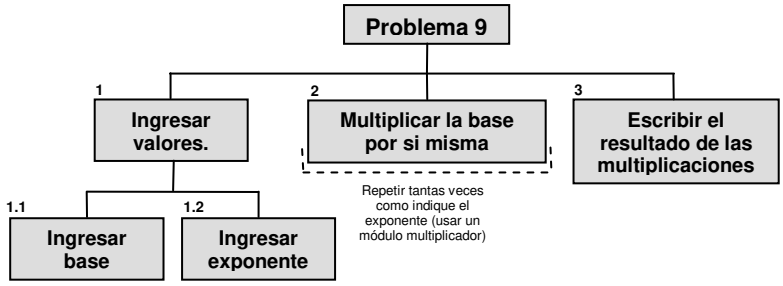
Al igual que los *procedimientos*, cada *función* también requiere un ***nombre*** que la identifica de forma única en el contexto del problema y que permitirá su invocación de manera simple. *Se insiste en recomendar muy seriamente que este nombre sea representativo de la tarea desarrollada por la función* y que se eviten las abreviaturas o siglas que puedan llevar a confusión.

La ***lista_de_parámetros*** y las ***variables_locales_de_la_función*** son totalmente análogas a los parámetros y variables locales de los procedimientos, y poseen exactamente las mismas propiedades.

Entre las etiquetas ***inicio*** y ***fin*** se escribe la secuencia de acciones que definen el comportamiento de la función, de forma totalmente análoga a lo que sucede en un programa convencional, pero en este caso la finalización de la ejecución del cuerpo de la función se realiza por medio de una acción específica llamada ***retorne***, la que debe ir seguida de una constante literal o una variable o una expresión de tipo acorde al declarado en el ***tipo_de_retorno***. Este valor asociado a la acción ***retorne*** es el que será devuelto al módulo llamador *por medio del*

nombre de la función. Finalmente, la invocación de una función también se realiza por medio de su nombre seguido de la lista de parámetros encerrada entre paréntesis.

Problema 9: Realizar un algoritmo que permita el ingreso de dos valores enteros (base y exponente) y permita calcular **base**$^{\textbf{exponente}}$ utilizando la técnica de productos sucesivos.




```
entero x, y, potencia
funcion calculaPotencia(entero base, entero expo):entero
entero i, resultado
inicio
      resultado ← 1    /*resultado es un multiplicador*/
      para i←1 hasta expo haga
            resultado ← resultado * base
      fin para
      retorne resultado
fin
Inicio
      escriba "Ingresar la base:"
      lea x
      escriba "Ingresar el exponente:"
      lea y
      potencia ← calculaPotencia( x, y )
      escriba x, "elevado a la", y, "es=", potencia
Fin
```

4.3 Pasaje de parámetros

Tal como se comentó en la descripción básica de los ***procedimientos*** y ***funciones***, los principales caminos de comunicación con el llamador que estos poseen están limitados a los parámetros presentes en su declaración, donde algunos podrán ser entradas de datos, otros podrán ser salidas de datos y algunos otros más podrán ser entradas o salidas según convenga.

Por defecto, y a menos que se especifique lo contrario, ***todos los parámetros formales de un subprograma se comportan solamente como entradas de datos***, es decir, el subprograma solo puede recibir valores por su intermedio pero no puede devolver resultados al llamador. Esto es lo que se denomina ***paso de parámetros por valor***, y equivale a **copiar el valor** enviado por el llamador (*parámetro actual*) en la variable representada por el parámetro en cuestión (*parámetro formal*). De esta forma, cualquier modificación que sufra el *parámetro*

formal en realidad la estará sufriendo *la copia del valor del parámetro actual*, y por ende, dichas modificaciones internas al subprograma serán completamente invisibles al llamador.

Si se desea que el intercambio de datos con el llamador sea de salida o bidireccional, el parámetro formal debe declararse precedido del modificador ***var***. Ahora, el llamador puede esperar que el subprograma modifique de alguna manera el valor del *parámetro formal*, de manera tal que al finalizar la ejecución del subprograma esta modificación aparezca replicada dentro del *parámetro actual*. Este proceso se denomina ***paso de parámetros por referencia***, y equivale a **copiar la dirección de memoria** donde reside la variable enviada por el llamador en la variable representada por el parámetro en cuestión. El siguiente ejemplo muestra un uso típico del paso de parámetros por referencia: *la lectura de valores dentro de un subprograma y su propagación al módulo llamador*.

```
entero xdato, xlimite
procedimiento leaYValide(VAR entero dato, entero limite)
inicio
      repetir
            escriba "ingrese un valor menor que", limite
            lea dato    /* modifica el parámetro
                        formal!!! */
      hasta dato < limite
fin
Inicio
      escriba "ingrese un valor limite"
      lea xlimite
      /* dato (parametro actual) no ha
          sido inicializado */
      llamar leaYValide(xdato, xlimite)
      /* luego de la invocacion del procedimiento, dato
          contiene un valor inferior al limite */
      escriba "El valor ingresado es: ", xdato
Fin
```

4.4 Los subprogramas y el diseño Top/Down

De lo expuesto se puede apreciar claramente la importancia del uso de subprogramas cuando lo que se pretende es reutilizar código (generando módulos portables entre distintos programas) y permitir el ensamble de módulos creados por desarrolladores diferentes. Pero el principal problema que surge a la hora de diseñar e implementar subprogramas queda simbolizado por la pregunta:

¿cual módulo se debe implementar como subprograma?

y el diseño **Top-Down** nuevamente provee la respuesta correcta:

Todos los módulos independientes que surgieron del análisis Top-Down son candidatos a convertirse en subprogramas.

Quien analice el problema deberá decidir cuales módulos serán construidos como subprogramas y cuales quedarán como código de unión e invocación entre ellos. Pero independientemente de esta decisión, y aún cuando no sea estrictamente necesario, **siempre es una buena práctica pensar los módulos como subprogramas**, ya que de esta forma se fortalece la relación entre el esquema descendente y la estructura del algoritmo creado a partir de él, a la vez que se generan módulos reusables que quedan disponibles para su uso en condiciones similares.

Para ejemplificar la metodología de diseño y uso de subprogramas, se trabajará sobre el ya analizado **Ejemplo 6** disponible en el apartado 3.2.2

Del diseño Top/Down puede verse que existen tres bloques de alto nivel que describen las tareas:

1. Leer las ventas de las sucursales.
2. Calcular del promedio de ventas.
3. Mostrar las sucursales con mayor venta que el promedio.

Los nodos **2.1** y **2.2** prácticamente son "acciones primitivas", por lo que la sugerencia es construir solo tres subprogramas, uno para cada módulo de alto nivel.

El primero de ellos procesa la lectura de las ventas de las sucursales, y para realizar su trabajo requiere disponer de acceso al conjunto de ventas y a la cantidad de sucursales, por lo que necesita dos canales de ingresos de datos y uno de egreso. Sin embargo, el paso de un arreglo a un subprograma **siempre** se realiza por referencia, por lo que la modificación que sufra el valor de cada elemento del arreglo será visible desde el llamador. Esto evita la necesidad del canal de egreso y el módulo puede construirse como un **procedimiento**:

```
Procedimiento leeVentas(arreglo[100] de real venta, entero cant)
Entero i
Inicio
      para i←1 hasta cant haga
          Escriba "ingrese la venta de la sucursal [",i,"]="
          Lea venta[i]
      fin para
Fin
```

El segundo subprograma es quien calcula el promedio de las ventas leídas, y para ello requiere acceso al conjunto de ventas y a la cantidad de sucursales. Al finalizar su trabajo, este subprograma deberá entregar como resultado el valor del promedio calculado, lo que sugiere que debe tratarse de una **función**.

```
Funcion calculaPromedio(arreglo[100] de real venta,
                        entero cant): real
Entero i
Real suma
Inicio
      suma←0
      para i←1 hasta cant haga
```

```
            suma ← suma + venta[i]
      fin para
      retorne (suma / cant)
Fin
```

Por último, el tercer subprograma es quien muestra las sucursales cuyas ventas fueron superiores al promedio, y para ello requiere acceso al conjunto de ventas, a la cantidad de sucursales y al promedio calculado. Dado que la tarea de este subprograma es mostrar por pantalla un conjunto de valores, la sugerencia es que sea un **procedimiento**.

```
Procedimiento mostrarMayoresVentas(arreglo[100] de real venta,
                                   entero cant, real prom)
Entero i
Inicio
      para i←1 hasta cant haga
            Si (venta[i] < prom) entonces
                  Escriba "Sucursal", i, "vendio menos de ", prom
            fin si
      fin para
Fin
```

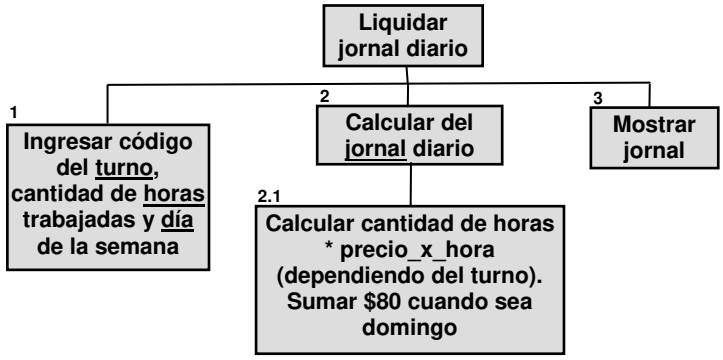
Y el programa final resulta:

```
Arreglo[100] de Real ventas
Real promedio
Entero cantSuc
/* En esta zona se declaran los subprogramas anteriores */
Inicio
      Escriba "Ingrese la cantidad de sucursales:"
      Lea cantSuc
      llamar leeVentas( ventas, cantSuc )
      promedio ← calculaPromedio( ventas, cantSuc )
      llamar mostrarMayoresVentas( ventas, cantSuc, promedio )
Fin
```

5 Ejercitación

5.1 Ejercicios Resueltos

1. *Los operarios de una empresa trabajan en dos turnos: uno diurno, cuyo código es menor que 10 y otro nocturno, de código mayor o igual a 10. Se desea calcular el jornal para un operario sabiendo que para el turno nocturno el pago es de 150 pesos la hora y para el turno diurno es de 120 pesos la hora, pero en este último caso, si el día es domingo, se paga un adicional de 80 pesos la hora.*



Las fórmulas con las cuales se debe calcular el jornal son:

1-Turno nocturno: Jornal ← 150 * horas_trabajadas (1)

2-Turno diurno:

a) si el día no es domingo.

Jornal ← 120 * horas_trabajadas (2a)

b) si el día es domingo.

Jornal ← 120 * horas_trabajadas + 80 * horas_trabajadas (2b)

Las acciones de este algoritmo no son primitivas y hay que aplicar la técnica de descomposición **Top-Down**.

a. Leer los valores correspondientes a las horas trabajadas, el código del turno y el día.
b.1. Si el turno es nocturno entonces
b.2. calcular el jornal de acuerdo con la **fórmula 1**
b.3. sino
b.4. calcular el jornal de acuerdo con la **fórmula 2**
b.5. fin si
c. Escribir el valor del jornal

Dado que la acción "calcular el jornal de acuerdo con la **fórmula 2**" no es primitiva, se debe realizar un nuevo refinamiento.

a. Leer los valores correspondientes a las horas trabajadas, el código del turno y el día.

b. Si el turno es nocturno entonces
b.1. Jornal ← 150 * horas_trabajadas
b.2. sino
b.2.1. Jornal ← 120 * horas_trabajadas
b.2.2. si el día es domingo entonces
b.2.3. Jornal ← 120 * horas_trabajadas + 80 * horas_trabajadas
b.2.4. fin si
b.3. fin si
c. Escriba el valor del jornal.

Ahora todas las acciones son primitivas y se pueden elegir las variables del algoritmo que ya fueron subrayadas en el diagrama descendente.

Variables	Descripción
HORAS	Variable de entrada de tipo numérico, cuyo valor es la cantidad de horas trabajadas en un día.
TURNO	Variable de entrada de tipo numérico entero, cuyo valor es el código del turno.
DIA	Variable de entrada, de tipo cadena, cuyo valor es el nombre del dia.
JORNAL	Variable de salida, de tipo numérico real, que contiene el pago que debe efectuarse.

El algoritmo completo resulta:

```
Entero TURNO, HORAS
Cadena[20] DIA
real JORNAL
Inicio
     Lea HORAS, TURNO, DIA
     si TURNO >= 10 entonces    /* Turno noche */
          JORNAL ← 150 * HORAS
     sino
          JORNAL ← 120 * HORAS
          si DIA = "domingo" entonces
               JORNAL ← JORNAL + 80 * HORAS
          fin si
     fin si
     Escriba JORNAL
Fin
```

2. Dadas las siguientes variables numéricas reales, expresarlos en notación de punto flotante normalizado con la precisión indicada en cada caso.

a) X = 347.5 con precisión de dos dígitos significativos

La notación en punto flotante es una notación especial en la cual sólo se usa un cierto número de dígitos de precisión seguidos por una potencia de diez adecuada. La notación en punto flotante normalizado, consiste en colocar la primera cifra significativa después del punto decimal. Dado un número, primero hay que determinar la precisión a utilizar y analizar si el número deberá ser

truncado o no. En el caso de que se deba truncar el número se debe efectuar un redondeo sobre el último dígito de precisión sumándole uno en caso de que el primer dígito descartado sea mayor que cinco. Luego hay que ajustar adecuadamente la magnitud del número: por cada lugar decimal que se mueva hacia la izquierda, se suma uno a la potencia de diez que se utiliza como multiplicador y por cada lugar que se mueve hacia la derecha, se resta uno a dicha potencia. En este caso:

X = 347.5 con dos dígitos de precisión, será:

$X = 0.35 * 10^{3}$ (El primer dígito descartado es un 7, por lo tanto, se suma un 1 al 4. El exponente 3 expresa el corrimiento del punto decimal tres lugares hacia la izquierda).

b) Y = 0.0003132 con precisión de tres dígitos significativos

$Y = 0.313 * 10^{-3}$ (En este caso, al descartar el 2 no se modifica el último dígito de precisión. El exponente -3 obedece a que el punto decimal fue trasladado tres lugares hacia la derecha.)

3. Evaluar las siguientes expresiones:

a. 5 * (7 + 3) / 2 + 1

Las reglas que permiten evaluar una expresión aritmética pueden resumirse del siguiente modo:

En primer lugar se resuelven las expresiones entre paréntesis antes de ser combinadas con otras porciones de la expresión completa. En el caso de paréntesis internos a otros paréntesis primero se evalúa la subexpresión ***más interna***.

Las operaciones aritméticas dentro de una expresión se ejecutan de acuerdo al orden de precedencia ya expuesto:

$$5*(\underbrace{7+3}_{10})/2+1$$

$$\underbrace{5*10}_{50}/2+1$$

$$\underbrace{50/2}_{25}+1$$

$$\underbrace{25+1}_{26}$$

4. Sean A, B, C y D variables numéricas: escribir los predicados correspondientes a los siguientes enunciados:

a. Las variables A, C y D tienen el mismo valor.

Respuesta: En este caso se pueden escribir distintos predicados que cumplan con el enunciado.

Por ejemplo:

A = C Y A = D

A = C Y C = D

A = D Y C = D

5.2 Ejercicios Propuestos

1. *Diseñar un algoritmo para inflar una rueda de bicicleta con un inflador de mano con manguera utilizando el diseño Top-Down.*

Ambiente
- inflador c/manguera
- rueda de bicicleta desinflada

Acciones primitivas
- poner manguera
- girar una vuelta la manguera a derecha o izquierda
- accionar 5 veces el émbolo del inflador

Condiciones
- manguera ajustada
- manguera suelta
- rueda inflada

2. *Diseñar un algoritmo para cambiar una lamparita quemada de un artefacto suspendido del techo. Utilizar la técnica de refinamiento sucesivo.*

Ambiente:
- lamparita quemada.
- lamparita nueva.
- escalera.

Acciones primitivas.
- situar la escalera debajo del artefacto que tiene la lamparita quemada.
- subir un peldaño de la escalera.
- bajar un peldaño de la escalera.
- dar un giro a la lamparita.
- poner lamparita.
- sacar lamparita.

Condiciones:
- la mano alcanza la lamparita.
- llegar al piso.
- lamparita suelta.
- lamparita ajustada.

3. *Descomponer la subtarea **t1.2** del ejemplo dado en 1.2.5*

4. *Sumar los números 29 y 45 usando una calculadora de bolsillo. Suponga que no tiene conocimiento de como utilizar la calculadora.*

5. *Indicar el tipo de cada una de las siguientes constantes.*

   -145 -145. -145.00 -145.01

6. *Describir el procesador (acciones primitivas soportadas y condiciones capaces de ser evaluadas) y el ambiente para desarrollar algoritmos que resuelvan los siguientes problemas:*
   a. Calcular la media aritmética de tres números con una calculadora de bolsillo.
   b. Buscar el rey de copas en un mazo de naipes.
   c. De una sola tirada de cinco dados, informar si se forma generala, tomando de un dado por vez y comparando de a dos dados.
   d. De un mazo de cartas españolas se desean formar cuatro pilas, una para cada palo, teniendo en cuenta que solo se puede mirar de una carta por vez.
   e. De una pila de tarjetas numeradas, informar si están o noten orden ascendente, tomando de a una tarjeta por vez y comparando de a dos números entre sí (no hay dos números iguales).
   f. Hacer una clasificación de los diferentes tipos de datos y dar ejemplos de cada uno de ellos. Indicar el tipo de cada una de las siguientes constantes:

      a) J = 517 b) V = 'F' c) A = $-2.106 * 10^{-3}$
      d) F = 517.0 e) H = F f) L = -615
      g) K = '517.0' h) B = -420.0 i) E = 0.03

7. *Dadas las siguientes variables reales, expresarlas en notación de punto flotante normalizado con una precisión de tres dígitos significativos.*

   a) X = 84.58 b) A = 3417.6101 c) P = -6173.0
   d) Y = 0.0001375 e) B = -0.40032 f) Q = 0.103
   g) Z = 1.2 h) C = 0.012422 i) R = 0.12358

8. *Escribir las siguientes expresiones usando los operadores aritméticos:*

$$G = \frac{(X * Y)^2}{X + Y}$$

Respuesta:
$G = ((X * Y)\text{^}2) / (X + Y)$

a) $F = \dfrac{A + B}{C - D}$ b) $Y = \dfrac{A + \dfrac{B + C}{D}}{\dfrac{E}{F + G}}$

c) $Z = \dfrac{\dfrac{A}{B} - 1}{1 + \dfrac{C}{D}}$ d) $X = \dfrac{\dfrac{A}{B + C} - \dfrac{1}{D - E}}{\dfrac{F}{A} + \dfrac{H^2}{1 + Y}}$

9. *Evaluar las siguientes expresiones:*
   a. 4 + 7 * 2 - (3 - 4)^2

b. 2 + ((5 - (9/3)) * 2 + 4)
c. 2/(1/4 + 2/(1/4 + 2))
d. 4 + 2 * 3.0^2
e. (4 * 3/6 + 1/3) - 15/(3^2 - 6)

10. *Dadas dos constantes numéricas enteras P=5 y Q=10 evaluar los siguientes predicados:*

a. $\underbrace{3 * P < Q}_{F}$

b. $\underbrace{\underbrace{P < 10}_{F} \text{ Y } \underbrace{Q = 15}_{F}}_{F}$

c. $\underbrace{F \text{ Y } F}_{F}$

d. P < 8 O Q >= 15
e. NO (P + 1 < 8 Y Q < 30)
f. 2 * Q <> 15 O 3 * P >= Q
g. P > 18 Y (Q < 50 Y Q >= 10)
h. 2 * P >= Q Y Q <= 3 * P
i. (Q > 2 * P O P * Q = 50) Y P + Q = 20
j. NO (10 * Q < 80 Y (20 * P <> 10 * Q O Q >= 10))
k. NO (NO (20 * P <= 150 O P < Q) O 30 * P > 200) Y Q = 15

11. *Sean A, B y C variables numéricas; M y N variables cadena y R y S variables lógicas, cuyos valores son:*

A = 8 C = 10 N = “computadora” S = V
B = -2 M = “cómputo” R = F

Se pide evaluar los siguientes predicados:

a. (A < 4 Y R) Y (M < N O A = C)
b. NO (M < 'computación') O (B < A Y (NO(A <> 2) Y S))
c. NO ((B > 0 Y R) O (R O (S O (R Y N > “compra”))))
d. ((NO (NO S O R) Y (R O R)) O NO (NO S Y S)) O (S Y S)

12. *Sean A, B, C y D variables numéricas: escribir los predicados correspondientes a los siguientes enunciados:*

a. Los valores de B y C son superiores al valor de A.
b. Los valores de A y B son iguales pero difieren del valor de C.
c. El valor de A es superior al de B y está estrictamente comprendido entre los valores de C y D.
d. El producto de A y B difiere en 5 de la suma de C y D.
e. Los valores de todas las variables son distintos entre sí y ninguno de ellos es mayor que 10.

6 Bibliografía

Apuntes de cátedra. Equipo de cátedra de Computación I e Informática I. Edición 2017.

Computational Thinking. Wing, Jeannette Marie (2006). Communications of the ACM, Vol. 49, No. 3. https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf (accedido en Mayo del 2018)

Computational Thinking: What and Why?. Wing, J.M. (2010). the LINK. https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why (accedido en Julio del 2019)

Abstracción y aprehensión. Diseño top-down de algoritmos. Parte 1 y 2. Mario Rodríguez Rancel. Curso gratuito "Bases de la Programación Nivel 1". www.aprenderaprogramar.com (accedido en Junio del 2016).

El Discurso del Método. René Descartes (1637). http://www.posgrado.unam.mx/musica/lecturas/LecturaIntroduccionInvestigacionMusical/epistemologia/Descartes-Discurso-Del-Metodo.pdf (accedido en Noviembre del 2016)

Top-down design and the modular approach: extra note. Mr. Smith's Weblog (2007). https://mrsmith321.wordpress.com/2007/09/17/top-down-design-and-the-modular-approach-extra-note/ (accedido en Mayo del 2019).

Top-Down Design Tools: Managing Complex Assemblies. Remmers, Victor. Holland Engineering Consultants BV. EE.UU. PTC 26.02.2009. pg.5

Programación Estructurada. Autor anónimo (2006). http://assets.mheducation.es/bcv/guide/capitulo/8448148703.pdf (accedido en Diciembre del 2012).

Flow diagrams, Turing machines and languages with only two formation rules. C. Böhm and G. Jacopini. Communications of ACM 9 (May 1966), 366–371. http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf (accedido en Noviembre del 2015)

Notes On Structured Programming. E.W. Dijkstra. http://dl.acm.org/citation.cfm?id=1243381 (accedido en Febrero del 2016).

A case against the GO TO statement. E.W. Dijkstra. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html (accedido en Abril del 2015).

Program Development By Stepwise Refinement. Wirth, Niklaus (Abril 1971). http://dl.acm.org/citation.cfm?id=362577 (accedido en Diciembre del 2016)

LPP (2016) https://mediatecnica.weebly.com/lpp.html