Universidad Nacional Autónoma de México

Facultad de Ciencias

Propuesta de tercer curso de programación para la Licenciatura en Ciencias de la Computación

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
MAXIMILIANO MONTERRUBIO GUTIÉRREZ

DIRECTOR DE TESIS:
DRA. ELISA VISO GUROVICH

2009

Propuesta de tercer curso de programación para la Licenciatura en Ciencias de la Computación

Maximiliano Monterrubio Gutiérrez

Hoja de Datos del Jurado

1. Datos del alumno Monterrubio Gutiérrez Maximiliano 55 29 38 66 54 Universidad Nacional Autónoma de México Facultad de Ciencias Ciencias de la Computación 405074855
2. Datos del tutor Dra. Elisa Viso Gurovich
3. Datos del Sinodal 1 Dr. Sergio Rajsbaum Gorodezky
4. Datos del Sinodal 2 Dr. José de Jesús Galaviz Casas
5. Datos del Sinodal 3 M. en I. María de Luz Gasca Soto
6. Datos del Sinodal 4 L. en C.C. Francisco Lorenzo Solsona Cruz
7. Datos del trabajo escrito Propuesta de tercer curso de programación para la Licenciatura en Ciencias de la Computación 165 páginas 2009

Índice general

Introducción

Hoy en día, el papel de la Ciencia de la Computación para la humanidad ha empapado una gran cantidad de áreas tanto de la investigación científica como la industria. Esta circunstancia ha causado que el perfil del científico de la computación moderno, a diferencia de los grandes matemáticos que fueron padres de esta ciencia a principios de siglo XX, sufra de manera simultánea una metamorfosis la cual permita adecuarlo al entorno profesional actual. En el presente, el científico de la computación no sólo se avoca a resolver problemas de carácter abstracto o de índole puramente matemática, sino que ahora juega un papel activo en los procesos de desarrollo tecnológico, específicamente en la industria del desarrollo de software y hardware.

La sofisticación tecnológica y la explosión del mercado de la tecnología en el ámbito comercial ha coadyuvado en la investigación y el desarrollo de esta ciencia como una parte fundamental del quehacer científico actual así como de la industria, tanto local como de clase mundial, ya sea pública, privada, o inclusive del sector gubernamental. Esta última circunstancia ha causado que los nuevos profesionales de la computación deban mejorar de manera sensible sus habilidades en desarrollo de software, específicamente en la programación, la cual es una habilidad fundamental para la manufactura, prueba y mantenimiento de estos productos.

Este trabajo tiene como objetivo proponer metodologías y material didáctico que ayuden al estudiante de Ciencias de la Computación de la Facultad de Ciencias a motivar, ampliar y mejorar sus prácticas y conocimientos en lo que a programación se refiere.

Capítulo 1

Motivación

1.1. Situación actual

Actualmente, el plan de estudios de la licenciatura en Ciencias de la Computación ofrece una sólida base teórica y científica para los estudiantes. Esto se puede ver reflejado al revisar que los primeros cuatro semestres del plan de estudios comprenden una gran cantidad de asignaturas enfocadas exclusivamente a la formación matemática del estudiante.

Por otro lado, si nos enfocamos en los cursos de la licenciatura donde una parte primordial se refiera a desarrollar la habilidad de programar en los estudiantes, sólo podemos vislumbrar dos cursos introductorios de programación:

1. Introducción a las Ciencias de la Computación I, donde se cubre el objetivo de aprender a programar.

2. Introducción a las Ciencias de la Computación II, el cual introduce el uso e implantación de estructuras de datos.

Posteriormente, pasando el quinto semestre, existen algunos cursos donde programar se convierte en materia importante como parte del desarrollo del curso (a través de prácticas o proyectos); sin embargo, dichos cursos no tienen como objetivo desarrollar esta habilidad en los estudiantes, sino utilizar la programación como una herramienta que ilustre el contenido de dichos cursos, pasando a un segundo plano el desarrollo de esta habilidad.

1.2. Entorno profesional en el exterior

Enfocándonos en el desarrollo de tecnología aplicada, principalmente software, la situación actual demanda recursos humanos con una alta experiencia y habilidad en el área de algoritmos, estructuras de datos y programación, esto debido al esfuerzo sustancial que sector comercial, sector público e instituciones

de investigación en el desarrollo de nuevas tecnologías deben invertir. Actualmente podemos decir que la computación se encuentra virtualmente en todos los nichos del mercado, lo cual incentiva su desarrollo.

Debido a la continua sofisticación de los sistemas de cómputo actual (incluyendo las componentes de hardware y software), los ingenieros y científicos aplicados a esta área deben contar con más habilidad y experiencia, en comparación con hace dos o tres décadas atrás. Específicamente, en el nicho del desarrollo de software, el nivel de complejidad de los sistemas ha crecido a un grado tal, que ahora la mayoría de los proyectos importantes son manejados por miles de personas alrededor del mundo utilizando novedosas técnicas de diseño y administración de proyectos. Sin pensar en proyectos muy grandes, actualmente para casi cualquier proyecto competitivo de desarrollo, aun de alcance local, es necesario contar con diversos recursos y personal para la realización del mismo.

Esto hace que la labor de los programadores se haga cada vez más compleja y requiera que tanto las partes del diseño y análisis, como la parte de implantación y prueba cuenten con una amplia experiencia y habilidad en la programación de computadoras, esto con la finalidad de desarrollar productos de mejor calidad y poder establecer un puente entre el aspecto teórico y el aplicado. Al hablar de establecer un puente, me refiero a poder conjuntar el trabajo teórico y el práctico de manera eficiente; es común desarrollar algoritmos eficientes que resuelven problemas complejos, sin embargo la implantación debe hacer uso de las estructuras de datos y técnicas que eficienten los desarrollos teóricos para así desarrollar herramientas útiles y de calidad.

1.3. Experiencia personal

En mi opinión, me he enfrentado a ciertas situaciones que sugieren la necesidad de una mejora sustancial en ciertos conocimientos y habilidades para los egresados de ciencias de la computación. Menciono dos eventos que de hecho fueron una parte muy importante para mi motivación personal en la realización de este trabajo, pero que además evidencian la necesidad de desarrollar otras habilidades que, si bien tienen mucha relación con la programación, son conocimientos que actualmente cualquier profesional de la computación debería tener. Dichos elementos se enlistan en cada apartado.

1.3.1. International Collegiate Programming Contest de ACM

El Concurso Colegial Internacional de Programación (en inglés *International Collegiate Programming Contest*[1] (ICPC), que organiza ACM[2] es uno de los concursos más importantes de programación a nivel mundial, tanto por la importancia de la institución que lo organiza, como también por la difusión y el patrocinio que realiza la empresa IBM para la realización del mismo.

El concurso consiste en resolver entre seis y siete problemas teniendo cinco horas de tiempo. El ganador es el equipo que resuelva la mayor cantidad de problemas, o en caso de empates, aquel equipo cuya suma de tiempos de ejecución para cada problema sea la menor. Se pueden presentar los programas en tres posibles lenguajes: `Java`, `C` ó `C++`. (Para mayor información referirse a la página web del concurso).

Debido al enfoque que tiene ACM como organización de investigación y desarrollo en computación, los problemas propuestos para el concurso ICPC se especializan fuertemente en el área de algoritmos, combinatoria y teoría de gráficas, dándole una gran importancia a la eficiencia de los programas y sobre todo, a evaluar la habilidad para programar de los concursantes, buscando resolver la mayor cantidad de problemas en el menor tiempo posible.

A juzgar por la experiencia que he tenido en este concurso (he participado dos veces, en 2006 y 2007), podría clasificar los problemas de la siguiente manera:

1. **Problemas de combinatoria**. Generalmente son problemas muy sofisticados de combinatoria, que requieren una gran habilidad para eficientar los algoritmos que los resuelven, así como tener mucha experiencia resolviendo problemas de ese tipo. Básicamente consisten en calcular cardinales de conjuntos de permutaciones de cadenas o conjuntos bajo ciertas restricciones.

2. **Teoría de gráficas**. En cada concurso siempre existe al menos un problema de teoría de gráficas. Los hay de todos tipos y dificultades, desde coloración de gráficas, hasta problemas NP-Completos.

3. **Algoritmos**. Se especifica el funcionamiento de un algoritmo y la meta es implantarlo rápida y eficientemente. Es un tipo de ejercicio muy común en el concurso, generalmente son esquemas de codificación, manejo sofisticado de matrices o cualquier tipo de algoritmo que utilice intensamente estructuras de datos sencillas, generalmente arreglos.

Después de haber participado en estos eventos considero que los estudiantes de ciencias de la computación, si bien conocemos de manera concisa los conceptos que se mencionan arriba, desafortunadamente carecemos de la experiencia para poder desarrollar de manera rápida y eficiente esas soluciones. Esto último no significa que no seamos capaces de realizarlas, de hecho es lo que apoya

[1] `http://icpc.baylor.edu`
[2] Association for Computing Machinery. (`http://www.acm.org`)

y fortalece más nuestra formación académica impartida con una perspectiva científica y sobre todo de carácter matemático formal; sin embargo, al aplicar este conocimiento al ámbito del código, carecemos de la habilidad suficiente, comparándonos con otras instituciones. Esto se evidencia al momento de consultar los resultados de los concursos y dar cuenta de quiénes son las instituciones que mantienen los primeros lugares año con año en México.

Esta carencia es causada por la falta de práctica en programación enfocada a la solución de problemas más complejos y no de carácter conceptual. Durante los cursos de ICC[3] 1 y 2 se resuelven problemas interesantes y se estudia a profundidad estructuras de datos; sin embargo, el tipo de problemas propuestos y el enfoque que se da en dichas materias atiende más a dar las bases a los estudiantes para poderse iniciar de manera rápida y sólida en la programación de computadoras, mas no a profundizar las habilidades y aplicación de conceptos para la solución rápida y eficiente de problemas. Análogamente sucede en los cursos posteriores donde, como menciono anteriormente, la programación se convierte en una herramienta y no en un objetivo.

1.3.2. Estancia en Microsoft

El día 12 de octubre de 2007 en el auditorio Raúl J. Marsal, ubicado en el edificio de la División de Estudios de Posgrado de la Facultad de Ingeniería (*DEPFI*) de la UNAM, se llevó a cabo un evento de reclutamiento por parte de la empresa Microsoft[4] al público en general. Dicho evento consistía en reclutar estudiantes de licenciatura o posgrado para ser parte de la división de desarrollo de dicha empresa, ya sea por medio de una estancia de aproximadamente tres meses (*Internship program*), o como empleado de tiempo completo. En ambos casos, dichas actividades se desempeñan en las instalaciones principales de Microsoft ubicadas en Redmond, Washington, Estados Unidos, o posiblemente en las instalaciones ubicadas en la conocida región de Sillicon Valley, ubicada en el estado de California del mismo país.

El proceso de selección consiste en tres etapas:

1. **Presentación de *curriculum vitae* y entrevista preliminar**. Esta etapa se desarrolla el mismo día del evento y se aplica para todos los estudiantes que tienen interés en participar en el concurso de selección. Se entrega el documento directamente a los reclutadores y posteriormente se realiza una breve entrevista (alrededor de 25 minutos) con alguno de los reclutadores. Dicha entrevista es en inglés y generalmente consistía en hablar un poco de la experiencia personal y resolver un par de problemas de estructuras de datos o algoritmos.

2. **Entrevista en la sede corporativa de Microsoft en México**. En caso de haber sido seleccionado en la etapa anterior, se convoca a una entrevista posterior en las instalaciones de Microsoft en México. Esta fase consiste

[3] Introducción a las Ciencias de la Computación
[4] http://www.microsoft.com

exclusivamente de una entrevista con duración de 25 minutos enfocada a explorar las habilidades del solicitante en las áreas de diseño orientado a objetos, programación, depuración y desarrollo de software para prueba automatizada.

3. **Serie de entrevistas en las instalaciones principales de Microsoft (Redmond, WA)**. La prueba final, consiste en cuatro entrevistas de una duración aproximada de una hora. En este caso se profundiza mucho más en la exploración de habilidades del estudiante o egresado en las áreas mencionadas en la segunda etapa. Para esta etapa, la dificultad de los problemas es sustancialmente mayor y la presentación de las soluciones propuestas es mucho más a detalle.

Para conocer con más precisión el proceso de selección se puede consultar en la web la parte de reclutamiento en Microsoft, en la sección *Microsoft College Careers*[5]

En este entorno específico, las primeras dos etapas de este proceso deberían ser razonablemente sencillas para cualquier estudiante dedicado a la licenciatura; sin embargo, al llegar a la tercer fase, hace falta desarrollar más la habilidad de programar, especialmente bajo presión (al igual en el caso de los concursos en ACM), emprendrendiendo un enfoque más práctico.

Otra parte importante a desarrollar es la que se refiere a las prácticas de programación que faciliten la depuración de errores y aprender a desarrollar software para pruebas (en inglés, *test suites*) las cuales son de uso muy común en la industria del desarrollo de software y hardware.

1.4. Recomendación

Habiendo mencionado algunos aspectos que considero representan parte de la problemática actual en torno a la formación de los estudiantes de la licenciatura respecto a los temas de programación, es importante proponer un espacio donde se desarrollen estas habilidades de manera más gradual y sistemática, tomando muy en cuenta la parte práctica, es decir, el desarrollo de una habilidad. De este modo, dicho espacio servirá como parte de la formación al perfil de egreso del estudiante de ciencias de la computación.

Como dijo el Dr. Pérez Pascual, director de la Facultad de Ciencias cuando se fundó la carrera:

> *"Es condición necesaria de todos los egresados de la licenciatura en ciencias de la computación que puedan programar con los ojos cerrados."*

Así pues, el motivo principal para el desarrollo de este trabajo se encuentra enfocado en elevar el nivel competitivo de los egresados de la licenciatura primordialmente en el ámbito industrial, es decir, impulsar el nivel de programación

[5] http://www.microsoft.com/college

de los estudiantes con la finalidad de incursionar en la industria de desarrollo de software, ya sea de alcance nacional o internacional, comprendiendo el ámbito privado o el sector público.

Capítulo 2

Lenguajes de Programación

En el desarrollo de este capítulo se presenta un panorama general respecto a los lenguajes de programación de uso más común en la actualidad; enumeraremos sus características y buscaremos clasificarlos con base en la filosofía de diseño que manejan, con la finalidad de establecer un criterio para su utilización en la solución de problemas; específicamente poder decidir, dado un problema, qué lenguaje es el más conveniente para atacarlo.

Adicionalmente, se mencionan algunos aspectos importantes en los lenguajes de programación que afectan tanto la fase de diseño, el estilo de desarrollo y el desempeño en tiempo de ejecución, aspectos de suma importante en la selección de un lenguaje de programación para la solución de un problema. Enumeraremos un conjunto de características importantes en la tecnología de intérpretes y compiladores que son de uso común en la actualidad y tabularemos algunos lenguajes de uso común para resumir dichas características.

2.1. Paradigmas de programación

Aunque existe una gran cantidad de paradigmas de programación, en el desarrollo de este trabajo tomaremos en cuenta exclusivamente dos paradigmas y dos subclasificaciones de cada uno de ellos. La razón es que la gran mayoría de los lenguajes de programación de propósito general de uso más amplio en la industria y la educación se pueden clasificar en estas categorías.

2.1.1. Programación imperativa y declarativa

Para los lenguajes de programación de propósito general, podemos clasificarlos en dos grandes grupos basándonos en el estilo de diseño y de desarrollo: los lenguajes *imperativos* y *declarativos*.

2.1.2. Programación imperativa

En resumen, podemos decir que este paradigma describe el cómputo como una secuencia de *enunciados* o instrucciones que alteran el *estado* de un programa. En general, podemos decir que un lenguaje imperativo representa un programa como un conjunto de comandos que la computadora ejecuta.

Una comparación conceptual muy aproximada a lo que es un programa escrito en el paradigma imperativo es una receta de cocina. Una receta de cocina nos muestra el conjunto de elementos que necesitamos para realizar el producto (los ingredientes) y nos muestra un procedimiento detallado que nos explica qué transformaciones hay que aplicar a los ingredientes para obtener el resultado. Análogamente, un programa imperativo utiliza como ingredientes un conjunto de variables o atributos y aplica procedimientos bien definidos que alteran el estado de dichos elementos, dándonos así una noción similar a lo que es una receta.

En el paradigma de la programación imperativa podemos vislumbrar dos grandes *subparadigmas* que se utilizan para programar en la actualidad. Aunque existen algunos más, estos dos son los que más popularidad han adquirido y son los que se usan con más frecuencia en la industria y en la educación:

1. Programación estructurada.
2. Programación orientada a objetos.

2.1.3. Programación declarativa

La programación declarativa se llama así porque el estilo de programación se enfoca en especificar la morfología de un problema y no tanto el procedimiento para resolverlo; es decir, nos concentramos en el *qué* y no en el *cómo*. En particular existen dos derivaciones importantes de este paradigma: la programación funcional y la programación lógica.

En general, podemos relacionar un programa escrito en un lenguaje declarativo de manera análoga a un plano de un edificio: un plano representa la estructura y morfología del edificio, aunque no define la forma en cómo se va a construir, es decir, muestra el *qué* y no el *cómo*. En general los programas escritos en un lenguaje lógico se apegan totalmente a esta definición, mientras que los lenguajes funcionales lo hacen en un menor grado aunque siguen este espíritu.

La principal diferencia entre la programación declarativa y la imperativa radica en la noción de estado. En los lenguajes imperativos podemos tener control del estado de un programa a través de variables o atributos de una clase, mientras que en un lenguaje declarativo no tenemos noción de estado, todo se presenta desde un punto de vista morfológico, el cual no especifica un procedimiento que transforma el estado del programa de manera explícita.

Un aspecto importante en estas dos grandes categorías es que los lenguajes de programación imperativos han sido muy exitosos y de uso muy amplio en la industria, mientras que los lenguajes declarativos tienen más penetración

en el ámbito académico y de investigación. Esto último surge debido a que los lenguajes declarativos son de muy alto nivel de abstracción y poseen propiedades lógicas y de carácter formal muy interesantes, las cuales son de suma importancia en la teoría de la computación, lógica y otras áreas, mientras que los lenguajes imperativos, debido a su estilo de especificar procedimientos, tienen un papel más activo en la industria tanto porque se someten a un proceso de ingeniería además de proporcionar la posibilidad de obtener intérpretes y compiladores de muy alto rendimiento, haciéndolos más atractivos para sistemas grandes.

A continuación se enumeran las características principales de estos cuatro estilos y se ejemplifican algunos lenguajes y aplicaciones.

2.1.4. Programación estructurada

Este subgénero de la programación imperativa comprende todos los lenguajes cuya unidad de cómputo es el *procedimiento* o *función*. Todo problema se puede subdividir o modularizar en un conjunto de procedimientos, y la solución se convierte en una secuencia o composición de ellos.

El ejemplo canónico de este paradigma es el lenguaje de programación `ALGOL` desarrollado a mediados de los años cincuenta. Fue el primer lenguaje de uso generalizado con este paradigma de programación y la base de inspiración de muchos otros lenguajes en los años posteriores.

En la actualidad, el lenguaje estructurado más común es el lenguaje `C`, desarrollado por Dennis Ritchie en los Laboratorios Bell. El propósito inicial de este lenguaje era ser el estándar de desarrollo para toda la familia de sistemas operativos tipo `UNIX` aunque posteriormente se convertiría en uno de los lenguajes más populares en toda la industria. Su gama de aplicación va desde sistemas de cómputo tradicionales, hasta el mercado de hardware móvil como microcontroladores, teléfonos móviles y hardware embebido.

La ventaja principal de este paradigma de programación es que para las variantes de lenguajes compilados se puede realizar una gran cantidad de optimizaciones y desarrollar código muy eficiente para aplicaciones críticas. En general la programación estructurada se refiere fuertemente al uso de `C` como lenguaje de programación y su aplicación es de uso muy común en software de sistema debido a la obtención de código objeto muy eficiente así como la manipulación más directa del hardware a comparación de otros paradigmas.

Como ejemplos de lenguajes de uso común en la actualidad podemos mencionar:

1. `C`.

2. `ECMAScript` (`JavaScript, JScript, ActionScript`). Usado ampliamente para implementar funcionalidad en software para web, especialmente en navegadores.

3. `FORTRAN`. (*Formula Translation*). Usado ampliamente para cómputo científico.

4. `perl` (*Practical Extraction and Reporting Language*). De uso muy común en sistemas `UNIX`, se utiliza como lenguaje auxiliar en una gran cantidad de aplicaciones por su poderoso sistema de expresiones regulares, su sintaxis altamente compacta y su versatilidad. Algunos de sus usos más comunes son para procesar y presentar información en la web, procesamiento de grandes volúmenes de texto, procesamiento de archivos de bitácora (*logfiles*), entre otras labores.

2.1.5. Programación orientada a objetos

Surge como una extensión de la programación estructurada buscando acercarse más al proceso cognitivo natural del ser humano. En lugar de utilizar a los procedimientos como unidades de desarrollo, se define un concepto más general: el *objeto*. Un objeto modela el comportamiento de un ente en base a su *estado* y su *comportamiento*. El estado de dicho objeto se representa por medio de atributos y su comportamiento como métodos o procedimientos miembro. La principal ventaja de este modelo es que se induce el uso de modelos más claros, más mantenibles y además explota la posibilidad de extender funcionalidad de un objeto por medio de las características de *herencia* y *polimorfismo*; las dos características más importantes de este paradigma.

Las principales ventajas de la programación orientada a objetos se encuentran en la gran capacidad que tiene para reutilizar trabajo previo así como la etapa de mantenimiento y depuración en todos los procesos de ingeniería de software. Adicionalmente es una excelente herramienta para la educación, ya que los lenguajes orientados a objetos representan de manera más familiar el diseño de software a comparación de un lenguaje con otro paradigma. Esta última razón explica por qué en una gran cantidad de instituciones de educación se utiliza algún lenguaje de programación orientado a objetos para un primer curso de programación.

El lenguaje pionero de la orientación a objetos fue `Simula`, diseñado en los años sesenta como una extensión del lenguaje `ALGOL`. Se utilizó principalmente para simulación de fenómenos físicos y biológicos.

El lenguaje de programación que por primera vez se le otorgó el calificativo de orientado a objetos fue `Smalltalk`, desarrollado en los años setentas como un proyecto la empresa Xerox por Alan Kay, Dan Ingalls, Adele Goldberg, entre otros. El aspecto más importante de `Smalltalk` es su categórica influencia en los lenguajes de programación orientados a objetos más modernos.

Ejemplificando algunos lenguajes que pertenecen a esta categoría:

1. `C++`. Originalmente concebido como una extensión de `C`.
2. `C#`. Desarrollado por Microsoft como una alternativa a `C++` y como lenguaje principal para la máquina virtual `.NET`.
3. `Java`.
4. `Objective-C`. El lenguaje de programación para desarrollar software en la plataforma de Apple Computer. Muy similar a `Smalltalk`.

5. `Python`. Uno de los lenguajes interpretados más usados a nivel mundial. De uso muy amplio en los sistemas `UNIX`. Surge como alternativa de `perl`.

6. `Ruby`. Moderno lenguaje de programación, multiparadigma, aunque naturalmente orientado a objetos. Está cobrando mucha fuerza en la actualidad por el ambiente de desarrollo web *Ruby on Rails*.

2.1.6. Programación funcional

Este paradigma, inspirado principalmente por el cálculo lambda de Alonso Church, se refiere al conjunto de lenguajes que tratan al cómputo como un conjunto de funciones al estilo matemático las cuales, por su naturaleza, impiden la noción de estado. Se le da un énfasis a la aplicación de funciones, en contraste con la programación imperativa que enfatiza la transformación de un elemento de un estado a otro.

En general, un programa escrito con el paradigma funcional se puede ver como un conjunto de declaraciones de funciones, mientras que su ejecución es una secuencia de composición de funciones las cuales se ven puramente como su definición matemática: reciben como argumento un conjunto D (dominio) y el resultado es un elemento en otro conjunto no necesariamente distinto C (codominio).

Existen dos aspectos muy importantes que marcan fuertemente la diferencia entre el estilo imperativo y el estilo de programación funcional:

1. **Funciones de primera clase**. En un lenguaje funcional, es posible definir funciones que reciben funciones como argumentos y no necesariamente datos. Si bien en algunos lenguajes como `C` y `C++` es posible simular este comportamiento con apuntadores a funciones, el estilo funcional es mucho más claro y formal.

2. **Recursión**. En los lenguajes imperativos, el concepto de iteración viene fuertemente ligado a estructuras de control tipo `while` o `for` de `C`. Sin embargo en los lenguajes funcionales, si bien existen algunas estructuras de control similares, no es un estilo común utilizar este método para iterar, sino que siempre se utiliza la recursión con la finalidad de reforzar el estilo funcional, es decir, componer funciones en vez de especificar pasos a seguir.

El lenguaje funcional más conocido es `Lisp`, desarrollado por primera vez en 1958 por John McCarthy. Además de ser el lenguaje más famoso en el paradigma funcional, es también el más antiguo y de hecho, es el segundo lenguaje de programación de alto nivel más antiguo en la historia (siendo su único predecesor `FORTRAN`). En la actualidad tiene una gran cantidad de aplicaciones y sobre todo dialectos. Una de las aplicaciones más conocidas de este lenguaje de programación se encuentra en el popular editor de texto `GNU Emacs`, desarrollado por Richard Stallman. Algunos dialectos populares de Lisp son:

1. `Common Lisp`. La versión de `Lisp` estandarizada por ANSI[1]
2. `Emacs Lisp`.
3. `Scheme`.

Otros lenguajes funcionales de uso común que no son dialecto de `Lisp` son:

1. `ML`.
2. `Haskell`.
3. `erlang`.

2.1.7. Programación lógica

Utiliza la lógica matemática como estilo de programación y representa de manera pura el paradigma declarativo. La tarea de solución de un problema se divide en dos partes: inicialmente el programador especifica cláusulas lógicas que representen la validez de un programa, y el proceso de solución se le adjudica a un algoritmo de demostración lógica automática o un generador de modelos el cual se encarga de analizar las cláusulas y poder encontrar ejemplares que mantienen la veracidad de las cláusulas y que, por lo tanto, son solución del problema.

Este estilo de programación tiene mucha influencia en el ámbito académico y de investigación debido a las propiedades matemáticas de los lenguajes lógicos, así como una fuerte relación con temas de inteligencia artificial debido a la posibilidad de que un programa escrito en el estilo lógico sólo especifique ciertas características de la solución y no del procedimiento para encontrarlas.

`Prolog`

Diseñado en 1972 por Allan Colmerauer, es el lenguaje más representativo de la programación lógica y su aplicación inicial fue el procesamiento del lenguaje natural. En particular en la Facultad de Ciencias, `prolog` se utiliza como lenguaje de programación principal en los cursos de análisis lógico e inteligencia artificial.

2.2. Tecnología de compiladores e intérpretes

Una vez recorrido de manera somera la clasificación de los lenguajes de programación de interés para este trabajo, es importante comparar elementos tecnológicos relacionados a los lenguajes de programación con la finalidad de mejorar diversos aspectos inherentes al comportamiento de un programa. Cada uno de estos elementos se enfoca en mejorar diversas áreas del comportamiento de los lenguajes de programación:

[1] Instituto Norteamericano de Estándares Nacionales (*American National Standards Institute*). `http://www.ansi.org`

- **Desempeño**. Se emplean sofisticados algoritmos de procesamiento de lenguaje para optimizar el desempeño del código o de la biblioteca de tiempo de ejecución (*runtime library*).
- **Facilidad de uso**. Sofisticaciones realizadas en la sintaxis y/o semántica del lenguaje de programación con la finalidad de facilitar su uso. Un ejemplo muy importante de estas mejoras son todos los lenguajes de programación que proporcionan manejo automático de memoria (recolección de basura).
- **Legibilidad**. Utilerías o agregados al lenguaje de programación para facilitar su comprensión por un tercero. En general este tipo de mejoras se cristalizan en utilerías para generación de documentación automática o bien pueden ser parte del diseño de la sintaxis del lenguaje (muy común en los lenguajes orientados a objetos, se busca encapsular funcionalidad en piezas de código independientes para facilitar tanto su legibilidad como su mantenimiento).
- **Depuración**. Durante el proceso de producción de software es posible que la parte que toma más tiempo en el desarrollo de un programa sea la detección y corrección de errores, factor por el cual muchos compiladores e intérpretes han desarrollado sofisticadas sintaxis y bibliotecas para detectar y corregir defectos de código. En esta categoría podemos mencionar el nacimiento del manejo de excepciones en los lenguajes de programación orientados a objetos, así como la implementación de depuradores, los cuales son piezas de software que sirven explícitamente para buscar y corregir defectos de código. En la actualidad casi todo paquete de software para un lenguaje de programación (ya sea intérprete o compilador) proporciona un depurador para la búsqueda y corrección de errores.
- **Seguridad**. Debido a la creciente explotación de vulnerabilidades en el software, los nuevos lenguajes de programación se someten a diseños que contemplan mejoras en la seguridad y estabilidad del sistema. Para ejemplificar podemos mencionar compiladores que producen código ejecutable encriptado, el cual se decodifica en tiempo de ejecución, el manejo de código firmado o certificado (como es el caso de `Java`), entre otros.
- **Concurrencia**. Actualmente el cómputo paralelo ha escalado a tal grado, que ahora inclusive las computadoras personales proporcionan funcionalidad de multi-procesador en un solo empaque (estampa). Como es sabido, el cómputo paralelo representa una gran cantidad de retos en el área de coordinación y sincronización de recursos compartidos. Debido a esto, diversos lenguajes de programación proporcionan infraestructura semántica y sintáctica que enfrente a estos problemas. El lenguaje `Java`, por ejemplo, provee de infraestructura de sincronización de hilos de ejecución de manera intrínseca en el lenguaje por medio de la palabra reservada `synchronized`, así como el manejo de monitores, semáforos y bloqueo de datos compartidos entre hilos de ejecución.

En los lenguajes funcionales, gracias a la ausencia de la noción de estado de un programa, la concurrencia es un elemento mucho más cómodo de manejar. El hecho de no existir datos compartidos entre funciones (estado) nos proporciona la posibilidad de paralelizar de manera automática la ejecución del código. A esta característica se le conoce como *transparencia referecial*. En algunos lenguajes como `Common Lisp`, la biblioteca de tiempo de ejecución está implementada de tal modo que pueda explotar al máximo la concurrencia gracias a la transparencia referencial.

- **Portabilidad**. En la actualidad, las nuevas tendencias del cómputo buscan estandarizar procesos y mejorar la compatibilidad entre sistemas heterogéneos con la finalidad de reutilizar componentes de hardware y de software. Refiriendo a lo anterior, los lenguajes de programación también hacen un papel importante en esta convergencia de estándares. Existen lenguajes de programación en los cuales la portabilidad es un elemento fundamental en su diseño e implementación, como el caso de `Java`. Adicionalmente, lenguajes que originalmente se veían como dependientes de la arquitectura, poco a poco comprenden más y más código portátil de manera sencilla entre plataformas, como la biblioteca estándar de `C/C++` e inclusive la nueva máquina virtual .NET de Microsoft la cual ahora está disponible para sistemas `UNIX` gracias a la contribución del proyecto `Mono`, lidereado por el desarrollador de software mexicano Miguel de Icaza.

Capítulo 3

Estructuras de datos

Es importante introducir estructuras de datos fundamentales en la ciencia de la computación que no son visitadas en los cursos de Introducción a las Ciencias de la Computación I y II para así dar al estudiante una mayor gama de posibilidades al momento de resolver un problema que involucre estos conceptos. Existen en particular dos estructuras de datos que no se estudian a profundidad en dichos cursos y que se cubren a continuación.

3.1. Tablas de dispersión

Una tabla de dispersión es una estructura de datos que se utiliza para implementar operaciones de un diccionario. En computación un diccionario se define como una estructura de datos que soporta las siguientes operaciones:

1. **Inserción**.

2. **Búsqueda**. Dada una cierta *llave*, encontrar el elemento del diccionario que satisface dicha propiedad.

3. **Eliminación**.

Es claro que existen estructuras de datos ya conocidas que sirven para implementar este comportamiento (arreglos, colas y listas enlazadas por ejemplo). Sin embargo cada uno de ellos presenta ciertas desventajas:

1. En el caso de los arreglos, a pesar de tener un excelence rendimiento en acceso a elementos, reemplazo y eliminación de $O(1)$ (esto en caso de admitir elementos nulos en el mismo), la llave de búsqueda *forzosamente* debe ser numérica y representa el índice del elemento. Existen casos donde podemos necesitar que la llave de búsqueda no sea un índice sino algún otro elemento, como una cadena de caracteres.

2. Para las estructuras de datos lineales (listas enlazadas) tenemos el problema de que las operaciones de búsqueda pueden requerir en el peor caso de $\Theta(n)$, análogamente para la eliminación en caso de requerir la búsqueda de un elemento, por lo tanto para tablas muy grandes esto puede ser ineficiente. Aún teniendo una lista almacenada en un arreglo, si no contamos con una llave que proporcione el índice del elemento en el arreglo tendremos un comportamiento similar al de las listas enlazadas. Podemos mejorar este rendimiento manteniendo *siempre* ordenada nuestra lista, consiguiendo que nuestros tiempos de búsqueda y eliminación decrezcan a$\Theta(\log\ n)$, causando un impacto negativo en la inserción que nos costará$\Theta(\log\ n)$.

En vista de lo anterior, es clara la necesidad de desarrollar una nueva estructura de datos que nos pueda ofrecer el rendimiento de las operaciones de diccionario al igual que un arreglo (con índices como llaves), pero con la mayor versatilidad de las listas enlazadas. De ahí la motivación para el uso de tablas de dispersión o *hash tables*.

3.1.1. Definición

Una tabla de dispersión o *hash table* es una generalización natural de la noción de arreglo. Básicamente, en un arreglo la forma de calcular la dirección de un elemento se deriva directamente del índice en el mismo. En el caso de las tablas de dispersión, la dirección de un elemento *se calcula* a partir de una llave de búsqueda específica, es decir, mientras que en un arreglo la forma de obtener un elemento se define como:

$$search(i) = A[i],$$

en el caso de una tabla de dispersión utilizaremos una función h que calcula la dirección del elemento que buscamos en un arreglo como:

$$search(key) = A[h(key)] \text{ donde } h : T \to \mathbb{N}.$$

De donde T está representando el tipo de dato de la llave de búsqueda y a h se le conoce como *función de dispersión*.

Un ejemplo muy sencillo de la función de dispersión es $h(x) = x$ a la que se le conoce como *función de direccionamiento inmediato* y se aplica para los tipos de datos cuya llave de búsqueda es un número entero. De hecho si vemos a los arreglos como una tabla de dispersión, utilizan precisamente esta función para realizar la búsqueda de elementos.

En la práctica, existen aplicaciones donde el direccionamiento inmediato es ineficiente, ya que de toda la gama de posibles valores de llave de búsqueda, sólo se utilizan unos pocos; por lo tanto, para un universo amplio de posibles llaves, el reservar un arreglo puede ser muy ineficiente (debido al desperdicio de ubicaciones de memoria) o inclusive imposible debido a las restricciones de memoria principal.

Cuando el conjunto K de llaves utilizadas en una aplicación es mucho menor que toda la posible gama de llaves U, utilizando una función de dispersión podemos reducir el tamaño del arreglo que almacena los elementos de $O(|U|)$ a $O(|K|)$.

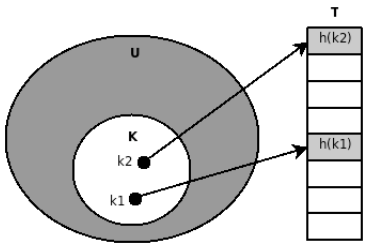

Figura 3.1: Una función de dispersión que mapea las llaves de búsqueda al arreglo T que almacena los datos.

3.1.2. Colisiones

Es común que al usar funciones de dispersión en un conjunto muy amplio de valores se pueda dar el caso de que $h(k) = h(l)$ con $k \neq l$. Este problema se conoce como *colisión*. Una forma sencilla de resolver el problema de colisión es utilizar la técnica de *encadenamiento* la cual consiste en usar un arreglo T de listas enlazadas en vez de contenedores, así que ahora el procedimiento para implementar el diccionario funcionaría así:

```
1  void insert(list t[], element x){
2          list_insert(t[h(x.key)]);
3  }
4
5  element search(list T[], key k){
6          // Busqueda lineal en la lista
7          // que contiene la llave especificada
8          return list_search(T[h(k));
9  }
10
11 void delete(list t[], element x){
12         // Delete utiliza tambien una busqueda
13         // lineal para realizar la eliminacion,
14         // se puede evitar esta situacion usando
15         // una lista doblemente ligada.
16         list_delete(t[h(x.key)]);
17 }
```

Es decir, al realizar una inserción, primero encontramos la lista enlazada correspondiente con la función de dispersión y posteriormente utilizamos el procedimiento de inserción, búsqueda o eliminación en la lista para cada operación respectivamente.

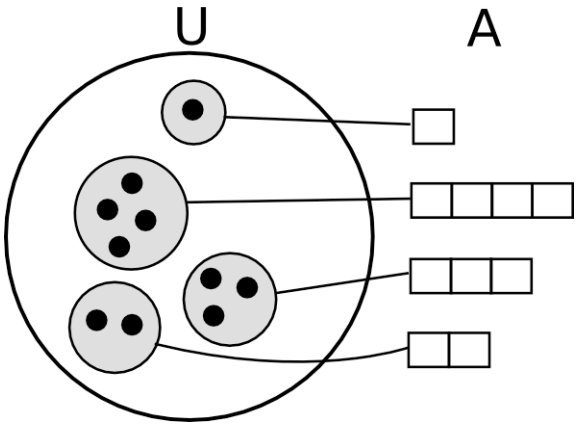

Figura 3.2: Una función de dispersión que mapea las llaves de búsqueda a un arreglo de listas enlazadas A que almacena los datos.

3.1.3. Funciones de dispersión

Una vez definida lo que es una tabla de dispersión es inmediato preguntarse cómo es posible diseñar funciones de dispersión eficientes, es decir, que distribuyan lo más uniformemente posible las llaves a través de todo el arreglo de datos y que además minimicen en lo posible las colisiones. En general una buena idea para desarollar una función de dispersión eficiente es convertir nuestro dominio de llaves en un dominio de enteros, por ejemplo, si vamos a desarrollar una función de dispersión que usa como llaves cadenas de caracteres, podemos interpretar dicha cadena como un polinomio base 127 (el número de caracteres del conjunto `ASCII`), evaluarlo y posteriormente aplicar h. Dicho polinomio se definiría como:

$$n(x) = x[0] + 127x[1] + 127^2x[2] + \cdots + 127^{len(x)-1}x[len(x) - 1]$$

Método de la división

El método de la división está basado en mapear una llave k dentro de un arreglo de m elementos de la siguiente manera:

$$h(k) = k \mod m$$

Al utilizar este método debemos escoger cuidadosamente el valor de m; por ejemplo m no debe ser un primo de la forma $p = 2^k - 1$ ya que calcular $h(k)$ es análogo a tomar los k bits menos significativos de p. Contrariamente, lo más recomendable es buscar un número primo lejano a una potencia de 2 para que así utilicemos en su totalidad todos los bits de k al calcular $h(k)$.

Método de la multiplicación

Otra forma para diseñar funciones de dispersión es utilizar el método de la multiplicación. Este método consiste en seleccionar un valor constante $A \in [0, 1]$ y multiplicarlo por la llave dada, seguido de obtener la parte fraccional de este producto, posteriormente multiplicar ese valor por una constante $m \in \mathbb{N}$ y tomar el piso. Este último valor será el valor de dispersión de la llave dada. Así pues, la función h quedaría como:

$$h(k) = \lfloor m\,(kA - parteEntera\,(kA))\rfloor$$

En este caso, el valor crítico para el método de la multiplicación no es m sino A. En *The Art of Computer Programming* por Donald Knuth, se sugiere que se utilice el valor $1 - \phi$:

$$A = 1 - \phi = 1 - \left(\frac{1+\sqrt{5}}{2}\right) \approx 0{,}61803398$$

3.2. Árboles-B

3.2.1. Motivación

La práctica más común en el área del análisis de algoritmos es analizar la propuesta de un algoritmo en base a dos criterios: *tiempo de ejecución* y *consumo de memoria*; esto en la mayoría de los casos es suficiente y nos ofrece una buena metodología para buscar y diseñar algoritmos que optimicen estas dos variables, sin embargo existen ciertas aplicaciones donde hay una tercer variable que es muy importante considerar: *los accesos a disco*. Inmediatamente surgen preguntas al respecto:

- *¿Por qué analizar accesos a disco?* En aplicaciones donde se realizan accesos intensos a disco es muy importante minimizar dicho proceso ya que la memoria secundaria (generalmente discos magnéticos) es *mucho más lenta* que la memoria principal (la memoria de acceso aleatorio, en sus siglas en inglés *RAM*). A la fecha que se realiza este documento, de acuerdo con *JEDEC*[1] los dispositivos más lentos de memoria tipo DDR[2] alcanzan una velocidad de acceso de 10 ns, mientras que los discos magnéticos más rápidos que hay en el mercado alcanzan velocidades de acceso de 2 ms (discos de 15,000 RPM). Esta cifra se puede calcular fácilmente viendo que 15,000 RPM = 250 r/s. Esto implica que el disco realiza una revolución en 4 ms. Podemos suponer que para leer un dato en promedio vamos a necesitar media vuelta del disco, dándonos el resultado de 2 ms.

[1] Joint Electron Device Engineering Council `http://jedec.org`. Este organismo se encarga de estandarizar interfaces electrónicas de alta integración en equipos de cómputo

[2] Double Data Rate RAM

Si comparamos la velocidad de la memoria más lenta (10 ns) contra la velocidad del disco magnético más rápido, la diferencia es dramática: entre 5 y 6 órdenes de magnitud. He aquí la razón por la cual, si realizamos muchos accesos a disco en alguna aplicación, debemos minimizar su número.

- *¿Qué aplicaciones realizan accesos intensos a disco?* El ejemplo clásico de una aplicación con acceso intenso a disco es un sistema manejador de bases de datos. En su mayoría, estos sistemas emplean estructuras de datos especiales que minimizan el impacto por los accesos a disco. Otra aplicación importante que realiza mucho acceso a disco son los sistemas de archivos utilizados en los sistemas operativos para representar los datos en una estructura jerárquica.

La estructura de datos conocida como árbol-B es un desarrollo enfocado a atacar el problema de la minimización de accesos a disco por una estructura de datos que implemente eficientemente las operaciones de un conjunto dinámico. Dichas operaciones son las mismas que en una estructura de datos lineal o de árbol:

- Inserción.
- Eliminación.
- Búsqueda.
- Extraer mínimo.
- Extraer máximo.
- Obtener sucesor.
- Obtener precedesor.

3.2.2. Definición de Árbol-B

Formalente un árbol-B se define como una estructura de datos jerárquica tipo árbol. Un árbol-B tiene las siguientes propiedades:

1. Un parámetro t que está directamente relacionado con el factor de crecimiento (el número de hijos que tiene cada nodo).
2. Cualquier nodo que no sea la raíz del árbol-B debe tener al menos t hijos y a lo más $2t$.

Cada nodo de un árbol-B debe tiene los siguientes atributos:

1. Un conjunto de valores que almacena. El número de valores que puede almacenar se encuentra entre $t - 1$ y $2t - 1$ (uno menos que los hijos); este conjunto se almacena tradicionalmente en un arreglo y dichos valores deben estar ordenados en orden ascendente. Adicionalmente, el árbol-B debe cumplir la propiedad de ser un árbol de búsqueda.

2. Un conjunto de apuntadores a los hijos. Al igual que el conjunto de valores que almacena, dicho conjunto se almacena generalmente en un arreglo.

Existe una relación entre los valores que almacena cada nodo del árbol-B y sus hijos: entre cada dos valores que almacena un nodo, tenemos un apuntador al hijo; así, se puede decir que cada valor tiene un hijo a su izquierda y uno a su derecha. Por lo tanto, si un árbol tiene $k \in [t-1, 2t-1]$ valores almacenados, debe tener $c = k + 1$ hijos. En la figura 3.2.2 se ejemplifica esquemáticamente este concepto.

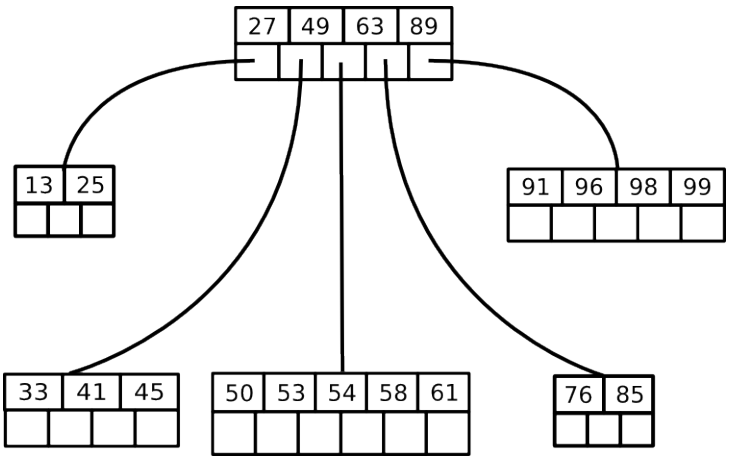

Figura 3.3: Representación gráfica de un árbol-B con parámetro $t = 3$.

El parámetro t

Si bien hemos definido el parámetro t como un número natural que se relaciona directamente con el factor de crecimiento del árbol-B, en realidad este parámetro está motivado por la estructura de un disco magnético. Debido a las propiedades mecánicas del disco, es común que al realizar una escritura sobre el mismo tengamos que escribir un bloque completo de n bytes, de ahí la motivación del parámetro t: generamos un árbol con un factor de crecimiento que esté apareado con el tamaño de bloque de un disco magnético; esto con la finalidad de minimizar el número de lecturas y escrituras a disco.

3.2.3. Consideraciones de implementación

Implementar un árbol-B puede ser relativamente complejo, por lo cual es importante definir de manera precisa las operaciones que realiza. En este caso utilizaremos el lenguaje `Java` para ejemplificar una implementación sencilla[3]

[3]Usamos `Java` porque es el lenguaje que se aprende en los cursos anteriores y es más claro y sencillo comparado con `C/C++` para explicar estructuras de datos.

Nuestra implementación estará constituida por dos clases: `BTree.java` y `BTreeNode.java`. La primera representa de manera global el árbol y la segunda representa de manera individual cada nodo del mismo. El esqueleto de la clase `BTree.java` es como sigue:

```
1  public class BTree {
2      // La raiz del arbol-B
3      protected BTreeNode root;
4      // El parametro t.
5      protected int t;
6
7      /**
8       * Crea un nuevo arbol-B vacio.
9       * @param t Parametro t
10      * relacionado con el factor
11      * de crecimiento.
12      */
13     public BTree(int t){
14         this.t = t;
15         root = new BTreeNode(t);
16     }
17
18     // Operaciones por implementar
19 }
```

Figura 3.4: La clase `BTree`.

Para implementar los contenedores del árbol-B utilizaremos arreglos: uno para los elementos que almacena la instancia del nodo (los elementos deberán implementar `Comparable` para poderse ordenar[4]), y otro para referenciar los nodos hijos. Dichos arreglos tendrán tamaño fijo ($2t - 1$ para el arreglo de elementos y $2t$ para el arreglo de referencias a hijos), por lo que necesitaremos dos enteros que nos indiquen la población de cada uno de estos arreglos. De este modo, la definición de un nodo del árbol-B queda definida como:

```
1  public class BTreeNode {
2      private Comparable[] keys;
3      private BTreeNode[] children;
4      private int keyCount;
5      private boolean leaf;
6
7      /**
8       * Crea un nuevo nodo de arbol-B
9       * con el parametro t especificado.
10      */
11     public BTreeNode(int t){
12         keys = new Comparable[2*t-1];
13         children = new BTreeNode[2*t];
14         keyCount = 0;
15         leaf = true;
```

[4]Desafortunadamente no se puede usar *generics* ya que no es posible instanciar arreglos genéricos en Java hasta el momento.

```
16      }
17
18      /** Devuelve el contenido almacenado
19       *  en el conjunto de valores de este nodo.
20       *
21       * @param i Indice del elemento a obtener.
22       * @return El elemento.
23       * @throws IndexOutOfBoundsException en caso
24       * de que se exceda el indice a la
25       * cantidad elementos o sea menor que cero.
26       * almacenados en este nodo.
27       */
28      public Comparable getKey(int i)
29           throws IndexOutOfBoundsException {
30          if(i < 0 || i >= keyCount)
31              throw new IndexOutOfBoundsException(String.valueOf(i));
32          else
33              return keys[i];
34      }
35
36      /**
37       * Establece una entrada en el
38       * conjunto de elementos de este nodo.
39       * @param i Indice donde colocar el elemento.
40       * @param k Elemento a colocar.
41       * @throws IndexOutOfBoundsException En caso
42       * de que el indice especificado sea invalido.
43       */
44      public void setKey(int i, Comparable k)
45           throws IndexOutOfBoundsException {
46          if(i < 0 || i >= keyCount)
47              throw new IndexOutOfBoundsException(String.valueOf(i));
48          else
49              keys[i] = k;
50      }
51
52      /**
53       * Establece una entrada del
54       * conjunto de hijos de este nodo.
55       * @param i Indice donde colocar el nuevo hijo.
56       * @param n Hijo a colocar.
57       * @throws IndexOutOfBoundsException En caso
58       * de que el indice especificado sea invalido.
59       * @throws IllegalArgumentException En caso
60       * de que el nodo dado sea nulo.
61       */
62      public void setChild(int i, BTreeNode n)
63           throws IndexOutOfBoundsException {
64          if(n == null)
65              throw new IllegalArgumentException("El nodo dado no
                    debe ser nulo");
66          if(i < 0 || i >= keyCount + 1)
67              throw new IndexOutOfBoundsException(String.valueOf(i));
68          leaf = false;
69          children[i] = n;
70      }
71
```

```
72      /**
73       * Evalua si este nodo es hoja
74       * @return <code>true</code> en caso
75       * de que el nodo sea hoja.
76       */
77      public boolean isLeaf(){
78          return leaf;
79      }
80
81      /**
82       * Calcula el numero de elementos
83       * almacenados en este nodo.
84       * @return numero de elementos.
85       */
86      public int getKeyCount(){
87          return keyCount;
88      }
89
90      /**
91       * Devuelve una referencia a i-esimo
92       * hijo de este arbol.
93       * @param i Indice del hijo a obtener.
94       * @return El hijo especificado.
95       * @throws IndexOutOfBoundsException en caso
96       * de que el indice especificado exceda
97       * la cantidad de hijos o sea menor que cero.
98       * @throws IllegalArgumentException en caso
99       * de intentar obtener hijos de un nodo hoja.
100      */
101     public BTreeNode getChild(int i)
102          throws IndexOutOfBoundsException {
103         if(leaf)
104             throw new IllegalArgumentException("No se puede obtener
                    hijos de un nodo hoja");
105         if(i < 0 || i >= keyCount + 1)
106             throw new IndexOutOfBoundsException(String.valueOf(i));
107
108         return children[i];
109     }
110
111     /**
112      * Incrementa el conteo
113      * de elementos de este nodo en 1.
114      */
115     public void incrementKeyCount(){
116         ++keyCount;
117     }
118
119      /**
120      * Decrementa el conteo de
121      * elementos de este nodo en 1.
122      */
123     public void decrementKeyCount(){
124         --keyCount;
125     }
126
127     /**
```

```
128         * Establece el numero de
129         * elementos que almacena este nodo.
130         */
131        public void setKeyCount(int kc){
132            keyCount = kc;
133        }
134    }
```

3.2.4. Operaciones del árbol-B

Búsqueda en un árbol-B

Para realizar la operación de búsqueda necesitamos recibir dos parámetros: el objeto a buscar y un contenedor donde podamos colocar el nodo del árbol-B que contiene al elemento que estamos buscando. En caso de que la búsqueda sea satisfactoria, devolvemos el índice donde se encuentra el elemento especificado y actualizamos la referencia en el arreglo dado para poder devolver el nodo contenedor.

Una vez definida la estructura de la llamada a método, el algoritmo funciona de la siguiente manera; si estamos buscando una llave k, el procedimiento se podría resumir como sigue:

1. Realizamos una búsqueda local en el conjunto de valores de cada nodo y localizamos la posición donde debería estar el valor que estamos buscando, esto es, dado un nodo n encontrar el índice i tal que $valor(n, i-1) <= k <= valor(n, i+1)k$. Esto siempre debe suceder en cada nodo ya que el árbol-B es un árbol de búsqueda.

2. En caso de que el valor de la entrada i-ésima del arreglo de valores coincida con k, hemos encontrado el valor; actualizamos la referencia del nodo donde lo encontramos y devolvemos i.

3. En caso contrario, si el nodo donde estamos colocados es una hoja del árbol, significa que el valor no pudo ser encontrado, por lo que establecemos la referencia del nodo como nula y devolvemos un índice sentinela -1.

4. Finalmente, si el nodo donde estamos colocados no es hoja, aplicamos recursivamente este procedimiento de búsqueda en el i-ésimo hijo de este nodo.

5. Empezamos el procedimiento de búsqueda a partir de la raíz y vamos avanzando a profundidad como se explica anteriormente.

```
1      // Implementacion recursiva de la busqueda.
2      private int search(Comparable key,
3                         BTreeNode[] node,
4                         BTreeNode startAt){
5          // Iteramos hasta encontrar
```

```
6           // la posicion donde deberia estar
7           // el elemento que estamos
8           // buscando. (Recordar que estamos
9           // usando un arbol de busqueda).
10          int i = 0;
11          int numKeys = startAt.getKeyCount();
12
13          // El metodo compareTo devuelve un
14          // valor menor que cero si la instancia
15          // que llama es menor a la instancia
16          // pasada como parametro, cero en caso de
17          // ser iguales y un valor mayor que cero
18          // en caso de que la instancia que llama
19          // sea mayor a la que se pasa como parametro.
20          while(i < numKeys && key.compareTo(startAt.getKey(i)) > 0){
21              ++i;
22          }
23
24          // Si el indice que encontramos
25          // es igual al elemento de busqueda,
26          // hemos terminado.
27          if(i < numKeys && startAt.getKey(i).compareTo(key) == 0){
28                  // Guardamos el nodo contenedor.
29                  node[0] = startAt;
30                  // Devolvemos el indice.
31                  return i;
32          }
33
34          // En caso de que el nodo sea hoja significa que
35          // ya no podemos avanzar y por tanto no encontramos
36          // el elemento.
37          if(startAt.isLeaf()){
38              node[0] = null;  // No hay contenedor.
39              return -1;       // Indice sentinela.
40          } else {
41              // En caso de que no hayamos encontrado
42              // el nodo buscado en esta instancia, realizamos
43              // busqueda recursiva a partir del hijo con el indice i
                    .
44              return search(key, node, startAt.getChild(i));
45          }
46      }
```

Inserción de elementos en un árbol-B

El proceso de insertar nodos en un árbol-B se complica más en comparación con el de búsqueda por el hecho de que insertar un nodo puede requerir reestructurar el árbol-B para mantener sus propiedades. Este procedimiento, a diferencia de los métodos ya conocidos para insertar nodos en árboles binarios de búsqueda no hace que el árbol crezca hacia abajo (a través de las hojas), sino crece por arriba (la raíz se modifica para mantener las propiedades). Debido a que al insertar es necesario reestructurar el árbol para ciertos casos, primero presentamos una operación importante de reestructuración: el método `splitChild`,

el cual recibe un nodo lleno del árbol-B (con $2t-1$ valores) y lo subvidide en 3 nodos, uno con un valor, y le cuelga dos hijos con $t-1$ valores.

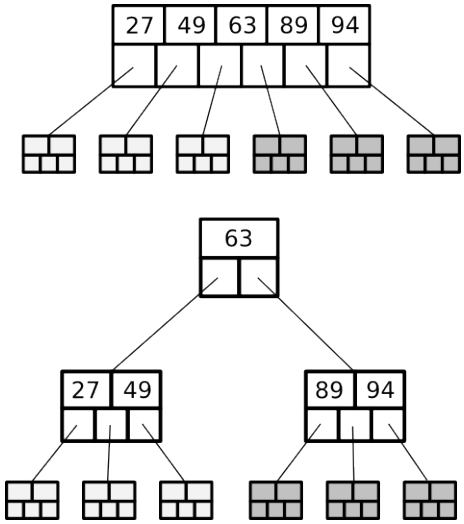

Figura 3.5: División de un nodo lleno en un árbol-B. Aquí estamos suponiendo $t = 3$.

Este método supone que el nodo padre dado como argumento tiene espacio para almacenar una llave y recibe 2 argumentos, el padre del nodo a dividir, y el índice donde está colocado el nodo a dividir; el procedimiento funciona como sigue:

1. Creamos un nuevo nodo con $t-1$ elementos.
2. Copiamos los valores del i-ésimo hijo del nodo dado a partir del índice t hasta $2t-1$, es decir, copiamos la segunda mitad del nodo a dividir a un nuevo nodo exluyendo el nodo intermedio.
3. En caso de que el nodo a dividir no sea hoja, realizamos el mismo procedimiento de copiado a los hijos del nuevo nodo creado.
4. Recorremos los valores e hijos del nodo padre a la derecha a partir del índice i, esto para insertar el valor de enmedio del nodo que partimos.
5. Insertamos el $t-1$-ésimo valor del nodo dividido en el padre colocándolo en el índice i.
6. "Recortamos" el nodo original a $t-1$ elementos.
7. Establecemos como el hijo $i+1$-ésimo del padre al nuevo nodo recién creado.

Una implementación del método `splitChild` podría ser:

```
protected void splitChild(BTreeNode p, int i){
        BTreeNode c = p.getChild(i);
        // Creamos el nuevo nodo.
        BTreeNode nn = new BTreeNode(t);
        nn.setKeyCount(t-1);
        // Copiamos los valores del nodo a partir desde t a 2t-1.
        for(int j = 0; j < t-1; ++j){
                nn.setKey(j, c.getKey(j+t));
        }
        // En caso de que el nodo no sea hoja, copiamos los hijos
            tambien.
        if(!c.isLeaf()){
                for(int j = 0; j < t; ++j){
                nn.setChild(j, c.getChild(j+t));
                }
        }
        // Recorremos los valores del padre a partir de la mitad
        // un lugar a la derecha
        p.incrementKeyCount();
        for(int j = p.getKeyCount() - 2; j >= i; --j){
                p.setKey(j+1, p.getKey(j));
        }
        // Agregamos el t-esimo hijo del nodo partido al padre.
        p.setKey(i, c.getKey(t-1));
        // Recortamos el nodo hijo a t-1 valores.
        c.setKeyCount(t-1);
        // Recorremos los hijos del padre a la derecha.
        for(int j = p.getKeyCount() - 1; j >= i + 1; --j){
                p.setChild(j+1, p.getChild(j));
        }
        // Insertamos el nuevo nodo como el i+1 esimo nodo del
            padre.
        p.setChild(i+1, nn);
}
```

3.2.5. Métodos *max* y *min*

Para extraer el valor máximo y mínimo del árbol-B se hace exactamente del mismo modo que cualquier árbol de búsqueda: para el valor mínimo, nos recorremos a la hoja más a la izquierda del árbol y devolvemos el valor con índice cero de ese nodo. Recíprocamente. para calcular el máximo, nos dirijimos a la hoja más a la derecha del árbol y devolvemos el valor con el índice máximo de ese nodo.

```
/** Devuelve el elemento minimo almacenado en el arbol-B.
*
* @return El elemento minimo.
*/
public Comparable min(){
        BTreeNode a = root;
        while(!a.isLeaf()){
        a = a.getChild(0);
        }
        return a.getKey(0);
```

```
11  }
12
13  /** Devuelve el elemento maximo almacenado en el arbol-B.
14  *
15  * @return El elemento maximo.
16  */
17
18  public Comparable max(){
19          BTreeNode a = root;
20          while(!a.isLeaf()){
21                  a = a.getChild(a.getKeyCount());
22          }
23          return a.getKey(a.getKeyCount()-1);
24  }
```

Operaciones de sucesor y predecesor

Funcionan de la misma manera que cualquier árbol de búsqueda; lo que hay que hacer es realizar un recorrido inorden del árbol-B. Un recorrido inorden ascendente de un nodo n se realiza de la siguiente manera:

1. Si el nodo n es hoja simplemente devolvemos los valores que almacena en el orden natural (ascendentemente sobre su índice).
2. En caso de no ser hoja, iteramos desde $i = 0$ hasta $i = numeroDeValores(n)$:
   a) Aplicamos recursivamente este procedimiento al hijo con índice i.
   b) Devolvemos el nodo i.
   c) Aplicamos recursivamente este procedimiento al hijo con índice $i+1$.

Para el caso de un recorrido descendente, simplemente basta con iterar desde $i = numeroDeValores(n)$ decrecientemente hasta $i = 0$ y avanzar primero al hijo $i + 1$ y luego al i en las llamadas recursivas.

Operación de eliminación de nodos

Esta es la operación más compleja a implementar del árbol-B y por cuestiones de claridad es más sencillo explicar cómo funciona a describir a detalle el código (lo cual tomaría varias páginas). Lo que complica eliminar nodos de un árbol-B es la condición del número mínimo de nodos $(t-1)$ en cualquier nodo que no sea la raíz) y mantenerlo como árbol de búsqueda. Por lo tanto, necesitamos realizar diversas operaciones para balancear el ábol en caso de eliminar una entrada de un nodo que hace que quede en déficit.

Si queremos eliminar un valor k del árbol, el procedimiento para realizarlo iría como sigue:

1. Buscamos el valor k en el árbol. Esto nos devuelve un nodo contenedor y un índice.

2. Si el nodo contenedor es una hoja:

   a) El caso más sencillo es cuando vamos a eliminar la entrada de una hoja con un número de valores $n > t - 1$. En este caso la única operación que hay que realizar es la eliminación.

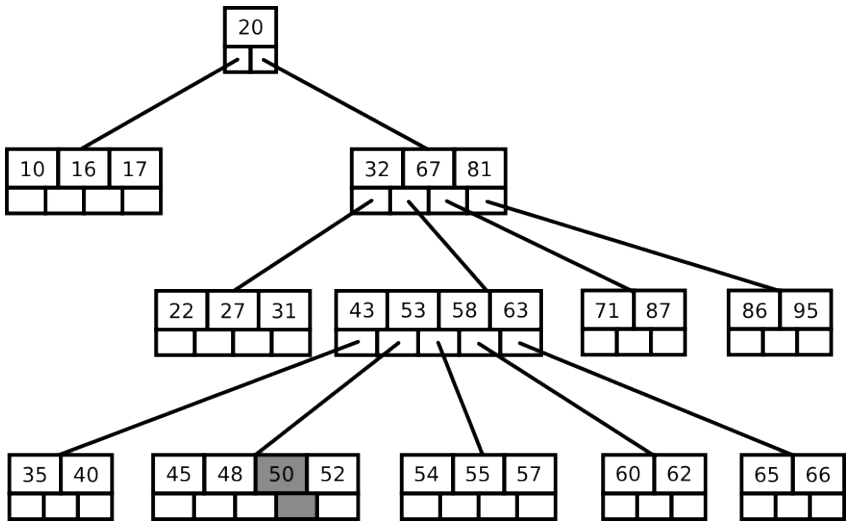

Figura 3.6: Eliminar un nodo de una hoja en un árbol-B (el árbol no se ha representado completo, se deben llenar todos los niveles).

   b) En caso de que borremos el elemento de una hoja con un número de valores $n = t - 1$ hay que rebalancear; para esto se dan tres posibles casos:

      1) Algún hermano tiene longitud $n > t - 1$ y el padre longitud $n > t - 1$. En este caso la solución es sustituir el valor a eliminar por el padre de este elemento. Posteriormente colocar en el lugar que tenía el padre el máximo o mínimo para el hermano a la izquierda o derecha respectivamente. Esto hace que se disminuya en uno el conteo de valores de algún hermano y mantiene la propiedad de árbol-B.

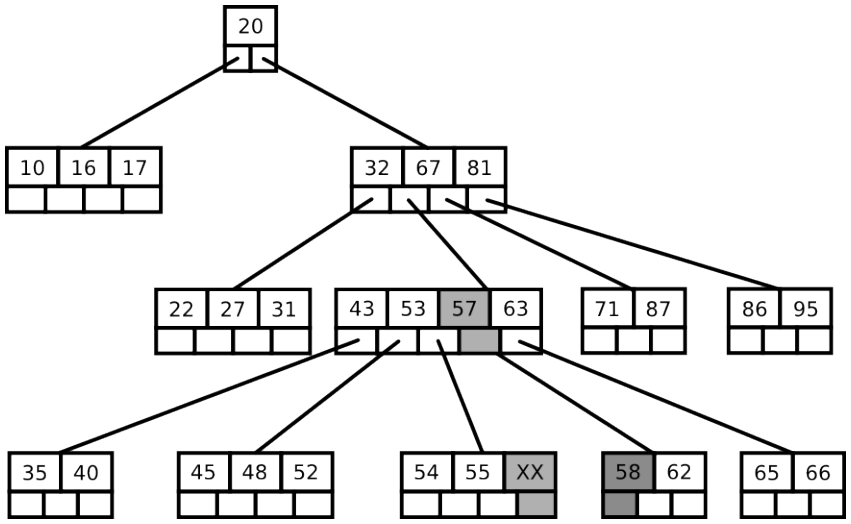

Figura 3.7: Caso 2.b.1. Eliminar el nodo con valor 60 genera este reacomodo (ver figura anterior)

2) Ambos hermanos tienen longitud $t-1$ y el padre longitud $n > t-1$. En esta situación, lo que hay que hacer es sustituir el nodo a eliminar por el padre y combinarlo con alguno de sus hermanos, dando así un árbol-B válido.

3) Ambos hermanos tienen longitud $t-1$ y el padre longitud $t-1$. Aplicamos el mismo procedimiento de arriba y posteriormente hay que balancear el árbol para que el padre tenga $t-1$ nodos de nuevo.
Para el análisis de estos casos, imaginemos un escenario inicial como el que se presenta en la figura 3.9

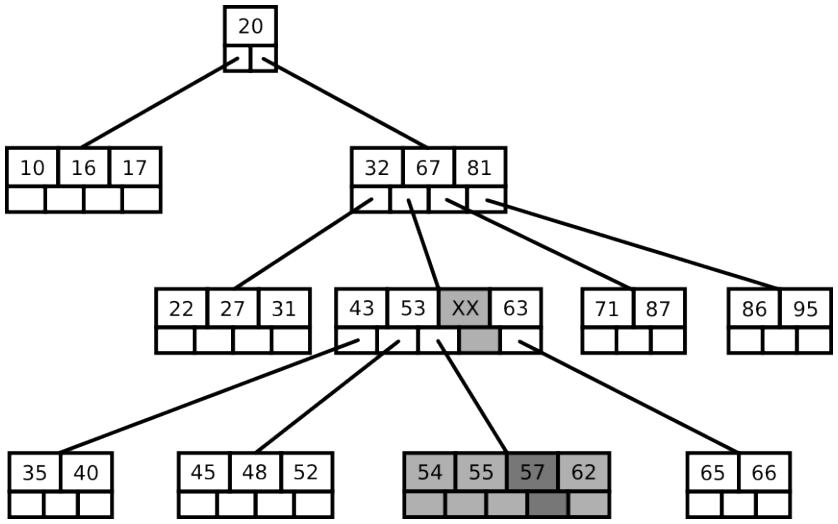

Figura 3.8: Caso 2.b.2. Eliminar el nodo con valor 58 genera este reacomodo (ver figura anterior)

De nuevo pueden darse dos casos:

a' Algún hermano del padre tiene longitud mayor a $t-1$. Tomamos al nodo padre y lo sustituimos por el nodo que queremos eliminar, sus hijos serán los hijos del nodo que eliminamos. Posteriormente tomamos prestado un hijo del hermano seleccionado (máximo o mínimo para hermano izquierdo y derecho respectivamente) y lo convertimos en el nuevo padre. Esto causa que un hijo del hermano quede colgando, y que al padre que hicimos descender le falte un hijo, por lo que asignamos este hijo al padre para compensar su déficit.

b' Si ningún hermano tiene longitud mayor a $t-1$ aplicamos la técnica de combinarnos con un hermano y así recursivamente hacia arriba.

3. En caso de querer borrar un nodo no hoja, existen dos posibles casos:

   a) El hijo de la izquierda o derecha contiene más de $t-1$ nodos. Lo que debemos hacer es seleccionar el hijo con un número mayor a $t-1$ nodos y tomar el máximo o mínimo para el hijo izquierdo y derecho respectivamente; posteriormente sustituimos el nodo que queremos eliminar con el hijo seleccionado, y así reducimos en una unidad el conteo de nodos del hijo que seleccionamos.

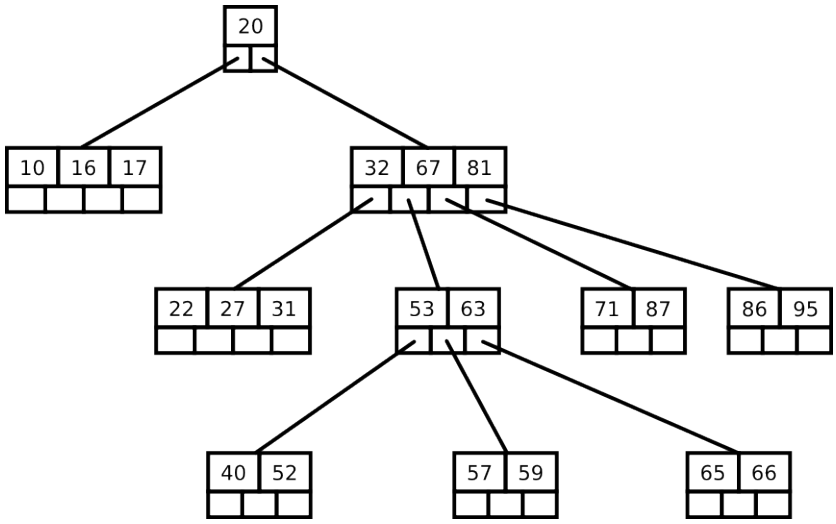

Figura 3.9: Escenario inicial para el caso 3.

b) Los hijos contienen exactamente $t - 1$ nodos. Realizamos el mismo procedimiento mencionado anteriormente con cualquier hijo y esto causará que a ese hijo le haga falta un nodo dando así un caso como los mencionados arriba. Hay que aplicar balanceo recursivo hacia arriba.

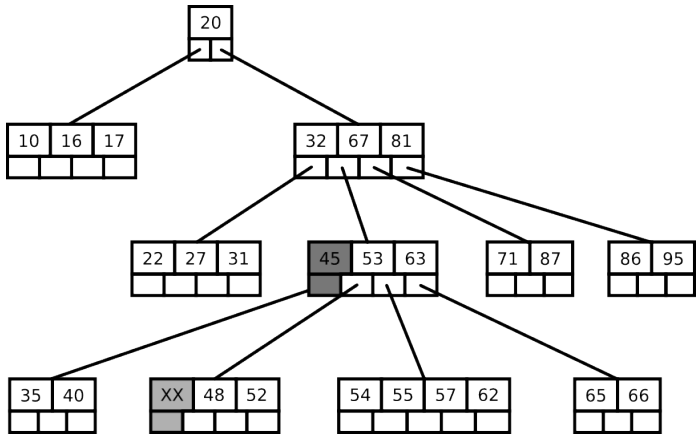

Figura 3.10: Caso 3.a. Eliminar el nodo con valor 43 genera este reacomodo. (ver figura del caso 2.b.2)

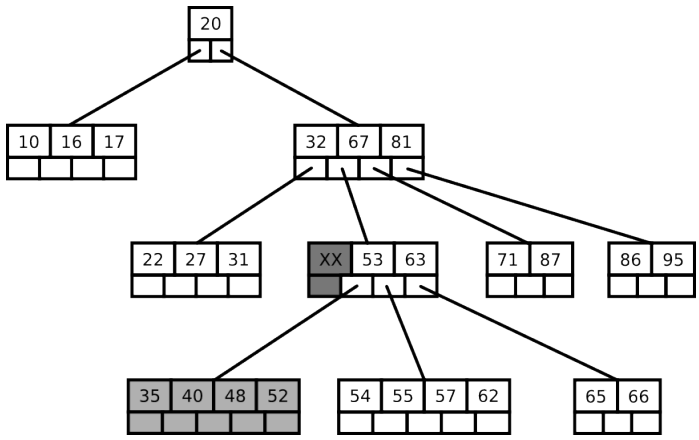

Figura 3.11: Caso 3.b. Eliminar el nodo con valor 45 genera este reacomodo (ver figura anterior)

Capítulo 4

Buenas Prácticas de Programación

Este capítulo se dedica a presentar un pequeño repertorio de prácticas sencillas de programación que se heredan de los conocimientos y desarrollos en ingeniería de software. La ingenería de software es una disciplina de la ciencia de la computación que se enfoca en la aplicación de metodologías sistemáticas, cuantificables y operativas con respecto al desarollo y mantenimiento del software. El análisis de los diversos patrones, prácticas y metodologías que maximizan la productividad en los procesos de desarrollo del software es el objetivo principal de esta disciplina.

Si bien no se estudia a profundidad y a completa amplitud toda la gama de metodologías en la ingeniería de software, se busca ofrecer al estudiante algunas metodologías muy utilizadas en el desarrollo de proyectos de programación de escala pequeña a mediana con la finalidad de mejorar las prácticas de desarrollo a lo largo de la licenciatura hacia los cursos avanzados.[5]

4.1. Patrones de Diseño

En cualquier proceso de desarrollo o diseño tecnológico es común que existan ciertas tareas o procedimientos que se relacionen por medio de algún tipo de patrón. Por ejemplo, en el diseño orientado a objetos es común diseñar clases con ciertos atributos de acceso privado y asignarles métodos para establecer y obtener atributos (*get/set*). Así como el ejemplo anterior, existe una serie de prácticas y patrones de uso muy común en la programación orientada a objetos.

Los patrones de diseño son un conjunto de metodologías y conceptos para mejorar las prácticas de desarrollo y mantenimiento de software. Si bien no se estudian todos en este apartado, se especifican los más comunes para proyectos de pequeña a mediana escala.

4.1.1. Definición

Un patrón de diseño es una generalización de un diseño aplicado a un problema común de desarrollo de software. Los patrones de diseño se enfocan más en la etapa de diseño que en la de implementación, pero buscan minimizar los problemas derivados de un mal diseño y ofrecen una visión simplificada de ciertos problemas.

En general, un patrón de diseño se debe pensar como una plantilla de diseño, no como una implementación específica. Sólo define ciertos elementos de diseño y algunas propiedades que cumple; el desarrollador debe posteriormente implementar el comportamiendo adecuado al problema a resolver manteniendo los lineamientos de diseño planteados por el patrón.

Los patrones de diseño tienen las siguientes características:

1. Un **nombre**. Es muy útil establecer un nombre claro y que trate de explicar someramente la idea del patrón.

2. Un **problema a resolver**. De manera general, especifica *cuándo* aplicar el patrón a un tipo de problema dado.

3. La **solución**. Describe los elementos de diseño, relaciones y propiedades que tiene el patrón.

4. **Consecuencias**. Aplicar un patrón de diseño ofrece ventajas y desventajas en una diversa gama de parámetros (extensibilidad, mantenibilidad, claridad del código, eficiencia, etcétera). Es importante conocer las consecuencias positivas y negativas al momento de aplicar un patrón para poder ponderar diversas posibilidades de diseño y decidir por la opción más apta al problema.

Si bien los patrones de diseño se pueden aplicar a diversos paradigmas de programación, la mayoría de ellos están pensados para ser abstraídos y aplicados en un entorno orientado a objetos.

4.2. Patrones de diseño de uso común

Existe una amplia gama de patrones de diseño descritos en diversas fuentes bibliográficas que atienden a diversas necesidades. En este apartado se enumeran y describen cinco patrones de diseño de uso muy común y de sencilla comprensión que serán de utilidad para el estudiante.

Los nombres de cada patrón se describen en inglés ya que su nomenclatura se ha estandarizado y esto ayuda a identificarlos fácilmente en la literatura disponible.

4.2.1. *Factory method* (Método fábrica)

Motivación

Existen diversas situaciones en las que utilizamos herencia para hacer más específica la implementación de una clase. Empleamos una clase abstracta o definimos una interfaz que modele de manera general un problema y empezamos a utilizar herencia para clases que problemas más específicos. Por lo tanto, para hacer uso del polimorfismo, no queremos utilizar el constructor de una clase concreta que herede de una abstracta (o implemente una interfaz), sino que queremos mantener el tipo más general definido por la interfaz o superclase, por lo que debemos ocultar el constructor y utilizar una forma alternativa de instanciación de clases. Esto se implementa por medio de un método que genera la clase que necesitamos; es decir, decide cuál de las subclases necesitamos, dependiendo de cierto parámetro o estado, y devuelve la instancia necesaria.

Ejemplo

Supongamos que queremos implementar una clase que sirve para cargar imágenes digitales en una estructura de datos. Esta clase puede cargar diversos formatos de imagen y la forma de proporcionar esta funcionalidad es implementando una interfaz que unifique el comportamiento de las clases que implementan la carga para cada formato de imagen. Cada clase implementa la carga para un formato de imagen distinto y el procedimiento para generar clases nuevas está encapsulado en una clase *fábrica* que va generando los objetos sobre demanda. Esto evita la necesidad de conocer el tipo de imagen a cargar en tiempo de compilación y genera un sistema con un comportamiento más dinámico.

La forma más clara de ver su funcionamiento es a través de un código de ejemplo:

```
1  // Interfaz para lectores de imagenes.
2  public interface ImageReader {
3          // Decodificar la imagen y cargarla
4          // a memoria.
5          public DecodedImage getDecodedImage();
6  }
7
8  // Fabrica de lectores de imagenes.
9  // Genera la subclase necesaria
10 // dependiendo del formato de la misma.
11 public class ImageReaderFactory
12 {
13         // Creamos clases anidadas para
14         // cada tipo de imagen a cargar.
15         // Esto oculta la implementacion al
16         // cliente.
17
18         // Lector de imagenes GIF
19         static class GifReader implements ImageReader
20         {
21                 private DecodedImage decodedImage;
22                 // Se oculta el constructor para
```

```
23                  // evitar ser instanciado por el
24                  // cliente.
25                  protected GifReader( InputStream in )
26                  {
27                          // Analiza la imagen, revisa que tenga el
                                tipo correcto
28                          // y carga la imagen en memoria.
29                  }
30                  public DecodedImage getDecodedImage()
31                  {
32                          return decodedImage;
33                  }
34          }
35          // Lector para JPEG.
36          static class JpegReader implements ImageReader
37          {
38          // etc...
39          }
40
41      // Metodo fabrica (factory method).
42      // Genera una nueva instancia del lector
43      // de imagenes analizando el tipo de
44      // La imagen.
45      public static ImageReader getImageReader(InputStream is)
46      {
47          // Inferimos el tipo de la imagen.
48          int imageType = figureOutImageType( is );
49
50          // Decidimos que clase crear
51          // con base en el tipo de la imagen.
52          switch( imageType )
53          {
54              case ImageReaderFactory.GIF:
55                  return new GifReader( is );
56              case ImageReaderFactory.JPEG:
57                  return new JpegReader( is );
58              default:
59                  String msg = "Unsupported image type";
60                  throw new IllegalArgumentException(msg);
61          }
62      }
63  }
```

En el código se puede ver que el proceso de generación de clases está encapsulado en la clase *fábrica*. La implementación de los diversos lectores para cada tipo de imagen quedan ocultos al cliente y se hace uso de polimorfismo para devolver el lector de imagen. El proceso de instanciación se hace por medio de un método fábrica y no por el constructor, dicho método decide qué clase necesitamos y llama al constructor adecuado para devolver la clase que carga las imágenes *ad-hoc* al formato de entrada.

Consecuencias

1. El patrón elimina la necesidad de enlazar clases que implementan una funcionalidad específica dentro del código. El cliente sólo debe preocuparse

por llamar al método fábrica y este devolverá la implementación necesaria con el tipo de la interfaz que el cliente necesita.

2. Una desventaja a consecuencia de este patrón es que puede producir demasiadas clases derivadas de diversos escenarios de implementación y que en algunos casos son clases muy similares, que sólo varían por pocos parámetros.

3. En casos de herencias de clases paralelas (es decir, dos herencias de clases que mantienen relaciones tipo consumidor-productor), se puede usar el patrón de fábrica en ambas y encapsular la implementación de cada una de las herencias y además reduce el acoplamiento de ellas.

En general las clases que sirven como fábricas deben tener una sola instancia a lo largo de su ejecución ya que siempre tendrá el mismo comportamiento sin importar la instancia que se utilice, por lo tanto podemos utilizar el patrón de diseño *Singleton* para asegurar que sólo exista una instancia de cada fábrica.

4.2.2. Patrón *singleton*

Motivación

Como se menciona en el patrón anterior, en algunas circunstancias es recomendable mantener una sola instancia de una clase dada. La forma de resolver esta problemática es diseñando una clase que pueda dar seguimiento y mantener *una sola instancia* de ella.

Ejemplos

Existen diversos ejemplos donde es necesario tener una sola instancia de una clase:

1. **Gestores de ventanas:** Un gestor de ventanas es un componente de software que se usa para manejar sistemas de interfaz gráfica basados en ventanas. El gestor se encarga de administrar la memoria y recursos necesarios para el manejo de ventanas gráficas (incluyendo iconos, memoria de video, entre otros). En general sólo vamos a tener un gestor de ventanas para un sistema operativo, por lo que podemos aplicar el patrón *singleton* para asegurar esta condición.

2. **Sistema de archivos:** Si modeláramos un sistema de archivos con orientación a objetos, no sería sensato tener diversas instancias de la clase, ya que todo equipo de cómputo tiene un solo sistema de archivos (no particiones ni tipos de sistemas de archivos, sino la jerarquía completa).

Implementación

A diferencia de la fábrica, el singleton es un patrón más sistemático y casi siempre mantiene la misma fórmula para desarrollarlo:

```
1  // Clase con patron singleton.
2  public class Singleton {
3      // La instancia del singleton.
4      private static Singleton instance = null;
5      // Ocultamos el constructor para
6      // que el cliente no pueda instanciar
7      // la clase.
8      protected Singleton(){
9          // Codigo de constructor aqui.
10     }
11
12     // Metodo que se encarga de manejar
13     // la unicidad del singleton.
14     public static Singleton getInstance(){
15         if(instance == null){
16             instance = new Singleton();
17             return instance;
18         } else
19             return instance;
20     }
21
22 }
```

Consecuencias

1. Acceso restringido a una sola instancia de la clase.
2. El patrón *singleton* es una mejora a las variables globales, reduce la *contaminación* del espacio de nombres evitando enlazar identificadores libres en el espacio global.
3. Permite un número variable de instancias controladas. Es posible modificar este código de manera sencilla para en lugar de tener control de una sola instancia, poder tener una colección acotada de instancias del singleton y poder controlar su creación. Esta variación sólo se refleja en el método que provee de acceso a las instancias controladas.
4. Ventaja sobre operaciones de clase (métodos y atributos estáticos). Ofrece más flexibilidad para poder implementar control sobre más de una instancia y además ayuda al polimorfismo y herencia, ya que podemos heredar de un singleton.

4.2.3. Patrón *Adapter*

Motivación

También conocido como *wrapper*, es un patrón estructural muy utilizado el cual consiste en desarrollar una clase que convierte la interfaz de una clase a

otra interfaz que necesita el cliente. Es una práctica muy común de reutilización de código.

Ejemplo

Un ejemplo ya conocido del uso de un adaptador es usar una lista enlazada para implementar la interfaz de una pila. La implementación del patrón de diseño quedaría como sigue:

```
1  /**
2   * Uso del patron de adaptador
3   * para exponer la funcionalidad de
4   * una pila por medio de una lista.
5   */
6
7  /** La interfaz de una pila */
8  interface Stack<T>
9  {
10   void push (T o);
11   T pop ();
12   T top ();
13 }
14
15 /* Lista doblemente enlazada.
16  * La clase que vamos a adaptar.
17  */
18 class DList<T>
19 {
20   public void insert (DNode pos, T o) { ... }
21   public void remove (DNode pos) { ... }
22
23   public void insertHead (T o) { ... }
24   public void insertTail (T o) { ... }
25
26   public T removeHead () { ... }
27   public T removeTail () { ... }
28
29   public T getHead () { ... }
30   public T getTail () { ... }
31 }
32
33 /* Adaptamos la lista enlazada
34  * a la interfaz de una pila.
35  */
36 class DListImpStack<T> extends DList<T> implements Stack<T>
37 {
38   public void push (T o) {
39     insertTail (o);
40   }
41
42   public T pop () {
43     return removeTail ();
44   }
45
46   public T top () {
47     return getTail ();
```

```
48     }
49 }
```

Consecuencias

Un adaptador:

1. Permite modificar el comportamiento de la clase a adaptar ya que la clase adaptador hereda de la adaptada. Esto nos ofrece más flexibilidad para desarrollar una clase que se adapte y acople mejor a nuestras necesidades

2. Permite trabajar con diversas clases a adaptar gracias a la herencia, es decir, si adaptamos una clase y esta tiene subclases, estas últimas también serán adaptadas por nuestro adaptador. Análogamente agrega o modifica funcionalidad de las subclases como se menciona en el punto anterior.

3. El trabajo necesario para adaptar una clase a una interfaz distinta depende mucho de la situación. En el ejemplo anterior se hizo de manera directa "traduciendo" los nombres de los métodos de la lista a la interfaz de pila que necesitamos, sin embargo, en otros casos es necesario implementar funcionalidad o algoritmos que nos sirvan para convertir la interfaz o las estructuras de datos que pueda manejar la clase cliente.

4.2.4. Patrón *Composite*

Motivación

El objetivo de este patrón es poder componer objetos en estructuras de tipo árbol para representar jerarquías. Este patrón permite a los clientes tratar objetos individuales y composición de los mismos de manera uniforme.

Ejemplo

Un ejemplo muy común que ilustra el funcionamiento del patrón de composición es una biblioteca para dibujar gráficos que soporte composición de elementos. Los elementos primitivos serían líneas, elipses, rectángulos, mientras que los compuestos podrían ser combinaciones de ellos para armar figuras más sofisticadas.

```
1  /** El compontente
2   *  para dibujar graficos.
3   */
4  interface Graphic {
5
6      /** Dibuja el grafico */
7      public void print();
8
9  }
10
11 /** Grafico compuesto. */
```

```
12  class CompositeGraphic implements Graphic {
13
14      /* Se implementa en base a
15       * una coleccion de
16       * graficos primitivos */
17      private List<Graphic> mChildGraphics = new ArrayList<Graphic>()
            ;
18
19      /* Imprime el grafico compuesto
20       * imprimiendo cada primitivo. */
21      public void print() {
22          for (Graphic graphic : mChildGraphics) {
23              graphic.print();
24          }
25      }
26
27      /* Agregar graficos a este
28       * grafico compuesto (se pueden
29       * agregar graficos compuestos
30       * tambien).
31      public void add(Graphic graphic) {
32          mChildGraphics.add(graphic);
33      }
34
35      /* Elimina graficos de esta
36       * coleccion */
37      public void remove(Graphic graphic) {
38          mChildGraphics.remove(graphic);
39      }
40
41  }
42
43  /** El primitivo de elipse */
44  class Ellipse implements Graphic {
45      public void print() {
46          // Codigo para imprimir elipses
47      }
48
49  }
50
51  /** Codigo cliente */
52  public class Program {
53      public static void main(String[] args) {
54          /* Creamos 4 elipses */
55          Ellipse ellipse1 = new Ellipse();
56          Ellipse ellipse2 = new Ellipse();
57          Ellipse ellipse3 = new Ellipse();
58          Ellipse ellipse4 = new Ellipse();
59
60          /* Creamos 3 graficos compuestos */
61          CompositeGraphic graphic = new CompositeGraphic();
62          CompositeGraphic graphic1 = new CompositeGraphic();
63          CompositeGraphic graphic2 = new CompositeGraphic();
64
65          /* Componemos */
66          graphic1.add(ellipse1);
67          graphic1.add(ellipse2);
```

```
68          graphic1.add(ellipse3);
69
70          graphic2.add(ellipse4);
71
72          /* Armamos el grafico principal */
73          graphic.add(graphic1);
74          graphic.add(graphic2);
75
76          /* Dibujamos el grafico */
77          graphic.print();
78      }
79  }
```

Consecuencias

El patrón de composición:

1. Define jerarquías constituidas por objetos primitivos y objetos compuestos. Los objetos primitivos se pueden componer para construir objetos más complejos los cuales pueden ser compuestos en nuevos objetos y así recursivamente. Tanto los objetos primitivos como los compuestos mantienen coherencia de tipos, característica muy útil para el cliente.

2. Simplifica el trabajo en el cliente ya que puede manipular objetos primitivos y compuestos de manera uniforme. Los clientes no necesitan saber en qué casos utilizan un objeto primitivo o un objeto compuesto, esto simplifica el código, mejora el principio del menor privilegio y desacopla la implementación de la jerarquía del cliente.

3. Facilita la posibilidad de agregar nuevos tipos de componentes. Tanto nuevos primitivos como objetos compuestos funcionan de manera automática con las estructuras existentes y el código del cliente.

4. Puede causar que el diseño sea más general de lo aceptable. En caso de necesitar que las clases se apeguen a ciertas restricciones, el uso del sistema de tipos del lenguaje de programación no será suficiente por lo que es necesario implementar las restricciones y evaluarlas en tiempo de ejecución.

4.3. Pruebas Unitarias

Una práctica muy importante en el desarrollo de software a nivel profesional es la utilización de técnicas de prueba automatizada de software. En cualquier proceso de ingeniería siempre se debe reservar una parte de dicho proceso para el control de calidad. En el caso del software, la elaboración de pruebas automatizadas y software de verificación es un esfuerzo por concretar la parte de control de calidad.

En el caso de sistemas de software muy grandes, existe una gran gama de pruebas y procesos de control de calidad. Inicialmente podemos clasificar los procesos de prueba y verificación de software por niveles de organización:

1. **Pruebas unitarias (*unit tests*).** Consiste en verificar la correcta funcionalidad de las unidades más sencillas de software. En el caso de la programación orientada a objetos esto correspondería a probar un método, mientras que en la programación funcional y procedural sería probar una función.

2. **Pruebas de integración (*integration tests*).** Consiste en verificar el correcto funcionamiento al *ensamblar* distintos componentes de software previamentes probados de manera individual. En esta etapa es cuando se detectan defectos de la arquitectura de un sistema, problemas de interfaz, entre otro tipo de defectos de carácter más relacionado con el diseño de la arquitectura de un sistema de software.

3. **Pruebas de sistema.** Se hacen pruebas generales a un sistema de software completamente integrado para dar a conocer si cumple con los requerimientos especificados al momento de desarrollarse.

4. **Pruebas de integración de sistemas.** En este caso se busca verificar si un sistema dado puede coexistir e integrarse correctamente con otros sistemas. Está muy relacionado con aspectos de migración de plataforma o implanatación de sistemas en una configuración de software y hardware preexistente.

Además de realizar pruebas de verificación del comportamiento de un programa en comparación de los requerimientos del mismo, existen otro tipo de pruebas de suma importancia para los sistemas de software:

1. **Pruebas de desempeño (*performance testing*).** Sirven para estimar de manera precisa el desempeño de un componente de software respecto a diversos tipos de recursos (tiempo de procesador, memoria, consumo de energía, ancho de banda en red, etcétera).

2. **Pruebas de usabilidad.** Se utilizan para verificar si un programa es fácil de usar y entender.

3. **Pruebas de seguridad.** Verifican la robustez de un sistema en base a ataques de seguridad malintencionados o de carácter accidental.

4. **Pruebas de localización e internacionalización.** En caso de que un componente de software se arroje al mercado mundial, es importante verificar que las versiones "localizadas" de dicho sistema funcionen correctamente.

Si bien todas estas técnicas son útiles para el programador de cualquier nivel, este documento se enfoca en el primer nivel de organización, las pruebas

unitarias, ya que para un estudiante de computación resulta ser una herramienta muy útil y relativamente sencilla de aprender para mejorar sus prácticas de programación y reducir sus tiempos de desarrollo para proyectos que se cubren en materias como lenguajes de programación, sistemas operativos, inteligencia artifical, redes de computadoras, entre otros.

4.3.1. ¿Por qué hacer pruebas unitarias?

La razón principal para realizar pruebas unitarias de software está motivada por el siguiente principio:

> Encontrar una falla en un componente individual aislado es mucho más sencillo que encontrar la misma falla en un sistema que utiliza este componente para alguna aplicación.

Adicionalmente, tanto la detección como reproducción de fallas en un sistema de software ya integrado resulta ser una tarea comúnmente complicada y en muchos casos se termina siendo necesario elaborarla manualmente, incrementando sensiblemente los tiempos de desarrollo. En el caso de las pruebas unitarias podemos dar un efoque como en un laboratorio de ciencia experimental: reproducir un fenómeno en un ambiente controlado. Para lograr este propósito se han desarrollado diversas aplicaciones y bibliotecas especializadas para elaborar pruebas unitarias para un cierto ambiente de desarrollo específico (lenguaje de programación, entorno de tiempo de ejecución, bibliotecas, etcétera). En general, este tipo de herramientas se encapsulan y distribuyen como una biblioteca adicional para prueba unitaria (*testing framework*) dependiente del entorno de desarrollo.

Una importante razón para usar pruebas unitarias automatizadas (generadas por código) es que son muy útiles para detectar regresiones. Una regresión ocurre cuando se actualiza o reemplaza un componente preexistente que originalmente mantenía la correctud del sistema, pero al ser sustituído o actualizado puede suponer ciertas precondiciones inexistentes o viola postcondiciones necesarias para el sistema causando que este deje de funcionar correctamente.

4.3.2. Componentes básicos de una biblioteca de pruebas unitarias

Una biblioteca de prueba unitaria generalmente contiene los siguientes componentes:

1. **Accesorios de prueba (*test fixtures*).** Representan un módulo de la biblioteca de prueba unitaria que sirve para poder armar un escenario para ejecutar la prueba, esto es, safistacer un conjunto de precondiciones y estados de los elementos que intervendrán en la prueba para poder realizarla. Por ejemplo, si queremos verificar algún componente que transmite datos por un canal de red, el accesorio se encargará de conectarse y establecer el

estado de la conexión necesario para que dicho componente pueda exponer su funcionalidad.

2. **Aseveraciones (*assertions*).** Al realizar una prueba unitaria, es necesario comparar el resultado actual del componente a prueba contra un resultado esperado. Una aseveración es un predicado que indica una condición que se debe cumplir para asegurar la correctez del componente de software. Por ejemplo, si estamos probando un método que suma enteros, una aseveración podría ser:

```
1 // Caso de prueba para metodo suma: 3 + 2 = 5
2 // La aseveraion consiste en validar el resultado
3 // esperado, con el resultado actual del metodo.
4  int val = someInstance.add(2, 3);
5  boolean assert = assertEquals(5, val);
```

   Para los lenguajes orientados a objetos es común utilizar como convención una clase estática que contiene todos los métodos necesarios para evaluar aseveraciones (*es igual*, *es nulo*, *es distinto*, *es verdadero*, etcétera). En el caso de lenguajes funcionales o procedurales, dicha funcionalidad se expone como un módulo de funciones.

3. **Objetos simulados (*mock objects*).** En algunos casos el poder armar un escenario controlado resulta ser una tarea muy complicada o que consume demasiados recursos. Un ejemplo común es el necesitar interconectar computadoras para validar algún componente de red. En vez de esto, se puede utilizar un objeto que simule el comportamiento de algún nodo de la red y que sea generado de manera controlada por el *test fixture*.

   Si bien los objetos simulaodos son muy útiles es muy díficil encontrar bibliotecas de prueba unitaria que cuenten con ellos, sin embargo existen bibliotecas adicionales que contienen dichos componentes. Para sistemas específicos complejos, es necesario desarrolar un objeto simulado de manera manual.

4.3.3. Caso de estudio: `JUnit`

Para el caso del entorno de desarrollo y ejecución de Java, existe una biblioteca muy conocida y la que es prácticamente el estándar para elaborar pruebas unitarias: `JUnit`[1]. La mejor forma de ilustrar cómo funciona es a través de un ejemplo.

Como ejemplo diseñaremos una clase sencilla que modela números racionales y soportaremos las cuatro operaciones básicas de la aritmética (suma, resta, multiplicación y división). Posteriormente utilizaremos `JUnit` para elaborar las pruebas unitarias e ilustrar su funcionamiento.

Introduciremos un error intencional en uno de los métodos de nuestra clase `Rational` con la finalidad de ilustrar el funcionamiento de `JUnit`. El defecto se encuentra en el método de multiplicación (`mul`).

[1] Se puede descargar `JUnit` de `http://www.junit.org`

```
1  public class Rational {
2      private int n, d;
3
4      public Rational(int num, int den){
5          String dz = "Denominador cero";
6          if(den == 0)
7              throw new IllegalArgumentException(dz);
8          n = num; d = den;
9          // Reducimos el racional
10         // usando el algoritmo de
11         // euclides.
12         euclides();
13     }
14
15     /** Suma de racionales */
16     public Rational add(Rational x){
17         int nn;
18         int cd = x.d * d;
19         nn = cd / d * n + cd / x.d * x.n;
20         return new Rational(nn, cd);
21     }
22
23     /** Diferencia de racionales */
24     public Rational sub(Rational x){
25         return this.add(x.mul(new Rational(-1, 1)));
26     }
27
28     /** Producto de racionales */
29     public Rational mul(Rational x){
30         // Tenemos un error al calcular
31         // el numerador, deberia ser
32         // * y no +.
33         return new Rational(n + x.n, d * x.d);
34     }
35
36     /** Cociente de racionales */
37     public Rational div(Rational x){
38         return new Rational(n * x.d, d * x.n);
39     }
40
41     /** Algoritmo de euclides
42      * para calcular el mcd.
43      */
44     private void euclides(){
45         int a, b;
46         a = n;
47         b = d;
48         while(b != 0){
49             int t = b;
50             b = a %b;
51             a = t;
52         }
53         // a es el mcd entre n y d.
54         n /= a;
55         d /= a;
56     }
57
```

```
58      @Override
59      public boolean equals(Object x){
60          // Si la clase a comparar no
61          // es un racional, devolvemos
62          // false.
63          if(x.getClass() != Rational.class)
64              return false;
65          Rational y = (Rational) x;
66          // Si recibimos racional,
67          // solo son iguales si
68          // numerador y denominador son
69          // iguales (ambos estan en su
70          // minima expresion).
71          return (n == y.n && d == y.d);
72      }
73
74      @Override
75      public String toString(){
76          return n + "/" + d;
77      }
78  }
```

Estructura típica de una clase de prueba de `JUnit`

En general una clase que utiliza `JUnit` se utiliza para probar una clase de código. La estructura general de un conjunto de pruebas para una clase con `JUnit` es:

```
1   // Importar el text fixture
2   // y el soporte de aseveraciones
3   import org.junit.After;
4   import org.junit.AfterClass;
5   import org.junit.Before;
6   import org.junit.BeforeClass;
7   import org.junit.Test;
8   import static org.junit.Assert.*;
9   public class ClassTest {
10      public ClassTest() {
11      }
12
13      @BeforeClass
14      public static void setUpClass() throws Exception {
15          // Codigo que se ejecuta antes de
16          // empezar a ejecutar los casos de prueba
17      }
18
19      @AfterClass
20      public static void tearDownClass() throws Exception {
21          // Codigo que se ejecuta despues de
22          // terminar los casos de prueba.
23      }
24
25      @Before
26      public void setUp() {
27          // Codigo que se ejecuta antes de
```

```
28          // la invocacion de cada caso de prueba.
29      }
30
31      @After
32      public void tearDown() {
33          // Codigo que se ejecuta despues de
34          // la invocacion de cada caso de prueba.
35      }
36
37
38      @Test
39      public void testMethod1() {
40          // Codigo para probar method1
41      }
42
43      @Test
44      public void testMethod2() {
45          // Codigo para probar method2
46      }
47
48      // ...
49  }
```

Existen 5 anotaciones que se pueden aplicar a los métodos de la clase de prueba para indicar cuándo se ejecutan y si son casos de prueba o no:

1. `@BeforeClass`. Esta anotación indica un método que se va a ejecutar antes de la ejecución de todos los casos de prueba.

2. `@AfterClass`. Recíprocamente, esta anotación se coloca en los métodos que queremos invocar al finalizar la ejecución de todos los casos de prueba.

3. `@Before`. Se usa para indicar el método que queremos ejecutar antes de la ejecución de **cada** caso de prueba.

4. `@After`. Análogo al caso anterior sólo que al finalizar cada caso de prueba.

5. `@Test`. Indica que el método anotado es un caso de prueba.

Adicionalmente, `JUnit` cuenta con una clase que proporciona métodos de aseveración: `org.junit.Assert`. Si bien esta clase tiene una gran cantidad de métodos para evaluar aseveraciones, todos ellos son simplemente sobrecarga del siguiente conjunto:

1. `Assert.assertArrayEquals(expected, actual)`. Evalúa si dos arreglos contienen los mismos datos.

2. `Assert.assertEquals(expected, actual)`. Evalúa la igualdad de dos instancias.

3. `Assert.assertFalse(boolean)`. Evalúa si la expresión o valor booleano son falsos.

4. `Assert.assertNotNull(object)`. Evalúa si el objeto no es nulo.
5. `Assert.assertNotSame(ptr1, ptr2)`. Evalúa si dos identificadores no apuntan al mismo objeto.
6. `Assert.assertTrue(boolean)`. Evalúa si la expresión o valor booleano es verdadero.
7. `Assert.fail()`. Falla el caso de prueba.
8. `Assert.failNotEquals(message, expected, actual)`. Falla el caso de prueba en caso de que las instancias proporcionadas no sean iguales. Manda el mensaje de texto especificado por `message`.
9. `Assert.failNotSame(message, expected, actual)`. Falla el caso de prueba en caso de que los identificadores proporcionados no apunten al mismo objeto.

Ejemplo de pruebas para la clase `Rational`

Ya que conocemos la estructura general de `JUnit` podemos elaborar una clase que pruebe nuestra implementación de los números racionales:

```
1  // No necesitamos ninguna de las
2  // anotaciones para ejecutar
3  // codigo antes o despues de
4  // cada caso o conjunto de pruebas.
5  import org.junit.Test;
6  import static org.junit.Assert.*;
7
8  public class RationalTest {
9      public RationalTest() {
10     }
11
12     /** Validamos el constructor
13      *  usando valores permitidos para
14      *  el denominador.
15      */
16     @Test
17     public void testConstructorLegal(){
18         System.out.println("Legal constructor");
19         Rational r = new Rational(1,5);
20         // Un caso de prueba pasa
21         // en caso de que ninguna aseveracion
22         // falle o no se arroje una excepcion.
23     }
24
25     /** Validamos el constructor
26      *  usando un valor invalido en
27      *  el denominador. Usamos el
28      *  parametro expected para
29      *  especificar que esperamos
30      *  que se arroje una excepcion.
31      */
32     @Test(expected=java.lang.IllegalArgumentException.class)
```

```
33      public void testConstructorIllegalDen(){
34          System.out.println("Illegal constructor");
35          Rational r = new Rational(0,0);
36      }
37
38      /**
39       * Prueba del metodo de suma.
40       */
41      @Test
42      public void testAdd() {
43          System.out.println("add");
44          // 3/4 + 1/4 = 1
45          Rational x = new Rational (3,4);
46          Rational instance = new Rational(1,4);
47          Rational expResult = new Rational(1,1);
48          Rational result = instance.add(x);
49          assertEquals(expResult, result);
50      }
51
52      /**
53       * Prueba del metodo de diferencia.
54       */
55      @Test
56      public void testSub() {
57          System.out.println("sub");
58          // 1 - 5/6 = 1/6
59          Rational x = new Rational(5,6);
60          Rational instance = new Rational(2,2);
61          Rational expResult = new Rational(1,6);
62          Rational result = instance.sub(x);
63          assertEquals(expResult, result);
64      }
65
66      /**
67       * Prueba del metodo de producto.
68       */
69      @Test
70      public void testMul() {
71          System.out.println("mul");
72          // 6/7 * 2/3 = 4/7
73          Rational x = new Rational(6,7);
74          Rational instance = new Rational(2,3);
75          Rational expResult = new Rational(4,7);
76          Rational result = instance.mul(x);
77          assertEquals(expResult, result);
78      }
79
80      /**
81       * Prueba del metodo de division.
82       */
83      @Test
84      public void testDiv() {
85          System.out.println("div");
86          // 1/4 div 1/2 = 1/2
87          Rational x = new Rational(1,2);
88          Rational instance = new Rational(1,4);
89          Rational expResult = new Rational(1,2);
```

```
90          Rational result = instance.div(x);
91          assertEquals(expResult, result);
92      }
93
94      /**
95       * Probamos el metodo que evalua si
96       * dos racionales son iguales.
97       */
98      @Test
99      public void testEquals(){
100         System.out.println("equals");
101         Rational a = new Rational(1,2);
102         Rational b = new Rational(2,4);
103         assertTrue(a.equals(b));
104         // Debe fallar, no tiene
105         // sentido la comparacion.
106         assertFalse(a.equals("foo"));
107     }
108 }
```

Compilación y ejecución de las pruebas de `JUnit`

Una vez que tenemos el archivo de pruebas, debemos proceder a compilarlo de la siguiente forma:

```
$ javac -cp .:<jar de JUnit> RationalTest.java
```

Después de finalizar el proceso de compilación, la forma más sencilla de ejecutar las pruebas es utilizando el arnés por omisión de `JUnit` de la siguiente manera:

```
$ java -cp .:junit-4.5.jar org.junit.runner.JUnitCore RationalTest
```

Después de ejecutar las pruebas recibiremos dos errores debido a la incorrecta implementación del método de multiplicación. Son dos errores ya que el método de diferencia utiliza la multiplicación para sacar el inverso aditivo del parámetro dado y luego sumarlo, esto con la finalidad de reutilizar código. La salida de `JUnit` será pues:

```
$ java -cp .:junit-4.5.jar org.junit.runner.JUnitCore RationalTest
JUnit version 4.5
.Legal constructor
.Illegal constructor
.add
.sub
E.mul
E.div
.equals

Time: 0.01
```

```
There were 2 failures:
1) testSub(RationalTest)
java.lang.AssertionError: expected:<1/6> but was:<5/3>
at org.junit.Assert.fail(Assert.java:91)
at org.junit.Assert.failNotEquals(Assert.java:618)
at org.junit.Assert.assertEquals(Assert.java:126)
at org.junit.Assert.assertEquals(Assert.java:145)
at RationalTest.testSub(RationalTest.java:51)
...
2) testMul(RationalTest)
java.lang.AssertionError: expected:<4/7> but was:<8/21>
at org.junit.Assert.fail(Assert.java:91)
at org.junit.Assert.failNotEquals(Assert.java:618)
at org.junit.Assert.assertEquals(Assert.java:126)
at org.junit.Assert.assertEquals(Assert.java:145)
at RationalTest.testMul(RationalTest.java:64)
...

FAILURES!!!
Tests run: 7,  Failures: 2
```

Después de corregir la falla, (sustituyendo el `+` por `*`), recompilando `Rational.java`, al volver a ejecutar las pruebas tenemos:

```
$ java -cp .:junit-4.5.jar org.junit.runner.JUnitCore RationalTest
JUnit version 4.5
.Legal constructor
.Illegal constructor
.add
.sub
.mul
.div
.equals

Time: 0.008

OK (7 tests)
```

4.4. Estrategias de desarrollo

A continuación se presentan dos metodologías de desarrollo muy populares en la actualidad. Su importancia radica tanto en utilizar una estrategia de desarrollo ágil de aplicaciones como también porque no requiere grupos muy grandes de desarrolladores, haciéndolas bastante aplicables para el caso de un estudiante de licenciatura que puede trabajar tanto de manera individual como en equipos muy pequeños (máximo 2 o 3 personas).

4.4.1. Desarrollo guiado por pruebas

Como su nombre lo indica, es la técnica de desarrollar software basándose en las pruebas unitarias antes de desarrollar el programa, es decir, primero escribimos las pruebas necesarias (que sabemos que deben fallar) y poco a poco vamos implementando funcionalidad hasta conseguir hacer pasar todas las pruebas. Esta metodología ofrece diversas ventajas:

1. **Simplificación del diseño.** Como queremos facilitar la elaboración de las pruebas unitarias para poder realizar el proceso de desarrollo, fomentamos un diseño sencillo y fácil de probar, mejorando la transparencia y reduciendo la complejidad del programa lo más posible.

2. **Mejora de la calidad de las pruebas.** El ciclo de desarrollo guiado por pruebas establece que cada vez que se quiere agregar nueva funcionalidad, hay que probar primero cada caso de prueba verificando si falla antes de desarrollar la funcionalidad. Esto representa un buen hábito de desarrollo y minimiza la posibilidad de desarrollar pruebas que pasen por sí solas, muchas veces ocultando defectos reales de código.

Ciclo de desarrollo

La serie de pasos a seguir para realizar un proceso de desarrollo de software utilizando la metología de desarrollo guiado por pruebas consiste en:

1. **Agregar un caso de prueba.** Inicialmente agregamos un único caso de prueba que verifique nueva funcionalidad a implementar. Esta práctica causa que el desarrollador se enfoque en los requerimientos del programa *antes* de implementar funcionalidad.

2. **Correr todos los casos y verificar que el nuevo caso falle.** Al momento de haber finalizado la escritura del caso de prueba, debemos asegurarnos que el caso falla para evitar tener falsos resultados satisfactorios en torno a una prueba.

3. **Escribir código.** Una vez que hay la certeza que el caso de prueba falla, se procede a desarrollar *mínima funcionalidad necesaria* para hacer que el caso de prueba pase, esto con la finalidad de siempre mantener código fuente 100 % probado y con menor probabilidad de contener defectos.

4. **Ejecutar los casos de prueba y verificar que el código es correcto.** Análogamente a las técnicas de desarrollo tradicionales, probamos que nuestro nuevo código pasa los casos de prueba.

5. **Reorganizar código.** Finalmente, después de saber que tenemos código correcto es necesario reorganizar y limpiar el código de tal modo que se mejoren aspectos de desempeño, legibilidad y escalabilidad. Esta última etapa es necesaria ya que en algunos casos existen piezas de código duplicadas que hacen pasar diversos casos de prueba independientes, entre otras

razones (restricción de acceso, comentarios, uso de objetos de simulación, etcétera).

6. **Repetir hasta finalizar el desarrollo.**

Limitaciones

Existen situaciones donde esta metodología no es fácil de aplicar:

1. **Verificación de usabilidad, interacción con el usuario, red, entre otros.** En general, el utilizar pruebas unitarias facilita la verificación de la funcionalidad central del programa; sin embargo, cuestiones relacionadas con diseño de interfaces, interacción usuario-máquina y requerimientos de comunicaciones son aspectos muy difíciles de probar por medio de pruebas unitarias y es posible que requieran otra técnica para verificarse.

2. **Sobrecarga de trabajo por pruebas y reorganización de código.** Si bien es cierto que usar pruebas unitarias puede mejorar significativamente los tiempos de desarrollo en la etapa de depuración, también es cierto que escribir pruebas unitarias consume tiempo, y para escenarios complicados puede ser que la implementación de la prueba sea más compleja que el código a probar. Adicionalmente hay que tomar en cuenta el tiempo invertido en la reorganización de código necesaria para el mantenimiento de la aplicación.

4.5. Programación en pares

Es una técnica común de desarrollo aplicada a procesos de desarrollo de software ágil. La ventaja de esta técnica es que se puede aplicar tanto para programadores expertos como principiantes haciéndola una práctica muy útil para estudiantes que están cursando materias de programación.

4.5.1. Definición

La programación en pares es una técnica de desarrollo en la cual dos programadores trabajan conjuntamente a través de un solo equipo de cómputo: uno se encuentra en control del equipo escribiendo código, mientras el otro revisa cada línea que va siendo introducida por el primer participante. El participante que está frente al equipo se denomina *controlador*, mientras que el participante que revisa el código se le conoce como *observador*. Una característica importante de esta técnica es que los roles de observador y controlador se deben cambiar periódicamente en intervalos de al menos media hora.

4.5.2. Ventajas

1. **Calidad de diseño.** Teniendo dos participantes se puede conseguir mejor legibilidad del código ya que ambos necesitan poderlo leer y entender, se

exploran más posibilidades de diseño propuestas por ambos roles y se pueden detectar defectos de diseño de manera más temprana teniendo dos participantes.

2. **Costo reducido de desarrollo.** Si bien la concurrencia de programadores es 1 (sólo uno a la vez escribe código), la detección de defectos de código (conocidos como *bugs*) por el observador reduce *significativamente* el tiempo de desarrollo, ya que la detección de defectos de código toma mucho más tiempo en etapas posteriores del desarrollo.

3. **Aprendizaje conjunto.** La colaboración entre roles incentiva la comunicación y el intercambio de conocimientos técnicos entre ambos.

4. **Moral mejorada.** Para muchos programadores, la programación en parejas es más agradable que programar solos.

5. **Mejora en la disciplina y administración de tiempo.** Se disminuye la tendencia a distraerse u omitir partes que en algunos casos suelen ser tediosas programando solo (escribir pruebas unitarias, comentar y documentar el código, etcétera).

4.5.3. Desventajas

1. **Preferencias.** Existen programadores que prefieren trabajar solos.

2. **Complicación para coordinar.** Poder coordinar tiempos y disposición de dos personas es mucho más complejo que una sola persona.

3. **Conflicto de hábitos.** Posibles conflictos por la intolerancia de un participante a ciertos hábitos personales del otro.

Capítulo 5

Programación concurrente

5.1. Introducción

Actualmente, la tecnología de cómputo está atravesando por un cambio importante en sus arquitecturas y diseño tecnológico. Este importante giro se ve motivado por la dificultad de incrementar la velocidad de reloj de sincronización debido al sobrecalientamiento de los circuitos en los nuevos procesadores que cuentan con un altísimo nivel de integración (millones de transistores en un centímetro cuadrado). Por lo tanto, la carrera del reloj se ha terminado; ahora viene una nueva etapa que consiste en incrementar las unidades de procesamiento en la misma estampa o *chip*. Este es el nacimiento de la nueva tecnología de procesadores llamada *multicore* (multinúcleo) y consiste en tener múltiples procesadores en la misma estampa compartiendo los mismos cachés de memoria o segmentando el caché para cada procesador individual.

El incremento de la tasa de conmutación del reloj de pulsos de un circuito daba de manera gratuita una reducción en el tiempo de ejecución de los programas; por lo tanto, cada vez que había una mejora tecnológica en la tecnología de procesadores, el beneficio se reflejaba inmediatamente en la ejecución de los programas; sin embargo, en la nueva era del multicore esto se ha terminado. Si bien la ley de Moore no se ha detenido[1], el incremento de transistores ya no se traduce en velocidad de procesamiento del CPU, sino en multiplicar el número de unidades de procesamiento (y cachés); el incremento en la velocidad de reloj ya se detuvo y de hecho ha disminuído para circuitos con muchos núcleos. Actualmente la ganancia derivada de la Ley de Moore es el número de núcleos por estampa; así pues, la nueva carrera está basada en el *paralelismo*.

Cada núcleo se puede ver como un procesador independiente y las nuevas aplicaciones deben de ser programadas y diseñadas de tal manera que exploten el paralelismo de núcleos, es decir, coordinar procesos y código para poder

[1]La Ley de Moore estipula que el número de transistores en un circuito integrado se duplica aproximadamente cada 18 meses. Es simplemente una ley empírica formulada por el cofundador de *Intel* Gordon E. Moore.

distribuir su carga en los diversos núcleos del procesador. Este proceso cambia radicalmente la forma de programar software y requiere de nuevas habilidades y técnicas que anteriormente se consideraban temas avanzados en las universidades e instituciones dedicadas a la computación.

5.1.1. Explotar el paralelismo

Hemos dicho que con las nuevas arquitecturas multicore es necesario explotar el paralelismo de la máquina modificando la programación de una aplicación. Las preguntas naturales a continuación serían: *¿cómo puedo explotar el paralelismo de mi máquina?¿Qué necesito para explotarlo?* y la respuesta es: por medio de la ejecución de diversos procesos coordinados, es decir, ejecutar diversas *instancias* de código que trabajen simultáneamente en diversos procesadores de manera coordinada para resolver un problema.

Los sistemas operativos modernos encapsulan y administran los núcleos del procesador de manera controlada por medio del *kernel* (o núcleo del sistema operativo), así que la forma de conseguir ejecutar código en paralelo es accediendo a los núcleos del procesador por medio de la interfaz del sistema operativo. Tradicionalmente, los sistemas operativos utilizan *procesos* para acceder al procesador y ejecutar código.

5.1.2. Procesos e hilos de control (*threads*)

Un proceso, a diferencia de un programa, se puede ver como una *instancia de un programa en ejecución*. El sistema operativo utiliza varias estructuras de datos para poder realizar la ejecución de procesos y poder administrar memoria y recursos. Tradicionalmente, un proceso está constituido por los siguientes componentes:

1. **Sección de código.** Corresponde al código máquna producto de compilar el código fuente desarrollado por el programador. Contiene las instrucciones a ejecutar por el proceso.

2. **Sección de datos.** Consiste en el conjunto de datos estáticos que requiere el programa para operar (cadenas de mensajes, direcciones de memoria, identificadores, variables, información de depuración, etcétera).

3. **Archivos abiertos.** El sistema operativo lleva la pista de qué archivos tiene abierto el proceso y examina el estado de los mismos (posición, tamaño, control de acceso, etcétera).

4. **Registros.** Se refiere a los valores de los registros del núcleo o procesador en el instante de ejecución del proceso.

5. **Pila de ejecución.** Para poder implementar llamadas a funciones, es necesario mantener una estructura de datos de tipo pila que vaya almacenando el estado de cada llamada.

6. ***Heap***. El *heap* es una región de memoria que asigna el sistema operativo a un proceso con la finalidad de ofrecer servicios de memoria dinámica.

Anteriormente, un proceso estaba manejado por un solo *hilo de control* o *thread*, esto es, el proceso sólo podía ejecutar una sola tarea al mismo tiempo, y por lo mismo, el sistema operativo sólo llevaba pista de un sólo contador de programa y una sola pila de ejecución sin la posibilidad de realizar diversas tareas de manera simultánea. Esto explica por qué en sistemas más antiguos no era posible ejecutar tareas en segundo plano, o realizar diversas tareas de manera independiente en la misma aplicación.

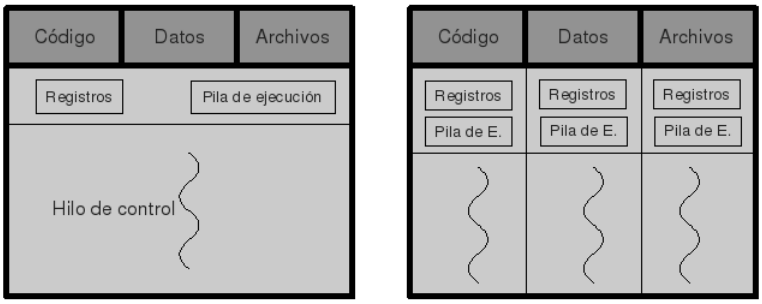

Figura 5.1: Comparación entre un proceso con un solo hilo de control y un proceso con varios hilos de control

Conociendo la definición de proceso, ya podemos estudiar cómo podemos explotar el paralelismo de una arquitectura multicore. En una primera aproximación, la forma más sencilla de explotar el paralelismo es asignar un proceso a cada procesador. Esta técnica se conoce como *paralelismo a nivel de proceso*. Si queremos que el procesador use todos sus núcleos para resolver un problema de manera colaborativa, podemos usar diversas instancias de un mismo programa para resolver la tarea y utilizar alguna biblioteca de comunicación entre procesos que nos permita comunicarlos entre ellos para intercambiar mensajes y resultados y, de este modo, coordinar los procesadores para resolver el problema.

El problema con esta aproximación es el desperdicio innecesario de recursos, ya que cada proceso tiene una copia local de la sección de datos, código y archivos abiertos. Si queremos resolver un problema en común a través de varios procesos, es natural que estas secciones sean las mismas en todas las instancias del programa, por lo que estamos desperdiciando espacio en memoria y complicando la administración al sistema operativo. Lo ideal sería poder compartir estas secciones en diversos hilos de control. En la actualidad, existe una gran cantidad de aplicaciones que siguen este modelo: *un solo proceso en ejecución, pero múltiples hilos de control por proceso*. Esto nos ofrece la posibilidad de, en un solo proceso, compartir las secciones de código, datos y archivos y realizar diversas tareas simultáneamente (o al menos, sin bloquear una a la otra en caso de arquitecturas de un solo procesador usando el calendarizador del sistema

operativo para cada hilo).

Si bien existen regiones compartidas por todos los hilos de ejecución, existirá información que sea necesario tener por thread: los registros y la pila de ejecución. Esto nos ofrece la posibilidad de que cada hilo se encuentre ejecutando una pieza de código diferente y con un estado distinto también.

Es importante mencionar que existen aplicaciones distribuidas o que ejecutan código de manera paralela usando ambos modelos, dependiendo de su aplicación. Aunque en la actualidad la gran mayoría utilizan múltiples hilos de control por proceso, existen algunas aplicaciones que se implementan por medio de la ejecución de distintos procesos, principalmente la familia de servicios de red de `UNIX`, comúnmente llamados *demonios* (*daemons*). La razón es porque requieren un grado alto de independencia o porque es código heredado de varios años atrás, donde la programación multihilo era incipiente o inexistente. Algunos ejemplos:

1. `Telnet`. Un servicio viejo de consola remota, actualmente reemplazado por secure shell.

2. Secure Shell (`ssh`).

3. El servidor web apache[2]. Actualmente uno de los servidores web más usados en todo el mundo. También existe una versión que funciona con hilos de control en vez de duplicación de procesos.

4. `nfs` (Network File System). El sistema de archivos de red de la familia de sistemas operativos tipo `UNIX`.

5.1.3. Bibliotecas para programación multiproceso/multihilo

En particular existen varias bibliotecas muy populares para programación concurrente, ya sea por medio de hilos o por duplicación de procesos:

1. Biblioteca estándar de `POSIX` (`unistd`). La biblioteca estandarizada para todos los sistemas tipo `UNIX`. Diseñada y utilizada en lenguaje `C`. Si bien no ofrece soporte de hilos, sí ofrece la posibilidad de clonar, comunicar y administrar diversos procesos.

2. Biblioteca `pthreads` (`POSIX` *threads*). La biblioteca más popular para desarrollar aplicaciones multihilo en sistemas tipo `UNIX`. Al igual que la biblioteca `unistd`, `pthreads` se desarrolla y utiliza por medio del lenguaje `C`.

3. Biblioteca de hilos de *Microsoft Windows*. Necesaria para desarrollar aplicaciones multhilo en el sistema operativo de Microsoft. Desarrollada en `C`.

[2] `http://www.apache.org`

4. La *API*[3] de `Java`. La máquina virtual de Java provee de forma nativa la posibilidad de desarrollar aplicaciones multihilo de manera sencilla y ofrece una gran cantidad de servicios para el control, sincronización y administración de los mismos. Es muy útil en el ámbito académico y de buen alcance para el desarrollo de aplicaciones comerciales.

5. La *API* de `.NET` de Microsoft. De reciente desarrollo y rápida penetración en el mercado, la máquina virtual de Microsoft (`.NET`) ofrece una biblioteca muy madura y poderosa para el manejo de hilos de control. Su uso es similar al de Java. Se puede utilizar a través de toda la gama de lenguajes soportados por los compiladores de `.NET` de Microsoft (`C`, `C++`, `C#`, `Visual Basic`, `Java`, entre otros).

5.1.4. Algunos ejemplos

En el caso de las bibliotecas de Microsoft, tanto la API de hilos nativa de Windows como la de .NET, no se muestran ejemplos ya que son muy similares a los ejemplos de pthreads y Java respectivamente. La documentación y ejemplos de las bibliotecas de hilos de Microsoft están disponibles en la red de desarrolladores de Microsoft (*MSDN*)[4].

A continuación se muestra la solución de un problema por medio del uso de algunas bibliotecas. Todos los ejemplos que siguen resuelven el mismo problema: ordenar un arreglo de enteros. Para dicho efecto, paralelizamos por medio de dos instancias de código que se ejecutan independientemente, ya sea por medio de dos procesos distintos o un solo proceso con dos hilos de control.

Multiples procesos con la biblioteca `unistd`

Inicialmente, se propone la siguiente solución:

```
1  #include <iostream>
2
3  using namespace std;
4
5  #include <cstdlib>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/wait.h>
9
10 #define TAM 20
11
12 /* Imprime un arreglo */
13 void print(int *nums){
14   int i;
15   for(i = 0; i < TAM - 1; ++i){
16     cout << nums[i] << ", ";
17     if(i == 9)
18       cout << endl;
```

[3] Interfaz de programación de aplicaciones (*Application Programming Interface*)

[4] Microsoft Developer Network. `http://msdn.microsoft.com`

```
19    }
20    cout << nums[i] << endl;
21  }
22
23  /* Genera un arreglo aleatorio */
24  int randomize(int *nums){
25    int i;
26    for(i = 0; i < TAM; ++i){
27      nums[i] = rand() % 100;
28    }
29  }
30
31  /* Implementacion del ordenamiento
32   * por seleccion.
33   */
34  void selectionsort(int *nums, int start, int end){
35    int i, j;
36    for(i = start; i < end; ++i){
37      for(j = i; j < end; ++j){
38        if(nums[j] < nums[i]){
39          int t = nums[i];
40          nums[i] = nums[j];
41          nums[j] = t;
42        }
43      }
44    }
45  }
46
47  int main(){
48    pid_t pid;
49    // Creamos el arreglo.
50    int *nums = new int[TAM];
51    // Inicializamos semilla aleatoria
52    // y aleatorizamos el arreglo.
53    srand(time(0));
54    randomize(nums);
55
56    cout << "Arreglo inicial aleatorizado: " << endl;
57    print(nums);
58
59    // Llamamos a fork, que crea un nuevo
60    // proceso hijo.
61    pid = fork();
62
63    // Si el identificador de proceso
64    // es menor que cero, hubo un error.
65    if(pid < 0){
66      cerr << "Error, no se pudo crear proceso hijo" << endl;
67    }
68
69    // Si el identificador es cero
70    // estamos ejecutando el proceso
71    // hijo, por lo tanto ordenamos
72    // la segunda mitad.
73    if(pid == 0){
74      cout << "Inicia proceso hijo, ordenando "
75           << "segunda mitad..." << endl;
```

```
76      selectionsort(nums, TAM/2, TAM);
77      cout << "Despues de ordenar en el hijo: " << endl;
78      print(nums);
79    } else {
80      // En caso de que el pid sea mayor que
81      // cero, estamos en el proceso padre,
82      // el cual se encargara de ordenar la
83      // primer mitad del arreglo.
84      int status = 0;
85      selectionsort(nums, 0, TAM/2 - 1);
86      // Esperamos a que el hijo termine.
87      wait(&status);
88      cout << "Despues de esperar al hijo y habiendo "
89           << "ordenado la primera mitad:" << endl;
90      print(nums);
91    }
92  }
```

Sin embargo, esta solución es *incorrecta*; la razón es que al momento de clonar procesos, tenemos una sección de datos independiente para cada uno (véase 5.1), esto causa que la llamada `fork()` duplique la sección de datos para el proceso hijo, por lo que el arreglo a ordenar está duplicado en secciones diferentes y lo que se imprime al final es la copia correspondiente al padre. Necesitamos un mecanismo de *memoria compartida* entre procesos hijo y padre, por lo cual utilizamos el servicio de memoria compartida de `UNIX`.

```
1   /* El codigo es igual al anterior,
2    * solo varia la funcion main.
3    */
4
5   #include <sys/ipc.h>
6   #include <sys/shm.h>
7
8   /* Implementa la mezcla de
9    * dos arreglos ordenados,
10   * dejando ordenado el arreglo
11   * original de manera total.
12   */
13  void merge(int *nums){
14    int s1 = 0;
15    int s2 = TAM/2;
16    int *temp = new int[TAM];
17    int i = 0;
18
19    while(s1 < TAM/2 && s2 < TAM){
20      if(nums[s1] < nums[s2]){
21        temp[i] = nums[s1];
22        ++s1;
23      } else {
24        temp[i] = nums[s2];
25        ++s2;
26      }
27      ++i;
28    }
29
30    while(s1 < TAM/2){
```

```
31      temp[i++] = nums[s1++];
32    }
33
34    while(s2 < TAM){
35      temp[i++] = nums[s2++];
36    }
37
38    for(i = 0; i < TAM; ++i){
39      nums[i] = temp[i];
40    }
41    delete[] temp;
42  }
43
44  int main(){
45    key_t key;
46    int shmid;
47    int *nums;
48    pid_t pid;
49
50    // Generamos un identificador
51    // de memoria compartida unico
52    // usando la funcion ftok que toma
53    // como argumentos un archivo y un
54    // caracter.
55    key = ftok("sort.cpp", 'a');
56
57    // En caso de que la funcion shmget
58    // nos devuelva -1, no fue posible
59    // reservar memoria compartida.
60    if((shmid = shmget(key, TAM * sizeof (int),
61                       0644 | IPC_CREAT)) == -1){
62      perror("shmget");
63      exit(1);
64    }
65
66    // Una vez teniendo memoria compartida
67    // asignamos el segmento a un apuntador.
68    nums = (int *) shmat(shmid, NULL, 0);
69
70    srand(time(0));
71    randomize(nums);
72
73    cout << "Arreglo inicial aleatorizado: " << endl;
74    print(nums);
75
76    pid = fork();
77    if(pid < 0){
78      cerr << "Error, no se pudo crear proceso hijo" << endl;
79    }
80
81    if(pid == 0){
82      cout << "Inicia proceso hijo, ordenando "
83           << "segunda mitad..." << endl;
84      selectionsort(nums, TAM/2, TAM);
85      cout << "Despues de ordenar en el hijo: " << endl;
86      print(nums);
87    } else {
```

```
88      int status = 0;
89      selectionsort(nums, 0, TAM/2);
90      wait(&status);
91      cout << "Despues de esperar al hijo y habiendo "
92           << "ordenado la primera mitad:" << endl;
93      print(nums);
94      cout << "Mezclando ambos subarreglos: " << endl;
95      merge(nums);
96      print(nums);
97      // 'Desconectamos' el apuntador
98      // de la region de memoria compartida.
99      shmdt(nums);
100     // Liberamos el segmento de memoria
101     // compartida.
102     shmctl(shmid, IPC_RMID, NULL);
103   }
104 }
```

Hilos con Java

```
1  public class ThreadedSort extends Thread {
2      // Arreglo a ordenar.
3      private int[] a;
4      // Tamano del arreglo de ejemplo.
5      public static final int TAM = 20;
6      private int s, e, id;
7
8      /** Crea un nuevo hilo de control
9       * para ordenar una parte del
10      * arreglo.
11      */
12     public ThreadedSort(int id, int[] a,
13                         int start, int end){
14         super("Thread " + id);
15         this.a = a;
16         s = start;
17         e = end;
18         this.id = id;
19     }
20
21     /** Implementacion del ordenamiento
22      * por medio de selection sort.
23      */
24     public void run(){
25         int i, j;
26         for(i = s; i < e; ++i){
27             for(j = i; j < e; ++j){
28                 if(a[j] < a[i]){
29                     int t = a[i];
30                     a[i] = a[j];
31                     a[j] = t;
32                 }
33             }
34         }
35         System.out.print("Finalizo la ejecucion ");
```

```
36          System.out.println("del thread " + id);
37      }
38
39      /** Ejemplo de uso */
40      public static void main(String[] args){
41          /* Creamos el arreglo */
42          int[] nums = new int[ThreadedSort.TAM];
43          for(int i = 0; i < nums.length; ++i){
44              nums[i] = (int) (Math.random() * 100);
45          }
46          System.out.println("Arreglo aleatorizado: ");
47          print(nums); // Mostramos el arreglo aleatorio
48          // Creamos ambos hilos que ordenen la
49          // primera y segunda mitad del arreglo
50          // respectivamente.
51          Thread ts1 = new ThreadedSort(0, nums, 0, TAM/2);
52          Thread ts2 = new ThreadedSort(1, nums, TAM/2, TAM));
53          // Ejecutamos los hilos.
54          ts1.start();
55          ts2.start();
56          // El hilo principal (main)
57          // debe esperar a que los hilos
58          // que ordenan terminen y no salir
59          // antes de tiempo.
60          try {
61              ts1.join();
62              ts2.join();
63          } catch(InterruptedException e){ }
64          System.out.print("Despues de ejecutar los ");
65          System.out.print("threads de ordendamiento ");
66          System.out.println("y mezclando el arreglo: ");
67          merge(nums); // Mezclamos los arreglos.
68          print(nums); // Solucion.
69      }
70
71      private static void merge(int[] a){
72          // Analogo al codigo en C++.
73      }
74
75      /** Imprime un arreglo */
76      private static void print(int[] a){
77          int i;
78          for(i = 0; i < a.length - 1; ++i){
79              System.out.print(a[i] + ",");
80              if(i == 9)
81                  System.out.println();
82          }
83          System.out.println(a[i]);
84      }
85  }
```

Múltiples hilos con la biblioteca `pthread`

```
1   #include <stdio.h>
2   #include <stdlib.h>
```

```
3  #include <pthread.h>
4
5  #define TAM 20
6
7  /* Estructura de datos para pasar
8   * argumentos a cada thread */
9  typedef struct p_attr_s p_attr;
10
11 struct p_attr_s {
12   int *a;
13   int start;
14   int end;
15 };
16
17 void *func(void *t){
18   p_attr *args = (p_attr *) t;
19   int i, j;
20   for(i = args -> start; i < args -> end; ++i){
21     for(j = i; j < args -> end; ++j){
22       if(args -> a[j] < args -> a[i]){
23         int t = args -> a[i];
24         args -> a[i] = args -> a[j];
25         args -> a[j] = t;
26       }
27     }
28   }
29   pthread_exit(NULL);
30 }
31
32 void merge(int *nums){
33   /* ... */
34 }
35
36 void print(int *a){
37   int i;
38   for(i = 0; i < TAM; ++i){
39     printf("%d, ", a[i]);
40     if(i == 9)
41       printf("\n");
42   }
43   printf("\n");
44 }
45
46 int main(){
47   pthread_t t1, t2;
48   pthread_attr_t attr;
49   p_attr a1, a2;
50   int a[TAM];
51   int i;
52
53   srand(time(0));
54   for(i = 0; i < TAM; ++i){
55     a[i] = rand() % 100;
56   }
57
58   printf("Arreglo aleatorizado:\n");
59   print(a);
```

```
60
61    /* Establecemos que los threads
62     * a crear soportan join */
63    pthread_attr_init(&attr);
64    pthread_attr_setdetachstate(&attr,
65                                PTHREAD_CREATE_JOINABLE);
66
67    /* Establecemos los parametros para
68     * cada thread */
69
70    /* Establecemos el
71     * arreglo a ordenar. */
72    a1.a = a2.a = a;
73    /* Establecemos las fronteras para
74     * cada thread al ordenar. */
75    a1.start = 0;
76    a1.end = a2.start = TAM/2;
77    a2.end = TAM;
78
79    /* Iniciamos cada thread */
80    pthread_create(&t1, &attr, func, (void *) &a1);
81    pthread_create(&t2, &attr, func, (void *) &a2);
82
83    /* Especificamos al hilo de
84     * control principal que espere
85     * hasta que terminen los
86     * threads que ordenan el arreglo.
87     */
88    void *status = NULL;
89    pthread_join(t1, status);
90    pthread_join(t2, status);
91
92    /* Despues de esperar,
93     * mezclamos las soluciones */
94    merge(a);
95
96    printf("Despues de ejecutar los threads y ");
97    printf("mezclar subarreglos:\n");
98    print(a);
99    return 0;
100 }
```

5.2. Conceptos básicos

El desarrollo de aplicaciones con ejecución multiproceso o multihilo ofrece una gran cantidad de retos y complicaciones de diseño e implementación. Estas complicaciones se originan por el hecho de que las arquitecturas multicore son *asíncronas*, es decir, los retrasos y la rapidez para ejecutar instrucciones es completamente variable entre núcleos. Si bien un núcleo puede estar ejecutando instrucciones directamente con registros (pocos ciclos de reloj), otro núcleo puede estar esperando un dato proveniente del caché a un registro (cientos o miles de ciclos), posiblemente esperando obtener un dato de la memoria principal a un registro (cientos de miles o millones de ciclos), o en el peor de los casos, espe-

rando un dato del disco (cientos de millones de ciclos), por lo tanto los retrasos son muy distintos y pueden ocurrir de manera independiente en cada proceso o hilo.

Estos retrasos de manera independiente no representan ningún reto en particular, el problema reside en el hecho de que la mayoría de las aplicaciones que intentan explotar el paralelismo operan sobre *datos compartidos*. Es decir, existe una cierta región de la memoria principal donde se encuentran almacenados datos que los diversos hilos van a explotar de manera independiente (ya sea leyendo o escribiendo) y es necesario un mecanismo de arbitraje o coordinación para evitar problemas de sincronización, por ejemplo, leer un dato que ya fue actualizado por alguien más y que ha sido invalidado.

5.2.1. *Ley de Amdahl*: un resultado desconcertante

Al paralelizar un programa, nosotros esperaríamos que al tener n núcleos de procesamiento pudiéramos incrementar en n la velocidad de procesamiento, desafortunadamente esto casi nunca sucede en la práctica. Una primera razón de esto es por lo que mencionamos anteriormente: se despercidia tiempo en la comunicación entre núcleos a la memoria compartida y también debido a los procesos de sincronización necesarios para coordinar cada núcleo.

Existe una situación adicional por la cual no es posible alcanzar una mejora del desempeño en un factor de n dados n núcleos de procesamiento que está relacionada con que la carga de trabajo no suele ser simétrica entre los núcleos o procesadores y que el tiempo total de ejecución de un proceso generalmente será el tiempo del proceso más largo a ejecutar en cada núcleo. De manera más formal: definimos S como la razón entre el tiempo que toma un procesador en terminar una tarea divida entre el tiempo que tardaría en n procesadores trabajando concurrentemente para finalizar la misma tarea. La *Ley de Amdahl* caracteriza el valor máximo de S (S por *speedup* del inglés) en términos del parámetro p que representa la fracción de la tarea que se puede ejecutar en paralelo. Supongamos que para una tarea dada el tiempo que consume un solo procesador toma 1 (tiempo normalizado). Con n procesadores trabajando de manera concurrente la parte paralelizable p se ejecuta en p/n y la parte que no se paraleliza (secuencial) toma tiempo $1-p$. Finalmente, el tiempo de ejecución que toman los n procesadores en realizar la tarea es:

$$1 - p + \frac{p}{n} \tag{5.1}$$

Por lo tanto la mejora o *speedup* S (la razón de tiempo que toma un procesador contra el tiempo que toman los n procesadores) se calcula como:

$$S = \frac{1}{1 - p + \frac{p}{n}} = \frac{n}{n - p(n-1)} \tag{5.2}$$

Para mostrar la importancia de este hecho, supongamos que tenemos 4 proce-

sadores disponibles, podemos tabular algunos valores de S en base a p:

p	S
0.5	1.60
0.7	2.11
0.9	3.07
0.95	3.48
0.98	3.77
0.99	3.88
1	4.00

Cuadro 5.1: Valores de S en base a p para 4 procesadores

Tabulando los mismos valores ahora teniendo 8 procesadores la diferencia es aún más dramática:

p	S
0.5	1.78
0.7	2.58
0.9	4.71
0.95	5.93
0.98	7.02
0.99	7.48
1	8.00

Cuadro 5.2: Valores de S en base a p para 8 procesadores

Como podemos ver, en el caso de 8 procesadores, el paralelizar el 50 % del código nos ofrece una mejora de sólo 1.78 veces a comparación de correrlo secuencialmente en un solo procesador, un desconcertante resultado. Incluso subiendo la cantidad de código paralelizado a 90 % ¡sólo ofrece una mejora de 4.71 veces! Por lo tanto, entre más procesadores tengamos disponibles para paralelizar una tarea, más necesario se hace maximizar la fracción de código paralelizable (al menos más del 95 % como se puede ver en la tabla). Este hecho demuestra que poder explotar el paralelismo de una máquina es una tarea *muy complicada* debido a que es necesario lograr paralelizar una fracción muy grande del código.

5.2.2. El problema de la sección crítica

Supongamos que estamos desarrollando un sistema de información para un banco que consiste en manejar retiros y depósitos en diferentes sucursales para los cuentahabientes de la institución. El cliente nos solicita que dicho sistema funcione en línea y que todas las sucursales sean capaces de realizar retiros y depósitos concurrentemente para cualquier cuenta en cualquier sucursal. La base

de datos donde se encuentra la información de las cuentas se encuentra centralizada en un solo equipo, por lo que todas las sucursales realizan operaciones sobre la misma base de datos, es decir, manejan *datos compartidos*.

Después de desarrollar el sistema, el banco nos notifica que existe un defecto en el mismo ya que los balances de las cuentas no coinciden, es decir:

$$d - r - s \neq 0,$$

donde d es el monto total de depósitos, r el monto total de retiros y s el saldo de la cuenta. Al examinar el código que implementa las operaciones de retiro y depósito encontramos algo similar a esto:

```
1  public class Cuentas {
2      private float[] cuentas;
3
4      public Cuentas(){
5          // Cargamos a memoria
6          // las cuentas de usuario
7          // desde disco.
8      }
9      /** Efectua un deposito en la cuenta especificada.
10      * @param monto El monto a depositar.
11      * @param cuenta Cuenta a depositar.
12      */
13     public void deposito(float monto, int cuenta){
14         cuentas[cuenta] += monto;
15     }
16
17     /** Efectua un retiro en la cuenta especificada.
18      * @param monto El monto a retirar.
19      * @param cuenta Cuenta a modificar.
20      * @return <code>true</code> en caso de que el retiro
21      * sea satisfactorio, <code>false</code> en caso de que la
22      * cuenta no tenga los fondos suficientes.
23      */
24     public boolean retiro(float monto, int cuenta){
25         if(cuentas[cuenta] - monto >= 0){
26             cuentas[cuenta] -= monto;
27             return true;
28         } else return false;
29     }
30
31     /* Devuelve el saldo de la cuenta especificada.
32      * @return saldo de la cuenta
33      */
34     public float saldo(int cuenta){
35         return cuentas[cuenta];
36     }
37 }
```

En este código, las únicas operaciones que pudieran causar una falla en la congruencia de las cuentas son las operaciones que realizan escrituras en la memoria compartida (retiros o depósitos), sin embargo, al examinar el código someramente parece ser correcto.

¿Qué estuvo mal? Si ejecutamos operaciones de depósitos o retiros de manera *secuencial* (es decir, una tras otra) este programa funcionará correctamente, sin embargo, si ejecutásemos dichas operaciones de manera concurrente, pueden ocurrir fallas: supongamos que tenemos dos sucursales S_1 que realiza un depósito y S_2 que realiza un retiro en la misma cuenta, la cual tiene un saldo inicial de $100 y las instrucciones se ejecutan en el orden que se muestra a continuación:

```
S1 ejecuta deposito(100, 1):        S2 ejecuta retiro(60, 1):

                                      // condicion if
r1 = cuentas[1] + 100                 r2 = cuentas[1] - 60
cuentas[1] = r1
// cuentas[1] = 200
                                      cuentas[1] = r2
                                      // cuentas[1] = 60
```

Los valores $r1$ y $r2$ representan registros del procesador donde se guarda temporalmente el valor de las operaciones intermedias antes de asignar a *cuentas*[1]. Como se puede ver, después de esta ejecución, el valor de *cuentas*[1] = 60, situación que evidencia que la correctud del sistema se ha violado.

La causa de este error es la *lectura invalidada*: si S_1 o S_2 se hubieran esperado a que la otra sucursal realizara la actualización de los datos, los valores de las cuentas serían congruentes, sin embargo, cuando S_2 realiza la escritura del retiro, dicho valor no se ha actualizado al depósito efectuado por la sucursal S_1.

Después de examinar este problema, podemos concluir que la solución es *restringir* la ejecución de la sección código de depósitos o retiros de tal manera que cada sucursal realice tales operaciones *una a la vez*. A este problema se le conoce como *sección crítica*, y se puede definir de la siguiente manera:

> Sea un sistema que consiste de n procesos $\{P_0, P_1, \ldots, P_{n-1}\}$ donde cada proceso P_i contiene una sección de código llama *sección crítica*, en la cual P_i realiza modificaciones en recursos compartidos (memoria, archivos, etcétera). La característica principal de este sistema es que mientras algún proceso P_i esté ejecutando su sección crítica, ningún proceso P_j $(j \neq i)$ esté ejecutando la sección crítica, esto es, no debe haber dos o más procesos ejecutando la sección crítica al mismo tiempo. Además, toda solución al problema de la sección crítica debe proporcionas las siguientes propiedades:
>
> 1. **Exclusión mutua.** Si P_i está ejecutando su sección crítica, ningún otro proceso P_j $(i \neq j)$ debe ejecutar su sección crítica.
> 2. **Progreso o viveza.** Si no hay ningún proceso que esté ejecutando su sección crítica, pero existen procesos que solicitan ejecutarla, sólo éstos pueden participar en la decisión de quién la ejecuta y además esta decisión no se puede posponer indefinidamente, es decir, debe haber uno que logre entrar.

3. **Espera acotada.** Debe existir una cota respecto al número de veces que los procesos se les permite ingresar a su sección crítica posterior a una solicitud por parte de un proceso a acceder a su sección crítica y antes de que dicha solicitud se haya permitido. Es decir, si un proceso solicita acceso a su sección crítica, ésta debe ser permitida en un tiempo acotado, evitando que el proceso se quede indefinidamente esperando a que se le permita el acceso.

5.3. Candados

La solución clásica al problema de la sección crítica es el candado o *lock*. El protocolo para el uso del mecanismo de bloqueo se realiza de la siguiente manera:

1. **Obtener el candado**. El candado debe ser un recurso compartido entre todos los hilos de ejecución que van a realizar operaciones en ciertos recursos compartidos que deben ser sincronizados por el mecanismo de bloqueo. En caso de que un hilo T_1 haya obtenido el candado anteriormente, al momento de que un hilo T_2 intenta obtener el candado, su ejecución se mantendrá *bloqueada* hasta que T_1 libere el candado.

2. **Ejecutar sección crítica**.

3. **Liberar el candado**. Una vez que ha finalizado la ejecución de la sección crítica, el candado se debe liberar para permitir que otros hilos de ejecución puedan ingresar a su sección crítica.

4. **Ejecutar código restante**. Todo código que no necesite mecanismo de sincronización se puede ejecutar posteriormente a la liberación del candado.

En código, el uso del mecanismo de bloqueo se puede resumir como:

```
1  public class MyClass {
2          private Lock lock;
3          //...
4          public Object operation(Object[] args){
5                  lock.lock();
6                  criticalSection();
7                  lock.release();
8                  remainderSection();
9          }
10         //...
11 }
```

5.3.1. El candado de Peterson

Una solución elegante en software para implementar el mecanismo de bloqueo es la propuesta por Gary Peterson en 1981. Esta solución sólo contempla la sincronización de dos procesos y se implementa puramente en software sin asumir ninguna estructura de hardware adicional:

```
1  public interface Lock {
2          /* Intenta adquirir el candado.
3           * @param id El identificador del hilo
4           * que solicita el candado.
5           */
6          public void lock(int id);
7          /* Libera el candado.
8           * @param id El identificador del hilo
9           * que solicita el candado.
10          */
11         public void release(int id);
12 }
13
14 public class PetersonLock implements Lock {
15         private volatile boolean flag[];
16         private volatile int turn;
17
18         public PetersonLock(){
19                 flag = new boolean[2];
20                 flag[0] = flag[1] = false;
21                 turn = 0;
22         }
23
24         public void lock(int id){
25                 flag[id] = true;
26                 int j = 1 - id;
27                 turn = j;
28                 while(flag[j] && turn == j);
29         }
30
31         public void release(int id){
32                 flag[id] = false;
33         }
```

La palabra clave `volatile` de Java especifica que las variables que son descritas con ese modificador accedan directo a memoria sin pasar por caché; es decir, al momento de leer o escribir una variable de tipo `volatile`, el caché siempre será invalidado para modificar directamente el valor en la memoria. La razón del uso de `volatile` en esta implementación se debe a que las variables a consultar se van a hacer por parte de diversos procesadores concurrentemente y los cachés no deben guardar copias de dichas variables ya que otro hilo puede modificarlas asíncronamente y causar una inconsistencia de datos en los hilos cuyos valores hayan sido invalidados.

El arreglo `flag` representa una bandera para cada hilo. Cuando un hilo quiere obtener el candado, levanta su bandera para indicarlo. La variable `turn` representa a qué hilo le toca entrar a su sección crítica (ya sea cero o uno). Examinando el código podemos explicar la operación de adquirir candado como:

1. Levanto mi bandera.
2. Cedo el lugar al otro hilo.
3. Mientras el otro hilo tenga su bandera levantada y sea su turno, espero.

Para liberar el candado simplementa basta con bajar la bandera, asegurando que el otro hilo pueda ingresar, ya que la condición en `while` se hace falsa.

Si bien el algoritmo de Peterson implementa el candado para dos hilos de ejecución, debemos demostrar que este algoritmo cumple las propiedades del problema de la sección crítica para asegurar correctud:

1. **Exclusión mutua.** Por contradicción: Sean los hilos P y Q con identificadores cero y uno respectivamente. Si suponemos que se viola la exclusión mutua, significa que ambos hilos ingresaron a su sección crítica, por lo tanto se debe satisfacer al mismo tiempo que:
   a) $flag[0] = flag[1] = true$ Es claro que esto puede cumplirse ya que `flag[0]` y `flag[1]` son variables separadas.
   b) P ve $turn = 0$ y Q ve $turn = 1$. En este caso, para que P pueda ver $turn = 0$ es necesario que Q haya sido el último hilo en escribir `turn`, si Q fue el último hilo en escribir `turn` entonces no es posible que Q vea $turn = 1$ y análogamente para el caso de que P haya sido el último hilo en escribir `turn`, por lo que caemos en una contradicción.
2. **Progreso.** Como se ve en el código, en caso de que un hilo no requiera el uso del candado, la condición `while` evalúa a `false` inmediatamente, ya que la bandera del otro hilo está abajo. Ahora, en caso de que los dos hilos estén requiriendo el candado, por la demostración anterior se hace evidente que solo una ingresa pero no ninguna, demostrando así que el progreso se preserva.
3. **Espera acotada.** La espera de un hilo es a lo más el tiempo que tarda el otro en ejecutar su sección crítica, ya que al momento de finalizar, y liberar el candado, la bandera del otro hilo se ha bajado, permitiendo al primero ingresar inmediatamente después de la liberación del candado.

En la práctica el algoritmo de Peterson es poco utilizado ya que las nuevas arquitecturas de computadoras proporcionan operaciones nativas de sincronización de memoria compartida eficientando su uso y minimizando la dificultad de implementación de mecanismos de sincronización en software.

5.4. Primitivas de sincronización por hardware

Como mencionamos anteriormente, existen algunas operaciones de hardware que nos facilitan la programación de operaciones de sincronización entre diversos hilos de ejecución o procesos. Todas estas operaciones consisten en el concepto

de leer y modificar una localidad de memoria de manera *atómica*, es decir, sin que otra instrucción pueda traslaparse en medio de dicha operación. Tales operaciones se muestran a continuación:

```
1  // RMW: Read-Modify-Write
2  // (Lee-Modifica-Escribe)
3  public class HardwareRMW {
4          private int v;
5
6          public HardwareRMW(int v){
7                  this.v = v;
8          }
9
10         public int get(){
11                 return v;
12         }
13
14         public void set(int v) {
15                 this.v = v;
16         }
17
18         public int getAndSet(int v){
19                 int old = this.get();
20                 this.set(v);
21         }
22
23         public void getAndIncrement(){
24                 return this.v++;
25         }
26
27         public void getAndDecrement(){
28                 return this.v--;
29         }
30
31         public void swap(HardwareRMW r){
32                 int temp = this.get();
33                 this.set(r.get());
34                 r.set(temp);
35         }
36
37         public boolean compareAndSet(int v, int e){
38                 if(this.get() == e){
39                         this.set(v);
40                         return true;
41                 } else return false;
42         }
43 }
```

1. Las operaciones `get` y `set` mantienen su significado tradicional.

2. `getAndSet`, `getAndIncrement` y `getAndDecrement`. Establece el valor de una región de memoria y devuelve su valor anterior de manera atómica.

3. `swap`. Intercambia atómicamente dos valores en memoria.

4. `compareAndSet`. Recibe dos argumentos: el valor a establecer y un valor esperado. Si el valor de la sección de memoria a modificar es igual al valor esperado, entonces se procede a sustituirla por el valor dado como argumento, en caso contrario se deja sin modificar. Esta instrucción se ejecuta atómicamente al igual que las anteriores.

5.5. Semáforos

El semáforo, desarrollado por el científico danés Edsger W. Dijkstra, es una de las primeras herramientas que se desarrollaron con la finalidad de resolver problemas de sincronización de procesos. Al igual que el concepto de candado, los semáforos consisten de dos operaciones: obtener (*acquire*) y liberar (*release*). La forma más sencilla de comprender el funcionamiento de un semáforo es a través de una implementación sencilla del mismo.

```
1  public class Semaphore {
2          private HardwareRMW value;
3
4          public Semaphore(int value){
5                  this.value = new HardwareRMW(value);
6          }
7
8          public void acquire(){
9                  while(value.get() <= 0);
10                 value.getAndDecrement();
11         }
12
13         public void release(){
14                 value.getAndIncrement();
15         }
16 }
```

Al igual que en los candados, los semáforos bloquean al hilo que ejecuta la operación *acquire* en caso de que el conteo de `value` se haya ido a cero, esperando a que otro hilo libere el semáforo e incremente el conteo a un valor mayor a cero. Los semáforos se clasifican en dos grupos importantes:

1. **Semáforos de conteo**: Aquellos que se utilizan para medir condiciones frontera y sirven para contar el número de accesos a cierto recurso. Generalmente se inicializan con un valor mayor que uno.

2. **Semáforos binarios o candados *mutex***. Un semáforo inicializado en uno se comporta como un candado de exclusión mutua como se ve en el apartado siguiente. Se le llama semáforo binario porque sólo puede tener dos valores: cero o uno.

5.5.1. Problema de la sección crítica con semáforos

Podemos implementar la solución del problema de la sección crítica por medio de semáforos de manera sencilla y, a diferencia del algoritmo de Peterson,

para n hilos de ejecución de la siguiente manera:

```
1  /** La implementacion de cada hilo
2   * de ejecucion
3   */
4  public class MyThread extends Thread {
5      private int id;
6      private Semaphore s;
7
8      public MyThread(Semaphore s, int id){
9          super("Thread: " + id);
10         this.s = s;
11         this.id = id;
12     }
13
14     public void run(){
15         s.acquire();
16         criticalSection();
17         s.release();
18         remainderSection();
19     }
20
21     private void criticalSection(){
22         // Seccion critica
23     }
24
25     private void remainderSection(){
26         // Seccion restante
27     }
28 }
29
30 /** Clase que ejecuta los hilos de
31  * control.
32  */
33 public class Main {
34     public static void main(String[] args){
35         Thread[] threads = new Thread[5];
36         Semaphore mutex = new Semaphore(1);
37
38         // Creamos los hilos y compartimos
39         // el semaforo entre ellos.
40         for(int i = 0; i < threads.length; ++i){
41             threads[i] = new MyThread(mutex, i);
42         }
43
44         // Iniciamos la ejecucion de los hilos.
45         for(int i = 0; i < threads.length; ++i)
46             threads[i].start();
47     }
48
49 }
```

5.5.2. Implementación práctica

El código del semáforo propuesto al inicio de este apartado adolesce de un problema: la *espera ocupada* o *busy waiting*. El ejecutar indefinidamente el `while`

que está preguntando si el valor almacenado en el semáforo es positivo se conoce como espera ocupada ya que el procesador está ejecutando código repetidamente al momento de realizar la pregunta. Esta situación es indeseable ya que estamos desperdiciando tiempo de procesador al estar preguntando indefinidiamente y en sistemas con un solo procesador esto representa una pérdida de tiempo *cada vez* que se ejecuta el hilo de control que está bloqueado.

Una forma de mejorar la implementación del semáforo es, en caso de que el hilo de control tenga que esperar porque el valor asociado del semáforo no es positivo, solicitar al calendarizador del sistema operativo que difiera la ejecución del hilo, sacándolo de contexto. En Java, dicha implementación es sencilla y quedaría como sigue:

```
1  // AtomicInteger nos ofrece la posibilidad
2  // de poder utilizar las operaciones RMW
3  // de hardware.
4  import java.util.concurrent.atomic.AtomicInteger;
5
6  public class Semaphore {
7      private AtomicInteger value;
8      public Semaphore(int value){
9          this.value = new AtomicInteger(value);
10     }
11
12     public void acquire(){
13         while(value.get() <= 0){
14                 // Solicitamos al calendarizador
15                 // de hilos de Java que se difiera
16                 // nuestra ejecucion. Mientras,
17                 // otros hilos se pueden ejecutar.
18                 Thread.yield();
19         }
20         value.decrementAndGet();
21     }
22
23     public void release(){
24         value.incrementAndGet();
25     }
26 }
```

5.5.3. Abrazo mortal (*deadlock*) y hambruna (*starvation*)

Al momento de usar candados o semáforos, existen dos efectos secundarios que pueden producirse debido a un error de programación o un mal diseño de un algoritmo o protocolo de sincronizacion:

- **Abrazo mortal o *deadlock*.** Es una situación causada por la espera circular de la liberación de recursos compartidos en un escenario de ejecución concurrente. En general, decimos que un conjunto S de hilos de control se encuentra en estado de abrazo mortal o *deadlock* cuando para todo hilo $t \in S$ sucede que t está esperando la liberación de un recurso el cual está siendo bloqueado por $s \in S$, $s \neq t$. Podemos ejemplificar esta

situación en términos de la ejecución de dos hilos de control P y Q como sigue:

```
       P                Q
lock1.lock();       lock2.lock();
lock2.lock();       lock1.lock();
   ...                 ...
lock1.release();    lock2.release();
lock2.release();    lock1.release();
```

En este escenario, P está esperando a que Q libere el candado `lock2` mientras que Q está esperando a que P libere el candado `lock1`, situación que causará una espera indefinida entre ambos hilos de control.

- **Hambruna o *starvation***. Esta situación se da cuando algún hilo de control o proceso nunca logra ingresar a su sección crítica debido a esperar indefinidamente por la liberación de un recurso. Si bien los demás hilos pueden mostrar progreso, éste jamás progresa y por ende se dice que cae en hambruna.

5.6. Monitores

En la práctica, las implementaciones de candados o *locks* son sonfisticadas y ofrecen una gama de servicios más amplias que sólo las operaciones *lock* y *unlock* (análogas al *acquire* y *release* en un semáforo binario). En el caso de Java, la interfaz `java.util.concurrent.atomic.Lock` está definida como sigue:

```
1 public interface Lock {
2         void lock();
3         void lockInterruptibly() throws InterruptedException;
4         boolean tryLock();
5         boolean tryLock(long time, TimeUnit unit);
6         Condition newCondition();
7         void unlock();
8 }
```

Dichas operaciones se pueden describir de la siguiente manera:

Operaciones *lock* y *unlock* Mantienen el mismo significado que el candado tradicional.

Operacion *lockInterruptibly* Realiza la operación de bloqueo soportando la posibilidad de interrumpir el hilo de control que llama mientras obtiene el bloqueo.

Operación *tryLock()* Obtiene el candado sólo en caso de que el candado se encuentre libre.

Operación *tryLock(long time, TimeUnit unit)* Obtiene el candado en caso de que el candado se libere en el tiempo especificado.

Operación *newCondition* Es un método tipo fábrica el cual crea y devuelve un objeto de tipo condición asociado con el candado. Este objeto es la unidad básica para la implementación de monitores como se verá posteriormente.

5.6.1. Variables de condición

Regresando al problema del buffer acotado, si asociamos el buffer con un candado *mutex*, en caso de que algún hilo esté esperando a que suceda alguna condición (ya sea que el buffer deje de estar vacío o deje de estar lleno, por ejemplo), es una buena idea liberar el candado por parte del hilo de control que se encuentra esperando, ya que en caso contrario cualquier otro hilo que pueda hacer verdadera la condición (insertar o eliminar un elemento del buffer) nunca podrá ingresar a su sección crítica, ya que el candado se encuentra poseído por el hilo que espera.

Los objetos tipo `Condition` obtenidos al ejecutar el método `newCondition` sirven precisamente para estos escenarios: en caso de que un hilo de ejecución haya obtenido un candado, es posible liberarlo de manera temporal por medio de una variable `Condition` asociada a dicho candado. En caso de que el hilo de control que posee un candado l llama el método `await()` de una condición obtenida a partir de `l.newCondition()`, esto causará que el hilo libere l y suspenda su ejecución, permitiendo que otros hilos puedan obtener ese candado y puedan modificar la condición que estamos esperando. Al momento que el hilo que espera regresar de la llamada a `await()` regresa, vuelve a obtener el candado l y continúa su ejecución.

El patrón que se sigue para el manejo de variables de condición y candados con soporte de condiciones se ilustra a continuación:

```
1  // ...
2  Condition condition = mutex.newCondition();
3  // ...
4  mutex.lock();  // Obtenemos el candado
5  try {
6          // property representa una propiedad
7          // que estamos esperando a que
8          // suceda para continuar la ejecucion
9          // del hilo de control.
10         while(!property)
11                 // Esperamos a que alguien nos
12                 // notifique que la propiedad
13                 // se cumple.
14                 condition.await();
15         // Aqui va el codigo posterior al
16         // cumplimiento de la propiedad.
17 } catch(InterruptedException e){
18         // Codigo que se ejecuta en caso
19         // de que el hilo de control haya
20         // sido interrumpido.
21 } finally {
22         // Pase lo que pase, siempre deberemos
23         // liberar el candado.
```

```
24          mutex.unlock();
25      }
```

Para explicar el funcionamiento de una variable de condición, lo más sencillo es ver su interfaz y entender qué hace cada método.

```
1   public interface Condition {
2           public void await();
3           public boolean await(long time, TimeUnit unit);
4           public long awaitNanos(long nanosTimeout);
5           public void awaitUninterruptibly();
6           public boolean awaitUntil(Date deadline);
7           public void signal();
8           public void signalAll();
9   }
```

`await()` Cuando un hilo de ejecución llama este método, en caso de mantener el candado asociado con esta condición, dicho candado se libera y el hilo que llama se suspende, esperando a que se levante una señal de dicha condición o el hilo sea interrumpido. Este escenario ofrece la oportunidad a otro hilo de adquirir el candado. Cuando el hilo que llama despierta, readquiere el candado que se liberó al momento de la llamada.

`await(long time, TimeUnit unit)`, `awaitNanos`, y `awaitUntil` Análogo a la llamada anterior, sólo que la llamada tiene un tiempo de espera acotado por los argumentos dados.

`signal()` Despierta a *un solo* hilo de ejecución que se mantiene esperando debido a que invocó el método `await`.

`signalAll()` Despierta a *todos* los hilos de ejecución que esperan por haber invocado `await`.

La combinación de candados con soporte para variables de condición, los objetos de condición y los métodos que emplean estas llamadas se conocen como *monitores*.

Para ejemplificar el funcionamiento de un monitor, se muestra la implementación de una cola acotada *FIFO*, la cual tiene dos condiciones: *notEmpty* y *notFull* que representan cuando la cola no ha sido vaciada o llenada. En caso de que cualquier hilo haga una llamada a encolar y la cola esté llena, o a desencolar y la cola esté vacía, este último quedará en espera y liberará el candado de acceso a la cola para permitir que otro hilo proporcione el escenario necesario para despertar el hilo que ha sido puesto en espera.

```
1   // Importamos el soporte de monitores de Java.
2   import java.util.concurrent.locks.ReentrantLock;
3   import java.util.concurrent.locks.Condition;
4   import java.util.concurrent.locks.Lock;
5
6   /** Clase que implementa el buffer acotado por medio de
7    *  monitores.
```

```
8   */
9   public class LockedQueue<T> {
10      // El candado asociado al buffer.
11      final Lock lock = new ReentrantLock();
12      // La condicion correspondiente a cuando el buffer
13      // no esta lleno.
14      final Condition notFull = lock.newCondition();
15      // La condicion correspondiente a cuando el buffer
16      // no esta vacio.
17      final Condition notEmpty = lock.newCondition();
18      final T[] items;
19      int tail, head, count;
20
21      /** Crea un nuevo buffer acotado con la capacidad
22       *  especificada.
23       * @param capacity Capacidad del buffer.
24       */
25      public LockedQueue(int capacity){
26          items = (T[]) new Object[capacity];
27          tail = head = count = 0;
28      }
29
30      public void enq(T o) throws InterruptedException {
31          // Bloqueamos para tener acceso exclusivo.
32          lock.lock();
33          try {
34              // En caso de que la cola este llena
35              // esperamos a la condicion notFull.
36              while(count == items.length)
37                  notFull.await();
38              items[tail] = o;
39              if(++tail == items.length)
40                  tail = 0;
41              ++count;
42              // Como la cola no puede estar vacia
43              // tras agregar un elemento, despertamos
44              // un hilo que se quedo en espera porque
45              // la cola estaba vacia.
46              notEmpty.signal();
47          } finally {
48              lock.unlock();
49          }
50      }
51
52      public T deq() throws InterruptedException {
53          // Acceso exclusivo.
54          lock.lock();
55          try {
56              // En caso de que la cola este vacia
57              // esperamos a la condicion notEmpty.
58              while(count == 0)
59                  notEmpty.await();
60              T x = items[head];
61              if(++head == items.length)
62                  head = 0;
63              --count;
64              // Al momento de quitar un elemento de
```

```
65              // la cola, despertamos un hilo que
66              // espera ya que la cola estaba llena.
67              notFull.signal();
68              return x;
69          } finally {
70              lock.unlock();
71          }
72      }
73  }
```

5.6.2. El problema de la señal perdida

Así como el mecanismo de bloqueo de hilos (ya sea por medio de candados o semáforos) sufren del riesgo de caer en abrazo mortal o *deadlock*, los objetos de condición son vulnerables al problema de la señal perdida (*lost wakeup*) el cual consiste en que uno o más hilos esperan indefinidamente por una condición sin darse cuenta que ya se ha cumplido.

Para ejemplificar este problema, supongamos que agregamos la siguiente "optimización" a nuestra implementación del buffer acotado: en vez de ejecutar el método `signal` cada que agregamos un elemento al buffer, ¿no bastaría con solo mandar una señal cada que el buffer pasa de estar vacío a no vacío?

```
1   public void enq(T o) throws InterruptedException {
2           lock.lock();
3           try {
4                   while(count == items.length)
5                           notFull.await();
6                   items[tail] = o;
7                   if(++tail == items.length)
8                           tail = 0;
9                   ++count;
10                  // Nueva optimizacion:
11                  // Solo mandamos senal cuando la cola
12                  // pasa de estar vacia a no vacia.
13                  if(count == 1)
14                          notEmpty.signal();
15          } finally {
16                  lock.unlock();
17          }
18  }
```

La respuesta es *no*. Y la razón es porque si tenemos más de dos hilos de ejecución consumiendo datos del buffer, sólo uno de ellos será notificado al momento que el buffer pasa de estar vacío a no vacío, por lo que todos los hilos que no hayan sido notificados siguen esperando a que la condición se cumpla siendo que ya se cumplió, solo que no recibieron notificación. De este modo, todos los hilos que no fueron notificados del cambio de condición quedarán esperando indefinidamente, una situación análoga al abrazo mortal.

Si bien el análisis cuidadoso del código es indispensable para evitar esta situación, existen algunas prácticas que pueden ser de utilidad para evitar el problema de la señal perdida:

- Siempre utilizar el método `signalAll` para despertar a todos los hilos que esperan una condición, no nada más uno.
- Especificar un tiempo de espera acotado (*timeout*) al momento de utilizar `await`.

5.7. Problemas clásicos de sincronización

A continuación se presentan tres problemas importantes de sincronización que han sido objeto de grandes esfuerzos en investigación y que representan los retos más comunes en sistemas concurrentes:

1. **Problema del buffer acotado.** Es un problema común en situaciones donde un proceso produce datos y otro proceso necesita consumirlos.
2. **Problema de lectores y escritores.** El problema clásico de los sistemas manejadores de bases de datos. Consiste en bloquear secciones de datos de tal manera que al escribir haya total exclusión mutua, pero al leer sólo se excluyan aquellos hilos que quieran escribir, pero no aquellos que lean.
3. **El problema de los filósofos comensales.** Supongamos que tenemos cinco filósofos en una mesa redonda con cinco sillas, cinco palillos chinos sobre la mesa y un platón de arroz en el centro. Los filósofos, naturalmente, se dedican a dos actividades: pensar o comer. Cuando un filósofo tiene hambre, intenta tomar los palillos que se encuetran a sus lados y empieza a comer del platón de arroz, sin embargo, existe una condición importante respecto a los palillos: cada filósofo puede tomar solo un palillo a la vez y no puede tomar un palillo que ha sido tomado por algún filósofo sentado al lado de él. En caso de que el filósofo logre sostener ambos palillos, procede a comer y al finalizar, deja los palillos sobre la mesa permitiendo que los demás filósofos puedan comer en caso de tener hambre. El problema consiste en diseñar un protocolo de sincronización de los filósofos de tal manera que se satisfaga la exclusión mutua (es decir, no pueden tener dos filósofos un palillo a la vez) y que ningún filósofo muera de hambre.

Las soluciones presentadas en este apartado resuelven el problema y además aseguran estar libres de abrazo mortal; sin embargo, no se asegura la posibilidad de hambruna. Las soluciones libres de hambruna para estos problemas se dejan como ejercicio al lector interesado.

5.7.1. Problema del *buffer* acotado

El problema del buffer acotado consiste en lo siguiente:

- **Memoria compartida acotada.** Se le llama el problema del buffer acotado porque tenemos una región de memoria compartida de tamaño acotado la cual compartiremos entre dos hilos de control.

- **Hilos *consumidor* y *productor***. Donde el hilo consumidor va consumiendo datos del buffer y el hilo productor va insertando datos en el mismo. Podemos ver al buffer como una cola acotada.
- **Condiciones frontera.** En caso de que el buffer esté vacío, el consumidor se debe bloquear y no leer nada hasta que el productor haya colocado algo en el buffer; análogamente, en caso de que el buffer esté lleno, el productor se debe bloquear y no escribir hasta que el consumidor haya consumido datos en el mismo.

Podemos resolver el problema del buffer acotado por medio de semáforos encapsulando el buffer acotado de la siguiente manera:

```
1  public interface Buffer<T> {
2      public void insert(T item);
3      public T remove();
4  }
5
6  public class BoundedBuffer<T> implements Buffer<T> {
7      private static final int BUFFER_SIZE = 5;
8      private Object[] buffer;
9      private int in, out;
10     private Semaphore mutex, full, empty;
11
12     public BoundedBuffer(){
13         // Buffer inicialmente vacio.
14         in = out = 0;
15         buffer = new Object[BUFFER_SIZE];
16         mutex = new Semaphore(1);
17         full = new Semaphore(BUFFER_SIZE);
18         empty = new Semaphore(0);
19     }
20
21     public T remove(){
22         empty.acquire();
23         mutex.acquire();
24
25         T item = (T) buffer[out];
26         out = (out + 1) % BUFFER_SIZE;
27
28         mutex.release();
29         full.release();
30
31         return item;
32     }
33
34     public void insert(T item){
35         full.acquire();
36         mutex.acquire();
37
38         buffer[in] = item;
39         in = (in + 1) % BUFFER_SIZE;
40
41         mutex.release();
42         empty.release();
43     }
```

```
44 }
```

En esta implementación estamos utilizando tres semáforos (dos de conteo y un candado *mutex*):

- Semáforos `full` y `empty`: Representan los casos en los que el buffer se ha llenado o vaciado respectivamente. En caso de que el buffer esté lleno, al momento de invocar *acquire* en el semáforo `full` el hilo de control se bloqueará hasta que otro hilo invoque *release* y denote que el buffer ya no está lleno. El comportamiento es análogo para el semáforo de buffer vacío `empty`. Mientras `empty` lleva una cuenta de cuántas veces se ha insertado en el buffer, `full` lleva la cuenta regresiva de cuántas entradas quedan disponibles en el buffer.
- El semáforo `mutex`. Se utiliza como candado para marcar la sección crítica de las operaciones de inserción y remoción en el buffer. Mantiene la exclusión mutua al momento de modificar el buffer y las referencias de posición para insertar y remover.

Ahora, la implementación del productor y consumidor quedaría como sigue:

```
1  public class Consumer implements Runnable {
2      private Buffer<Integer> buffer;
3
4      public Consumer(Buffer<Integer> b){
5          buffer = b;
6      }
7
8      public void run(){
9          while(true){
10             // Dormimos el thread entre
11             // cero y dos segundos
12             try {
13                 Thread.sleep((int) (Math.random() * 2000));
14             } catch(InterruptedException e){ }
15             System.out.print(buffer.remove());
16             System.out.println(" has been consumed.");
17         }
18     }
19 }
```

```
1  public class Producer implements Runnable {
2      private Buffer<Integer> buffer;
3
4      public Producer(Buffer<Integer> b){
5          buffer = b;
6      }
7
8      public void run(){
9          while(true){
10             // Dormimos el thread entre
11             // cero y dos segundos
12             try {
```

```
13                  Thread.sleep(((int) (Math.random() * 2000)));
14              } catch(InterruptedException e){}
15              int val = (int) (Math.random() * 100);
16              buffer.insert(val);
17              System.out.println("Producing: " + val);
18          }
19      }
20  }
```

Finalmente, para iniciar la ejecución del consumidor y productor, necesitamos un hilo de control principal que cree el buffer acotado, genere el productor y consumidor, asigne el buffer compartido a ambos hilos e inicie su ejecución:

```
1   public class Main {
2       public static void main(String[] args){
3           Buffer buffer = new BoundedBuffer();
4           Thread producer = new Thread(new Producer(buffer));
5           Thread consumer = new Thread(new Consumer(buffer));
6           // Iniciamos productor y consumidor.
7           producer.start();
8           consumer.start();
9       }
10  }
```

5.7.2. Problema de Lectores-Escritores

Supongamos que estamos compartiendo una base de datos entre varios hilos de control. Dichos hilos los podemos clasificar en dos grupos:

- **Hilos lectores.** Aquellos que sólo ejecutarán operaciones de lectura sobre la base de datos.

- **Hilos escritores.** Aquellos que realizarán operaciones de escritura sobre la base de datos, ya sean inserciones, eliminaciones o actualizaciones sobre los registros de la misma.

Es claro que si múltiples hilos lectores acceden de manera concurrente a la base de datos, no hay posibilidad de sufrir de algún problema de sincronización. El problema surge cuando algún hilo escritor quiere realizar alguna operación sobre la base de datos. En base a lo anterior, queremos implementar un protocolo de sincronización que asegure que los hilos escritores tengan acceso exclusivo a la base, es decir, cuando un hilo escritor intente escribir sobre la base de datos, *ningún otro hilo*, ya sea lector o escritor pueda tener acceso hasta que este escritor haya terminado de utilizar la base, pero que en caso de que ningún escritor esté haciendo uso de la base de datos, los lectores puedan leer sin necesidad de bloquearse entre ellos.

Una forma de resolver este problema es implementando un candado con cuatro tipos de llamadas:

1. **Obtener y liberar candado de lectura.** Utilizado por los lectores, permite que varios lectores adquieran el candado simultáneamente y puedan acceder a la base de datos, sin embargo, en caso de que un escritor requiera el candado de escritura, se bloqueará este tipo de candado y ningún lector podrá acceder a la base hasta que ningún escritor tenga el candado de escritura.

2. **Obtener y liberar candado de escritura.** Usado sólo por los escritores, asegura el acceso exclusivo a la base, es decir, ningún escritor o lector podrá ingresar a su sección crítica una vez que este candado haya sido obtenido por algún escritor.

Por lo tanto, la interfaz a implementar para el candado de lectores y escritores debe ser similar a ésta:

```
1  public interface RWLock {
2          public void acquireReadLock();
3          public void releaseReadLock();
4          public void aquireWriteLock();
5          public void releaseWriteLock();
6  }
```

La estructura general de un hilo de control lector puede representarse como sigue:

```
1  public class Reader implements Runnable {
2          private RWLock lock;
3
4          public Reader(RWLock lock){
5                  this.lock = lock;
6          }
7
8          public void run(){
9                  while(true){
10                         // Codigo de inicializacion del lector.
11                         lock.acquireReadLock();
12                         // Seccion critica la cual solo realiza
13                         // lecturas en la base de datos.
14                         lock.releaseReadLock();
15                         // Codigo restante.
16                 }
17         }
18 }
```

Para el caso de un hilo de control escritor, el código luciría como:

```
1  public class Writer implements Runnable {
2          private RWLock lock;
3
4          public Writer(RWLock lock){
5                  this.lock = lock;
6          }
7
8          public void run(){
```

```
9                   while(true){
10                          // Codigo de inicializacion del escritor.
11                          lock.acquireWriteLock();
12                          // Seccion critica la cual realiza lecturas
13                          // y escrituras en la base de datos.
14                          lock.releaseWriteLock();
15                          // Codigo restante.
16                  }
17          }
18  }
```

La solución del problema de los lectores y escritores se puede realizar a partir de semáforos y su implementación es muy sencilla, como se muestra a continuación:

```
1   public class MyRWLock implements RWLock {
2       private int readerCount;
3       private Semaphore mutex;
4       private Semaphore db;
5
6       public MyRWLock(){
7           readerCount = 0;
8           mutex = new Semaphore(1);
9           db = new Semaphore(1);
10      }
11
12      public void acquireWriteLock(){
13          db.acquire();
14      }
15
16      public void releaseWriteLock(){
17          db.release();
18      }
19
20      public void acquireReadLock(){
21          mutex.acquire();
22          ++readerCount;
23
24          // En caso de ser el primer lector,
25          // notificar a los demas que la base
26          // de datos esta siendo leida.
27          if(readerCount == 1)
28              db.acquire();
29
30          mutex.release();
31      }
32
33      public void releaseReadLock(){
34          mutex.acquire();
35          --readerCount;
36
37          // En caso de ya no haber lectores,
38          // notificar a los demas que la base
39          // de datos ya no esta siendo leida.
40          if(readerCount == 0)
41              db.release();
42
```

```
43            mutex.release();
44        }
45    }
```

El funcionamiento del candado se explica a continuación:

1. **Escritor.** Simplemente llaman al semáforo `db` que funciona como candado *mutex*. Cualquier llamada a `db.acquire()` bloqueará al hilo que invoca hasta que el escritor finalice la ejecución de su sección crítica.

2. **Lector.** Utiliza el semáforo `db` para obtener uso exclusivo de la base de datos y un segundo semáforo `mutex` que sirve para bloquear la variable `readerCount` la cual lleva el conteo de lectores que están utilizando la base de datos.

   Al momento de solicitar el candado de lector, basta con solicitar el candado `db` una sola vez para evitar que algún escritor acceda a la base de datos, mientras que las subsecuentes llamadas a `acquireReadLock` no realizarán ninguna acción sobre el semáforo `db`.

   En caso de liberar el candado de lector, sólo basta con evaluar si somos el último lector en turno, para liberar el semáforo `db`.

Este tipo de candado es de uso muy común en la práctica y generalmente se utiliza en las siguientes situaciones:

- Cuando es fácil discernir entre los hilos de control lectores y escritores.

- Cuando tenemos un número de lectores considerablemente mayor al número de escritores. Si bien tenemos una mejora en la concurrencia para el caso de los hilos lectores, es claro que también hay una sobrecarga en la implementación de este tipo de candado, por lo que debemos buscar balancear la sobrecarga maximizando la concurrencia del sistema.

5.7.3. El problema de los filósofos comensales

Como ya se mencionó al inicio de este apartado, el problema de los filósofos comensales representa un reto de sincronización entre recursos compartidos y diversos hilos de control. En este caso los hilos de control serían los filósofos, los cuales exponen un comportamiento asíncrono entre ellos y acceden a los datos compartidos asíncronamente también (los datos compartidos serían los palillos para comer). La solución de este problema se realiza por medio de semáforos.

Solución con semáforos

Podemos modelar a los palillos como un candado *mutex* (o semáforo binario) cada uno, así pues, nuestros datos compartidos serían:

```
1         Semaphore[] chopStick = new Semaphore[5];
2         for(int i = 0; i < chopStick.length; ++i){
3                 chopStick[i] = new Semaphore(1);
4         }
```

La solución que se puede ocurrir inmediatamente para resolver este problema sería:

> Para cada filósofo F_i bloqueamos los palillos `chopStick[i]` y `chopStick[(i+1) % 5]`, comemos y luego liberamos los candados.

Dicha solución quedaría implementada como:

```
1  while(true){
2         // obtengo los palillos de mi izquierda
3         // y derecha respectivamente.
4         chopStick[i].acquire();
5         chopStick[(i+1)%5].acquire();
6
7         eat();
8
9         // libero los dos palillos, permitiendo
10        // que otros filosofos puedan comer.
11        chopStick[i].release();
12        chopStick[(i+1)%5].release();
13
14        think();
15 }
```

Sin embargo esta solución es *incorrecta* y la razón es porque si ejecutamos este código concurrentemente para todos los filósofos resulta en un abrazo mortal ya que cada filósofo toma el palillo a su izquierda y espera indefinidamente a que el filósofo de su derecha libere el palillo que tomó el cual a su vez espera al filósofo a su derecha y así sucesivamente, provocando una situación de espera circular. A pesar de que esta solución es incorrecta porque nos lleva a una situación de abrazo mortal, ciertamente satisface la exclusión mutua.

Curiosamente, podemos reutilizar esta solución haciendo una pequeña modificación que resolverá el problema de abrazo mortal: implementar una solución asimétrica. Una forma de implementar una solución con esta característica es:

- En caso de ser filósofo con identificador par, intento tomar primero el palillo izquierdo y luego el derecho.
- En caso de ser filósofo con identificador impar, intento tomar primero el palillo derecho y luego el izquierdo.

De este modo evitamos la situación de espera circular Finalmente, la implementación para cada filósofo quedaría como:

```
1  public class Philosopher implements Runnable {
2      private Semaphore[] chopSticks;
3      private int id;
```

```
4
5       public Philosopher(Semaphore[] chopSticks, int id){
6           this.chopSticks = chopSticks;
7           this.id = id;
8       }
9
10      public void run(){
11          while(true){
12              int first, second;
13              // Si el filosofo es par toma primero el
14              // palillo a su izquierda y luego el
15              // palillo que esta a su derecha.
16              // En caso contrario, los toma en orden
17              // inverso.
18              if((id % 2) == 0){
19                  first = id;
20                  second = (id+1)%5;
21              } else {
22                  first = (id+1)%5;
23                  second = id;
24              }
25
26              // Bloqueo del semaforo de cada palillo.
27              chopSticks[first].acquire();
28              chopSticks[second].acquire();
29              System.out.println("Soy el filosofo " + id +
30                                 " y procedo a comer");
31
32              // El filosofo simula comer
33              // por medio de un retraso
34              // de entre 1 y 2 segundos.
35              try {
36                  Thread.sleep(1000 + ((int) Math.random() * 1000));
37              } catch(InterruptedException e){}
38
39              // Libera los semaforos de los palillos.
40              chopSticks[second].release();
41              chopSticks[first].release();
42              System.out.println("Soy el filosofo " + id +
43                                 " y he terminado de comer.");
44          }
45      }
46  }
```

5.8. Caso de estudio: soporte de concurrencia en Java

Después de haber visitado los temas centrales de programación concurrente, es momento de explorar una implementación. Seleccionamos a Java ya que soporta nativamente el manejo de concurrencia desde el lenguaje de programación, sin necesidad de incluir ninguna biblioteca de concurrencia. El soporte nativo de concurrencia de Java es a través de monitores, donde cada objeto en Java tiene asociado un candado y una variable de condición.

5.8.1. La palabra reservada `synchronized`

La palabra reservada `synchronized` de `Java` se utiliza para bloquear objectos, es decir, asignar un candado a un objeto específico. Existen dos maneras de utilizar `synchronized`:

- **Colocar `synchronized` en la declaración de un método.** El colocar la palabra `synchronized` en la definición de un método causa que el candado asociado a la instancia a la cual se invoca el método sea obtenido por el hilo que llama. Para ejemplificar, tomemos en cuenta el ejemplo del incio del capítulo: las cuentas bancarias.

```
1  public class Account {
2      private double balance;
3
4      public Account(){
5          balance = 0.0;
6      }
7
8      public synchronized void deposit(double amount){
9          balance += amount;
10     }
11
12     public synchronized void withdraw(double amount)
13         throws IllegalStateException {
14         if(balance - amount < 0)
15             throw new IllegalStateException("Fondos
                   insuficientes");
16         balance -= amount;
17     }
18
19     public synchronized double balance(){
20         return balance;
21     }
22 }
```

  Al momento de llamar `deposit`, `withdraw` o `balance` la instancia de `Account` quedará bloqueada y sólo un hilo de ejecución podrá ejecutar el cuerpo del método (el cual representa una sección crítica) a la vez.

- **Utilizar un bloque `synchronized`**. El encerrar una sección de código dentro de un bloque `synchronized` causa que se bloqueen los objetos especificados dentro del paréntesis a lo largo de la ejecución del bloque, por ejemplo:

```
1  /** Transfiere fondos de la cuenta actual
2  * a la cuenta especificada
3  * @param a La cuenta a donde se transfieren los fondos.
4  * @param amount El monto a transferir.
5  */
6  public synchronized void transfer(Account a, double amount){
7          // Incluimos a la cuenta a transferir
8          // en un bloque synchonized para mantener
9          // acceso exclusivo a ambas cuentas.
```

```
10          synchronized(a){
11                  if(balance < amount)
12                          throw new IllegalStateException("
                                Fondos insuficientes");
13                  balance -= amount;
14                  a.balance += amount;
15          }
16  }
```

5.8.2. Monitores por medio de los métodos `wait` y `notify`

Como se mencionó anteriormente, cualquier objeto de `Java` tiene asociada una variable de condición y un candado mutex. El uso de la palabra reservada `synchronized` es el mecanismo para poder hacer uso del candado mutex de cada objeto, sin embargo, para hacer uso de la variable de condición utilizamos los métodos `wait`, `notify` y `notifyAll`. Su semántica es la misma que los métodos `await`, `signal` y `signalAll` de los objetos `Condition` del paquete `java.util.concurrent.locks` mencionados en el capítulo de monitores.

Para ejemplificar el uso de las primitivas de sincronización de `Java`, mostramos la implementación de una cola concurrente.

```
1   /** Implementacion sencilla de una cola concurrente. */
2   public class ConcurrentQueue<T> {
3       // Usamos un contenedor para armar una lista
4       // doblemente ligada.
5       private final class Container<T> {
6           public Container next, prev;
7           public T o;
8
9           public Container(T o){
10              this.o = o;
11              next = prev = null;
12          }
13
14      }
15
16      // La cola concurrente tiene apuntadores
17      // a la cabeza y cola de la lista.
18      private Container<T> head, tail;
19
20      public ConcurrentQueue(){
21          // La lista es inicialmente vacia.
22          head = tail = null;
23      }
24
25      public synchronized void enq(T o){
26              // En caso de que la lista este
27              // vacia, la cabeza y cola son iguales.
28              if(head == null){
29                  head = new Container(o);
30                  tail = head;
31              } else {
32                  Container c = new Container(o);
33                  c.prev = tail;
```

```
34                  tail.next = c;
35                  tail = c;
36              }
37              // Notificamos a los hilos que estan
38              // esperando a que la cola deje de ser
39              // vacia.
40              notify();
41          }
42
43      public synchronized T deq() throws InterruptedException {
44              // Si la cola es vacia, espero.
45              while(head == null)
46                  wait();
47
48              T o = head.o;
49              head = head.next;
50              return o;
51          }
52  }
```

Capítulo 6

Ejercicios y proyectos de programación

Como se menciona en la introducción, una buena forma de mejorar la habilidad para desarrollar software es en base a la práctica. Este capítulo está dedicado a proponer un catálogo de proyectos de programación dirigidos a los estudiantes, así como recursos para obtener ejercicios y mantener al estudiante practicando regularmente a lo largo del curso.

6.1. Ejercicios y retos

Este apartado enlista dos recursos en la web que proporcionan ejercicios y retos de programación. La mayoría de ellos son problemas de competencia de nivel internacional y se espera que los estudiantes sean capaces de resolverlos en lapsos de entre 30 y 60 minutos por problema.

1. `http://acmicpc-live-archive.uva.es/nuevoportal/`

   El juez en línea de ACM. Contiene el catálogo de problemas de todos los concursos regionales y finales mundiales desde el año 2000. Consta de más de 300 problemas de programación de diferente dificultad.

2. `http://icpcres.ecs.baylor.edu/onlinejudge/`

   Juez en línea de la Universidad de Valladolid[1]. Cuenta con diversos catálogos de problemas:

   a) Catálogo de las finales mundiales del concurso de programación de ACM desde 1990 hasta el 2000.

   b) Catálogo de problemas del libro *Programming Challenges* de Steven S. Skiena y Miguel Revilla. Es un recurso muy utilizado para preparar

[1] `http://www.uva.es/`

estudiantes a los concursos de programación de ACM y la Olimpíada Internacional de Informática (IOI)[2].

3. Un catálogo propio con alrededor de 2000 problemas.

6.2. Información general de los proyectos

Los proyectos presentados en este apartado tienen como finalidad que el estudiante enfrente proyectos de programación de alcance más amplio que requiera un nivel mayor de esfuerzo y organización. Todos los proyectos presentan las siguientes características:

- Están pensados para ser implementados en un lenguaje de programación imperativo (ya sea procedural u orientado a objetos).
- Se espera que el código de los programas (sin tomar en cuenta comentarios, documentación y pruebas unitarias) sea de entre 1000 a 1500 líneas.
- Los conocimientos previos necesarios para implementar los proyectos están cubiertos por cursos de semestres anteriores en el plan de estudios. Los conceptos técnicos que pudieran no haber sido cubiertos en cursos anteriores se proporcionan en este documento.

6.2.1. Requisitos de entrega

Se recomienda al profesor o ayudante de laboratorio que los requisitos necesarios para la entrega de los proyectos sean al menos los siguientes:

- Código fuente comentado.
- Documentación técnica y de usuario.
- Conjunto de pruebas unitarias para la validación básica de la funcionalidad de los componentes del programa.
- Que la entrega del proyecto sea un mes después a partir de la fecha de presentación del mismo.
- La realización de los proyectos debe ser de manera individual o a lo más en parejas.
- Que los estudiantes realicen los proyectos utilizando al menos una de las técnicas de desarrollo mencionadas en este documento.

[2] *International Olympiad in Informatics*. `http://ioinformatics.org`

6.3. Proyecto 1: *Ray tracer*

6.3.1. Descripción

Un *ray tracer* es un componente de software que sintetiza imágenes en 3D. Recibe como entrada la especificación de una escena, la cual está compuesta por diversos objetos tridimensionales compuestos por modelos, texturas y materiales, así como la configuración de iluminación y posición de la cámara a partir de donde se visualiza la escena. La salida del programa es una imagen en archivo o memoria que representa la visualización de la cámara en la escena correspondiente.

En la actualidad, los *ray tracer* son de un uso intenso en la industria (arquitectura, diseño gráfico, publicidad, entretenimiento, entre otras) siendo el cine una aplicación muy socorrida para el uso de esta tecnología. Un ejemplo muy popular es *Renderman*[3], el *ray tracer* implementado por Pixar Animation Studios[4] el cual es el motor que ha producido diversos largometrajes animados para la compañía cinematográfica Disney[5].

Figura 6.1: Imagen producida por *Yafray*, un *ray tracer* libre.

El auge de la tecnología de gráficos y la industria del entretenimiento han promovido el desarrollo de diversas implementaciones del algoritmo de *ray tracing*, algunas de las más populares son:

- **Software propietario:**

[3] `http://renderman.pixar.com/`
[4] `http://www.pixar.com/`
[5] `http://disney.go.com/`

  - *Mental Ray*. La implementación usada por el popular sistema de producción y animación 3D *Maya*. Desarrollada por la compañía de software *Autodesk*[6].
  - *Renderman*. Desarrollado por Pixar Animation Studios.

- **Software libre:**
  - *Blender internal render*. El motor de síntesis incluído en la suite de producción 3D *Blender*[7].
  - *Yaf(a)ray*. `http://www.yafaray.org/`
  - *Lux Render*. `http://www.luxrender.net/`
  - *Indigo Renderer*. `http://www.indigorenderer.com/joomla/`

6.3.2. Arquitectura

Un sistema de *ray tracing* está compuesto por los siguientes componentes:

- **Módulo de lectura de datos.** Se encarga de cargar una escena a partir de un archivo. Inicializa y configura las estructuras de datos necesarias que consume el algoritmo de *ray tracing* para sintetizar una imagen.
- **Subsistema de álgebra lineal.** Su función es proveer la funcionalidad necesaria para el álgebra de vectores y operaciones con matrices.
- **El algoritmo de *ray tracing*.** Sintetiza la imagen.
- **Materiales.** Representan el material del que está hecho cada objeto. Existen diversos modelos matemáticos para simular materiales y gracias a la arquitectura modular de un *ray tracer* es fácil implementarlos como *plug-ins*.
- **Texturas.** Algoritmos que modifican el aspecto de la superficie de los objetos. Sirven para dar mayor realismo a los modelos tridimensionales. Al igual que los materiales, las texturas se pueden implementar como *plug-ins*. Existen dos grandes tipos de texturas:
  - **Texturas procedimentales.** Desarrolladas 100 % en código, utilizan procedimientos de software para simular diversos tipos de texturas.
  - **Texturas de imagen.** Utilizan un archivo de imagen de entrada y la mapean a un modelo por medio de alguna transformación.

[6] `http://usa.autodesk.com`
[7] `http://www.blender3d.org`

- **Módulos de síntesis paralela y distribuida.** El algoritmo de *ray tracing* consume una gran cantidad de recursos (especialmente tiempo de procesador), a causa de esto, la mayoría de los sistemas modernos proporcionan soporte para realizar síntesis en paralelo (por medio de varios hilos de ejecución en un equipo de cómputo) y síntesis distribuida, es decir, coordinar diversos equipos de cómputo conectados por medio de una red para realizar el trabajo de síntesis de manera conjunta.

6.3.3. El algoritmo de *ray tracing*

La técnica de *ray tracing* para sintetizar imágenes es un intento burdo de simular a la naturaleza. Como ya sabemos, lo que vemos a través de nuestros ojos son rayos de luz que se originan de alguna fuente (ya sea el sol o alguna lámpara, por ejemplo) y rebotan en nuestro entorno. Al final, la imagen que vemos, son un conjunto de rayos de luz que al rebotar inciden en nuestro ojo.

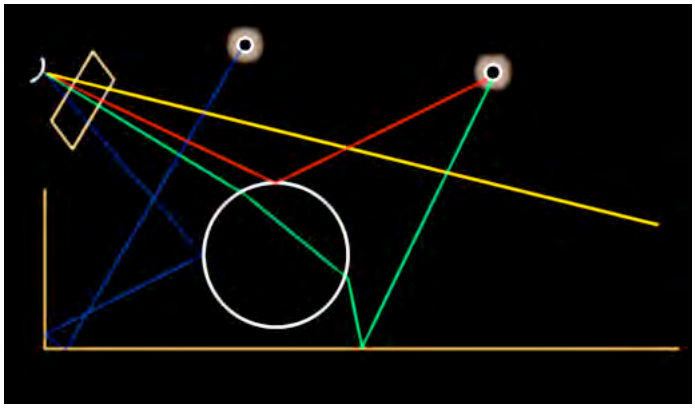

Figura 6.2: Diversos rayos de luz incidiendo en un espectador

Como se puede ver en la figura 6.3.3 los rayos de luz que proceden de una fuente de luz pueden incidir directa o indirectamente hacia nuestro ojo, posiblemente tras rebotar varias veces antes de llegar. En la ilustración se muestran diversos tipos de rayos:

- Rayos que rebotan en un objeto y posteriormente inciden en el ojo (rayo rojo).
- Rayos que se reflejan en una superficie hasta incidir en el ojo (rayo azul).
- Rayos que son desviados por la refracción al incidir en un objeto (rayo verde).

El rayo amarillo representa la incidencia de algún rayo de luz que escapa a la escena actual. Este tipo de rayo es importante a la hora de implementar el algoritmo de síntesis por razones que serán obvias posteriormente.

En la realidad, los rayos trazados en el esquema se originan en la fuente de luz y terminan en múltiples direcciones. En el caso de aquellos rayos que inciden en el espectador, la dirección de los rayos es hacia el espectador, sin embargo, para la imagen que queremos sintetizar sólo nos interesan aquellos rayos que inciden en el ojo, por lo que, podemos producir los rayos en sentido inverso: a partir del ojo hacia afuera y ver hacia donde se dirigen y calcular su iluminación total.

El algoritmo de síntesis por *ray tracing* intenta simular la ruta de los rayos de luz que inciden en una cámara virtual produciendo una cantidad finita de rayos los cuales *dispara* de un origen (el punto focal del observador) y los hace viajar por el espacio tridimiensional hasta intersectarlo con un objeto. La imagen resultante es el valor de color calculado por cada rayo que atraviesa por un plano (llamado *plano focal*).

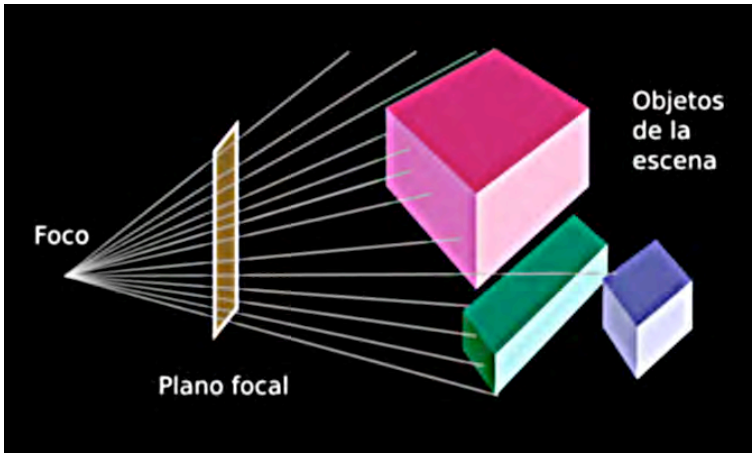

Figura 6.3: Diversos rayos generados a partir del punto focal y proyectados en el plano focal de la imagen

De este modo, podemos idear el siguiente algoritmo para producir la imagen que quedaría plasmada en el plano focal:

```
1  foreach ( pixel en el plano focal ) do {
2          # Construir rayo que parta del foco y pase por el pixel,
3          # es decir, su vector posicion sea origin y
4          # su direccion es pixel - origin.
5          Ray r = createRay(origin, pixel - origin);
6          # Encontrar el primer objeto que intersecta el rayo.
7          Primitive p = r.intersectFirst();
8          # Determinar el color en el punto de interseccion.
9          Color c = p.getColor();
10         # Guardar el color del pixel en la imagen.
11         image.setPixel(pixel.getPosition(), c);
12 }
```

Es algoritmo representa la idea central del algoritmo de *ray tracing*. El tipo de datos *Ray* representa una semirrecta o rayo, la cual se puede ver como una recta paramétrica en 3D con un parámetro t no negativo (para que sea semirrecta). El tipo de dato *Primitive* representa un objeto en el espacio tridimensional, el cual debe implementar el método *intersect* el cual devuelve la distancia de una recta paramétrica y el punto más cercano a ella. La función *intersectFirst* devuelve el objeto intersectado más cercano para el rayo dado, y a partir de ees objeto calculamos el color del pixel de la imagen.

Como veremos más adelante, el procedimiento del cálculo del color del píxel involucra muchos algoritmos intermedios los cuales pueden involucrar fenómenos ópticos como:

- **Reflexión.** Cuando un rayo rebota en una superficie especular (como un espejo).
- **Refracción.** Un rayo es desviado por cambiar de medio (objetos de cristal).
- **Textura.** Si un objeto tiene asociada una textura su color es afectado por la forma de la textura.
- **Iluminación.** Dependiendo de la configuración de las fuentes de luz en la escena, es como se debe "colorear" el objeto (ya sea número de fuentes, localización y color de las mismas).

6.3.4. Elementos soportados en el proyecto

Por razones de tiempo y complejidad del proyecto, limitaremos la implementación del algoritmo de *ray tracing* a los siguientes elementos[8]:

- **Objetos primitivos:**
  - Paralelepípedos coplanares a los planos $X-Y$, $Y-Z$ y $X-Z$.
  - Esferas.
  - Planos infinitos.
- **Elementos ópticos:**
  - Reflexión.
  - Modelo de iluminación de *Phong*. Sombreado difuso y sombreado especular.
  - Sobremuestreo (*supersampling*).
- **Fuentes de luz:**
  - Luz unidireccional con sombreado duro.
  - Luz puntual con sombreado duro.
  - Luz de superficie con sombreado suave basado en muestreo.

[8]Todos los elementos descritos se explican a detalle posteriormente

6.3.5. *Etapa 1:* Construcción del subsistema de álgebra lineal

Antes de iniciar el desarrollo del algoritmo de síntesis, lo primero que debemos implementar es el subsistema de álgebra lineal. Este sistema se encarga de realizar todos los cálculos necesarios en $\mathbb{R}^3$ que requeriremos a lo largo del desarrollo del proyecto.

El subsistema de álgebra lineal debe proporcionar las siguientes operaciones (todas en $\mathbb{R}^3$):

- **Aritmética de vectores:** Suma, diferencia, producto escalar, producto punto y producto cruz.
- **Matrices de transformación (de 3×3):** Suma, y producto de matrices (matriz por matriz y matriz por vector). Matrices de cambio de coordenadas (rotación y traslación).

Si bien es un módulo muy sencillo de implementar, es muy importante que esté bien probado para evitar problemas a futuro. No es necesario implementar ninguna otra operación mas las mencionadas en listado anterior.

6.3.6. *Etapa 2:* Construcción del módulo de lectura

Para implementar el componente que sintetiza imágenes, antes debemos tomar en cuenta el módulo que lee los datos de archivo y arma la estructura de datos que recibirá el *ray tracer* para sintetizar la imagen. Si bien existen diversas formas para codificar los datos de una escena en un archivo, por simplicidad, utilizaremos un formato de archivo de texto bien conocido: *XML*. Representaremos la escena a sintetizar como un documento *XML* que especificará de manera precisa la escena que recibirá el algoritmo de *ray tracing*.

En el archivo *XML* deberemos incluir los siguientes datos:

- Datos generales de la escena:
  - Tamaño de la imagen a sintetizar (largo por ancho en píxeles).
  - Número de muestras a tomar por píxel. Un valor entero positivo.
  - Posición y dirección de la cámara. Dos vectores en $\mathbb{R}^3$: uno que representa la posición, y un vector unitario que representa la dirección de la cámara.
  - Número de rebotes de los rayos a trazar (para implementar reflexión). Un valor entero.
- Definiciones de los materiales usados en la escena:

- Color. Como un vector *RGB* (*Red*, *Green*, *Blue*)[9] representado por aritmética de punto flotante acotado en el intervalo $[0, 1]$. Por ejemplo, un tono de gris sería: $< 0{,}5, 0{,}5, 0{,}5 >$ y el color blanco sería representado por $< 1{,}0, 1{,}0, 1{,}0 >$.
- Coeficiente de sombreado difuso. Un valor real en el intervalo $[0, 1]$. Representa qué tanta luz difusa refleja el objeto.
- Coeficiente de sombreado especular. Un valor real en el intervalo $[0, 1]$. Representa que tan "pulida" está la superficie del objeto.
- Dureza especular. Un valor entero en el intervalo $[20, 150]$. Representa que tanta difusión tiene el sombreado especular.
- Reflectividad. Un valor real en el intervalo $[0, 1]$. Representa qué tanta luz se refleja en este material al igual que en un espejo.

▪ Definiciones de fuentes de luz:

- Para todos los tipos de fuentes de luz:
  - Color de la fuente de luz. Como vector *RGB*.
  - Intensidad. Un coeficiente real que representa la intensidad de la fuente de luz.
- Para la luz unidireccional:
  - Vector dirección. Un vector en $\mathbb{R}^3$ que representa la dirección de la luz.
- Para la luz puntual:
  - Vector posición. Un vector en $\mathbb{R}^3$ que representa la posición de la fuente de luz.
- Para la luz de superficie:
  - Tamaño. Un valor real que representa el tamaño de la fuente de luz. La fuente de luz de superficie se modelará como un cuadrado en el espacio con la longitud por lado especificada en este parámetro.
  - Número de muestras a tomar. Un valor entero que representa cuántas muestras se tomarán por cada primitivo para calcular la sombra que proyecta sobre otro objeto.
- Objetos primitivos. Todos deberán hacer referencia al material del que "están hechos".
  - Planos.
    - Vector posición y vector normal. La descripción más popular de un plano. Ambos son vectores en $\mathbb{R}^3$.
  - Esferas.

[9] Rojo, Verde y Azul, los colores primarios luz. Mezclando tres fuentes de luz de estos colores es posible producir luz de cualquier color del espectro visible.

- ◇ Vector posición. Ubicación del centro de la esfera (vector en $\mathbb{R}^3$).
- ◇ Radio de la esfera. Representado como un valor real.

◦ Paralelepípedos.

- ◇ Posición de alguna esquina. Representada como un vector en $\mathbb{R}^3$.
- ◇ Ancho, alto y profundidad. Representados por un valor real para cada propiedad. Se puede ver como un vector en $\mathbb{R}^3$ que representa el desplazamiento del vector posición hacia la esquina opuesta del paralelepípedo.

Una propuesta para un archivo de entrada XML que almacena estos datos podría ser:

```
<scene width="800" height="600" samples="8" campos="0,-2,3"
        camdir="0,0.5,-1" raybounces="5">
        <materials>
                <material color="1.0,1.0,1.0" diffuse="0.6"
                        name="Material1" specular="0.1"
                        spechard="40" reflect="0.0" />
                <material color="0.5,0,0.8" diffuse="0.7"
                        name="Material2" specular="0.7"
                        spechard="20" reflect="0.1" />
                <material color="0.8,0.1,0.1" diffuse="0.9"
                        name="Material3" specular="0.3"
                        spechard="50" reflect="0.9" />
        </materials>
        <lights>
                <light color="1,1,0.7" pos="1,3,6"
                        dir="0,0,-1" intensity="1"
                        type="surface" name="Light1"
                        samples="4" size="1" />
                <light color="0.6,0.6,1.0" pos="1,-3,6"
                        dir="0,0,-1" intensity="0.5"
                        type="surface" name="Light2"
                        samples="8" size="2.2" />
        </lights>
        <primitives>
                <plane name="Plane1" pos="0,0,1"
                        normal="0,0,1" material="Material3" />
                <sphere radius="1.3" pos="-2,0,2"
                        name="Sphere1" material="Material2" />
                <sphere radius="2.4" pos="2,0,2"
                        name="Sphere2" Material="Material1" />
                <box color="0.5,0,0.8" pos="-0.5,1,1"
                        width="1" height="1" depth="1"
                        name="Box1" material="Material1" />
        </primitives>
</scene>
```

Ya que conocemos los datos que necesitamos y cómo representarlos en disco, podemos presentar qué estructuras de datos necesitamos para implementar el sistema de *ray tracing*:

- **Clase Vector.** Un vector en $\mathbb{R}^3$. Dicha clase debe estar implementada en el subsistema de álgebra lineal.

- **Clase Color.** En realidad es también un vector en $\mathbb{R}^3$ sólo que con valores acotados en el intervalo $[0, 1]$ para cada entrada. Adicionalmente, debemos implementar la operación *blend* que mezcla dos colores. Esto se consigue multiplicando entrada a entrada ambos colores y devolviendo el resultado. Es decir, si $C = (r_1, g_1, b_1)$ y $D = (r_2, g_2, b_2)$, entonces:

$$blend(C, D) = (r_1 r_2, g_1 g_2, b_1 b_2) \tag{6.1}$$

- **Clases contenedoras.** Deben almacenar los atributos de los diversos primitivos y fuentes de luz. Lo más recomendable es usar herencia de clases para las propiedades que se comparten entre los diversos tipos de primitivos y fuentes de luz.

- **Clase escena.** La estructura de datos que será la entrada del *ray tracer*. Esta clase deberá contener los atributos generales de la escena a sintetizar y tres colecciones (una para materiales, otra para primitivos y otra para fuentes de luz).

6.3.7. *Etapa 3:* Implementando el algoritmo

Después de leer la entrada y construir las estructuras de datos necesarias, estamos en condiciones de implementar el algoritmo de *ray tracing*. Como se menciona anteriormente, cada píxel de la imagen se calcula a partir del valor de color devuelto al intersectar cada recta paramétrica con el primitivo más cercano.

En una primera aproximación, podemos ignorar temporalmente la posición y dirección de la cámara y anclar nuestro espectador en $(0, 0, -5)$ y ubicar el plano focal en $(x, y, 0)$ con $x \in [-4, 4]$ y $y \in [-3, 3]$ para tener una relación de aspecto 4:3 como la mayoría de las pantallas actuales. En una etapa posterior podremos ajustar el algoritmo para que pueda soportar posición y dirección de la cámara, así como ajustar automáticamente la relación de aspecto en base a la resolución especificada en el XML. Para esta implementación, es importante que los archivos de entrada mantengan una resolución con la relación de aspecto 4:3, si no la imagen resultante se verá mal proporcionada.

Posteriormente, para armar los rayos a trazar, debemos construir una partición regular del plano focal cuyos puntos representan la posición por donde pasa cada rayo, por lo que cada rayo será de la forma:

$$\left[(0, 0, -5)\,, \frac{d}{||d||}\right]$$

donde el vector $d = (x, y, -5)$ y las variables x e y representan las coordenadas de cada punto de la partición del plano focal. Un método de `Java` para realizar este algoritmo sería:

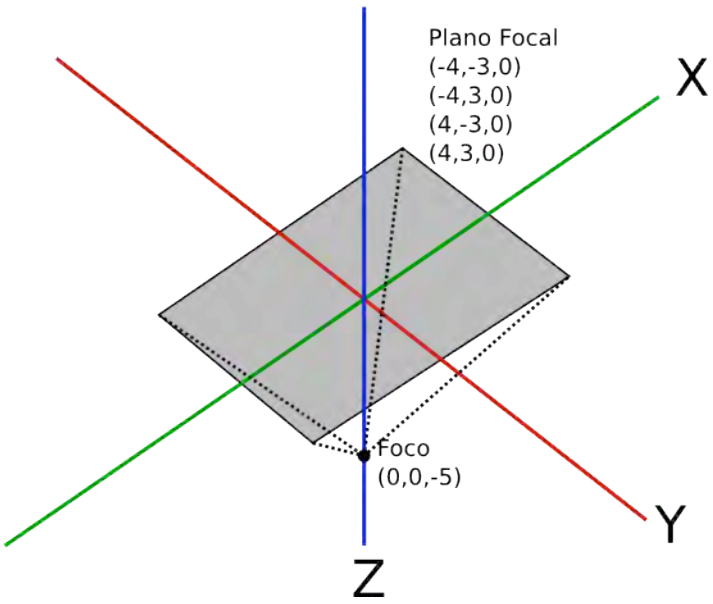

Figura 6.4: Anclando el plano focal en XY y el foco en $(0, 0, -5)$. Las líneas punteadas representan la dirección de los rayos que pasan por las esquinas del plano focal

```
1     /** Construye los rayos a trazar por el
2      * algoritmo de ray tracing.
3      * @param width Ancho de la imagen en pixeles.
4      * @param height Alto de la imagen en pixeles.
5      * tanto el ancho como el alto de la imagen
6      * deben mantener una relacion de aspecto 4:3
7      * para mantener la proporcion correcta.
8      * @return Un arreglo de rayos para ser
9      * trazado por el algoritmo de ray tracing.
10     */
11    public static Ray[] buildRay(int width, int height){
12        // El foco del espectador se encuentra en
13        // (0,0,-5)/
14        Vector3D origin = new Vector3D(0,0,-5);
15        // Construimos un vector para calcular la
16        // direccion de cada rayo.
17        Vector3D pos = new Vector3D();
18        Ray[] rays = new Ray[width*height];
19        for(int i = 0; i < height; ++i){
20            for(int j = 0; j < width; ++j){
21                // La posicion del plano focal
22                // hacia donde se dirige cada rayo
23                // se calcula parametrizando el
24                // intervalo ([-4,4], [-3,3], 0)
25                // en una particion regular de width*height.
26                pos.x = -4 + ((j + 1)*8.0 / width);
```

```
                pos.y = -3 + ((i+1)*6.0 / height);
                pos.z = -5;
                // Creamos el nuevo rayo, cuyo punto
                // de partida siempre es el origen y
                // la direccion hacia donde se dirige es
                // el vector que calculamos de la
                // particion del rectangulo.
                Ray r = new Ray(origin, pos.normalize());
                rays[i*width + j] = r;
            }
        }
        return rays;
    }
}
```

Al momento de desarrollar el *ray tracer* es recomendable que se construya cada rayo y se calcule el valor del pixel correspondiente y no como se propone en el método anterior, donde se producen todos los rayos y luego se procesan. La razón es por el elevado consumo de memoria que requiere guardar la lista de rayos, es más eficiente ir calculando el valor de color de cada rayo y liberar la memoria para ahorrar espacio.

Posteriormente, al ir procesando cada rayo, debemos encontrar la intersección más cercana con los objetos de la escena y calcular el color del punto de intersección. Utilizando un lenguaje orientado a objetos, es fácil generalizar este proceso por medio de herencia de clases definendo una clase abstracta que represente cualquier objeto de la escena:

```
public abstract class Primitive {
    /** El nombre del primitivo. */
    protected String name;
    /** El material asociado a este primitivo. */
    protected Material material;

    /** Calcula la distancia de interseccion del primitivo
     * con un rayo dado.
     * @param ray Rayo a intersectar.
     * @return Un <code>float</code> que representa
     * la distancia de interseccion.
     * En caso de no haber interseccion con el rayo,
     * el primitivo debe reportar el valor
     * <code>Double.POSITIVE_INFINITY</code>.
     */
    public abstract double intersect(Ray ray);

    /** Devuelve el vector normal a la
     * superficie del primitivo con respecto
     * a la posicion especificada.  Util para calcular
     * iluminacion.
     * @param pos Posicion.
     * @return La clase que implementa debe devolver un
     * vector normalizado representando la direccion
     * de la normal a la superficie del primitivo.
     */
    public abstract Vector3D getNormal(Vector3D pos);

```

```
29      public void setMaterial(Material m){
30          material = m;
31      }
32
33      public String getName(){
34          return name;
35      }
36
37      public Material getMaterial(){
38          return material;
39      }
40  }
```

Para ejemplificar una clase que extiende a `Primitive` mostramos la clase `Sphere` que representa una esfera en el espacio.

```
1   public class Sphere extends Primitive {
2       // Centro de la esfera.
3       Vector3D p;
4       // Radio de la esfera.
5       double r;
6
7       /**
8        * Crea un nuevo ejemplar de una esfera.
9        * @param name Nombre del ejemplar.
10       * @param m Material.
11       * @param radius Radio de la esfera.
12       * @param pos Centro de la esfera.
13       */
14      public Sphere(String name, Material m,
15                    double radius, Vector3D pos){
16          super();
17          p = pos;
18          r = radius;
19          this.name = name;
20          this.material = m;
21      }
22
23      public Vector3D getNormal(Vector3D pos){
24          return pos.sub(p).normalize();
25      }
26
27      public double intersect(Ray r){
28          Vector3D v = r.getOrigin().sub(p);
29          double b = 2.0f*v.dot(r.getDirection());
30          double c = v.sqrNorm() - this.r*this.r;
31          double disc = b*b - 4*c;
32          if(disc < 0){
33              return Double.POSITIVE_INFINITY;
34          }
35          disc = Math.sqrt(disc);
36          double x1 = disc*-1.0f - b;
37          double x2 = disc - b;
38          x1 /= 2.0f;
39          x2 /= 2.0f;
40          if(x1 < 0.0f && x2 < 0.0f){
41              return Double.POSITIVE_INFINITY;
```

```
42          }
43          if(x1 > 0.0f && x2 > 0.0f){
44              return Math.min(x1, x2);
45          } else {
46              return Math.max(x1, x2);
47          }
48      }
49  }
```

Para calcular la intersección más pequeña, basta con barrer todos los primitivos y encontrar aquel cuyo método `intersect` devuelve el valor más pequeño.

Una vez encontrado el primitivo con la intersección más cercana al rayo dado, debemos calcular su valor de color. Esta es la parte más importante del algoritmo de *ray tracing* ya que aquí se implementan los fenómenos ópticos que suceden en la naturaleza. En esta etapa del desarrollo calcularemos exclusivamente el color y sombreado de la superficie del primitivo. Posteriormente calcularemos reflexión, refracción y sombras.

Cálculo de intersección para primitivos sencillos

Para efectos del proyecto mostraremos la forma de calcular la intersección con los tres primitivos soportados: esferas, planos infinitos y paralelepípedos alineados a los ejes coordenados:

- **Planos.** Usando la definición *punto/normal* del plano, definimos la ecuación del plano como:

$$n \cdot (x - p) = 0 \tag{6.2}$$

  Donde n representa un vector normal al plano, p un punto del plano y x la variable independiente de la ecuación. Para calcular la intersección de un plano con una recta paramétrica $o + td$ sustituimos a x por la definición de la recta quedando entonces:

$$\begin{aligned} n \cdot (o + td - p) &= n \cdot ((o - p) + td) \\ &= t(n \cdot d) + n \cdot (o - p) \\ &= 0 \end{aligned} \tag{6.3}$$

  El cual induce una ecuación de primer grado. Basta con despejar t y devolver su valor. Es importante que en caso de que t sea negativa se devuelva un valor bandera donde se especifique que el plano no fue intersectado, ya que geométricamente una t negativa representa objetos que se encuentran atrás del observador, por lo que no queremos que aparezcan en la escena.

  Para calcular el vector normal a la superficie del plano, basta con devolver n.

- **Paralelepípedos.** Decidimos escoger paralelepídedos alineados a los ejes coordenados ya que existe un truco sencillo para calcular su intersección con un rayo: si suponemos que un paralelepípedo alineado a los ejes es en

realidad la intersección de tres secciones de cada eje (se ven como regiones acotadas por dos planos), podemos calcular el punto de intersección realizando la intersección de todos los intervalos o secciones.

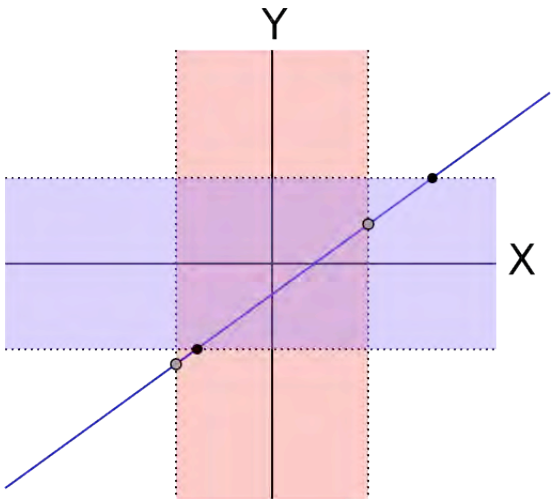

Figura 6.5: La región acotada por la intersección de dos intervalos de cada eje representa un rectángulo alineado a los ejes coordenados. La extensión al espacio tridimensional es intersectando tres intervalos.

Si la intersección de los intervalos es vacía, significa que el rayo no toca el paralelepípedo. En caso contrario debemos calcular cuál es el intervalo de intersección y finalmente, el punto en $\mathbb{R}^3$ donde inicia el intervalo, el cuál será el punto de intersección más cercano. Para realizar este cálculo debemos proceder coordenada a coordenada. Si tenemos una caja con esquinas (a_1, a_2, a_3), (b_1, b_2, b_3), debemos intersectar una recta paramétrica $o + td$ en cada coordenada. En el caso del eje x la condición es $t \in [a_1, b_1]$. Sin embargo, si $a_1 < b_1$, el valor de t va decreciendo, situación que complica los cálculos, por lo que lo más conveniente es definir los intervalos de la siguiente forma:

$$t \in [min\{a_i, b_i\}, max\{a_i, b_i\}] \text{ con } i \in 1, 2, 3$$

Finalmente, basta con calcular la intersección de todos los intervalos y devolver la cota inferior del mismo. Al igual que los planos, siempre debemos validar que el valor de t sea positivo.

Para calcular el vector normal de la superficie del plano, necesitamos definir en qué cara intersecta el rayo al plano y devolver un vector normal a dicha cara. Esto se logra fácilmente guardando la pista de a qué coordenada pertenecía la cota inferior del intervalo de intersección.

- **Esferas.** Si definimos a la esfera como el conjunto de puntos que equidistan en R unidades del centro o podemos proponer la ecuación:

$$||x - o||^2 - R^2 = 0,$$

donde x representa los puntos de la esfera, o el centro y R el radio. Si queremos intersectar una esfera con una recta paramétrica $p + td$ (con d unitario), sustituimos x por $p+td$ y despejamos t para calcular la distancia a la cual la recta paramétrica intersecta la esfera, de este modo obtenemos:

$$\begin{aligned} ||p + td - o||^2 - R^2 &= ||(p - o) + td||^2 - R^2 \\ &= ((p - o) + td) \cdot ((p - o) + td) - R^2 \\ &= (p - o) \cdot (p - o) + 2(p - o) \cdot (td) + (td) \cdot (td) - R^2 \\ &= t^2 + 2t((p - o) \cdot d) + ((p - o) \cdot (p - o) - R^2) \\ &= 0 \end{aligned} \tag{6.4}$$

Al final de la manipulación algebraica hemos definido una ecuación de segundo grado para calcular el valor de t. Como el vector $p-o$ es constante, podemos definirlo como un vector c y obtenemos el siguiente polinomio de segundo grado:

$$t^2 + 2(c \cdot d)t + (c \cdot c - R^2) = 0 \tag{6.5}$$

Al igual que en los primitivos anteriores, la solución del problema de la intersección con la esfera es el valor positivo más pequeño de la solución de la ecuación cuadrática.

Para obtener el vector normal a una esfera, basta con usar el hecho de que todo vector normal a la superficie de una esfera va en dirección al centro, por lo que, dado v, un vector normal a la superficie de la esfera con dirección a v sería:

$$n = \frac{v - o}{||v - o||} \tag{6.6}$$

Iluminación

Para calcular el color del punto de intersección del rayo con el primitivo más cercano, necesitamos un modelo de iluminación que describa de manera precisa la intensidad de la luz que se refleja en ese punto. El modelo más sencillo es el modelo de *Phong*[10] el cual consiste en dos elementos de iluminación:

[10]En honor a su creador, Bui Tuong Phong, investigador de la Universidad de Utah en Estados Unidos el cual publicó este modelo en su tesis doctoral en 1973.

Figura 6.6: El *shader* de Phong

- **Luz difusa.** Representa la cantidad de luz reflejada por el objeto en cada punto de su superficie. La expresión que calcula el coeficiente de sombreado en cada punto de la superficie del primitivo está dada por:

$$phong_d = m_{dif} l_{pow} \left(l_{dir} \cdot l_n \right) \tag{6.7}$$

  Donde:

  m_{dif} Es el coeficiente de luz difusa del material asociado al primitivo.

  l_{pow} Es la intensidad luminosa de la fuente de luz.

  l_{dir} Es un vector que representa la dirección donde está la fuente de luz con respecto al punto de intersección con el rayo.

  l_n Representa el vector normal de la superficie del primitivo con respecto al punto de intersección con el rayo.

- **Luz especular.** Representa la luz que "rebota" directamente al punto focal del espectador; se ven como partes brillantes. La ecuación que calcula el valor especular se define como:

$$phong_s = s^{m_h} \tag{6.8}$$

  de donde

  s se define como:

$$s = \left[l_{dir} - 2 \left(l_{dir} \cdot l_n \right) l_n \right] \cdot r_{dir}, \tag{6.9}$$

  donde r_{dir} representa la dirección del rayo que intersecta el primitivo dado.

  m_h representa el atributo de dureza especular definido en el material asociado al primitivo.

Una vez calculados los valores de reflexión difusa y especular, debemos calcular el color final del píxel representado por el rayo trazado. El valor de color

se obtiene acumulando las aportaciones de luz producidas por cada fuente luminosa. De este modo, para toda fuente luminosa, la ecuación que acumula el color en cada píxel está dada por:

$$c = c + phong_d m_c + phong_s W, \tag{6.10}$$

donde:

c representa el color del píxel.

m_c representa el color del material asociado al primitivo.

W es el color blanco $(1, 1, 1)$. Es posible escoger otro color para la componente especular colocando un segundo valor de color en la estructura de datos de los materiales. Otra opción sería utilizar el color de la fuente de luz.

6.3.8. *Etapa 4:* Sombras, reflexión y refracción

Al finalizar las tres primeras etapas del proyecto deberíamos tener una versión sencilla pero funcional del *ray tracer*. Las siguientes etapas representan características avanzadas que elevan el realismo y calidad de imagen que sintetiza el proyecto.

Sombras

El procedimiento requerido para calcular sombras se basa en la construcción de un rayo secundario[11] que parte del punto de intersección del primitivo con un rayo primario[12] y va en dirección a la fuente de luz a evaluar si está ocluida o no por algún otro objeto.

La idea para calcular las sombras es muy sencilla como se muestra en el siguiente código:

```
1  /** Evalua si la fuente de luz que llega al punto de
2   * interseccion dado esta siendo ocluida por algun
3   * objeto entre el punto de interseccion y la fuente
4   * luminosa.
5   * @param l Fuente de luz a probar.
6   * @param ld Direccion donde se encuentra la fuente luminosa.
7   * @param ip Punto de interseccion del rayo primario con el
8   * primitivo.
9   */
10 double shadow(Light l, Vector3D ld, Vector3D ip){
11   // Construimos el rayo secundario que parte del
12   // punto de interseccion y va en direccion a
13   // la fuente luminosa.
14   Ray shadowRay = new Ray(ip, ld);
15   // Intersectamos el rayo con todos los primitivos
```

[11] Un rayo secundario se define como un rayo producido a partir de cálculos que involucran un rayo primario.

[12] Los rayos que se generan a partir del foco y el plano focal al inicio del algoritmo de *ray tracing*.

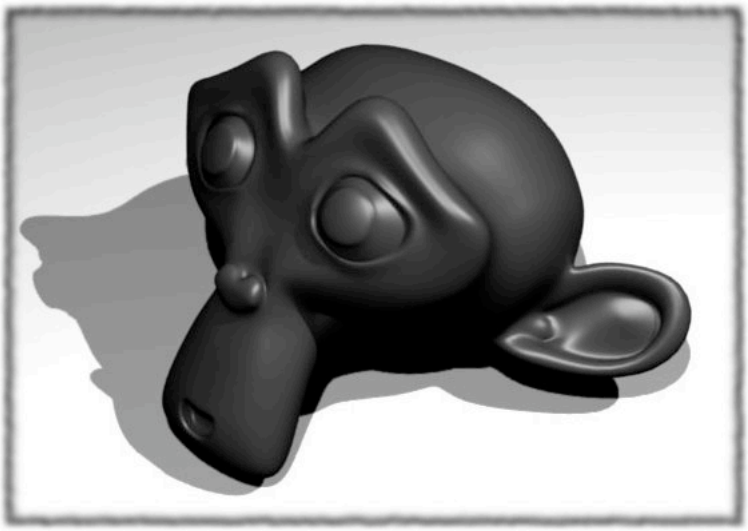

Figura 6.7: Ejemplo de la proyección de sombras dadas dos fuentes de luz usando el algoritmo propuesto. Nótese como en la parte donde hay intersección de las sombras los píxeles son totalmente negros.

```
16    // y buscamos si existe algun primitivo que
17    // se encuentre ocluyendo la fuente luminosa.
18    Primitive si = shadowIntersect(lr, l);
19    // En caso de que haya un primitivo devolvemos cero
20    // en caso contrario uno.
21    return si == null ? 1.0 : 0.0;
22  }
```

El procedimiento `shadowIntersect` es muy similar al de intersección de primitivos salvo un detalle: la distancia máxima de intersección es precisamente la distancia donde se encuentra la fuente luminosa, esto con la finalidad de sólo buscar objetos que se encuentren entre el punto de intersección del primitivo con el rayo primario y la fuente luminosa. No nos interesan objetos que se encuentren atrás del punto de intersección o atrás de la fuente luminosa ya que no ocluyen la iluminación representada por el rayo primario.

Una vez calculado el valor de sombra, modificamos la ecuación de iluminación de la siguiente forma:

$$c = c + s\left(phong_d m_c + phong_s W\right), \tag{6.11}$$

donde s es el valor de sombra que acabamos de calcular.

Reflexión

El fenómeno de reflexión es aquel que percibimos cuando observamos una imagen reflejada en un espejo. El efecto producido es la reproducción de los

rayos de luz que inciden sobre él respecto a cierto ángulo, como se muestra posteriormente.

Al igual que el cálculo de sombras, la reflexión se calcula a partir de la producción de rayos secundarios, sin embargo, el reflejar un rayo en una superficie no necesariamente se produce sólo un rayo, sino que se pueden producir varios rayos dependiendo del número de rebotes que especifiquemos en el algoritmo.

El procedimiento para calcular el color aportado por la reflexión de un rayo se realiza de manera recursiva donde la condición de parada es un número límite de rebotes de un rayo a través de diversas superficies reflejantes ubicadas en la escena a sintetizar. La idea general para implementar la reflexión se presenta en el siguiente código:

```
1  /** Calcula el color aportado por la reflexion
2   * de un rayo sobre un primitivo.
3   * @param p El primitivo donde incide el rayo.
4   * @param ip Punto de interseccion del rayo y el
5   * primitivo.
6   * @param r El rayo.
7   * @param depth Profundidad de recursion actual.
8   */
9  private Color getReflectColor(Primitive p, Vector3D ip,
10         Ray r, int depth){
11         // Generamos un nuevo color inicialmente negro.
12         Color rc = new Color();
13         // Calculamos el vector normal a la
14         // superficie del primitivo.
15         Vector3D n = p.getNormal(ip);
16         // Aplicamos la ley de reflexion.
17         double rs = 2.0f*n.dot(r.getDirection());
18         Vector3D rr =
19                 r.getDirection().sub(n.scalarProd(rs)).normalize();
20         // Construimos un nuevo rayo que parte
21         // del punto de interseccion y va en la
22         // direccion calculada por la ley de reflexion.
23         // La adicion del vector de reflexion escalada por
24         // una epsilon mayor a cero sirve para controlar
25         // la inestabilidad numerica.
26         Ray rray = new Ray(ip.add(rr.scalarProd(Vector3D.EPSILON)),
27                 rr);
28         // Aplicamos el ray trace recursivo.
29         rayTrace(rray, rc, depth+1);
30         // Escalamos el valor de color en base al coeficiente
31         // de reflexion almacenado en el material asociado
32         // al primitivo.
33         rc.scale(p.getMaterial().reflect);
34         return rc;
35 }
```

El vector `rr` se calcula en base a la ley de reflexión. Observando la figura 6.3.8 podemos calcular el vector r como:

$$r = d + 2a = d + 2\frac{d \cdot n}{||n^2||}n \tag{6.12}$$

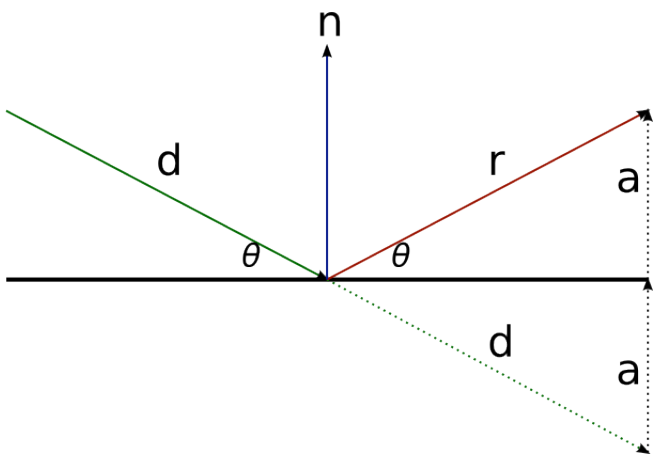

Figura 6.8: El vector d siendo reflejado sobre una superficie, produciendo un vector r. El vector n representa un vector normal a la superficie donde se refleja d. Nótese como el ángulo de incidencia de d se mantiene en r

La componente a se calcula directamente del hecho que a es una proyección del vector d sobre n; de este modo podemos utilizar la ecuación de la proyección de dos vectores por medio del producto interior.

El implementar la reflexión también causa que el algoritmo general de *ray tracing* cambie, siendo necesario implementar el caso base de la recursión donde se devuelve el color negro y finaliza las llamadas recursivas.

Refracción

La refracción es el fenómeno que se observa en los objetos de cristal (como las lentes). Específicamente se define como la desviación de un rayo de luz al momento de cambiar de medio de transmisión (en este caso de aire a cristal).

Para modelar la refracción, definimos una constante conocida como *índice de refracción* para cada tipo de medio de transmisión. Por simplicidad, en el algoritmo de *ray tracing* supondremos que el índice de refracción del aire es la unidad y el índice de refracción de cada material estará especificado por medio del archivo de entrada.

La ecuación que describe el ángulo de desviación del rayo de luz que incide en la superficie que refracta se conoce como la *ley de Snell* y define la siguiente relación:

$$\frac{\sin\theta}{\sin\theta'} = \frac{n_2}{n_1}, \tag{6.13}$$

donde θ y θ' representan el ángulo incidente y resultante respectivamente y n_1 y n_2 los índices de refracción del medio inicial y el medio final respectivamente. Para efectos de implementación, necesitamos realizar diversos cálculos

Figura 6.9: Ejemplo de refracción en las gotas de agua que penden de la planta. Imagen obtenida de *http://www.flickr.com/photos/ecstaticist/530238735/* bajo licencia Creative Commons.

para poder hacer uso de la ley de Snell, ya que nuestra representación de datos no contempla ángulos, necesitamos utilizar operaciones con vectores para poder incluir esta ecuación en el programa.

Asumiendo que d y n son vectores unitarios (que sólo representan una dirección) si observamos la figura 6.3.8 podemos observar que n y b forman una base ortonormal en el plano de refracción. Por lo tanto podemos expresar los vectores d y t como:

$$\begin{aligned} d &= \sin\theta b - \cos\theta n \\ t &= \sin\theta' b - \cos\theta' n \end{aligned} \tag{6.14}$$

de este modo podemos calcular el valor de b como:

$$b = \frac{d + \cos\theta n}{\sin\theta}$$

Obteniendo la siguiente ecuación:

$$\begin{aligned} t &= \frac{\sin\theta'}{\sin\theta}(d + \cos\theta n) - \cos\theta' n, \text{ usando ley de Snell:} \\ &= \frac{n_1}{n_2}(d + \cos\theta n) - \cos\theta' n \end{aligned} \tag{6.15}$$

Posteriormente para calcular $\cos\theta' n$ utilizamos la identidad trigonométrica $\cos^2\theta + \sin^2\theta = 1$:

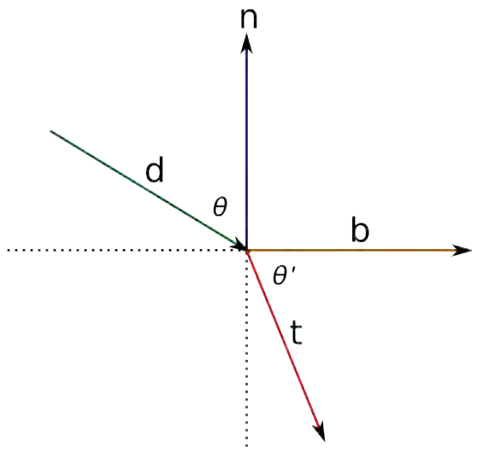

Figura 6.10: *Ley de Snell*. Nuestro objetivo es calcular el vector t conociendo los vectores d, n y los índices de refracción de ambos medios. El vector b se puede calcular partiendo de d y n

$$\begin{aligned}
\cos^2\theta' &= 1-\sin^2\theta' = 1-\frac{{n_1}^2}{{n_2}^2}\sin^2\theta \\
&= 1-\frac{{n_1}^2}{{n_2}^2}\left(1-\cos^2\theta\right) \\
\cos\theta' &= \sqrt{1-\frac{{n_1}^2}{{n_2}^2}\left(1-\cos^2\theta\right)}
\end{aligned} \tag{6.16}$$

Sustituyendo $\cos\theta'$ y usando el hecho que $\cos\theta = d\cdot n$ obtenemos finalmente:

$$t = \frac{n_1}{n_2}\left(d+(d\cdot n)\,n\right) - \sqrt{1-\frac{{n_1}^2}{{n_2}^2}\left(1-(d\cdot n)^2\right)}n \tag{6.17}$$

ecuación que se encuentra en términos de variables conocidas por el algoritmo de *ray tracing*.

Conociendo la forma de calcular t el procedimiento requerido para implementar la refracción sería:

```
1  /** Calcula la aportacion del fenomeno de
2   * refraccion al color del pixel.
3   * p Primitivo mas cercano al rayo.
4   * r Rayo a analizar.
5   * ip Punto de interseccion del rayo y el primitivo.
6   * depth Profundidad de la recursion (igual que en
7   * la reflexion.
8   */
9  private Color getRefractionColor(Primitive p,
10         Vector3D ip, Ray r, int depth){
```

```
11          // Obtenemos el indice de refraccion del material.
12          double ior = p.getMaterial().ior;
13          double rc;
14          Vector3D d = r.getDirection();
15          Vector3D n = p.getNormal(ip);
16          // Si el coseno del angulo que forman n y d
17          // es negativo significa que estamos en
18          // el exterior del objeto que refracta luz
19          // por lo que el cociente de los indices de
20          // refraccion es 1/ior.
21          if(n.dot(d) < 0){
22                  rc = 1.0f/ior;
23          // En caso contrario, el cociente es ior.
24          // Debemos invertir el vector normal para
25          // que apunte al interior del objeto.
26          } else {
27                  rc = ior;
28                  n = n.inv();
29          }
30          // Usamos ley de Snell para calcular el vector t.
31          Vector3D t = refract(n, d, rc);
32          Color rcol = new Color();
33          // Construimos un rayo secundario en direccion a t
34          // que parte del punto de interseccion.
35          rayTrace(new Ray(ip.add(t.scalarProd(Vector3D.EPSILON)),
36                  t), rcol, depth+1);
37          return rcol;
38  }
```

6.3.9. *Etapa 4:* Posición de la cámara y relación de aspecto

Orientación de la cámara

Hasta el momento hemos desarrollado el *ray tracer* ignorando la posición y dirección de la cámara. Una vez que tenemos todos los elementos básicos de la óptica, podemos proceder a implementar la parte que se encarga de colocar la cámara en posición. Para ello necesitaremos del uso de matrices de cambio de coordenadas.

Si movemos la posición del espectador al punto p de tal manera que esté dirigiendo la mirada en dirección d (d unitario), se podría decir que estamos cambiando nuestro sistema de coordenadas del origen y los ejes x, y, z al origen centrado en p y con tres vectores que formen una base ortonormal derecha relacionados con la dirección hacia donde mira el espectador.

Normalizando los vectores u, v y d podemos construir una matriz de rotación para armar el cambio de coordenadas. La matriz de rotación que pasa los ejes x, y y z al sistema u, v, d sería:

$$r = \begin{bmatrix} u_x & v_x & d_x \\ u_y & v_y & d_y \\ u_z & v_z & d_z \end{bmatrix}^t = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ d_x & d_y & d_z \end{bmatrix} \tag{6.18}$$

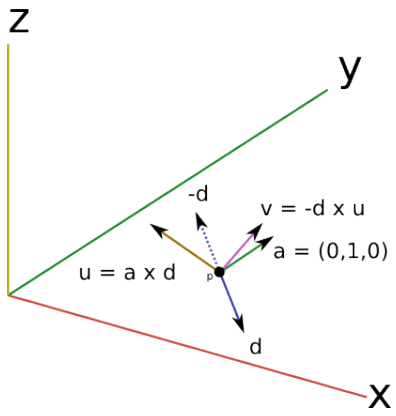

Figura 6.11: *Colocando la cámara en posición.* El vector p representa la posición del espectador y d la dirección hacia donde voltea. Los vectores u, v y d forman una base ortonormal centrada en el espectador donde el análogo del eje Z es d

Relación de aspecto

En las etapas anteriores existía la restricción de que la resolución de la imagen sintetizada tenía que ser forzosamente a razón $4:3$. Es el momento de dar libertad al usuario de escoger la resolución y mantener la relación de aspecto correcta.

El problema consiste en armar un rectángulo que mantenga la misma relación de aspecto que la resolución de entrada y además que se mantenga un ángulo de visión correcto. Si tenemos una resolución de 400×400 píxeles, podríamos usar el cuadrado unitario $(x, y, 0)$ con $x \in [-0{,}5, 0{,}5]$ y $y \in [-0{,}5, 0{,}5]$; sin embargo podríamos usar un cuadrado de área distinta y mantendría la misma relación de aspecto.

Necesitamos un tercer parámetro para definir unívocamente el plano focal de la imagen: el *ángulo de visión horizontal*. Este ángulo representa la amplitud de visión que tiene el espectador con respecto a su horizontal; por ejemplo, si el espectador tiene un ángulo de visión de 180 grados, puede ver la imagen de lado a lado en su totalidad. Un efecto común derivado del ángulo de visión es la *distorsión geométrica* causada debido a que el espectador está "aplanando" un campo de vista esférico en el plano focal. En el campo de la fotografía este efecto se le conoce como *ojo de pescado* (*fisheye* en inglés).

La distancia focal d se puede dejar como constante así como lo hicimos en nuestra versión inicial ($d = 5$). De este modo, es posible calcular el ancho del plano focal de la siguiente forma:

$$w_f = 2d \tan\left(\frac{\alpha}{2}\right), \tag{6.19}$$

quedando $10 \tan(\alpha/2)$ en el caso de $d = 5$.

Una vez calculado el ancho del plano focal, podemos calcular la altura del

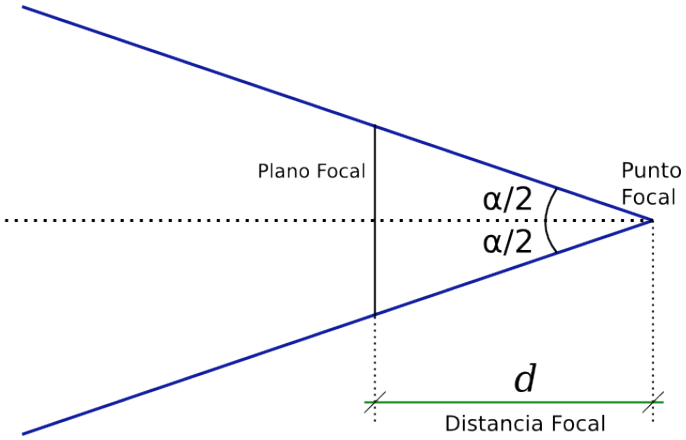

Figura 6.12: *Ángulo de vision.* La imagen muestra las variables que se involucran en el cálculo del plano focal, distancia focal y ángulo de visión. (La imagen está vista desde arriba).

mismo utilizando la resolución de entrada que se proporciona en el archivo de escena sabiendo que la relación de aspecto se debe mantener, es decir:

$$\frac{w}{h} = \frac{w_f}{h_f}$$

Así pues, despejando h_f tenemos que:

$$h_f = \frac{hw_f}{w} \tag{6.20}$$

Construcción del plano focal orientado

Conociendo la matriz de transformación que rota el plano en dirección al espectador, la posición del espectador y el tamaño del plano focal, estamos en condiciones de proponer un procedimiento que coloque la cámara en posición.

La forma más sencilla de obtener el resultado que queremos es volver a armar nuestro plano focal coplanar al plano XY del sistema coordenado, colocar al espectador en $o = (0, 0, -5)$ y posteriormente aplicar la matriz de rotación y trasladar el plano a la posición de la cámara. El punto focal de la cámara estará 5 unidades atrás del plano focal, el cual está en la posición dada por el usuario.

El plano focal se construye usando los valores w_f y h_f produciendo el rectángulo:

$$P = (x, y, 0) \text{ con } x \in \left[-\frac{w_f}{2}, \frac{w_f}{2}\right], y \in \left[-\frac{h_f}{2}, \frac{h_f}{2}\right] \tag{6.21}$$

De este modo, el código que realiza la orientación de la cámara luciría como sigue:

```
// Distancia focal.
private final double d = 5.0;
// El nuevo origen donde parten los
// rayos a trazar.
private Vector3D origin;

/** Arma el plano focal de la imagen con la orientacion
* especificada por la camara.
* @param p Posicion de la camara.
* @param d Direccion de la camara.
* @param width Ancho de la imagen en pixeles.
* @param height Alto de la imagen en pixeles.
* @param aov Angulo de vision en grados sexagesimales.
*/
public void setFocalPlane(Vector3D p, Vector3D d,
                          int width, int height, int aov){
        // Calculamos el tamano del
        // plano focal.
        double wf, hf;
        wf = 2*this.d*Math.atan(Math.PI*aov/360.0);
        hf = height * wf / width;

        // Construimos los puntos que
        // constituyen el plano focal.
        Vector3D ul = new Vector3D(wf/2, hf/2, 0);
        Vector3D ur = new Vector3D(-1 * wf/2, hf/2, 0);
        Vector3D ll = new Vector3D(wf/2, -1 * hf/2, 0);
        Vector3D lr = new Vector3D(-1 * wf/2, -1 * hf/2, 0);

        // Construimos el nuevo sistema
        // de coordenadas basado en la
        // orientacion de la camara.
        Vector3D up = new Vector3D(0,1,0);
        Vector3D u = up.cross(d).normalize();
        Vector3D v = u.cross(d.inv());

        // Construimos la matriz de rotacion.
        TransformMatrix m = new TransformMatrix(u, v, d);
        m = m.transpose();

        // Aplicamos la matriz a cada esquina del
        // plano focal.
        ul = m.apply(ul).add(p);
        ur = m.apply(ur).add(p);
        ll = m.apply(ll).add(p);
        lr = m.apply(lr).add(p);

        // Calculamos el nuevo origen donde
        // parten los rayos a trazar.
        Vector3D f = new Vector3D(0,0,-1.0 * this.d);
        f = m.apply(f).add(p);
        origin = f;
}
```

6.3.10. *Etapa 5:* Multimuestreo y sombras suaves

Multimuestreo

Existe un efecto producido por la implementación actual del *ray tracer* que hemos dejado de largo: *aliasing*. El efecto de *aliasing* es muy estudiado en el área de proceso digital de señales y consiste en un déficit de muestras tomadas por un sistema digital a una variable continua el cual causa artefactos indeseables en la señal representada por el sistema digital. Es un tema de investigación amplio y escapa a los objetivos de este documento, sin embargo, la forma más sencilla de mostrar el *aliasing* en una imagen sintetizada es por medio de dos imágenes: una con *aliasing* y una sin él.

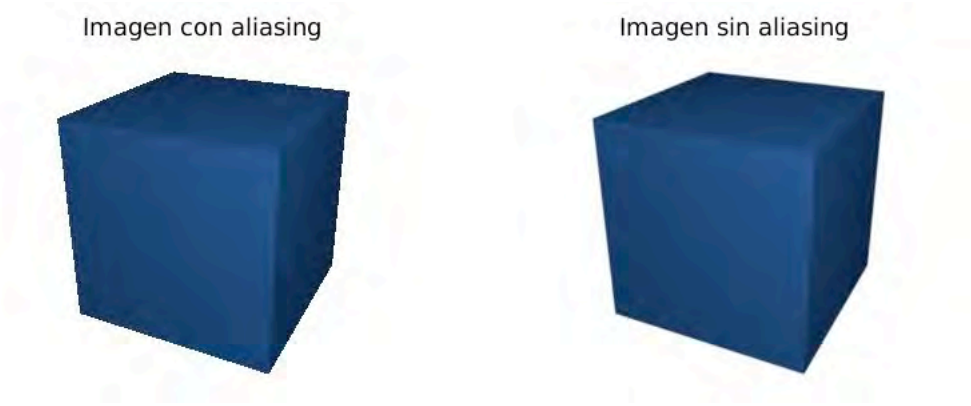

Figura 6.13: *Aliasing.* Se ven como bordes dentados en las imágenes.

Si bien el fenómeno de *aliasing* es inherente a la discretización producida por un sistema digital, en este caso, cuando se calcula el color de cada píxel por un solo rayo, es posible mejorar la calidad de la imagen utilizando la técnica de multimuestreo por píxel (*oversampling*) que, como su nombre lo indica, consiste en tomar varias muestras para calcular el color resultante del píxel. Las muestras en este caso son los rayos que disparamos en el algoritmo.

La forma más sencilla de implementar el multimuestreo es creando una partición regular de cada píxel, tirar un rayo por cada punto de la partición y aplicar la media aritmética de los colores resultantes de cada rayo. A esta técnica se le conoce como filtro de caja (*box filter*) y aunque no es la técnica más efectiva, es la más sencilla de implementar. Existen diversas técnicas para mezclar las muestras y producir el píxel resultante, siendo la más usada el filtro gaussiano que realiza un promedio pesado entre las diferentes muestras por medio de una ventana de convolución que se aproxima a la distribución de normal de Gauss.

Sombras Suaves

Las sombras que implementamos en la etapa 4 se les conoce como *sombras duras* y la razón de ello es porque generan bordes bien definidos en la superficie de proyección de la sombra. Este modelo se aproxima muy bien a la realidad en caso de que la luz sea unidireccional (como la luz del sol), cuando la fuente luminosa tiene un área muy pequeña y se puede suponer como puntual, o cuando la fuente luminosa se encuentra muy cerca del objeto. Sin embargo, en casos donde la fuente luminosa se origina de una superficie grande, generalmente vemos las sombras desvanecerse paulatinamente sobre la superficie donde se proyectan.

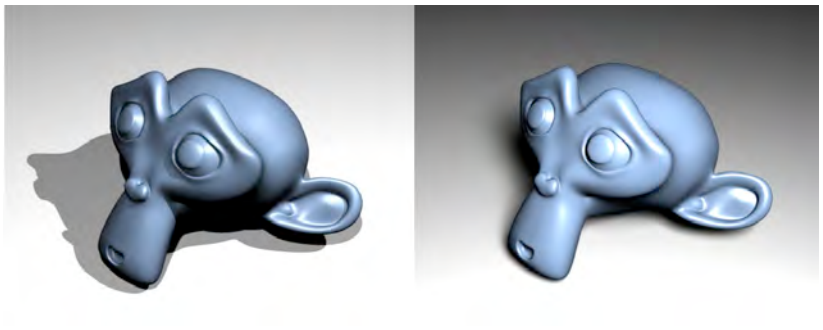

Figura 6.14: *Sombras suaves.* Comparación entre el algoritmo de sombreado de la etapa 4 (izquierda) y el algoritmo de sombreado que se presenta en este apartado (derecha).

Con la finalidad de implementar esta técnica de sombreado, introducimos un nuevo tipo de fuente luminosa: la *luz de superficie*. Como su nombre lo indica, es una fuente de luz la cual en vez de emitir desde un punto, emite uniformemente desde una superficie. Por simplicidad y para fines ilustrativos, las luces de superficie descritas para este proyecto siempre serán cuadrados.

Las luces de superficie tienen dos parámetros de posición: *dirección* y *posición*. Al igual que el plano focal, la luz de superficie se puede construir generando cuatro puntos en el plano XY y posteriormente rotando y trasladando dichos puntos a su posición final. El estudiante puede agregarle atributos así como intensidad luminosa, tamaño de la superficie, etcétera.

Una vez descrita la luz de superficie, el procedimiento para implementar las sombras suaves es por medio de *muestreo*, una técnica similar a la utilizada para contrarrestar los efectos de *aliasing* descrita anteriormente. En el caso de la luz de superficie, en vez de tirar un solo rayo secundario en dirección a la luz puntal, procedemos a crear una partición regular de la superficie luminosa y producimos tantos rayos secundarios como puntos de la partición definamos. Finalmente, el valor de sombreado no será cero o uno como en los casos anteriores, sino un promedio de todos los rayos que muestreamos de la luz de superficie, siendo cero aquellos que fueron ocluidos y uno aquellos que lograron alcanzar la fuente luminosa.

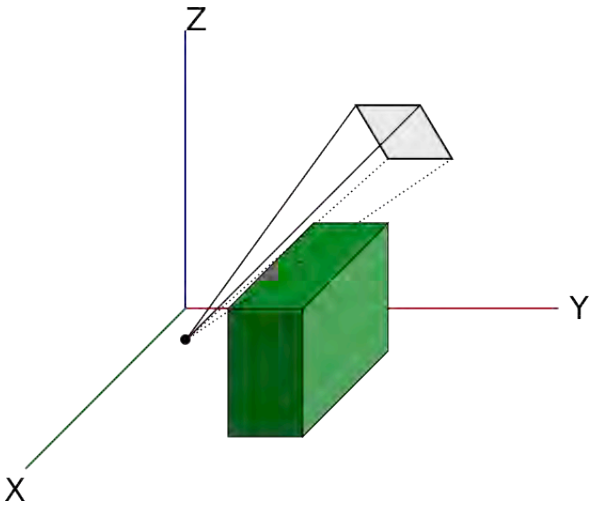

Figura 6.15: *Muestreo de sombras.* Usando múltiples rayos secundarios para implementar el sombreado suave. Las líneas punteadas representan muestras de la luz de superficie que han sido ocluidas por un objeto.

6.3.11. Temas para investigar

Si bien la propuesta de proyecto para este curso es sencilla y no es comparable con sistemas de síntesis a nivel industrial, después de desarrollar este proyecto el estudiante conocerá los aspectos fundamentales del diseño e implementación de estas tecnologías, así como una experiencia de desarrollo que involucra conceptos de matemáticas que se presentan en los primeros semestres del plan de estudios.

A continuación se enumeran diversos temas para investigar los cuales podrían mejorar o sofisticar el funcionamiento del algoritmo de *ray tracing* que no se mencionan en este documento:

- **Más primitivos.**
  - Triángulos y mallas de triángulos. Actualmente son la técnica más popular por el software comercial para modelaje en 3D .
  - Superficies paramétricas por medio de *NURBS* (*Non-Uniform Rational B-Spline*). Una técnica muy común en el diseño industrial que utiliza superficies paramétricas modeladas por medio de polinomios.
- **Texturas.** Actualmente los materiales de los primitivos son sólidos y sin textura aparente, un proyecto interesante sería la implementación de texturas.
  - Texturas de imagen. Obtenidas por medio de un archivo de imagen raster.

  - Texturas procedurales. Algoritmos que modelan matemáticamente texturas comunes como ruido gaussiano, madera, turbulencia, etcétera.
  - Diversos mapeos de textura. El poder mapear texturas al color difuso, especular, reflectividad, refracción, vector normal (*bumpmapping*), etcétera.

- **Iluminación global.** Un conjunto de algoritmos avanzados para representar de manera más realista la iluminación.
  - Oclusión ambiental. Proceso que involucra muestreo de rayos para obtener iluminación indirecta de la escena.
  - Fotones. Utiliza métodos de muestreo de luz para simular los artefactos que se producen al doblar rayos a través de superficies de cristal (cáusticos). Se ven como patrones luminosos en la superficie inmediata después del material refractivo.

- **Fenómenos ópticos.**
  - Dispersión y aberración cromática. El efecto que se aprecia al ver una imagen a través de una lente: al observar los bordes de la imagen se perciben pequeños patrones de luz similares a los de un arcoiris debido a la diferencia del índice de refracción para cada longitud de onda de luz.
  - Profundidad de campo. En las cámaras fotográficas, el variar la apertura de una lente (la anchura con la que se abre el diafragma que permite el paso de luz) hace variar la profundidad de campo. Cuando una fotografía tiene poca profundidad de campo, sólo los objetos más cercanos a la cámara se ven enfocados y poco a poco los objetos que se van alejando pierden enfoque suavizando la imagen (este efecto se conoce como *bokeh*). Recíprocamente, si la apertura de la lente es muy baja, casi toda la imagen aparece enfocada, dando una mayor profundidad de campo al espectador.

- **Síntesis paralela y distribuida.** Como ya vimos, el proceso de cálculo de cada rayo es independiente de los demás, por lo que esta tarea es fácilmente paralelizable y distribuible entre varios procesadores. Un proyecto interesante sería emplear los conocimientos obtenidos en programación multihilo y realizar un programa que pueda sintetizar imágenes en paralelo por medio de *ray tracing*.

6.4. Proyecto 2: Máquina virtual y ensamblador

6.4.1. Descripción

A lo largo de la licenciatura hemos desarrollado programas en Java, los hemos compilado y producido *bytecode* que finalmente hemos ejecutado en la máquina

virtual de Java. Sin embargo, ¿qué es y cómo funciona una máquina virtual?

El concepto de máquina virtual nació mucho antes de Java, y en general consiste en desarrollar un paquete de software que simule el funcionamiento de la especificación de una máquina desde el punto de vista de hardware. En el caso de la máquina virtual de Java, la especificación de esa máquina representa un entorno de ejecución completo (*runtime*), no sólo la definición del hardware de una máquina.

En el desarrollo de este proyecto nos enfocaremos en la implementación de una máquina virtual que simula una máquina de hardware hipotética muy simple descrita en este documento. Esta máquina virtual interpretará y ejecutará código binario, al igual que lo hace un microprocesador.

Una vez que se tenga la máquina virtual funcionando, debemos proporcionar al programador un lenguaje de programación que pueda utilizar para programar la máquina. Sería demasiado complicado desarrollar los programas por medio de código de máquina de manera directa, por lo que desarrollaremos un procesador de lenguaje ensamblador para facilitar el desarrollo de aplicaciones para la máquina virtual.

Escogemos un lenguaje ensamblador ya que el desarrollo del procesador es considerablemente más sencillo que un compilador de lenguaje de alto nivel. Dicha tarea requeriría de un curso de compiladores, teoría de la computación y lenguajes de programación.

6.4.2. Arquitectura

El diagrama de bloques de nuestro programa luciría como el siguiente:

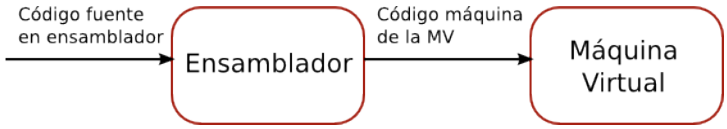

Figura 6.16: Diagrama de bloques de la máquina virtual y ensamblador

Nuestro proyecto se dividirá en dos etapas principales: la construcción de la máquina y posteriormente la elaboración del procesador, siendo esta última la etapa más complicada pero la más interesante.

Antes de describir y desarrollar la máquina virtual, debemos conocer algunos conceptos básicos de la arquitectura de una computadora. Actualmente, las computadoras siguen un modelo conocido como *arquitectura de Von Neumann*; este modelo de computadora describe una máquina en *hardware* que consta de los siguientes componentes:

Memoria principal Lo que conocemos como memoria RAM[13]. El gran logro de Von Neumann fue el especificar que tanto los datos de un programa como el código que comprende su ejecución se encuentran almacenados en

[13] *Random Access Memory*. Memoria de acceso aleatorio.

una sola categoría de memoria, haciendo posible que un programa pueda ser la entrada de datos de otro programa.

Unidad aritmético-lógica o ALU Consiste en los componentes que implementan toda la aritmética entera, flotante y de bits, así como las operaciones lógicas que realiza la computadora para realizar cálculos.

Unidad de control Se encarga de controlar el flujo de ejecución del programa, así como del control de errores y situaciones excepcionales. En este componente se encuentra la semántica asociada a las operaciones de salto, bifurcaciones, soporte de excepciones, sincronización, etcétera.

Además de estos, en los procesadores modernos tenemos algunos otros componentes de gran importancia en el funcionamiento de una computadora:

Registros Son pequeñas localidades de memoria de muy alta velocidad (sincronizados a la velocidad del procesador) donde se realizan cálculos en el procesador. Todas las operaciones aritméticas y lógicas utilizan los registros para colocar resultados y recibir operandos.

Caché Su propósito es proporcionar una pequeña porción de memoria de alta velocidad para procesar pequeñas cantidades de datos y código con la finalidad de evitar traer información de la memoria principal. El motivo de usar un caché es que la velocidad de operación de un procesador es generalmente alrededor de 6 órdenes de magnitud más rápida que la memoria principal, no así con un caché, el cual puede ser de menos de 3 órdenes de magnitud más lento que un CPU.

Todos estos componentes se encuentran alambrados en lo que conocemos como CPU (*Central Processing Unit*) o Unidad Central de Proceso la cual se encuentra encapsulada como un circuito altamente integrado también conocido como el *microprocesador* de la máquina. Con la llegada de las arquitecturas multinúcleo (o *multicore*), es posible colocar diversas unidades de procesamiento en un solo empaque de circuito integrado, ofreciendo la posibilidad de dos procesadores físicos en la misma estampa.

El ciclo fetch

Como sabemos, el código ejecutable de un programa se encuentra codificado en lo que conocemos como *código de máquina* y básicamente es un conjunto de instrucciones codificadas numéricamente por medio del sistema de numeración binario. La especificación de los códigos numéricos y organización de las instrucciones está dado de forma única por cada procesador o familia de procesadores; por ejemplo, la familia de procesadores Intel x86 tiene un formato de código compatible hacia atrás desde el 8088 hasta los modernos Intel Core 2 o Intel Core i7, mientras que los procesadores SPARC que se utilizan en los servidores de equipos Sun utilizan una especificación de instrucciones incompatible con la especificación de Intel. A esto se le conoce como la *arquitectura* del procesador.

Para ejemplificar la ejecución de código mostraremos cómo es que un procesador de la familia de Intel realiza la operación de adición de dos números de 32 bits en un registro. Para ello, mostraremos el código de un programa en lenguaje ensamblador que implementa la suma de dos números (aunque no imprime el resultado) y la devuelve como código de salida del programa:

```
B80500          mov ax,5    ; Colocamos el valor 5 en el registro AX
BB1000          mov bx,16   ; Colocamos el valor 16 en el registro BX
01D8            add ax,bx   ; Colocamos en AX la suma de AX + BX
```

Los valores que vemos en la columna izquierda representan los bytes en hexadecimal del código de máquina para Intel que ejecuta esas instrucciones. En la columna central, vemos el código de este programa en lenguaje ensamblador, por lo que la secuencia de bytes `B80500BB100001D8` representa sumar $5 + 16$ y guardarlo en el registro `AX`.

Observando cuidadosamente la columna izquierda podemos ver que las instrucciones de Intel tienen longitud variable: cuando realizamos asignaciones en registros, las instrucciones tienen una longitud de 3 bytes, mientras que la operación de suma consta exclusivamente de dos bytes.

Una vez definida la estructura de una computadora es necesario definir el proceso que emplea el procesador para ejecutar las instrucciones dadas en un programa. Este proceso se conoce como *ciclo fetch* y es una secuencia de pasos fijos. Si bien en las arquitecturas modernas existen muchas sofisticaciones, explicaremos una versión muy sencilla que fue válida hasta la llegada de los procesadores con caché y técnicas de entubamiento (o *pipelining*):

1. **Obtener instrucción de la memoria.** Como mencionamos, tanto el código como los datos del programa se encuentran en memoria, por lo que hay que traer la instrucción actual de la memoria y colocarla en un registro, conocido como *registro de instrucción*. Para saber en qué instrucción vamos, necesitamos de otro registro conocido como *contador de programa*.

2. **Decodificación de la instrucción.** Una vez que tenemos la instrucción en el registro, necesitamos decodificarla para saber qué hay que ejecutar. Generalmente esto se hace por medio de una tabla que mapea códigos de operación en binario (*opcodes*) y modificando el flujo de ejecución del procesador por medio de un circuito de control. La tarea de decodificación es realizada por la unidad de control.

3. **Ejecución de la instrucción.** Una vez que se sabe qué hay que ejecutar, la unidad de control hace uso de la unidad aritmética y lógica para realizar la operación especificada y obtener los resultados. Los resultados generalmente se guardan en un registro.

4. **Procesamiento de resultados.** Una vez finalizada la ejecución de la instrucción, es posible que el resultado se almacene en un registro del procesador o sea transferido a una región en memoria. Al finalizar el proceso de transferencia del resultado de la operación, se procede a actualizar el contador de programa apuntando a la siguiente instrucción a ejecutar.

Conociendo de manera general el funcionamiento de una máquina, es posible especificar cualquier máquina y simularla por medio de un programa. El procedimiento es similar al ciclo fetch, sólo que usamos un programa que simula este proceso y simula algunos componentes de una máquina (específicamente la RAM y los registros). La unidad aritmética y lógica y la unidad de control se simulad, sino que al momento de decodificar la instrucción, utilizamos un lenguaje de alto nivel que realice la traducción e interprete el código.

Llamadas al sistema

Además de las operaciones ofrecidas por el procesador de manera nativa, existen algunas operaciones que ofrece el sistema operativo subyacente. Estos servicios se conocen como llamadas al sistema (*syscalls*, *kernel traps*, o *software interrupts*). Las llamadas al sistema se hacen por medio de un código de operación especial que interrumpe la ejecución de un programa y pasa el control al núcleo (o *kernel*) del sistema operativo, el cual se encarga de proporcionar el servicio. Al igual que una instrucción de CPU, las llamadas al sistema se encuentran codificadas en una tabla, y es necesario proporcionar argumentos para poder ser ejecutadas, de la misma manera que una función o método en un lenguaje de alto nivel.

Para ejemplificar el funcionamiento de un *syscall* proporcionamos el famoso programa *Hello World* escrito en lenguaje ensamblador para Intel x86 con sistema operativo Linux:

```
;; Aqui colocamos los datos que usamos
;; para el programa.  En particular
;; la cadena 'Hello World' seguida de un
;; salto de linea y el caracter nulo que
;; indica su fin.

;; Adicionalmente calculamos su longitud
;; y la guardamos en hwlen
section .data

hwstr: db "Hello World!", 10, 0 ; 'Hello World\n\0'
hwlen: equ $-hwstr              ; $ significa la direccion actual

;; Seccion de codigo.
section .text
   global main        ; habilitamos el procedimiento main para ser
                      ; visto por el procesador
main:
   mov rax, 4         ; 4 es el codigo del syscall para escribir
                      ; un dato en un archivo
   mov rbx, 1         ; 1 es el descriptor stdout (la pantalla)
   mov rcx, hwstr     ; colocamos la cadena a imprimir
   mov rdx, hwlen     ; su longitud
   int 0x80           ; el codigo 0x80 es el syscall de UNIX.
   mov rax, 1         ; el codigo del syscall para salir del programa
   mov rbx, 0         ; el valor de retorno por el programa
   int 0x80           ; ejecutamos el syscall
```

Conociendo la existencia de las llamadas al sistema y cómo funciona un procesador de manera muy general, podemos implementar un simulador que proporcione tanto un conjunto limitado de operaciones de un procesador hipotético, como un conjunto pequeño de llamadas al sistema para ofrecer servicios al usuario.

6.4.3. *Etapa 1:* Construcción dla máquina virtual

Necesitamos una especificación precisa de una arquitectura para poder implementarla. En este apartado mostramos la definición de una máquina virtual *muy simple* con el soporte para llamadas al sistema.

La implementación de la máquina virtual constará de los siguientes componentes:

1. **Registros**. La máquina virtual constará de 14 registros.
2. **Memoria Primaria**. La memoria es simplemente un arreglo de bytes que almacenan datos, por lo cual se puede tratar como tal.
3. **Llamadas al sistema**. La implementación de esta máquina virtual sólo dispondrá de 2 tipos de llamadas: imprimir y leer de consola, haciendo posible la comuniación con el usuario.
4. **Manejo de errores**. En caso de que haya un fallo en la máquina (causado por el programador), si se trata de un error fatal, se debe detener la ejecución de manera controlada. Para reportar errores es necesario detener la ejecución de la máquina virtual y devolver el control al sistema operativo regresando un código de error, además de almacenar un volcado de la memoria (como en las máquinas reales) en un archivo.

Especificación de la máquina virtual

La máquina virtual constará de:

1. **16** *opcodes*. 4 para arimética entera, 4 operaciones de bits, 4 operaciones de memoria, 3 operaciones de salto de instrucción y la instrucción de llamada al sistema.
2. **12** registros. 8 registros para propósito general, dos registros de argumentos para llamada a sistema, un registro de retorno de datos de llamada a sistema y un registro de contador de programa.
3. **7** *syscalls*. 3 instrucciones para leer de consola, 3 para escribir y una para salir del programa.

Registros

La máquina contará con 12 registros de 32 bits cuyo uso se especifica a continuación:

Registros	Descripción
`0`,...,`7`	Registros de propósito general
`8`,`9`	Argumentos para llamada al sistema
`10`	Retorno de datos de llamada al sistema
`11`	Contador de programa

Códigos de operación (*opcodes*)

La máquina virtual deberá implementar los siguientes códigos de operación:

Operación	Código (en hex)	Descripción
add	`0x0`	Suma entera (con signo).
sub	`0x1`	Diferencia entera (con signo).
mul	`0x2`	Producto entero (con signo).
div	`0x3`	Cociente entero (con signo), siendo fatal la división entre cero.
and	`0x4`	Operador de bits `AND`.
or	`0x5`	Operador de bits `OR`.
xor	`0x6`	Operador de bits `XOR`.
not	`0x7`	Operador de bits `NOT`.
lb	`0x8`	Cargar byte.
lw	`0x9`	Cargar palabra (4 bytes).
sb	`0xA`	Guardar byte
sw	`0xB`	Guardar palabra
li	`0xC`	Cargar valor constante
b	`0xD`	Salto incondicional
beqz	`0xE`	Salto si es igual a cero
bltz	`0xF`	Salto si es menor que cero
syscall	`0x10`	Llamada al sistema

Llamadas al sistema (*syscalls*)

Como se menciona en la especificación general, la máquina virtual sólo proveerá llamadas al sistema para leer y escribir en consola. La forma de operar de las llamadas al sistema es igual que en el ejemplo de *Hello World*: cargan un código de llamada al sistema en un registro, un argumento en uno o más registros y el sistema devolverá algún dato en un tercer registro de retorno. Las convenciones de entrada y salida de datos de las llamadas al sistema serán:

1. **Registro para código de llamada:** `8`.
2. **Registro para pasar un argumento a la llamada:** `9`.
3. **Registro donde se reciben datos de la llamada:** `10`.

Los códigos de llamada al sistema serán los siguientes:

- Códigos para leer de consola.

Código	**Significado**	**Retorno** (en registro 10)
`0x0`	Leer entero.	Entero leído.
`0x1`	Leer caracter.	Caracter leído.
`0x3`	Leer cadena.	Número de caracteres leídos. Se debe especificar como en el registro 9 la dirección en memoria donde se guarda la cadena

- Códigos para escribir en consola.

Código	**Significado**	**Argumento** (en registro 9)
`0x4`	Escribir entero.	Entero a escribir
`0x5`	Escribir caracter.	Caracter a escribir
`0x7`	Escribir cadena.	Dirección de memoria donde empezar a imprimir. La cadena se delimita por el caracter nulo `0` (al igual que en `C`).

- Salir del programa.

Código	**Significado**	**Argumento** (en registro 9)
`0x8`	Salir del programa	No utilizado

Codificación de instrucciones

Las instrucciones de la máquina virtual serán de 32 bits (4 bytes) y estarán codificadas de la siguiente manera:

`0`	`8`	`16`	`24`
OpCode	*dr*	*Op1*	*Op2*

Donde *OpCode* representa el código de operación, *dr* representa el número de registro donde guardar el resultado, y *Op1* y *Op2* representan los números de registro de los operandos.

En el caso de las instrucciones `BEQZ` y `BLTZ` la dirección de salto estará en *dr* y el registro a evaluar será *op2*.

Para las instrucciones que sólo tienen un operando (carga y almacenamiento en memoria, saltos de instrucción y el operador `NOT`) sólo se toma en cuenta como operando el valor almacenado en el campo *Op2*.

El caso de la operación `li` es especial, ya que es una instrucción que tiene un tamaño de 6 bytes y su decodificación va como sigue:

0	8	16	24	32	48
`0xC`	*rd*	Constante de 32 bits			

Códigos de error

En caso de ocurrir un error fatal, la máquina virtual deberá finalizar su ejecución devolviendo un código de error que especifica la falla ocurrida. Adicionalmente, se debe guardar un volcado de la memoria primaria en un archivo para propósitos de depuración.

En caso de que se dese devolver mensajes de error, es recomendable enviar mensajes por medio del flujo de datos de error (*stderr*). Será necesario devolver el control al sistema operativo devolviendo un *exit code* como aparece en la siguiente tabla:

Código de Error	Significado
1	División entre cero
2	Dirección de memoria inválida
3	Memoria agotada
4	Número de registro inválido
5	Código de operación inválido
6	Código de llamada a sistema inválido

Requerimientos del simulador

La máquina virtual, además de poder ejecutar programas escritos en el lenguaje de máquina especificado en la sección anterior, deberá implementar las siguientes características:

Argumentos de línea de comando

La invocación de la máquina deberá ser de la forma:

```
$ ./myvm -m 65536 helloworld.bin
```

Donde el argumento `-m` representa el tamaño de la memoria principal (medida en bytes), y el archivo `helloworld.bin` representa un programa escrito en el lenguaje máquina que interpreta el programa.

Si el código de máquina produce un error fatal en la máquina virtual, la ejecución se deberá finalizar inmediatamente y se debe guardar un volcado de la memoria en el archivo `dumpfile.bin`.

6.4.4. *Etapa 2:* Definición del lenguaje del ensamblador

Un ensamblador es un procesador de un lenguaje de bajo nivel que produce código de máquina. A comparación de un compilador de un lenguaje de alto nivel, el ensamblador es mucho más sencillo y las instrucciones se traducen casi directamente a código de máquina. Para definir de manera precisa el lenguaje ensamblador que es soportado por el proyecto, es necesario definir la estructura del mismo, de este modo es necesitario definir los componentes fundamentales del mismo:

Palabras reservadas Palabras que se utilizan para definir construcciones del lenguaje.

Sintaxis La forma como se organizan las palabras y estructura de las sentencias y expresiones del lenguaje de programación. Específicamente se definen por medio de una *gramática*.

Semántica Especifica el significado que tienen las sentencias y expresiones del lenguaje de programación, es decir, explica qué hace cada posible construcción sintáctica.

Palabras reservadas

El ensamblador es sensible a mayúsculas (*case sensitive*) y constará de las siguientes palabras reservadas:

```
add    sub    mul    div
and    or     xor    not
lb     lw     sb     sw
li     b      beqz   bltz
mov    .text  .ascii .asciiz
syscall
```

Construcciones básicas

Comentarios Para marcar comentarios dentro de un programa se deberá hacer por medio del caracter `;`. Todo texto seguido del caracter `;` se debe ignorar.

Espacios en blanco El ensamblador ignorará el espacio en blanco, (espacio o tabulador).

Identificadores Los identificadores (o nombres de variables) se definen exclusivamente por letras y números y deben empezar con letra (se debe soportar mayúsculas y minúsculas).

Literales de cadena Similares a las literales de cadena de `C/C++/Java`. Sólo se soportan los siguientes caracteres escapados:

- `\n` Caracter de nueva línea
- `\t` Tabulador
- `\f` Alimentación de línea
- `\b` Caracter de retroceso (*backspace*)
- `\\` Diagonal invertida (*backslash*)

Nombres de registros Se deberán usar los siguientes nemónicos para nombrar los registros de la máquina virtual:

1. `$r0` ... `$r7` Registros de propósito general.

2. `$a0, $a1` Registros de argumentos para llamadas a sistema.
3. `$r0` Registro de retorno de datos de llamada al sistema.
4. `$pc` Registro contador de programa.

Gramática del ensamblador

Una vez definidos los lexemas (o *tokens*) del lenguaje, podemos definir la organización de los mismos los cuales definirán qué construcciones son sintácticamente correctas. Esto se define por medio de una *gramática*. Una notación muy popular para denotar gramáticas de lenguajes formales es la forma *BNF* (*Backus-Naur form*) la cual define recursivamente la gramática de un lenguaje.

Para entender de manera sencilla el uso de la forma BNF basta con ver un ejemplo sencillo: la gramática de expresiones aritméticas.

```
OPERATOR ::=  '+' | '-' | '/' | '*'
   DIGIT ::=  '0' | '1' | '2' | '3' | '4' |
              '5' | '6' | '7' | '8' | '9'
 INTEGER ::=  DIGIT+
   FLOAT ::=  INTEGER '.' INTEGER
    EXPR ::=  INTEGER
    EXPR ::=  FLOAT
    EXPR ::=  EXPR OPERATOR EXPR
    EXPR ::=  '(' EXPR ')'
```

Al igual que en los lenguajes de programación, las cadenas delimitadas por ' o " representan literales de cadena. En este caso definimos los operadores aritméticos de suma, resta, multiplicación, división, los dígitos y el punto decimal. Estos símbolos se les conoce como *símbolos terminales* ya que definen de manera precisa un elemento de la gramática.

Las demás cadenas de texto (`OPERATOR`, `DIGIT`, etcétera) representan construcciones sintácticas. La primera línea de nuestra definición se podría leer como:

> `OPERATOR` se define como `'+'` ó `'-'` ó `'/'` ó `'*'`.

Por lo tanto, podemos decir que cualquier operador aritmético está definido por el símbolo `OPERATOR`. A estos símbolos se les denomina no-terminales, ya que están en función de otros símbolos, ya sea terminales o no-terminales.

Existen tres operadores para definir no-terminales en la forma BNF:

`|` Operador de disyunción. Define un no-terminal como un conjunto de casos separados por `|`. Es análogo al operador `OR`.

`*` Operador estrella. Es un operador unario y se puede leer como *cero o más veces)*; por ejemplo: `(bla)*` representa las cadenas {"","*bla*","*blabla*","*blablabla*",...}.

`+` Operador de cerradura positiva. Se podría leer como *una o más veces*. Formalmente, si tenemos una expresión $e+$, esta sería equivalente a $ee*$. Si usamos el ejemplo de la cadena `"bla"`, la construcción `(bla)+` sería el conjunto $\{"bla","blabla","blablabla",\dots\}$.

Para ejemplificar, supongamos la expresión:

$$3 \div (2 - 6)$$

Primero, observamos que las cantidades 3, 2 y 6 son dígitos, por lo que están incluidas en el no-terminal `DIGIT`. A su vez, estos dígitos están incluidos en el no-terminal `INTEGER` ya que cumplen con ser uno o más dígitos. Podemos ver que en la definición de `EXPR`, el no-terminal `DIGIT` es parte de su definición, por lo que, todos los números cumplen con ser `EXPR`.

Posteriormente, armamos la expresión $2 - 6$ usando la penúltima regla de la gramática, posteriormente construimos las expresiones parentizadas usando la última regla, y al final unimos el cociente con la penúltima regla.

En general, podemos decir que una gramática induce un árbol de sintaxis, en nuestro ejemplo, el árbol de sintaxis sería:

```
EXPR
   EXPR
      INTEGER
         DIGIT
            3
   '/'
   EXPR
      '('
      EXPR
         EXPR
            INTEGER
               DIGIT
                  2
         '-'
         EXPR
            INTEGER
               DIGIT
                  6
      ')'
```

Habiendo explicado la definición de una gramática por medio de la forma BNF, definimos la gramática del ensamblador como:

```
     PROGRAM ::= DECLARATION* ".text" SUBROUTINE+
 DECLARATION ::= ".ascii" NUM ID |
                 ".asciiz" STR ID
  SUBROUTINE ::= ID ':' INSTRUCTION+
```

```
  INSTRUCTION ::= TRINARYOPCODE REG ',' REG ',' REG |
                  BINARYOPCODE REG ',' REG |
                  "li" REG "," NUM |
                  "li" REG "," ID  |
                  "b" REG |
                  "syscall"
TRINARYOPCODE ::= "add" | "sub" | "mul" | "div" |
                  "and" | "or"  | "xor"
 BINARYOPCODE ::= "not" | "lb" | "lw" |
                  "mov" | "sb" | "sw" |
                  "beqz" | "bltz"
          REG ::= "$r" [0-7] | "$a0" | "$a1" |
                  "$s0" | "$pc"
           ID ::= [A-Za-z][A-Za-z0-9]+
          NUM ::= [0-9]+
          STR ::= '"' <Literal de cadena> '"'
```

Semántica del ensamblador

Podemos describir la semántica del ensamblador analizando cada cláusula de la gramática:

- Un programa está constituido por un conjunto (posiblemente nulo) de declaraciones, seguidos de la palabra reservada `.text` y un conjunto de subrutinas.
- Las declaraciones definen literales de cadena y arreglos de bytes. Las literales de cadena se definen por medio de la palabra reservada `.asciiz` la cual almacena una cadena de caracteres ASCII delimitada por el caracter nulo. Los arreglos se definen por medio de la palabra reservada `.ascii` y definen un arreglo de n bytes especificado después del identificador que lo referencia.
- Una subrutina está definida por medio de un identificador, seguida del caracter `:` y un conjunto no nulo de instrucciones.
- Las instrucciones se pueden definir mediante cinco casos:
  - **Operaciones trinarias:** Son aquellas que requieren tres operandos (registros). El primer operando representa el registro de destino donde almacenar el resultado de la operación y los siguientes dos representan los operandos de la operación a realizar.
  - **Operaciones binarias:** En el caso de la operación `not`, el primer operando representa el destino donde guardar la palabra negada en bytes, y el segundo operando el registro cuyo contenido se va a negar. Para las operaciones de carga y almacenamiento en memoria el primer operando representa el registro donde se almacenarán o

cargarán los datos y el segundo registro almacena la dirección de memoria a emplear. La instrucción `mov` es equivalente a la asignación: colocar en el primer operando el valor del segundo operando. Si bien no existe esta operación en la especificación de la máquina, es fácil simularla por medio de una operación trinaria (por ejemplo, usando la idempotencia del `or`-lógico). Finalmente, las operaciones `beqz` y `bltz` significan salto en caso de ser cero y salto en caso de ser menor que cero, respectivamente. El primer operando representa el registro a probar, y el segundo operando la dirección a saltar.

- **Salto incondicional:** Representado por la palabra reservada `b`, indica saltar a la dirección de memoria especificada en el operando dado.
- **Llamada al sistema:** La palabra reservada `syscall` indica que se debe ejecutar una llamada al sistema.
- **Carga inmediata:** La palabra reservada `li` representa la carga de valores literales a un registro. Existen dos posibles casos:
  - **Carga inmediata de valor literal entero.** Carga el valor entero dado después de la coma en el registro especificado.
  - **Carga de dirección de memoria definida por un identificador.** Carga en el registro especificado la dirección de memoria respectiva al identificador dado (ya sea una declaración o nombre de subrutina).

Arquitectura del ensamblador

Un compilador de un lenguaje de alto nivel está constituido por dos partes esenciales:

- **Frontend.** Este componente se encarga de procesar el código fuente el cual consiste en dos procesos básicos:
  - **Análisis léxico.** El análisis léxico consiste en analizar palabra por palabra de código, ignorar comentarios y espacios en blanco y verificar que todos los lexemas o palabras sean válidos, es decir, palabras reservadas, operadores, signos, identificadores, etcétera. La salida del analizador léxico (o *lexer*) es generalmente una lista enlazada de lexemas ya clasificados para ser consumidos por el analizador sintáctico. En este caso, el conjunto de palabras reservadas, símbolos y operadores se encuentran definidos en la gramática dada en la definición del lenguaje.
  - **Análisis sintáctico.** El análisis sintáctico consiste en validar la gramática del programa, es decir, si la estructura de las palabras es válida bajo la sintaxis del lenguaje de programación. El analizador sintáctico (o *parser*) se encarga tanto de validar la correctud de la gramática como de armar una estructura de datos conveniente para representar la gramática del programa (generalmente un árbol).

- **Backend.** La segunda y más compleja mitad de un compilador. El *backend* se encarga de realizar la traducción a código de máquina. Generalmente, el *backend* consiste de dos componentes principales:

  - **Analizador semántico.** En esta etapa se realizan diversos análisis dependientes del contexto, siendo los más sobresalientes el chequeo de tipos y el alcance de variables. La salida de este componente es un árbol de sintaxis abstracta (similar al del *parser*) "adornado" con información adicional perteneciente a los análisis realizados por el analizador semántico.
  - **Generador de código.** Una vez finalizada la etapa de análisis del lenguaje se procede a realizar la traducción a código de máquina. En la mayoría de los compiladores modernos, esta etapa se divide en dos: la etapa de optimización y la de generación final de código en la que el optimizador encuentra representaciones equivalentes de código con un ahorro de espacio, tiempo o ambas. Al finalizar la etapa de optimización se procede a construir la traducción a código de máquina.

En la implementación del procesador no será necesario hacer un análisis semántico muy complejo, además que el generador de código será muy sencillo ya que estamos compilando un lenguaje de bajo nivel. La intención principal de este proyecto es el uso de estructuras de datos y conocer el funcionamiento general de un compilador sencillo sin necesidad de ahondar mucho en los detalles.

6.4.5. *Etapa 3:* Construcción del analizador léxico

Podemos construir el analizador léxico simplemente leyendo el archivo de entrada y partiéndolo en cadenas delimitadas por los caracteres de espacio en blanco, ignorando las cadenas que empiezan con `';'`. La mayoría de los lenguajes de programación ofrecen este servicio por medio de una biblioteca de manejo de cadenas. Posteriormente debemos validar si las cadenas que recibimos representan identificadores, palabras reservadas o constantes numéricas válidas, y en caso de no ser así reportar el error correspondiente (por ejemplo, un identificador inválido podría ser aquel que empieza con un número).

Otra forma de construir el analizador léxico es por medio de una herramienta que produce automáticamente el analizador. Una implementación muy popular de un generador de analizadores léxicos es GNU `flex`[14].

`flex` recibe de entrada un archivo de texto que especifica los lexemas de nuestra definición del lenguaje y cada que detecta un lexema ejecuta una serie de acciones escritas en lenguaje `C`. Si bien existen diversas implementaciones de `lex` para diversos lenguajes de programación, `flex` está diseñado para trabajar en lenguaje `C`. La salida de `flex` es un programa en `C` que al compilarlo produce el analizador léxico.

[14] `http://flex.sourceforge.net/`

`flex` es una herramienta muy poderosa y se ha convertido en el estándar para construir analizadores léxicos, su uso es sencillo y se ha estandarizado en varios sistemas operativos. El uso de `flex` escapa a los objetivos de este documento, sin embargo, se recomienda al profesor de la asignatura proporcionar una introducción sencilla al uso de `flex`, ya que la estructura del ensamblador es muy sencilla y es muy fácil construir un analizador por medio de `flex` que implemente el analizador léxico del ensamblador del proyecto.

Además de identificar los lexemas del código fuente, es labor del analizador léxico ignorar el espacio en blanco, los comentarios y llevar el conteo de las líneas.

Hasta el momento hemos mencionado cómo identificar los lexemas del lenguaje, sin embargo, debemos también definir cuál es la salida del analizador. Lo más sencillo es devolver una lista enlazada de lexemas (`tokens`) los cuales ya vengan categorizados, es decir, si son identificadores, palabras clave, literales, etcétera. Adicionalmente al lexema, tipo y contenido, debemos agregar el número de línea para que, en caso de haber un error, podamos reportar a detalle donde se encuentra la falla.

6.4.6. *Etapa 4:* Construcción del analizador sintáctico

Habiendo construido el analizador léxico, podemos suponer que recibimos una lista enlazada de lexemas los cuales ya vienen clasificados por tipo (palabra reservada, literal entera, símbolo, operador, etcétera). Si estuviéramos tratando con un lenguaje de alto nivel, la labor del analizador sintáctico sería construir un árbol de sintaxis abstracta (una representación similar al árbol que se muestra en la explicación de la forma BNF), sin embargo, por tratarse de un lenguaje de bajo nivel, podemos construir una representación intermedia muy cercana al código de máquina que se va a generar. En particular, podemos construir una lista enlazada de objetos que representan instrucciones de código de máquina pero que no tienen resueltas las direcciones de los identificadores (declaraciones de variables), ya que eso será tarea del analizador semántico.

Análisis sintáctico por descenso recursivo

La técnica de descenso recursivo es sin duda la más sencilla para implementar un analizador sintáctico; sin embargo, es la más restrictiva en cuanto a la gramática que puede analizar. Para explicar cómo funciona el descenso recursivo, consideremos la siguiente gramática:

```
E  ::= N EP
EP ::= + E |
       - E |
       epsilon
N ::= [0-9]+
```

En esta gramática incluimos un símbolo importante: `epsilon` (ϵ) que representa la cadena vacía. La gramática de arriba sirve para representar expresiones

de sumas y restas de números enteros. Utilizamos ϵ en el caso que estemos usando una expresión que conste exclusivamente de un número, por ejemplo:

$$E \rightarrow NEP \rightarrow 46EP \rightarrow 46\epsilon = 46$$

Una solución en `Java` para implementar el *parser* de descenso recursivo de la gramática propuesta sería:

```
1  import java.util.Scanner;
2
3  public class Parser {
4      // La clase Scanner de Java implementa
5      // el analizador lexico del programa
6      private Scanner sc;
7
8      public Parser(Scanner input){
9          sc = input;
10     }
11
12     // La condicion para que una cadena
13     // de entrada sea expresion es que
14     // cumpla con ser el no-terminal E
15     // y que hayamos consumido completamente
16     // la entrada.
17     public boolean parse(){
18         return (ntE() && !sc.hasNext());
19     }
20
21     // E ::= N EP
22     private boolean ntE(){
23         return (ntN() && ntEP());
24     }
25
26     // N ::= <numero>
27     // Scanner se encarga de analizar
28     // si la expresion dada es un entero.
29     private boolean ntN(){
30         try {
31             System.out.print(sc.nextInt() + " ");
32             return true;
33         } catch(Exception e){
34             return false;
35         }
36     }
37
38     // EP ::= + E | - E | epsilon
39     private boolean ntEP(){
40         // + E | - E
41         if(sc.hasNext()){
42             String t = sc.next();
43             if(!(t.equals("+") || t.equals("-"))){
44                 return false;
45             }
46             System.out.print(t + " ");
47             return ntE();
48         }
49         return true; // epsilon
```

```
50      }
51  }
```

Para probar la clase `Parser` basta con utilizar un `Scanner` en el flujo de entrada estándar (`System.in`).

```
1   public class ParserTest {
2       // Para probar el parser hay que alimentar
3       // una expresion por medio de la entrada
4       // estandar.
5       public static void main(String[] args){
6           Parser p = new Parser(new Scanner(System.in));
7           if(p.parse()){
8               System.out.println("Parse successful");
9           } else {
10              System.out.println("Parse error");
11          }
12      }
13  }
```

Al probar nuestro *parser* con dos posibles cadenas (una válida y otra inválida), el programa devuelve los siguientes resultados:

```
$  echo -n "4 + 3 - 12 + 7" | java Parser
4 + 3 - 12 + 7 Parse successful
```

Ahora, alimentando el *parser* con una entrada inválida tenemos:

```
$ echo -n "4 + hello world" | java Parser
4 + Parse error
```

La labor de este analizador sintáctico es repetir los lexemas que pudo interpretar y se detiene en el lexema inmediato donde la entrada ya no es parte de la gramática a analizar.

Al ver la implementación del *parser*, vemos que la forma de implementarlo es construyendo un método o función por cada no-terminal, y llamando recursivamente a los métodos que analizan cada no-terminal (de ahí el nombre de descenso recursivo) que se encuetran del lado derecho de cada producción. Este tipo de analizador sintáctico es del tipo *top-down* ya que analiza la expresión construyendo el árbol desde la raíz a las hojas.

Sin embargo, la simplicidad de la implementación de un *parser* de descenso recursivo se compensa con que las gramáticas que puede analizar son muy restringidas. Como podemos ver, una gramática más natural para las expresiones aritméticas que constan exclusivamente de sumas y restas hubiera sido:

```
E ::= E + E |
      E - E |
      [0-9]+
```

Desafortunadamente, esta gramática no puede ser analizada por medio de descenso recursivo ya que el no-terminal `E` tiene recursión izquierda, es decir,

se define en términos de sí mismo inmediatamente en el lado derecho de la producción. Si quisieramos implementar el no-terminal `E` con descenso recursivo definido de este modo, se produciría una recursión infinita, ya que para validar si la entrada representa una producción de `E` se llamaría indefinidamente al método que analiza el no-terminal `E` al intentar analizar el operando anterior al `'+'` ó `'-'`.

Esto explica por qué utilizamos la gramática propuesta al implementar el analizador en Java: dicha gramática no tiene recursión izquierda. Es importante mencionar que la recursión izquierda puede no ser directa como en el siguiente ejemplo:

```
A ::= BC
B ::= AD
```

Si fusionamos la producción `B` en `A`, es decir, sustituimos el lado derecho de `B` en todas las ocurrencias de `B` en la producción de `A`, tenemos `A ::= ADC` la cual también adolece del problema de recursión izquierda. Este hecho muestra que analizar si una gramática tiene o no recursión izquierda requiere de un esfuerzo extra a sólo examinar la recursión inmediata en las producciones.

A pesar de esto, es posible construir una gramática libre de recursión izquierda a partir de una gramática con recursión izquierda. Supongamos la siguientes producciones:

$$\begin{aligned} A &\rightarrow A\alpha \\ &\mid \beta \end{aligned}$$

donde A representa un no-terminal y α, β cadenas de terminales y no-terminales. Para eliminar la recursión izquierda directa en la producción, definimos la siguiente gramática equivalente:

$$\begin{aligned} A &\rightarrow \beta B \\ B &\rightarrow \alpha B \\ &\mid \epsilon \end{aligned}$$

La cual elimina la recursión izquierda introduciendo un nuevo no-terminal B.

Para eliminar la recursión izquierda indirecta, podemos crear un algoritmo que utilice fusión de producciones y posteriormente aplicar la transformación de eliminación de recursión izquierda inmediata. Supongamos que tenemos una gramática con no-terminales ordenados de la forma A_i con $i \in \{1, \ldots, n\}$. El algoritmo general para eliminar la recursión izquierda quedaría como:

1. Para toda i desde 0 hasta n
   a) Para toda j desde 0 hasta $i - 1$

> Si $\exists$ la producción $A_i \rightarrow A_j\gamma$ entonces usar fusion de la producción A_j en A_i. En caso de existir recursión izquierda, usar la transformación de recursión izquierda inmediata para eliminarla

Para que una gramática pueda ser analizada por medio de descenso recursivo no basta tener una gramática libre de recursión izquierda, como se puede ver en el siguiente ejemplo:

```
S ::= A 'a' 'b'
A ::= 'a' | epsilon
```

En este caso, el análisis de esta gramática no basta con sólo tener un símbolo no-terminal en el buffer de entrada por parte del analizador léxico, ya que si estamos en la producción `S` y tenemos en el buffer del analizador léxico el no-terminal `'a'` no podemos saber si tenemos que ir a la producción `A` o continuar leyendo la entrada y ver si aparece una `'b'`. Si pudiésemos adelantar dos símbolos en la entrada (lo que se conoce como *lookahead*), podríamos decidir si vamos a `A` o nos quedamos en `S`; sin embargo, entre más símbolos necesitemos adelantar, la complejidad para analizar los casos crece exponencialmente con respecto al tamaño del *lookahead*, ya que hay que analizar potencialmente todas las combinaciones de símbolos no-terminales en el buffer de entrada.

Esta situación está relacionada con el conjunto de símbolos no-terminales con los que puede empezar cada producción, por lo que, podemos checar si una gramática es analizable por descenso recursivo analizando el conjunto de símbolos terminales con los que empieza cada producción; para este efecto, introducimos los conjuntos *First* de cada símbolo por medio del siguiente algoritmo:

1. Si α es terminal, entonces $First(\alpha) = \alpha$.
2. Si α es no-terminal, inicializamos $First(\alpha) = \phi$ y repetimos el siguiente procedimiento hasta que los conjuntos *First* dejen de cambiar: para cada producción $A \rightarrow \beta$ y $\beta = \beta_1\beta_2 \cdots \beta_k$
   a) $First(A) \leftarrow First(A) \cup (First(\beta_1) - \{\epsilon\})$.
   b) Mientras $\epsilon \in First(\beta_i)$ con $i \in \{1, \ldots, k-1\}$
      1) $First(A) \leftarrow First(A) \cup (First(\beta_{i+1}) - \epsilon)$.
   c) Si $\epsilon \in First(\beta_k)$ entonces $First(A) \leftarrow First(A) \cup \{\epsilon\}$.

Los conjuntos *First* denotan los posibles terminales con los que puede empezar cada símbolo de la gramática (inluyendo ϵ).

Además de los conjuntos *First* necesitamos calcular otra clase de conjuntos: los conjuntos *Follow*. Estos conjuntos denotan qué terminales podrían seguir después de un no-terminal que tiene una producción ϵ. Es necesario hacer este análisis, ya que las producciones ϵ generan derivaciones sin necesidad de revisar el flujo de entrar del analizador léxico, por lo que es necesario revisar las producciones posteriores a la producción ϵ y ver qué posibles terminales inician en tal producción (por medio de los conjuntos *First*).

El algoritmo para calcular los conjuntos *Follow* se describe como:

Para todo no-terminal A en la gramática, inicializamos
$Follow(A) = \phi$,
$Follow(S) = eof$
donde S representa el no-terminal inicial de la gramática y eof indica el fin de archivo. Posteriormente repetimos el siguiente procedimiento hasta que los conjuntos $Follow$ dejen de cambiar: Para cada producción $A \leftarrow \beta_1\beta_2 \cdots \beta_k$

1. $Follow(\beta_k) \leftarrow Follow(\beta_k) \cup Follow(A)$
2. $Trailer \leftarrow Follow(A)$
3. Para toda i desde k hasta 2:

   a) Si $\epsilon \in First(\beta_i)$ entonces

   $$Follow(\beta_{i-1}) \leftarrow Follow(\beta_{i-1}) \cup \{First(\beta_i) - \epsilon\} \cup Trailer$$

   En caso contrario

   $$\begin{aligned} Follow(\beta_{i-1}) &\leftarrow Follow(\beta_{i-1}) \cup First(\beta_i) \\ Trailer &\leftarrow First(\beta_i) \end{aligned}$$

Una vez teniendo los conjuntos $First$ y $Follow$ la condición que debe cumplirse para determinar su una gramática se puede analizar por descenso recursivo es: para todo no-terminal A con producciones $A \rightarrow \beta_1|\beta_2|\cdots|\beta_n$ debe cumplirse que:

$$First^+(\beta_i) \cap First^+(\beta_j) = \phi, \forall i, j \text{ tal que } 1 \leq i < j \leq n \tag{6.22}$$

Donde $First^+(\alpha)$ se define como:

1. $First(\alpha)$ si $\epsilon \notin First(\alpha)$
2. $First(\alpha) \cup Follow(\alpha)$ en caso contrario.

Este tipo de gramáticas se conocen como gramáticas $LL(1)$ (*Left to right parsing with Leftmost derivation*) ya que exploran de izquierda a derecha la lista de lexemas y construyen derivaciones por la izquierda. El número uno indica que sólo se utiliza un símbolo de *lookahead* para decidir qué derivación tomar de la lista de producciones de la gramática.

Si bien los algoritmos para determinar si una gramática es o no $LL(1)$ pudieran parecer complejos, resultan fáciles de automatizar, y en la práctica, en caso de diseñar una gramática, siguiendo las siguientes recomendaciones, podemos minimizar la probabilidad de que existen conflictos $First/First$ ó $First/Follow$ (es decir, que una pareja de conjuntos $First$ o $Follow$ tengan intersección no vacía):

1. **Eliminar la recursión izquierda inmediata.** Si bien no aseguramos que no exista recursión izquierda indirecta, en caso de una gramática sencilla (como la del ensamblador) es poco probable que caigamos en este caso.

2. **Factorizar no-terminales comunes en producciones.** Si tenemos la producción:

   ```
   A ::= XA |
         XB
   ```

   Es recomendable factorizar el no-terminal X y separar los sufijos en otra producción:

   ```
   A ::= XS
   S ::= A | B
   ```

   Evitando así un conflicto $First/First$, ya que al factorizar el no-terminal X las producciones de A dejan de tener intersección en conjuntos $First$ debido a la existencia de un factor común.

3. **Fusión de producciones para conflictos** $First/Follow$**.** En caso de que exista intersección no vacía en conjuntos $First$ y $Follow$ de algún no-terminal, podemos resolver el conflicto por medio de fusión de producciones. Si bien este procedimiento puede generar conflictos tipo $First/First$ podemos usar factorización y eliminación de recursión izquierda para resolverlos.

Estructuras de datos necesarias

Para implementar el analizador sintáctico completo, necesitamos algunas estructuras de datos que guarden los siguientes elementos:

1. **Tabla de símbolos.** Guarda los identificadores que va encontrando el analizador léxico. Esta tabla es de uso muy intenso por las etapas posteriores y es recomendable implementarla por medio de una tabla de dispersión. Cada símbolo debe constar de los siguientes campos:

   a) **Id**

   b) **Tamaño** (en bytes).

   c) **Contenido**. En caso de ser una cadena de texto.

   d) **Dirección**.

2. **Tabla de procedimientos.** Representa la lista de subrutinas del programa. Debe almacenar tanto el nombre del procedimiento como la dirección donde se encuentra la primera instrucción del mismo.

3. **Lista de instrucciones.** Codifica la representación intermedia que devuelve el analizador sintáctico después de finalizar el análisis en caso de ser satisfactorio. La lista de instrucciones es una colección de contenedores que encapsulan la información de cada instrucción que será generada por el *backend* del procesador. Cada elemento de la colección debe tener los siguientes campos:

   a) **Código de operación.**

   b) **Códigos de los operandos.**

   c) **Número de línea.** En caso de llamar un identificador inválido, el analizador semántico debe reportar en qué línea ocurrió el error.

   d) **Valores constantes.** En caso de ser una carga inmediata (`li`).

Diseño e implementación del *parser*

Una vez descrita la gramática del ensamblador y la técnica para escribir el analizador sintáctico, falta explicar qué operaciones se deben realizar al reconocer cada símbolo de la gramática. El ejemplo en Java del analizador sintáctico sólo reproduce la entrada que se analiza correctamente, pero nuestro analizador sintáctico debe realizar algunos procedimientos un poco más complejos.

El *parser* debe realizar las siguientes operaciones:

1. **Guardar la lista de declaraciones en la tabla de símbolos** (lo que se encuentra antes del lexema `.text`). Esto se hace al ir reconociendo los no-terminales `DECLARATION`.

2. **Llevar el desplazamiento actual de dirección.** Cada dirección tiene un tamaño de 4 ó 6 bytes, por lo tanto, al ir interpretando cada no-terminal `INSTRUCTION` se debe analizar el tamaño de la instrucción e ir llevando el desplazamiento relativo al inicio de la lista de instrucciones. La razón es que por medio de este conteo, podemos resolver de manera directa las direcciones de cada subrutina.

3. **Llenar la lista de procedimientos.** Como se menciona en la parte anterior, llevando la dirección de desplazamiento actual, al detectar un no-terminal `SUBROUTINE`, podemos almacenar su dirección relativa al inicio de la lista de instrucciones sin necesidad de hacer un análisis adicional.

4. **Poblar la lista de instrucciones.** Al ir reconociendo los no-terminales `INSTRUCTION` construimos un ejemplar de la estructura de datos que almacena la información referente a cada instrucción y la anexamos a la lista de instrucciones.

Para el reporte de errores, basta con reportar la línea de código donde se encontró la falla y además dar un mensaje que explique el motivo de la falla, generalmente serán mensajes del tipo:

```
Línea XX: Se esperaba <Tipo de lexema>
```

6.4.7. *Etapa 5:* Construcción del *backend*

Al finalizar el procesamiento realizado por el *frontend*, el analizador semántico se encarga de llenar las direcciones sin resolver que se encuentran la lista de instrucciones generadas por parte del analizador sintáctico. Finalmente, al haber resuelto las direcciones, es tarea del generador de código traducir las direcciones y colocar la sección de datos embebida en alguna parte del binario del programa.

Analizador semántico

En un lenguaje de alto nivel, el analizador semántico se encarga de realizar el chequeo de tipos y la resolución de nombres de la tabla de símbolos. Sin embargo, para efectos de este procesador, no necesitamos hacer chequeo de tipos y alcance ya que no estamos tratando con un lenguaje de alto nivel. En el caso de un ensamblador, la tarea del analizador semántico es resolver las direcciones de los identificadores que se encuentran en la tabla de símbolos y colocar las direcciones resueltas en las instrucciones que se dejaron incompletas por parte del análisis sintáctico; es decir, las instrucciones de carga inmediata (`li`) cuyo argumento es un identificador.

La entrada del analizador es la lista de instrucciones construidas por el analizador sintáctico, de este modo, el procedimiento que debe serguir el analizador semántico para el ensamblador se describe a continuación:

1. Resolver las direcciones de datos (arreglos y cadenas). Si colocamos esta sección inmediatamente después a la sección de código que va a ser generada a partir de la lista de instrucciones, las direcciones de los identificadores de datos serán el total de bytes de la sección de código más un desplazamiento basado en el tamaño de cada sección, la cual se calcula iterando cada identificador de datos en la tabla de símbolos y calculando su longitud.

2. Al finalizar el proceso de resolución de direcciones, basta con llenar las direcciones que no fueron resueltas por la etapa anterior en la lista de instrucciones; así pues, dicha lista será la entrada a la etapa final: la generación de código.

3. Otro análisis importante es buscar y encontrar el punto de entrada del programa. Siguiendo la vieja convención que el punto de entrada de un programa sea la subrutina o procedimiento `main`, revisamos en la tabla de procedimientos la existencia de dicho procedimiento.

Generador de código

Para efectos de este proyecto, el generador de código es la etapa más sencilla por implementar (contrario a los compiladores de lenguajes de alto nivel, la cual es posiblemente la etapa más compleja). El flujo de trabajo del generador de código se divide en dos partes:

1. **Generación de la sección de código.** Consiste en construir el binario a partir de la lista de instrucciones. Su estructura general es:

   a) **Salto incondicional al procedimiento** `main`. Indispensable para iniciar correctamente el programa.

   b) **Código traducido.** La traducción de direcciones se hace de manera directa: simplemente se revisa en la especificación de la máquina destino la tabla de códigos de operación y códigos de registro y posteriormente se procede a hacer una traducción uno a uno de las instrucciones a código de máquina siguiendo los valores definidos por la tabla. Como las direcciones y datos ya han sido resueltas por parte del analizador semántico, el generador sólo debe armar la sección de código en base a la lista de instrucciones.

   Es importante mencionar que las direcciones que contiene la lista de instrucciones del analizador semántico son relativas el inicio de la lista, por lo que hay que agregarle el desplazamiento resultante de agregar la instrucción de salto a la dirección del procedimiento `main`. Análogamente para las direcciones de las regiones de datos, se debe agregar el desplazamiento producido por toda la sección de código.

2. **Generación de la sección de datos.** Contiene las regiones de memoria correspondientes a literales de cadena y arreglos. Simplemente es un arreglo de bytes producto de concatenar todas las regiones de memoria requeridas por el programa. Lo más recomendable es colocarlas al final de la sección de código.

Bibliografía

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliffor Stein, *Introduction to Algorithms, Second Edition*, MIT Press, 2001.

[2] John L. Hennesy, David A. Patterson, David Goldberg, *Computer Architecture: A quantitative approach, Third Edition*, Morgan Kaufmann, 2002.

[3] Maurice Herlihy, Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[4] Abraham Silverschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts with Java, 7th Edition*, John Wiley & Sons, 2007.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[6] Peter Shirley, *Realistic Raytracing*, A K Peters, 2000.

[7] Cem Kaner, Jack Falk, Hung Q. Nguyen, *Testing Computer Software, 2nd Edition*, John Wiley & Sons, 1999.

[8] Keith Cooper, Linda Torczon, *Engineering a Compiler*, Morgan Kaufmann, 2003.