FACULTAD DE CIENCIAS E INGENIERÍA

Construcción de un Compilador de Asertos de Programación Metódica

Tesis para optar por el título de Ingeniero Informático que presenta:

Diego Berolatti Gonzales

2006409

ASESORA: Claudia María Del Pilar Zapata Del Rio

Lima, 24 de Octubre del 2014

CAPÍTULO 1. PROBLEMÁTICA 5

1.1. Problemática 6

1.2. Objetivo general 8

1.3. Objetivos específicos 8

1.4. Resultados esperados 8

1.5. Herramientas, métodos y procedimientos 9

1.5.1 Descripción de los métodos, herramientas y procedimientos 10

1.6. Delimitación del Proyecto 11

1.7. Justificación y viabilidad 12

CAPÍTULO 2. GENERALIDADES 13

2.1 Marco teórico 13

2.1.1 Lógica de Hoare: 13

2.1.2 Derivación de programas: 14

2.1.3 Programación metódica: ..... **¡Error! Marcador no definido.**

2.2 Estado del arte 19

2.2.1 Forma computarizada aproximada de resolver el problema 19

2.2.2 Procedimientos aproximados para resolver el problema 20

2.2.3 Productos comerciales para resolver aproximadamente el problema 21

2.2.4 Productos no comerciales (de investigación) para resolver aproximadamente el problema 21

2.2.5 Conclusiones sobre el estado del arte 21

CAPÍTULO 3. REGLAS DE LA PROGRAMACIÓN METÓDICA Y DE APLICACIÓN EN UN COMPILADOR 23

3.1. Definiciones 25

3.2. Reglas de Aplicación de la Metodología 26

3.3. Reglas de Derivación (o propuestas de derivación) 29

CAPÍTULO 4. ANALIZADOR LÉXICO, SINTÁCTICO, SEMÁNTICO 38

4.1 Conceptos 38

4.2 Analizador Léxico 39

4.2.1 Lista de tokens: 40

4.3 Analizador Sintáctico 42

4.3.1 Conceptos relacionados a la gramática libre de contexto: 43

4.3.2 Definición de la gramática libre de contexto 43

4.4 Analizador Semántico 45

4.4.1 Árbol de Traducción 45

REFERENCIAS BIBLIOGRÁFICAS 46

Resumen

Siempre ha existido la necesidad de validar la codificación de un programa. Este proyecto tiene como objetivo la implementación de un compilador que, mediante notaciones matemáticas que especifican un programa, genere las instrucciones de manera automática. El resultado tiene como principal característica que es formalmente correcto. Esto se da debido a que el compilador implementa una metodología llamada derivación de programas la cual garantiza esa característica. La implementación de esta metodología se da mediante la estructura formal de un compilador y la inclusión de un autómata adaptativo capaz de aplicar las reglas de programación metódica. El proyecto tiene como alcance el no aplicar ninguna regla que implique resolver un problema de complejidad np. Debido a esto la expresividad del lenguaje y su capacidad de generación automática se encuentra limitada. El resultado es un compilador capaz de generar código de manera automática en base a las especificaciones que el compilador es capaz de compilar. Este proyecto es la base de los compiladores de programación automática.

Introducción

Debido a la gran demanda de conocimientos técnicos o de tecnología y, también, por la gran complejidad que existe en la aplicación de esta, la programación metódica fue dejada de lado como metodología de comprobación de rectitud de programas.En el primer capítulo se describe a fondo como esta problemática justifica la necesidad de un compilador de asertos de programación metódica. Además se describe el objetivo general, los objetivos específicos relacionados a este y finalmente los resultados necesarios para la obtención de estos. En un segundo capítulo se hace un repaso de los principales conceptos teóricos relacionados al proyecto, como son la lógica de Hoare, la derivación de programas entre otros y un resumen del estado del arte en el que se encuentra. Los siguientes capítulos hacen una descripción de cada resultado realizado para obtener cada objetivo específico relacionado a este. Luego se hace una discusión de todo lo obtenido en el proyecto y, finalmente, se llevan a cabo las conclusiones que constatan el cumplimiento del objetivo.

Capítulo 1. Problemática

En el presente capítulo se describe la permanente necesidad que ha existido de contar con una herramienta que pueda garantizar la correctitud de los programas construidos.

En función a esta necesidad se propone un compilador que traduce expresiones matemáticas y que aplica reglas que garantizan la correctitud del programa de manera automática.

Luego se definen los objetivos específicos y los resultados esperados relacionados a estos.Después se describen las herramientas que permiten lograr cada uno de los resultados esperados. Finalmente, se planteanlos límites y viabilidad del proyecto como marco para el desarrollo de la solución.

1.1. Problemática

Existen metodologías, técnicas yherramientas cuya función es la de corregir errores cometidos durante la codificación de un programa de computadora. Sin embargo,éstos no garantizan que el software producido contenga la menor cantidad de imperfecciones posible, excepto aquellos que se basan en las leyes de Charles Richard Hoare [Hoare, 1969]. La lógica de Hoarees un sistema formal que proporciona una serie de reglas de inferencia para razonar sobre la corrección de programas imperativos con el rigor de la lógica matemática y su aplicación garantiza que el programa que las utilice cumpla con las especificaciones que este tenga a priori [Balcazar, 1993].

Para entender estar reglas es importante explicar antes lo que es el triple de Hoare. El triple de Hoaretiene la forma $\{P\}\ C\ \{Q\}$ y es una estructura formada pordosexpresiones matemáticas con un resultado de tipo booleano llamadas asertos (*P* y *Q*) y un comando o instrucción (*C*). El aserto P, llamado precondición, es una expresión que representa el estado del programa antes de ejecutar la instrucción C. Mientras que el aserto Q, llamado postcondición, representa el estado del programa después de ejecutar la instrucción C.

El conjunto de reglas busca en cada momento que la precondición y la postcondición sean verdaderos, en consecuencia el conjunto de instrucciones formadas en C van a cumplir las especificaciones. Esto debido a que las

condiciones iniciales de las especificaciones de C son representadas en la precondición y el resultado de estas en la postcondición. Es por eso que todo programa, que sea capaz de seguir estas reglas o cualquier conjunto de reglas derivadas,es considerado “formalmente correcto”. Según [Balcázar, 1993] estas reglas son tan importantes en el desarrollo de programas como lo son las leyes físicas al momento de construir un edificio o hacer un circuito electrónico.

Una adaptación de estas reglas, llamada programación metódica, permite mediante la inclusión de métodos formales deducir el conjunto de instrucciones C. Esto se logra a partir de definir la precondición y la postcondición en función de las especificaciones deseadas para las instrucciones. Esta forma de construir código a partir de asertos, llamada derivación, conserva las mismas cualidades que la lógica de Hoaredebido a que siguen sus reglas. La programación metódica permite entonces construir instrucciones formalmente correctas, siendo su aplicación una forma de garantizar que el software producido tenga una mínima cantidad de errores.

Sin embargo, de acuerdo a [Feinerer, 2005], debido al corto tiempo del que ahora disponen los programadores para realizar esta tarea y a que el dominio de las nuevas tecnologías se ha convertido en lo más importante durante el desarrollo de un producto de software es que ha existido siempre la necesidad de automatizar técnicas que permiten eliminar errores. Además afirma que la enseñanza en el pregrado de los métodos formales aplicadosen la programación metódicaha disminuido su relevancia frente a otros temas.

Por otro lado los compiladores de los lenguajes más usados actualmente no incluyen alguna técnica automatizada que permita corregir errores en los programas [Feinerer, 2005].

Es entonces que ante esta problemática, se desarrollará un compilador de asertos de programación metódica cuyo fin sea el uso de estos métodos formales de manera automática y cuyo resultado se encuentre en un lenguaje

de alto nivel.De esta forma se obtendrá un programa formalmente correcto con menor esfuerzo de parte del programador.

El proyecto utilizará la estructura general de un compilador convencional adaptando cada una de sus partes en función a lo requerido.La primera parte de este proyecto se va a dar por el análisis de viabilidad y documentación del conjunto de reglas a aplicar. Luego y en función a ellas, se desarrollará un compilador, siguiendo las etapas de análisis léxico, sintáctico y semántico. A continuación, se implementará el traductor automático y finalmente se codificará una extensión de IDE capaz de soportar la compilación de asertos, a través del cual se probará la solución planteada.

1.2. Objetivo general

Implementar un compilador de asertos de programación metódica hacia un lenguaje de alto nivel para la construcción automática de programas formalmente correctos.

1.3. Objetivos específicos

O. 1. Establecer las reglas que el compilador va a ejecutar de manera automática para garantizar la aplicación correcta de la programación metódica.

O. 2. Implementar el analizador léxico, sintáctico y semántico de la notación matemática utilizada en la programación metódica para satisfacer las especificaciones que tenga cada programa generado por el compilador.

O. 3. Implementar el traductor de código intermedio que sea capaz de generar programas en el lenguaje de alto nivel para garantizar la correctitud formal de los programas y su aplicación en el desarrollo de software.

1.4. Resultados esperados

R.1. Mapeo de reglas de la programación metódica existentes. (O.1)

R.2. Conjunto de casos que verifiquen la implementación las reglas definidas en el mapeo. (O1)

R.3. Análisis, clasificación y descripción de la aplicación de cada regla de la programación metódica. (O.1)

R.4. Implementación del analizador léxico. (O.2)

R.5. Implementación del analizador sintáctico. (O.2)

R.6. Implementación del analizador semántico. (O.2)

R.7. Implementación del traductor de código intermedio. (O.3)

R.8. Pruebas de un conjunto de casos que verifiquen la implantación correcta de las reglas en el compilador (O.2, O.3)

1.5. Herramientas, métodos y procedimientos

A continuación se presenta la correlación entre los resultados buscados y las herramientas que permitirán obtenerlos.

Tabla 1. Mapeo de Resultados Esperados

Resultados esperados	Herramientas a usarse
R.1,R2,R3: Mapeo, análisis, clasificación y descripción de las reglas de la programación metódica existentes.	**Análisis, Resumen y Síntesis en Base a Estudios Relacionados al Tema,**es el procedimiento de análisis, resumen y síntesis en base a libros, papers y otros escritos relacionados a un tema en específico.
R.4,R.5,R.6: Diseño e implementación de los analizadores léxico,sintáctico y semántico.	**Modelo de Análisis y Síntesis de la Compilación,** método para definir la estructura general de un compilador y sus partes.
R.4,R.5,R.6: Diseño e implementación de los analizadores léxico,sintáctico y semántico.	**Extended Backus-Naur Form,** método matemático que describe de manera formal un lenguaje [GALLES, 2005]; consta en nombrar un símbolo seguido de las reglas que este representa.

R.4,R.5,R.6: Diseño e implementación de los analizadores léxico,sintáctico y semántico	**Context-Free Grammar,** es un conjunto de componentes y reglas que ayudan en la creación y comprensión de la gramática de un lenguaje fuente.
R7: Diseño e implementación del traductor de código intermedio.	**Autómatas Adaptativos,** es un modelo formal, compuesto por un conjunto de autómatas los cuales son diseñados para describir lenguajes.
R.8, R9: Análisis, diseño e implementación de una extensión de un IDE que soporte la creación de asertos e integre el compilador	**XtremeProgramming,** metodología ágil para el desarrollo de software.

1.5.1 Descripción de los métodos, herramientas y procedimientos

Metodología 1: Análisis, Resumen y Síntesis en Base a Estudios Relacionados al Tema

- Se encarga de transformar una gran cantidad de información en conceptos adecuados para su análisis y comparación. Es el primer paso para la adaptación de las reglas de la programación metódica en el compilador.

Metodología 2: Modelo de Análisis y Síntesis de la Compilación

- Definen la estructura general de un compilador y son esenciales para la elaboración de uno. [GALLES, 2005]

Metodología3: Extended Backus-Naur Form

- El lenguaje definido por ésta gramática representa el conjunto de cadenas de texto que son aceptadas con este lenguaje. Aplicado ya que se piensa desarrollar un lenguaje. [GALLES, 2005]

Metodología 4: Context-Free Grammar

- Útil para el análisis, diseño y creación de la gramática de un lenguaje fuente. Fundamental para la creación de lenguaje del proyecto. [GALLES, 2005]

Metodología 5: Autómatas Adaptativos

- Útil para el análisis y diseño de mecanismos de traducción de lenguajes y para la resolución de cualquier problema en general. [NETO, 1994]

Metodología 6: XtremeProgramming

- Esta metodología propone que el desarrollo del software pueda ser afectado por posibles cambios que surgieran dentro del ciclo de vida del desarrollo del proyecto lo cual lo hace ideal para este tipo de proyectos. [BECK, 2006]

1.6. Delimitación del Proyecto

El proyecto está dirigido a personas con conocimientos básicos en el área de las Ciencias de la Computación. Incluye un compiladorque aplica de manera automática las reglas de la programación.

La primera etapa del proyecto va a ser el del mapeo de reglas. Será de vital importancia para el resto del mismo porque buscará estudiar la aplicación de cada una de las reglas, recolectar casos de prueba para verificar su correcta aplicación y su documentación. Para lograr este estudio, se analizará regla por regla y verificará la posibilidad de aplicarla de manera automática por un computador, descartando solo aquellas que impliquen la resolución de un problema de complejidad NP.

Luego y en base a las reglas ya documentadas, se procederá a implementar el compilador siguiendo la metodología formal de construcción de compiladores. Durante la etapa de construcción del traductor del compilador se utilizará un autómata adaptativo que sirva como modelo del algoritmo de

traducción de estas notaciones hacia un lenguaje de alto nivel. Este lenguaje será definido en este punto en función de las capacidades de traducción.

Finalmente, se implementará una extensión de un IDE el cual pueda ser utilizado para compilar las expresiones hacia el lenguaje de alto nivel. Es entonces que esta extensión integrada con el compilador será capaz de garantizar la correctitud formal de los programas construidos en este. Esta correctitud se encontrará limitada por las reglas de la lógica de Hoare.

1.7. Justificación y viabilidad

La finalidad de este proyecto es la de obtener un producto que sirva como herramienta de desarrollo de software el cual sea formalmente correcto. Para que pueda ser usado por cualquier usuario con conocimientos básicos de programación y matemática, sin necesidad de tener conocimientos avanzados de programación metódica. Los programas generados serán considerados formalmente correctos y se reducirá con esto en gran medida la cantidad de errores que este pueda contener.

Para justificar la viabilidaddel proyecto se realizó un trabajo de aprendizaje sobre la programación metódica, la construcción formal de compiladores, la implementación de autómatas adaptativos y el desarrollo de software. Además del aprendizaje de uso de las metodologías y herramientas a aplicar en el proyecto.

Se cuenta con todos los recursos físicos necesarios para el desarrollo de este proyecto como un computador, acceso a libros y artículos relacionados al proyecto.

Capítulo 2. Generalidades

En este capítulo se profundizará en el desarrollo de los conceptos teóricos relacionados con la lógica de Hoare, la derivación de programas y la programación metódica.

2.1 Marco teórico

Los conceptos a profundizar en esta parte son conocimientos básicos de las reglas de la lógica de Hoare y la derivación de programas. Las ideas de la primera sección son sacadas en mayor parte del ensayo de Hoare [Hoare, 1969] y del segundo del libro de Balcázar [Balcázar, 1993].

2.1.1 Lógica de Hoare:

- Triple de Hoare: es el corazón de la lógica de Hoare, es la comprensión de asertos P y R llamados precondición y postcondición que permiten establecer el estado en el que se encuentra la ejecución de un programa antes y después de la ejecución de una instrucción:

  {P} Q {R}

  Para Hoare esto es interpretado así: si un aserto P es cierto antes de la iniciación del programa Q entonces el aserto R va a ser cierto cuando Q termine.

- Reglas de Inferencia: son el conjunto de reglas que establece la lógica de Hoare y son:
  - Regla de Composición, la inferencia relacionada a esta regla es que si el resultado de la primera parte de un programa es idéntico con la precondición de la segunda parte del programa que produce un resultado,
    entonces todo el programa produce este resultado, siempre y cuando se satisfaga la precondición de la primera parte.
    En términos más formales:

$$Si\ \{\boldsymbol{P}\}\ \boldsymbol{Q_1}\ \{\boldsymbol{R}\}\ y\ \{\boldsymbol{R_1}\}\ \boldsymbol{Q_2}\{\boldsymbol{R}\}\ entonces\ \{\boldsymbol{P}\}\ \boldsymbol{Q_1}; \boldsymbol{Q_2}\ \{\boldsymbol{R}\}$$

- Regla de Iteración: El razonamiento que conlleva esta regla es así; supongamos que P es un aserto que siempre es cierto en cuantos se ejecute S, y este es también cierto al principio. Entonces P va a ser cierto
  después de cualquier número de iteraciones del estado S (incluso sin haber uno). Además se sabe que la condición de la iteración que llamaremos B es falsa cuando la iteración termina. Además B puede ser cierta al iniciar S. Formalmente:

$$Si\ \boldsymbol{P} \wedge \{\boldsymbol{B}\}\ \boldsymbol{S}\ \{\boldsymbol{P}\} entonces \{\boldsymbol{P}\}\ \boldsymbol{while\ B\ do\ S}\ \{\sim \boldsymbol{B} \wedge \boldsymbol{P}\}$$

2.1.2 Derivación de programas:

La derivación de programas es una metodología que permite determinar un conjunto de instrucciones de un programa cuya especificación se encuentra en forma de notaciones matemáticas. Estas notaciones llamadas asertos se rigen por las reglas de lógica de Hoare y mediante la aplicación de estas se puede llegar a un conjunto de instrucciones que cumplen con las especificaciones y que es formalmente correcto. Para esto existen algunas definiciones esenciales:

Definiciones Esenciales

Concepto	**Descripción del Concepto**
Cuantificador	Son formados por variables ligadas y un dominio.
Conjunción	Parte de un cuantificador que se encuentra relacionado a un valor booleano.
Postcondición	Cuantificador que se encarga de explicar la relación con los resultados.
Variable Ligada	Variable vinculada siempre a un cuantificador.
Variable Libre y Constantes	Variable inicial o constante del programa.

Precondicion	Cuantificador que se encarga de imponer las condiciones a los datos.
La instrucción de asignación simple	Corresponde a la modificación de una determinada variable, y por tanto conlleva a una modificación del estado. Se denota con ":=".
La instrucción de asignación múltiple	Corresponde a la modificación de una dos o más variables, y por tanto conlleva a una modificación del estado. Se denota con *"a,b:=c,d"* siendo asignado *a* igual a *b* y *c* a *d* respectivamente.
La instrucción alternativa	Corresponde a la ejecución de una instrucción dependiendo de si su protección se encuentra validada.
Función de cota	También llamada función limitadora se encarga de validar el razonamiento por inducción en los casos de programación recursiva.
Transformador de predicados pmd	Se encarga de transformar la postcondición, con el fin de debilitar la postcondición. La aplicación de este transformador sucesivas veces conlleva a la equivalencia entre la precondicion y la postcondicion. Su notación lógica es la siguiente: $\{\boldsymbol{A_1}\}\,\boldsymbol{C}\,\{\boldsymbol{A_2}\}$ *si y solo si* $\boldsymbol{A_1} \Rightarrow \boldsymbol{pmd(C, A_2)}$

Además se cuenta con los siguientes conceptos de [Balcazar, 1993]:

"El transformador de predicados pmd cuenta con las siguientes reglas:

- *Ley de exclusión de milagros:*

$$\boldsymbol{pmd(P, falso) = falso}$$

- *Ley de distributividad respecto a la conjunción :*

$$\boldsymbol{pmd(P, A_1) \wedge pmd(P, A_2) = pmd(P, A_1 \wedge A_2)}$$

- *Ley de monotonía:*

$$Si\ \boldsymbol{A_1} \Rightarrow \boldsymbol{A_2}\ entonces\ \boldsymbol{pmd(P, A_1)} \Rightarrow \boldsymbol{pmd(P, A_2)}$$

- *Ley de distributividad respecto a la disyunción :*

$$\boldsymbol{pmd(P, A_1)} \vee \boldsymbol{pmd(P, A_2)} \Rightarrow \boldsymbol{pmd(P, A_1 \vee A_2)}$$

Propuestas básicas de derivación de programas:

- *Si en la postcondición aparecen igualdades entre variables del programa y expresiones, puede intentarse satisfacer éstas mediante asignaciones simples o múltiples.*

- *Si en la postcondición aparecen disyunciones, existen varias maneras de satisfacerla. Un criterio útil puede ser intentar diseñar una alternativa, cada una de cuyas ramas obtenga la postcondición a base de satisfacer una de las disyunciones.*

- *Si en la postcondición aparecen conjunciones, puede ser útil considerar cada una de ellas aisladamente, e intentar satisfacerlas por separado; sin embargo, es preciso recordar que al intentar satisfacer otra, lo cual conduce a un proceso de prueba y error en el cual las decisiones tomadas en cada momento siempre pueden ser objeto de reconsideración. Otra posibilidad es la de crear alternativas, en las cuales algunas conjunciones se satisfacen mediante la instrucción protegida y las demás constan en la protección."*

Conceptos de derivación de programas:

Semántica axiomática: representa la definición de un lenguaje de programación al cual se le aplicará la programación metódica. Cuenta con las siguientes partes:

- *La instrucción nula: Denotada también seguir y su semántica viene definida por la siguiente regla:*

$$Para\ un\ aserto\ cualquiera\ \boldsymbol{A_1}, siempre\ es\ cierto\ que \{\boldsymbol{A_1}\}\ \boldsymbol{seguir}\ \{\boldsymbol{A_1}\}$$

- *La instrucción de composición: corresponde a la composición secuencial de otras acciones, para ser ejecutadas en el orden indicado. Se denota separando las acciones que la forman con un punto y coma: $P_1; P_2$ .Su definición semántica es:*

$$Para\ demostrar\ \{\boldsymbol{A_1}\}\boldsymbol{P_1}; \boldsymbol{P_2}\{\boldsymbol{A_3}\}\ es\ preciso\ encontrar\ un\ \boldsymbol{A_2}\ tal\ que$$
$$se\ pueda\ demostrar\ que\ \{\boldsymbol{A_1}\}\boldsymbol{P_1}\{\boldsymbol{A_2}\}\ y\ \{\boldsymbol{A_2}\}\boldsymbol{P_2}\{\boldsymbol{A_3}\}\ .$$

- *La instrucción de asignación simple: representa la modificación de una determinada variable, y por tanto una modificación del estado. Se denota escribiendo la variable que recibe el valor a la izquierda del símbolo "$:=$" y la expresión que denota el valor a asignar a la derecha del símbolo. Para demostrar que :*

$$\{A_1\}\ x := E\ \{A_2\}$$

Además es preciso demostrar:

1. *E puede evaluarse sin errores. Es decir deducir de A_1las condiciones de dominio que aseguren que se puede evaluar la expresión E sin errores.*
2. *$A_1 => A_2[x \leftarrow E]$ .Si en A_2se menciona a x esta tiene que cumplir con que $x \leftarrow E$ .Si en A_2 no se menciona a x entonces $x := E$ se comporta como una instrucción nula.*

- *La instrucción de asignación múltiple: corresponde a la asignación simultánea de varias variables. Dada una cantidad fija de variables distintas $x_1, x_2, \ldots x_n$, expresiones $E_1, E_2, \ldots E_n$, y suponiendo que el tipo de cada expresión coincide con el declarado para la variable correspondiente, se denota:*

$$\langle x_1, x_2, \ldots x_n \rangle := \langle E_1, E_2, \ldots E_n \rangle$$

Para demostrar que

$$\{A_1\}\langle x_1, x_2, \dots x_n\rangle := \langle E_1, E_2, \dots E_n\rangle\{A_2\}$$

basta simplemente con demostrar por cada variable y expresión i entre 1 y n que

$$\{A_1\}x_i := E_i\{A_2\}$$

- *La instrucción alternativa: permite seleccionar acciones a ejecutar en función del estado en que se encuentre el algoritmo. Basado en el concepto de instrucción protegida, define un par formado por una expresión booleana llamada protección y una instrucción cualquiera; si la protección evalúa a cierto diremos que está abierta, y cerrada en otro caso. Solo se ejecuta una instrucción protegida si su protección está abierta. La sintaxis será:*

$$[B_1 \rightarrow I_1$$
$$[]B_2 \rightarrow I_2$$
$$\vdots$$
$$[]B_n \rightarrow I_n$$

]

Donde B_i es de tipo booleano. Se ejecuta evaluando todas las protecciones, si todas están cerradas, la ejecución del algoritmo se aborta, es decir, se detiene en un estado del cual no podemos garantizar ningún aserto; en otro caso se hace una selección indeterminista de una protección abierta y se ejecuta su instrucción asociada.

La razón del indeterminismo es debido a que se deben tomar la menor cantidad de decisiones arbitrarias ya que estas pueden ser definidas por las características del compilador y pueden ser distintas.

Ya que no existe ninguna opción implícita se tienen que tomar todos los casos posibles de tal manera de evitar omitir la consideración de un caso. Para demostrar:

$$\{A_1\}$$
$$[B_1 \ \rightarrow I_1$$
$$[]B_2 \ \rightarrow I_2$$
$$\vdots$$
$$[]B_n \ \rightarrow I_n$$

]

$$\{A_2\}$$

se tiene que probar que :

1. *Se cumplan las condiciones de dominio que justifiquen que todas las protecciones puedan evaluarse o no.*
2. *Que al menos una protección este abierta.*
3. *Que cualquier protección que este abierta tras la ejecución de su instrucción protegida de lugar a un estado que cumpla con A_2*

, lo que significa incluso que ambos pueden ser erróneos. En el caso que alguna operación se encuentre indefinida esta es llamada ecuación de error.

2.2 Estado del arte

A continuación se mencionaran proyectos ya existentes relacionados:

2.2.1 Forma computarizada aproximada de resolver el problema

TheKeYSystem: [Feinerer, 2005] Software de verificación de programas, soporta un subconjunto de estructuras del lenguaje Java llamado JavaCard y cuya verificación está basada en lógica dinámica, una generalización de la lógica de Hoare. Utiliza además para la especificación de objetos Uml y Ocl, No es lo suficientemente expresiva como para especificar el comportamiento de un programa. Además la verificación se realiza después de construir el programa.

2.2.2 Procedimientos aproximados para resolver el problema

a) GuardedCommandLanguage [Dijkstra, 1976]: es un lenguaje definido por EsdgerDijkstra que incorpora la lógica de Hoare mediante una estructura llamado “GuardedCommand” o instrucciones protegidas. Tienen el mismo funcionamiento que las instrucciones protegidas de las instrucciones alternativas de la programación metódica pero se extienden a cualquier condicional del lenguaje y ofrecen una variante que ayuda a garantizar algoritmos determinísticos. Solo integra (instrucciones protegidas) de manera parcial las reglas de la lógica de Hoare.

b) Diseño por contrato [Meyer, 1992]: es una forma de diseñar software, se utiliza la lógica de Hoare en forma de metáfora afirmando que para uno existen obligaciones y beneficios a los cuales uno está ligado mediante un contrato. Mediante esto contrato uno asume roles de proveedor o de cliente buscando en cualquiera de los casos encontrar todas las formas de las cuales las obligaciones y los beneficios de un contrato se garanticen. Se encarga de garantizar el diseño del software pero no garantiza la correctitud formal del programa generado.

c) Asertos embebidos [Rosenblum, 1995]: herramienta desarrollada con el fin de detectar automáticamente errores de ejecución en distintas versiones de un sistema de software. Utiliza asertos los cuales especifican lo que el sistema debería hacer más que él como lo hace.

d) Análisis de programas estáticos [Wichman, 1995]: es el análisis de software sin ejecutarlo, se analiza mayormente el código fuente buscando probar que cumpla con los objetivos que propone el software

2.2.3 Productos comerciales para resolver aproximadamente el problema

a) PerfectDeveloper: [Feinerer, 2005] Es un software de verificación de programas que utiliza un lenguaje llamado Perfect, cuyas características incluyen una herramienta para demostrar automáticamente un teorema y un traductor de Perfect a Java, Ada y C++. Sin embargo al ser este un proyecto muy avanzado se requiere de muchos conocimientos previos relacionados a la verificación de programas para poder utilizarlo.

b) PrototypeVerificationSystem: [Feinerer, 2005] Es un verificador automatico de teoremas que no genera código de programa verificado, sino que prueba automáticamente las propiedades de los algoritmos. Este es muy versátil y necesita de un profundo conocimiento en lógica formal para poder ser utilizado.

2.2.4 Productos no comerciales (de investigación) para resolver aproximadamente el problema

a) FredgeProgramProver: [Feinerer, 2005] Software de verificación de programas, también conocido como FPP incluye estructuras de programas imperativos típicos (como el while, if y case). Solo permite enteros como variables y el lenguaje es muy restrictivo a la hora de enunciar las precondiciones y postcondiciones.

2.2.5 Conclusiones sobre el estado del arte

Las formas y productos que buscan resolver el problema establecen la teoría de la lógica de Hoare de manera implícita en su solución como metodología o lenguaje. Es decir, solo utilizan los conceptos pero no aplican de manera automática estos. Además ninguno incluye la capacidad de derivación automática de programas a partir de asertos. Sino la verificación de estos a través de las reglas.

Es entonces que solo el especialista de estas metodologías o lenguajes puede aplicar los beneficios de las soluciones fuera de los límites de aplicabilidad de las mismas. En cambio la solución propuesta ofrece un mecanismo que, mediante la ejecución automática de las reglas teóricas de la programación metódica, provee una alternativa capaz de aplicar de manera directa estos conceptos sin tener conocimientos profundos sobre la programación metódica. Desde el momento en que uno aprende a construir asertos matemáticos es suficiente como para desarrollar aplicaciones que tengan todos los beneficios que tienen aplicar la metodología.

La importancia de poder aplicar estas reglas en cuestión es tan importante como tener presente la ley de la gravedad para un ingeniero civil; es muy claro hoy en día que la mayoría de desarrolladores tienen la noción que no existen programas sin errores. Siendo una analogía la de afirmar que siempre una casa se caerá, siempre una pista terminara rajada, siempre se caerá el servidor; esta forma de pensar no es más que una consecuencia de trabajo mal hecho; y que, sin embargo, existen formas de garantizar la calidad de lo que se produce y no solo mediante pruebas. La principal razón por la cual la construcción de software es tan costosa es que la mayoría de involucrados desconocen o ignoran la lógica de Hoare o no ven facilidad al aplicarlas. Es entonces que se propone esta herramienta como un esfuerzo para la difusión de las leyes de la lógica de Hoare en forma de compilador de aplicación de reglas de programación metódica de manera automática.

Capítulo 3. Reglas de la Programación Metódica y de Aplicación en un Compilador

A continuación se definirán los tipos de reglas a aplicar en el proyecto, estas reglas se encuentran relacionadas tanto con la estructura del lenguaje, el comportamiento del compilador, las propias reglas de la programación metódica y su aplicación. Estas reglas se encuentran dividas en función de la estructura formal de compiladores de [Aho, 1990] y por las reglas de la programación metódica de [Balcázar, 1993]. Tenemos los siguientes tipos de reglas:

- Reglas para el Lenguaje Fuente y Final:
  - Reglas de Gramática: Relacionados al análisis del conjunto de palabras en un lenguaje, se aplicaran las reglas relacionadas a la construcción de un compilador especificado en el siguiente capítulo.
  - Reglas de Semántica: Relacionadas al análisis de las características de cada palabra en particular, las cuales forman parte del lenguaje, se aplicaran las reglas relacionadas a la construcción de un compilador especificado en el siguiente capítulo.
  - Reglas de Sintaxis: Relacionadas al análisis de la funcionalidad de cada palabra del lenguaje, se aplicaran las reglas relacionadas a la construcción de un compilador especificado en el siguiente capítulo.

- Reglas para el Compilador:
  - Reglas de Aplicación Matemática:
    - Despeje Ecuacional: Se utilizaran las reglas relacionadas al despeje ecuacional de una variable en función a las otras, así como el remplazar una variable por otra.

Ejemplo:

a+4d-2f/c=K +4a (despejar a en función de d,f,c y k)

a+4d-2f/c -a=K +4a –a (igualdad en los lados)

4d-2f/c=K+3a (simplificación)

4d-2f/c-K=3a (igualdad y simplificación)

(4d-2f/c-K)/3=a (cambio de factor)

- Lógica Matemática: Se utilizaran las reglas relacionadas al análisis lógico de una sentencia lógico-matemática.

Ejemplo:

p AND q = p

p AND (p AND q) (reemplazo)

p AND p (simplificación)

p

- Aritmética Matemática: Se utilizaran las reglas de la suma, resta, multiplicación de naturales y enteros.

- Reglas de Aplicación de la Metodología:
  - Propuestas de Deducción: Este tipo de reglas son propuestas que intentan, pero no garantizan, llegar a una solución que lleve a una derivación adecuada.

    Ejemplo:

    “Cuando en la postcondicion tenemos una conjunción de igualdad entre solo dos variables podemos asignar una variable a otra.”

  - Aplicación de la Metodología: Son reglas fijas de la metodología para su aplicación.

Ejemplo:

> "El aserto generado por la derivación de una postcondición ha de ser suficientemente fuerte para garantizar el cumplimiento de esta postcondicion."

3.1. Definiciones

Se ha profundizado en investigar en las reglas de aplicación de la metodología en un compilador, para esto se han establecido algunas definiciones:

Definiciones Esenciales

Concepto	**Descripción del Concepto**
Conjunción	Parte de un cuantificador que se encuentra relacionado a un valor booleano.
Postcondición	Cuantificador que se encarga de explicar la relación con los resultados.
Variable Ligada	Variable vinculada siempre a un cuantificador.
Variable Libre y Constantes	Variable inicial o constante del programa.
Cuantificador	Son formados por variables ligadas y un dominio.
Precondicion	Cuantificador que se encarga de imponer las condiciones a los datos.
La instrucción de asignación simple	Corresponde a la modificación de una determinada variable, y por tanto conlleva a una modificación del estado. Se denota con ":=".
La instrucción de asignación múltiple	Corresponde a la modificación de una dos o más variables, y por tanto conlleva a una modificación del estado. Se denota con *"a,b:=c,d"* siendo asignado *a* igual a *b* y *c* a *d* respectivamente.

La instrucción alternativa	Corresponde a la ejecución de una instrucción dependiendo de si su protección se encuentra validada.
Función de cota	También llamada función limitadora se encarga de validar el razonamiento por inducción en los casos de programación recursiva.

3.2. Reglas de Aplicación de la Metodología

A continuación se listarán y explicarán las reglas de aplicación directa de la metodología, estas reglas se verán implementadas como condicionales imperativas del compilador y deben ser respetadas para la correcta implementación de la programación metódica.

Tabla de Aplicación de la Metodología

Id	**Descripción de la regla**
1	Una variable libre no puede ser asignada.
2	No se pueden asignar constantes a una variable. Este tipo de asignaciones se hacen estas a través de otra variable cuyo valor sea el de la constante.
3	El aserto generado por la derivación de una postcondicion ha de ser suficientemente fuerte para garantizar el cumplimiento de esta postcondicion.
4	Si una conjunción propuesta por un aserto generado es la igualdad entre dos constantes este debe ser descartado.
5	El aserto generado por una instrucción alternativa es la disyunción de todas las protecciones.
6	Si la precondicion es "cierto" se tiene que analizar la postcondicion para cada caso posible igualando el acierto o falsedad hasta que todos los resultados sean cierto.

Descripción

1. "*Una variable libre no puede ser asignada*". Como principal característica de distinción entre las variables libres y ligadas es que, a diferencia de las variables ligadas, las variables libres no pueden ser asignadas a ningún valor.

   Ejemplo: Si se declara

   {Pre: x + y = T}
   {Post: x + y = T AND x = z}
   (Free: z)

   Declarando a z como una variable libre entonces el estado (z:= E) de asignar un valor a z, cualquiera fuera este, no puede pertenecer al conjunto de estados de la solución.

2. "*No se pueden asignar constantes*". A una variable no se le puede asignar directamente el valor de una constante, sin embargo , si una variable posee el valor de la constante y esta igualdad se encuentra establecida en la precondición entonces se puede asignar su valor a través de esta variable.

   Ejemplo:Si se declara

   {Pre: x + y = T}
   {Post: x + y = T AND x = z}
   (Free: z)

   Dado que T es una constante el estado (x:= T) de ser asignada directamente la constante T, cualquiera fuera este, no puede pertenecer al conjunto de estados de la solución.

3. *"El aserto generado por la derivación de una postcondicion ha de ser lo suficientemente fuerte como para garantizar el cumplimiento de esta postcondición*". Eso quiere decir que el aserto generado por la derivación y la ejecución del estado posterior deben ser equivalentes a la postcondición y cumplir que el conjunto de conjunciones del

nuevo aserto tras ser ejecutado el estado debe ser equivalente al conjunto de conjunciones de la postcondición.

Ejemplo:Si el nuevo aserto A1 genera el estado x:=z,:

{A1: z + y = T}

x := z

{Post: x + y = T AND x = z}

Entonces este debe ser lo suficientemente fuerte como para garantizar la postcondición. Para lograr esto, tras ejecutarse el estado, las ecuaciones del aserto "x + y = T" y "x = z" tienen que ser verdaderas, lo cual se cumple en este caso; ya que al asignar x a z y reemplazarla en las ecuaciones del aserto, estas serán equivalentes cumpliendo así con ambas ecuaciones.

4. *"Si una conjunción propuesta por un aserto generado es la igualdad entre dos constantes este debe ser descartado*". Debido a que no se puede garantizar el cumplimiento de este caso. De manera similar si las conjunciones propuestas por un aserto generado incluyen dos igualdades de una variable a distintas constantes entonces este aserto debe ser descartado ya que no se puede garantizar el cumplimiento de esta igualdad.

Ejemplo:Para los asertos A1 o A2:

{A1: x = X AND x = Y}

{A2: X = Y}

En ambos casos no se puede garantizar que X = Y dado que pueden ser iguales como pueden no serlo. Debido a que esto no se puede verificar entonces no se puede utilizar un aserto con esta ecuación.

5. *"El aserto generado por una instrucción alternativa es la disyunción de todas las protecciones*". Al tener una disyunción, entonces existen instrucciones diferentes en función a cada disyunción. Ejemplo:

    Para

    [… → …

    [] … → …

    ⋮

    ]

    {Post: (z = x OR z = y) AND…}

    Se tiene que tener un conjunto de instrucciones por cada alternativa, es decir por lo menos una para z=x y una para z=y.

6. *"Si la precondición es "cierto" se tiene que analizar la postcondicion para cada caso posible igualando el acierto o falsedad hasta que todos los resultados sean cierto*". Al tener una precondición cierto, esta permite cualquier caso posible como inicial, entonces para poder derivar correctamente el programa se tiene que probar que cada conjunción de la postcondición son verdaderos. Ejemplo:

    Para:

    {Pre: cierto}

    y:=z

    x:=z

    {Post: z = x AND z = y}

    Entonces dado los estados y:=z y x:=z se puede decir que para cada caso posible siempre se va a cumplir tras la ejecución de ambos estados que la postcondición es válida.

3.3. Propuestas de Derivación

A continuación se listan y explican las propuestas que se pueden o no aplicar en la resolución de problemas relacionados a la programación metódica. Estos no garantizan la resolución del problema pero proponen una alternativa

de solución. Estas reglas también son las que se aplicarán directamente en el autómata adaptativo para la toma de decisiones de aplicación de reglas:

Tabla 2. Propuestas de Derivación

ID	Regla de Propuesta de Derivación
1	"Cuando en la postcondición tenemos una conjunción de igualdad entre solo dos variables podemos asignar una variable a otra."
2	"Si en la postcondición existe una única variable ligada, siendo las demás libres o constantes, habremos de efectuar una asignación sobre esta variable en función a todas las variables (variables de la precondicion y la postcondicion)"
3	"Si en la postcondición tenemos una conjunción de igualdad entre una variable y una constante y esta constante se encuentra asignada a un variable en la precondicion entonces se puede obtener asignando sobre esta variable la variable de la precondición con la constante asignada."
4	“Si la regla 1 no se puede aplicar en ninguna de las conjunciones se puede asignar la variable a una nueva variable”
5	Si en la postcondición existe disyunción entre dos conjunciones se sugiere entonces por cada una establecer un conjunto de instrucciones alternativas.
6	Si en la postcondición existen conjunciones y ninguna es de igualdad entonces se puede establecer una asignación múltiple de todas las variables ligadas.
7	Si al generar un nuevo aserto este contiene conjunciones que no pueden ser garantizadas por la precondicion entonces se puede utilizar la condición (la conjunción que no puede ser garantizada) como protección.

Descripción:

1. *"Cuando en la postcondición tenemos una conjunción de igualdad entre solo dos variables podemos asignar una variable a otra*". Si en la postcondición nos encontramos que una ecuación, dentro de un conjunto de ecuaciones, es la igualdad entre dos variables entonces son propuestas de estado la asignación de una variable a otra. Cualquiera de ellas puede ser la asignada siempre y cuando se cumplan con las reglas de aplicación de la metodología.
   Ejemplo:

   {Post: … AND x=z}

   Siendo propuestas válidas"z:=x"y "x:=z"

2. *"Si en la postcondición existe una única variable ligada, siendo las demás libres o constantes, habremos de efectuar una asignación sobre esta variable en función a todas las variables (variables de la precondicion y la postcondición)".* Si en la postcondición dentro de cada ecuación y conjunción existe solo una variable ligada, es entonces la asignación a esta variable ligada una propuesta de estado de derivación.
   Ejemplo:

   {Pre: x+y=T}

   {Post: z+y=T}

   Siendo x una variable que no se encuentra en la postcondición, z una variable libre y T una constante entonces: y:= E(x,y,z) es una propuesta.

3. *"Si en la postcondición tenemos una conjunción de igualdad entre una variable ligada y una constante y esta constante se encuentra asignada a un variable en la precondición entonces se puede efectuar una asignación sobre esta variable con la variable de la precondición la cual tiene a la constante asignada".* Si en la postcondición existe

dentro de una ecuación una igualdad entre una variable ligada y una constante, es una propuesta la asignación a esa variable del valor de la constante, el cual sea igual al valor de otra variable en la precondición.

Ejemplo:

{Pre: x=X}

{Post: y=X}

Debido a que la variable y es equivalente a la constante X en la postcondición y a que existe en la precondición una variable que es equivalente a esta constante, entonces y:=x seria la propuesta generada por esta regla.

4. *"Si la regla 1 no se puede aplicar en ninguna de las conjunciones se puede asignar la variable a una nueva variable "*. Si por alguna razón la regla 1 o la regla de igualdad entre dos variables no se puede aplicar o no lleva a ningún resultado, se puede aplicar asignar la variable a una nueva variable creada.

Ejemplo:

{Post: x=y}

Asumiendo que no se puede aplicar la regla uno entonces se propone hacer x:= z siendo z una variable nueva.

5. *"Si en la postcondición existe disyunción entre dos conjunciones se sugiere entonces por cada una establecer un conjunto de instrucciones alternativas"*. Si la postcondición está conformada por disyunciones de conjunciones entonces es una propuesta un conjunto de instrucciones alternativas donde para cada una se le proponga cada disyunción por separado.

Ejemplo:

Si la postcondición es:

{Post: (z = x OR z = y) AND…}

Entonces se propone,

[… →… para el aserto {z = x AND …} y

[]…→… para el aserto {z = y AND …}

6. *"Si en la postcondición existen conjunciones y ninguna es de igualdad entonces se puede establecer una asignación múltiple de todas las variables ligadas"*. Si la postcondición está conformada por conjunciones y ninguna ecuación es de igualdad entonces se propone una asignación múltiple a cada variable ligada.

   Ejemplo:

   Si a y b son variables iniciales:

   {Pre: a= 2*b*c + r AND r< 2*b}

   {Post: a= b*c + r AND r< b}

   Entonces se puede intentar hacer una asignación múltiple con ambos:

   {Pre: a= 2*b*c+r AND r< 2*b}

   c,r:=E1,E2

   {Post: a= b*c+r AND r< b}

7. *"Si al generar un nuevo aserto este contiene conjunciones que no pueden ser garantizadas por la precondición entonces se puede utilizar la condición (la conjunción que no puede ser garantizada) como protección"*. Es una forma de establecer las protecciones de las instrucciones alternativas, se establecen como condiciones para la aplicación de la instrucción, lo cual es a su vez la nueva postcondición. La disyunción de todas representan el nuevo aserto generado por el conjunto de instrucciones alternativas.

Ejemplo:

{Pre: a= 2*b*c+r AND r< 2*b}

[r<b → c:= 2*c

[] ? → …

]

{Post: a= b*c+r AND r< b}

Dado que el nuevo aserto de c:= 2*c es r<b este actúa también como protección de la instrucción. Siendo a un vector, x un entero y k un numero natural.

Durante el desarrollo de este resultado obtenido se lograron recopilar las reglas para la aplicación de la programación metódica, siendo uno de los resultados más complejos debido al trabajo de recolección, análisis y síntesis de cada regla, al igual que su clasificación y generalización. Durante este desarrollo se llegó a la conclusión de no incluir las reglas relacionadas a estructuras de datos debido a que en la mayoría de casos las reglas se aplicaban de manera arbitraria.

En particular las propuestas de derivación se caracterizan por tener la capacidad de resolver una gran cantidad de problemas si se aplican en combinación con las demás reglas pero también por ser las más difíciles de implementar.

Capítulo 4. Conjunto de casos de verificación.

A continuación se encuentran la lista de casos que el compilador pretende resolver. Estos casos fueron tomados en algunos casos del libro de Balcazar [Balcazar, 1993] y del libro deKaldewaij [Kaldewaij, 1990] y en otros casos son variaciones de estos. Las soluciones de los casos fueron construidas siguiendo la metodología planteada por la programación metódica. Debido a que existe más de una respuesta correcta lo que se busca es tener un resultado equivalente al planteado en los casos. Cada caso tiene dificultad distinta debido a que se busca evaluar distintas partes del compilador.

Tabla 3 Casos De Verificación

Identificador	Condición de Entrada		Solución
ID01	Variable libre	z	y:=x+y-z
	Precondición	{x+y=T}	x:=z
	Postcondición	{x+y=T AND x=z}	
ID02	Variable libre	-	z:=y
	Precondición	{x+y=T}	
	Postcondición	{x+y=T AND x=z}	
ID03	Variable libre	-	a:=x
	Precondición	{x=X AND y=Y}	y:=x
	Postcondición	{x=Y AND y=X}	x:=a
ID04	Variable libre	-	a:=y
	Precondición	{x=X AND y=Y AND z=Z}	y:=z
	Postcondición	{x=Y AND y=Z AND z=X}	z:=x
			x:=a
ID05	Variable libre	x, y	[] x>=y -> z:=x
	Precondición	{cierto}	[x<y -> z:=y
	Postcondición	{(x=z OR z=y)AND (x<=z AND y<=z}	]
ID06	Variable libre	x, y	[] x<=y -> z:=x
	Precondición	{cierto}	[x>y -> z:=y
	Postcondición	{(x=z OR z=y)AND (x>=z AND y>=z}	]
ID07	Variable libre	-	x:=y
	Precondición	{y=Y}	
	Postcondición	{x=Y}	
ID08	Variable libre	x	x:=y

	Precondición	{cierto}	
	Postcondición	{x=y}	
ID09	Variable libre	z	y:=x-y-z
	Precondición	{x-y=K}	x:=z
	Postcondición	{x-y=K AND x=z}	
ID10	Variable libre	-	-
	Precondición	{cierto}	
	Postcondición	{cierto}	
ID11	Variable libre	-	b:=0
	Precondición	{cierto}	x:=0
	Postcondición	{x=SUMA(a,0,N)}	while (b<N){
			x:=a(b)+x
			b:=b+1
			}
ID12	Variable libre	-	b:=0
	Precondición	{cierto}	x:=0
	Postcondición	{x=MULT(a,0,N)}	while (b<N){
			x:=a(b)*x
			b:=b+1
			}
ID13	Variable libre	-	c:=0
	Precondición	{cierto}	while (c<N){
	Postcondición	{x=CONT(a,0,N)}	c:=c+1
			}
			x:=c
ID14	Variable libre	-	b:=0
	Precondición	{cierto}	c:=0
	Postcondición	{x=MAX(a,0,N)}	x:=a(0)
			while (b<N){
			c:=a(b)
			[]x<c-> x:=c
			[x>=c ->seguir
			]
			b:=b+1
			}
ID15	Variable libre	-	b:=0
	Precondición	{cierto}	c:=0
	Postcondición	{x=MIN(a,0,N)}	x:=a(0)
			while (b<N){

			c:=a(b) [] x>c -> x:=c [x<=c ->seguir] b:=b+1 }

ID 16	Variable libre	-	x:=1
			c:=0
			while (c<N AND x=1){
			[]K=a(c) ->c:=c+1
			[K<>a(c)-> x:=0
			]
			}

	Precondición	{cierto}	
	Postcondición	{x=TODO(a,0,N,=,K)}	

ID 17	Variable libre	-	x:=0
			c:=0
			while (c<N AND x=0){
			[] K=a(c) ->x:=1
			[K<>a(c) ->c:=c+1
			]
			}

	Precondición	{cierto}	
	Postcondición	{x=ALME(a,0,N,=,K)}	

ID 18	Variable libre	-	x:=1
			c:=0
			while (c<N AND x=1){
			[]a(c)>=K->c:=c+1
			[a(c)<K-> x:=0
			]
			}

	Precondición	{cierto}	
	Postcondición	{x=TODO(a,0,N,>=,K)}	

ID 19	Variable libre	-	x:=0
			c:=0
			while (c<N AND x=0){
			[]a(c)<K->x:=1
			[a(c)>=K->c:=c+1
			]
			}

	Precondición	{cierto}	
	Postcondición	{x=ALME(a,0,N,<,K)}	

Capítulo 5. Analizador Léxico, Sintáctico, Semántico

A continuación se presentan conceptos relacionados a la construcción de compiladores. Conceptos necesarios para la comprensión de este capítulo.

4.1 Conceptos

Un compilador generalmente se divide en dos partes, la primera parte dedicada al análisis del leguaje base y la segunda parte dedicada a la síntesis de este y la posterior transformación al lenguaje final [Gómez, 2012]:

De la primera parte:

- Análisis Léxico: es el que divide el código fuente en componentes básicos del lenguaje a compilar. Estos componentes básicos (llamados tokens) son una secuencia de caracteres del programa fuente y pertenecen a una categoría gramatical: números, identificadores de usuario (variables, constantes, tipos, nombres, nombres métodos), palabras reservadas, signos de puntuación, etc.

- Análisis Sintáctico: comprueba que el programa fuente respete las directrices del lenguaje que se compila.

- Análisis Semántico [Aho, 1990]: revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones. Un componente importante del análisis semántico es la verificación de tipos, en donde el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente.

De la segunda parte:

- Generación de código Intermedio: genera un código independiente de la maquina muy parecido al ensamblador. Se puede considerar un programa para una máquina abstracta.
- Generación de código final: crea el bloque de código final en base al código intermedio.

Generalmente, se suele agrupar las tres fases del análisis y la fase de generación de código intermedio en una gran fase llamada "**frontend"**, en donde se depende del lenguaje fuente pero no de la plataforma; y las dos últimas de la síntesis (optimización y generación de código máquina) constituyen el "**back end"**, ya que son independientes del lenguaje fuente pero si dependientes directamente de la plataforma.

4.2 Analizador Léxico

La función específica del analizador léxico es la de convertir las líneas de texto de entrada en una lista de tokens. Para lograr esto es necesario profundizar en la estructura general del lenguaje fuente. Esta estructura se encuentra formada por 3 elementos:

(1) *(definición de variables no ligadas)*

(2) *{estructura del aserto correspondiente a la precondición}*
{estructura de aserto correspondiente a la postcondición}

(3) *…*

La primera parte (1) de la estructura se va a encontrar formada por dos paréntesis y entre ellos la definición de variables no ligadas separadas por comas.

Ejemplo:

$$(a, b, c)$$

En este ejemplo las letras "*a"*, "*b"* y "*c"* representan las variables no ligadas.

La segunda (2) parte de la estructura van a estar formada por una dupla de notaciones matemáticas cada una entre dos llaves.

Ejemplo:

*{ a= 2*b*c+r AND r< 2*b}*

*{ a= b*c+r AND r< b}*

La estructura de las notaciones matemáticas va a ser la estructura clásica de formación de notaciones matemáticas a excepción de los símbolos relacionales “^” y “v”. Estos símbolos van a ser reemplazados debido a la dificultad de escribirlos en un editor de textos convencional.
La tercera (3) parte se va a encontrar formada por un número finito de duplas de asertos de la forma definida ya en la segunda (2) parte.

A continuación el analizador se va a encargar de convertir las líneas de texto de entrada en una secuencia de tokens. Este analizador léxico forma parte de la construcción del compilador de traducción de asertos. Los asertos están representados por los tokens. Estos tokensy las reglas de identificación de los mismos se encuentran definidos a continuación.

4.2.1 Lista de tokens:

Para el desarrollo del analizador léxico es importante describir los tokens a analizar. Estos tokens se encontraran en la lista de variables libres y en los asertos. Los tokens a ser analizados por el compilador son:

Tabla 4. Tabla de tokens

Id	Ejemplo de Gramática	Nombre	Nombre del Token	Descripción
1	"x,y,z,a,b,c"	Variables	VARIABLE	Representado por las letras minúsculas del abecedario del idioma inglés.
2	"T,X,Y,Z,A,B"	Constantes	CONSTANTE	Representado por las letras mayúsculas del abecedario del idioma inglés.

Id	Ejemplo de Gramática	Nombre	Nombre del Token	Descripción
3	"+,-,*,/"	Símbolos aritméticos	SUMA, RESTA, MULT, DIVI	Representados por los símbolos especificados en el ejemplo.
4	"AND, OR"	Símbolos lógicos relacionales	AND, OR	Representados por las palabras reservadas especificadas en el ejemplo.
5	">= ,<= ,> ,< ,="	Símbolos lógicos comparadores	MAYORIGUAL, MENORIGUAL, MENOR, MAYOR, IGUAL.	Representados por las palabras reservadas y los símbolos especificados en el ejemplo.
6	cierto	Palabra afirmativa	CIERTO	Representado por la palabra reservada “cierto”.
7	1, 2, 3…	Números enteros	ENTERO	Representado por los números naturales y enteros.

Id	Ejemplo de Gramática	Nombre	Nombre del Token	Descripción
8	“,”	Coma	COMA	Representado por el símbolo “,”.
9	“{” , “}”	Llaves	LLAVEIZQ, LLAVEDER	Representado por los símbolos “{” y “}”.
10	“(” ,”)”	Paréntesis	PARIZQ, PARDER	Representados por los símbolos “(” y “)”
11	SUMA,MULT, CONT,MAX,MIN,TODO,ALME	Palabra SUMA,MULT, CONT,MAX, MIN,TODO,ALME	CUANT_SUMA,CUANT_MULT,CUAN_MAX,CUANT_MIN,CUANT_TODO,CUANT_ALME	Representado por la palabra reservada “SUMA”,”MULT”,”CONT”,”MAX”,”MIN”,”TODO”,”ALME”.

Para la formación de este analizador en Java se utilizó la herramienta FLEX y se especificó cada token para cumplir con las características de este proyecto. En el anexo A se incluyen las reglas aplicadas en lenguaje .flex.

4.3 Analizador Sintáctico

El siguiente paso en el proceso de compilación es determinar si la secuencia de tokens formada es sintácticamente correcta. Para esto es necesaria la definición de una gramática libre de contexto.

4.3.1 Conceptos relacionados a la gramática libre de contexto:

Es un conjunto de reglas utilizadas para verificar la validez sintáctica de una secuencia de tokens en un compilador. Según [Galles, 2005] se encuentra formada por cuatro componentes:

- Un conjunto de símbolos terminales T, los cuales son tokens del lenguaje.
- Un conjunto de símbolos no terminales N.
- Un conjunto de reglas de equivalencia R, las cuales están formadas en su lado izquierdo por un símbolo no terminal y en el lado derecho por una secuencia de símbolos terminales y no terminales.
- Un símbolo especial S no terminal que pertenece a N, el cual es el símbolo de inicio.

El objetivo de la gramática libre de contexto es el de generar una cadena de tokens que cumplan con el conjunto de reglas de igualdades R. Esto se logra de esta manera:

- Se empieza como cadena base el símbolo especial S no terminal.
- Mientras exista símbolos no terminales en la cadena se procede a aplicar una regla del conjunto de reglas de equivalencia R.

Por ejemplo esta es la formación de una gramática libre de contexto:

Terminales {id, num, if, then, else, print, =,{, }, (,) }

No-Terminales {S,E,B,L}

Reglas (1) S →print(E);

(2) S→while(B) do s

(3) S→ {L}

(4) E→ id

(5) B→ E > E

(6) L → S

(7) L → S L

Símbolo inicial S

4.3.2 Definición de la gramática libre de contexto

Esta gramática sirve para establecer las especificaciones necesarias para cumplir con las reglas de formación de asertos de la programación

metódica.Para la realización del analizador sintáctico se va a definir la siguiente gramática libre de contexto.

Terminales:

{AND,OR,"{","}","(",")",MAYORIGUAL,MENORIGUAL,MAYOR,MENOR,IGUAL ,SUMA,RESTA,MULT,DIVI,CIERTO, PARIZQ, PARDER,CONSTANTE, ENTERO,VARIABLE,CUANT_SUMA,CUANT_MULT,CUAN_MAX,CUANT_M IN,CUANT_TODO,CUANT_ALME}

No-Terminales:

{PROGRAM, DEFVAR, DUPLAS, ASERTO, LISTAVAR, ECUACION, RELACION, EXPRESION, COMPA, MODIFIC, VALOR}

Reglas:

(1) PROGRAM → DUPLAS

(2) PROGRAM→ DEFVAR DUPLAS

(3) DUPLAS→ LLAVEIZQ ASERTO LLAVEDER LLAVEIZQ ASERTO LLAVEDER DUPLAS

(4) DUPLAS→ LLAVEIZQ ASERTO LLAVEDERLLAVEIZQ ASERTO LLAVEDER

(5) DEFVAR→ PARIZQ LISTAVAR PARDER

(6) LISTAVAR → VARIABLE COMA LISTAVAR

(7) LISTAVAR → VARIABLE

(8) ASERTO→ ECUACION RELACION ASERTO

(9) ASERTO→ ECUACION

(10) RELACION → AND

(11) RELACION → OR

(12) ECUACION→ EXPRESION COMPA ECUACION

(13) ECUACION→ EXPRESION

(14) ECUACION→ PARIZQ ECUACION PARDER

(15) ECUACION→ CIERTO

(16) ECUACION→ CUANTIFICADOR

(17) CUANTIFICADOR→ VALOR IGUAL CUANT_TRIP

(18) CUANTIFICADOR→ VALOR IGUAL CUANT_CUAT

(19) CUANT_TRIP→CUANT_SUMA (VALOR,VALOR,VALOR)

(20) CUANT_TRIP→CUANT_MULT (VALOR,VALOR,VALOR)

(21) CUANT_TRIP→CUANT_CONT (VALOR,VALOR,VALOR)

(22) CUANT_TRIP→CUANT_MAX (VALOR,VALOR,VALOR)

(23) CUANT_TRIP→CUANT_MIN (VALOR,VALOR,VALOR)

(24) CUANT_CUAT→CUANT_TODO (VALOR,VALOR,COMPA,VALOR)

(25) CUANT_CUAT→CUANT_ALME (VALOR,VALOR,COMPA,VALOR)

(26) COMPA→ MAYORIGUAL

(27) COMPA→ MENORIGUAL

(28) COMPA→ MAYOR

(29) COMPA→ MENOR

(30) COMPA→ IGUAL

(31) EXPRESION→VALOR MODIFIC EXPRESION

(32) EXPRESION→VALOR

(33) EXPRESION → PARIZQ EXPRESION PARDER

(34) MODIFIC→ SUMA

(35) MODIFIC→ RESTA

(36) MODIFIC→ MULT

(37) MODIFIC→ DIVI

(38) VALOR→ CONSTANTE

(39) VALOR→ENTERO

(40) VALOR→VARIABLE

Símbolo inicial PROGRAM

Para la formación de este analizador en Java se utilizó BYACC y se especificó cada terminal y no terminal para cumplir con las características de este proyecto. En el anexo B se incluyen las reglas aplicadas en lenguaje .y .

4.4 Analizador Semántico

El analizador semántico se encarga de formar el árbol de traducción y la tabla de traducción a partir de la gramática libre de contexto y los tokens definidos anteriormente. Son el paso previo a la traducción en si del código fuente.

4.4.1 Árbol de Traducción

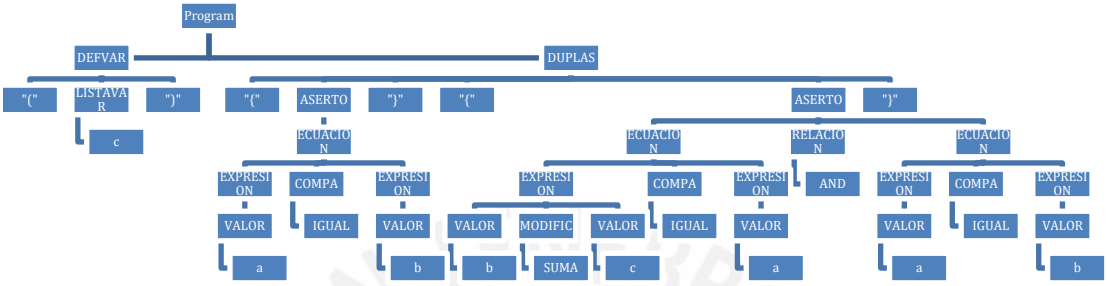
El arból va a estar formado a partir de un código fuente específico. Por ejemplo si tenemos:

(c)

{a=b}

{b+c=a AND a=b}

El árbol va a estar formado de esta manera:



Para formar este árbol se crearon las clases necesarias que soporten esta estructura. Por ejemplo:

- Para los valores se diseño una clase llamada Valor la cual tiene como atributos el tipo de valor y su valor en forma de variable o constante según sea el caso.
- Para las ecuaciones se diseño una clase llamada Ecuación la cual esta formada por dos clases llamadas Expresion (que forman una expresión) y un enumerador de tipo Comparador que representa el comparador de la ecuación.

Capítulo 6. Implementación del traductor de código intermedio.

Para la implementación del traductor de código intermedio (del árbol de objetos formado por los analizadores), se utilizó un algoritmo denominado autómata adaptativo; la conceptualización de este algoritmo se presentara en la primera parte, luego se explicara su aplicación en el traductor y, por último, se describirá otros algoritmos utilizados en la ejecución del traductor.

6.1 Autómata Adaptativo:

Un autómata adaptativo es un modelo formal, compuesto por un conjunto de autómatas los cuales son diseñados para describir lenguajes. Se compone por un conjunto de autómatas, denominados submaquinas. Cada una de estas submaquinas le correspone efectuar una tarea de reconocimiento en función de las distintas clases de cadenas que componen el texto de entrada. A partir de una configuración inicial fija , cada reconecedor intermedio se obtiene del anterior por medio de autotransformaciones del texto de entrada, estas transformaciones se dan de manera gradual de acuerdo a las necesidades de análisis. Este comportamiento se logra insertando y quitando estados. Cada una de estas modificaciones se aplica en paralelo.

6.2 Aplicación del Autómata Adaptativo

Debido a que este modelo se usa normalmente para la lectura de lenguaje natural se tuvo que adaptar el algoritmo para cumplir con las necesidades planteadas. Para este proyecto a cada submaquina se le fue asignada una propuesta de derivación. El trabajo de cada submaquina es que, en función a los asertos generen propuestas de derivación en forma de estados. Luego a partir de cada uno de estos estados se generan nuevos asertos (debilitando la postcondición) llegando a una solución parcial. Esto genera por cada estado una lista de parejas de asertos (o duplas) los cuales son registrados en una tabla. Cada una de estas duplas son comparadas, si estas son equivalentes se devuelve el conjunto de estados (instrucciones), en otro caso se vuelve a aplicar el

análisis de las submaquinas siempre y cuando esta dupla no haya sido analizada antes. En resumen:

1. Se crea una dupla en función de dos asertos.
2. Se valida la equivalencia de los dos asertos.
3. Si son equivalentes se devuelve la lista de estados involucrados en la transformación de los asertos.
4. Si no son equivalentes y no han sido analizados antes se aplica el análisis de las submaquinas.
5. Se registra la dupla en la lista de duplas ya analizadas.
6. Se recibe cada resultado parcial en forma de lista de estados de cada submaquina.
7. Por cada uno de estos estados y en función de la dupla actual se genera una nueva dupla.
8. Por cada nueva dupla formada que no haya sido analizada se aplica el paso 2.
9. El algoritmo termina si no se cuenta con duplas que no hayan sido analizadas con anterioridad con resultado de error de capacidad de derivación.

6.2 Otros algoritmos involucrados

1. Algoritmo de análisis de equivalencia de asertos, este algoritmo se encarga de transformar los asertos (árbol de objetos) a una estructura capaz de hallar la equivalencia de dos asertos. En una primera etapa se encarga de formar una lista de afirmaciones. La afirmación es una estructura capaz de representara una ecuación en un aserto y su principal característica es que cada una representa a un conjunto de ecuaciones equivalentes. En algunos casos una ecuación puede ser representada por dos afirmación ya que esta ecuación podría representar dos ecuaciones (por ejemplo para las ecuaciones con comparador >= y <=). En una segunda etapa y en función de cada relación existente entre las ecuaciones se forman listas de afirmaciones o verdades para luego formar una lista de verdades por cada aserto. La principal

característica de la lista de verdades es que basta que una verdad sea igual a la verdad de otro aserto para los asertos sean considerados equivalentes.

2. Algoritmo de despeje y simplificación de ecuaciones, se encarga de comparar ecuaciones y aplicar el despeje y simplificación de cada ecuación. Se utiliza tanto en la formación de nuevos estados como en la generación de nuevas postcondiciones.
3. Traductor de estados, al finalizar el autómata adaptativo de manera satisfactoria, la lista de estados (lista de objetos) se traduce en un lenguaje de salida Java. Se traduce cada estado y se genera un archivo .java con el resultado obtenido.

Capítulo 7. Discusión del Proyecto.

El proyecto es la aplicación de una serie de algoritmos que las personas utilizan normalmente en la construcción de programas. El análisis y diseño de la adaptación programática de estos algoritmos se encuentra siempre limitado por la arbitrariedad con la que se utiliza estos algoritmos en función de cada problema específico. Sin embargo, gracias a la lógica de Hoare y a la programación metódica se obtiene una metodología con la suficiente flexibilidad como para ser aplicada de manera automática en un programa. La limitación del proyecto y las dificultades encontradas para la aplicación de esta metodología no se encontraron en la aplicación de las propuestas. La dificultad se encontró, en muchos casos, en que la aplicación partía de una sugerencia de juicio experto aplicable solo a algunos casos en específico y en otros casos se necesitaba de otras metodologías. Estas metodologías implicaban procesos complejos de difícil adaptación programática como la deducción por inducción, la generación de programas recursivos, la equivalencia de asertos, entre otros. Para este proyecto se trabajaron versiones simplificadas de estos algoritmos que dentro de su alcance logran cumplir cada uno con su objetivo. Una desarrollo de algoritmo más avanzado para cada uno de estos algoritmos daría por aumentar el alcance del proyecto siendo esta una de las diversas propuestas de trabajos futuros.

Capítulo 8. Conclusiones.

Durante el desarrollo del proyecto se encontraron diversas dificultades relacionadas a la aplicación automática de procesos que para el ser humano con entrenamiento adecuado puede aplicar con facilidad pero que son difíciles de aplicar para un computador; en la mayoría de estos casos se optó por diseñar un algoritmo capaz de aplicar estos procesos, sin embargo, la limitación de estos algoritmos reducen el alcance del proyecto.

Durante el análisis y síntesis de las reglas a aplicar por el compilador se tuvieron que descartar propuestas de derivación debido a que su aplicación requerían de algoritmos especializados como la generación de programas recursivos, aplicaciones en función a una estructura abstracta de datos específica y algoritmos de transformación por inducción. En otros casos las reglas no podían ser generalizadas y eran aplicadas en función a sugerencias arbitrarias del autor. Sin embargo se incluyeron suficientes reglas como para satisfacer los objetivos del proyecto y el alcance del mismo, debido a que se incluyeron todas las reglas relacionadas a la mecánica de aplicación de propuestas de derivación. Es decir, se incluyeron todas las reglas que forman la base de la aplicación de esta metodología.

Para el análisis léxico, semántico y sintáctico se diseñó un lenguaje capaz de expresar los asertos que el compilador puede resolver. Se incluyeron variables, enteros, constantes, ecuaciones y cuantificadores. Estos últimos son los únicos capaces de representar un vector y su transformación se hace de manera automática. Era necesario incluir los vectores para enriquecer el lenguaje, pero debido a que las propuestas de derivación relacionadas a los vectores implicaban en su gran mayoría la inclusión de algoritmos np se opto por derivar de manera automática cada cuantificador sin utilizar las propuestas no implementadas. De esta manera se logró incluir la representación de vectores en el lenguaje cumpliendo con la expresividad necesaria para satisfacer los objetivos del lenguaje.

En la implementación del traductor de código intermedio se aplicaron y adaptaron los conceptos relacionados a un autómata adaptativo. El algoritmo

no solo es capaz de utilizar las propuestas de derivación de programas con éxito, sino que también soporta la inclusión de nuevas reglas ya que la formación de cada submaquina es independiente del funcionamiento del algoritmo. La expresividad del lenguaje, en algunos casos, excede la capacidad de traducción del compilador sin embargo, en estos casos el compilador no resuelve dar ningún resultado a dar uno erróneo. As así que el traductor es capaz de generar instrucciones en un lenguaje de alto nivel, y garantizar la correctitud formal de los programas que pueden ser generados por el compilador.

De esta manera se logró cumplir con el objetivo del proyecto estableciendo además la base para la derivación automática de programas y la construcción de un compilador de programación automática. Los programas obtenidos por este compilador son regidos por las reglas de programación metódica y al cumplir estas son formalmente correctos lográndose cumplir con el objetivo. Este proyecto es la base de muchos proyectos que se pueden generar en pro de la programación automática.

5.3 Trabajos Futuros

Debido a que este proyecto es la base de la programación automática y cuenta en su estructura con algoritmos de limitado alcance existen varios trabajos futuros que se pueden generar de este proyecto.

- Debido a que el resultado del traductor intermedio es una representación de instrucciones universal es posible traducir el lenguaje a cualquier lenguaje de alto nivel. Para la traducción solo sería necesaria la adaptación adecuada de las estructuras esenciales de todo lenguaje como son las variables, constantes, asignación de vectores, las condicionales y los bucles.
- El aumentar en el lenguaje la capacidad de incluir especificaciones de algebraicas de tipos abstractos de datos se podría incluir mas reglas relacionadas a estas estructuras.
- Se puede implementar un algoritmo de deducción por inducción e incluir más reglas de programación metódica.

- Si se implementa un mecanismo de aprendizaje se podría entrenar al compilador para resolver los problemas cuya solución tienen un origen arbitrario o son tomados a partir de juicio experto.
- Durante la presentación del conjunto de estados que forman parte de la solución de cada derivación se tiene como información la regla aplicada por cada estado. Es decir que si se deseara aplicar este proyecto como herramienta de aprendizaje de la metodología, es posible adaptar el compilador para que este explique paso a paso la derivación de cada par de asertos.

Referenciasbibliográficas

- HOARE, Charles

1969 An Axiomatic Basis For Computer Programming, New York, Communications of the ACM, Volumen 12, Número 19, pp 576-580.

- DIJKSTRA, Esdger

1976 Guarded Commands, Non-Determinacy And Formal Derivation Of Programs, New York, Lecture Notes in Computer Science, Volumen 46 ,pp111-124.

- MEYER**,** Bertrand

1992 Applying “Design By Contract”, USA, IEEE-Computer, USA, Volumen 25, Número 10, pp40-51.

- BALCAZAR, José Luis

1993 Programación Metódica, Madrid, McGRAW-HILL/INTERAMERICANA DE ESPAÑA S. A.

- SETHI, Ravi

2011 Programming Languages Concepts & Constructs, New Jersey, Pearson.

- GOMEZ, Juan Renzo

2012 *Intérprete para un Lenguaje de Programación Orientado a Objetos, con Mecanismos de Optimización y Modificación Dinámica de Código*, Tesis de licenciatura en Ciencias e Ingeniería con mención en Ingeniería Informática. Lima: Pontificia Universidad Católica del Perú, Facultad de Ciencias e Ingeniería.

- AHO, Alfred

1990 Compiladores Principios, Técnicas y Herramientas, Delaware, Addison Wesley Iberoamericana, S. A., Delaware

- ROSENBLUM, David

1995 *IEEE Transactions on Software Engineering*, ”A Practical Aproach to Programming With Assertions”,New Jersey, 1995, Volumen 21 -1 ,pp19-31

- WICHMANN, Bob

1995 *Software Engineering Journal*, ”Industrial perspective on static analysis”,Los Angeles, 1995, Volumen 10-2 ,pp69-75

- ROBERTS, ERIC

2008 «Programming Abstractions».*Stanford University* Fecha de consulta:26/03/2014. <http://cs.stanford.edu/people/eroberts/courses/cs106b///>.

- DRESDEN, TU

2012 «TheAlgorithmVerificationTool». *J-Aalgo* Fecha de consulta:14/09/2012. <http://j-algo.binaervarianz.de//>.

- FEINERER, Ingo

2005 Formal Aspects of Computing, “Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations”, Viena,2005, Volumen 21,pp 293-301 .

- Project Management Institute

2008 Guia de los Fundamentos para la Dirección de Proyectos (Guia del PMBOK) 4ta Edición. Pennsylvania.

- GALLES, David

2005 Modern CompilerDesign, San Francisco, Pearson.

- BECK, Kent

2006 Extreme Programming Explained, San Francisco, Pearson.

- NETO, João José

1994 Adaptiveautomataforcontext-dependentlanguages , São Pablo ,1994 Escola Politécnica da Universidade de São Paulo.

- CAYA, Rosalia Edith

2009 Estudio de la Mejora de la Calidad de un Sintetizador de Voz para el Castellano Utilizando el Método de Autómatas Adaptativos. Tesis correspondiente al grado de Ingenieria Informática. Lima: Pontificia Universidad Católica del Perú, Ingenieria Informática.

- KALDEWAIJ ,Anne

1990 Programming:The Derivation Of Algorithms, Inglaterra, Prentice Hall International.