TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

INTÉRPRETE PARA UN LENGUAJE DE PROGRAMACIÓN ORIENTADO A OBJETOS, CON MECANISMOS DE OPTIMIZACIÓN Y MODIFICACIÓN DINÁMICA DE CÓDIGO

Tesis para optar por el Título de Ingeniero Informático, que presentan los bachilleres:

Renzo Gonzalo Gómez Díaz
Juan Jesús Salamanca Guillén

ASESOR: MSc. Viktor Khlebnikov

Lima, febrero de 2012

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Resumen

Este trabajo trata sobre la implementación de un intérprete para un lenguaje propio, que incluye algunas características que no son abordadas en cursos básicos de Desarrollo de Compiladores. Estas características son: lenguaje de programación orientado a objetos, modificación dinámica de código y optimización de código intermedio.

El objetivo de este proyecto es presentar estas características, proponer una forma de implementación de las mismas y finalmente proceder a implementarlas. De tal manera que este trabajo contribuya al aprendizaje de construcción de intérpretes o compiladores, sirviendo como un caso de estudio para aquellas personas que tengan como objetivo profundizar en el tema, y por consiguiente un posible punto de partida para futuros trabajos. Por otro lado, es necesario validar los resultados obtenidos por el optimizador, así como la eficiencia de la forma implementación escogida, por lo que se incluye también una experimentación numérica que permite comprobar las hipótesis planteadas al inicio.

En la primera parte, se define el problema identificado, luego se describe un breve marco teórico con los principales conceptos involucrados en el desarrollo del proyecto, seguidamente se muestra el estado del arte con relación a compiladores e intérpretes y se describe la solución al problema planteado al inicio. En la segunda parte, principalmente, se describen los objetivos del proyecto, los aportes específicos, los resultados esperados y las hipótesis.

Como se mencionó anteriormente, uno de los objetivos que se persigue es que la tesis pueda servir como un caso de estudio para las personas interesadas y una posible base para trabajos futuros; por lo tanto, es necesario explicar la implementación con un nivel de detalle adecuado. En ese sentido, se describe las distintas partes de la implementación escogida: se comienza con la descripción de la gramática del lenguaje, después se describen las estructuras utilizadas, algunas operaciones primitivas, el código intermedio generado, las principales acciones semánticas, la interpretación, la administración de memoria, los algoritmos de optimización, el diseño del IDE y el ambiente de desarrollo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Dedicado a:

Mis padres.

Agradecimientos a:

Dios, porque es Él quien me da fuerzas para seguir adelante.
Mis padres, por el apoyo incondicional que me han brindado a lo largo de mi vida.
Mis profesores, en especial a Viktor y Martin, por las enseñanzas académicas y personales, a lo largo de mi formación profesional, que forjaron mi vocación a la investigación.
Mis amigos por su apoyo y ánimo, especialmente al Capítulo de Estudiantes de ACM, por ser mi segunda familia, con los que comparto un sueño.
Finalmente, al coautor de este trabajo, por su esfuerzo para lograrlo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Dedicado a:

Mis abuelos Virgilia y Jesús que están en el cielo. Mi madre.

Agradecimientos a:

Dios, porque Él nos da su amor y gracia, hace posible todo e ilumina nuestra vida.
Mi madre y mis hermanos, por ser mi inspiración para salir adelante en todo momento; y cuyo amor, cariño, apoyo y comprensión han sido una tierna luz en mis años de vida.
Toda mi familia, en especial mi padre, por estar conmigo en este camino y brindarme su apoyo, cariño y motivación.
Mis profesores, por guiarme hacia el camino de la investigación, especialmente a nuestros asesores Martin y Viktor, muchas gracias por su apoyo, paciencia, dedicación y esfuerzo.
Las personas que no menciono explícitamente pero que son muy especiales para mí, porque en ellas he encontrado amistad verdadera e incondicional, comprensión y cariño que me ha dado fuerzas para seguir adelante.
Finalmente, al coautor de esta tesis, por aquellas horas de esfuerzo y trabajo intenso en el proyecto, por su amistad incondicional y por su apoyo en estos años que han pasado.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Tabla de contenido

Introducción ... 1
1. Generalidades ... 3
1.1. Definición del problema ... 3
1.2. Marco Conceptual ... 5
1.3. Estado del arte ... 9
1.4 Descripción y sustentación de la solución ... 14
2. Aporte de la investigación ... 17
2.1. Objetivos de la investigación ... 17
2.2. Aportes específicos ... 18
2.3. Resultados esperados ... 19
2.4. Metodología de desarrollo ... 20
2.5. Ciclos de vida para el desarrollo del proyecto ... 21
2.6. Aporte de la investigación a la problemática planteada ... 23
2.7. Planteamiento de la hipótesis ... 24
3. Implementación del aporte ... 26
3.1. Diseño de la gramática para un lenguaje orientado a objetos ... 26
3.1.1. Diseño de la jerarquía de clases ... 30
3.2. Estructuras de datos ... 31
3.2.1. Descripción de la tabla de código ... 31
3.2.2. Descripción de la tabla de clases ... 34
3.2.2.1. Descripción de la tabla de atributos ... 35
3.2.2.2. Descripción de la tabla de métodos ... 36
3.3. Descripción de algunas operaciones primitivas del lenguaje Pas++ 38
3.3.1. Instrucción new ... 38
3.3.2. Instrucción delete ... 39
3.3.3. Instrucción insert ... 39
3.3.4. Instrucción remove ... 40
3.3.5. Instrucción modify ... 40
3.4. Descripción del código intermedio generado ... 41
3.4.1. Operaciones binarias con resultado ... 42
3.4.2. Operaciones binarias sin resultado ... 42
3.4.3. Operaciones unarias con resultado ... 43
3.4.4. Operaciones unarias sin resultado ... 43
3.4.5. Operaciones con múltiples operandos ... 43

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.4.6. Operaciones de lectura/escritura 43
3.4.7. Operaciones para manejo de objetos 44
3.4.8. Operaciones para modificación dinámica de código 44
3.5. Acciones semánticas principales 45
3.6. Interpretación de código intermedio y administración de memoria...51
3.6.1. Llamadas a métodos 52
3.6.2. Instanciación y eliminación de objetos 54
3.6.3. Operaciones con atributos 55
3.6.4. Modificación dinámica de código 56
3.7. Algoritmos de optimización 57
3.7.1. Ejecución en tiempo de compilación 59
3.7.2. Propagación de copias 60
3.7.3. Eliminación de las subexpresiones comunes 61
3.8. Diseño del entorno de desarrollo 64
3.9. Ambiente de desarrollo 69
3.9.1. Flex 69
3.9.2. Bison/Yacc: 70
3.9.3. Netbeans: 71
4. Experimentación numérica 72
4.1. Introducción a la experimentación numérica 72
4.2. Primera parte de la experimentación numérica 73
4.2.1. Definición del problema de investigación 73
4.2.2. Variables de respuesta 73
4.2.3. Hipótesis 73
4.2.4. Consideraciones de la ejecución 74
4.2.5. Determinación del estadístico: T-Student 74
4.2.6. Resultados de la experimentación 75
4.3. Segunda parte de la experimentación numérica 78
4.3.1. Definición del problema de investigación 78
4.3.2. Variables de respuesta 78
4.3.3. Hipótesis de la Experimentación numérica 78
4.3.4. Consideraciones de la ejecución 79
4.3.5. Determinación del estadístico: T-Student 80
4.3.6. Resultados de la experimentación 80
5. Observaciones, conclusiones y recomendaciones 87
5.1. Observaciones 87
5.2. Conclusiones 90

Tesis publicada con autorización del autor
No olvide citar esta tesis

5.3. Recomendaciones y trabajos futuros 92
6. Referencias 95

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

ET LUX IN TENEBRIS LUCET MCMXVII PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Índice de figuras

Figura 1.1 Ejemplo de gramática de contexto libre ____ 7

Figura 2.1 Prácticas de la metodología XP ____ 21

Figura 3.1 Gramática de Pas++ (herencia) ____ 30

Figura 3.2 Estructura de tcodigo y tab_code ____ 32

Figura 3.3 Ejemplo de un programa fuente en Pas++ ____ 33

Figura 3.4 Representación del programa en la tabla de código ____ 34

Figura 3.5 Estructura de tclass ____ 34

Figura 3.6 Estructura de tclass y ClassLoader ____ 34

Figura 3.7 Estructura de tatrib ____ 35

Figura 3.8 Estructura de tmetodo ____ 36

Figura 3.9 Definición de tsimbolo ____ 38

Figura 3.10 Función generaCodigo() ____ 45

Figura 3.11 Acciones semánticas para declaración de clases ____ 47

Figura 3.12 Ejemplo de complementoObAt y complementoMet ____ 47

Figura 3.13 Acciones semánticas para new ____ 49

Figura 3.14 Acciones semánticas para delete ____ 50

Figura 3.15 Acciones semánticas para “insert” ____ 50

Figura 3.16 Acciones semánticas para “modify” ____ 51

Figura 3.17 Acciones semánticas para “remove” ____ 51

Figura 3.18 Ejemplo de optimización por ejecución en tiempo de compilación ____ 59

Figura 3.19 Ejemplo de optimización por propagación de copias ____ 60

Figura 3.20 Algortimo de propagación de copias ____ 61

Figura 3.21 Ejemplo de optimización por subexpresiones comunes ____ 62

Figura 3.22 Algoritmo de subexpresiones comunes ____ 63

Figura 3.23 El entorno de desarrollo ____ 64

Figura 3.24 Editor de texto ____ 64

Figura 3.25 Funciones básicas ____ 65

Figura 3.26 Ejecución de un programa en el IDE ____ 65

Figura 3.27 Funciones del intérprete ____ 66

Figura 3.28 Modificación dinámica de código ____ 66

Figura 3.29 Tabla de código ____ 67

Figura 3.30 Tabla de símbolos ____ 68

Tesis publicada con autorización del autor
No olvide citar esta tesis

Figura 3.31 Tabla de clases_______________________________________ 68
Figura 3.32 Flujo en Flex ____ 69
Figura 3.33 Secciones de Flex_____________________________________ 70
Figura 3.34 Flujo en Bison ____ 70
Figura 3.35 Secciones de Bison ___________________________________ 71
Figura 4.1 Resultados de tiempos de la experimentación numérica (en microsegundos)____ 76
Figura 4.2 Tabla de código con optimización (izquierda) y sin optimización (derecha) 77
Figura 4.3 Diagramas de Gantt de los proyectos desarrollados (de izquierda a derecha: intérprete, IDE y optimizador) ___________________________ 83

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Introducción

En la presente tesis se desarrolla un intérprete con tres características particulares: lenguaje orientado a objetos, mecanismos de modificación dinámica de código y opciones de optimización; las cuales son muy útiles e interesantes, pero que normalmente no son abordadas en cursos básicos de Desarrollo de Compiladores. Por lo tanto, se realiza un estudio de dichas características, se propone una forma de implementación para las mismas y finalmente se procede a implementarlas. De esta manera se busca contribuir con el aprendizaje e investigación en el área de Ciencias de la Computación, específicamente, en el tema de Compiladores y Lenguajes de Programación; además de servir como modelo de implementación a seguir para proyectos futuros en el tema.

La tesis presentada constituye, en la opinión de los autores, un esfuerzo para contribuir en alguna medida con el aprendizaje, teórico y práctico, de cómo se logra construir un intérprete para un lenguaje propio. Sin embargo, por motivos prácticos no es suficiente la construcción del intérprete, se necesita un elemento integrador que facilite su uso y permita captar más rápidamente los flujos de datos del intérprete. Por lo tanto, en la tesis también se considera

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

la construcción de un entorno de desarrollo, mediante el cual podremos hacer uso del intérprete de una manera más fácil; además de poder observar gráficamente el árbol de clases y las tablas necesarias para la interpretación, optimización y modificación dinámica de código.

Finalmente, es necesario analizar qué tan eficiente fue la implementación elegida y si las características del intérprete cumplen su función. En ese sentido, se utilizará un ciclo de vida diferente para la implementación de cada característica del intérprete, y con los resultados de tiempos obtenidos se intentará definir qué tipo de ciclo de vida de software es el más adecuado para un proyecto de construcción de compiladores o intérpretes; por otro lado, también se harán mediciones de tiempos a programas optimizables para comprobar si los mecanismos de optimización cumplen su función.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

1. Generalidades

En este capítulo se muestra el problema identificado, el marco conceptual acerca del tema de compiladores e intérpretes, el estado del arte y la descripción de la solución.

1.1. Definición del problema

Las Ciencias de la Computación han cobrado gran importancia especialmente en los países más desarrollados, ver en "Computing and higher education in Peru" [ABP2008]. Es por ello, que en el primer mundo tanto el Estado como las empresas privadas invierten en formar profesionales de alto nivel académico en esta área, con la finalidad de contribuir a la generación de conocimiento y para el beneficio de sus intereses.

En nuestro país, el área de Ciencias de la Computación no se encuentra muy difundida, prueba de esto es que hace poco tiempo no existía ninguna universidad que ofreciera esta especialidad como parte de su programa de pregrado. Esto debido a que no se brinda el apoyo suficiente, tal vez porque no se ha tomado conciencia de lo importante que puede ser para el desarrollo tecnológico del país y sólo se toma en cuenta la situación actual donde su campo laboral es casi inexistente. Sin embargo, en la actualidad la carrera ha

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

sido creada en algunas universidades peruanas, donde las actividades de investigación en esta área recién están iniciándose. Entonces, ante una realidad adversa, es importante resaltar que se han hecho algunos trabajos e investigaciones, pero es necesario que se incrementen en los años venideros para que esta área pueda realmente progresar en Perú.

Uno de los temas importantes en Ciencias de la Computación es el de Lenguajes de Programación y Compiladores, por lo tanto, su aprendizaje debería ser indispensable en una carrera orientada a esta área en pregrado, incluyéndose la implementación de un intérprete o compilador de nivel básico. Por lo tanto, se necesitan trabajos que puedan servir como referencia o casos de estudio para apoyar la construcción de intérpretes o compiladores de niveles más complejos.

En el Perú no se tratan tópicos de Compiladores como: optimización de código intermedio, soporte para programación orientada a objetos o modificación dinámica de código. Esto debido a la problemática descrita anteriormente y también por ser temas complicados, que requieren del tiempo adecuado para su aprendizaje y de un nivel de abstracción muy alto.

Por lo tanto, es necesario realizar proyectos que permitan difundir el tema de Lenguajes de Programación y Compiladores, específicamente la construcción de un compilador o intérprete con características no tratadas anteriormente. Asimismo, herramientas de software que logren facilitar el uso de las funcionalidades del intérprete y que puedan llamar más fácil la atención de los estudiantes. Además, concreticen conceptos involucrados con compiladores, desde los más simples hasta los más complejos (como por ejemplo la modificación dinámica de código y la optimización de código).

También es importante considerar que para lograr que las personas interesadas puedan tomar este trabajo como un caso de estudio y posteriormente puedan construir un compilador o intérprete que extienda las funcionalidades del desarrollado en el presente proyecto es necesario explicar detalladamente las características más complejas en su implementación y de una manera detallada.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

1.2. Marco Conceptual

Los conceptos más importantes acerca de la construcción de compiladores e intérpretes y que explicaremos seguidamente serán: traductor, intérprete, compilador, gramáticas de contexto libre, metacompilador y modificación dinámica de código; además de las etapas de análisis (léxico, sintáctico y semántico) y de síntesis (generación de código intermedio, optimización de código y generación de código máquina).

Un traductor se define como un programa que **traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino**. Un compilador es un traductor que toma como entrada la especificación de un programa ejecutable y produce como salida la especificación de otro programa ejecutable equivalente, ver en el libro “Engineering a Compiler” de Keith Cooper y Linda Torczon [KCLT2004]. Generalmente, tiene como entrada un archivo en lenguaje formal y como salida un archivo ejecutable o en código objeto, es decir, realiza una traducción de un código de alto nivel a código máquina. Por otro lado, un intérprete es un **traductor que toma como entrada la especificación de un programa ejecutable y produce como salida el resultado de ejecutar la especificación**, ver en [KCLT2004]**.** Entonces, no se tiene un resultado físico (código máquina) sino lógico (la ejecución).

A pesar de las diferencias citadas anteriormente, un intérprete tiene la misma estructura que un compilador. Según se especifica en el libro “Compiladores: principios, técnicas y herramientas” de Aho, Sethi y Ullman [ASU2007], una descomposición típica de los procesos que siguen ambos es en dos etapas: la de análisis y síntesis. La etapa de análisis se divide en análisis léxico, sintáctico y semántico:

- **Análisis léxico:** divide el programa fuente en los componentes básicos del lenguaje a compilar. Cada componente básico es una secuencia de caracteres del programa fuente y pertenece a una categoría gramatical: números, identificadores de usuario (variables, constantes, tipos, nombres de métodos), palabras reservadas, signos de puntuación, etc.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **Análisis sintáctico:** comprueba que la estructura de los componentes básicos sea correcta según las reglas gramaticales del lenguaje que se compila.

- **Análisis semántico:** comprueba que el programa fuente respete las directrices del lenguaje que se compila (todo lo relacionado con significado del programa): chequeo de tipos, rangos de valores, existencia de variables, etc.

La etapa de síntesis se divide en generación de código intermedio, fase de optimización y generación de código máquina.

- **Generación de código intermedio:** genera un código independiente de la máquina muy parecido al ensamblador. Se puede considerar un programa para una máquina abstracta.

- **Fase de optimización**. la optimización puede realizarse sobre el código intermedio (de forma independiente de las características concretas del microprocesador), sobre el código máquina o sobre ambos. El objetivo de esta fase es mejorar el código intermedio, tratando de buscar un código más eficiente.

- **Generación de código máquina:** crea un bloque de código máquina ejecutable, así como los bloques necesarios destinados a contener los datos.

Generalmente, se suele agrupar las tres fases del análisis y la fase de generación de código intermedio en una gran fase llamada "**front end**", 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.

Para definir la sintaxis del lenguaje Pas++ utilizaremos Gramáticas de Contexto Libre (CFG), **la cual es una notación que permite definir la sintaxis de un lenguaje en particular, describiendo la estructura**

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

jerárquica de las construcciones de un lenguaje de programación. Según [ASU2007] se componen de cuatro elementos:

- Un conjunto de tokens o símbolos terminales.
- Un conjunto de símbolos no terminales.
- Un conjunto de producciones, donde cada producción consiste de un símbolo no terminal (lado izquierdo de la producción), una flecha, y una secuencia de tokens y/o símbolos no terminales (el lado derecho de la producción).
- El símbolo de inicio, que es uno no terminal.

Un metacompilador es un compilador de compiladores. **Se trata de un programa que acepta como entrada la descripción de un lenguaje (gramática) y produce el compilador o intérprete de dicho lenguaje**, ver en [ASU2007]. No existen metacompiladores completos, pero sí parciales que permiten añadir código para completar el resto del compilador. Ejemplos de metacompiladores son: Lex, YACC, FLex, Bison, JavaCC, JLex, Cup, etc. En este proyecto utilizaremos herramientas de metacompilación como Yacc y Bison, aprovechando el uso de las Gramáticas de Contexto Libre (CFG).

A continuación presentamos algunos ejemplos de la gramática que podríamos utilizar para el lenguaje Pas++

```
instrList : bloque instrList | ;

bloque : TBEGIN instrList TEND |  instr;

instr :  declar ‘;’ ;

declar : tipo idList ;

idList :  TID | idList ‘,’ TID
```

Figura 1.1 Ejemplo de gramática de contexto libre

Por otro lado, también utilizaremos el concepto de **modificación dinámica de código (self-modifying code) que se define como código que cambia su propia estructura durante su ejecución.** Este cambio en tiempo de ejecución es posible debido a que las computadoras actualmente se basan en la arquitectura Von Neumann, donde la CPU ejecuta instrucciones desde la

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

memoria principal del computador. Por tal motivo, a pesar que un programa se divide en segmentos de código, datos, pila, etc, esta división es puramente lógica. De esta forma, una vez que código y datos son cargados en memoria, ya no existe una división física entre ellos y de esta manera un programa puede contener instrucciones que accedan a estas posiciones de memoria y cambiar su contenido.

Por último, en el desarrollo de la investigación y el intérprete se incluirá la optimización de código**. Con respecto a este punto, un optimizador en general trata de analizar el código intermedio obtenido del intérprete/compilador, para realizar mejoras que influyan en el tiempo de ejecución**. Algunas fuentes posibles de optimización podrían ser la eliminación de código muerto (se define como código que nunca se ejecuta por medio del análisis del flujo del programa) y el reemplazo de subexpresiones comunes para evitar el cálculo repetido de las mismas, ver en [KCLT2004].

También es importante definir ciertos conceptos que serán mencionados de manera frecuente en las optimizaciones propuestas, estos son: bloque básico de código y grafo de control de flujo. Por un lado, según [ASU2007], un bloque básico es todo conjunto de instrucciones contiguas, en donde el flujo de control del programa ingresa en la primera instrucción y sale en la última instrucción sin posibilidad de salto, salvo en la última instrucción. Por otro lado, el grafo de control de flujo es un grafo que modela la ejecución de un programa, que también puede simplificar el flujo del programa si encuentra regiones imposibles de alcanzar o expresiones que no son importantes.

La optimización se puede clasificar por la manera en la que se analiza el código objeto para realizar la optimización, de esta forma se clasifican los métodos de optimización de la siguiente manera:

- **Métodos locales:** este tipo de métodos centran su atención en bloques básicos de código para realizar las mejoras. Al restringir su análisis a bloques básicos se pueden probar hechos que se cumplen solo en estos bloques y no a una escala mayor.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **Métodos superlocales:** este tipo de métodos se aplican a bloques extendidos de código que se definen como un conjunto de bloques b1, b2…., bn, donde b1 puede tener múltiples predecesores en el flujo del programa pero b2,…, bn solo posee un único predecesor en el bloque extendido. La definición de estos bloques permite que hechos probados para un predecesor sean aplicados a el (los) bloque(s) que le sigue(n), y sean posibles mayores optimizaciones.

- **Métodos regionales:** estos métodos se enfocan en alcances mayores a un bloque extendido pero menor al ámbito de un procedimiento. En general, lo que diferencia a los métodos regionales de los superlocales es que los primeros toman en cuenta los puntos de fusión dentro del CFG (grafo de control de flujo) para encontrar fuentes de optimización.

- **Métodos globales:** también llamados métodos intraprocedurales, éstos evalúan un procedimiento completo para encontrar hechos que permitan la optimización de código.

- **Métodos de todo el programa:** aquí el ámbito para la optimización es el programa entero. También es llamado método interprocedural. Se realiza esto con la idea que se pueden encontrar mayores y mejores fuentes para la optimización, aunque esto no siempre sucede.

1.3. Estado del arte

En esta sección se presenta un análisis de lo hecho anteriormente alrededor del tema de construcción de compiladores e intérpretes. Con relación a los mismos, podemos decir que existen una gran variedad para diversos lenguajes de alto nivel. A nosotros nos interesan especialmente los relacionados a Pascal, C y sus sucesores, de los cuales deriva Pas++.

El primer compilador para Pascal fue creado por Niklaus Wirth y el segundo por Ammann, ambos eran compiladores de una sola pasada, es decir, solo analizaban el código fuente una vez para producir el lenguaje máquina esperado, lo cual traía como resultado un programa ineficiente y de mucho

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

consumo de recursos, ya que no se realizaban las optimizaciones que se hacen en un compilador de múltiples pasadas; su principal ventaja era ser fáciles de implementar, ver en [ASU2007].

Luego, fueron muy populares el Turbo Pascal y Delphi de Borland [BP], el primero tuvo una distribución masiva y en sus versiones posteriores introdujo programación orientada a objetos, el segundo se creó para reemplazar al primero y fue diseñado a partir del lenguaje Object Pascal [OP] de Apple, el cual después se llamaría Delphi. Actualmente, existen muchos compiladores e intérpretes para Pascal de descarga gratuita, el más usado es Free Pascal [FP], el cual es potente y estable, además de soportar muchas plataformas y sistemas operativos.

Con respecto a los compiladores para C, entre los primeros podemos mencionar al compilador creado por Ritchie para la PDP-11 y la familia de compiladores PCC escrito por Johnson. Ambos compiladores son esencialmente de dos pasos, pero el primero tiene una optimización adicional del código ensamblador intermedio, que eliminaba redundancia y sentencias inaccesibles. Es importante resaltar que el PCC usa un analizador sintáctico generado por la herramienta Yacc y además los tres cuartos de su código son independientes de la arquitectura de la máquina en la cual se va a utilizar, ver en [ASU2007].

Con la llegada de C++, los compiladores tuvieron que soportar la programación orientada a objetos, donde uno de los problemas más difíciles es lograr conectar las estructuras en tiempo de ejecución que el programa atraviesa para encontrar métodos, atributos y otros objetos. Actualmente, los compiladores más utilizados para C/C++ son GNU gcc/g++ para Linux y Windows, Borland C++ y MS Visual C++.

Un proyecto que se llevó a cabo y que resume muy bien la creación de un compilador que optimiza código es el de Modula-2, ver en [ASU2007]. Este compilador fue desarrollado con la intención de producir un código eficiente, usando optimizaciones que no cuesten mucho esfuerzo, es así que su autor logró producir un código excelente a través de un compilador escrito en pocos meses y por una sola persona. Los 5 pasos que siguió en el front-end son: parseo, resolver referencias para los identificadores, optimizaciones de código

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

intermedio, calcular conteo de referencias y asignación de registros, y generar el p-code (compatible con el de Pascal). El analizador sintáctico que usó este compilador fue creado usando Yacc y produce árboles de sintaxis en dos pasos.

A nivel de la PUCP, se han realizado algunas tesis sobre el tema de compiladores e intérpretes, entre ellos "**Intérprete de páginas web dinámicas para el servidor Apache"**, que explica cómo funciona la tecnología Server-Side Scripting en la generación de- páginas web dinámicas. Se desarrolló dos versiones de un intérprete basado en las reglas del lenguaje Pascal, la primera de estas versiones corre como un módulo DSO (Dynamic Shared Object) integrada al servidor Apache, mientras que la segunda es un intérprete independiente que genera las páginas dinámicamente interactuando con el servidor Apache vía CGI, ver en [MKM2004].

Otro proyecto que podemos mencionar es la implementación de un "**Intérprete para el aprendizaje de lenguajes de programación estructurada"**, en donde se trata de crear una herramienta útil para que los alumnos de los primeros ciclos de pregrado puedan aprender los conceptos básicos de programación, a partir de un nuevo lenguaje estructurado cuya sintaxis utiliza palabras del castellano y, por lo tanto, permitía a los alumnos expresar sus ideas en código más fácilmente, ver en [LHM2007].

Además, existen algunos trabajos de intérpretes del lenguaje SQL, como la implementación de un "**Intérprete SQL en managed code para dispositivos móviles"**, ver en [SMS2007]. Este trabajo está basado en el diseño de un intérprete SQL publicado por la Universidad Nacional Mayor de San Marcos en [HHL2002]. Este último se enfoca en lograr un sistema que le ofrezca al usuario la posibilidad de escribir sentencias SQL que puedan ser ejecutadas si son correctas; en la construcción del compilador utiliza Yacc para reducir la complejidad de la sintaxis de los selects.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

Es importante resaltar también el aporte del Dr. Maynard Kong mediante el desarrollo de herramientas que permiten facilitar las fases de análisis sintáctico y léxico para distintos lenguajes de programación[1].

Como se ha podido observar, la construcción de compiladores e intérpretes se realiza con la ayuda de herramientas que nos permiten crear analizadores léxicos y sintácticos a través de la declaración de la sintaxis del lenguaje de alto nivel, para lo cual se utiliza las gramáticas de contexto libre. Pero no existen herramientas conocidas que nos permitan crear un optimizador de código.

Con respecto a la **modificación dinámica de código**, generalmente es implementada usando el lenguaje ensamblador ya que se tienen mucho menos restricciones para acceder a memoria y es más factible realizar cambios al código en tiempo de ejecución. También es importante mencionar que esta característica ha sido implementada en algunos lenguajes de alto nivel (como: Perl, Python, PHP, etc.), donde se permiten a los programas crear nuevo código en tiempo de ejecución y ejecutarlo usando una función "eval". A continuación listaremos otras aplicaciones y trabajos realizados en este tema:

En primer lugar, es usado para proteger la propiedad intelectual en el desarrollo de software haciendo más complicado el uso de ingeniería reversa para obtener el código fuente de los programas, ver en el artículo "A model for self-modifying code" de Bertrant Anckaert, Matias Madou y Koen De Bosschere [AMDB2006]; asimismo, en "A taxonomy of self-modifying code for obfuscation" de Mavrogiannopoulos, Kisserli y Preneel [MKP2011] se propone una taxonomía para clasificar las técnicas que usan self-modifying code para este fin. Posteriormente, se ha usado para el ocultamiento de código malicioso (virus, troyanos, etc.).

En segundo lugar, es usado para la optimización dinámica de código. Esta utilidad tuvo su inicio en 1992 con el desarrollo del sistema operativo Synthesis [HM1992]. El cual, mediante el uso de objetos llamados "quajects", atiende las llamadas al sistema. Lo importante de estos objetos es que se

[1] Yack (Yet Another Compiler Kong) programa similar a Bison, fue distribuido en el curso de Teoría de Compiladores (2010-0)

Tesis publicada con autorización del autor
No olvide citar esta tesis

crean en tiempo de ejecución y generan su código de una manera sistemática para realizar una tarea específica, lo que permite al núcleo aplicar un conjunto de optimizaciones (como desenvolvimiento de constantes o eliminación de subexpresiones comunes) que reducen en gran medida el tiempo de ejecución. Asimismo, otra aplicación importante se da en los compiladores JIT (just in time) que generan código máquina optimizado en tiempo de ejecución a partir de un código intermedio (por ejemplo "bytecode") para una máquina abstracta.

En tercer lugar, el self-modifying code se aplica en la programación genética. Por ejemplo, en "Developments in Cartesian Genetic Programming: self-modifying CGP" de Harding, Miller y Banzhaf [HMB2010] se propone un método mediante el cual un programa (llamando genotipo en programación genética) puede evolucionar a través del tiempo, adquiriendo nuevas entradas o produciendo nuevas salidas. Además, se puede citar el trabajo de Peter Nordin y Wolfgang Banzhaf [PNWB1995] donde se utiliza la modificación dinámica de código para transformar código de programación genética en su equivalente para programación imperativa (en este caso lenguaje C).

En cuarto lugar, en el área de inteligencia artificial, especialmente en aprendizaje automático (machine learning), se resalta el trabajo realizado por Jürgen Schmidhuber [JS1987] quien aplico la modificación dinámica de código en el meta-aprendizaje. Tradicionalmente, en el aprendizaje automático se provee, de antemano, a la máquina de un conjunto de algoritmos para el aprendizaje. El trabajo antes mencionado, propone algoritmos para que el mismo sistema sea capaz de cambiar sus propios algoritmos para el aprendizaje.

Por último, se han realizado trabajos relacionados a la optimización en compiladores que permiten modificación dinámica de código: anteriormente, en un programa que contenía "self-modifying code" carecía de utilidad la generación de un grafo de control de flujo para realizar optimizaciones, ya que el cambio en el código en tiempo de ejecución hace que la estructura del grafo (aristas, bloques) también se transforme, lo que hace inútil un análisis a priori (compilación) para realizar optimizaciones. Sin embargo, en [AMDB2006] se propone un método para la creación de un grafo de control de flujo para un programa que contiene self-modifying code. Esto se hace posible

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

mediante el uso de información adicional en el grafo que permite describir todos los estados por los cuales puede pasar el programa. Asimismo, se provee de ejemplos para realizar optimizaciones clásicas (propagación de constantes, eliminación de código muerto y desenvolvimiento de bucles).

1.4 Descripción y sustentación de la solución

Descrito el problema en la sección 1.1, se plantea la solución: básicamente, lo que se propone es realizar primero un estudio de cómo se podrían implementar las tres características para el intérprete mencionadas anteriormente haciendo uso de bibliografía relacionada y artículos científicos en el área de Compiladores. Luego, se procede a implementar un intérprete que tenga dichas características siguiendo los pasos descritos en el marco teórico, para ello necesitaremos las siguientes herramientas:

- **Flex**: es un generador de analizadores léxicos. Toma como entrada una especificación del analizador léxico y devuelve como salida el código fuente implementando el analizador léxico en C. Es la versión libre de Lex.

- **Yacc/Bison**: es un generador de analizadores sintácticos. Dada una especificación sintáctica del lenguaje (gramática de contexto libre) hecha en la notación BNF, Yacc/Bison genera el código necesario para realizar el análisis sintáctico del lenguaje. El código generado por este programa es en lenguaje C. Bison es la versión GNU de Yacc.

- **Netbeans:** es un IDE para el desarrollo de proyectos en C, C++, Java, etc. Este entorno tiene licencia GNU.

El primer paso es **diseñar la gramática** de nuestro lenguaje, el Pas++, el cual tiene una sintaxis similar a la de Pascal, pero con algunas diferencias en la declaración de variables y en los tipos de datos que tendrá (más parecido a cómo se realiza en C). Se escogió Pascal, por ser un lenguaje de alto nivel muy fácil de aprender y con una sintaxis simple (en la opinión de los autores este debería ser el lenguaje con el que los estudiantes de pregrado se inicien en el mundo de la programación). La gramática obtenida en esta primera parte nos servirá de input para construir el analizador sintáctico.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Seguidamente se deberá implementar el **analizador léxico**, cuya función es la de clasificar a determinada palabra o lexema como un determinado token, y éstos sirvan de input para el analizador sintáctico. Aquí tenemos dos opciones: utilizar una herramienta (como **Flex**) que construya el "scanner" sólo definiendo reglas (expresiones regulares), o hacer un analizador léxico propio. En nuestro proyecto optaremos por la primera opción ya que se tendrá mayor potencia y rapidez para realizar el análisis léxico.

Luego, procederemos a obtener el **analizador sintáctico**, teniendo como entrada la gramática de nuestro lenguaje con soporte de objetos. Para ello utilizaremos la herramienta **Yacc/Bison**, la cual nos devolverá el código en lenguaje C del analizador sintáctico.

En este punto es necesario realizar la integración entre el analizador léxico y el sintáctico. Además de corregir errores típicos de conflictos por el uso de las herramientas anteriores e incluir **acciones semánticas** para lograr la interpretación considerando las **características de programación orientada a objetos y modificación dinámica de código**. También se implementará un **optimizador de código** que deberá incluirse en el intérprete, así podremos comparar la performance entre un programa interpretado con optimización de código y otro sin este paso adicional.

Nos será de ayuda, especialmente para fines prácticos y de usabilidad, **la construcción del entorno de desarrollo**; ya que en él incluiremos al intérprete y podremos utilizarlo para visualizar de manera gráfica los resultados y las estructuras de datos.
Además nos servirá para probar la característica de modificación dinámica de código, mediante opciones que permitan usar las funciones "insert", "remove" y "modify". También se podrá probar la ejecución de un programa usando o no el optimizador.

Un punto importante en nuestro trabajo es **la implementación del optimizador de código intermedio**, ya que no se cuenta con herramientas que nos permitan construirlo solo definiendo reglas, así como el intérprete. Por lo tanto, la investigación sobre el tema y el aporte de los autores es crucial para la realización del mismo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Finalmente, es importante resaltar las características que incluye el intérprete implementado en esta tesis y que son raramente abordadas en el contexto peruano

- El intérprete **tendrá la característica de optimización de código**. Es complicado implementar esta característica por ser un paso adicional a la realización del intérprete.

- El lenguaje Pas++ tiene **soporte para programación orientada a objetos**. No obstante, no se incluirán todas las características de la misma, sino las principales como: herencia simple, constructores, instanciación de objetos, ámbito de atributos y métodos.

- La característica de **modificación dinámica de código**, que se logrará implementando tres instrucciones: “insert”, “modify” y “remove”. De esta manera se podrá cambiar la estructura de un programa en ejecución. Pudiéndose observar los cambios que sufre la tabla de código de una manera gráfica a través del entorno de desarrollo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

2. Aporte de la investigación

En el siguiente capítulo se explican los objetivos de la investigación, los aportes específicos de la tesis y los resultados esperados al finalizar el proyecto. Asimismo, se define la metodología de desarrollo que se utilizará a lo largo del proyecto y se explica cuáles son los aportes de la investigación.

2.1. Objetivos de la investigación

En base al aporte que se desea brindar con el proyecto de tesis, se han definido los siguientes objetivos.

- **Objetivo general:**

  Presentar, estudiar e implementar tres características para un intérprete: lenguaje orientado a objetos (Pas++), mecanismos de modificación dinámica de código y opciones de optimización; además de desarrollar un entorno de desarrollo (IDE) para dicho lenguaje. De tal manera que este trabajo pueda ser usado como un caso de estudio y punto de partida para aquellas personas interesadas en hacer futuros trabajos en el tema.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **Objetivos específicos:**

  - Diseñar la gramática del lenguaje Pas++.
  - Implementar un intérprete para el lenguaje Pas++.
  - Implementar el mecanismo de modificación dinámica de código en el intérprete.
  - Implementar mecanismos de optimización de código para el lenguaje Pas++.
  - Implementar un entorno de desarrollo para el lenguaje Pas++.
  - Realizar una comparación entre los ciclos de vida utilizados para el desarrollo de los componentes principales del proyecto. Además de validar los resultados de las optimizaciones.

2.2. Aportes específicos

Los aportes específicos de la tesis son:

- Se implementará el intérprete de tal manera que éste **soporte la declaración de clases y el manejo de objetos**. Si bien no se implementarán todas las características de este tipo de programación, se implementarán las principales:

  - Se podrá definir herencia simple de clases. En la clase derivada se podrán sobrescribir métodos.
  - Se definirán los alcances privado, público y protegido para métodos y atributos.
  - Se podrán definir constructores y destructores (no habrán por defecto), e instanciar objetos.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

ET LUX IN TENEBRIS LUCET MCMXVII PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- Se podrán **mostrar las tablas de clases (incluyendo métodos, atributos y símbolos por método) y de código de una manera detallada**, para lo cual se tendrá en el entorno de desarrollo dicha opción. Con esto buscamos mostrar concretamente las estructuras generadas por el proceso de compilación que se conocen pero de una manera abstracta.

- Se tendrá **el mecanismo de modificación dinámica de código**. Con ello podremos utilizar las operaciones primitivas “insert”, “modify” y “remove”, las que nos permitirán insertar, modificar o remover código en una parte determinada del programa.

- El intérprete tendrá **mecanismos de optimización**, así podremos mejorar el código intermedio de un programa que tiene la posibilidad de optimizarse.

2.3. Resultados esperados

En base a los objetivos planteados, se han definido los siguientes resultados esperados.

- **La gramática del lenguaje Pas++:** este resultado hace referencia a obtener la especificación de la gramática que utilizará el analizador sintáctico del lenguaje Pas++ usando gramáticas de contexto libre.

- **El intérprete para el lenguaje Pas++:** es el elemento que engloba la realización de la fase de compilación y ejecución, por lo que se encargará de traducir un archivo de código fuente en lenguaje Pas++ para ser ejecutado en la computadora. Contará con las opciones de ejecutar, “generar código intermedio”, “ejecutar y modificar dinámicamente”, y “optimizar y ejecutar”.

  - **El mecanismo de modificación dinámica de código:** nos permitirá insertar, modificar o remover determinadas instrucciones

Tesis publicada con autorización del autor
No olvide citar esta tesis

en el programa en tiempo de ejecución. Esto se podrá hacer por medio del IDE u operadores primitivos en el mismo programa.

- **Un optimizador de código para el lenguaje Pas++:** este resultado del proyecto hace referencia a la optimización de código intermedio que realizará el intérprete.

- **Un entorno de desarrollo para el lenguaje Pas++:** este resultado esperado permitirá facilitar la escritura de programas para el lenguaje Pas++, asimismo proporciona un rápido acceso a las funciones básicas del intérprete (ejecutar, optimizar, modificación dinámica de código, etc.). Además brinda la opción de visualizar las estructuras generadas (tabla de clases y código) en el proceso de compilación de manera gráfica.

- Al final de proyecto podremos conocer aproximadamente cuál(es) de los ciclos de vida del software se adapta mejor a un proyecto orientado a las Ciencias de la Computación.

2.4. Metodología de desarrollo

Para el desarrollo del proyecto se ha tomado algunas prácticas ideales de la metodología XP (eXtreme Programming). Según [XP2010]: “Extreme Programming es una disciplina de desarrollo de software basada en valores de simplicidad, comunicación, retroalimentación y coraje. Funciona al unir todo el equipo ante prácticas simples, con suficiente retroalimentación para permitirle ver dónde se encuentra y, así, adecuar las prácticas a su situación particular”.

Algunas de las prácticas ideales que tiene XP y se utilizarán en el proyecto son:

- **Desarrollo guiado por pruebas:** esto significa que se escribe un test automático y. a continuación, se debe escribir el código suficiente para pasar dicho test. Luego se ''refactoriza” el código, para mejorar la legibilidad y eliminar duplicaciones. Esta característica de XP es muy

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

importante para el proyecto, ya que permite que cada elemento del compilador (analizador sintáctico, optimizador de código, etc.) sea probado y con esto tener componentes cuyo funcionamiento se encuentran totalmente verificados.

- **Integración continua:** esta práctica promueve el hecho que todos los integrantes del equipo de desarrollo sean capaces de corregir y extender cualquier parte del proyecto. En nuestro caso, permite conocer todo el flujo de trabajo del intérprete, lo cual es de vital importancia para la detección de errores y la integración de los componentes del mismo.

Figura 2.1 Prácticas de la metodología XP

2.5. Ciclos de vida para el desarrollo del proyecto

Para el desarrollo del proyecto, se ha tomado en cuenta dividir el desarrollo en tres grandes partes:

- Desarrollo del intérprete.
- Desarrollo del optimizador de código.
- Desarrollo del entorno de desarrollo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

En general, en el desarrollo de este tipo de aplicación (orientada a las Ciencias de la Computación) existe poca documentación acerca de qué tipo de ciclo de vida de desarrollo es más recomendado. Por tal motivo, se ha decidido, tomar un ciclo de vida distinto para cada uno de los componentes antes mencionados.

Se utilizarán los siguientes ciclos de vida, ver en "Implementación y Debugging" de Cantone [CD2006]:

- **En cascada:** admite iteraciones, pero se basa principalmente en que después de cada etapa se realizan una o varias revisiones para comprobar si se puede pasar a la etapa siguiente. Por lo tanto, es rígido y con restricciones.

- **Incremental:** se basa en construir incrementado las funcionalidades del programa. Se construye por módulos que cumplen las diversas funcionalidades del software. Se puede aumentar gradualmente las capacidades del mismo

- **Evolutivo:** aquí los requerimientos pueden cambiar en cualquier momento. Afronta un problema mediante una iteración de ciclos requerimiento-desarrollo-evaluación.

Para el desarrollo del **intérprete** se seguirá un ciclo de vida **en cascada**, porque la forma en que se compone un intérprete (analizador sintáctico, analizador semántico, etc.) se asemeja a un sistema en serie, lo cual aparentemente es idóneo para este ciclo de vida. En éste no se pasa a la fase siguiente sin haber terminado y revisado las anteriores, como el flujo de trabajo entre los componentes de un intérprete. Los resultados prácticamente se ven en las etapas finales y se necesitan todos los requerimientos desde el inicio.

Para el desarrollo del **entorno de de desarrollo**, se ha definido usar un ciclo de vida **incremental**, ya que las funcionalidades se irán implementando poco apoco, pero desarrollándose modularmente y respetando en lo posible los

Tesis publicada con autorización del autor
No olvide citar esta tesis

requerimientos iniciales. Esto se puede realizar debido a que al principio no es necesario que todas las funcionalidades estén implementadas.

Por último, para el desarrollo del **optimizador de código** se ha elegido utilizar un ciclo de vida **evolutivo**, debido a que éste se podrá realizar de distintas maneras, por lo tanto, tendremos la facilidad de agregar mejoras al mecanismo a medida que se vaya desarrollando, cambiando los requerimientos en cualquier momento.

2.6. Aporte de la investigación a la problemática planteada

Como se mencionó en el punto 1.1 (Definición del problema), el área de Ciencias de la Computación y, específicamente, el tema de Compiladores no se encuentra muy difundido en Perú, por lo que no se tocan algunos tópicos interesantes pero que podrían ser considerados complicados (como por ejemplo los que se tocan en el presente proyecto); y no existen los suficientes trabajos que permitirían ayudar a un estudiante de pregrado a reforzar o profundizar en estos temas. Por tal motivo, el presente trabajo tiene como finalidad ahondar en el tema de construcción de compiladores e intérpretes, con el estudio e implementación de temas que no han sido abordados por otro proyecto de fin de carrera anterior en Perú, como son: soporte para programación orientada a objetos, mecanismos de optimización y modificación dinámica de código.

En este sentido, la presente tesis servirá a personas interesadas en el área como una base para poder realizar mayores aportes e investigaciones en el tema. Asimismo, la creación del IDE tiene como finalidad la de brindar una interfaz que facilite la usabilidad de las funciones del intérprete. Además de mostrar las estructuras que resultan de la compilación, esto podría servir de apoyo a los estudiantes en la compresión de ciertos conceptos.

En el desarrollo de los distintos componentes del compilador (intérprete, entorno de desarrollo y optimizador de código) se está proponiendo diferentes ciclos de vida para realizar su implementación. Esto se realiza con la intención de poder obtener datos acerca de la eficiencia de los ciclos de vida

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

antes mencionados aplicados al desarrollo de proyectos en el área de las Ciencias de la Computación y analizar cuál se adapta mejor.

2.7. Planteamiento de la hipótesis

En el desarrollo del proyecto se ha especificado la optimización de código intermedio generado antes que se produzca la interpretación. Por lo tanto, es necesario probar que la implementación utilizada para el mecanismo de optimización es correcta y éste cumple su propósito. Para esto, se tomará como variable experimental la media del tiempo de ejecución de un mismo programa utilizando la optimización de código y otra en la cual no se utiliza. Esto nos permite analizar si en realidad se están obteniendo los resultados que se esperaban del optimizador.

Entonces tomando en cuenta lo anterior se definen las variables experimentales.

Sean:

- Z: un programa escrito en el lenguaje Pas++.
- X_1: media del tiempo de ejecución de Z luego de ser interpretado sin usar optimización de código.
- X_2: media del tiempo de ejecución de Z luego de ser interpretado usando optimización de código.

Hipótesis 1:

- H_0: $X_1 \leq X_2$: *"El tiempo de ejecución de un programa interpretado con optimización de código no es menor al interpretado sin usar esta opción".*

- H_1: $X_1 > X_2$: *"El tiempo de ejecución de un programa interpretado sin usar optimización de código es mayor al interpretado usando esta opción".*

Tesis publicada con autorización del autor
No olvide citar esta tesis

Asimismo se desea medir la efectividad de la implementación propuesta para el desarrollo del intérprete, el entorno de desarrollo y optimizador de código, esto se logrará evaluando los ciclos de vida utilizados para cada uno de ellos. Por este motivo, se definen las siguientes variables experimentales.

Sean:

- X_1: La media del tiempo de desarrollo del intérprete.
- X_2: La media del tiempo de desarrollo del entorno de desarrollo.
- X_3: La media del tiempo de desarrollo del optimizador.

Hipótesis 2:

- H_0: $X_1 \leq X_2$: *"El ciclo de vida en cascada se adapta mejor que el ciclo de vida incremental para un proyecto de este tipo".*

- H_1: $X_1 > X_2$: *"El ciclo de vida incremental se adapta mejor que el ciclo de vida en cascada para un proyecto de este tipo".*

Hipótesis 3:

- H_0: $X_3 \leq X_2$: *"El ciclo de vida evolutivo se adapta mejor que el ciclo de vida incremental para un proyecto de este tipo".*

- H_1: $X_3 > X_2$: *"El ciclo de vida incremental se adapta mejor que el ciclo de vida evolutivo para un proyecto de este tipo".*

Hipótesis 4:

- H_0: $X_1 \leq X_3$: *"El ciclo de vida en cascada se adapta mejor que el ciclo de vida evolutivo para un proyecto de este tipo".*

- H_1: $X_1 > X_3$: *"El ciclo de vida evolutivo se adapta mejor que el ciclo de vida en cascada para un proyecto de este tipo".*

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3. Implementación del aporte

En este capítulo se abordará el diseño y la implementación escogidos para el intérprete. Se comenzará con la descripción de la gramática del lenguaje Pas++, después se describirán las estructuras utilizadas, algunas estructuras primitivas, el código intermedio generado, las principales acciones semánticas, la interpretación, la administración de memoria, los algoritmos de optimización, el diseño del IDE y el ambiente de desarrollo.

3.1. Diseño de la gramática para un lenguaje orientado a objetos

Como se mencionó anteriormente, para el diseño de la gramática del lenguaje Pas++ necesitamos usar las **Gramáticas de Contexto Libre** (Context Free Grammars, CFG). Gracias a éstas podremos especificar la sintaxis de nuestro lenguaje ya que tienen la propiedad de poder describir la estructura jerárquica de las construcciones de los lenguajes de programación.

Para poder construir el analizador sintáctico utilizaremos como input la gramática del lenguaje, pero antes de esto necesitamos definir bien nuestros símbolos terminales o tokens. Estos símbolos terminales serán reconocidos

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

por el analizador léxico, el cual le indicará al analizador sintáctico qué token está actualmente en la entrada.

Con el objetivo de obtener el analizador léxico utilizaremos una herramienta llamada *Flex*, la cual necesita como entrada las reglas o patrones definidos para los tokens. Estas reglas se especifican utilizando un conjunto extendido de las expresiones regulares (véase el anexo 1 para conocer todos los tokens o símbolos no terminales que se definen para la gramática del lenguaje Pas++).

Para reconocer estos patrones se utilizan **expresiones regulares**, donde se utilizan las definiciones regulares letra y dígito**.** Las **definiciones regulares** son identificadores para definir patrones o un conjunto de caracteres. Este diseño sirve como un input para poder obtener el analizador léxico del intérprete para Pas++, que procesará *Flex* como un archivo con extensión ".yy" (para más información del archivo ".yy" véase el anexo 2). Esta definición de los tokens y de los patrones para reconocerlos es fundamental para proceder con el diseño de la gramática del lenguaje Pas++ como veremos posteriormente.

Ahora nos enfocaremos en el diseño de la gramática del lenguaje. Esta gramática debe adaptarse a lo que pretendemos, es decir, a que tenga la capacidad de soportar la declaración de clases, la instanciación de objetos y la eliminación de los mismos; además de herencia simple, para poder así reutilizar atributos y métodos declarados anteriormente; y las operaciones de modificación dinámica de código ("insert", "remove" y "modify"). Entonces, básicamente un programa escrito en Pas++ será una lista o conjunto de clases:

program : classList

Esta lista de clases debe tener una clase **"Main"**, la cual debe tener un método llamado **"main"**, el cual será el punto de arranque para nuestro programa.

Una clase estará compuesta por una lista de atributos, un constructor, un destructor y una lista de métodos: se debe definir por cada uno de los

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

métodos y atributos el nivel de acceso (público, privado o protegido) y el tipo de dato (que devolverán, en el caso de los métodos). Se debe tener la opción de herencia, para lo cual se debe indicar el identificador de la clase de la cual se está heredando. Los constructores y destructores tendrán una estructura similar a un método (para más información sobre las reglas que permiten esto véase el anexo 3).

Las producciones que complementan las anteriores son las que definen los tipos, el nivel de acceso, los parámetros formales y el conjunto de instrucciones. Con respecto a los tipos, se permitirán los tipos: int, float, string, char, bool, void y los tipos definidos por la declaración de clases (class <clase>). Por otro lado, las reglas que permiten definir los parámetros formales, permiten el paso de parámetros por referencia y por valor; los primeros se distinguirán por tener un '&' adelante. Además las reglas que permiten definir el conjunto de instrucciones de cada método nos permitirán la utilización de bloques "begin .. end" (para más información sobre estas producciones véase el anexo 4).

A partir de las producciones anteriores se desprenden las reglas que permitirán definir a las instrucciones que soportará Pas++. Éstas las podemos separar en **tres grupos**: las instrucciones simples (declar, asig, read, write, wriln, return, metCall y delete), las de estructuras de control (if, while, repeat, for y case) y las instrucciones de modificación dinámica de código (para más información sobre estas producciones véase el anexo 5).

Las reglas para definir "declar", "return" y "del" son simples y no necesitan mayor explicación (véase anexo 6). Para definir las producciones de **"asig"** es necesario explicar cómo se representarán las llamadas a los atributos. Para representar las llamadas a los atributos en la parte izquierda de una asignación utilizaremos el símbolo no terminal **"complementoObAt"**, que debe derivar a cualquier combinación sucesiva de "TID" y puntos ('.'), para ello necesitamos de otro símbolo no terminal complementario llamado complementoObAt2 (véase en el anexo 7 la definición de las reglas para asig).

En las reglas de "asig" la parte derecha de la asignación está representada por una expresión ("expr"), la cual es un símbolo no terminal que queda

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

definido por reglas que involucran al mismo “expr” y a “term”, que es otro símbolo no terminal utilizado para distinguir la mayor precedencia de algunos operadores sobre otros (“and”, “*” y “/” tienen mayor precedencia sobre “or”, “+”y “-“) (véase en el anexo 8 las reglas para “expr”).

El símbolo no terminal **“fact”** es usado para representar constantes, como: números enteros, de punto flotante, cadenas de caracteres, caracteres y booleanos; pero también podría significar un identificador (una variable), una sucesión de identificadores y puntos (atributos de objetos o llamadas a métodos de objetos), comparaciones booleanas, valores negativos, o una instanciación de un objeto mediante “new”. Para las llamadas a métodos o atributos de un objeto se aplica algo parecido a “complementoObAt”, llamado **“complementoMet”**, casi tiene la misma lógica pero permite la llamada de un método al final. Para llamar a una variable simple o a un método de la misma clase se hace uso del símbolo no terminal **“complemento”**, ya que existe un conflicto “shift-reduce”, es decir, el parser no sabe si aplicar una operación de reducción o de desplazamiento. Ahora que se ha definido “expr” podemos definir las estructuras simples que faltan (véase en el anexo 9 las reglas para definir “fact” y las estructuras simples faltantes).

Si bien algunas construcciones al parecer son redundantes, sin embargo, son de vital ayuda en el análisis semántico, específicamente en la construcción de acciones semánticas. Las estructuras de control que tiene Pas++ son las clásicas de los lenguajes de programación (véase en el anexo 10 las reglas para definirlas).

Por último, definimos las reglas para las operaciones de modificación dinámica de código, en el caso de **“insert”** se dan 2 variantes: una que recibe sólo una cadena de caracteres que puede ser una constante o una variable, y la otra que además de la cadena recibe un entero como salto que indica cuantas instrucciones en adelante se hará el “insert” (en el primer caso se asume un salto de uno). Para el caso de **“modify”** sólo existe el caso con salto y a diferencia del “insert” no realiza una inserción de código sino sobreescribe instrucciones. La operación de **“remove”** recibe dos enteros, uno es la instrucción inicio a eliminar, y el otro entero es el número de instrucciones a eliminar (para más información sobre estas reglas véase el anexo 11).

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.1.1. Diseño de la jerarquía de clases

Como mencionamos anteriormente el lenguaje Pas++ soporta programación orientada a objetos, y en este paradigma la herencia es fundamental para la reutilización de código. Por ello se pueden construir nuevas clases partiendo de las clases que ya existen, que constituye una jerarquía de clases y que no es necesario modificar. Esta jerarquía de clases se va construyendo a través de la herencia simple que existe entre una clase base y una clase derivada, es decir, un objeto hereda y puede extender las características de otro objeto (atributos y métodos), pero solamente de aquel. Además cabe mencionar que no se implementará la herencia múltiple por quedar fuera del alcance definido para el proyecto.

La regla de definición de clases como se pudo observar anteriormente es:

```
class : TCLASS TID  complementoClass;

complementoClass: TBEGIN atribList const dest  metList TEND
                | ':'  TID  TBEGIN atribList const dest metList TEND ;
```

Figura 3.1 Gramática de Pas++ (herencia)

Por lo tanto, se podrá incluir o no, herencia simple al declarar una clase. La clase padre se indicará después de ':'. Como se indicó anteriormente, dentro de la definición de la clase se hará la declaración de sus atributos y métodos, los cuales pueden ser públicos, privados o protegidos. Si el atributo o método es declarado como público, se puede acceder a él desde cualquier clase, si es privado sólo se puede acceder a él desde la clase en la que ha sido declarado y si es protegido puede ser accedido desde las clases derivadas pero no de otras clases externas.

La clase derivada puede invocar al constructor de la clase base desde su propio constructor. Además, los constructores y destructores deben tener el mismo identificador de la clase definida, pero para los segundos este identificador debe ser precedido por el carácter '%'. Además, los métodos heredados pueden ser redefinidos en la clase derivada, pero si esto sucede los métodos que habían sido heredados son renombrados con el nombre de

Tesis publicada con autorización del autor
No olvide citar esta tesis

la clase base seguido de '_' y el nombre original del método, pudiendo ser invocados con este nuevo nombre (véase en el anexo 12 un ejemplo donde se utiliza herencia en Pas++).

3.2. Estructuras de datos

Para la interpretación se necesita guardar el código intermedio que se va a interpretar, además de las clases declaradas, y por cada clase se deben guardar los atributos, los métodos y los símbolos de cada método. Entonces, en la implementación planteada se definieron las siguientes estructuras de datos:

- **Tabla de código:** en ella se guardan las operaciones de código intermedio, junto a los argumentos necesarios para ejecutarlos.

- **Tabla de clases:** en ella se guarda la declaración de las clases (símbolos y métodos).

  - **Tabla de atributos:** existen para cada clase, en ella se guardan las características de los atributos ("tatributo") de una clase.

  - **Tabla de métodos:** existen para cada clase, en ella se guardan los métodos ("tmetodo") con las características necesarias para ejecutarlos.

    - **Tabla de símbolos:** en ella se guardan los símbolos ("tsímbolo") con las características necesarias para después cargarlos en memoria ("stack"). Por razones propias de la implementación, estas tablas existen por cada método.

3.2.1. Descripción de la tabla de código

La tabla de código es la encargada de guardar el conjunto de instrucciones de un programa fuente en Pas++ en una representación intermedia de la cual se hablará posteriormente. La tabla de código es una estructura (tab_code) con

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

una lista de registros tcode y un campo entero que nos indica el número de instrucciones en la tabla, su definición en C se presenta a continuación:

typedef struct {
    int op;
    int a[NARGS];
} tcode;
	typedef struct {
	    int ninst;
	    tcode code[MAXCODE];
	} tab_code;

Figura 3.2 Estructura de tcodigo y tab_code

Descripción:

- **op:** representa al identificador de la instrucción en código intermedio.
- **a[ ]:** es un arreglo de enteros que podría guardar direcciones de memoria (offsets) de las variables involucradas en la operación indicada por op y la dirección donde se guardará el resultado de la operación (en caso existiese). También podría guardar otro tipo de operandos que dependen exclusivamente de "op".
- **NARGS:** es una constante, que indica el número máximo de argumentos que podría tener una operación.
- **ninst:** número de instrucciones de la tabla de código.
- **code:** es un arreglo o lista de tcodes.
- **MAXCODE:** es una constante, que indica el número máximo de códigos que podría existir en la tabla de código.

Cabe resaltar que durante la compilación del código fuente, específicamente en el análisis semántico se van generando los códigos que son guardados en esta tabla. Las instrucciones se agregan a la tabla de código de tal forma que las pertenecientes a un método se encuentran en direcciones contiguas dentro de la tabla de código. Por otro lado, durante el proceso de interpretación (ejecución) del código intermedio, se usa un puntero que va recorriendo cada entrada de esta tabla.

A continuación se muestra como sería la representación en la tabla de código de un programa en Pas++:

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

```
class Persona
begin

   private int edad;
   private string nombre;
   private float talla;

   public Persona()
   begin
                  this.edad=12;
                  this.nombre="marco";
                  this.talla=1.34;
   end

   sub public void escribir()
   begin
                  wriln("los datos de la persona son: \n");
                  wriln(this.nombre, "-",  this.edad, "-", this.talla);
   end

end

class Main
begin
          sub public void main()
          begin
                  Persona p;
                  p=new persona();
                  call p.escribir();
          end
end
```

Figura 3.3 Ejemplo de un programa fuente en Pas++

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Nº	Operanción	Op 1	Op 2	Op 3	Op 4	Op 5	Op
0	asignor	this	Persona	edad	12		
1	asignor	this	Persona	nombre	"marco"		
2	asignor	this	Persona	talla	1.34		
3	return						
4	wriln	string	_T_0				
5	writeo	this	Persona	nombre			
6	write	string	"-"				
7	writeo	this	Persona	edad			
8	write	string	"-"				
9	writeo	this	Persona	talla			
10	return						
11	new	this	Main				
12	new	_T_1	Persona				
13	argv	0	_T_1	Persona	Persona		
14	call	Persona	_T_1	Persona			
15	=	_T_0	_T_1	object			
16	argv	0	_T_0	escribir	Persona		
17	call	escribir	_T_2	Persona			
18	return						

Figura 3.4 Representación del programa en la tabla de código

3.2.2. Descripción de la tabla de clases

La tabla de clases es la encargada de guardar el conjunto de clases de un programa fuente en Pas++, haciendo posible disponer de la información que se necesita de una clase en el momento de la interpretación o en el análisis semántico.

La tabla de clases es una estructura (**ClassLoader)** con la lista de registros tclass y un campo entero que nos indica el número de clases guardadas, su definición en C se presenta a continuación:

```
typedef struct {
        char name[MAXNAME];
        tatrib lstAtrib[MAXATRIB];
        tmetodo lstMetod[MAXMETOD];
        int natrib;
        int nmetod;
        int size;
        int padre;
} tclass;
```

Figura 3.5 Estructura de tclass

```
typedef struct {
        tclass
        tab_class[MAXCLASS];
        int nclass;
}ClassLoader;
```

Figura 3.6 Estructura de tclass y ClassLoader

Tesis publicada con autorización del autor
No olvide citar esta tesis

Descripción:

- **name:** indica el nombre de la clase.
- **lstAtrib:** lista de atributos o tabla de atributos.
- **MAXATRIB:** es una constante, que indica el número máximo de atributos de una clase.
- **lstMetod:** lista de métodos o tabla de métodos.
- **MAXMETOD:** es una constante, que indica el número máximo de métodos de una clase.
- **natrib:** número de atributos de la clase.
- **nmetod:** número de métodos de la clase.
- **size:** espacio ocupado por los atributos de la clase en memoria (“heap”). Esto será una información de vital importancia al momento de instanciar un objeto.
- **padre:** indica el identificador de la clase padre.
- **tab_class:** tabla de clases o lista (arreglo) de tclass.
- **nclass:** número de clases.
- **MAXCLASS:** es una constante, que indica el número total de clases.

Como se puede notar, cada clase tiene una lista o tabla de métodos y otra lista o tabla de atributos:

3.2.2.1. Descripción de la tabla de atributos

La tabla de atributos es un arreglo de registros “tatrib” se utiliza para guardar la información necesaria que se necesita de los atributos de una clase, por ejemplo, para utilizarla al instanciar un objeto o darle valor a un atributo. La definición de “tatrib” se presenta a continuación:

```
typedef struct {
        char name[MAXNAME];
        int type;
        int access;
        int class;
        int offset;
        char depadre;
}tatrib;
```

Figura 3.7 Estructura de tatrib

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Este tipo de estructura, tiene los siguientes campos:

- **name:** guarda el nombre del atributo.
- **offset:** guarda la dirección relativa al inicio del objeto en memoria ("heap") que tendrá el atributo.
- **type:** guarda el tipo de dato del atributo.
- **class:** guarda el número que identifica a una clase en la tabla de clases (índice) si el atributo es de tipo objeto.
- **access:** tipo de acceso del atributo (público, protegido o privado).
- **depadre:** si tiene el valor de 1 el atributo ha sido heredado, en caso contrario no proviene de herencia

3.2.2.2. Descripción de la tabla de métodos

La tabla de métodos es la encargada de guardar la información necesaria que se necesitan de los métodos de una clase para su utilización, por ejemplo, al momento de invocar un método o retornar del mismo. La definición de "tmetodo" se presenta a continuación:

```
typedef struct {
        char name[NAME];
        int narg;
        int res_dir;
        int nsim;
        int ntemp;
        int size;
        int type_dev;
        int class;
        int access;
        int init_code;
        int end_code;
        tsimbolo tab_sim[MAXSIM];
        char depadre;
}tmetodo;
```

Figura 3.8 Estructura de tmetodo

Descripción:

- **name:** indica el nombre del método.
- **narg:** número de argumentos en el momento de llamar a dicho método.
- **res_dir:** dirección donde se devolverá el resultado que devuelva el método.
- **nsim:** número de símbolos que tendrá el método.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **ntemp:** número de variables temporales que tendrá el método.
- **size:** tamaño en bytes que tendrá la tabla de símbolos del método cuando se cargue en memoria (“stack”).
- **type_dev:** tipo de dato que devolverá el método (por ejemplo: TTINT, TTFLOAT, etc).
- **class:** índice o identificador de la clase, en caso que el método devuelva un objeto.
- **access:** tipo de acceso del método (público, protegido o privado).
- **init_code:** número de código de inicio del método en la tabla de código.
- **end_code:** número de código de fin del método en la tabla de código.
- **tab_sim:** tabla de símbolos del método. Que se describirá más adelante.
- **depadre:** si tiene el valor de 1 el método ha sido heredado, en caso contrario no proviene de herencia.
- **MAXSIM:** es una constante que indica el máximo número de símbolos.

3.2.2.3. Descripción de la tabla de símbolos

La tabla de símbolos es básicamente un arreglo de registros tsimbolo. Este tipo de estructura, tiene los siguientes campos:

- **name:** guarda el nombre del símbolo.
- **offset:** guarda la dirección relativa al inicio del método en memoria que tendrá el símbolo.
- **type:** guarda el tipo de dato del símbolo. Ejemplo: TTINT, TTCHAR, etc. Es útil para saber cuántos bytes ocupará en memoria.
- **token:** guarda el tipo de token que le corresponde al símbolo. Es útil en el caso de ser un parámetro que se pasa en un método, para saber si es uno por valor o por referencia, también podría indicar si el símbolo es una constante (entera, real, carácter, etc.). Ejemplo: TARGR, TARGV, TTINT, etc.
- **class:** guarda el número que identifica a una clase en la tabla de clases (índice) si el símbolo es de tipo objeto.

Esta estructura es útil para reunir a todos los símbolos de un determinado método, guardando información asociada a los nombres de los mismos. Cada

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

vez que se encuentra un nuevo nombre, se busca si existe un símbolo con ese nombre en la tabla de símbolos de ese método, entonces se puede validar (si es una variable) si el símbolo ha sido declarado o no, o crear uno nuevo si es un declaración o un valor constante(ejm: 9, 'a',etc). También se pueden producir actualizaciones de dichos símbolos, si ese es el caso se cambia los valores de los campos de **tsimbolo**.

```
typedef struct {
        char nombre[MAXNAME];
        int type;
        int token;
        int offset;
        int class;
} tsimbolo;
```

Figura 3.9 Definición de tsimbolo

3.3. Descripción de algunas operaciones primitivas del lenguaje Pas++

En el lenguaje Pas++ se definen ciertas operaciones primitivas que son necesarias para la creación de objetos y la modificación dinámica de código. A continuación se explica las acciones que se realizan para ejecutar las operaciones primitivas del lenguaje Pas++:

<u>Creación y destrucción de objetos</u>

3.3.1. Instrucción new

La instrucción "new" es la instrucción necesaria para la creación de un objeto de una clase particular. Esta operación realiza una separación de memoria para la declaración del objeto.

El espacio de memoria asignado se encuentra en el "heap", para ello se reserva un espacio dependiendo de la cantidad de atributos y del tipo de los mismos. Esta operación genera una entrada en la lista de objetos ("ObjectList") y un índice para acceder a ella en dicha lista. Este índice es guardado después en parte de memoria asignada para "id". Asimismo, se

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

hace una llamada al constructor si este ha sido definido en la declaración de la clase.

La sintaxis de la instrucción es la siguiente:

```
id = new clase(p1,p2,..);
```

Donde **"id"** indica la variable a la cual se le asignará la referencia (índice) del objeto creado, mientras que "clase" indica el nombre de la clase de la cual se está creando el objeto y **"p1,p2,…"** aquellos parámetros que se le pasan al método constructor si existiese.

3.3.2. Instrucción delete

La instrucción delete es la instrucción utilizada para la eliminación del espacio en memoria en el "heap" que ocupa un objeto. Esta operación en primer lugar elimina la entrada correspondiente al objeto en la lista de objetos "ObjectList". Asimismo, se libera el espacio separado en el "heap" durante la declaración del objeto, para que pueda ser utilizado posteriormente. También se podría hacer una llamada al método destructor si éste ha sido definido.

La sintaxis de la instrucción es la siguiente:

```
delete id(p1,p2,…);
```

Donde **"id"** indica a la variable que guarda la referencia (índice) al objeto que se desea eliminar y **"p1,p2,…"** aquellos parámetros que se le pasan al método destructor si existiese.

<u>Modificación dinámica de código</u>

3.3.3. Instrucción insert

La instrucción "insert" es una de las operaciones que implementan la modificación dinámica de código. Esta operación realiza una inserción de una instrucción determinada en la tabla de código en tiempo de ejecución. La inserción se realiza en una determinada posición, por lo que el orden de algunas instrucciones se altera.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

La instrucción recibe como parámetro una cadena de caracteres que representa una instrucción del lenguaje Pas++ y cuántas posiciones en adelante se insertará la instrucción.

La sintaxis de la instrucción "insert" es la siguiente:

insert(instrucción, n_posiciones);

Donde "**instrucción"** indica la cadena de caracteres que representa una instrucción del lenguaje y **"n_posiciones"** indica el número de posiciones en adelante donde se desea añadir la instrucción. Se puede omitir el segundo argumento y por defecto se considera como 1 (véase en el anexo 16 un ejemplo de un programa que utiliza esta operación).

3.3.4. Instrucción remove

La instrucción "remove" es la segunda operación que implementa la modificación dinámica de código. Esta instrucción se puede considerar como la operación inversa a la instrucción "insert", ya que elimina una entrada de la tabla de código en tiempo de ejecución.

La instrucción recibe como parámetro el número de instrucciones en adelante de donde se quiere comenzar a eliminar instrucciones y el número de instrucciones a eliminar.

La sintaxis de la instrucción "remove" es la siguiente:

remove(n_posiciones , cantidad);

Donde **"n_posiciones"** indica el número de posiciones en adelante para comenzar a eliminar instrucciones, y **"cantidad"** el número de instrucciones a eliminar.

3.3.5. Instrucción modify

La instrucción "modify" es la tercera operación que implementa la modificación dinámica de código. Esta instrucción modifica (sobrescribe) una entrada de la tabla de código en tiempo de ejecución.

Tesis publicada con autorización del autor
No olvide citar esta tesis

La instrucción recibe como parámetro una cadena de caracteres que representa una instrucción del lenguaje Pas++ y cuántas posiciones en adelante se sobrescribirá la instrucción.

La sintaxis de la instrucción “modify” es la siguiente:

modify(“instrucción”, n posiciones);

Donde **“instrucción”** indica la cadena de caracteres que representa una instrucción del lenguaje y **“n_posiciones”** indica el número de posiciones en adelante para comenzar a sobreescribir instrucciones.

3.4. Descripción del código intermedio generado

Anteriormente se indicó que durante la fase de análisis y síntesis del intérprete se traduce el código fuente a una representación intermedia. Posteriormente, este código intermedio es el que finalmente se interpreta en la fase de ejecución.

Las razones para transformar el código fuente en una representación intermedia se presentan a continuación:

- Si se quisiera generar código nativo, se tendría la facilidad de hacerlo para distintas plataformas. Así sólo cambiaríamos una parte del compilador y no en su totalidad.
- Realizar optimizaciones al código intermedio permite que éstas sean independientes de la plataforma donde se está ejecutando el programa.
- Si se quisiera realizar descompilación de código, es mucho más fácil hacerlo en la representación intermedia que en código máquina.

El código intermedio es almacenado en la tabla de código. Cada entrada en la tabla código representa una instrucción en representación intermedia. La representación está dada en septetos. Cada septeto guarda un código de operación y 6 miembros.

Tesis publicada con autorización del autor
No olvide citar esta tesis

Como se sabe, una operación binaria es la que utiliza dos operandos para realizarse, asimismo una operación unaria es la que necesita un solo operando. Según esto, se han clasificado las operaciones de código intermedio en ocho tipos dependiendo si son operaciones unarias o binarias, si devuelven o no algún tipo de resultado, si son de entrada/salida, si están vinculadas a modificación dinámica de código o manipulación de objetos:

3.4.1. Operaciones binarias con resultado

Las operaciones de este tipo guardarán el resultado de la operación en el primer elemento del septeto, y la operación propiamente dicha se aplicará al segundo y tercer miembro. Para la mayoría de estas operaciones se necesita saber el tipo de dato a la que pertenece el resultado, el cual se encontrará en el cuarto miembro (esto último no aplica para TAND y TOR).

Operación	Descripción
‘+’	Indica la operación aritmética de adición.
‘-’	Indica la operación aritmética de substracción.
‘*’	Indica la operación aritmética de multiplicación.
‘/’	Indica la operación aritmética de división.
TAND	Indica AND booleano.
TOR	Indica OR booleano.
TIGU	Indica la operación “Igual que”.
TDIF	Indica la operación “diferente que”.
‘<’	Indica la operación “menor que”.
‘>’	Indica la operación “mayor que”.
TMENORI	Indica la operación “menor igual que”.
TMAYORI	Indica la operación “mayor igual que”.

3.4.2. Operaciones binarias sin resultado

Este tipo de operación necesitan de dos operandos para realizarse, los que estarán guardados en el primer y segundo miembro del septeto.

Operación	Descripción
JUMPF	Operación “saltar si es falso”. Si el primer operando es falso, se salta a la dirección dada por el segundo operando.
JUMPT	Operación “saltar si es verdadero”. Si el primer operando es verdadero, se salta a la dirección dada por el segundo operando.
‘=’	Operación de asignación. Copia el valor del operando dos al operando uno.
CDIR	Indica la copia de la dirección de un argumento pasado por referencia
ASIGR	Indica una asignación de una variable que es referencia

Tesis publicada con autorización del autor
No olvide citar esta tesis

3.4.3. Operaciones unarias con resultado

Este tipo de operación necesitan de un operando para realizarse, la operación se aplica al segundo miembro y se guarda en el primer miembro.

Operación	Descripción
NEG	Indica la operación menos unaria.
NOTB	Indica la operación not booleana
CFLOAT	Operación para convertir un int en float.

3.4.4. Operaciones unarias sin resultado

Este tipo de operación necesitan de un operando para realizarse, la operación se aplica al primer o segundo miembro.

Operación	Descripción
TINC	Incremento unitario.
TDEC	Decremento unitario.
JUMP	Salto incondicional.

3.4.5. Operaciones con múltiples operandos

Estas operaciones tienen más de 2 miembros y no caben en ninguna de las categorías anteriormente mencionadas. Los parámetros que utilizan se encuentran en las posiciones del septeto de manera indistinta.

Operación	Descripción
TCALL	Operación que realiza la llamada a un método.
TRETURN	Indica el retorno de un método, si éste devuelve un resultado el operando uno guarda la dirección en donde se guardará.
TARGV	Indica un argumento pasado por valor.
TARGR	Indica la un argumento pasado por referencia.

3.4.6. Operaciones de lectura/escritura

Este tipo de operaciones provienen de una instrucción en Pas++ de lectura o escritura con un número de argumentos variable. En general, cuando hay más de un argumento en la instrucción, estas operaciones de código intermedio van seguidas unas de otras.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Operación	Descripción
TWRILN	Escritura del operando con un cambio de línea al final.
TWRITE	Escritura del operando.
TREAD	Lectura de un operando.

3.4.7. Operaciones para manejo de objetos

Los parámetros que utilizan se encuentran en las posiciones del septeto de manera indistinta. Algunas son operaciones que extienden operaciones anteriores para permitir el manejo de objetos.

Operación	Descripción
TNEW	Operación para crear un objeto de una clase determinada.
TDELETE	Operación para la eliminación de memoria generada dinámicamente.
ASIGNO	Operación que se realiza cuando en una asignación interviene el atributo de un objeto o un objeto, al lado izquierdo y derecho de la misma.
BOBJETO	Operación para copiar el índice en la tabla de objetos de un atributo de tipo objeto a una variable temporal (buscar un objeto).
ASIGNOR	Operación que se realiza cuando en una asignación interviene el atributo de un objeto o un objeto, al lado izquierdo o derecho de la misma.
TREADO	Lectura del atributo de un objeto.
TWRITEO	Escritura del atributo de un objeto.

3.4.8. Operaciones para modificación dinámica de código

Los parámetros que utilizan se encuentran en las posiciones del septeto de manera indistinta. Son operaciones para la modificación dinámica de código.

Operación	Descripción
TINSERT	Operación que permite insertar o modificar ("insert" o "modify") instrucciones en la tabla de código.
TREMOVE	Operación que permite remover instrucciones en la tabla de código.

Véase el anexo el anexo 13 para mayor detalle acerca de los miembros de las operaciones de código intermedio.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.5. Acciones semánticas principales

Las acciones semánticas permiten ir armando las estructuras (descritas anteriormente) que nos permitirán realizar la interpretación de código intermedio y ejecución del programa. Éstas están inmersas en la gramática de Pas++ creada con la sintaxis de Bison, de esta manera cuando se genere el compilador, determinada acción semántica se ejecutará cuando se reconozca la producción de la gramática correspondiente. Las acciones semánticas generalmente generarán un código intermedio, para ello invocarán a la función generaCodigo():

```
int generaCodigo(int op,int *a){
        int i = ct.ninst;
        ct.code[i].op = op;
        memcpy(ct.code[i].arg,a,NARGS*sizeof(int));
        for (int j=0; j<NARGS; j++) a[j]=-1;
        ct.ninst++;
        return i;
}
```

Figura 3.10 Función generaCodigo()

La mayoría de acciones semánticas tienen significados, así que alguien que ya ha usado Yacc o Bison para construir un compilador o un intérprete sencillo, podría entender la mayoría de las acciones semánticas que se usan en Pas++. Sin embargo, es necesario explicar aquellas que estén relacionadas a las dos características más importantes de este intérprete: programación orientada a objetos y modificación dinámica de código. Es importante señalar en la descripción posterior se omiten las acciones semánticas relacionadas a verificación de tipos y validaciones.

Con respecto a la programación orientada a objetos, se tienen tres grupos de acciones semánticas relacionadas a:

- La declaración de clases
- ComplementoObAt, complementoObAt2 y ComplementoMet
- New y Delete

Tesis publicada con autorización del autor
No olvide citar esta tesis

Las acciones semánticas relacionadas a la declaración de clases permiten la búsqueda o definición en el ClassLoader de métodos, atributos y clases; además proporcionan la capacidad para crear herencia.

Las acciones de búsqueda permiten básicamente encontrar el índice (posición) de una clase, método o atributo en su correspondiente lista, usando como parámetro de búsqueda el nombre (y la clase, en los casos de método y atributo). Las acciones de definición inicializan una entrada de una clase, método o atributo en su correspondiente lista (tabla de clase, lista de métodos o lista de atributos). Para la creación de herencia se realiza una copia parcial de una entrada de la tabla de clases en otra, además se definen en la clase derivada los métodos y atributos de la clase base, pero activando el flag "depadre" para indica que son heredados. A continuación se muestra algunos fragmentos de la gramática donde se utilizan las acciones semánticas anteriormente descritas:

```
class : TCLASS TID { strcpy(strAux,$2.text); } complementoClass;

complementoClass:
{ idx_clase = definirClase(strAux,-1); }
TBEGIN atribList const dest  metList TEND
|
{ idx_clase = definirClase(strAux,-1); }
':'  TID
{ int h = buscarClase(yytext); crearHerencia(h,idx_clase); }
TBEGIN atribList const dest  metList TEND
;

const: | TID
{idx_metodo =definirMetodo(idx_clase,yytext,$1.a1,TTOBJECT,idx_clase); }
'(' formalList ')' TBEGIN  instrList TEND
;

dest :   | '%'TID { sprintf(aux, " %s", yytext); aux[0]='%'; }
{idx_metodo = definirMetodo(idx_clase,aux,$1.a1,TTOBJECT,idx_clase); }
'(' formalList ')' TBEGIN instrList TEND
;
```


Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

```
atribList :  atribList atributo |    ;

atributo:  acceso tipo TID
{ definirAtributo(idx_clase,yytext,$2.a1,$1.a1,$2.a2); } ';'   ;

metList:  metList metodo |  ;

metodo:  TSUB acceso tipo TID
{ idx_metodo = definirMetodo(idx_clase,yytext,$2.a1,$3.a1,$3.a2);
if (idx_metodo==-1) YYABORT;}
'('   formalList ')' TBEGIN instrList TEND      ;
```

Figura 3.11 Acciones semánticas para declaración de clases

La producción ComplementoObAt permite reconocer aquellas secuencias de objetos que tienen como atributos otros objetos, que pueden continuar la recursividad. Se tiene el siguiente ejemplo:

```
(…)
Libro compiAho;
compiAho= new Libro();
(…)
compiAho.autor= new Persona();
compiAho.autor.origen=new Pais();
compiAho.autor.origen.nombre="Canada";
(...)
wriln(compiAho.autor.origen.nombre);
```

Figura 3.12 Ejemplo de complementoObAt y complementoMet

Como se puede observar tenemos un objeto (compiAho) que es de la clase Libro, que tiene un atributo autor (objeto perteneciente a la clase Persona), éste a su vez tiene un atributo origen (un objeto perteneciente a la clase Pais) que tiene como uno de sus atributos a nombre (de tipo string). Esta secuencia puede haber sido más larga, si los atributos de un objeto hubieran continuado siendo también objetos. Las producciones complementoObAt y complementoObAt2 que permiten el reconocimiento de estas secuencias fueron descritas anteriormente, pero es importante explicar las acciones semánticas involucradas.

Con respecto a complementoObAt2, reconoce una secuencia de identificadores y puntos, o sólo un identificador. Es importante resaltar que los resultados de esta producción se guardan en a[0] y a[1]. Si es un solo identificador, en a[0] se guarda el índice del símbolo (con relación a la tabla de símbolos) que representa al objeto representado por el identificador (TID)

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

actual y en a[1] la clase a la que pertenece este símbolo (objeto). Si es una secuencia de identificadores y puntos se debe generar el código BOBJETO, que copiará el valor del índice (con relación a la lista de objetos) del objeto que actualmente se busca a un símbolo temporal en memoria; luego se guarda en a[0] el índice de este temporal en la tabla de símbolos y en a[1] la clase a la que pertenece el objeto actual.

Luego, en complementoObAt se cuenta con el resultado de complementoObAt2 y teniendo la certeza de que es final de una secuencia de identificadores y puntos; entonces, sólo se guarda en la pila ($$) el offset del símbolo que proviene de complementoObAt2 que se encuentra en a[0], también la clase que está guardada en a[1] y por último el índice del atributo cuyo nombre está guardado en el identificador actual (TID) (véase el anexo 14 para conocer con más detalle las acciones semánticas de complementoObAt2 y complementoObAt).

Con respecto a complementoMet, tiene una lógica muy parecida a complementoObAt, con la diferencia que permite invocar un método al final. Utiliza también a complementoObAt2, cuyas acciones semánticas ya han sido descritas. Por lo que recibe en a[0] el índice del símbolo (en la tabla de símbolos) que representa al objeto actual y en a[1] la clase a la que pertenece este objeto. Luego, ya que se tiene la certeza de que es final de una secuencia de identificadores y puntos, se tienen dos posibles casos: que el último identificador sea un atributo o que sea el nombre de un método, al que le deben seguir sus parámetros. Entonces, si se está en el primer caso la situación es idéntica a complementoObAt, sólo se guarda en la pila ($$) el offset del símbolo que proviene de complementoObAt2 que se encuentra en a[0], también la clase que está guardada en a[1] y por último el índice del atributo cuyo nombre está guardado en el identificador actual (TID). En el segundo caso, se debe generar el código TARGV (parámetro por valor) para pasar el índice o puntero al objeto (como el puntero this en C++) que invoca al método como primer parámetro. Luego, se hace uso de actualList que también genera códigos TARGV O TARGR dependiendo si los parámetros que se pasan son por valor o referencia, después se genera el temporal donde se guardará el resultado devuelto, y por último se genera el código TCALL enviándole el índice del método, la clase y el offset del temporal

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

generado (véase el anexo 15 para mayor detalle sobre las acciones semánticas correspondientes a complementoMet).

El tercer grupo de acciones semánticas relacionadas a programación orientada a objetos es el que permite instanciar y eliminar objetos creados en el “heap” (“new” y “delete”). Con respecto a “new”, forma parte de la producción fact y las acciones semánticas principales permiten primero referenciar la clase que se usa para instanciar el objeto; luego generar el código TNEW pasándole como parámetros el offset del temporal tipo objeto donde se guardará el índice (en la lista de objetos) del objeto que se está creando y el índice de la clase; y por último, ejecutar las mismas acciones semánticas que se usan para invocar un método y que se explicaron en complementoMet (se busca el método, se pasa como parámetro por valor el puntero al objeto actual, se invoca a actualList y se genera el código TCALL) . A continuación las acciones semánticas para “new” (se han omitido aquellas que nos permiten invocar a un método):

```
fact: TNEW TID
{int i = referenciarClase(yytext);
 int temp = genera_temporalCC(idx_clase,idx_metodo,TTOBJECT,i);
 $$.a1 = temp;
 a[0] = cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[temp].offset;
 a[1] = i;
 a[2] = TTOBJECT;
 $$.a2=i;
 generaCodigo(TNEW,a);
} '(' actualList ')' ;
```

Figura 3.13 Acciones semánticas para new

Para el caso de delete se realizan acciones semánticas parecidas a “new”, con la diferencia que primero se busca el símbolo objeto que se va a eliminar, luego se invoca al método (similar a TNEW y complementoMet), y por último se genera el código TDELETE pasándole como parámetros el offset del temporal tipo objeto donde se guardará el índice (en la lista de objetos) del objeto que se está eliminando y el índice de la clase. A continuación las acciones semánticas (se omiten las que permiten invocar al método):

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

```
del : TDELETE TID {
tsimbolo *aux= cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim;
int s=buscar_simbolo( idx_clase, idx_metodo, $2.text);
$$.a1=s;
int cla= aux[s].class;
$$.a2=cla;
} '('  actualList ')' {
a[0] = cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[$3.a1].offset;
a[1] = $3.a2;
generaCodigo(TDELETE,a);
};
```

Figura 3.14 Acciones semánticas para delete

Con relación a la modificación dinámica de código, se tienen tres grupos de acciones semánticas relacionadas a:

- Insert
- Modify
- Remove

Las acciones semánticas de "insert" buscan generar el código intermedio TINSERT, que recibe como parámetros el offset del símbolo string (que es el código a insertar), la clase, el método, el offset del símbolo entero (que es la cantidad de instrucciones hacia adelante donde se insertará el código) y un flag que indica si se va a realizar un "insert" o un "modify" (en este caso es 0 por ser un "insert"). Como se explicó anteriormente, si no se incluye el segundo parámetro del "insert" (el salto) este por defecto es 1. A continuación se muestran las acciones semánticas para "insert" (se omite el caso por defecto):

```
insert : TINSERT '(' expr {
    a[2]=$3.a2;
}complementoInsert;

complementoInsert:  ')' |
',' expr {
int t=cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[$2.a2].type;
} ')'
{a[0] = cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[a[2]].type;
 a[1] = cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[a[2]].offset;
 a[2]=idx_clase;
 a[3]=idx_metodo;
 a[4]=cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[$2.a2].offset;;
 a[5]=0;
 generaCodigo(TINSERT,a); };
```

Figura 3.15 Acciones semánticas para "insert"

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Las acciones semánticas relacionadas a “modify” son semejantes a “insert”, incluso generan el mismo código intermedio TINSERT con la diferencia de que el flag que se guarda en a[5] está en 1. Además en “modify” se necesitan los dos parámetros obligatoriamente, por lo que su gramatica sólo tiene una regla.

```
modify :   TMODIFY '(' expr { a[2]=$3.a2; }  ',' expr  ')'  {
a[1] = cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[a[2]].offset;
a[2]=idx_clase;
a[3]=idx_metodo;
a[4]=cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[$6.a2].offset;
a[5]=1;
generaCodigo(TINSERT,a);     };
```

Figura 3.16 Acciones semánticas para “modify”

A diferencia de las dos anteriores, “remove” no recibe el offset de un símbolo string, sino dos offsets a símbolos enteros que son las instrucciones hacia adelante (o salto) y la cantidad de instrucciones a remover.

```
remove : TREMOVE '(' expr { a[2]=$3.a2; }  ',' expr  ')'  {
a[1] = cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[a[2]].offset;
a[2]=idx_clase;
a[3]=idx_metodo;
a[4]=cl.tab_class[idx_clase].lstMetod[idx_metodo].tab_sim[$6.a2].offset;
generaCodigo(TREMOVE,a);     };
```

Figura 3.17 Acciones semánticas para “remove”

3.6. Interpretación de código intermedio y administración de memoria

La interpretación de código intermedio permite realizar la función final del intérprete, la ejecución del programa. Para ello se tiene la tabla de código y la tabla de clases, productos finales de la fase de análisis (y de algunas de síntesis), que deberán interpretarse siguiendo las definiciones de las operaciones de código intermedio. Específicamente, se utilizan los siguientes parámetros, estructuras y espacios de memoria:

- **idx_clase**: la clase actual que se está interpretando.
- **idx_metodo:** el método actual de la clase que se está interpretando.
- **Memory:** el espacio de memoria del “stack”

Tesis publicada con autorización del autor
No olvide citar esta tesis

- **dir_init:** la primera dirección libre en el "stack".
- **cx:** el contador del programa.
- **Sra:** pila de registros de activación.
- **Ct:** tabla de código.
- **O:** lista de objetos.
- **H:** "heap".
- **Cl:** tabla de clases.

La administración de memoria se realiza en base a los conceptos de "stack" y "heap", en el primero se almacenan las variables locales de los métodos declarados y en el segundo se almacenan los valores de los atributos de los objetos instanciados. En tiempo de ejecución, el "heap" va creciendo conforme vaya avanzando el programa y es utilizado cuando se necesite conocer o guardar el valor de un atributo de un determinado objeto.

Al realizar la interpretación, el primer paso es la búsqueda de la clase Main y dentro de ella del método main en la tabla de clases. Si éste clase o método no ha sido definido entonces el programa no puede continuar. En caso contrario, se comienza a ejecutar un bucle que irá obteniendo cada instrucción en código intermedio de la tabla de código. El contador del programa (cx) tendrá como inicio la primera instrucción del método main de la clase Main, e irá cambiando en cada interpretación de un código intermedio hasta llegar a la última instrucción.

Básicamente, cuando se manejan direcciones por cada símbolo o atributo, éstas son relativas al método o al objeto al que pertenecen, es decir, son un "offset". Así, cuando inicializamos las constantes en memoria al llamar a un método, o cuando se desea almacenar u obtener el valor de una variable, siempre se considera la dirección de inicio del método ("dir_init") en el "stack" y el "offset" de la variable. De manera similar, cuando se asigna o se obtiene el valor de un atributo de un objeto se debe tomar en cuenta el offset y la dirección inicial del objeto en el "heap".

3.6.1. Llamadas a métodos

Para realizar una llamada a un método se utilizan bloques contiguos de almacenamiento, a los que se les llama **registros de activación**. En ellos se

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

guarda la información necesaria para la ejecución de un método de una clase. Es común usar una pila para manejar de manera adecuada los registros de activación. La llamada a un método implica empujar un nuevo registro a la pila ("push"), mientras que el retorno de un método implica sacar el último registro empujado ("pop").

La estructura de un registro de activación incluye:

- La dirección o posición de retorno en la tabla de código, es decir, el contador del programa antes de realizar la llamada.
- La dirección de memoria de inicio en el "stack" de la activación del método.
- La dirección (offset) de la variable que guardará el resultado de la llamada al método. Esta dirección es con respecto al inicio del método que hace la llamada.
- La clase y método donde se hace la llamada al método.

La interpretación de las siguientes instrucciones de código intermedio utiliza la pila de registros de activación:

TCALL

Permite la llamada a un método y los parámetros que recibe son mencionados en la sección 3.4. Lo que básicamente hace la interpretación de este código intermedio es hacer "push" del registro de activación correspondiente a este método.

Cuando se realiza una llamada a un método, además de hacer un "push" a la pila de registros de activación, se hace una inicialización de las constantes de ese método en el "stack".

Hacer "push" a la pila de registros de activación, implica crear un nuevo registro de activación para el método que se está invocando, es decir, establecer en éste la posición de retorno en la tabla de código, el inicio en el "stack", la dirección del resultado, el método y la clase de donde se produce la invocación del método. Por otro lado, se actualizan parámetros globales como cx, dir_init, idx_clase e idx_metodo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

TRETURN

Permite el retorno de un método y los parámetros que recibe son mencionados en la sección 3.4. Lleva a cabo un "pop" a la pila de registros de activación, lo que implica actualizar el contador del programa, dir_init, idx_clase e idx_metodo por sus valores anteriores guardados en el registro de activación. También se debe almacenar el resultado que retorna del método (si existiese) en la dirección de memoria reservada previamente para tal propósito y que se puede obtener también del registro de activación.

3.6.2. Instanciación y eliminación de objetos

Para la instanciación la manipulación de objetos se utiliza una estructura adicional a las descritas anteriormente llamada **lista de objetos**. Esta lista guarda un registro por cada objeto instanciado con new, luego este registro es eliminado al hacer delete. Cada registro "tobject" guarda lo siguiente:

- El índice de la clase a la cual pertenece el objeto.
- El offset con relación al inicio del "heap" donde comienza a guardarse el objeto en memoria.

La interpretación de las siguientes instrucciones de código intermedio permite la manipulación de objetos:

TNEW

Permite la creación de un objeto en el "heap", los parámetros que recibe son mencionados en la sección 3.4. Realiza básicamente una reservación de espacio de memoria en el "heap" para este objeto y crea una entrada en la lista de objetos, estableciendo el offset correspondiente y su clase. Además, guarda el índice del objeto en el espacio de memoria en el "stack" reservado para el símbolo (variable) tipo objeto (parte izquierda en la asignación del new).

TDELETE

Permite la destrucción de un objeto en el "heap", los parámetros que recibe son mencionados en la sección 3.4. Se realiza una liberación de espacio en el "heap", además se elimina la entrada correspondiente en la lista de objetos y se actualiza ésta.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.6.3. Operaciones con atributos

Para realizar operaciones con atributos de un objeto se necesita ir hasta la lista de objetos para conocer el offset del objeto en el "heap", luego se debe obtener el offset del atributo con respecto al inicio del objeto, esto se obtiene directamente de la tabla de clases. Lo anterior se puede lograr teniendo el índice del objeto en la tabla de objetos y la clase a la que pertenece.

Algunas instrucciones de código intermedio relacionadas a operaciones con atributos son:

ASIGNO

Permite la asignación de un atributo pudiendo estar al derecho de la asignación otro atributo, los parámetros que recibe son mencionados en la sección 3.4. Utiliza los índices guardados en los símbolos tipo objeto, que permiten ir a la lista de objetos y obtener los offsets para hacer la asignación en memoria.

BOBJETO

Permite copiar a una variable temporal el índice de un objeto que es un atributo de otro objeto, los parámetros que recibe son mencionados en la sección 3.4. Utiliza un índice para ir a la lista de objetos, obtener el offset del objeto, luego el offset del atributo tipo objeto e ir al "heap" para recuperar el valor de dicho atributo, el cual será un índice del objeto en la lista de objetos. Finalmente hace la asignación de este valor a la variable temporal en "stack".

ASIGNOR

Permite una asignación donde un atributo está involucrado, pudiendo estar éste al lado derecho o izquierdo de la asignación Los parámetros que recibe son mencionados en la sección 3.4. Utiliza el índice guardado en el símbolo tipo objeto, que permite ir a la lista de objetos y obtener el offset para hacer la asignación en memoria. La asignación cambia si el atributo está al lado derecho o izquierdo de la asignación, pudiendo ser éste asignado a una variable del "stack" si está al lado derecho, o asignándole a éste una variable del "stack" si está al lado izquierdo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.6.4. Modificación dinámica de código

Para la modificación dinámica de código se recibe la instrucción que se quiere insertar, modificar o remover y ésta debe pasar por el analizador léxico y sintáctico, además de ejecutar las reglas semánticas que produzcan el código intermedio correspondiente, para lo cual se debe volver a llamar a la función *yyparse()*.

Las instrucciones de código intermedio relacionadas a modificación dinámica de código son:

TINSERT

Permite la inserción o modificación de instrucciones en la tabla de código, los parámetros que recibe son mencionados en la sección 3.4. Podría realizar un “insert” o un “modify” dependiendo de uno de los parámetros, pero en cualquiera de los dos casos se activa un flag que indica que se está en modificación dinámica de código (enMDC), si se está en “modify” también se activa otro flag (enModify); luego, se obtiene el salto; y, finalmente, se hace la llamada a *yyparse()* para que éste reciba como entrada el string que contiene a la instrucción en código fuente y genere nuevas instrucciones de código intermedio en la tabla de código. En el caso de un “insert” se debe hacer un corrimiento antes de hacer la inserción, en cambio en un “modify” se sobreescriben instrucciones, por lo que es una operación muy riesgosa y que puede dejar el programa prácticamente inservible.

TREMOVE

Permite la remoción de instrucciones en la tabla de código, los parámetros que recibe son mencionados en la sección 3.4. Primero, se obtiene el salto y la cantidad de instrucciones a remover, luego se borran las instrucciones comprometidas y se reordenan las que quedan.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.7. Algoritmos de optimización

La optimización de código es el conjunto de transformaciones que se le pueden realizar a un código intermedio para lograr que éste produzca un programa más rápido y con menor gasto de memoria. Sin embargo, según [ASU2007], existen algunas condiciones que se deben de cumplir:

- La optimización debe preservar el significado de los programas.
- Debe también, en promedio, mejorar la velocidad de los programas de manera **apreciable**.
- Debe valer el esfuerzo llevarla a cabo.

Algunas transformaciones que preservan la funcionalidad y mejoran el programa son:

- **Eliminación de las subexpresiones comunes:** esta transformación funciona cuando existen varias expresiones que calculan el mismo valor. Básicamente lo que se hace aquí es observar si una expresión E ha sido calculada previamente y el valor de las variables en E no han cambiado desde el último cálculo, si es así se evita calcularla otra vez. Al principio, esta técnica puede aplicarse a sólo bloques, después se puede buscar subexpresiones de manera global.

- **Propagación de copias:** esta transformación funciona junto con la anterior. Lo que se intenta es usar la copia de una variable (en vez de la misma) en las instrucciones que continúan el programa. Por ejemplo, si hay una asignación f=g, la idea es usar g en vez de f.

- **Eliminación de código muerto:** una variable se dice que está viva en un punto del programa si su valor puede ser usado subsecuentemente en el mismo, de otra manera está muerta en ese punto.
  Lo que se hace, entonces, es eliminar las partes de código que contienen asignaciones de estas variables muertas, o también eliminar partes de código que nunca serán alcanzadas ya que la variable que permite llegar a esa parte del flujo es constante.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **Optimizaciones de bucles:** se podría producir una mejora de tiempo en un programa si se disminuye el número de instrucciones en un bucle, en donde el programa pasa mucho tiempo. Para conseguir esta optimización existen tres técnicas importantes:

    - **Movimiento de código:** que se basa en mover el código del interior del bucle hacia fuera de él. Este código generalmente es una expresión que siempre da el mismo resultado dentro del bucle, por lo tanto, se le debe colocar antes del mismo.

    - **Eliminación de variables de inducción:** estas variables van cambiando de forma constante en el transcurso de un bucle, entonces esta técnica se basa en reducir todas las variables de este tipo a sólo una.

    - **Reducción en fuerza:** se trata de reemplazar una operación costosa por una más barata en términos de procesamiento. Por ejemplo, una suma por una multiplicación.

Es importante tomar en cuenta que antes de aplicar alguna de estas optimizaciones es necesario crear el grafo de control de flujo. Para realizar la creación del grafo es necesario encontrar los nodos (bloques básicos) y unirlos dependiendo de la estructura del programa. Para lograr esto, se analiza la tabla de código, encontrando los líderes (primera instrucción de un bloque básico). Una instrucción es un líder si:

- Es la primera instrucción del programa.
- Es una instrucción siguiente a un salto.
- Es el destino de un salto.

Finalmente la extensión de un bloque básico está limitada por su líder y la instrucción anterior al siguiente líder en la tabla de código. A continuación describiremos a mayor profundidad los tres algoritmos que utilizaremos para optimizar código Pas++.

Tesis publicada con autorización del autor
No olvide citar esta tesis

3.7.1. Ejecución en tiempo de compilación

Según [MAM2006], esta optimización se encarga de realizar la ejecución de diversas operaciones aritméticas (+, -, * y /) y conversiones de tipo en el momento de la compilación. Al realizarse la ejecución es necesario guardar el resultado en una subtabla T con pares (ID, VALOR). El algoritmo se describe a continuación (véase en el anexo 18 un ejemplo de esta optimización aplicada a un programa en Pas++):

- Si la operación de código intermedio tiene la forma (op, op1, op2, res), donde op1 es un identificador y (op1,v1) está en la tabla T, sustituimos en la cuádrupla op1 por v1.
- Si la operación de código intermedio tiene la forma (op, op1, op2, res), donde op2 es un identificador y (op2,v2) está en la tabla T, sustituimos en la cuádrupla op2 por v2.
- Si la operación de código intermedio tiene la forma (op, v1, v2, res), donde v1 y v2 son valores constantes o nulos, eliminamos la operación de la tabla de código, eliminamos de T el par (res, v), si existe, y añadimos a T el par (res, v1 op v2), a menos que v1 op v2 produzca un error, en cuyo caso daremos un aviso y dejaremos la operación como está.
- Si la operación de código intermedio tiene la forma (=, v1, , res), eliminamos de T el par (res, v), si existe. Si v1 es un valor constante, añadimos a T el par (res, v1).

A continuación se muestra un ejemplo de esta optimización:

Nº	Instrucción original	Operaciones de código intermedio	Optimización
1	I = 2 + 3	(+, 2, 3, t1)	Elim, T = {(t1,5)}
		(=, t1, , i)	Sust por (=,5,,i), T = {(t1,5),(i,5)}
2	i = 4	(=, 4, , i)	T = {(t1,5),(i,4)}
3	f = i + 2.5	(CIF, i, , t2)	Sust por (CIF,4,,t2), Elim, T = {(t1,5),(i,4),(t2,4.0)}
		(+, t2, 2.5, t3)	Sust por (+,4.0,2.5,t3) Elim,T= {(t1,5),(i,4),(t2,4.0),(t3,6.5)}
		(=, t3, , f)	Sust por (=,6.5,,f)

Figura 3.18 Ejemplo de optimización por ejecución en tiempo de compilación

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Es importante mencionar que se debe reinicializar el valor de las variables (inicializar la tabla T, total o parcialmente). Esto puede ocurrir si aparece:

- Una llamada a un método, si se pasan variables por referencia o variables globales.
- Una instrucción de lectura.

3.7.2. Propagación de copias

En el presente trabajo, se implementó la propagación de copias como medio de optimización de código. La propagación de copias se realiza a nivel de bloques básicos.

Como se mencionó anteriormente, lo que se intenta es utilizar la copia de una variable (en vez de la misma) en las instrucciones que continúan el programa y en las cuáles realizar este reemplazo no modifica el resultado de las operaciones afectadas.

A continuación se muestra un ejemplo de lo que realiza esta optimización:

Nº	Instrucción original	Instrucción actualizada
1	b = a	b = a
2	c = b + 1	c = a + 1
3	d = b	d = a
4	b = d + c	b = a + c
5	b = d	b = a

Figura 3.19 Ejemplo de optimización por propagación de copias

En el ejemplo anterior se observa cómo se propaga la asignación “b=a” a lo largo de las instrucciones que la suceden (2,3) hasta que es invalidada en la línea 4. Asimismo la asignación “d=b” en la línea 3 se propaga a la línea 5.

El algoritmo de propagación de copias es el siguiente (véase en el anexo 19 esta optimización aplicada a un programa en Pas++):

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

```
Propagacion_copias
Inicio
Para cada Bloque Básico B hacer
        Limpiar_Tabla(tabla)
        Para cada instrucción i en B hacer
                Si i es de la forma “res = op1 operación op2” entonces
                        opd1 = Reemplazar (opd1, tabla)
                        opd2 = Reemplazar (opd2, tabla)
                Caso Contrario Si i es de la forma “res = var” entonces
                        var = Reemplazar(var, tabla)
                Fin Si
                Si i es de la forma “res = var” entonces
                        Remover(res,tabla)
                        Insertar( (res,var) , tabla)
                Fin Si
        Fin Para
Fin Para
Fin

Reemplazar (x, tabla)
Inicio
        Para cada entrada (a, b) de tabla hacer
                Si a = x entonces
                        Reemplazar <- b
                Fin Si
        Fin Para
        Reemplazar <- x
Fin
```

Figura 3.20 Algortimo de propagación de copias

3.7.3. Eliminación de las subexpresiones comunes

El tercer algoritmo de optimización implementado es el de eliminación de subexpresiones comunes. Como en el caso anterior el algoritmo trabaja a nivel de bloques básicos de código.

La optimización consiste en guardar en una tabla operaciones del tipo “a = b op c” de tal manera que si más adelante en el código se encuentra con la expresión “b op c”, y las variables b y c no han cambiado su valor desde la asignación “a = b op c”, la expresión “b op c” será reemplazada por “a”.

En este trabajo se ha implementado la optimización para que trabaje con tipos primitivos de datos (enteros y flotantes) y con las operaciones aritméticas +, -, * y /. Además, se ha tomado en cuenta la conmutatividad de las operaciones + y * para ampliar el rango de coincidencias.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

A continuación se presenta un ejemplo de la eliminación de subexpresiones comunes:

Nº	Instrucción original	Instrucción actualizada
1	c = a + b	c = a + b
2	d = m * n	d = m * n
3	h = b + a	h = c
4	k = m * n	k = d
5	a = -b	a = -b

Figura 3.21 Ejemplo de optimización por subexpresiones comunes

Como se puede observar, las instrucciones 3 y 4 fueron actualizadas por asignaciones, debido a que en ambos casos existía una variable con el mismo valor. Nótese como se aplica la conmutatividad en la substitución de la instrucción 3. A continuación se presenta el algoritmo de eliminación de subexpresiones comunes (véase en el anexo 17 un ejemplo de esta optimización aplicada a un programa en Pas++):

```
Eliminacion_Subexp_comunes
Inicio
        Para cada Bloque Básico B hacer
                Limpiar_Tabla(tabla_expr)
                Para cada instrucción i en B hacer
                        Si i es de la forma "res = op1 operación op2" entonces
                                Cambio  <-  Encontrar_op(tabla_exp,op1,  op2,
operación)
                                Si cambio >= 0 entonces
                                        Cambiar_instruccion(i,"res=cambio")
                                Fin Si
                        Fin Si

                        Si i es de la forma "res = op1 operación op2" O
                        i es de la forma "res = op1" entonces
                                Eliminar_Referencia(tabla_expr,res)
                        Fin Si
                Fin Para
        Fin Para
Fin

Encontrar_op(tabla_exp,op1, op2, operación)
Inicio
        Cambio <- -1
        Para cada entrada e de tabla_exp hacer
                Si operación = '+' O operación = '*' entonces
                        Si (e.op1 = op1 Y e.op2 = op2) O
                        (e.op1 = op2 Y e.op2 = op1) entonces
                                Cambio <- e.var
                        Fin Si
```


Tesis publicada con autorización del autor
No olvide citar esta tesis

```
Caso contrario
                              Si (e.op1 = op1 Y e.op2 = op2) entonces
                                        Cambio <- e.var
                              Fin Si
                    Fin Si
          Fin Para
          Encontrar_op <- Cambio
Fin
```

Figura 3.22 Algoritmo de subexpresiones comunes

Tesis publicada con autorización del autor
No olvide citar esta tesis

3.8. Diseño del entorno de desarrollo

El entorno de desarrollo nos debe permitir mostrar todas las funcionalidades del intérprete, además de hacer visibles las estructuras internas. El entorno de desarrollo se muestra a continuación:

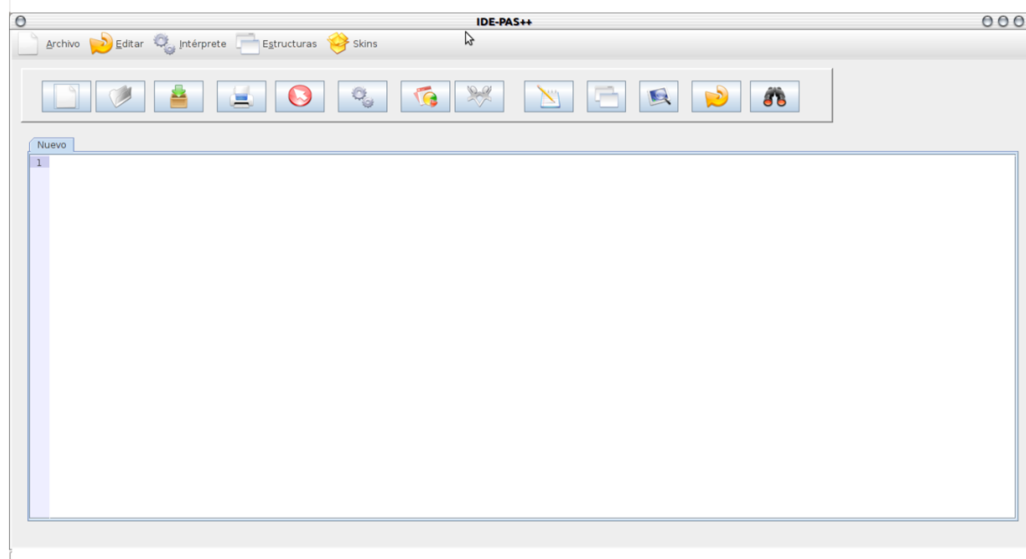

Figura 3.23 El entorno de desarrollo

Se tiene en cuenta las siguientes funcionalidades para el entorno de desarrollo:

- **El editor de texto:** el cual nos permite escribir programas fuente, además de copiar, cortar y pegar texto.

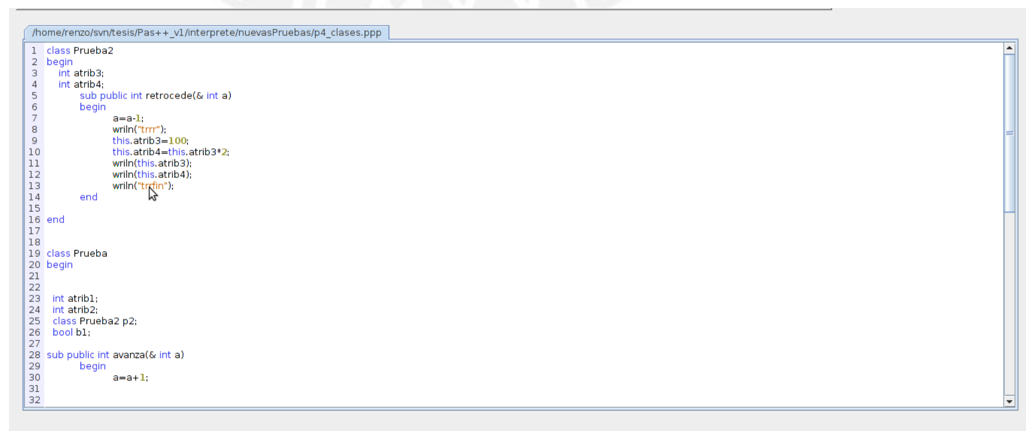

Figura 3.24 Editor de texto

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **Funciones básicas:** como nuevo, abrir, guardar, guardar como e imprimir.

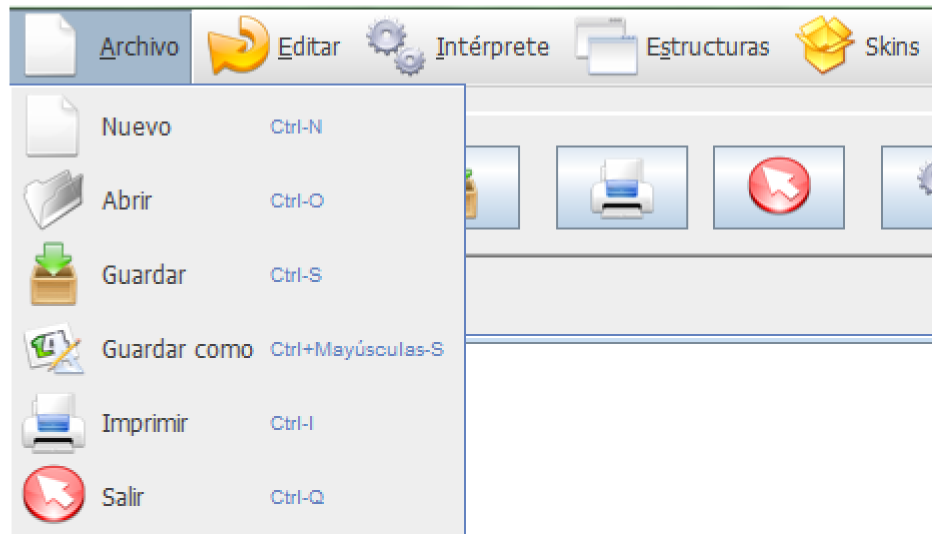

Figura 3.25 Funciones básicas

- **La opción de interpretación:** la cual nos permite compilar y ejecutar programas escritos en Pas++.

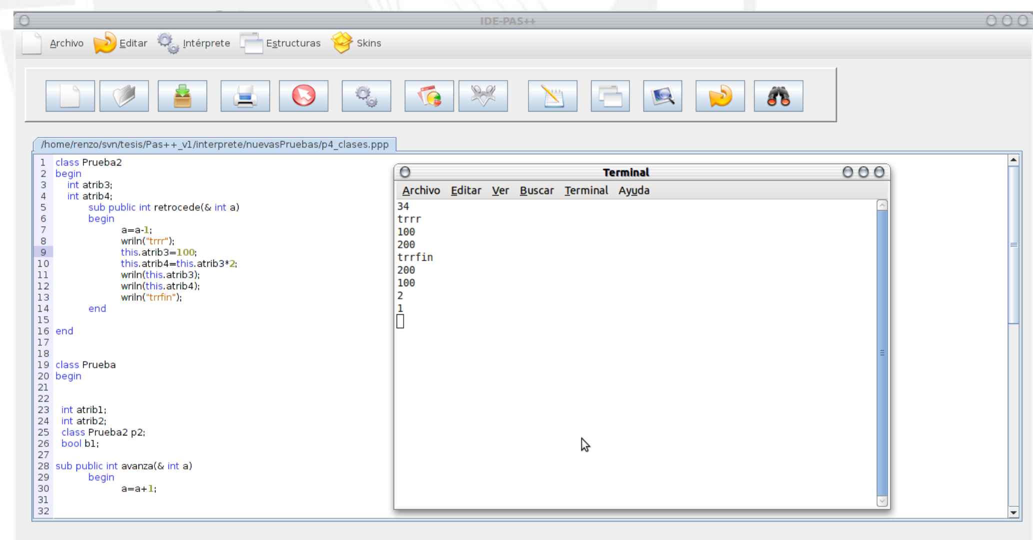

Figura 3.26 Ejecución de un programa en el IDE

- **La opción de optimización**: la cual nos permite optimizar el código intermedio de nuestro programa, para posteriormente visualizarlo en la tabla de código generada.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

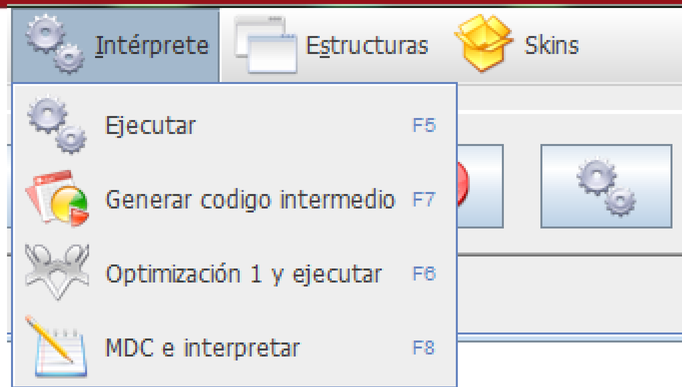

Figura 3.27 Funciones del intérprete

- **La opción de modificación dinámica de código:** la cual nos permite usar las opciones "insert", "remove" y "modify"; para insertar, remover y modificar código del programa respectivamente.

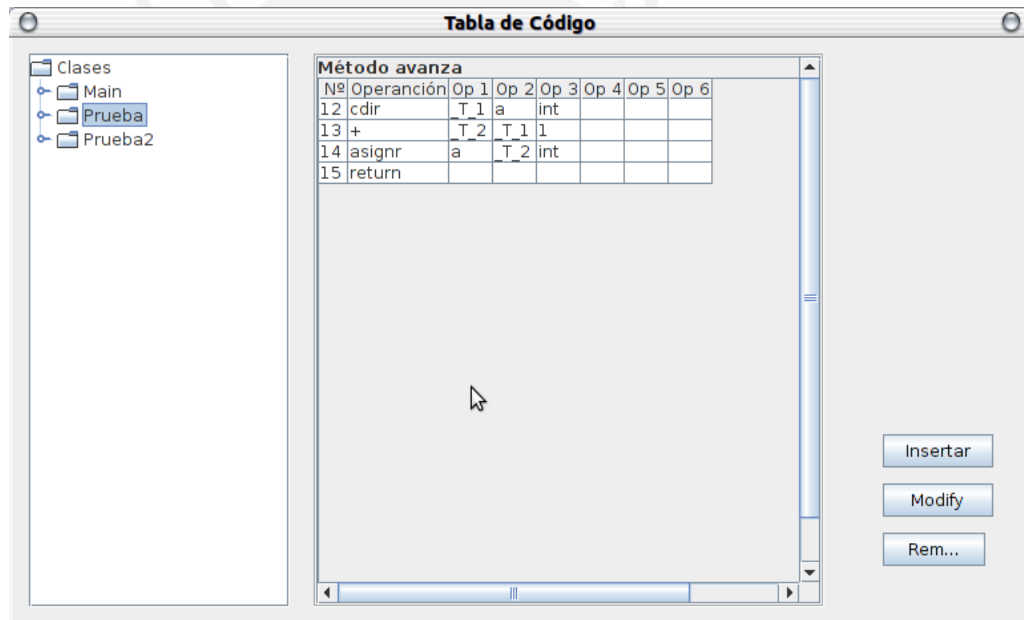

Figura 3.28 Modificación dinámica de código

Tesis publicada con autorización del autor
No olvide citar esta tesis

- **La opción de mostrar la tabla de código:** la cual nos permite observar dicha tabla, es decir, una lista operaciones de código intermedio. Se mostrará por cada uno: el nombre de la operación (por ejemplo, TREAD), y los argumentos necesarios para ejecutar dicha operación.

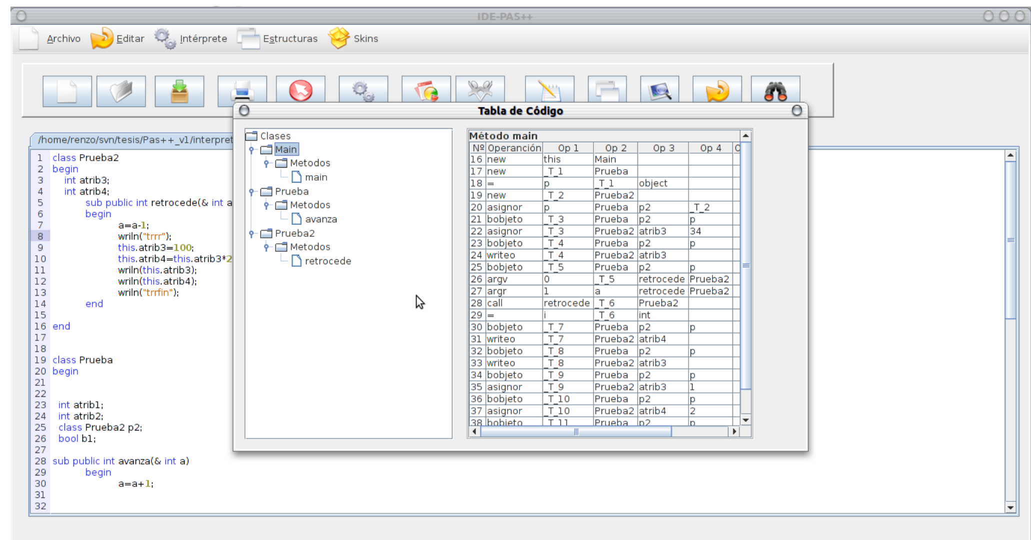

Figura 3.29 Tabla de código

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **La opción de mostrar la tabla de símbolos:** la cual nos permite observar dicha tabla, es decir, una lista de símbolos por cada método. Se toma en cuenta para cada símbolo: el nombre, el tipo, la clase (si fuese un objeto) y la dirección relativa (offset).

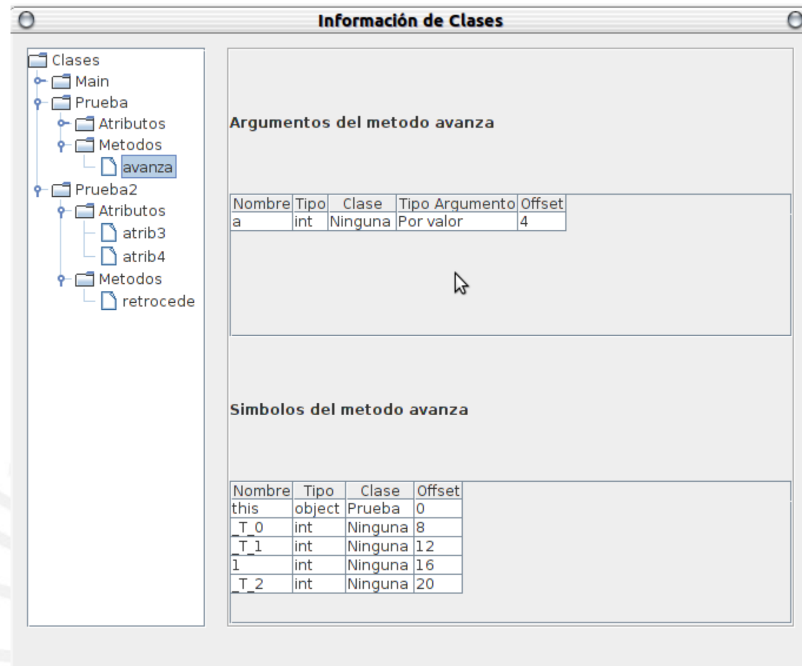

Figura 3.30 Tabla de símbolos

- **La opción de mostrar la tabla de clases:** la cual nos permite observar la tabla de clases declaradas, es decir, una lista de métodos y símbolos (tablas de métodos y símbolos).

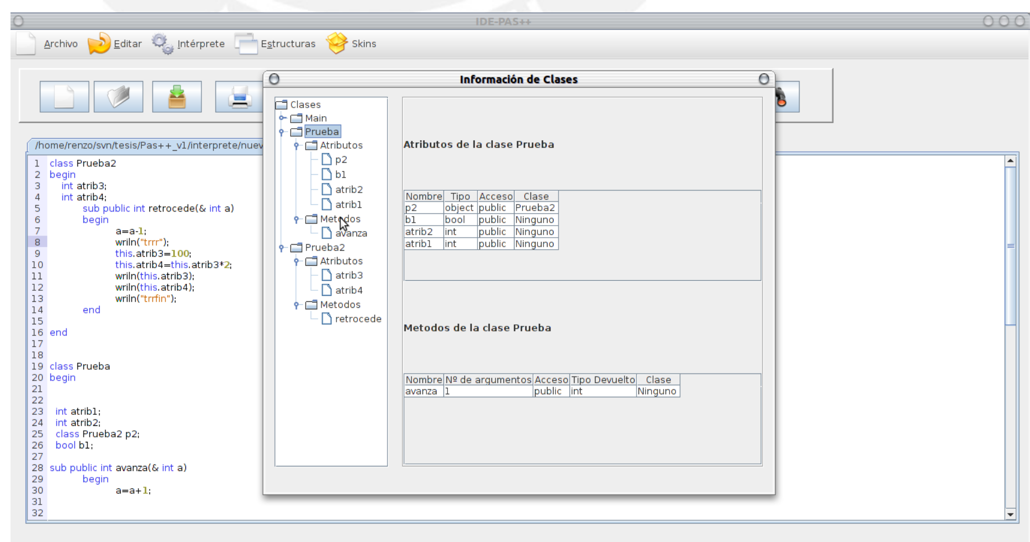

Figura 3.31 Tabla de clases

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.9. Ambiente de desarrollo

Para el desarrollo del proyecto de tesis se ha utilizado un conjunto de herramientas que facilitan el desarrollo del intérprete y el entorno de desarrollo. A continuación se detallará acerca de las herramientas utilizadas en desarrollo del proyecto.

3.9.1. Flex

Esta herramienta permite construir un analizador léxico a partir de las reglas (patrones) definidos para los lexemas. Estas reglas fueron definidas en el diseño del intérprete (sección 3.1) y se han especificado utilizando un conjunto extendido de las expresiones regulares. El analizador léxico generado se encarga de realizar el particionamiento de un texto ingresado por entrada estándar o un archivo de texto en patrones léxicos especificados por el usuario.

Al ejecutar Flex teniendo como entrada un archivo fuente (con extensión *.yy) se obtiene un archivo de código fuente en C que posee la función yylex. Esta función es la encargada (al ser llamada) de devolver el siguiente token perteneciente al texto ingresado de acuerdo a los patrones definidos anteriormente.

Figura 3.32 Flujo en Flex

Los archivos fuente de Flex tienen una estructura propia, donde se pueden distinguir 3 secciones diferentes (separadas por un doble '%'). En la primera sección se introducen las definiciones regulares, la segunda sección sirve para definir las reglas que permitirán reconocer a los patrones y las acciones que se realizarán al reconocer a un patrón, y en la tercera sección se puede introducir código en C.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

```
/*Sección 1*/
%{
#include <stdio.h>
...
%}
Letra [a-zA-Z]
Digito [0-9]
%%
/*Sección 2*/
{letra} ({letra}|{digito})* return ID;
%%
/*Sección 3*/
Main(){
}
```

Figura 3.33 Secciones de Flex

3.9.2. Bison/Yacc:

Esta herramienta sirve para realizar una descripción de la estructura sintáctica de los lenguajes de programación. Recibe como entrada las reglas que conforman la gramática del lenguaje y genera el analizador sintáctico

Figura 3.34 Flujo en Bison

En la especificación de cada regla Bison se permite la inserción de bloques de código fuente (en general en lenguaje C) llamados acciones, las cuales se ejecutan al identificar el símbolo no terminal que conforma la regla.

Una vez ejecutado el Yacc se obtiene un archivo fuente en C (o en el lenguaje que sea soportado) que principalmente contiene dos funciones.

- **yyparse:** esta función realiza el análisis sintáctico de los programas escritos en el lenguaje que se especificó en la gramática.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **yylex:** esta función se encarga de partir el programa ingresado en unidades léxicas elementales (tokens) y se los entrega al yyparse para que él determine la regla que le corresponde. Se puede utilizar la salida que se obtiene al usar Flex para la generación de esta función.

Los archivos fuente en Yacc/Bison tienen una estructura similar a la de Flex, tienen 3 secciones: en la primera se realiza la definición de tokens, en la segunda la definición de las reglas (gramática) y en la tercera se inserta código que se transcribirá al archivo de salida.

```
/*Sección 1: Definición de Tokens */
%token X Y Z
%%
/*Sección 2: Reglas de Gramática */
a : X Y b
  | Y X c ;
b : Z b
  | ;
%%
/*Sección 3: Código en C*/
Main(){
}
```

Figura 3.35 Secciones de Bison

3.9.3. Netbeans:

Se utilizará esta herramienta para la implementación del entorno de desarrollo del lenguaje Pas++. El entorno de desarrollo se desarrollará utilizando el lenguaje de programación Java que posee librerías que facilitan el desarrollo de la interfaz gráfica.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

4. Experimentación numérica

En este apartado se describirá la realización de la experimentación numérica. Se explicarán los pasos que se deben seguir para la ejecución del experimento, se explicarán los resultados obtenidos después de la ejecución y, por último, se analizarán y validarán los resultados con respecto a las hipótesis planteadas.

4.1. Introducción a la experimentación numérica

La primera parte de la experimentación numérica se ejecutó luego de haber implementado tanto el intérprete para el lenguaje Pas++y el optimizador de código, y tener la certeza de que funcionan correctamente. Luego de la ejecución de esta parte de la experimentación, se procede a comparar los resultados de ejecutar un programa con código optimizado y otro simplemente interpretado, para lo cual básicamente se compararán los tiempos promedios de ejecución utilizada de dos muestras de datos, una de ellas utilizando el optimizador y la otra no.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

La segunda parte de la experimentación numérica consiste en comparar los tiempos que se tardó en implementar completamente el intérprete, el entorno de desarrollo y el optimizador; como se explicó en el capítulo dos, estos tres componentes se llevaron a cabo siguiendo un distinto ciclo de vida. Para el intérprete se siguió el ciclo de vida en cascada, para el entorno de desarrollo se utilizó un ciclo de vida incremental y para el optimizador un ciclo de vida evolutivo.

Para el análisis de los resultados se utilizó el estadístico T-Student que comprende un análisis con la distribución normal de los datos obtenidos.

4.2. Primera parte de la experimentación numérica

4.2.1. Definición del problema de investigación

El problema que se busca resolver a través de esta parte de la experimentación numérica es demostrar que el optimizador de código que se ha construido está funcionando correctamente, es decir, produce una notoria ventaja en el rendimiento de ejecución de un programa.

4.2.2. Variables de respuesta

La siguiente variable de respuesta determinará si existe una mejora del rendimiento de un programa al utilizar el optimizador:

- **Tiempo de ejecución:** es el tiempo que demora el procesador en ejecutar un programa escrito en Pas++.

4.2.3. Hipótesis

Sean:

- Z: un programa escrito en el lenguaje Pas++.
- X_1: media del tiempo de ejecución de Z luego de ser interpretado sin usar optimización de código.
- X_2: media del tiempo de ejecución de Z luego de ser interpretado usando optimización de código.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- H_0: $X_1 \leq X_2$: *"El tiempo de ejecución de un programa interpretado con optimización de código no es menor al interpretado sin usar esta opción".*

- H_1: $X_1 > X_2$: *"El tiempo de ejecución de un programa interpretado sin usar optimización de código es mayor al interpretado usando esta opción".*

Aquí se están comparando las medias del tiempo de ejecución de un programa escrito en Pas++ para analizar el funcionamiento del optimizador de código.

4.2.4. Consideraciones de la ejecución

- Se ejecutó 31 veces un programa escrito en Pas++ utilizando el optimizador y el mismo programa se volvió a ejecutar 31 veces sin el uso del mismo.

- Las pruebas se realizaron sobre un mismo computador, disminuyendo así factores externos, tales como: diferencias en el procesador, memoria RAM, disco duro, memoria caché, etc.

- Se tomaron dos muestras aleatorias de:

  Tamaño n1: representa el tamaño de la muestra de los resultados obtenidos usando el optimizador.

  Tamaño n2: representa el tamaño de la muestra de los resultados obtenidos sin usar el optimizador.

- Se consideró para el estadístico t-student: grados de libertad n1+n2-2

- Para nuestro experimento n1=n2=n=31, donde n representa el tamaño de la muestra de resultados obtenidos al ejecutar el programa con el optimizador y sin el mismo.

4.2.5. Determinación del estadístico: T-Student

Se considera un valor de $\alpha = 5\%$ que es el nivel de riesgo o significación, es decir, que se tiene una probabilidad de 0.05 de rechazar H_0 siendo ésta cierta (hay otros valores que también suelen usarse, como 1% o 10%).

Tesis publicada con autorización del autor
No olvide citar esta tesis

Sean:

S_1: la varianza muestral del tiempo de ejecución de Z usando el optimizador.
S_2: la varianza muestral de Z sin usar el optimizador
g.l.: n1+n2-2

T-Student:

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{{S_1}^2}{n_1} - \frac{{S_2}^2}{n_2}}}$$

Teniendo en cuenta el nivel de significación definido anteriormente se rechazará H_0 si el estadístico **t >2.00**. Este valor de 2.00 se extrae de la tabla de t-student con 60 grados de libertad y una probabilidad de 0.975

4.2.6. Resultados de la experimentación

Para la realización de la experimentación se utilizó un programa que realiza mil iteraciones sobre un conjunto de operaciones que presenta características optimizables (eliminación de subexpresiones comunes). Se tomaron 62 muestras de las cuales la mitad fueron ejecuciones sin usar el optimizador y la otra mitad se realizaron utilizando el optimizador. Cabe resaltar que la unidad de medición para el tiempo es microsegundos.

A continuación se muestra una tabla con las 62 ejecuciones realizadas, de las cuales la mitad fue tomada con optimización y la otra mitad sin optimización. Además, se muestra el código intermedio generado del programa con y sin optimización.

Ejecución	Sin Optimización	Con Optimización
1	7179	6136
2	7165	6016
3	5599	6315
4	3936	3370
5	3911	3388
6	4995	3576

Tesis publicada con autorización del autor
No olvide citar esta tesis

7	6005	4513
8	7250	5674
9	7137	6052
10	7188	6020
11	7164	6001
12	7275	6093
13	7163	6270
14	7159	6117
15	7205	6007
16	7246	6007
17	7174	6062
18	7687	6108
19	7176	6019
20	7418	6019
21	7290	6070
22	7158	6081
23	7118	6492
24	7235	6043
25	7163	5893
26	7183	6086
27	7258	6071
28	7337	6084
29	7522	5994
30	7165	6017
31	7163	6059

Figura 4.1 Resultados de tiempos de la experimentación numérica (en microsegundos)

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Nº	Operación	Op 1	Op 2	Op 3	Op 4	Op 5	Op 6
0	new	this	Main				
1	=	i	0	int			
2	=	limit	10000	int			
3	<=	_T_1	i	int			
4	jumpf	_T_1	22				
5	=	a	1	int			
6	=	b	2	int			
7	/	_T_2	a	b			
8	=	c	_T_2	int			
9	=	_T_3	_T_2	int			
10	=	d	_T_3	int			
11	=	_T_4	_T_2	int			
12	=	e	_T_4	int			
13	=	_T_5	_T_2	int			
14	=	f	_T_5	int			
15	=	_T_6	_T_2	int			
16	=	h	_T_6	int			
17	=	_T_7	_T_2	int			
18	=	j	_T_7	int			
19	+	_T_8	i	1			
20	=	i	_T_8	int			
21	jump	3					
22	return						

Nº	Operación	Op 1	Op 2	Op 3	Op 4	Op 5	Op 6
0	new	this	Main				
1	=	i	0	int			
2	=	limit	10000	int			
3	<=	_T_1	i	int			
4	jumpf	_T_1	22				
5	=	a	1	int			
6	=	b	2	int			
7	/	_T_2	a	b			
8	=	c	_T_2	int			
9	/	_T_3	a	b			
10	=	d	_T_3	int			
11	/	_T_4	a	b			
12	=	e	_T_4	int			
13	/	_T_5	a	b			
14	=	f	_T_5	int			
15	/	_T_6	a	b			
16	=	h	_T_6	int			
17	/	_T_7	a	b			
18	=	j	_T_7	int			
19	+	_T_8	i	1			
20	=	i	_T_8	int			
21	jump	3					
22	return						

Figura 4.2 Tabla de código con optimización (izquierda) y sin optimización (derecha)

Se utilizó el software estadístico SPSS para realizar la prueba T-Student sobre los resultados obtenidos. A continuación se muestra las tablas extraídas luego del análisis.

Group Statistics

	Tipo	N	Mean	Std. Deviation	Std. Error Mean
SO	NORMAL	31	***6858,8387***	951,67509	170,92589
	OPTIMIZADO	31	***5763,0000***	830,94717	149,24252

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means	
		F	Sig.	t	df
SO	Equal variances assumed	***,353***	,555	***4,829***	60

Tesis publicada con autorización del autor
No olvide citar esta tesis

Los resultados de la prueba de Levene para corroborar la asunción que las varianzas de ambas muestras son iguales nos arroja un valor de **0.353 < F(0.05,1,60) = 4.00**, con lo cual se puede afirmar con un nivel de significación de 5.00% que las varianzas no son diferentes.

Como se puede observar de la experimentación se obtuvo **t = 4.829 > 2.00** con lo cual se rechaza H_0. Por lo tanto, se puede afirmar que un programa sin optimización tiene un tiempo de ejecución mayor a uno optimizado con un nivel de significación de 5.00%. Asimismo, comparando las medias obtenidas se tiene que **el tiempo de ejecución mejoró en un 16% aproximadamente.**

4.3. Segunda parte de la experimentación numérica

4.3.1. Definición del problema de investigación

El problema que se busca resolver a través de esta parte de la experimentación numérica es conocer qué ciclo de vida se adecúa mejor al desarrollo de un componente del intérprete.

4.3.2. Variables de respuesta

Las siguientes variables de respuesta determinarán cuál de los ciclos de vida es el mejor.

- **Tiempo de realización:** es el tiempo que se demora en realizar un componente del intérprete

4.3.3. Hipótesis de la Experimentación numérica

Sean:

- X_1: La media del tiempo de desarrollo del intérprete.
- X_2: La media del tiempo de desarrollo del entorno de desarrollo.
- X_3: La media del tiempo de desarrollo del optimizador.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Hipótesis:

- H_0: $X_1 \leq X_2$: *"El ciclo de vida en cascada se adapta mejor que el ciclo de vida incremental para un proyecto de este tipo".*

- H_1: $X_1 > X_2$: *"El ciclo de vida incremental se adapta mejor que el ciclo de vida en cascada para un proyecto de este tipo".*

Hipótesis:

- H_0: $X_3 \leq X_2$: *"El ciclo de vida evolutivo se adapta mejor que el ciclo de vida incremental para un proyecto de este tipo".*

- H_1: $X_3 > X_2$: *"El ciclo de vida incremental se adapta mejor que el ciclo de vida evolutivo para un proyecto de este tipo".*

Hipótesis:

- H_0: $X_1 \leq X_3$: *"El ciclo de vida en cascada se adapta mejor que el ciclo de vida evolutivo para un proyecto de este tipo".*

- H_1: $X_1 > X_3$: *"El ciclo de vida evolutivo se adapta mejor que el ciclo de vida en cascada para un proyecto de este tipo".*

4.3.4. Consideraciones de la ejecución

- Se evaluarán los tiempos de desarrollo de cada componente en 5 fases.

- Se tomarán dos muestras aleatorias de:

  Tamaño n1: representa el tamaño de la muestra de los resultados obtenidos para el intérprete.

  Tamaño n2: representa el tamaño de la muestra de los resultados obtenidos para el entorno de desarrollo.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Tamaño n3: representa el tamaño de la muestra de los resultados obtenidos para el optimizador.

- Se considerará para el estadístico t-student: grados de libertad n1+n2-2

- Para nuestro experimento n1=n2=n3=n=5, donde n representa el tamaño de la muestra de resultados obtenidos al desarrollar un componente del intérprete.

4.3.5. Determinación del estadístico: T-Student

Se considerará un valor de $\alpha = 5\%$ que es el nivel de riesgo o significación, es decir, que se tendrá una probabilidad de 0.05 de rechazar H_0 siendo ésta cierta.

Teniendo en cuenta el nivel de significación definido anteriormente se rechazará H_0 si el estadístico **t >2.30**. Este valor de 2.30 se extrae de la tabla de t-student con 8 grados de libertad y una probabilidad de 0.975.

4.3.6. Resultados de la experimentación

Para realizar la experimentación se tomaron cinco muestras (fases) de cada componente. A continuación, se indican cuáles fueron las 5 fases en las que se dividió cada componente:

- **Intérprete:**

  - ✓ Implementación de reglas del lenguaje estructurado.
  - ✓ Implementación de clases, métodos, atributos, asignación de memoria dinámica.
  - ✓ Implementación de constructores, herencia, modificadores de acceso.
  - ✓ Implementación de “insert” en MDC.
  - ✓ Implementación de “remove” y “modify” en MDC.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- **Entorno de desarrollo:**

  - ✓ Implementación de editor de texto y herramientas de edición.
  - ✓ Visualización de tablas de código, de símbolo y clases.
  - ✓ Integración con el intérprete.
  - ✓ Integración con la modificación dinámica de código.
  - ✓ Integración con el optimizador.

- **Optimizador**

  - ✓ Análisis de control de flujo.
  - ✓ Eliminación de subexpresiones comunes.
  - ✓ Propagación de copias.
  - ✓ Ejecución en tiempo de compilación.
  - ✓ Integración con el intérprete.

A continuación, se muestran las fases en los que se dividió la implementación del intérprete:

Fase	Duración (días)
Implementación de reglas del lenguaje estructurado.	9
Implementación de clases, métodos, atributos, asignación de memoria dinámica	8
Implementación de constructores, herencia, modificadores de acceso	6
Implementación de “insert” en MDC	7
Implementación de “remove” y “modify” en MDC	6

Asimismo, las fases en las que se dividió el desarrollo del IDE:

Fase	Duración (días)
Implementación de editor de texto y herramientas de edición.	7
Visualización de tablas de código, de símbolo y clases	5

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Integración con el intérprete	3
Integración con la modificación dinámica de código	4
Integración con el optimizador.	3

Finalmente, las fases en las que se dividió el desarrollo del optimizador:

Fase	Duración (días)
Análisis de control de flujo	5
Eliminación de subexpresiones comunes	6
Propagación de copias	6
Ejecución en tiempo de compilación	6
Integración con el intérprete	5

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

A continuación, se muestra los diagramas Gantt de las fases por componente:

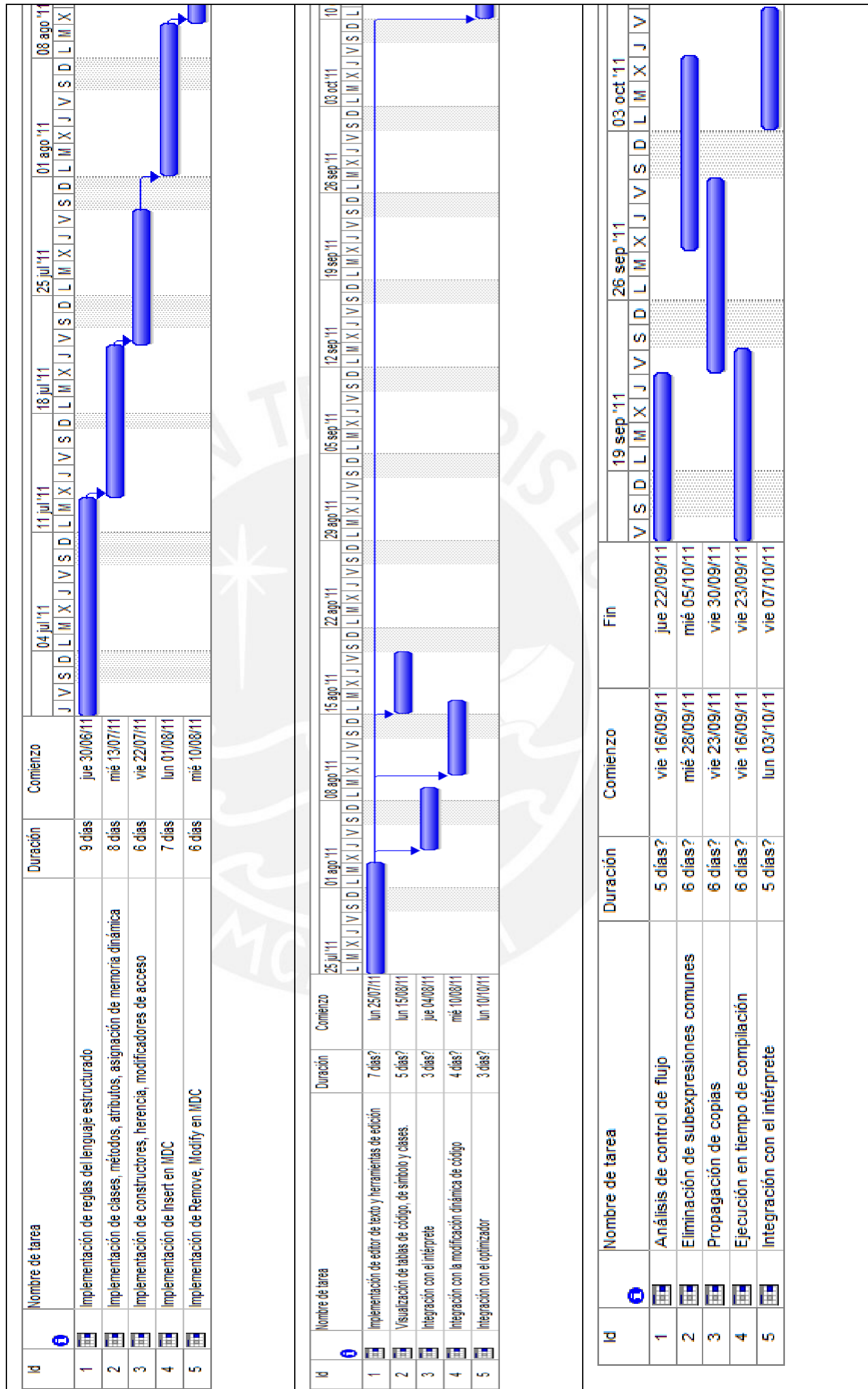

Figura 4.3 **Diagramas de Gantt de los proyectos desarrollados (de izquierda a derecha: intérprete, IDE y optimizador)**

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Haciendo uso del software SPSS se ingresó los datos de las muestras tomadas para el caso del proyecto con ciclo de vida en cascada (C) y el proyecto con ciclo de vida incremental (I). Los resultados obtenidos se muestran a continuación:

Group Statistics

	Proyecto	N	Mean	Std. Deviation	Std. Error Mean
Dias	C	5	***7,20***	1,304	,583
	I	5	***4,00***	1,225	,548

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means	
		F	Sig.	T	df
Dias	Equal variances assumed	***,275***	,614	***4,000***	8

Los resultados de la prueba de Levene para corroborar la asunción que las varianzas de ambas muestras son iguales nos arroja un valor de **0.275 < F(0.05,1,8) = 5.32**, con lo cual se puede afirmar con un nivel de significación de 5.00% que las varianzas no son diferentes.

Como se puede observar se obtuvo un valor **t = 4.00 > 2.30**, con lo cual se puede afirmar que utilizar un **ciclo de desarrollo incremental** para este tipo de proyectos (Compiladores) conlleva a obtener **un menor tiempo de implementación** que si se utilizara un **ciclo de vida en cascada** con un nivel de significación de 5.00%.

Asimismo, se procede a comparar las muestras obtenidas durante el desarrollo del proyecto con ciclo de vida evolutivo (E) y el proyecto con ciclo de vida incremental (I). A continuación se muestra los resultados de la experimentación:

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Group Statistics

	Proyecto	N	Mean	Std. Deviation	Std. Error Mean
Dias	E	5	***5,60***	,548	,245
	I	5	***4,00***	1,225	,548

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means	
		F	Sig.	T	df
Dias	Equal variances assumed	***,719***	,421	***2,667***	8

Los resultados de la prueba de Levene para corroborar la asunción que las varianzas de ambas muestras son iguales nos arroja un valor de **0.719 < F(0.05,1,8) = 5.32**, con lo cual se puede afirmar con un nivel de significación de 5.00% que las varianzas no son diferentes.

Los resultados obtenidos muestran que se obtuvo un valor de **t = 2.667 > 2.30** con lo cual se puede afirmar que utilizar un **ciclo de desarrollo incremental** para este tipo de proyectos (Compiladores) conlleva a obtener un **menor tiempo de implementación** que si se utilizara un **ciclo de vida evolutivo** con un nivel de significación de 5.00%.

Por último, se comparan las muestras obtenidas de los proyectos desarrollados con ciclo de vida en cascada (C) y con ciclo de vida evolutivo (E).

Group Statistics

	Proyecto	N	Mean	Std. Deviation	Std. Error Mean
Dias	C	5	***7,20***	1,304	,583
	E	5	***5,60***	,548	,245

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

ET LUX IN TENEBRIS LUCET MCMXVII PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means	
		F	Sig.	T	Df
Dias	Equal variances assumed	***4,356***	,070	***2,530***	8

Los resultados de la prueba de Levene para corroborar la asunción que las varianzas de ambas muestras son iguales nos arroja un valor de **4.356 < F(0.05,1,60) = 5.32**, con lo cual se puede afirmar con un nivel de significación de 5.00% que las varianzas no son diferentes.

Como se puede observar se obtuvo un valor **t = 2.530 > 2.30**, con lo cual se puede afirmar que utilizar un **ciclo de desarrollo evolutivo** para este tipo de proyectos (Compiladores) conlleva a obtener un **menor tiempo de implementación** que si se utilizara un **ciclo de vida en cascada** con un nivel de significación de 5.00%.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

5. Observaciones, conclusiones y recomendaciones

En este último capítulo se dan a conocer las observaciones obtenidas a lo largo del desarrollo del proyecto y las conclusiones a las que hemos llegado. Es parte también del análisis formular nuevas preguntas o cuestiones afines que han surgido con esta tesis a fin de plantear una serie de recomendaciones para posibles trabajos posteriores sobre este tema.

5.1. Observaciones

Las observaciones más importantes a tomar en cuenta durante el desarrollo del proyecto son las siguientes.

- En el desarrollo de las características de POO, en especial en la implementación de la herencia, el intérprete internamente realiza un renombramiento de un método heredado cuando existe una repetición en el nombre con los de la clase hija. La regla que se utilizó para la generación del nuevo nombre es <nombre_clase_padre>_<nombre_método>.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- Con respecto al intérprete y a la declaración de atributos, sólo se admite que un atributo tipo objeto sea de una clase definida anteriormente. Esto se decidió hizo así para evitar la complejidad de referencias cruzadas que queda fuera del alcance de este proyecto.

- Por motivos prácticos se tomó la decisión de que el programa arranque en un método “main” de una clase “Main”. Así podemos evitar situaciones donde hayan varios métodos main en diferentes clases y de esta manera el programa se pueda ejecutar de todas formas.

- Con respecto a la tabla de símbolos se decidió implementar una tabla de símbolos por cada método, así se puede realizar una búsqueda mucho más eficiente de un determinado símbolo, y resolver el problema de repetición de nombres.

- En la estructura “tmetodo” se incluye dos campos que indican el número de la instrucción de inicio y fin de ese método en la tabla de código. Esto nos permite tener la información necesaria cuando se quiere mostrar el código de cada método por separado.

- Un punto importante a resaltar es el manejo de atributos, tanto almacenamiento como uso; ya que cuando se trabaja con variables comunes se sabe su ubicación en memoria en tiempo de compilación, pero en este caso no. Lo que se hizo fue establecer que cada objeto es un índice a la tabla de objetos y en la entrada correspondiente en la tabla se guarda la ubicación de ese objeto en el heap; de esta manera conociendo un offset determinado se puede guardar o usar atributos de ese objeto.

- Un objeto puede ser atributo de otro objeto, entonces fue necesario manejar las producciones que permitan reconocer esta gramática y que se genere el código intermedio adecuado que permita buscar a un objeto en cada momento que se necesite hacerlo (código intermedio BBOJETO).

- Con respecto a la modificación dinámica de código, al realizar la inserción, modificación o eliminación de una instrucción, se observó

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

que era necesario realizar nuevamente el proceso de análisis sintáctico para verificar la validez del nuevo código intermedio. Por tal motivo, al realizarse la interpretación se realiza un análisis adicional a la instrucción añadida, se crean nuevas variables en tiempo de ejecución y se genera nuevo código intermedio que altera la tabla de código. Mediante el IDE se puede observar el cambio estructural que sufre el programa luego de alguna de estas operaciones. Además, utilizando el código intermedio, el IDE permite ingresar estas instrucciones en la tabla de código para observar los resultados en tiempo de ejecución.

- Al realizar la experimentación numérica de la optimización de código, se midió el tiempo de ejecución en microsegundos para obtener una medición más precisa y la diferencia en tiempos sea notoria. Además, se tomó en cuenta solo el tiempo propio de la ejecución descartando el tiempo que el optimizador toma para realizar el análisis y el cambio en el código intermedio.

- Asimismo, como se mencionó anteriormente, se ha implementado la propagación de copias y la eliminación de subexpresiones comunes en su versión para bloques básicos de código. Por esta razón, al realizar la experimentación numérica, se consideró un programa que presente características optimizables (en este caso subexpresiones comunes) dentro de un bloque básico.

- Además, al realizarse la optimización al código intermedio para obtener uno optimizado, se puede tomar ambos códigos (código intermedio sin optimizar y optimizado) para mostrar sus respectivas tablas en el IDE y observar las diferencias estructurales que realizó el proceso de optimización.

- En la recolección de tiempos para la realización del experimento acerca de los ciclos de vida de desarrollo, se ha tomado en cuenta los días que se necesitaron no solo en la implementación de las partes constituyentes de cada componente sino también en su diseño y la validación (pruebas) de los mismos.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

5.2. Conclusiones

Las principales conclusiones de esta investigación son listadas a continuación:

- El objetivo principal de la investigación es estudiar e implementar un intérprete con tres características: lenguaje orientado a objetos, mecanismos de modificación dinámica de código y opciones de optimización. De esta forma, se busca contribuir al aprendizaje e investigación de construcción de intérpretes o compiladores. Con el desarrollo del presente trabajo, se tiene la certeza que un proyecto de este tipo realmente permite aprender más acerca de estos temas, ya que se pueden plasmar los conocimientos teóricos en el desarrollo de un caso real. Por lo tanto, este trabajo es un caso de estudio donde se establecen un conjunto de pasos adecuados y buenas prácticas para que cualquier persona con conocimientos básicos e interesado en el tema pueda consolidarlos y realizar proyectos más avanzados.

- El intérprete desarrollado, a pesar de sólo incluir un subconjunto de las características de un lenguaje orientado a objetos, brinda a las personas interesadas en este tema una referencia de cómo se podría implementar un intérprete con todas las características propias de éste paradigma. De esta manera, se cumple el objetivo de ser un punto de partida para trabajos futuros.

- La tesis ha tomado como referencia trabajos desarrollados anteriormente, sin embargo, mediante el desarrollo de características como la programación orientada a objetos, optimización y modificación dinámica de código se ha logrado ir más allá de lo desarrollado por anteriores proyectos de fin de carrera y afines, por lo tanto, éste trabajo puede servir como base para que otros estudiantes realicen proyectos que mejoren el presente e implementen nuevas características.

- La modificación dinámica de código es una característica muy peculiar del intérprete que cambia el flujo normal de cualquier compilador, ya

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

que se vuelve a realizar el análisis sintáctico cuando ya se está ejecutando el programa. Esta característica tiene como principal uso el “debugging”, ya que se puede usar si se quiere modificar algo en el programa que se escribió pero ya se está ejecutando.

- En la implementación del optimizador se restringió el análisis a bloques básicos de código. Mediante la experimentación se logró comprobar que es posible reducir el tiempo de ejecución de un programa que presenta características optimizables. Sin embargo, las optimizaciones se aplicaron a un ámbito local (bloques básicos) y no a un programa completo. Por lo tanto, realizar un análisis de control de flujo de datos permitiría aplicar las optimizaciones a un ámbito global y lograr una reducción mayor en el tiempo de ejecución en ciertos casos.

- Se definió una tabla de símbolos por cada método en la implementación del intérprete y esto ayudó a simplificar el análisis de variables redefinidas, ya que no se tuvo que considerar el caso de variables con nombres iguales en métodos diferentes, lo que hubiese sido necesario si sólo se hubiera considerado una tabla de símbolos para todo el programa .

- Con respecto a las hipótesis de la experimentación numérica, por un lado, en el caso del optimizador se puede concluir con un nivel de significación del 5% que la implementación realizada mejora el tiempo de ejecución de programas que presentan características optimizables sobre bloques básicos.

- Por otro lado, en la comparación de ciclos de vida aplicados a un proyecto de este tipo, se puede afirmar con un nivel de significación del 5% que el ciclo de vida incremental permite un desarrollo más rápido del proyecto.

- Este trabajo es una muestra palpable de que en nuestro país se pueden realizar proyectos en el área de Ciencias de la Computación, particularmente en el área de Compiladores e Intérpretes. La tesis, por lo tanto, se suma a otros trabajos realizados anteriormente para

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

constituir un punto de partida para lograr que se tome conciencia de la importancia de realizar investigación en esta área.

5.3. Recomendaciones y trabajos futuros

En esta sección se tratará acerca de cuestiones que surgieron a lo largo del desarrollo del proyecto de tesis y que pueden ser tomadas como base para futuras investigaciones.

A continuación se presenta una lista de recomendaciones para trabajos futuros relacionados con el tema.

- En el desarrollo del proyecto se implementó el lenguaje Pas++. Este lenguaje permite la programación orientada a objetos, pero solo incluye un subconjunto de características de un lenguaje convencional como herencia simple, constructores, instanciación de objetos, ámbito de atributos y métodos. Debido a esto, un posible trabajo futuro seria implementar las demás características de la programación orientada a objetos como: polimorfirsmo, atributos y métodos estáticos, espacio de nombres, paquetes, hilos, etc.

- Con respecto a la liberación de memoria para los objetos, el enfoque utilizado es la liberación en el mismo instante en el cual se ejecuta la orden (delete); asimismo, es responsabilidad del programador incluir instrucciones para este fin. Sin embargo, en los lenguajes de programación orientada a objetos modernos como Java o C#, el enfoque seguido es el de “Garbage Collection”, mediante el cual, el programador no se preocupa de esta tarea. Es el mismo intérprete, mediante algoritmos específicos, quien se encarga de este trabajo. Por lo tanto, un posible trabajo futuro puede ser estudiar este tipo de algoritmos, ampliar y modificar el intérprete para que los utilice.

- En el presente trabajo de tesis se realizó la implementación de algunas técnicas para la optimización de código. Un trabajo futuro puede abordar el tema de la optimización de código de una manera

Tesis publicada con autorización del autor
No olvide citar esta tesis

más amplia, implementando una mayor cantidad de métodos de optimización. Asimismo, se puede optar por un enfoque moderno como lo es la Exploración de Espacios de Optimización. Estos algoritmos eligen las mejores técnicas a utilizar según la situación, lo que permite observar que técnicas son las más apropiadas para lograr una mayor optimización en términos de memoria y tiempo de ejecución.

- En la implementación del manejo de objetos por parte del intérprete se hace uso de referencias (punteros) que guardan su ubicación en memoria. Por tal motivo, un trabajo futuro puede abordar el desarrollo de "points-to analysis" (análisis de punteros), el cual es un análisis estático que intenta calcular las posibles direcciones a las que puede referenciar un puntero durante la ejecución del programa. De esta forma, se podría conocer en tiempo de compilación los objetos a los que podría referenciar una variable y realizar optimizaciones de código, como la eliminación parcial de redundancias, en el caso de tener varias variables referenciando a un mismo objeto.

- Además, como se mencionó en las observaciones, el intérprete realiza las optimizaciones en un ámbito local. En cambio, al realizar un análisis global se puede encontrar nuevas fuentes de optimización para un código intermedio. Por tal motivo, un trabajo futuro puede ampliar el análisis a todo un programa realizando un análisis de control de flujo de datos.

- Sería interesante que en un posible trabajo futuro el intérprete tenga la capacidad de traducir y ejecutar código embebido de otro lenguaje orientado a objetos (como Java, C#, etc.) dentro del código Pas++. Con esto se podría ampliar las funcionalidades del lenguaje Pas++ adoptándolas de otro que ya las tenga desarrolladas.

- Para aprovechar las características del entorno de desarrollo que se ha implementado (como visualización de tablas, ejecución, etc.) se podría ampliar su funcionalidad para que permita el uso de otros intérpretes desarrollados además de Pas++.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- Finalmente, un posible trabajo futuro puede aprovechar el hecho que las computadoras modernas poseen por lo general dos o más núcleos e implementar un mecanismo en base a hilos para el funcionamiento del intérprete. De este modo, se podrá agilizar el proceso de generación de código intermedio, optimización e interpretación.

Tesis publicada con autorización del autor
No olvide citar esta tesis

6. Referencias

- [ASU2007] Aho, Sethi y Ullman. Compiladores: principios, técnicas y herramientas. Addison-Wesley Iberoamericana, 2001.

- [ABP2008] Marco A. Alvarez , José Baiocchi , José Antonio Pow Sang, Computing and higher education in Peru, ACM SIGCSE Bulletin, v.40 n.2, June 2008.

- [MKM2004] Martin Kong, Intérprete de páginas Web dinámicas para el servidor Apache, Pontificia Universidad Católica del Perú, 2004.

- [KCLT2004] Keith Cooper y Linda Torczon, Engineering a Compiler. Elsevier Science, 2004.

- [LHM2007] Layla Hirsh, Intérprete y entorno de desarrollo para el aprendizaje de lenguajes de programación estructurada, Pontificia Universidad Católica del Perú, 2007.

Tesis publicada con autorización del autor
No olvide citar esta tesis

- [CD2006] Cantone Dante, Implementación y Debugging. MP Ediciones, 2006.

- [AMDB2006] Bertrant Anckaert, Matias Madou y Koen De Bosschere, A model for self-modifying code, Ghent University Electronics and Information Systems Department, 2006.

- [XP2010] http://xprogramming.com/xpmag/whatisxp, Mayo 2010.

- [HHL2002] UNMSM Hinojosa Lazo, Hilmar, "Diseño de un intérprete SQL", *Industrial Data*, Lima, 2002, Vol. 5, Nº1, pp. 49 – 54.
  Última consulta: 14 de Junio del 2010.
  http://sisbib.unmsm.edu.pe/bibvirtual/publicaciones/indata/v05_n1/dise%C3%B1o.htm.

- [SMS2007] Mariano Salomón, Sebastián, Implementación de un intérprete SQL en managed code para dispositivos móviles, Universidad Tecnológica Nacional, 2007.

- [JS1987] J. Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.

- [PNWB1995] Evolving Turing-Complete Programs for a Register Machine with Self-modifying code. Universitat Dortmund, Fachbereich Informatik, 1995.

- [HM1992] Massalin, Henry. Synthesis: An Efficient Implementation of Fundamental Operating System Services. Columbia University, 1992.

- [MKP2011] Mavrogiannopoulos, Nikos, Kisserli, Nessim, Preneel, Bart. A taxonomy of self-modifying code for obfuscation. Katholieke Universiteit Leuven, 2011.

Tesis publicada con autorización del autor
No olvide citar esta tesis

- [HMB2010] Harding, Simon, Miller, Julian, Banzhaf, Wolfgang. "Developments in Cartesian Genetic Programming: self-modifying CGP", Genetic Programming and Evolvable Machines, 2010, Vol. 11. Nº3, pp. 397-439.

- [MAM2006] Manuel Alfonseca Moreno. Compiladores e intérpretes: teoría y práctica. Pearson Education, 2006.

- [BP] Borland Pascal. http://info.borland.com/pascal/tp7fact.html
Última consulta: 23 de Agosto de 2010.

- [OP] Object Pascal
http://www.mactech.com/articles/mactech/Vol.02/02.12/ObjectPascal/
Última consulta: 23 de Agosto de 2010.

- [FP] Free Pascal. http://www.freepascal.org/
Última consulta: 23 de Agosto de 2010.

Tesis publicada con autorización del autor
No olvide citar esta tesis