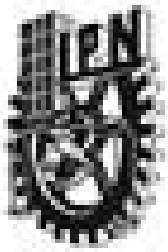
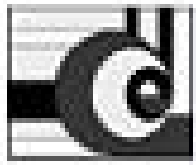


INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN
LABORATORIO DE SIMULACIÓN Y MODELADO



UN MODELO DE INTÉRPRETE PARA UN LENGUAJE DE PROGRAMACIÓN DE SISTEMAS BASADO EN C

TESIS

QUE PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS DE LA COMPUTACION

PRESENTA:

ING. ROBRIGO ALEXANDER CASTRO CAMPOS

DIRECTORES DE TESIS:

M. EN C. GERMAN TÉLLEZ CASTILLO
CIC, IPN

DR. FRANCISCO JAVIER ZARAGOZA MARTÍNEZ
UAM AZCAPOTZALCO

MÉXICO D. F. ENERO 2011

Tabla de contenido

Resumen

El significado de lo que es un entorno de programación moderno ha cambiado significativamente en los últimos años. Los lenguajes de programación tradicionales como C, C++ o Fortran han perdido terreno tanto en la industria como en la academia ante tecnologías aparentemente más flexibles. Esto es una consecuencia de la importancia que ha tomado la computación en la industria y el comercio y una fuente importante de debates pues diferentes sectores tienen diferentes necesidades.

La mayoría de los lenguajes creados a partir de 1980 han optado por la facilidad de uso a cambio de sacrificar el rendimiento en tiempo y el consumo de memoria de los programas resultantes, dejando de lado lenguajes más eficientes pero que generalmente son percibidos como complejos. Si bien buena parte del problema radica en la formación de los programadores, mucha de la percibida complejidad tiene origen en el lento proceso de desarrollo y estandarización de muchos de estos lenguajes. Además, anteriormente existían limitantes tecnológicas importantes que hacían inviable la implementación de mejores diseños.

Esta tesis tiene como motivación la creación de un lenguaje de programación que sea al menos tan eficiente como C y que sea más expresivo que los lenguajes de propósito general más usados actualmente. C++ tiene objetivos similares pero pretende ser compatible con C a nivel de código fuente, heredando de esta forma muchos de sus problemas. El lenguaje presentado en esta tesis no se impone esta restricción.

Esta tesis presenta también el diseño y la implementación de un intérprete para este lenguaje que sea capaz de ejecutar parcialmente programas con errores léxicos o sintácticos y hacerlo de manera eficiente, eliminando así algunos de los inconvenientes que se presentan al querer llevar a ejecución el código fuente. La implementación de este intérprete demuestra también la factibilidad de implementar el lenguaje propuesto.

Abstract

The meaning of a modern programming environment has changed significantly in the last years. Traditional programming languages such as C, C++ and Fortran have lost terrain in both the enterprise and the academy against technologies that appear to be more flexible. This is a consequence of the commercial impact of enterprise computing and an important source of discussions as different sectors have different needs.

Most of the programming languages invented after 1980 have opted for easy of use to the detriment of both runtime performance and memory consumption of the resulting programs, ignoring more efficient but often considered more complex alternatives. While many of the current problems in software engineering have their root in the programmers' education, much of the complexity of some programming languages is caused by their slow evolution and standarization. Also, these languages often include less than optimal design decisions that were forced by the technological limitations that existed several decades ago.

The motivation of this thesis is the creation of a programming language that is as or more efficient than C while being at the same time more expressive than the most used general purpose programming languages. C++ has similar goals but it pretends to be compatible with C at source code level, inheriting many of its problems. The language presented in this thesis does not have this restriction.

This thesis also presents the design and the implementation of an interpreter for the mentioned language that is able to partially execute some lexically or sintactically bad formed programs while doing this efficiently, eliminating some of the inconvenients typically found while trying to execute source code. The implementation of this interpreter also shows the feasibility of implementing the proposed language.

1 Introducción

1.1 Antecedentes y planteamiento del problema

Los lenguajes de programación de sistemas pueden definirse como aquellos lenguajes dirigidos a construir aplicaciones donde los factores constantes de sobrecarga son muy importantes además de que permiten interactuar directamente con el hardware [1]. Sin embargo desde la década de los 80 no ha surgido ningún lenguaje de este tipo que goce de gran popularidad y que no introduzca sobrecargas implícitas en tiempo de ejecución o consumo de memoria. En lugar de eso se le ha dado una mayor importancia a la programación de aplicaciones donde se prefiere la productividad del programador.

Uno de los avances más importantes en la computación para el desarrollo de software fue la invención del lenguaje C que ha sido considerado como un ensamblador universal por su alta disponibilidad y por el rendimiento de los programas escritos en él [2] y al que a veces se le ha dado su propia categoría como un lenguaje de mediano nivel para diferenciarlo de los lenguajes de bajo nivel (ensambladores) y de aquellos que ofrecen mecanismos de mayor abstracción (lenguajes de alto nivel). No es de extrañar que este lenguaje sea todavía de los más populares en aplicaciones de alto rendimiento o donde los recursos son escasos, como en las aplicaciones incrustadas.

La invención de C++ tuvo como propósito explícito [2] crear un lenguaje que fuera tan eficiente como C pero que tuviera mecanismos de abstracción suficientemente expresivos como para programar aplicaciones grandes de una manera mucho más sencilla; gracias a esto C++ fue el lenguaje por excelencia de las décadas de los 80 y 90. Una de las razones de su éxito fue su compatibilidad a nivel de código fuente con C, lo que permitía reutilizar código escrito anteriormente y facilitaba la transición de programadores de C a C++. Su compatibilidad con C, sin embargo, hizo que C++ heredara muchos de los inconvenientes sintácticos y semánticos de aquél.

La introducción de Java [3] a mediados de los 90 produjo un cambio de enfoque hacia la productividad del programador al volver populares las verificaciones implícitas en tiempo de ejecución y la recolección automática de memoria que consiste en liberar parcialmente al programador del manejo manual de la memoria a cambio de introducir un agente adicional que la administra durante la ejecución del programa. Además pretendía ser más simple que C++ al adoptar de manera exclusiva el paradigma de programación

orientada a objetos. Desafortunadamente obligaba a aquellos recién iniciados en la computación que eligieran (o fueran obligados a elegir) a Java como su primer lenguaje a aprender conceptos avanzados de programación orientada a objetos desde el inicio. Java fue rechazado por aquellos sectores que requerían alto rendimiento en sus aplicaciones pero revivió las investigaciones en torno a las optimizaciones de intérpretes y la compilación justo a tiempo, que es la generación de código nativo en tiempo de ejecución [4].

La reutilización de código es la base que ha permitido, con el paso de los años, crear aplicaciones cada vez más complejas para satisfacer las demandas de la industria. Éste ha sido un factor fundamental en la aceptación de la programación orientada a objetos, y por lo tanto de Java y de otros lenguajes relacionados. Por otra parte, la creación y posible comercialización de componentes genéricos, de calidad probada y de máxima eficiencia (tan eficientes como un componente de propósito específico) permitiría satisfacer las necesidades de cualquier sector de la industria y no solamente de alguna organización o empresa en particular, lo que representaría una revolución en la forma de construir programas. Debido a la necesidad de eficiencia máxima, prácticamente todos los avances en esta área de investigación se han dado utilizando C++ y el resultado, conocido como programación genérica [5] y que incluso ha logrado influir en el propio Java, ha dejado entrever que es necesario construir un nuevo lenguaje desde los fundamentos para habilitar la expresividad requerida pues a pesar de que la programación genérica está más desarrollada en C++, éste sigue limitándola de manera importante.

Esta tesis presenta un lenguaje de programación que elimina tanto las limitantes como los inconvenientes de C y C++ y que al mismo tiempo no presenta las debilidades encontradas en estos y otros lenguajes con respecto al soporte para la programación genérica. Presentamos además el diseño y la implementación de un moderno compilador justo a tiempo para este lenguaje que capitalice los beneficios de no necesitar compilación previa para permitir la ejecución parcial de programas con problemas de sintáxis.

1.2 Objetivos

Objetivo general

Diseñar e implementar un lenguaje de programación de propósito general con énfasis en programación de sistemas y con soporte para la programación genérica, cuya implementación sea un intérprete de alto rendimiento que elimine algunos inconvenientes comunes en la ejecución de programas fuente.

Objetivos particulares

- Determinar las ventajas y desventajas de los lenguajes tradicionales de programación y sus implementaciones, especialmente C y C++.
- Identificar las principales debilidades de los lenguajes de programación actuales con respecto a la programación genérica.
- Diseñar un lenguaje de programación que elimine las desventajas y debilidades encontradas al nivel del lenguaje de programación.
- Implementar un intérprete para el lenguaje desarrollado que demuestre la factibilidad del lenguaje, que lleve a cabo el proceso de compilación eficientemente y que evite algunos inconvenientes de otras implementaciones de lenguajes.

1.3 Justificación

En la industria del software, las prácticas y metodologías han cambiado drásticamente en las últimas dos décadas pues la demanda de aplicaciones cada vez más grandes y complejas ejerce una fuerte presión sobre la capacidad de ingeniería de los programadores actuales y las empresas que los contratan. Es por esto que la reutilización de código se ha convertido en un eje vital para muchas organizaciones pues codificar aplicaciones desde cero implicaría muchas veces un gasto no permisible en tiempo y recursos humanos.

En parte por lo anterior, los lenguajes con bibliotecas de componentes más extensas son los más populares para lo que se conoce como aplicaciones empresariales. Desafortunadamente la mayoría de estos lenguajes y de estas bibliotecas muchas veces son inutilizables para aplicaciones que requieran alto rendimiento por lo que las organizaciones que las construyen se ven forzados a reimplementar dichos componentes, muchas veces por su propia cuenta y únicamente para su uso propio.

La programación genérica describe un estilo de programación en el cual los componentes creados son de máxima eficiencia y reutilizables en una inmensa cantidad de situaciones, lo cual es logrado mediante la búsqueda de las abstracciones correctas y la implementación de las mismas en un lenguaje adecuado. El poco soporte para este estilo actualmente representa un obstáculo para crear una industria que comercialice componentes de software probados y de máxima eficiencia. El lenguaje C++, considerado por muchos como el mejor lenguaje sobre el cual aplicar este estilo de programación, presenta suficientes inconvenientes como para considerar la creación de un nuevo lenguaje.

1.4 Beneficios esperados

La mayor parte de la investigación sobre la programación genérica todavía está dirigida a encontrar el soporte adecuado para ésta en los diferentes lenguajes de programación. Sin embargo consideramos que el objetivo de la programación genérica, la construcción sistemática de componentes eficientes y genéricos, es un área de investigación más útil y desafiante. Esta tesis tiene por meta encontrar el diseño de un lenguaje que permita a la investigación sobre la programación genérica finalmente avanzar a esta nueva etapa.

Aunque no se pretende innovar en el área de implementación de compiladores e intérpretes consideramos que, en conjunto con un adecuado diseño del lenguaje, se puede obtener una implementación que reduzca significativamente el tiempo de compilación requerido por implementaciones de otros lenguajes. Esto permite ofrecer tanto compiladores como intérpretes para este lenguaje que realicen su trabajo en una fracción del tiempo acostumbrado en otros lenguajes y con un bajo consumo de memoria.

1.5 Alcances y límites

En la actualidad los entornos de programación incluyen una serie de herramientas como bibliotecas de utilidades, depuradores y editores de texto. En esta tesis no se aborda la creación de ninguna de estas herramientas y se hace énfasis exclusivamente en el diseño del lenguaje desarrollado y en una implementación interpretada del mismo.

Con respecto al lenguaje, la meta es lograr eliminar los problemas de diseño más importantes de C y C++ y superar las limitantes presentes en los lenguajes de programación actuales en cuanto a su soporte para la programación genérica. El diseño del lenguaje está finalizado cuando éste logre lo anteriormente mencionado.

Con respecto a la implementación del lenguaje, ésta se considera completa al implementarse todas las etapas en las que está dividida y al existir una forma básica de generación de código que permita realizar algunas pruebas sencillas de funcionamiento. En general, se dedica poco tiempo a la etapa de generación de código puesto que los compiladores e intérpretes actuales implementan optimizaciones agresivas o complicadas que difícilmente se podrían llegar a igualar para fines comparativos durante la elaboración de la tesis aquí presentada.

1.6 Organización de la tesis

Esta tesis está dividida en ocho capítulos, siendo la introducción el capítulo 1. El capítulo 2 introduce algunos conceptos de la teoría de lenguajes la cual es necesaria para poder analizar las decisiones de diseño de la sintaxis de C y C++, así como para poder discutir la sintaxis del lenguaje propuesto. Además se presenta el estado de la técnica en compiladores e intérpretes y una lista de los problemas de optimización más importantes que se presentan en la implementación de lenguajes.

En el capítulo 3 examinamos las metas de diseño de C y C++, la evolución de estos lenguajes y las ventajas que se desean conservar de ellos. Se analizan los problemas, inconsistencias e inconvenientes sintácticos y semánticos más importantes de estos lenguajes. Se describen también las dificultades que surgen al implementarlos.

En el capítulo 4 se explica a detalle en qué consiste el paradigma de la programación genérica. Se justifica por qué es relevante, se señalan las limitantes inherentes de la programación orientada a objetos y se muestra cómo la programación genérica no tiene esas limitantes. Los problemas presentados en el capítulo 3 toman relevancia al examinar los problemas en la evolución de C++ con respecto a la programación genérica.

En el capítulo 5 se presenta el diseño de un lenguaje que elimina los problemas encontrados en C y C++. También se muestra cómo este lenguaje es adecuado para el desarrollo y la evolución de la programación genérica en base a lo discutido en el capítulo 4.

En el capítulo 6 se muestra el diseño de un compilador justo a tiempo de alto rendimiento que implementa el lenguaje propuesto y que facilita la ejecución de programas al eliminar algunas restricciones encontradas usualmente en otros ambientes de programación. Se demuestra la factibilidad del lenguaje al presentar la implementación del mismo.

El capítulo 7 presenta las conclusiones sobre el trabajo realizado haciendo énfasis en el cumplimiento de los objetivos planteados. El capítulo 8 presenta las posibles actividades a realizar a futuro además de señalar algunas áreas de investigación en las cuales valdría la pena incursionar tomando como base inicial el trabajo presentado en esta tesis.

2 Preliminares sobre lenguajes

En este capítulo se introducen algunos conceptos de la teoría del lenguaje necesarios para la comprensión de algunas explicaciones posteriores. También se da una pequeña reseña sobre la evolución del diseño e implementación de los lenguajes de programación a través de la historia. Se presenta el estado de la técnica para las implementaciones interpretadas y una lista de las optimizaciones más comunes de los compiladores e intérpretes actuales.

2.1 Expresiones regulares y lenguajes regulares

Un lenguaje formal es un conjunto numerable de cadenas finitas. Las cadenas están formadas por símbolos que pertenecen a un conjunto finito llamado alfabeto. Un lenguaje es regular si puede ser reconocido por un autómata finito determinista. Un automáta finito determinista es una quinteta $<Q, \Sigma, \delta, q_0, F>$ donde:

- Q es un conjunto finito de estados,
- Σ es un conjunto finito de símbolos y es denominado alfabeto de entrada,
- δ es un conjunto de reglas de transición de la forma $Q \times \Sigma \rightarrow Q$,
- q_0 es un estado perteneciente a Q y es el estado inicial y
- F es un subconjunto de Q y denota el conjunto de estados de aceptación.

El autómata opera de la siguiente forma: se lee un símbolo de la entrada. De acuerdo al símbolo leído el autómata pasa a un nuevo estado y avanza al siguiente símbolo de la cadena. Si el autómata termina en un estado de aceptación al consumir la totalidad de la cadena, la cadena es aceptada.

Las expresiones regulares son cadenas escritas en un lenguaje formal y son capaces de describir la forma de las cadenas que pertenecen a un lenguaje regular. Son equivalentes en poder a los autómatas finitos. Es más común describir un lenguaje regular en términos de una expresión regular que en términos de las reglas de transición de un autómata finito.

2.2 Gramáticas y lenguajes libres de contexto

Un lenguaje también puede describirse mediante una gramática, la cual describe la forma de generar las cadenas que pertenecen a dicho lenguaje. Una gramática es una cuarteta $<N, \Sigma, P, S>$ donde:

- N es un conjunto finito de símbolos no terminales,
- Σ es un conjunto finito disjunto a N de símbolos terminales,
- P es un conjunto finito de reglas de producción de la forma $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$ y
- S es un elemento de N llamado símbolo inicial.

Arriba, * es el operador de Kleene que actúa sobre un conjunto de símbolos y que denota al conjunto de cadenas formadas por la concatenación de cero o más de las cadenas contenidas en el operando.

Una gramática también es llamada sistema de reescritura pues las reglas de producción son usadas como un mecanismo de expansión que permite la generación de cadenas diversas. El análisis sintáctico es el proceso de reconocer si una cadena pertenece a un lenguaje mediante la aplicación de las reglas de producción. La generación de las cadenas del lenguaje se logra mediante la reescritura de un símbolo inicial y la aplicación sucesiva de las reglas de producción las veces que sean necesarias y en cualquier orden. Si existe más de una manera de generar la misma cadena, la gramática se denomina ambigua.

Una gramática libre de contexto es aquella en la que las reescrituras se aplican recursivamente pero no se sobreponen. Un ejemplo de una gramática que no es libre de contexto es una gramática que deba verificar paréntesis balanceados pero a la que se le pide que deba poder manejar el caso " ([ )] " donde la verificación del cierre de ") " interrumpiría la verificación del cierre del " [" abierto si esto se hiciera de manera recursiva. Un lenguaje es libre de contexto si existe una gramática libre de contexto que lo genere.

Para expresar una gramática libre de contexto usualmente se utiliza la notación Backus-Naur [6] y algunas variantes como la Backus-Naur extendida:

no_terminal =
    regla de producción ;

regla de producción:

,	concatenación
|	disyunción
"…"	terminal
[…]	opcional
{…}	repetición (cero o más veces)
(…)	agrupamiento
"0" ... "9"	secuencia especial

Para reconocer un lenguaje libre de contexto un autómata de pila es suficiente y puede demostrarse que son equivalentes. Un autómata de pila es una septeta $<Q, \Sigma, \Gamma, \delta, q_0, Z, F>$ donde

- Q es un conjunto finito de estados,
- Σ es un conjunto finito de símbolos y es denominado alfabeto de entrada,
- Γ es un conjunto finito de símbolos y es denominado alfabeto de la pila,
- δ es un conjunto de reglas de transición de la forma $Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \longrightarrow Q \times \Gamma^*$,
- q_0 es un estado perteneciente a Q y es el estado inicial,
- Z es un símbolo perteneciente a Γ y es el símbolo inicial de la pila y
- F es un subconjunto de Q y denota el conjunto de estados de aceptación.

El autómata opera de la siguiente forma: se lee un símbolo de la entrada y el símbolo de la cima de la pila. De acuerdo a los símbolos leídos el autómata pasa a un nuevo estado y opcionalmente puede modificar la pila apilando un nuevo símbolo o bien desapilando el símbolo actual.

El análisis sintáctico puede hacerse principalmente de dos maneras:

- *Análisis de arriba hacia abajo*: Intenta encontrar la derivación más a la izquierda utilizando la expansión de las reglas de producción. Los símbolos son consumidos de izquierda a derecha. Un ejemplo de analizadores de arriba hacia abajo son los analizadores LL y las gramáticas que pueden ser analizadas por estos son llamadas gramáticas LL. Un analizador LL es llamado analizador LL(k) si puede analizar k símbolos anticipadamente para tomar decisiones. Un analizador LL(*) es aquél que no tiene un límite superior en el número de símbolos que puede necesitar analizar anticipadamente. Las gramáticas con $k > 1$ requieren usar búsqueda con retroceso o utilizar una gran cantidad de memoria [7].

- *Análisis de abajo hacia arriba*: El analizador empieza consumiendo la entrada e intentado reescribirla para generar el símbolo de inicio, es decir, el proceso se realiza de las hojas del árbol de derivación equivalente hacia la raíz. Un analizador LR es un tipo de analizador de abajo hacia arriba y consume símbolos de izquierda a derecha y produce la derivación más a la derecha. El término LR(k) se refiere a la capacidad de analizar anticipadamente k símbolos para decidir qué hacer (ya sea agregar un nuevo símbolo a la entrada o intentar transformar la entrada ya leída). Debido a la dificultad de codificación, para gramáticas LR usualmente se usan generadores de analizadores sintácticos que toman como entrada la gramática.

Pueden existir diferentes gramáticas que describan el mismo lenguaje así como diferentes maneras de implementar un analizador sintáctico para la misma gramática. En el contexto del análisis LL, se dice que si existe un analizador LL(k) para una gramática entonces la gramática es LL(k). De manera similar, se dice que si para un lenguaje existe una gramática LL(k) entonces el lenguaje es LL(k). Lo análogo ocurre para LR.

2.3 Precedencia y asociatividad de operadores

Las reglas de precedencia y asociatividad determinan de manera inequívoca el orden en el que deben realizarse las operaciones de una expresión. La precedencia tiene prioridad y las reglas de asociatividad se usan en caso de que las operaciones en conflicto tengan la misma precedencia.

Las reglas de precedencia usualmente se describen asignando un nivel a cada operador o función utilizada en una expresión. Además, existen símbolos de agrupamiento que permiten manipular manualmente el orden de las operaciones. Por ejemplo, en aritmética las operaciones de multiplicación y división tienen una precedencia más alta que las operaciones de suma y resta por lo que la expresión A + B * C debe evaluarse como A + (B * C) donde la multiplicación se efectúa antes de realizar la suma. Los paréntesis son los símbolos de agrupamiento más usados.

Cuando un operando está precedido y seguido por operadores con la misma precedencia, las reglas de precedencia por sí solas no determinan a qué operación pertenece el operando. La elección es determinada por la asociatividad de los operadores: si un operador tiene asociatividad hacia la izquierda las operaciones se agrupan de izquierda a derecha, de lo contrario se agrupan de derecha a izquierda.

El operador + que denota la suma aritmética tiene asociatividad izquierda, es decir A + B + C se interpreta como (A + B) + C. Los operadores de asignación de varios lenguajes de programación usualmente tienen asociatividad derecha, es decir A ← B ← C se interpreta como A ← (B ← C). La precedencia y la asociatividad de operadores pueden describirse mediante gramáticas libres de contexto.

2.4 El proceso de compilación

La compilación es la transformación de código fuente de un lenguaje de programación a código en otro lenguaje más cercano a la máquina,

generalmente siendo éste el lenguaje ensamblador de algún procesador. Los compiladores generalmente presentan las siguientes fases:

- *Análisis léxico*: Consiste en la identificación y separación de símbolos y palabras reconocidas por el lenguaje de programación. Su tarea principal es la de eliminar los detalles de formato irrelevantes que presente el código fuente, aunque esto depende del lenguaje. Lenguajes como ECMAScript [8] o Python [9] dan significado semántico al espaciado (saltos de línea o identación) mientras que lenguajes como C o Java son libres de formato. El analizador léxico se implementa generalmente utilizando expresiones regulares. Existen programas que generan analizadores léxicos, un ejemplo es Lex [10].

- *Preprocesamiento*: Consiste en ejecutar directivas que generalmente manipulan directamente el código fuente (expansión de macros, inclusión de archivos) aunque en algunos lenguajes como Lisp [11] se permite modificar el código del propio programa en tiempo de ejecución.

- *Análisis sintáctico*: Es el proceso de reconocer si el programa (transformado ya en un flujo de unidades reconocidas denominadas tokens o palabras y emitido por el analizador léxico) está bien formado con respecto a la gramática del lenguaje. El análisis sintáctico generalmente no es suficiente para determinar si un programa es correcto bajo las reglas de un lenguaje pues únicamente analiza la forma del programa y no la semántica del mismo (por ejemplo, comprobación de tipos). La salida del analizador léxico usualmente es un árbol llamado árbol de sintaxis abstracta (AST por las siglas en inglés de *abstract syntax tree*) que mantiene implícitos en su estructura detalles del código fuente (como agrupación de paréntesis y asociatividad de operadores). Ver figura 1.

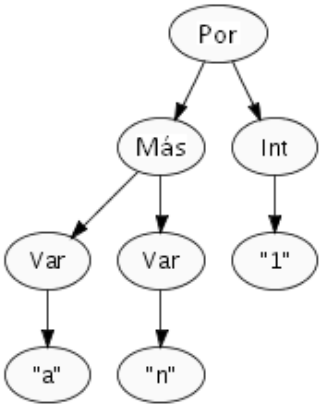


Figura 1. AST para la expresión (a + n) * 1

Existen programas que toman como entrada una gramática y generan el analizador sintáctico correspondiente; algunos ejemplos son ANTLR [12], Bison [13] y YACC [14]. El uso de estos programas es casi universal para gramáticas más complicadas que LL(k) debido a las numerosas reglas de transición que resultan y a la dificultad que representa codificarlas manualmente.

- *Análisis semántico*: Consiste en determinar si el programa está semánticamente bien formado. Se implementa comúnmente visitando el árbol resultante del análisis sintáctico. Se encarga de construir las tablas de símbolos, verificar la coincidencia de tipos y la consistencia entre declaraciones. Produce como salida una representación del programa en el lenguaje destino o en un formato cercano a éste.

- *Optimización y generación de código*: Es el proceso de convertir el programa resultante del análisis semántico en un programa equivalente pero más rápido y generar código en el lenguaje destino si esto no se ha hecho anteriormente.

2.5 Evolución de los lenguajes y sus implementaciones.

El surgimiento de los compiladores se da gracias a la creación de los lenguajes de alto nivel. El primero de ellos, Fortran [15], fue inventado por John Backus en 1957. En la década de los 50 los programas aún eran almacenados en tarjetas perforadas por lo que las primeras versiones de Fortran no tenían una sintaxis libre de formato, facilitando la implementación pero dificultando su uso. A pesar de que el rendimiento de los programas era inferior al código ensamblador escrito a mano, éste era aceptable y el lenguaje aumentaba la productividad de los programadores a tal grado que su aceptación fue casi inmediata; el reto pasó a ser reducir la brecha entre el código generado por el compilador y el código escrito a mano. Con la aparición de los lenguajes de alto nivel, el compilador pasó a ser además un mecanismo para lograr la portabilidad; sólo era necesario escribir un compilador para cada plataforma en lugar de reescribir las aplicaciones.

La principal motivación para la creación de C [16] fue el desarrollo del sistema operativo UNIX el cual tuvo como meta fundamental la portabilidad. Al ser UNIX un sistema operativo, la eficiencia era indispensable. C fue el intento más exitoso de un lenguaje de alto nivel que mapeara eficientemente las primitivas más comunes de los procesadores de aquel entonces en un lenguaje relativamente independiente de la máquina. Desafortunadamente las limitaciones tecnológicas de los años 70 seguían dificultando el diseño de los

lenguajes de programación. Kenneth Thompson, el creador del lenguaje B, antecesor de C y el sucesor de BCPL, rediseñó la sintaxis de este último para permitir realizar la compilación en una sola pasada puesto que la computadora disponible para desarrollo e investigación en los laboratorios Bell (actualmente de Alcatel-Lucent y fundados por AT&T) sólo contaba con 8KB de memoria. El diseño minimalista de B fue heredado por C con el mismo propósito.

Estilos de programación reconocidos actualmente como superiores en cuanto a productividad de los programadores, como lo es la programación orientada a objetos introducida por el lenguaje Simula de 1967, no fueron bien recibidos inicialmente debido a que los programas ejecutables producidos eran lentos [2]. C++, un lenguaje de 1979 basado en C y que adoptó la programación orientada a objetos, permitió la gradual aceptación de este paradigma a expensas de la simplificación del diseño de Simula y requiriendo mejorar las optimizaciones realizadas por el compilador.

El lenguaje Ada [17] de 1978 introdujo la parametrización de código (ya sea con tipos o valores constantes) lo que permite escribir algoritmos y estructuras en términos más generales lo que dio origen a los primeros intentos de programación genérica [18]. La instanciación es el proceso de generar una versión ejecutable de dicho código al conocer los valores reales de los parámetros utilizados en la parametrización. Ada requiere realizar las instanciaciones manualmente y también requiere conocer de antemano ciertas condiciones que deben cumplir los valores parametrizados; el código parametrizado debe limitarse a usar las características impuestas por dichas condiciones. Dicha restricción permite, sin embargo, instanciar código en tiempo de ejecución y no requiere reanalizar el código parametrizado para cada instanciación diferente.

En 1990 C++ introduce un mecanismo similar [2] con algunas diferencias:

- Los requerimientos sobre los valores parametrizados son implícitos en el código; la verificación de código bien formado se realiza únicamente en el momento en que se realiza la instanciación.
- Las instanciaciones se realizan automáticamente por el compilador y cada instanciación se analiza indivualmente.
- Existe la especialización, es decir, el programador puede proporcionar manualmente el código que deba utilizarse para valores específicos de los parámetros de una instanciación.

Debido a lo anterior, las instanciaciones únicamente pueden realizarse en tiempo de compilación. La capacidad de proporcionar especializaciones hizo que este mecanismo fuera Turing-completo [19] lo cual trae como consecuencia

la capacidad de realizar cálculos arbitrarios en tiempo de compilación pero que al mismo tiempo vuelve indecidible el problema de determinar si el proceso de compilación terminará. Por estas razones, C++ es uno de los lenguajes más demandantes para los implementadores de compiladores.

A pesar de que C y C++ son relativamente independientes de la máquina sobre la que se ejecutan sus programas, muchas de las utilerías necesarias en la programación de aplicaciones grandes son dependientes del sistema operativo. Java volvió popular el concepto de máquina virtual a nivel de proceso que permite que un proceso se ejecute sobre una máquina hipotética la cual se emula sobre el sistema operativo y para lograrlo se precompila a un lenguaje intermedio el cual es interpretado al momento de su ejecución e introduce verificaciones implícitas en tiempo de ejecución para evitar que ocurran errores como accesos inválidos a memoria o desbordamientos. Tanto Java como muchos de los lenguajes inspirados en él implementan optimizaciones potentes [20] que los acercan al rendimiento de los lenguajes compilados gracias a los avances logrados en los últimos años en el área de optimizaciones para intérpretes y en la compilación justo a tiempo.

El lenguaje ECMAScript es un lenguaje en el que la semántica de sus operaciones dependen en gran medida de lo que ocurra en tiempo de ejecución, lo que lo cataloga como un lenguaje dinámico. Con la explosión del uso de Internet, la creciente popularidad de ECMAScript y la complejidad de las aplicaciones que hacen uso de él, se ha obligado a los implementadores de este lenguaje a investigar nuevas técnicas que permitan lograr un rendimiento aceptable, ya que las primeras implementaciones han resultado ser insuficientes para la ejecución de aplicaciones modernas. Esto ha resultado una tarea complicada debido al alto dinamismo del lenguaje.

Los modelos de intérpretes usados son los siguientes [21, 22]:

- *Interpretación línea a línea del código fuente*: El intérprete visita cada línea del código fuente y la interpreta en ese momento. Para programas cortos es el modelo más eficiente pero se vuelve rápidamente inadecuado para programas que contengan y ejecuten ciclos, pues las líneas deben reevaluarse sintáctica y semánticamente cada vez que son ejecutadas. Por su naturaleza, la implementación de lenguajes dinámicos es sencilla.

- *Interpretación mediante la construcción y visita del árbol sintáctico*: El intérprete construye el árbol abstracto de sintaxis en una primera pasada. La ejecución real del programa se implementa mediante una visita repetida a las secciones del árbol que representen las instrucciones. Es esencialmente igual al modelo anterior excepto que evita la sobrecarga

de analizar sintácticamente cada línea más de una vez, pues aunque la semántica asociada al comando pueda cambiar, no cambia su sintaxis. Las primeras implementaciones de ECMAScript fueron de este tipo.

- *Interpretación mediante la generación de código intermedio*: El intérprete genera instrucciones en un lenguaje intermedio y la ejecución real del programa es la ejecución de esas instrucciones. Un diseño adecuado del lenguaje intermedio es de gran importancia en la construcción de intérpretes de este tipo y para lenguajes dinámicos es importante que el lenguaje sea lo suficientemente abstracto y general. Dos factores, el diseño del lenguaje y el mecanismo de despacho de las instrucciones, son determinantes en el rendimiento de estos intérpretes por lo que han surgido diferentes diseños y modelos de implementación. Las primeras versiones de Java eran intérpretes de este tipo.

- *Interpretación con compilación justo a tiempo*: El intérprete genera código opcionalmente en un lenguaje intermedio y eventualmente genera código en lenguaje ensamblador. La generación de código ensamblador puede ser costosa por lo que algunas implementaciones sólo realizan ésto en bloques de código ejecutados frecuentemente. Las implementaciones más recientes de Java usan este modelo. La implementación de ECMAScript de Chrome, el explorador de Internet de Google, genera código ensamblador directamente [23].

Cuando se decide usar un lenguaje intermedio, el diseño de éste y la forma en que las instrucciones serán despachadas para su ejecución son determinantes en el rendimiento de los intérpretes por lo que han surgido diferentes diseños y modelos de implementación. Existen dos modelos básicos en cuanto al diseño del lenguaje intermedio [24]:

- *Lenguajes con un conjunto de instrucciones basados en un modelo de pila*: Existe un área especializada para los operandos a usar próximamente, denominada la pila, y sólo se pueden realizar cálculos con los valores que estén almacenados en ella. La sentencia A=B+C en este modelo puede escribirse como:

```
push b
push c
add
store a
pop
pop
```

- *Lenguajes con un conjunto de instrucciones basados en un modelo de registros*: Existen un conjunto de registros y operaciones que pueden realizar

cálculos sobre uno o más de estos registros. La sentencia A=B+C en un conjunto de instrucciones basado en registros puede escribirse como:

```
add a, b, c
```

Java usa el model basado en pila mientras que algunas máquinas virtuales recientes como SquirrelFish [25] (el motor de ECMAScript del explorador de Internet Safari) usan el model basado en registros.

El despacho de las instrucciones consiste en determinar el código a ejecutarse para cada instrucción. Los siguientes modelos son los más eficientes y los más usados [24, 26]:

- *Código hilado*: Se decodifican las instrucciones y se produce una tabla de direcciones a las subrutinas que implementan cada una de las instrucciones.

- *Código hilado mediante token*: Se decodifican las instrucciones y se produce una tabla de desplazamientos a un arreglo (que dependiendo del tamaño del entero necesario para direccionar el arreglo puede ser más compacta que una tabla con direcciones) y éste es el que contiene las direcciones de las subrutinas. Java y el entorno de programación .NET utilizan este modelo.

- *Código hilado directo*: Se decodifican las instrucciones y se produce una tabla de direcciones a porciones de código ensamblador que implementan cada instrucción. La última instrucción de cada porción aumenta el índice de la tabla de direcciones y realiza el salto a la siguiente instrucción. SquirrelFish usa este modelo.

- *Código hilado mediante subrutinas*: Se decodifican las instrucciones y se produce código ensamblador que es una sucesión de llamadas a las subrutinas que implementan las instrucciones. SquirrelFish-Extreme [27] usa este modelo.

2.6 Optimizaciones relevantes en lenguajes de alto nivel

El arte de la optimización es un área complicada. El problema de decidir si dos máquinas de Turing son equivalentes es indecidible, por lo que las optimizaciones deben realizarse con cuidado. Afortunadamente existen muchas optimizaciones simples y que reportan gran beneficio o bien que son requeridas

por la especificación de los lenguajes de programación. Algunas de estas optimizaciones son:

- *Expansión de funciones en línea*: El paradigma de la programación orientada a objetos tiende a generar muchas funciones pequeñas. Algunas veces la sobrecarga del mecanismo de llamada a subrutina es mayor que el trabajo real realizado por la función. En estos casos expander la función en línea genera ejecutables más rápidos y más compactos. El rendimiento de muchos lenguajes, especialmente de C++, depende en gran medida de la capacidad del compilador para generar funciones en línea.

- *Evaluación y propagación de constantes*: Consiste en evaluar expresiones que resultan en valores constantes y en sustituir variables nombradas cuyo valor sea conocido con el valor mismo. Ciertos lenguajes como C requieren que el compilador evalúe expresiones constantes en aquellos lugares donde esto sea necesario, como en la declaración de arreglos. Además, los ensambladores pueden necesitar evaluar operaciones aritméticas simples en el contexto de instrucciones que tomen un operando inmediato.

Por otro lado, existen ciertas optimizaciones que actualmente son sumamente importantes en la implementación de la mayoría de los lenguajes de programación pero corresponden a problemas difíciles (problemas NP-Completos para los que no se conoce ningún algoritmo que corra en tiempo polinomial con respecto al tamaño de la entrada).

- *Asignación de registros*: Aunque los registros pueden trabajar a la misma velocidad que el procesador, la velocidad de la memoria RAM es aproximadamente cien veces menor. Como la cantidad de registros es limitada, la asignación de estos constituye una optimización primordial para código de alto rendimiento. Lamentablemente el problema de la asignación óptima de registros es NP-Completo pues se puede plantear como un problema de coloración de grafos de la siguiente manera: se asigna un nodo del grafo para cada variable nombrada y para cada variable temporal. Aquellas variables que convivan en el tiempo son unidas mediante una arista. Si se tienen R registros, la pregunta es si se pueden colorear todos los nodos del grafo utilizando a lo mucho R colores distintos con la condición de que no pueden existir dos nodos unidos que estén coloreados del mismo color.

  Atacar el problema de la asignación de registros requiere actualmente el uso de heurísticas. Se han propuesto diferentes algoritmos con

complejidad $O(N^2)$ como el algoritmo de Chaitin [28] y otros con complejidad O(N) como el de cuenta de usos y el de exploración lineal [29] siendo estos últimos más útiles en la implementación de compiladores justo a tiempo, donde el tiempo de ejecución es en parte el tiempo de la generación de código.

- *El problema de los alias* [30]: En lenguajes en donde se permiten accesos indirectos a variables como en C con el uso de apuntadores, el problema de los alias consiste en determinar si el valor de una variable debe manipularse constantemente en memoria o si es seguro mantener el valor almacenado en un registro y realizar la actualización hasta el último momento. La razón es que es más rápido acceder a los registros del procesador que a la memoria. El problema se complica considerablemente cuando las optimizaciones potenciales requieren un análisis global del programa, lo cual puede ser sumamente complicado y costoso en términos de los recursos del compilador además de que casi imposible si no se cuenta con el código fuente de todo el programa.

- *Fragmentación de memoria*: Consiste en el desperdicio paulatino de la memoria que se origina al recibir solicitudes de bloques contiguos de cierto tamaño y no poder reasignar bloques que ya fueron liberados pero que son de tamaño menor. Lenguajes como Java, que utilizan la recolección automática de la memoria que ya no es referida dentro del programa, no dan facilidades para lograr un control total sobre la asignación de memoria. Esto ha limitado la adopción de este mecanismo en aplicaciones de alto rendimiento pues la fragmentación sigue siendo un problema abierto [31].

3 El estado de C y C++

El lenguaje de programación C ha sido uno de los lenguajes más exitosos e influyentes de la historia. Comúnmente C es el primer lenguaje de alto nivel que es implementado cada vez que surge una nueva plataforma, por lo que es considerado el lenguaje más cercano a ser un ensamblador universal. C y C++ representan dos de los lenguajes más usados en la industria y la influencia sintáctica de C puede observarse en muchos de los lenguajes de reciente creación.

El lenguaje C++, que es el lenguaje más exitoso basado en C, es en la actualidad el lenguaje por excelencia para el desarrollo de software de sistemas y de aquellas aplicaciones que requieran rendimiento. Prácticamente la totalidad de los sistemas operativos y muchas de las aplicaciones que se ejecutan de manera nativa sobre la computadora están desarrolladas en C o C++ [32]. A pesar de lo anterior y después de un crecimiento explosivo de usuarios en la década de los 80 cuando se duplicaba el número de programadores cada siete meses [2] el uso de estos lenguajes no ha crecido de manera significativa desde 1995; existen aproximadamente 4 millones de programadores de C++ mientras que el lenguaje Java reporta al menos 6 millones [33].

Existen varias razones para lo anterior. Una de las más poderosas es que ambos lenguajes están estandarizados ante la Organización Internacional para la Estandarización (ISO por las siglas en inglés de *International Organization for Standardization*) por lo que los cambios a estos lenguajes deben acordarse mediante un proceso de votación ante un comité formado por representantes de diversos países que son en su mayor parte voluntarios que realizan su labor con recursos propios. Además de que los cambios asociados a los lenguajes deben acompañarse con la publicación de estándares actualizados, el comité encargado debe tomar en cuenta la compatibllidad hacia atrás de los cambios que realicen; muchos de los miembros del comité son representantes de la industria e implementadores de compiladores por lo que las modificaciones hechas a los lenguajes también son analizadas desde el punto de vista comercial.

Los lenguajes auspiciados por corporaciones comerciales como Oracle y Microsoft en los casos de los lenguajes Java y C# [34] o bien aquellos que únicamente son guiados por la comunidad de usuarios como el caso de Python pueden evolucionar más rápido para atender las necesidades cambiantes de los programadores actuales. En contraste, tanto C como C++ han ido evolucionado lentamente desde la década de los 70 y la retroalimentación de los diseñadores

con los usuarios estaba limitada en gran medida por los medios de comunicación existentes en esa época, lo cual constituye una desventaja importante al compararlo con las herramientas con las que se disponen en la actualidad, como Internet.

Este capítulo describe la evolución tanto de C como de C++ mencionando sus metas de diseño, las características más importantes de ambos lenguajes, así como detallando los problemas que resultaron de procurar la compatibilidad hacia atrás con otros lenguajes o con código existente y de la falta de previsión al introducir nuevas características; esto último en gran parte debido a la larga evolución a la que han estado sujetos. Se explica brevemente el grado de avance de las próximas actualizaciones de los estándares correspondientes a ambos lenguajes, los cuales constituirán el estado que mantendrán éstos dentro de los próximos cinco a diez años.

Para las explicaciones técnicas se hablará sobre C++ y se hará explícito cuando se necesiten mencionar algunas de las diferencias que existen con C, dando por entendido que el resto de las características o son exclusivas de C++ o se comparten con C.

3.1 Antecedentes y metas de diseño de C

La explicación que se presenta a continuación está en su mayor parte condensada de [16].

El lenguaje de programación C es un lenguaje de programación de sistemas y fue diseñado e implementado por Dennis MacAlistair Ritchie en los Laboratorios Bell de la Bell Telephone Company (actualmente AT&T) principalmente en la década de los 70. A pesar de ser considerado actualmente como un ensamblador portable, ésta no fue una decisión de diseño puesto que la creación de C fue motivada explícitamente por la implementación del sistema operativo UNIX.

En los últimos años de la década de los 60, el proyecto Multics de los Laboratorios Bell, un sistema operativo innovador para la época, fue abandonado por considerar que finalizarlo tomaría demasiado tiempo a un costo muy alto. Poco antes de que el proyecto fuera abandonado, un grupo de investigadores comandado por Kenneth Lane Thompson comenzó un proyecto alternativo mucho más pequeño y por lo mismo, más viable, que retomaba las ideas más innovadoras de Multics.

Thompson diseñó el lenguaje de programación B, que es una versión simplificada del lenguaje de programación BCPL pero que podía ser implementado en una máquina PDP-7 de sólo 8KB de memoria con palabras de 18 bits, que es el hardware con el que contaba durante el inicio del desarrollo de su proyecto. A diferencia del compilador de BCPL que leía todo el código fuente en memoria para posteriormente emitir código nativo, las limitantes de almacenamiento que tuvo que enfrentar Thompson lo obligaron a que el lenguaje B tuviera que poderse analizar en una sola pasada emitiendo código tan pronto como fuera posible. La implementación de B fue un intérprete de código hilado.

Para 1970 el proyecto, que fue nombrado UNIX, era lo suficientemente prometedor como para que pudiera ser adquirida una nueva máquina DEC PDP-11 con 24KB de memoria y que, a diferencia de la máquina anterior, podía direccionar bytes en lugar de sólo palabras. Este cambio de máquina volvió inadecuado mucho del modelo de máquina manejado por el lenguaje B. Algunos de los cambios deseables eran la incorporación del manejo de bytes en el lenguaje y la preparación ante la inminente aparición de los primeros dispositivos de cálculo en punto flotante. Además se deseaba evitar la sobrecarga implicada por el hecho de que B fuera hasta ese momento interpretado en lugar de compilado.

En 1971 Dennis Ritchie extiende el lenguaje B para realizar los cambios mencionados, al mismo tiempo creando uno de los primeros compiladores que era capaz de generar código ensamblador tan eficiente como para poder competir con el código ensamblador escrito manualmente por un programador. Esta extensión de B, llamado "Nuevo B" fue bien recibida ya que se puso mucho énfasis en que la transición de B a este nuevo lenguaje fuera sencilla a pesar de los cambios semánticos requeridos. Después de la incorporación de éstos y otros cambios y de la implementación del nuevo compilador, Ritchie decidió nombrar a este nuevo lenguaje como "C".

Para 1973, C era suficientemente maduro y fue posible reescribir el kernel de Unix en C para la PDP-11. Lo exitoso de la tarea motivó al equipo de desarrollo a portar algunas herramientas de UNIX a otras plataformas, sin embargo el problema principal constituía no el cambio de plataforma sino la adaptación de las herramientas a otros sistemas operativos por lo que en 1978 se decidió portar UNIX a la computadora Interdata 8/32, modificando el lenguaje como conviniera para minimizar los problemas de compatibilidad. En ese mismo año Brian Wilson Kernighan y Dennis Ritchie publicaron el libro "El lenguaje de programación C" que sirvió como referencia del lenguaje hasta su estandarización.

Al lograr portar UNIX exitosamente a la nueva plataforma, otro equipo decidió hacer lo mismo para la DEC VAX 11/780, máquina que se volvió popular rápidamente y que contribuyó a la propagación de UNIX y de C. Durante los 80 el uso de C se extendió masivamente y empezaron a surgir compiladores para prácticamente todas las plataformas y sistemas operativos. El éxito de C y su uso en proyectos comerciales y gubernamentales fueron motivantes para iniciar su estandarización ante el Instituto Nacional Estadounidense de Estándares (ANSI por las siglas en inglés de *American National Standards Institute*) y posteriormente ante la ISO. La estandarización ante la ANSI se logró en 1989 [35] y en 1990 ante la ISO como el estándar ISO/IEC 9899-1990 [36]; el lenguaje descrito en este estándar es comúnmente denominado C89. El estándar de C fue actualizado en 1999 ante la ISO como el estándar ISO/IEC 9899:1999 [36], adoptado por la ANSI en el 2000 y se conoce como C99.

El comité encargado de la estandarización de C tiene como objetivo fundamental conservar el "espíritu de C" que puede resumirse en las siguientes frases [37]:

1. Confiar en el programador.
2. No evitar que el programador haga lo que necesita hacer.
3. Mantener el lenguaje pequeño y simple.
4. Proporcionar una única manera de realizar una operación.
5. Si algo puede implementarse como biblioteca, debe hacerse así.
6. La biblioteca estándar debe estar escrita en el propio lenguaje.
7. El soporte en tiempo de ejecución debe ser mínimo.
8. El lenguaje debe ser rápido, incluso si no es portable.

El último punto tiene que ver con permitir que la implementación controle ciertos aspectos de la semántica del lenguaje, como ocurre en el caso de algunos operadores, si esto permite que el código generado sea más eficiente. En muchos de estos casos, la semántica de algunas operaciones es la semántica que el hardware dicte para ellas. Esto ocurre, por ejemplo, en las operaciones de desplazamiento de bits pues el resultado de estas operaciones no está completamente especificado si alguno de los operandos es negativo o si el operando de la derecha representa un número de bits mayor al número de bits del operando de la izquierda.

3.2 Antecedentes y metas de diseño de C++

La explicación que se presenta a continuación está en su mayor parte condensada de [2].

Las experiencias que Stroustrup tuvo con los lenguajes Simula y BCPL durante el desarrollo de su tesis de doctorado fueron contrastantes. Debiendo implementar un sistema que simulara la ejecución de software sobre un sistema distribuido, eligió inicialmente utilizar el lenguaje de programación Simula 67. Aunque la experiencia de codificar el simulador en este lenguaje fue inspiradora por la elegancia de la implementación, el ejecutable resultante fue inutilizable y casi hizo fracasar el proyecto, que debía correr bajo un mainframe IBM 360/165. La razón de esto era la importante sobrecarga en tiempo de ejecución que era impuesta tanto por la implementación como por la especificación del lenguaje, que requería características como la verificación en tiempo de ejecución de acceso a arreglos o la inicialización automática de variables.

Para evitar fracasar en el proyecto, Stroustrup reimplementó el simulador en BCPL y el ejecutable resultante era suficientemente rápido como para obtener información útil que sería la base de su tesis. Sin embargo, BCPL es un lenguaje no tipado y demasiado cercano al ensamblador lo que, comparado con las facilidades de codificación de Simula, hizo que Stroustrup, una vez graduado, decidiera que era necesario crear una herramienta que combinara lo mejor de ambos lenguajes. En particular deseaba:

- Una herramienta que permitiera organizar el código de una aplicación de manera similar a Simula y que diera facilidades para crear aplicaciones grandes.
- Una herramienta que produzca código tan eficiente como BCPL y que permita combinar partes de un programa compiladas separadamente.
- Una herramienta disponible rápidamente y suficientemente sencilla como para poder portarla a otras plataformas de una manera relativamente fácil.

Después de incorporarse a los Laboratorios Bell en 1979, Stroustrup empezó con el desarrollo de C++ implementando un preprocesador que agregaba a C la característica de tipos jerárquicos (clases) definidos por el usuario de Simula. Este lenguaje fue llamado “C con clases” y tenía como meta específica permitir una mejor organización de programas, apoyándose en C para la parte de cálculo, manteniendo compatibilidad con él y sin incurrir en ninguna sobrecarga adicional en tiempo de ejecución o en memoria consumida.

La elección de ser compatible con otro lenguaje era importante para tener una herramienta útil en poco tiempo. La elección de C ha sido defendida por Stroustrup con lo siguientes argumentos:

- *C es flexible*: El lenguaje no tiene limitaciones inherentes que eviten usar alguna técnica de programación o utilizarlo en alguna área de aplicación.
- *C es eficiente*: Los conceptos manejados por C coinciden con los conceptos fundamentales de las computadoras de la época y es posible utilizar los recursos del hardware desde C.
- *C está disponible*: Existe un compilador de C de buena calidad para la gran mayoría de las plataformas.
- *C es portable*: Aunque la portabilidad no está garantizada completamente, portar una aplicación es generalmente viable.

El éxito de C con clases fue limitado, sin embargo, en lugar de abandonar el proyecto se decidió en 1982 crear un nuevo lenguaje que eventualmente fue llamado C++, más poderoso que el anterior y con un compilador propio llamado Cfront, aunque éste generaba código C en lugar de código nativo por lo que estrictamente era un traductor.

Los objetivos de diseño de C++ eran los siguientes:

- C++ debe ser un mejor C.
- C++ debe soportar la abstracción de datos.
- C++ debe soportar la programación orientada a objetos.

Las reglas de diseño que consideramos más importantes son las siguientes:

- C++ debe ser implementable con tecnología sencilla.
- No deben existir incompatibilidades gratuitas con C.
- No debe existir un lenguaje de más bajo nivel con excepción de ensamblador.
- No debe existir ninguna sobrecarga adicional por características que el usuario no usa.

Creemos que la última regla es la más importante pues pretende evitar que se repita la experiencia que Stroustrup tuvo con Simula: que existan características inherentes al lenguaje que impongan sobrecargas inevitables a todos los usuarios del mismo. Además, una característica no debe imponer mayor sobrecarga que la que el programador hubiera introducido si hubiera implementado dicha característica manualmente en C.

Las especificaciones de las sucesivas versiones lenguaje han sido presentadas en el libro "El lenguaje de programación C++" de Bjarne Stroustrup [38] siendo la primera versión publicada en 1985 (C++ 1.0). La versión de C++ con la que se iniciaron los trabajos de estandarización está documentada en el libro "El manual de referencia anotado de C++" [39] de 1990 y fue

implementada en Cfront en 1991. Esta versión es generalmente llamada "ARM C++" por *Annotated Reference Manual*.

Para la década de los 90 el éxito de C++ era indiscutible y ya existían compiladores comerciales y desarrollados independientemente. La iniciativa para estandarizar C++ fue presentada conjuntamente por IBM, Hewlett-Packard, AT&T y DEC en 1989. Existían numerosas preocupaciones que tuvieron que ser atendidas por el comité formado ya que hasta ese momento la comunidad de desarrolladores de C++ no estaba bien organizada. Algunas de ellas eran la falta de bibliotecas disponibles de manera generalizada, la compatibilidad con C que había sido recientemente estandarizado y la poca uniformidad de características que existía entre las diversas implementaciones disponibles. La estandarización duró casi una década y se logró en 1998 ante la ANSI y ante la ISO como el estándar ISO/IEC 14882: 9819 [40].

3.3 Los estándares C1X y C++0x

La tercera actualización al estándar de C, denotado como C1X, se espera esté lista para su publicación en 2012 [41]. Para esta nueva versión del lenguaje, el comité encargado se ha enfocado en lograr los siguientes objetivos:

- Mejorar el soporte para aplicaciones con más de un hilo de ejecución.
- Mejorar el soporte para Unicode, un estándar que unifica la codificación y representación de los juegos de caracteres existentes.
- Mejorar el soporte de algunas características de bajo nivel.
- Dar más facilidades para la codificación de aplicaciones seguras.
- Dar más facilidades para crear código genérico.

La segunda actualización al estándar de C++, denotado como C++0x, sin embargo, estaba planeada para su publicación en 2009 [42, 43]. Los objetivos de C++0x son:

- Mejorar el soporte para aplicaciones con más de un hilo de ejecución.
- Mejorar el soporte para Unicode, un estándar que unifica la codificación y representación de los juegos de caracteres existentes.
- Mejorar el soporte de algunas características de bajo nivel.
- Generalizar características que permitan eliminar casos especiales y den mayor flexibilidad al lenguaje.
- Generalizar características que permitan conseguir mayor rendimiento.
- Incrementar la seguridad del sistema de tipos y dar mayores facilidades a los programadores principantes.

Durante los trabajos de estandarización fue cada vez más claro que la fecha límite no se cumpliría, por lo que se hicieron una serie de compromisos para reducir los puntos que abarcaría el nuevo estándar y lograr cumplir con los tiempos establecidos. A pesar de lo anterior existieron numerosos contratiempos, muchos de ellos por problemas técnicos en varias de las características ya incluídas en el texto de trabajo, que fueron retrasando cada vez más la planificación. El borrador final de comité (FCD por las siglas en inglés de *Final Committee Draft*) fue publicado en marzo de 2010 y se espera que la publicación del nuevo estándar se haga a finales de 2011 o principios de 2012 [44].

3.4 Problemas e inconvenientes de C++

A continuación se presenta un resumen de las debilidades más importantes de C++ tanto sintácticas como semánticas y de implementación. Se asume cierta familiaridad con C y C++ por lo que no se presenta un tutorial del lenguaje aunque se dan explicaciones breves para entender el contexto de las características descritas y se presentará el nivel de detalle suficiente para su comprensión. Si el lector lo estima necesario, le recomendamos consultar [38].

Se inicia con la parte más básica del lenguaje e incluyen algunas de las características más relevantes de las ya incorporadas en el borrador final de comité de C++0x. 'Conceptos', una de las características rechazadas de último momento para C++0x, se describe en el capítulo 4 debido a su importancia y a que se planea su inclusión para el estándar siguiente al de C++0x.

3.4.1 Tipos y declaraciones

El lenguaje C++ proporciona una serie de tipos predefinidos, llamados tipos primitivos, que generalmente corresponden a los tipos que participan en las instrucciones del procesador. Estos tipos se clasifican en los siguientes grupos:

Enteros

`short int`	Rango garantizado de $-2^{15}-1$ a $2^{15}-1$.
`int`	Rango garantizado de $-2^{15}-1$ a $2^{15}-1$.
`long int`	Rango garantizado de $-2^{31}-1$ a $2^{31}-1$.
`long long int`	Rango garantizado de $-2^{63}-1$ a $2^{63}-1$.
`unsigned short int`	Rango garantizado de 0 a $2^{16}-1$.
`unsigned int`	Rango garantizado de 0 a $2^{16}-1$.
`unsigned long int`	Rango garantizado de 0 a $2^{32}-1$.

`unsigned long long int`	Rango garantizado de 0 a 2^{64}-1.

Con respecto a los rangos y al consumo de memoria de los tipos anteriores, se debe cumplir la siguiente relación tanto para los enteros con signo como para los enteros sin signo correspondientes:

`short int` ≤ `int` ≤ `long int` ≤ `long long int`

Debido a las pocas garantías que se dan, el uso de estos tipos constituye un problema para el desarrollo de código portable. El estándar C99 corrige este problema incluyendo sinónimos para tipos enteros específicos como `int32_t` para un entero de 32 bits con signo y `uint64_t` para un entero de 64 bits sin signo.

Las literales enteras tienen por defecto el tipo `int` pero puede controlarse el tipo de la literal especificando los sufijos `L` para `long`, `LL` para `long long`, `UL` para `unsigned long` y `ULL` para `unsigned long long`. Existen casos en el que es obligatorio modificar el tipo de la literal por lo que los sufijos no son sólo una comodidad:

```
long long n = 34359738368;
// error, literal demasiado grande para int

long long m = 34359738368LL;
// correcto
```

Las literales enteras también pueden escribirse en hexadecimal o en octal, con los prefijos `0x` y `0` respectivamente. El prefijo elegido para indicar literales en base octal puede prestarse a confusión fácilmente. Además, para las literales en hexadecimal y en octal el compilador elige automáticamente el tipo de la literal siendo `int` por defecto, o el tipo más pequeño mayor a `int` que pueda representar el valor de la literal. Esto puede dar lugar a sorpresas, `0xFFFF` es -1 en máquinas con `int` de 16 bits y 65535 para máquinas con `int` de 32 bits o más.

Flotantes

`float`	Corresponde al formato sencillo de IEC 60559.
`double`	Corresponde al formato doble de IEC 60559.
`long double`	Corresponde al formato extendido de IEC 60559, a algún otro formato extendido con al menos tanta precisión como el formato doble de IEC 60559 o a dicho formato doble.

El estándar IEC 60559 corresponde al estándar para aritmética binaria en punto flotante de la IEEE cuya última versión es la IEEE 754−2008 [45]. Sin embargo los últimos bocetos de los estándares tanto de C como de C++ aún no hacen referencia a esta versión [46, 47]. Una cláusula relevante del estándar de 2008 habla sobre la reproducibilidad de resultados y está dirigida a los diseñadores de lenguajes de programación.

El tipo por defecto de las expresiones con operandos de punto flotante es `double`, incluyendo a las literales de números flotantes. De manera similar a los sufijos de literales enteras, existe el sufijo `F` para literales de tipo `float` y el sufijo `L` para literales de tipo `long float`. Sin embargo el tipo `double` no siempre es el tipo natural de la unidad de procesamiento de punto flotante del procesador (FPU por las sigles en inglés de *floating point unit*). Las plataformas de Intel posteriores a la x87 tienen como tipo natural del FPU un tipo extendido de 80 bits mientras que el tipo `double` es de 64 bits. Peor aún, en este FPU los cálculos siempre se realizan con operandos en precisión extendida [48]. Ya que el compilador está obligado a almacenar las variables de tipo `double` en memoria con una precisión que es inherentemente menor, es probable que el siguiente fragmento de programa imprima `"error"`:

```
double a = aleatorio( );
double b = a;

// no se lee ni se modifica a
// no se modifica b pero se lee frecuentemente

if (a != b)
    printf("error");
```

Si como optimización el compilador mantuvo a la variable `b` en uno de los registros de 80 bits pero almacenó a la variable `a` en memoria, los bits perdidos en esta última operación causarán que la condición sea verdadera cuando se vuelva a cargar la variable `a` ya con cierta pérdida de precisión en uno de los registros de 80 bits para efectuar la comparación con `b`.

Caracteres

`char`	Corresponde a `signed char` o a `unsigned char`.
`wchar_t`	Corresponde a un tipo caracter ancho.
`signed char`	Corresponde a 1 byte, rango mínimo de -2^7-1 a 2^7-1.
`unsigned char`	Corresponde a 1 byte, rango mínimo de 0 a 2^8-1.

Un byte corresponde a un octeto (8 bits) por lo que se usarán ambos términos indistintamente. Los caracteres anchos dependen de las propiedades de la localidad activa y del soporte para internacionalización de la plataforma.

Aunque menos problemático que con los tipos enteros, el hecho de que la representación real del tipo char pueda cambiar entre plataformas constituye un problema de portabilidad por lo puede convenir especificar siempre la presencia o no de signo. Sin embargo, char es considerado un tipo diferente tanto a `signed char` como a `unsigned char` por lo que las bibliotecas que usen el tipo `char` directamente pueden ser incompatibles con el uso de los tipos más específicos.

Las literales de caracter se especifican entre comillas simples y puede especificarse el valor entero en hexadecimal del caracter deseado según corresponda con la plataforma y la localidad.

```
char c = '@';
char d = '\x40';
```

Misceláneos

`bool`	Ocupa el espacio de un char. Almacena 0 ó 1.
`void`	Tipo incompleto que denota ausencia de un valor.

Por definición, en C++ el tamaño mínimo de un objeto en memoria es de 1 byte. Si bien es exagerado utilizar un byte para almacenar una variable booleana, la razón de que esto sea así en C++ tiene que ver con el hecho de que la gran mayoría de las plataformas actuales de computadoras sean incapaces de direccionar bits individuales. Las literales booleanas son las palabras reservadas `true` y `false`.

No es posible declarar una variable de un tipo incompleto. La prohibición de tener objetos que conceptualmente son de tamaño 0 tiene como justificación garantizar que objetos diferentes tengan direcciones diferentes. Sin embargo, esto trae consecuencias negativas como se explicará a continuación.

3.4.2 Estructuras, apuntadores, constantes y arreglos

Estructuras

C++ permite definir nuevos tipos de dato que son descritos como una agrupación de tipos existentes. A estos tipos se les denomina estructuras y se declaran con la palabra reservada `struct`. Un tipo `fecha` podría definirse y usarse como en el siguiente ejemplo:

```
struct fecha {
    int dia;
```

```
    int mes;
    int anyo;
};

fecha f;
f.dia = 5;
```

Sin embargo el tamaño de una estructura no siempre corresponde a la suma del tamaño de sus miembros. Además de un tamaño, los tipos primitivos tienen restricciones sobre las posiciones de memoria en las que pueden ser almacenados. La alineación corresponde al número mínimo de bytes que debe existir entre dos direcciones donde sea válido almacenar variables de cierto tipo. La alineación es obligatoria para ciertas plataformas y esto tiene que ver con la manera en que el procesador lee los datos desde memoria; para un procesador que lee únicamente palabras completas desde memoria comenzando por la palabra en la dirección 0 puede ser problemático si un `int` está almacenado de modo de que esté traslapado en dos palabras diferentes. Los procesadores de Intel más modernos pueden manejar muchos de los accesos desalineados pero requieren más de un acceso a memoria en dichos casos, lo que alenta la ejecución de un programa. Por lo anterior, generalmente las variables sólo son almacenadas en posiciones de memoria que cumplan con las restricciones de alineación aunque esto implique dejar algunos bytes sin utilizar (que se denominan relleno o *padding* en inglés).

Al definir una estructura, el compilador puede necesitar insertar campos de relleno entre los miembros de la misma para cumplir con los requerimientos de alineación. Por ejemplo considérese los requerimientos de alineación de la plataforma Intel 64 [48] y la definición de la siguiente estructura:

```
struct par {
    char a;             // tamaño: 1 byte, alineación: 1 byte
    double b;           // tamaño: 8 bytes, alineación: 8 bytes
};
```

El compilador deberá ajustar la estructura de modo de que las variables cumplan los requerimientos de alineación por lo que un `par` en memoria se verá de la siguiente manera:

```
struct par {            // tamaño: 16 bytes, alineación: 8 bytes
    char a;             // tamaño: 1 byte, alineación: 1 byte
    char _relleno[7];   // relleno para alinear b
    double b;           // tamaño: 8 bytes, alineación: 8 bytes
};
```

Para este caso lo mismo aplicaría aunque los miembros del `par` estuvieran declarados en orden inverso pues sería necesario insertar un relleno al final de la estructura. Esto es debido a que se debe garantizar la correcta

alineación de todos los miembros de un arreglo que almacene estructuras par. Dicho lo anterior, se puede ver que la regla que obliga al compilador a asignar un byte para tipos que de otro modo tendrían tamaño 0 puede aumentar el tamaño de una estructura debido a los requerimientos de alineación.

Apuntadores

Un apuntador es una variable que almacena un entero que puede ser interpretado como una dirección de memoria. Durante la declaración de una variable se puede utilizar el modificador `*` para indicar que la variable es un apuntador a una variable del tipo indicado. Sintácticamente es un operador prefijo sobre el identificador de la variable, sin embargo C es libre de formato por lo que la ubicación del operador puede dar lugar a confusión:

```
int* ap0;               // ap0 es un apuntador a int
int* ap1, ap2;          // ap2 es un int, no un apuntador
int *ap3;               // ap3 es un apuntador a int
int *ap4, *ap5;         // ap4 y ap5 son apuntadores a int
```

Aunque históricamente no existían literales de tipo apuntador, se puede utilizar la literal `0` para indicar el apuntador nulo. Esto puede confundir al compilador en ciertos contextos (por ejemplo al llamar funciones sobrecargadas) pues la conversión de la literal `0` a un apuntador nulo sólo se mantiene en la expresión inmediata; después adquiere el tipo `int`.

El operador `&` prefijo se usa para obtener la dirección de una variable pero no se puede tomar la dirección de un resultado temporal pues en la máquina abstracta definida por C los temporales no tienen dirección.

```
int n;
int* q = &(n + n);      // error, dirección de un temporal
int* p = &n;            // ok, apuntador a int
```

Debido a lo anterior, tampoco se le puede asignar valor a un temporal. En terminología de C y C++ a una variable temporal se le denomina valor-d (valor-derecho o *rvalue* en inglés, por sólo poder participar del lado derecho de una asignación). Al resto se les denomina valores-i (valor-izquierdo o *lvalue* en inglés, por poder participar del lado izquierdo de una asignación).

No existen variables de tipo `void` pero sí apuntadores a `void` (`void*`) que comúnmente se usan como apuntadores a variables de tipo desconocido.

En C++0x se proporciona la palabra reservada `nullptr` para denotar un apuntador nulo. El tipo de `nullptr` no es un apuntador, es un tipo sintetizado por el compilador y diferente a cualquiera de los otros tipos primitivos. Se

puede obtener el tipo de `nullptr` aplicando el operador `decltype` que devuelve el tipo de una expresión.

Constantes

Una constante es una variable que no puede ser modificada una vez inicializada. Para declarar una constante se utiliza la palabra reservada `const`.

Es posible declarar campos de estructuras como `const` pero deben inicializarse directamente al momento de declarar la variable utilizando una lista de inicialización, que es una lista de expresiones delimitada por llaves. De otro modo la variable no podrá ser inicializada posteriormente:

```
struct s {
    const int n;
};

s v0;
v0.n = 5;              // error, n es constante
s v1 = { 5 };          // ok
```

La palabra `const` es más que un modificador que restringe las operaciones que pueden hacerse sobre una variable, ésta crea un tipo de dato diferente. Las consecuencias de esto son varias y se hablará de ellas conforme se presenten las características que interactúan de manera problemática con esta decisión de diseño.

El modificador `const` se propaga al tomar la dirección de una variable. Es un error tomar la dirección de una variable `const` y almacenarla en un apuntador que no respete `const`. También se pueden declarar apuntadores `const`, que son apuntadores que no puede asignarse una vez inicializados.

```
const int n = 5;
const int *p0 = &n;     // ok, apuntador a const int

int m, q;
int *const p1 = &m;     // ok, apuntador const a int

p1 = &q;                // error, no se puede asignar
```

Las declaraciones en C fueron pensadas para imitar el uso de las variables en expresiones:

```
char *a;
// *a devuelve un char

const char **c;
// *c devuelve un const char*
```

```
// **c devuelve un const char
```

Sin embargo esto se vuelve inmanejable para declaraciones complicadas y es una una fuente constante de problemas para los programadores principiantes. A primera vista las primeras dos líneas del siguiente fragmento de código parecen válidas, sin embargo no lo son; si lo fueran se podría violar el sistema de tipos:

```
char* b;
const char** c = &b;

const char a = '@';
*c = &a;
*b = '1';              // modificación indirecta de a
```

Denotando a `T` como algún tipo arbitrario, la conversión de un apuntador a `T` hacia un apuntador a `const T` es segura. Sin embargo, esa no es la conversión que se está llevando a cabo en el ejemplo anterior. Si reescribimos las declaraciones confusas con una sintaxis más explícita esto es evidente:

```
apuntador(char) b;
apuntador(apuntador(char)) q = &b;
apuntador(apuntador(const char)) c = q;

// para T = apuntador(char)
// el tipo de q es de la forma apuntador(T)
// el tipo de c no es de la forma apuntador(const T)
```

Aunque tanto C como C++ tienen el modificador `const`, éste actúa de manera diferente en ambos lenguajes. En C++ el compilador puede poner a una variable declarada originalmente como `const` en un área de memoria con permisos de sólo lectura por lo que quitar el modificador `const` mediante un moldeado o *cast* puede causar comportamiento indefinido. En C el compilador no puede tomarse esta libertad.

Arreglos

Un arreglo es un agregado homogéneo de variables donde cada miembro se denota por su posición en el arreglo. Un arreglo tiene un tamaño conocido de antemano por el compilador pero puede inferirse si se usa una lista de inicialización.

```
int arr0[3];
int arr1[] = { 1, 2, 3 };     // arr1 tiene tamaño 3
int arr2[aleatorio( )];       // error en C++
```

C99 permite declarar arreglos de tamaño no conocido hasta tiempo de ejecución. Sin embargo esto obliga a relegar ciertas garantías. Por ejemplo, el operador `sizeof` regresa el tamaño que ocupa un tipo y tradicionalmente se evalúa en tiempo de compilación. Sin embargo, en C99 `sizeof` debe evaluarse en tiempo de ejecución al aplicarse sobre arreglos de tamaño desconocido. Además resulta en comportamiento indefinido si la expresión que denota el tamaño del arreglo es un valor negativo, situación que el compilador puede prevenir en C89 y en C++.

Un arreglo sólo puede inicializarse mediante una lista de inicialización y no puede asignarse. Se prohiben los arreglos de tamaño 0. Además, los arreglos tienen una conversión implícita a apuntador:

```
int arr2[] = arr1;          // error
int* arr3 = arr0;           // ok, conversión implícita
```

La conversión implícita a apuntador surgió de la necesidad de compatibilidad hacia atrás con los lenguajes B y BCPL pues en estos lenguajes un arreglo era un apuntador que contenía la dirección de un bloque de memoria asignado en algún otro lugar. Dennis Ritchie sin embargo deseaba eliminar la sobrecarga de dicho apuntador extra pues idealmente una estructura compuesta por un arreglo de cuatro `int`s debía ser equivalente en consumo de memoria a una estructura compuesta por cuatro campos de tipo `int`. Aunque en el contexto histórico fue una decisión correcta, en la actualidad es considerada como una decisión de diseño muy problemática en ciertos contextos, especialmente en el uso de funciones. De esto se hablará posteriormente.

Las literales de cadena, que son delimitadas por comillas dobles, pueden usarse para inicializar arreglos de caracteres. Estas literales tienen un caracter nulo (con valor entero 0) implícito al final y puede confundir:

```
char c[4] = "hola";         // error, arreglo demasiado chico
char d[5] = "hola";         // ok
```

El tipo real de las literales de cadena es `const char*` pero se permite su asignación a variables `char*` por compatibilidad hacia atrás con código escrito antes de que existiera `const`. Sin embargo resulta en comportamiento indefinido intentar modificar alguno de los caracteres de la literal y es un error frecuente entre programadores principiantes.

```
char* c = "hola";
c[0] = 'g';             // comportamiento indefinido
```

3.4.3 Expresiones y sentencias

Una expresión es una combinación de literales, variables, constantes, operadores y llamadas a función que es evaluada de acuerdo a un conjunto de reglas y que efectúa un cálculo. A diferencia de muchos otros lenguajes, C++ da pocas garantías sobre la manera en que las expresiones son evaluadas y el comportamiento es indefinido en casos como los desbordamientos aritméticos y las divisiones entre cero.

Orden de evaluación

En términos generales el orden de evaluación de las expresiones está definido por la implementación. Sin embargo se define como punto de secuencia el punto donde se garantiza que todos los efectos secundarios de las expresiones anteriores a él ya se han llevado a cabo mientras que ningún efecto secundario de las expresiones posteriores ha ocurrido. Los puntos de secuencia son:

- El final del primer operando de los siguientes operadores: AND logico `&&`, OR lógico `||`, condicional `?`, operador coma `,`.
- El final de una declaración.
- El final de una expresión completa: lista de inicialización, expresiones controladoras de una sentencia de flujo, expresión de la sentencia `return` y expresión delimitada por un punto y coma `;`.
- Antes de llamar a una función; esto con respecto a las expresiones usadas como parámetros.

Un ejemplo común para ilustrar los problemas que pueden ocurrir si no se toman en cuenta las garantías de evaluación es el siguiente:

```
int i = 1;
int arr[2];

arr[i] = i++;
```

Ya que no se dan garantías de en qué momento se incrementa `i`, es posible que el programa anterior intente escribir en una posición fuera del arreglo, lo que podría ocasionar que el programa termine abruptamente. Dar libertades sobre el orden de evaluación, por otro lado, puede generar código más eficiente en algunas ocasiones.

Operadores básicos

La siguiente tabla muestra la lista de los operadores comunes de C y C++. Están agrupados por orden de precedencia, de la más alta a la más baja.

Operador	Descripción	Asociatividad

`()`
`[]`
`.`
`->`
`++ --`
	Llamada a función
	Acceso a arreglo
	Selección de miembro
	Selección de miembro mediante apuntador
	Incremento/decremento posfijos
		Izquierda

`++ --`
`+ -`
`! ~`
`(`*tipo*`)`
`*`
`&`
`sizeof`
	Incremento/decremento prefijos
	Más/menos prefijos
	Negación lógica/binaria
	Moldeado
	Dereferencia
	Dirección
	Tamaño en bytes
		Derecha

`* / %`	Multiplicación/división/módulo	Izquierda
`+ -`	Suma/resta	Izquierda
`<< >>`	Desplazamiento izquierdo/derecho	Izquierda
`< <=` `> >=`	Menor/menor o igual Mayor/mayor o igual	Izquierda
`== !=`	Igual/diferente	Izquierda
`&`	AND binario	Izquierda
`^`	OR exclusivo binario	Izquierda
`|`	OR inclusivo binario	Izquierda
`&&`	AND lógico	Izquierda
`||`	OR Lógico	Izquierda
`?:`	Condición ternaria	Derecha

`=`
`+= -=`
`*= /=`
`%= &=`
`^= |=`
`<<= >>=`
	Asignación
	Suma/resta con asignación
	Multiplicación/división con asignación
	Módulo/AND binario con asignación
	OR exclusivo/inclusivo con asignación
	Desplazamiento izquierdo/derecho con asignación
		Derecha

`,`	Coma (separación manual de expresiones)	Izquierda

La precedencia de los operadores binarios `&` y `|` son resultado de un accidente histórico [16]. Ésta implica que el siguiente fragmento de código:

```
if (a & b == 5) {
   //...
}
```

se interpreta realmente como

```
if (a & (b == 5)) {
   //...
}
```

El significado de los operadores `&` y `|` eran dependientes del contexto en B y BCPL. Mientras que en operaciones aritméticas tenían la semántica usual, en expresiones condicionales actuaban como operadores en corto circuito: para el operador `&` su segundo operando no era evaluado si el primero evaluaba como 0 y para el operador `|` su segundo operando no era evaluado si el primero evaluaba como 1. Es decir, estos operadores también eran usados como conectivas lógicas y su nivel de precedencia original así los evidenciaba.

Posteriormente Dennis Ritchie decidió evitar confusiones e introdujo los operadores `&&` y `||` como operadores lógicos en corto circuito. Sin embargo para facilitar la transición de B a C mantuvo la precedencia de los operadores `&` y `|`. Esto es reconocido por él como un error de diseño pues las precedencias actuales son fuente de muchas sorpresas.

Aunque es de esperarse que un lenguaje de programación de sistemas no tenga verificaciones implícitas en tiempo de ejecución para prevenir operaciones que resulten en desbordamientos aritméticos o divisiones entre cero, en C y C++ el resultado de algunas operaciones han sido históricamente dependientes de la plataforma aún en casos habituales. Tanto C89 como C++ dejan como dependiente de la implementación el resultado de la operación `a % b` si alguno de los operandos es negativo y sólo se pide que se cumpla la relación `a == (a / b) * b + a % b` [35, 47]. Por otra parte, C99 requiere que la división redondee hacia cero y que el signo del residuo sea igual que el signo del dividendo [46] lo que corresponde con el cálculo efectuado por la instrucción IDIV de la plataforma x86 [49].

Como ya se mencionó, el resultado de las operaciones de desplazamiento a nivel de bit no está completamente especificado: si el operando de la derecha es negativo o si representa un número de bits mayor al número de bits del operando de la izquierda el resultado es indefinido; si el operando de la izquierda es negativo y se efectúa un desplazamiento a la derecha entonces el comportamiento es definido por la implementación y generalmente se hace un desplazamiento con extensión de signo que causa que el bit del signo se propague a la derecha; los demás casos se calculan de la manera esperada.

Por último, las expresiones de la forma `a < b < c` no se evalúan como normalmente se intuye en un contexto matemático sino que se evalúan como `(a < b) < c` que difícilmente es lo que un programador principante espera. El

problema se complica pues usualmente la expresión está bien formada y el compilador no emite ninguna advertencia.

Una nota de implementación: el análisis de la precedencia de operadores hecho por un analizador sintáctico guiado por una gramática puede requerir una llamada a función por cada nivel de precedencia. Ya que C++ tiene muchos niveles de precedencia, la implementación más sencilla requeriría catorce llamadas anidadas a función para analizar sintácticamente una asignación. Por supuesto que existen maneras más eficientes de realizar dicha tarea aunque eso requiere codificar esa parte del analizador manualmente.

Conversiones implícitas y explícitas

Debido a los orígenes no tipados de C, una operación entre variables de diferentes tipos estaba definida simplemente si la operación estaba definida en el hardware. Por ejemplo, un entero con signo puede convertirse a un entero sin signo mediante una asignación ordinaria. Desafortunadamente muchas de esas conversiones implican pérdida de precisión como en la conversión de un `double` a un `int` y se pueden manifestar problemas semánticos:

```
int a = 100000;
short b = a;

if (a != b) {
    // cierto para ints de 32 bits y shorts de 16 bits
    print("error");
}
```

Otro error muy frecuente es causado por la similitud entre el operador de asignación = y el operador de igualdad relacional == y que es agravado por el hecho de que un tipo entero puede convertirse implícitamente a booleano dentro de una condición:

```
int a = 2;

if (a = 3) {      // probablemente error de codificación
      //...
}
```

En C++ existen en total seis maneras diferentes de hacer un moldeado explícito. A continuación se describen brevemente:

- `(tipo)expresión`: la sintaxis de C. El lenguaje establece para qué tipos involucrados está definido el moldeado y la semántica de cada caso. Por ejemplo `(int)3.14` trunca,`(long)-5` adapta la literal a un tipo más ancho y `(void*)0` reinterpreta el patrón binario del operando.

- `tipo(expresión)`: notación funcional que semánticamente es idéntica a la anterior. Sin embargo tiene un defecto importante pues el tipo debe ser léxicamente un token individual; tanto `long long(5)` como `void*(0)` son errores de sintaxis.
- `static_cast<tipo>(expresión)`: incluye el subconjunto de conversiones definidas entre tipos relacionados como enteros con flotantes, caracteres con enteros y diferentes tipos de apuntadores.
- `const_cast<tipo>(expresión)`: permite agregar o quitar `const`.
- `reinterpret_cast<tipo>(expresión)`: reinterpreta el patrón binario del operando sin importar que la conversión pudiera tener una semántica alternativa conocida por el compilador.
- `dynamic_cast<tipo>(expresión)`: moldeado verificado en tiempo de ejecución entre apuntadores o referencias a variables de tipos relacionados jerárquicamente.

Los dos primeros tipos de moldeado son los más comunes pero también los más problemáticos pues con ellos es difícil escribir código genérico que tenga una semántica predecible para cualquier pareja de tipos. Sin embargo son los más sencillos sintácticamente hablando.

Sentencias de control

Las sentencias encontradas en C++ son la usuales:

Sentencia	Descripción

`if (condición) {`
`  ...`
`}`
`else if (condición) {`
`  ...`
`}`
`else {`
`  ...`
`}`
	Condicional simple

`while (condición) {`
`  ...`
`}`
	Ciclo con condicional verificada al inicio de cada iteración

`do {`
`  ...`
`} while (condición);`
	Ciclo con condicional verificada al final de cada iteración

`for (inicializacion;`
`  condición;`
`  actualización) {`
`  ...`
`}`
	Ciclo con actualización

`switch (expresión) {` `case expresión:`	Condicional simple mediante

... case expresión: ... default: ... }	elección de casos
break;	Terminación de ciclo o de caso
continue;	Terminación de iteración
goto etiqueta;	Salto incondicional
return expresión;	Retorno de función

Cuando dentro del bloque de instrucciones de alguna de las sentencias condicionales o de iteración sólo hay una instrucción, las llaves pueden omitirse:

```
if (aleatorio( ) == 2)
    print("par");
```

Aunque esto puede resultar conveniente en ciertos casos y es aparentemente inofensivo, tiene varios aspectos negativos. Uno relativamente fácil de evitar está relacionada con el formato que se le da al código:

```
if (aleatorio( ) == 2)
    print("hola");
    print("mundo");
```

La instrucción `print("mundo")` se ejecutará siempre; al no haber llaves que delimiten el bloque de la condicional sólo la primera instrucción forma parte de ella. Otro problema del mismo estilo es el siguiente:

```
if (aleatorio( ) == 2); {
    print("hola");
}
```

La única sentencia afectada por la condicional es la sentencia vacía que está denotada por un ; sencillo. Generalmente lo anterior es un simple problema de codificación aunque el compilador no considera esto un error.

A pesar que los inconvenientes anteriores puedan ser considerados como poco importantes, la comodidad de omitir las llaves vuelve imposible [50] el análisis sintáctico del lenguaje con analizadores LL(k) y LR(k) ya que se genera la llamada ambigüedad del `else` colgante en la que no se sabe con certeza a cuál `if` corresponde un `else`. El siguiente fragmento de código tiene dos posibles árboles de sintaxis bajo la gramática dada por lo que la gramática es ambigua:

```
if (aleatorio( ) == 2)
if (aleatorio( ) == 3)
print("hola");
else
print("adios");
```

sentencia =
sentencia_if | sentencia_otra ;

sentencia_if =
"if" condición sentencia |
"if" condición sentencia "else" sentencia ;

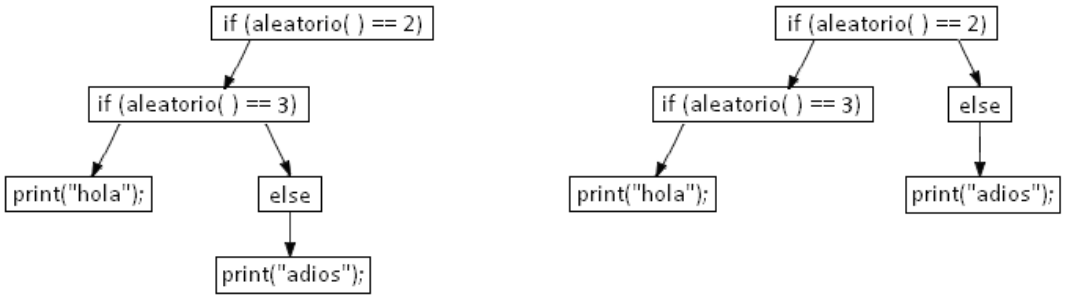


Figura 2. Árboles de sintaxis distintos

A pesar de lo anterior, los lenguajes con esta ambigüedad siguen siendo libres de contexto. No obstante la ambigüedad debe tratarse manualmente o debe abordarse con técnicas de análisis sintáctico más poderosas. Consideramos que la conveniencia de omitir las llaves no justifica las dificultades generadas.

El punto y coma al final de una sentencia `do` es olvidado frecuentemente. Probablemente se deba a que el paréntesis de cierre de la condición es suficiente para desambiguar la sintaxis.

El comportamiento de los `cases` en un `switch` es el de ejecución en cascada por lo que es obligatorio usar sentencias `break` para obtener el comportamiento usualmente deseado. La ejecución en cascada también impide declarar variables locales dentro de los `cases` a no ser de que estén delimitadas por un bloque. Mientras que tanto `break` como `continue` tienen una semántica dentro de un ciclo, sólo `break` lo tiene dentro de una sentencia `switch` lo que es inconsistente:

```
switch (1) {
case 1:
    print(“1”);
case 2:
    print(“2”);
    break;
case 3:
    // int n;     no permitido, el case 4 la omitiría
    // continue;  error de sintaxis si no hay un ciclo
    {
       int n;         // ok, variable en bloque anónimo
    }
case 4:
    break;
}

// imprime 12
```

Los bloques anónimos tienen la desafortunada consecuencia de obligar a que una llave al inicio de una instrucción siempre abra un bloque anónimo mientras que una llave dentro de una expresión es una lista de inicialización:

```
int arr[2] = { 1, 2 };        // lista de inicialización

{ 1, 2 }[0] + 5;              // error, llave abre bloque
({ 1, 2 })[0] + 5;            // ok
```

C++0x define el tipo de una lista de inicialización homogénea como `initializer_list<T>` donde `T` es el tipo de los elementos de la lista por lo que una lista de inicialización es una expresión. El problema mencionado anteriormente implica que aunque `A + { B, C }` sea una expresión bien formada es posible que `{ B, C } + A` sea un error de sintaxis.

El tipo `initializer_list<T>` extrañamente no es un tipo primitivo. Es un tipo de biblioteca que tiene privilegios especiales por parte del compilador. En versiones preliminares del borrador de C++0x, el uso de las listas de inicialización como expresiones requería incluir la biblioteca correspondiente.

3.4.4 Referencias y funciones

Referencias

Una referencia es un alias de una variable y se declara con el operador `&` que es un prefijo sobre el identificador de la variable. Una referencia debe inicializarse forzosamente con un valor-i. El tipo de una referencia a un valor-i de tipo `T` es `T&` por lo que al igual que `const`, una referencia es un tipo distinto.

Es inválido declarar un arreglo de referencias o un apuntador a referencia, sin embargo es posible declarar una referencia como miembro de una estructura y declarar un arreglo o apuntador a ésta. Aplicarle `sizeof` a dicha estructura devuelve un tamaño dependiente de la implementación lo cual es inconsistente con el comportamiento usual de `sizeof` para referencias:

```
struct s { double& r; };

print(sizeof(double&));
// imprime sizeof(double), el tipo referido
print(sizeof(s));
// dependiente de la implementación, usualmente sizeof(void*)
```

Ya que tanto `const` como las referencias crean tipos nuevos es sintácticamente posible crear referencias `const` y referencias a `const`. Sin embargo las referencias `const` no tienen sentido pues aún sin ser `const` una

referencia no puede referir a una variable distinta una vez inicializada. La palabra reservada `typedef` introduce un nombre alternativo para un tipo y por lo mismo también funciona con referencias. Esto permite crear sintácticamente referencias a referencias. Antes de la revisión de 2003 del estándar de C++ eran un error pero ahora colapsan a una referencia simple [51]:

```
typedef int& T;
T& a;                   // colapso de referencia a referencia
```

Un `int&&` no es una referencia a referencia, es parte de una característica de C++0x de la que se hablará posteriormente. Como un caso especial se permite inicializar una referencia a `const` con un valor-d. El valor-d es transformado internamente en un valor-i y es destruído cuando la referencia sale del ámbito de su declaración.

```
int& a = 5;             // error
const int& b = 5;       // ok, valor-d convertido a valor-i
```

Esta inconsistencia aparentemente menor causó dificultades importantes al diseñar la característica de C++0x mencionada arriba.

Funciones

Una función es una porción de código que realiza una tarea específica y que puede ser invocada múltiples veces durante un mismo programa. Una función puede recibir parámetros y devolver un valor que represente el cálculo hecho. El método por defecto para pasar un parámetro y para regresar un valor es llamado paso por valor e implica pasar una copia de los argumentos usados en la invocación de la función y devolver una copia del valor de retorno. Se puede pasar y devolver por referencia usando referencias o simulándolo mediante apuntadores.

Ya que tanto en B como en BCPL un arreglo era un apuntador y en C existe la conversión implícita correspondiente, no es posible pasar o regresar un arreglo por valor:

```
void g(int arr[2])      // equivalente a void g(int* arr)
{
    arr[0] = 0;
}
int arr[] = { 1, 1 };

g(arr);
print(arr[0]);          // imprime 0
```

En el paso de arreglos multidimensionales (matrices) es obligatorio indicar el tamaño de todas las dimensiones excepto la primera, a pesar de que esto no es obligatorio al momento de declararla como variable local. Esto se debe a que la información sobre las dimensiones de la matriz se pierde en la conversión implícita a apuntador y a que una matriz es realmente un arreglo en memoria donde la matriz es simulada mediante cálculos aritméticos para los desplazamientos:

```
void f(int mat[][5][5], int i, int j, int k)
{
    mat[i][j][k] = 0;
}

// debe transformarse internamente a

void __f_int___5_5(int* mat, int i, int j, int k)
{
    mat[i * 25 + j * 5 + k] = 0;
}
```

Las funciones pueden sobrecargarse, es decir, declarar varias funciones con parámetros diferentes pero con el mismo nombre. El compilador elegirá la función más adecuada. Sin embargo existen ciertas combinaciones prohibidas:

```
void f(int n);               // función #1
void f(double d);            // función #2
void g(int);                 // función #3
void g(const int&);          // función #4

f(5);       // llama a la función #1
f(5.5);     // llama a la función #2

const int n = 55;
g(5);       // llamada ambigua, valor-d ⟶ const int&
g(n);       // llamada ambigua
```

No es posible llamar a una función que no haya sido declarada anteriormente. Una función puede recibir un número desconocido de argumentos y C++ proporciona lo necesario para poder examinar los argumentos almacenados en la pila en tiempo de ejecución:

```
void imprime(int cuantos, ...)
{
    va_list vl;
    va_start(vl, cuantos);

    for (int i = 0; i < cuantos; ++i) {
        print(va_arg(vl, int));
    }

    va_end(vl);
```

```
}

imprime(3, 1, 2, 3);
```

Sin embargo esto tiene consecuencias importantes en cuanto a la generación de código. Ya que la función llamada no conoce de antemano el número de parámetros que le son enviados, el código de limpieza de la pila no puede colocarse en el código de la función llamada sino que debe hacerse del lado del llamante. En ensamblador x86 lo anterior se ilustraría de la siguiente forma:

```
; limpieza del lado del llamado (convención stdcall)

PUSH 1
PUSH 2
CALL funcion
PUSH 3
PUSH 4
CALL funcion

funcion:
  POP ; liberar espacio usado por un argumento
  POP ; liberar espacio usado por un argumento
  RET

; limpieza del lado del llamante (convención cdecl)

PUSH 1
PUSH 2
CALL funcion
POP         ; liberar espacio usado por un argumento
POP         ; liberar espacio usado por un argumento
PUSH 3
PUSH 4
CALL funcion
POP         ; liberar espacio usado por un argumento
POP         ; liberar espacio usado por un argumento

funcion:
  RET
```

Es fácil ver que aunque ambas convenciones realizan el mismo trabajo, la convención `cdecl` tiende a generar ejecutables más grandes lo cual puede llegar a perjudicar en el rendimiento si existe poca memoria caché disponible. Ésta es desafortunadamente la convención de llamada usual para código C y C++ [52].

3.4.5 Funciones virtuales

Un tipo derivado es una estructura que tiene un campo implícito de otro tipo denominado tipo base y en el que se establece una relación jerárquica:

```
struct empleado {
    double salario;
};

// un gerente también es un empleado
struct gerente : empleado {
    empleado* subalterno;
};

empleado e;
gerente g;

g.salario = 10;         // campo heredado de empleado
g.subalterno = &e;      // campo propio de gerente
```

El campo implícito es el primer campo del tipo derivado por lo que el almacenamiento de una variable de este tipo es idéntico al del tipo base excepto que el tipo derivado puede tener opcionalmente campos extras al final. Por lo anterior, un apuntador o referencia al tipo base pueden referir a una variable del tipo derivado:

```
gerente g;
empleado* p = &g;       // ok
```

Una función miembro es una función declarada dentro de una estructura y que hace referencia implícita a una variable del tipo estructurado:

```
struct empleado {
    double salario;

    void imprime_salario( )
    {
        print(salario);
    }
};
```

es conceptualmente equivalente a:

```
struct empleado {
    double salario;
};

void imprime_salario(empleado*& e)
{
    print(e->salario);
}
```

Una función virtual es una función miembro que es elegida en tiempo de ejecución en base al tipo real de la variable implícitamente referida en la función miembro. Una función virtual se declara con la palabra reservada `virtual`:

```
struct empleado {
    int salario;

    virtual void imprime_salario( )
    {
        print(salario);
    }
};

struct supervisor : empleado {
    virtual void imprime_salario( )
    {
        print(salario + 5);
    }
};

supervisor s;
s.salario = 10;

empleado* e = &s;
e->imprime_salario( );        // imprime 15
```

La implementación del mecanismo de funciones virtuales es dependiente de la implementación lo que es ocasionalmente criticado por algunos programadores de sistemas. Sin embargo la mayoría de los compiladores cumplen con la interfaz binaria de Intel Itanium para C++ [53]. En dicha interfaz se incurre en las siguientes sobrecargas para un tipo que declare (o que tenga un tipo base que declare) funciones virtuales:

- Un campo implícito de un tipo apuntador. Este apuntador es llamado apuntador a tabla virtual.
- Una asignación implícita del apuntador a tabla virtual durante la creación de una variable.
- Una invocación a función virtual es una llamada a función mediante apuntador el cual está almacenado en un arreglo referido por el apuntador a tabla virtual. El índice del arreglo es conocido en tiempo de compilación.

Aunque el comité de estandarización de C++ concluye en un reporte técnico sobre el rendimiento de C++ que la sobrecarga del mecanismo de funciones virtuales es despreciable [54], los desarrolladores de aplicaciones de alto rendimiento consideran que se ignoró la sobrecarga que se incurre cuando la función a ser ejecutada no está en la memoria caché [31], por lo que dicha sobrecarga es usualmente mayor que la sugerida por el reporte.

Si bien pareciera que el mecanismo de funciones virtuales cumple con la regla de no imponer sobrecargas a los usuarios que no usan una característica, esto no necesariamente se cumple para el desarrollo de bibliotecas pues el autor de un tipo debe decidir con anticipación si el tipo proporcionará o no funciones

virtuales. Si el autor decide no proporcionarlas, entonces el tipo no tendrá sobrecargas añadidas pero no podrá ser usado como tipo base para alguien que requiera dicho mecanismo. En cambio si el autor decide utilizar funciones virtuales entonces se impondrá una sobrecarga inevitable para todos los usuarios de dicho tipo aún si no necesitan ese mecanismo. Creemos que ésta es una debilidad fundamental del mecanismo de funciones virtuales de C++.

3.4.6 Plantillas

Una plantilla es un fragmento parametrizado de código que puede ser instanciado para ser usado en el programa. Los parámetros de una plantilla pueden ser direcciones constantes, constantes enteras, tipos y plantillas de tipos. Se pueden parametrizar estructuras y funciones. Una plantilla se declara con la palabra reservada `template` seguido de la lista de parámetros de plantilla delimitada por corchetes angulares.

En el caso de funciones, los parámetros de plantilla pueden inferirse si dependen de los tipos de los argumentos usados para la invocación:

```
template<typename T>
T minimo(T a, T b)
{
    return (b < a ? b : a);
}

minimo(1, 1);                 // T inferido como int
minimo<int>(1, 1);            // T dado explícitamente
```

El parámetro de plantilla debe ser el mismo para todas las posibles inferencias. Esto implica que la plantilla `minimo` presentada arriba no podrá comparar un `int*` con un `const int*` u otra pareja de tipos relacionados:

```
float f;
double d;

minimo(f, d);                 // error, T = float y double
minimo<double>(f, d);         // ok, T dado explícitamente
```

Una posible solución consiste en enlazar cada parámetro de función a un parámetro de plantilla diferente, sin embargo expresar el tipo de retorno se complica: la operación `minimo` entre un `float` y un `double` debe regresar un `double` sin importar si el `double` es el primer o el segundo parámetro. C++0x proporciona una manera alternativa de declarar una función que permite expresar el tipo de retorno correcto utilizando `decltype`:

```
template<typename T, typename U>
auto minimo(T a, U b)->decltype(b < a ? b : a)
{
    return (b < a ? b : a);
}
```

Sin embargo aún para los casos simples como `minimo` esto puede resultar tedioso y difícil de entender. Para casos complicados esto puede ser inviable pues todo el código del cual dependa el tipo de retorno debe proporcionarse a `decltype` y este operador sólo maneja expresiones.

Tres de los problemas sintácticos más graves de C++ son consecuencia del uso de paréntesis angulares para indicar la lista de parámetros de plantilla:

```
// vector es una plantilla de tipo, denota un arreglo

vector<int> vec;              // vector de ints
vector<vector<int>> mat;      // vector de vectores de ints
```

El análisis léxico de C++ y de muchos lenguajes usualmente siguen el principio de la máxima concordancia de símbolos para separar los tokens; el cierre doble de la lista de parámetros de plantilla será interpretado como el operador de desplazamiento hacia la derecha >> y el árbol de sintaxis generado será incorrecto. C++0x corrige esto introduciendo una regla léxica que intenta balancear los paréntesis angulares y predecir el significado del token >> evitando este problema en la mayoría de las situaciones pero cambiando el significado de algunos programas anteriores a la nueva versión del estándar.

Además, la clase de analizadores LL(k) es insuficiente para analizar sintácticamente C++ debido a que una expresión de la forma `A<B<C<D<E…` requerirá leer una cantidad no acotada de tokens para determinar si dicha expresión puede ser un uso de plantilla (los paréntesis angulares están balanceados y cerrados correctamente) o si se trata de comparaciones encadenadas.

Además, la expresión `A<W<X>(Y)>(Z)` tiene varios significados potenciales y por lo mismo varios árboles de sintaxis diferentes: si `A` es una plantilla y `W` no lo es entonces la expresión es una llamada a función o una declaración, en otro caso la expresión representaría comparaciones encadenadas con paréntesis redundantes. La técnica usual para minimizar estos problemas es llevar a cabo parte del análisis semántico durante el análisis sintáctico para guiar la construcción del árbol de sintaxis correcto mediante el análisis del significado de los símbolos que aparecen en dichas expresiones. Sin embargo existen casos que no pueden ser manejados ni siquiera con la tabla de símbolos pues el árbol sintáctico correcto sólo puede conocerse hasta el momento de la instanciación

de la plantilla. En estos casos el programador debe guiar manualmente al compilador con la palabra reservada `typename`:

```
struct s {
    typedef int f;      // alias de tipo
};

template<typename T>
void g( )
{
    T::f& a;            // AND binario entre T::f y a
    T::f c[1];          // error de sintaxis

    typename T::f* a;   // declaración de apuntador a T::f
}
```

Una plantilla de función puede especializarse totalmente, es decir, especificar una implementación en particular cuando de los parámetros de plantilla coinciden con valores específicos.

```
template<typename T>
void f(T);              // plantilla general

template<>
void f<>(int);          // plantilla para T = int

f('a');          // llama a la plantilla general
f(5);            // llama a la plantilla especializada
```

Sin embargo el orden en el que las especializaciones son declaradas importa: las especializaciones no se toman en cuenta durante la resolución de plantillas de función sobrecargadas y una especialización se enlaza con la plantilla de función más específica declarada anteriormente.

```
template<typename T>
void g(T);

// caso 1
template<typename T> void g(T*);
template<> void g<>(int*);
// caso 2
template<> void g<>(int*);
template<typename T> void g(T*);

g<int*>(0);
// llamaría a void g<>(int*) en el caso 1
// llamaría a void g(T*) en el caso 2
```

En ambos casos se prefiere a la plantilla que toma `T*`. Sin embargo, en el primer caso la especialización de `int*` se enlazaría a dicha plantilla mientras que en el segundo caso dicha especialización se enlazaría a la plantilla que toma

T, por lo que sería ignorada a pesar de ser una coincidencia perfecta con el parámetro de plantilla.

Si los argumentos usados en una invocación a función no dependen de algún parámetro de plantilla entonces la función llamada se determina antes de la instanciación. En el caso contrario la función se determina hasta el momento de la instanciación, lo que puede resultar en que llamadas a función con el mismo nombre y los mismos argumentos invoquen funciones diferentes:

```
void f(double)
{
    print("double ");
}

template<typename T>
void g(int a, T b)
{
    f(a);         // no dependiente, sólo f(double) visible
    f(b);         // dependiente
}

void f(int)
{
    print("int");
}

g(5, 5);          // imprime "double int".
```

3.4.7 Espacios de nombres

Un espacio de nombres es un ámbito. Los identificadores declarados en un espacio de nombres se acceden mediante un prefijo formado por el nombre del espacio y el operador :: que es llamado operador de resolución de ámbito. El uso de espacios de nombres permite declarar identificadores iguales pero que pueden coexistir en un mismo programa:

```
namespace uno {
    void f( );
}

namespace dos {
    struct f { };
}

uno::f( );
dos::f v;
```

Sin embargo no todo puede usarse de forma conveniente si está dentro de un espacio de nombres. C++ permite sobrecargar algunos de los operadores:

```
complejo operator+(complejo a, complejo b)
{
    return { a.real + b.real, a.imag + b.imag };
}

namespace ns {
    complejo operator-(complejo a, complejo b);
}

complejo a;

a + a;                  // equivalente a operator+(a, a)
a - a;                  // error, no se encuentra operator-
ns::operator-(a, a);    // ok
```

Además, para muchos tipos de dato como listas y colas existen funciones usuales como `tamanyo`, `apilar` y `buscar` para los que sería incómodo tener que usar la resolución explícita constantemente:

```
namespace uno {
    struct lista;
    int tamanyo(lista&);
}

namespace dos {
    struct lista;
    int tamanyo(lista&);
}

uno::lista v1;
dos::lista v2;

tamanyo(v1);            // error en un diseño simplista
tamanyo(v2);            // error en un diseño simplista

uno::tamanyo(v1);       // ok, resolución explícita
dos::tamanyo(v1);       // ok, resolución explícita
```

C++ ataca el problema definiendo lo que se conoce como búsqueda dependiente de los argumentos. El algoritmo se lleva a cabo cuando se hace una invocación a función y no se usa la resolución explícita sobre la función a llamar y su versión simplificada consiste en lo siguiente [47]:

- Se encuentra el nombre buscado dentro de la resolución ordinaria.
    - Si se encuentra una declaración que no es una función, entonces se usa esta declaración y se termina la búsqueda.
- Si se encuentra una función posiblemente sobrecargada entonces se agrega la función al conjunto de funciones precandidatas.
- Para cada argumento usado en la invocación a función:
    - Se determina el tipo resultado de ignorar los modificadores de arreglos, apuntadores, referencias y `const` al tipo del argumento

    - Se determina el espacio de nombres en donde está declarado dicho tipo.
    - Se buscan en ese espacio todas las funciones que tengan el mismo nombre que la función a invocar y se agregan al conjunto de funciones precandidatas.
- Se realiza la resolución sobre el conjunto de funciones precandidatas.

Aunque este algoritmo permite llamar a la función `tamanyo` del ejemplo anterior sin especificar explícitamente el espacio de nombres, las consecuencias no eran bien comprendidas al momento de proponerlo:

```
namespace biblioteca {
    template<typename AP>
    void copia(AP destino, AP origen, int n)
    // copia n elementos de origen a destino
    {
        for (int i = 0; i < n; ++i) {
            *destino++ = *origen++;
        }
    }

    template<typename AP>
    void clona(AP& destino, AP origen, int n)
    // pide memoria para n elementos y copia
    {
        destino = pide_memoria(sizeof(*origen) * n);
        copia_n(destino, origen, n);
    }
}

namespace usuario {
    struct mio {
        int n;
    };

    void copia_n(mio* a, mio* b, int n)
    // copia el valor n al rango [a, b)
    {
        while (a < b) {
            a++->n = n;
        }
    }
}

usuario::mio arr[5];
usuario::mio* nuevo;
biblioteca::clona(nuevo, arr, 5);
```

Desafortunadamente, en el ejemplo anterior la llamada a `copia_n` dentro de la función `biblioteca::clona` llamará a la función `usuario::copia_n`, que es inyectada por el algoritmo de búsqueda dependiente y que tiene una semántica diferente a la función `biblioteca::copia_n` que es la función que el autor de la

biblioteca intentaba llamar. Aunque el autor de la biblioteca pudo haber documentado que la función `biblioteca::clona` depende de la funcion `biblioteca::copia_n` como una advertencia para los usuarios, la única manera práctica de resolver este problema es usar la resolución explícita aún dentro del mismo espacio de nombres, que es lo que el algoritmo de búsqueda dependiente intentaba solucionar y que desafortunadamente complicó más:

```
namespace biblioteca {
    template<typename AP>
    void clona(AP& destino, AP origen, int n)
    {
        destino = malloc(sizeof(*origen) * n);
        biblioteca::copia_n(destino, origen, n);
    }
}
```

3.4.8 Manejo de excepciones

Una excepción es un retorno multinivel no estructurado. Las excepciones son usualmente utilizadas para reportar errores y son una alternativa al uso de valores de retorno con significados especiales.

Una sentencia `throw` es similar a un `goto` que envía un valor y la pareja de sentencias `try` y `catch` denotan algo similar a una etiqueta destino que captura dicho valor. Al lanzar una excepción las funciones activas en ese momento regresan hasta que la excepción llega al `catch` correspondiente. Después de esto se continúa la ejecución normal del programa:

```
void recursivo(int i)
{
    if (aleatorio( ) == 12345) {
        throw i;
    }

    recursivo(i + 1);
}

try {
    recursivo(1);
}
catch (int n) {
    print("recursión terminada en el nivel: ", n);
}
```

Ya que, excepto en casos triviales, el `catch` que corresponde a la elevación de una excepción sólo puede determinarse en tiempo de ejecución, es posible que una implementación sencilla de este mecanismo requiera ejecutar

instrucciones adicionales durante el transcurso del programa para prepararse ante la eventual elevación de una excepción.

Al diseñar el mecanismo de manejo de excepciones, Stroustrup decidió que dicho mecanismo sólo se incorporaría al lenguaje si se podía garantizar que un programa que no elevara excepciones en tiempo de ejecución no tendría ninguna sobrecarga adicional en tiempo [2]. Si bien la interfaz binaria de Intel Itanium para C++ garantiza que ninguna instrucción extra de la máquina abstracta definida por C es ejecutada si una excepción no es elevada la presión que se ejerce sobre el optimizador puede impactar en la calidad de la generación de código a comparación con el código que se generaría en un lenguaje sin manejo de excepciones [55].

Un problema en el desarrollo de bibliotecas es causado por tener que decidir si usar excepciones o valores de retorno como método preferido para el reporte de errores. Aunque las excepciones son comúnmente aceptadas como una alternativa superior al uso de valores de error, éstas crean retornos implícitos y código que no esté preparado puede tener problemas en presencia de las mismas. En estas circunstancias el uso de bibliotecas que usen excepciones se complica pues no existe una manera universal de convertir automáticamente dichas bibliotecas a bibliotecas que usen valores de error.

3.4.9 Organización física de programas

El mecanismo provisto por C++ para la implementación de programas formados por múltiples archivos de código fuente es la sentencia `#include` la cual incluye textualmente el contenido del archivo indicado. Una unidad de traducción es la entrada presentada al compilador una vez que se ejecutan todas las sentencias de preprocesamiento y que puede ser procesada de manera independiente a otras unidades de traducción. Las unidades de traducción son posteriormente enlazadas para formar el ejecutable final.

Tradicionalmente los archivos fuente se dividen en dos grupos: cabeceras y archivos de implementación. Las cabeceras contienen únicamente declaraciones. Los archivos de implementación contienen las definiciones de funciones y variables globales y generalmente corresponden a una unidad de traducción cada uno.

Ya que las cabeceras están pensadas para incluir declaraciones comunes a varias unidades de traducción, éstas son reprocesadas por el compilador cada vez que se desea compilar una unidad de traducción diferente. Generalmente

esto no ha sido problema pues las declaraciones pueden ser procesadas rápidamente por el compilador.

La definición de una plantilla, sin embargo, debe estar visible durante el procesamiento de cada unidad de traducción que haga uso de dicha plantilla: la característica de plantillas compiladas separadamente definida por C++98 únicamente fue implementada por el compilador EDG y tanto la complejidad de implementación como la ausencia de los beneficios esperados causaron que dicha característica fuera removida de C++0x [56, 57].

Debido a lo anterior, las plantillas compartidas por varias unidades de traducción deben estar tanto declaradas como definidas en las cabeceras. El hecho de que las plantillas tengan que ser procesadas repetidamente, en cambio, sí representa una sobrecarga considerable en tiempo de compilación: la cantidad de código fuente presente en las cabeceras de proyectos que hagan un uso extensivo de plantillas usualmente supera la cantidad de código presente en los archivos de implementación.

Un programa no puede tener dos definiciones diferentes para una misma entidad y se le conoce como la regla de la definición única. Cumplirla puede complicarse seriamente con las plantillas; la instanciación es dependiente del contexto:

```
// cabecera.h

template<typename T>
void f(T a);

template<typename T>
void g(T a)
{
    f(a);
}

// archivo1.cpp

#include "cabecera.h"

void h1(int n)
{
    g(a);   // T = int, redirige a f(T) con T = int
}

// archivo2.cpp

#include "cabecera.h"
void f(int);
void h2(int n)
{
    g(a);   // T = int, redirige a f(int) no a la plantilla
```

```
}
```

El estándar de C++ no obliga a la implementación a notificar violaciones a la regla anteriormente mencionada y está permitido que el enlazador descarte de manera arbitraria definiciones repetidas de la misma entidad sin necesidad de verificar si difieren en algún aspecto por lo que el problema puede pasar desapercibido. Para una implementación que desee verificar violaciones a la regla, por otro lado, el tiempo de compilación puede crecer dramáticamente pues es usual que una plantilla dependa transitivamente de otras más, cada instanciación reactiva las fases de análisis semántico y generación de código y se tendría que instanciar una plantilla tantas veces como unidades de traducción haya aún si los parámetros de plantilla son los mismos en todas las instanciaciones.

Por último cabe mencionar que el orden en el que se enlaza el código objeto resultado de la compilación de múltiples unidades de traducción no se garantiza: aunque las variables globales se inicialicen en el orden en el que aparecen en su unidad de traducción, el orden entre unidades de traducción diferentes no está garantizado:

```
// archivo1.cpp
int n1 = 5;

// archivo2.cpp
extern int n1;
int n2 = n1;            // valor indefinido
```

3.4.10 Otras características

Referencias a valores-d

Una referencia a valor-d se declara con el operador `&&` prefijo sobre el identificador. Estas referencias permiten sobrecargar una función que distinga entre una variable ordinaria y un temporal:

```
void f(int &n);         // referencia a valor-i
void f(int &&n);        // referencia a valor-d

int n;

f(n);                   // llama a f(int&);
f(n + n);               // llama a f(int&&);
```

Para tipos de datos complicados, distinguir entre un valor-i y un valor-d puede habilitar optimizaciones importantes:

```
struct cadena {
    int longitud;
    char* memoria;
};

void copia(cadena& destino, const cadena& origen)
// crea una copia física del bloque de memoria
{
    destino.memoria = pide_memoria(origen.longitud);
    destino.longitud = origen.longitud;

    for (int i = 0; i < origen.longitud; ++i) {
        destino.memoria[i] = origen.memoria[i];
    }
}

void copia(cadena& destino, cadena&& origen)
// 'roba ' la memoria del temporal sin copiarla
{
    destino.memoria = origen.memoria;
    destino.longitud = origen.longitud;

    origen.memoria = 0;
    origen.longitud = 0;
}
```

El robo de recursos de un temporal es seguro pues éstos sólo pueden participar en la expresión inmediata a la de su creación y posteriormente serán destruídos; otras partes del programa no podrán darse cuenta. A esto se le conoce como semántica de movimiento.

Las referencias a valores-d permiten diseñar tipos que no pueden copiarse pero sí pueden reubicarse en memoria. Muchos algoritmos útiles, como los algoritmos de ordenamiento, no necesitan realizar copias reales pues reubicar los elementos es suficiente. Operaciones comunes de algunas estructuras de datos, como la reubicación de un arreglo que crezca dinámicamente, también se benefician de la semántica de movimiento al omitir copias innecesarias.

El problema del reenvío [58] consiste en poder declarar una función que tome un argumento y lo reenvíe a otra función preservando sus propiedades iniciales, incluyendo si el argumento recibido era un temporal o no. C++ lo resuelve usando la sintaxis de referencia a valor-d y definiendo los siguientes casos de colapso de referencias que se aplican durante la inferencia de un tipo dependiente de un parámetro de plantilla:

Tipo del argumento	Parámetro declarado	Tipo deducido
T&	T&	T&
T&&	T&	T&

T&	T&&	T&
T&&	T&&	T&&

Un programa de ejemplo es el siguiente:

```
template<typename T>
void f(T&&);

int n;

f(n);       // T& y T&& ⟶ T&, f se instancia como f(int&)
f(n + n);   // T&& y T&& ⟶ T&&, f se instancia como f(int&&)
```

Una consecuencia directa es que no es posible declarar una plantilla de función que únicamente admita referencias a valores-d de algún tipo no especificado. El diseño de esta característica fue problemático pues como ya se mencionó, una referencia a valor-i `const` puede enlazarse con un valor-d, por lo que tuvo que idearse este mecanismo alternativo.

Expresiones constantes

Una expresión constante es aquella que puede ser evaluada en tiempo de compilación. C++ garantiza esto para ciertas expresiones. C++0x proporciona la palabra reservada `constexpr` para declarar variables que deban ser inicializadas con expresiones constantes y para solicitar que una función sea evaluada en tiempo de compilación si cumple ciertas restricciones [59].

```
constexpr int duplica(int n)
{
    return n * 2;
}

constexpr int i = 0;                // ok
constexpr int k = duplica(2);       // ok

constexpr int j = aleatorio( );
// error, inicializador no constante

constexpr int m = duplica(aleatorio( ));
// error, inicializador no constante
```

La única sentencia permitida dentro de una función `constexpr` es la sentencia `return`. El modificador `constexpr` aplicado a una función es ignorado silenciosamente si alguno de los argumentos recibidos no es constante.

Es imposible pasar explícitamente valores constantes como parámetros de plantilla a un constructor (un constructor es una función miembro llamada

implícitamente durante la creación de una variable) pues no existe una sintaxis que lo permita:

```
template<int N>
void f( );  // ningún parámetro deduce a N

struct s {
    template<int N>
    s( );   // ningún parámetro deduce a N
};

f<5>( );    // ok, N explícito pues no puede deducirse
s<5> v;     // error, s no es una plantilla
```

Creemos que este ejemplo ilustra la necesidad de unificar ambos mecanismos ya que tampoco es posible declarar un parámetro de función como `constexpr`.

Inicialización uniforme

C++0x generaliza las listas de inicialización para invocar un constructor. Las sintaxis disponibles anteriormente eran las siguientes:

```
int a;           // construcción por defecto
int b(5);        // construcción con inicialización
int c = 5;       // construcción con inicialización
```

La sintaxis `T variable(argumentos)` es necesaria para invocar constructores que tomen más de un argumento además de que los constructores marcados con la palabra reservada `explicit` no pueden ser invocados con la sintaxis de inicialización mediante asignación.

```
struct vector {
    explicit vector(int n)
    {
        // construye un vector de tamaño inicial 5
    }
};

vector v1 = 5;    // error, constructor es explícito
vector v2(5);     // ok
```

Sin embargo esta sintaxis no generaliza si la lista de argumentos está vacía:

```
vector v3( );     // declaración de función
v3 = v2;          // error, v3 es una función
```

C++0x permite utilizar una lista de inicialización en un intento por uniformizar la sintaxis de inicialización [60]:

```
struct s {
    s( );
    s(int);
    s(int, double);
};

s p1 = { };             // se invoca s( )
s p2 = { 5 };           // se invoca s(int)
s p3 = { 5, 5.5 };      // se invoca s(int, double)
```

Surgen problemas si el usuario decide agregar posteriormente un constructor que toma un `initializer_list`:

```
struct s {
    s(int, double);
    s(initializer_list<int>);
};

s p4 = { 5, 5.5 };
// error, s(initializer_list<int>) tiene prioridad
// pero la lista de inicialización no es homogénea
```

Es inválido modificar el contenido de un `initializer_list` pues almacena su contenido como variables `const`, por lo que no es posible reubicar el recurso de un elemento de la lista (aunque sean valores-d, serían valores-d `const`):

```
template<typename T>
struct vector {
    vector(initializer_list<T>);
};

struct arbol;
// no se permite copiar pero puede reubicarse copiando el
// apuntador a la raíz y dejando al temporal con raíz nula

arbol crea_arbol( );
// función que regresa un arbol valor-d

arbol a = crea_arbol( );
// ok, construcción reubica el recurso del temporal

arbol v1[] = { crea_arbol( ) };
// ok, inicialización manejada por el compilador

vector<arbol> v = { crea_arbol( ) };
// error, los valores-d de un initializer_list son const
```

4 La programación genérica

El término de programación genérica fue presentado en 1988 por Alexander Stepanov y David Musser [18]. En su publicación original describen a la programación genérica como el estilo de programación que se centra en la abstracción de algoritmos concretos y eficientes, actividad que permita a dicho algoritmo ser usado con diversas representaciones de datos y con el objetivo de construir componentes reutilizables.

La programación genérica no es una tecnología, es un estilo de programación y por lo mismo éste puede seguirse en casi cualquier lenguaje de programación. Sin embargo, el objetivo de la programación genérica es el de, eventualmente, construir grandes catálogos de componentes de software de máximo rendimiento y de calidad probada que puedan reutilizarse para construir todo tipo de sistemas. Desafortunadamente, el requerimiento de que dichos componentes sean de máximo rendimiento descarta muchas de las tecnologías más usadas en la actualidad.

Aunque muchos lenguajes tienen características útiles en la implementación de algoritmos y de estructuras de datos que sigan el estilo que la programación genérica dicta, la creación de un lenguaje de programación apropiado que permita que los esfuerzos por construir dicho catálogo de componentes converjan representaría un avance importante para la industria del software. Tradicionalmente el lenguaje por excelencia de la programación genérica ha sido C++, sin embargo la programación genérica aún es severamente limitada en este lenguaje [55].

En este capítulo describimos la programación genérica a detalle, abordamos las limitantes de C++ y de otros lenguajes para el soporte de la programación genérica y también describimos cómo fue que los esfuerzos por solucionar dichas limitantes en C++ fallaron debido a una serie de problemas acumulados durante la larga evolución del lenguaje.

4.1 Definición y analogía matemática

Nosotros seguimos la definición presentada en [62] la cual reproducimos a continuación:

La programación genérica es una subdisciplina de las ciencias de la computación que se encarga de la búsqueda de representaciones abstractas de algoritmos eficientes,

estructuras de datos, otros conceptos de software y de su organización sistemática. El objetivo de la programación genérica es el de expresar algoritmos y estructuras de datos en una forma adaptable e interoperable que permite su uso directo en la construcción de software.

Las ideas principales incluyen:

- *Expresar algoritmos con las suposiciones mínimas sobre las abstracciones de datos usadas y viceversa, haciéndolos así tan interoperables como sea posible.*
- *Elevar un algoritmo concreto a su forma más general posible sin perder eficiencia; esto es, la forma más abstracta de algoritmo tal que cuando es especializado de vuelta a su caso concreto sea tan eficiente como el algoritmo original.*
- *Cuando el resultado de la abstracción anterior no sea suficientemente general para cubrir todos los casos de uso del mismo, proveer una forma aún más general, pero asegurándose de proveer una forma especializada del algoritmo que sea elegida automáticamente cuando sea aplicable.*
- *Proveer más de una versión del mismo algoritmo genérico con el mismo propósito y a mismo nivel de abstracción cuando ninguna es superior a las otras en cuestión de eficiencia para todas las entradas posibles. Esto introduce la necesidad de proveer caracterizaciones suficientemente precisas sobre el dominio para el que cada versión es la más eficiente.*

Dar una analogía matemática de la programación genérica partiendo de la definición es difícil pues ésta sólo constituye una serie de requerimientos. Sin embargo es posible construir la analogía partiendo de cómo se llevan a la práctica los postulados de este paradigma en los diferentes lenguajes de programación. Por ejemplo, se ha usado [63] la teoría de categorías al describir a la programación genérica en términos de los lenguajes de programación funcionales que son aquellos inspirados por el cálculo lambda de Alonzo Church [64] y en los que el cómputo es la evaluación de expresiones matemáticas sin incluir la noción de memoria o de estado del programa.

La analogía construida a partir de la práctica de la programación genérica en C++ toma el enfoque del álgebra abstracta, la cual es el área de las matemáticas que estudia las estructuras algebraicas. Una estructura algebraica está formada por un conjunto y un grupo de operaciones sobre dicho conjunto las cuales cumplen con ciertas suposiciones o postulados. En nuestra analogía consideraremos a los tipos de datos como conjuntos y a las funciones definidas sobre ellos como las operaciones sobre el conjunto.

Las ideas principales de esta analogía son las siguientes:

- Las plantillas de C++ permiten implementar algoritmos sin mencionar tipos en específico. Sin embargo los algoritmos tienen requerimientos implícitos en el código:

```
template<typename T>
T minimo(T a, T b)
{
    return b < a ? b : a
}
```

Se asume que el tipo `T` forma una estructura algebraica para la que existe un operador denotado por el símbolo < y que proporciona un orden total estricto[1] a los elementos del tipo. Los requerimientos sobre los tipos de una plantilla pueden ser descritos en términos de las estructuras algebraicas que deben modelar dichos tipos.

- C++ ofrece el mecanismo de sobrecarga de funciones el cual permite elegir la función más específica sobre un conjunto de funciones que tienen el mismo nombre. Esto es necesario pues la implementación más eficiente de un algoritmo no siempre es la más general.

```
template<typename T>
int distancia(T a, T b)
// algoritmo general, tiempo lineal
{
    int cuenta = 0;

    while (a++ != b) {
        ++cuenta;
    }

    return cuenta;
}

int distancia(int a, int b)
// algoritmo específico, tiempo constante
{
    return b - a;
}
```

Más aún, el segundo algoritmo no es exclusivo para `int` sino que funcionaría para todos los tipos con un operador resta de la forma

[1] Un orden total estricto es una relación binaria R sobre un conjunto S tal que es:

- Irreflexiva: $\neg(a < a) \; \forall \; a \in S$
- Asimétrica: $a < b \longrightarrow \neg(b < a) \; \forall \; a, b \in S$
- Transitiva: $a < b \wedge b < c \longrightarrow a < c \; \forall \; a, b, c \in S$
- Total: $a < b \vee a > b \vee a = b \; \forall \; a, b \in S$

operador - : T × T ⟶ entero

con la semántica esperada y que se ejecute en tiempo constante. Esto es cierto, por ejemplo, para los apuntadores. En otras palabras, pueden existir algoritmos más eficientes sobre estructuras algebraicas que tengan más propiedades que otras para los que el algoritmo también esté definido.

- Muchos algoritmos deben ser expresados en términos de tipos y de tipos dependientes a éstos. Por ejemplo el algoritmo `acumula` toma una secuencia y suma todos sus miembros, sin embargo para poder especificar el tipo de retorno del algoritmo debemos conocer el tipo de los elementos almacenados y éste depende de la secuencia:

```
template<typename T>
typename T::elemento acumula(T secuencia)
{
    typename T::elemento resultado = 0;

    for (int i = 0; i < secuencia.tamanyo; ++i) {
        resultado += secuencia[i];
    }

    return resultado;
}
```

  Los algoritmos que trabajan sobre estructuras algebraicas con operaciones que involucran a más de un tipo, como en el caso de los espacios vectoriales, necesitan que exista una manera de especificar las relaciones existentes entre tipos. C++ permite declarar tipos dentro de estructuras y tradicionalmente se han usado para especificar tipos dependientes a la misma.

4.2 Limitaciones de la programación orientada a objetos

El principal mecanismo de soporte a la programación orientada a objetos en C++ y en lenguajes relacionados como Java y C# es el de las funciones virtuales. Como ya se mencionó, este mecanismo gira en torno al tipo del parámetro implícito de las funciones miembro. Durante la construcción de una jerarquía de tipos sólo se permite variar este parámetro; el resto de los parámetros deben ser los mismos en toda la jerarquía:

```
interface mamifero {                // código Java
    void aparea(mamifero);
}
```

```
class gato implements mamifero {
    void aparea(gato);              // error
    void aparea(mamifero);          // ok
}
```

Esto implica que un tipo que desee pertenecer a los `mamiferos` deberá implementar la función `aparea` que toma cualquier otro `mamifero`. Por otro lado, lo anterior puede expresarse correctamente en términos matemáticos:

$$\texttt{T} \text{ modela } \texttt{mamifero} \Rightarrow \textit{aparea: } T \times T \longrightarrow \textit{void} \text{ está definido}$$

El prototipo de la función también puede expresarse correctamente en C++:

```
template<typename T>
void aparea(T, T);
// declaración general

void aparea(gato, gato);
// implementación específica
```

La programación orientada a objetos agrupa los algoritmos en torno a tipos individuales; no tiene el concepto de tipos dependientes, sólo de tipos relacionados jerárquicamente. Matemáticamente esto le permite agrupar algoritmos en torno a estructuras algebraicas como semigrupos, grupos y anillos pero no en torno a espacios vectoriales y otras que requieran la colaboración de más de un tipo [65].

4.3 La programación genérica en C++98

La programación genérica en C++ hace uso extensivo de las plantillas pero desafortunadamente no existe una manera de describir las propiedades que los parámetros de plantilla deben cumplir. Esto imposibilita poder agrupar a los tipos en torno a las estructuras algebraicas equivalentes y trae como consecuencia que un algoritmo no pueda distinguir entre qué versión elegir:

```
template<typename T>
void avanza(T& iterador, int pasos)
// algoritmo general en tiempo lineal
// requerimientos sobre T: operator++
{
    for (int i = 0; i < pasos; ++i) {
        ++posicion;
    }
}

template<typename T>
void avanza(T& iterador, int pasos)
// algoritmo en tiempo constante
```

```
// requerimientos sobre T: operator+=
{
    iterador += pasos;
}
```

En el ejemplo anterior se asume que el `operator+=` es semánticamente equivalente a aplicar el `operator++` repetidamente y es más eficiente, lo cual es cierto para todos los tipos primitivos con estos operadores.

Ya que ambas sobrecargas tienen el mismo prototipo de función cualquier llamada a la función `avanza` será ambigua; los requerimientos sólo pueden especificarse como comentarios y no es posible decir qué versión es preferible. Usualmente el algoritmo general es usado para iteradores sobre listas enlazadas, sin embargo la segunda versión del algoritmo no sólo es útil para apuntadores; se puede implementar un iterador sobre un arreglo circular que salte sobre su secuencia en tiempo constante de la siguiente manera:

```
struct iter_circular {
    int* memoria;
    int indice;
    int tamanyo;
};

int& operator*(const iter_circular& i)
// desreferencia del iterador
{
    return i.memoria[i.indice];
}

void operator+=(iter_circular& i, int pasos)
// avance del iterador en tiempo constante
{
    i.indice += pasos;
    i.indice %= tamanyo;
}
```

La manera en la que esta limitante se esquiva en C++ es mediante la definición de una plantilla de rasgos de tipo:

```
struct con_adicion { };
struct sin_adicion { };

template<typename T>
struct rasgo {
    typedef sin_adicion valor;
    // versión por defecto del rasgo
};

template<typename T>
void avanza_detalle(T&, int pasos, sin_adicion);
// algoritmo en tiempo lineal
```

```
template<typename T>
void avanza_detalle(T&, int pasos, con_adicion);
// algoritmo en tiempo constante

template<typename T>
void avanza(T& iterador, int pasos)
{
    avanza_detalle(
        iterador,
        pasos,
        typename rasgo<T>::valor( )
    );
}
```

La sobrecarga se controla agregando un parámetro extra a las implementaciones reales de `avanza` y guiando la resolución usando el tipo indicado en la plantilla `rasgo`. Por defecto todos los tipos tendrán el rasgo `sin_adicion` y el usuario debe especializar manualmente la plantilla para indicar que su tipo es diferente:

```
template<>
struct rasgo<iter_circular> {
    typedef con_adicion rasgo;
    // para iter_circular sí existe el operator+=
};
```

Esto implica que los implementadores de bibliotecas deberán crear plantillas de rasgos de manera similar y solicitar a los usuarios que las especialicen explícitamente para sus propios tipos. Además de ser tedioso, esto puede causar problemas de mantenibilidad del código si un rasgo se vuelve obsoleto en el futuro o si se encuentran implementaciones más eficientes de un algoritmo que serían correctas para un tipo del usuario pero que no podrán ser habilitadas hasta que se cree el rasgo y la especialización correspondientes.

Aunque en la mayoría de los casos los tipos anidados son suficientes para denotar tipos dependientes, esto no generaliza correctamente para los tipos primitivos:

```
template<typename T>
struct iterador {
    typedef T elemento;       // tipo apuntado
    elemento& operator*( );   // desreferencia
};

template<typename T>
void intercambia_apuntado(T iter1, T iter2)
{
    typename T::elemento temp = *iter1;

    *iter1 = *iter2;
    *iter2 = temp;
```

```
}

iterador<int> a, b;
intercambia_apuntado(a, b);
// ok, iterador<int>::elemento es int

int *c, *d;
intercambia_apuntado(c, d);
// error, int* no es un struct, no hay int*::elemento
```

Para esquivar el problema se deben usar nuevamente los rasgos de tipo con los problemas e inconvenientes que esto representa.

Ya que el compilador no puede filtrar los parámetros de plantilla en base a las propiedades que deben cumplir, los mensajes de error en tiempo de compilación pueden ser sumamente confusos. La siguiente declaración resumida es parte de la biblioteca estándar de C++:

```
template<typename T>
void sort(T inicio, T fin);
// ordena la secuencia delimitada por [inicio, fin)
// algoritmo en tiempo O(n log n)
//
// requerimientos sobre T: tipo iterador con
//          desreferencia     *iter
//          saltos            iter + n
//                            iter - n
```

El error de compilación del compilador g++ 4.5.1 para la llamada a función `sort(1, 1)` con `T = int` es generado por el fallo de las instanciaciones de plantillas anidadas y causado porque un `int` no puede desreferenciarse. El mensaje de error es el siguiente:

```
In file included from
mingw32/4.5.1dw2/include/c++/bits/stl_algobase.h:67:0,
from mingw32/4.5.1dw2/include/c++/bits/char_traits.h:41,
from mingw32/4.5.1dw2/include/c++/ios:41,
from mingw32/4.5.1dw2/include/c++/ostream:40,
from mingw32/4.5.1dw2/include/c++/iostream:40,
In instantiation of 'std::iterator_traits<int>':
instantiated from 'void std::sort(_RAIter, _RAIter)
[with _RAIter = int]'
error: 'int' is not a class, struct, or union type

In file included from mingw32/4.5.1dw2/include/c++/algorithm:63:0,
In function '_RandomAccessIterator
std::__unguarded_partition_pivot(_RandomAccessIterator,
_RandomAccessIterator) [with _RandomAccessIterator = int]':
instantiated from 'void std::__introsort_loop(
_RandomAccessIterator, _RandomAccessIterator, _Size)
[with _RandomAccessIterator = int, _Size = int]'
instantiated from 'void std::sort(_RAIter, _RAIter)
```

```
[with _RAIter = int]'
error: invalid type argument of unary '*'
```

4.4 La programación genérica en otros lenguajes

Las principales debilidades en torno al soporte para la programación genérica en distintos lenguajes son las siguientes:

1. *Un algoritmo no puede sobrecargarse*: Versiones diferentes del mismo algoritmo deben tener diferente nombre. Esto hace imposible que la especialización de los algoritmos sea transparente para el usuario.

2. *Un algoritmo genérico es menos eficiente*: Algunos lenguajes generan sólo una versión de código ejecutable para todas las posibles instanciaciones de una plantilla. Si bien esto ayuda a disminuir la cantidad de código generado, las operaciones que dependan del valor real de los parámetros de plantilla están prohibidas o se tienen que resolver en tiempo de ejecución.

3. *No es posible expresar tipos dependientes*: Para poder expresar tipos dependientes se necesitan funciones que devuelvan tipos o bien tipos anidados. Sin embargo la información sobre tipos anidados no se almacena en el mecanismo de funciones virtuales. Para los lenguajes que dependan de la resolución en tiempo de ejecución de las operaciones dependientes de los parámetros de plantilla esta información no está disponible.

4. *No se permiten especializaciones parciales de algoritmos*: Sólo puede existir la declaración para el caso general del algoritmo genérico. Los rasgos de tipo pueden usarse para solucionar este problema pero este mecanismo no puede usarse en lenguajes donde no exista una manera de expresar tipos dependientes en tiempo de compilación.

5. *La agrupación de tipos en torno a las estructuras algebraicas que modelan no es posible o es explícita*: Un tipo que cumpla las propiedades de una estructura algebraica no será reconocido como tal si no se indica explícitamente. Esto es común en lenguajes que utilizan la herencia como mecanismo de agrupación. La agrupación también puede ser simulada mediante las plantillas de rasgos pero deben especializarse manualmente y no proveen de un mecanismo general para verificar restricciones y emitir mensajes de error claros en tiempo de compilación.

6. *No existe un orden parcial en torno a la agrupación de tipos*: En algunos lenguajes las restricciones deben ser expresadas en las declaraciones de función en forma de predicados arbitrarios para los cuales el compilador no puede decidir si uno es un caso más específico de otro. Debido a esto el programador debe especificar restricciones mutuamente excluyentes en las diferentes versiones del algoritmo y así guiar al compilador en su decisión sobre qué versión usar. En caso de ambigüedad se genera un error o el orden de declaración determina la resolución.

7. *No existe una manera general de agregar funcionalidad a un tipo*: Existe una diferencia de sintaxis entre llamar a una función ordinaria o a una función miembro. Generalmente sólo el autor de un tipo puede declarar una función miembro; si un algoritmo genérico llama a funciones miembro no será posible adecuar aquellos tipos que no posean estas funciones para que puedan ser usados por el algoritmo. En el caso de funciones ordinarias las reglas de resolución son inadecuadas en la presencia de espacios de nombres o módulos.

Gracias a la popularidad de las plantillas de C++ muchos de los lenguajes más usados actualmente han desarrollado mecanismos de abstracción similares. Sin embargo ninguno de estos lenguajes tiene el soporte suficiente para poder cumplir de manera conveniente con todos los principios de la programación genérica. C++ tiene los problemas 5, 6 y 7; C# tiene los problemas 2, 3, 4, 5, 6 y 7; el lenguaje de programación Clay [66] tiene los problemas 6 y 7; el lenguaje de programación D [67] tiene los problemas 6 y 7; Java tiene los problemas 2, 3, 5 y 7; el lenguaje de programación Go [68] tiene los problemas 1, 2, 3 y 4.

En estudios comparativos sobre el nivel de soporte para la programación genérica la familia de los lenguajes funcionales ha demostrado tener un nivel de soporte superior a los lenguajes imperativos [62, 69]. Desafortunadamente los lenguajes funcionales puros son incapaces [70] de igualar la complejidad asintótica lograda en la implementación imperativa de muchos algoritmos útiles lo que los hace inviables como lenguajes de programación de sistemas.

Conviene discutir el problema 7 y lo que nosotros consideramos reglas de resolución inadecuadas. Ya hemos hablado de los problemas que introduce el algoritmo de búsqueda dependiente de C++ y los diseñadores de otros lenguajes han querido evitar añadir mecanismos similares. Sus propuestas, sin embargo, tienen defectos. En el lenguaje D el siguiente código es válido:

```
void f(T)(T v)          // plantilla en D
{
    g(v);               // f redirige a g
}
```

```
class c;                // struct en D
void g(c);

c c1;
f(c1);                  // bien, T = c, se redirige a g(c)
```

Pero no lo es si la plantilla está en su propio archivo y éste es importado:

```
// biblioteca.d

void f(T)(T v)          // plantilla en D
{
    g(v);
}
```

```
// usuario.d

import biblioteca;
// error en compilación separada de biblioteca.d
// no se encuentra la declaración de g

class c { }
void g(c) { }

c c1;
f(c1);                  // error
```

Para que un algoritmo genérico pueda declararse en un archivo separado se deberán declarar los prototipos de todas las funciones que éste use aunque dependan de los parámetros de plantilla. Esto es en el mejor de los casos impráctico; en el ejemplo anterior el usuario tendrá que declarar la función g en el archivo `biblioteca.d` o si no tiene disponible el código fuente entonces `biblioteca.d` deberá tener un nombre de módulo asignado y el usuario tendrá que crear otro archivo fuente con el mismo nombre de módulo y deberá declarar g ahí.

En el lenguaje Clay cada archivo fuente es un módulo. El lenguaje permite invocar una función sin calificación explícita del módulo en muchos casos sencillos:

```
// biblioteca.clay

record arreglo {
    tam: Int;
}

tamaño(a: arreglo) returned: Int
{
    return a.tam;
}
```

```
// usuario.clay

main( )
{
    var a = biblioteca.arreglo( );

    tamaño(a);
    // ok, se llama a biblioteca.tamaño
}
```

Sin embargo las funciones de otros módulos son ocultadas fácilmente por funciones evidentemente no relacionadas:

```
// usuario.clay

record archivo {
    bytes: Int;
}

tamaño(a: archivo) returned: Int
{
    return a.bytes;
}

main( )
{
    var a = biblioteca.arreglo( );

    tamaño(a);
    // error, usuario.tamaño oculta biblioteca.tamaño

    prelude.tamaño(a);
    // ok, prelude = biblioteca, calificación-comodín
}
```

4.5 Los esfuerzos de C++0x

Una vez publicada en 2003 la errata técnica del estándar de 1998 de C++ se iniciaron los trabajos para C++0x [71, 72]. En ese entonces ya eran bien comprendidas las debilidades de C++ con respecto a la programación genérica y una de las metas más importantes para C++0x era la de eliminar dichas debilidades.

El diseño de la característica dirigida a solucionar los problemas de las plantillas inició ese mismo año por Stroustrup y por varios investigadores de la Universidad de Indiana y de la Universidad de Texas A&M [73]. Incluso se desarrolló el lenguaje de programación G [74] como tesis doctoral en la Universidad de Indiana con el fin de validar el diseño de esta característica, la cual fue llamada conceptos. La versión final de la propuesta fue presentada ante

el comité en junio de 2008 e incorporada en el texto de trabajo del comité en noviembre de ese mismo año [75]. Si embargo el comité revirtió su decisión y la característica fue removida en julio de 2009 [76]. Aunque la intención del comité es la de continuar con el diseño de esta característica, la estimación más optimista es la de su inclusión en C++ hasta 2015 y se cree que será necesario replantearse las decisiones de diseño fundamentales [77].

Los conceptos permitían especificar una serie de requerimientos sobre los tipos. Existen dos tipos de requerimientos: tipos asociados y funciones:

```
auto concept secuencia<typename T> {
    typename elemento;  // tipo de los elementos de la secuencia
    typename iterador;  // tipo de los iteradores de la secuencia

    iterador begin(const T&);
    iterador end(const T&);
}

auto concept arreglo<typename T> : secuencia<T> {
    elemento& operator[](T&, int);
}
```

La palabra reservada `auto` le indica al compilador que todo tipo para el que pueda comprobarse que cumple con los requerimientos entonces cumple el concepto. Si el usuario no usa la palabra `auto` el cumplimiento deberá indicarse explícitamente:

```
concept no_auto<typename T> { }

struct s;
concept_map no_auto<s>;            // s cumple no_auto
```

La palabra reservada `concept_map` también sirve para indicar modelos de tipos sobre conceptos, lo que permite la adaptación del tipo:

```
struct sin_f;

concept con_f<typename T> {
    void f(T);
}

concept_map con_f<sin_f> {
    void f(sin_f)
    {
        // implementación de f para sin_f
    }
}
```

Una plantilla puede entonces restringir sus parámetros de plantilla:

```
template<secuencia T>
void imprime(const T&);

imprime(5);       // error, int no es una secuencia
```

Sin embargo existen muchos problemas con la característica anterior. Existen dos maneras sintácticamente no equivalentes de declarar una función:

```
concept con_f<typename T> {
    void f(T);                // f(variable)
    void T::f( );             // variable.f( )

    T operator+(T, T);        // variable + variable
    T T::operator+(T);        // variable + variable
}
```

Hasta el momento en que la propuesta fue retirada de C++0x no se había llegado a un concenso sobre la equivalencia de ambas formas de declaración. Además, tanto los constructores como los destructores deben ser forzosamiento funciones miembro. Sin embargo los tipos primitivos no son `structs`, por lo que no poseen funciones miembro. En esos casos y antes de abortar con un error, el compilador debe generar modelos implícitos que intenten adaptar a los primitivos en un esquema de funciones miembro. Los modelos deben ser dados explícitamente mediante un `concept_map` en el caso de los tipos asociados a tipos primitivos:

```
concept iterador<typename T> {
     typename elemento;     // tipo al desreferenciar
}

template<typename T>
concept_map iterador<T*> {
    typename T elemento;
}
```

Ya que tanto `const` como las referencias forman tipos distintos, surgen las preguntas, ¿qué sucede cuando se solicita mediante un concepto verificar la existencia de constructores y destructores para referencias? ¿existe un destructor de `const T`? ¿cuál es la diferencia con un destructor de `T`?

```
auto concept con_constructor<typename T> {
    T::T( );
}

template<con_constructor T>
void f( );

f<const int&>( );
// con_constructor<const int&>, ¿válido?
// ¿existe el prototipo const int&::const int&( )?
```

El uso de parámetros restringidos mediante conceptos impide usar funciones que no se hayan especificado en la lista de restricciones:

```
struct s { };

void f(s);
void g(s);

template<con_f T>
void h(T v)
{
    f(v);         // ok, f(T) listado en con_f
    g(v);         // error incluso si T = s
}
```

Esta restricción está pensada para facilitar la compilación separada de plantillas pero tienen efectos prácticos importantes:

```
auto concept entero<typename T> : comparable<T> {
    T operator+(T, T);
    T operator-(T, T);
    //...
    T operator+(T);
    T operator-(T);
}

template<entero T>
T valor_absoluto(T v)
{
    T res = (v >= 0 ? v : -v);

    print("depurando valor_absoluto: ", v, " ", res);
    return resultado;
}

valor_absoluto(0);  // error: print no ha sido listado en entero
```

No es posible llamar a una función más restringida desde una función menos restringida, a pesar de que el tipo cumpla las restricciones de ambas. Tampoco es posible llamar a una función no restringida desde una restringida.

El resultado de lo anterior es que no usar conceptos es mucho más flexible que usarlos. Además, el diseño tal como fue presentado no era suficientemente poderoso para poder expresar algunos conceptos, por lo que se tuvo que dar una lista de conceptos manejados internamente por el compilador. También se había decidido que la gran mayoría de los conceptos de la biblioteca no fueran `auto`, lo que implicaba dar modelos explícitos. Esto fue abordado por Stroustrup en un intento por presentar de última hora soluciones a los problemas más graves de los conceptos [78]. La propuesta fue rechazada y la característica removida de C++0x como medida precautoria.

5 El diseño del lenguaje propuesto

En este capítulo se describe el diseño de un lenguaje de programación de sistemas con soporte para la programación genérica. Se exponen las metas de diseño y se presentan las características del lenguaje de un modo que sea fácil identificar las soluciones a los problemas de C++ presentados en el capítulo 3.

Durante la presentación de las características del lenguaje describimos el diseño de un mecanismo que unifica el manejo de expresiones constantes en el lenguaje en conjunto con un rediseño del mecanismo de plantillas de C++. Se presenta también un algoritmo de resolución de funciones en presencia de espacios de nombres que elimina los problemas conocidos en otros lenguajes. Discutimos las posibles debilidades de este algoritmo y justificamos por qué estas debilidades son poco relevantes.

Además describimos los problemas fundamentales que deben ser resueltos al diseñar una sintaxis uniforme de inicialización en C++ y se da la solución a estos problemas. Por último presentamos el diseño de una característica que permite agrupar tipos en torno a sus propiedades. Mostramos que esta característica no presenta los problemas de la solución propuesta para C++0x y que el lenguaje desarrollado en esta tesis no tiene ninguna de las debilidades identificadas en el capítulo 4 con respecto a la programación genérica.

5.1 Metas de diseño

El lenguaje de programación desarrollado en esta tesis es un lenguaje de programación de sistemas que tiene por objetivos:

- *Ser tan o más eficiente en tiempo y espacio que C*: No se deben incluir mecanismos que requieran soporte en tiempo de ejecución (como la elevación de excepciones o la recolección automática de memoria) y debe ser posible realizar acciones de muy bajo nivel. Existen al menos dos aspectos en los que se puede superar el rendimiento de C. El primero implica resolver el problema de los alias [30] y no lo consideramos importante. El otro aspecto es la eficiencia de convención de llamada usada en C.

- *Ser mucho más pequeño y fácil de aprender que C++*: En C++ existe soporte tanto para la programación genérica como para la programación

orientada a objetos y éstos tienen sus propias reglas y estilos de codificación. No pretendemos dar soporte a la programación orientada a objetos por lo que el tamaño del lenguaje y el número de conceptos que necesiten aprenderse pueden disminuir.

- *Dar soporte adecuado a la programación genérica*: En base a lo expuesto en el capítulo 4, se deben proporcionar características que faciliten la codificación de programas que cumplan con los postulados de la programación genérica.

Como ya se ha hecho ver, se necesita de un lenguaje poderoso para poder dar un soporte adecuado para la programación genérica. A pesar de ello pensamos que el modelo de programación establecido por C es una base suficiente para un nuevo lenguaje. Consideramos que la principal aportación de C++ fue la de mostrar cómo se podían generalizar algunas características básicas de C para lograr un lenguaje mucho más poderoso y expresivo. Sin embargo C++ es también orientado a objetos y la lenta incorporación de mecanismos dirigidos a mejorar el soporte de la programación genérica ha hecho evidentes muchas de sus debilidades y errores de diseño.

Por lo anterior no es de sorprender que las metas de diseño del lenguaje sean parecidas a las de C y C++. Existen, sin embargo, diferencias importantes que surgen de dar un énfasis más fuerte al desarrollo de bibliotecas y de querer facilitar el uso de los lenguajes de programación de sistemas.
Las metas generales de diseño son las siguientes.

- *No debe existir ninguna característica que imponga sobrecargas inevitables al usuario de una biblioteca*: Respetar la propiedad `const` de las variables puede impactar globalmente la codificación de un programa pero no trae consigo una sobrecarga ni en tiempo ni en espacio. La declaración de funciones virtuales y la elevación de excepciones por parte de una biblioteca escrita en C++, en cambio, imponen sobrecargas que no pueden ser evitadas sin reescribir la biblioteca.

- *No dar lugar a un lenguaje de más bajo nivel con excepción de ensamblador*: El acceso a los recursos del hardware debe ser posible desde el lenguaje. Además, para ciertas aplicaciones debe ser posible codificar fragmentos del programa directamente en ensamblador en caso de que el optimizador no sea suficientemente agresivo o no conozca todas las propiedades de la plataforma.

- *Permitir que el programador haga lo que tenga que hacer, pero requerir una solicitud explícita en casos que son usualmente errores*: El lenguaje debe ser

más estricto que C en cuanto a las conversiones que impliquen pérdidas de precisión o posibles errores semánticos. Ninguna de estas conversiones debe habilitarse implícitamente pero debe ser fácil para el programador hacer uso de ellas cuando lo requiera.

- *El lenguaje no debe necesitar un preprocesador*: El preprocesador desconoce la semántica del lenguaje y las macroinstrucciones pueden incomodar al usuario de una biblioteca que las defina. Su uso impacta el tiempo de compilación y la necesidad de recurrir a él para realizar tareas simples sugiere que el lenguaje no tiene suficiente poder expresivo.

- *El orden de declaración de tipos y funciones no debe importar:* Las computadoras actuales tienen la capacidad de almacenamiento suficiente para no necesitar diseñar lenguajes que deban ser compilados en una sola pasada. Tener que duplicar las declaraciones de las entidades aumenta el tamaño del código fuente y causa problemas de mantenibilidad sin un beneficio práctico.

- *El uso de múltiples archivos fuentes debe ser un detalle menor*: El proceso de compilación de un programa completo debe ser esencialmente idéntico si éste tiene uno o varios archivos fuente; la separación del programa es meramente física. El orden de inicialización de las variables globales declaradas en diferentes archivos debe ser predecible a simple vista.

- *Importa que la compatibilidad binaria con C sea posible, pero no tiene por qué ser automática*: Existen diseños elegantes que se vuelven inviables si se desea mantener una compatibilidad binaria directa con C. Además, optimizar la convención de llamada utilizada en este lenguaje puede ser importante por lo que vale la pena explorar decisiones de diseño aunque éstas resulten en incompatibilidades. Por otra parte, es importante que sea posible utilizar bibliotecas desarrolladas en ese lenguaje.

- *No debe existir un tratamiento especial para tipos incorporados en el lenguaje sobre tipos definidos por el usuario*: Si algún tipo definido por el lenguaje es útil entonces los usuarios querrán definir tipos similares; el lenguaje debe ser suficientemente general como para permitir dar dichas definiciones. Además los algoritmos escritos en términos abstractos no deben necesitar distinguir entre tipos del lenguaje y tipos del usuario.

Durante esta tesis nos hemos dado cuenta que la dificultad de implementación de las características del lenguaje es un factor suficientemente importante como para influir directamente en el diseño de las mismas. Stroustrup ha mencionado que los problemas más graves del lenguaje

surgieron por incluir cambios a la especificación antes de tener la implementación correspondiente. Incluso el comité de estandarización de C++0x está considerando remover de última hora características que no tienen una implementación experimental [79]. Debido a esto, también hemos establecido metas de diseño con respecto a la implementación. Para nosotros ninguna característica es suficientemente importante como para hacer una excepción a las reglas listadas a continuación:

- *Los análisis léxico, sintáctico y semántico deben estar bien delimitados*: Las diferentes fases del proceso de compilación deben poder ser implementadas de manera independiente ya que esto facilita la codificación de compiladores y otras herramientas como entornos de desarrollo, además de que permite el uso de generadores léxicos y sintácticos convencionales.

- *La gramática del lenguaje debe ser LL(1)*: Los analizadores sintácticos LL(1) son poco poderosos pero son suficientemente simples para que su implementación manual sea viable. El lenguaje Python es LL(1) [80] y este tipo de analizadores son óptimos en tiempo pues realizan el análisis en tiempo lineal al ser capaces de guiar a un autómata de pila determinista con sólo leer un símbolo de la entrada a la vez y sin repetir.

- *Toda característica debe implementarse antes de su inclusión en el lenguaje y su implementación debe ser fácil*: La implementación de una característica es la mejor manera de identificar sus interacciones con el resto del lenguaje. Aunque una característica sea fácil de explicar en términos generales, si es difícil de implementar entonces es difícil de explicarla en su totalidad.

- *La implementación de cualquier característica debe ser eficiente*: La programación genérica complica seriamente la compilación separada tradicional por lo que es necesario que el proceso de compilación pueda llevarse a cabo de una manera excepcionalmente rápida. Todas las características deben tener una implementación eficiente que incluso permita que el lenguaje pueda ser interpretado en lugar de compilado.

5.2 Tipos y declaraciones

Los tipos primitivos se clasifican en los siguientes grupos:

Enteros

`int8` Rango de -2^7-1 a 2^7-1.

`int16`	Rango de $-2^{15}-1$ a $2^{15}-1$.
`int32`	Rango de $-2^{31}-1$ a $2^{31}-1$.
`int64`	Rango de $-2^{63}-1$ a $2^{63}-1$.
`int`	El tipo entero con signo natural de la plataforma.
`uint8`	Rango de 0 a $2^{8}-1$.
`uint16`	Rango de 0 a $2^{16}-1$.
`uint32`	Rango de 0 a $2^{32}-1$.
`uint64`	Rango de 0 a $2^{64}-1$.
`uint`	El tipo entero sin signo natural de la plataforma.

El tamaño de los tipos enteros naturales de la plataforma es normalmente de una palabra. El uso de tipos enteros con tamaños específicos facilita la portabilidad del código y es usual que el programador que considere inadecuados los tipos naturales de la plataforma sepa con certeza cuál tipo cumple sus necesidades.

No existen sufijos para controlar el tipo de las literales enteras ya que no son necesarios; el mecanismo que permite eliminarlos del lenguaje se explicará posteriormente al tratar el tema de las expresiones constantes. Esto implica que la siguiente declaración es válida:

```
int64 n = 34359738368;        // correcto
```

Las literales enteras también pueden escribirse en hexadecimal, en octal o en binario con los prefijos `0x`, `0c` y `0b` respectivamente. Esto elimina la confusión que generan las literales octales en otros lenguajes y proporciona la extensión usualmente útil de expresar números binarios. La literal entera debe comenzar con un dígito pero se permiten guiones bajos como separadores:

```
int64 m = 1_000_000_000;      // correcto
```

Flotantes

`float32`	Corresponde al formato sencillo de IEC 60559.
`float64`	Corresponde al formato doble de IEC 60559.
`float80`	Corresponde al formato extendido de la x87.
`float`	El tipo flotante natural de la plataforma.

El comportamiento de los flotantes es el mismo que en C y C++ excepto que el tipo por defecto de expresiones que involucran números flotantes no forzosamente es el flotante de doble precisión. Esto permite elegir el tipo más adecuado para la plataforma lo cual es especialmente importante para la plataforma x87 la cual, como se comentó anteriormente, siempre ejecuta los cálculos con operandos en precisión extendida.

No existen sufijos para las literales flotantes. Al igual que las literales enteras se pueden utilizar guiones bajos como separadores. Cabe resaltar que la última versión del estándar IEC 60559 denota como `binary32` al tipo anteriormente llamado `single` y como `binary64` al tipo anteriormente llamado `double` por lo que consideramos adecuados los nombres propuestos en el lenguaje presentado en esta tesis.

Caracteres

`char8`	Rango de 0 a 2^8-1.
`char16`	Rango de 0 a 2^{16}-1.
`char32`	Rango de 0 a 2^{32}-1.
`char`	Corresponde a 1 byte, rango mínimo de 0 a 2^8-1.

Todos los tipos de caracter son internamente enteros sin signo lo cual resuelve el problema sobre la incompatibilidad en C++ entre `char` y los tipos `signed char` y `unsigned char` descrito con anterioridad. Justificamos esta decisión en base a que Unicode no asigna valores negativos a ningún carácter del espacio de caracteres definido en ese estándar; la versión 6.0 de Unicode define el espacio de caracteres en el rango de valores de 0 a 1,114,111 lo que a su vez hace necesario proporcionar un tipo `char32` [81].

Al igual que en C++ las literales de caracter se especifican entre comillas simples y también se puede especificar el valor entero del carácter deseado de la misma forma en que se hace esto en C++.

Tuplas

El lenguaje proporciona una manera de crear tipos estructurados sin necesidad de declarar el equivalente a un `struct`. Estos tipos son llamados tuplas y son colecciones heterogeneas de elementos. Una literal de tupla es una lista de expresiones delimitadas por llaves:

```
tuple(int, char) t = { 5, '@' };
print(t[0]);            // imprime 5
```

La definición de la función `tuple` no es especial y puede implementarse en el lenguaje por lo que no es una palabra reservada. Existen tuplas de cero y un elementos y su uso y sintaxis son los mismos que el del resto de las tuplas.

Los elementos de la tupla son accedidos de manera similar a los elementos de un arreglo. Sin embargo el índice debe ser conocido en tiempo de compilación:

```
print(t[aleatorio( )]);
// error, aleatorio( ) no devuelve un valor conocido
```

Esto es necesario puesto que una tupla es heterogenea y no sería posible predecir el tipo del elemento accedido. El uso de memoria de una tupla es el mismo que el de un `struct` en C.

Misceláneos

`bool`	Ocupa el espacio de un char. Almacena 0 o 1.
`void`	Tipo con tamaño 0.

La única diferencia entre el tipo `bool` del lenguaje presentado con el de C++ es que las literales booleanas `true` y `false` no son palabras reservadas pues su semántica es reproducible; esto se explica en el tema de expresiones constantes. Por otra parte, el tipo `void` de este lenguaje es un tipo con tamaño 0 y alineación 1 en lugar de ser un tipo incompleto. El tipo `void` no es especial en este aspecto pues un arreglo de tamaño 0 o una estructura sin miembros también tendrán tamaño 0.

Permitir tipos de tamaño 0 reduce el consumo de memoria de algunas estructuras y elimina en algunos casos la necesidad de insertar relleno adicional debido a los requerimientos de alineación en memoria pero implica que variables diferentes pueden ser asignadas a la misma dirección. No consideramos que esto sea un problema: si las variables tienen tipos diferentes entonces la comparación de direcciones entre variables con tipos diferentes es inválida; si las variables tienen el mismo tipo entonces no existe manera de distinguirlas ya que ninguna de ellas puede almacenar información.

5.3 Estructuras, apuntadores, constantes y arreglos

Estructuras

Las estructuras se declaran con la palabra reservada `type` y son idénticas a los `structs` de C excepto que no es necesario terminar la declaración con `;` y como ya se dijo, pueden ser de tamaño 0 a diferencia de lo que ocurre en C++:

```
type vacio { }

vacio v1;
vacio v2;

v1 = v2;                    // correcto aunque no hace nada
print(sizeof(vacio));       // imprime 0
```

Apuntadores

Existen cambios sintácticos con respecto a C++ en cuanto a la declaración y uso de apuntadores. Un apuntador ahora se declara indicado el tipo entre corchetes y para obtener la dirección de una variable se usa el operador `@`. La desreferencia se realiza también indicando la variable entre corchetes:

```
int n = 0;

[int] p = @n;                 // p apunta a n
[p] = 1;                      // desreferenciar p

print(n);               // imprime 1
```

Una consecuencia de este cambio es que la declaración de múltiples apuntadores en una misma sentencia no requiere repetir el modificador de apuntador por cada variable declarada:

```
[int] a, b, c;          // tres apuntadores a int
```

Además se vuelve posible realizar el análisis sintáctico sin la necesidad de consultar la tabla de símbolos, lo que es necesario en C++ (y no siempre posible debido a las plantillas) para tratar la ambigüedad sintáctica entre una declaración de apuntador y una multiplicación.

El apuntador nulo se denota mediante el identificador `null` que tampoco es una palabra reservada; por el momento mencionaremos que `null` actúa como una constante genérica para apuntadores pero se podrá entender su semántica real al tratar el tema de las expresiones constantes:

```
[int] n = null;         // correcto
[char] c = null;        // correcto también
```

Constantes

Al igual que en C y C++ se proporciona la palabra reservada `const` que previene modificaciones a una variable una vez inicializada. Como en C y a diferencia de C++ una variable declarada `const` es ordinaria en memoria y puede modificarse de manera segura con un moldeado que remueva `const`.

Una diferencia fundamental con C y C++ es que el tipo de una variable declarada como `const T` es `T`; `const` no forma parte del tipo, sólo denota una restricción de acceso sobre la variable. Formalmente `const` modifica lo que nosotros denominamos el enlace de un nombre o referencia en relación a la localidad de memoria que denota.

Se permite el uso de `const` para la declaración de apuntadores que al desreferenciarse devuelven un valor-i con enlace `const`. La distinción sintáctica entre apuntadores `const` y apuntadores a `const` es más clara debido a la nueva sintaxis de declaración de apuntador:

```
[const int] pc;         // apuntador a const int
const [int] cp;         // apuntador const a int
```

Los tipos `[T]` y `[const T]` son diferentes. No obstante, el tipo de la expresión que resulta al desferenciarlos en ambos casos es `T`. La diferencia consiste en que el enlace del valor-i obtenido al desreferenciar un `[const T]` tiene permisos de sólo lectura:

```
[int] p;
[const int] pc;

type([p]) a;            // equivalente a int a;
type([pc]) b;           // equivalente a int b;

[p] = 0;                // ok
[pc] = 0;               // error, el acceso es const
```

El miembro de una estructura es descrito únicamente mediante un tipo y un identificador por lo que `const` no se permite en ese contexto:

```
type prueba {
    const int n;        // error, uso inválido de const
}
```

Arreglos

Los arreglos son similares a los de C++ y C89 ya que no existe verificación implícita al acceder a un elemento y el tamaño debe ser una constante conocida en tiempo de compilación aunque ésta puede ser 0.

Existen dos diferencias sintácticas menores en cuanto a la declaración de arreglos: los corchetes se aplican al tipo de los elementos del arreglo en lugar de al identificador y la inferencia del tamaño del arreglo se indica con un signo de interrogación ? entre los corchetes:

```
int[2] a, b, c;             // tres arreglos
int[?] arr = { 1, 2, 3 };   // se infiere el tamaño
```

Aunque de poca importancia, decidimos usar el indicador ? en lugar de los corchetes vacíos pues en otros lenguajes el tamaño de un arreglo no pertenece al tipo mientras que en este lenguaje sí. Además el operador `[]` puede sobrecargarse y puede actuar unaria o binariamente: unariamente denota la

desreferencia de un apuntador y binariamente el acceso a un arreglo; el uso de los corchetes vacíos podría prestarse a confusión en ese contexto.

Los arreglos, a diferencia de C++, pueden pasarse y devolverse por valor:

```
int[2] f(int[2] a)
{
    a = { 0, 0 };
    return a;
}

int[2] a = { 10, 10 };
int[2] b = f(a);

print(a[0] + b[0]);     // imprime 10
```

Al no existir una conversión implícita a apuntador con la pérdida del tipo real correspondiente que esto implica, no es necesario indicar ninguna de las dimensiones de una matriz usada como parámetro de función:

```
void g(int[?][?][?]);

int[2][2][2] mat;
g(mat);                 // ok, g(int[2][2][2])
```

5.4 Expresiones y sentencias

Como es usual en los lenguajes de programación de sistemas, no se protege al programador de errores como desbordamientos aritméticos, accesos inválidos a memoria y divisiones entre cero. Sin embargo se dan más garantías que en C++ y no se es tan flexible con respecto a las conversiones implícitas.

Orden de evaluación

A diferencia de C++ el orden de evaluación de las expresiones se lleva a cabo de una manera específica y no dependiente de la implementación. El orden se determina mediante la aplicación de las siguientes reglas:

- En una operación binaria se evalúa primero el parámetro izquierdo.
- Todos los operandos de un operador, con excepción de los operadores `&`, `|` y `?`, se evalúan completamente antes de realizar la operación.
- Se respetan las reglas de precedencia y la agrupación de expresiones.
- Los argumentos de una función se evalúan de izquierda a derecha.

Por las reglas anteriores se tiene que el siguiente fragmento de código imprimirá `2` a pesar de modificar `i` más de una vez en la misma sentencia:

```
int i = 0;
print(i++ + ++i);
```

Las mismas reglas de evaluación están presentes en el lenguaje Java [82]. Cabe mencionar que siempre y cuando el comportamiento visible del programa sea el mismo, se permiten realizar optimizaciones que no sigan las reglas de evaluación presentadas.

Operadores básicos

La siguiente tabla muestra la lista de los operadores del lenguaje desarrollado. Están agrupados por orden de precedencia, de la más alta a la más baja.

Operador	**Descripción**	**Asociatividad**

`()`
`[]`
`.`
`->`
`++ --`
	Llamada a función
	Acceso a arreglo
	Selección de miembro
	Selección de miembro mediante apuntador
	Incremento/decremento posfijos
		Izquierda

`++ --`
`+ -`
`! ~!`
`@`
	Incremento/decremento prefijos
	Más/menos prefijos
	Negación lógica/binaria
	Dirección
		Derecha

`^`	Potencia	Derecha
`* / %`	Multiplicación/división/módulo	Izquierda
`+ -`	Suma/resta	Izquierda
`<< >>`	Desplazamiento izquierdo/derecho	Izquierda
`~&`	AND binario	Izquierda
`~+`	OR exclusivo binario	Izquierda
`~|`	OR inclusivo binario	Izquierda

`< <=`
`> >=`
`== !=`
	Menor/menor o igual
	Mayor/mayor o igual
	Igual/diferente
		Izquierda

`&`	AND lógico	Izquierda
`|`	OR Lógico	Izquierda
`?:`	Condición ternaria	Derecha
`=`	Asignación	Derecha

`+=  -=`
`*=  /=`
`%=  ~&=`
`~+= ~|=`
`<<= >>=`
`&= |=`
`^=`
	Suma/resta con asignación
	Multiplicación/division con asignación
	Módulo/AND binario con asignación
	OR exclusivo/inclusivo con asignación
	Desplazamiento izquierdo/derecho con
	AND/OR lógico con asignación
	Potencia con asignación	

Los operadores `&` y `|` son el equivalente a los operadores `&&` y `||` de C++. Además de incorporar las versiones con asignación de estos operadores se ajustó su precedencia para evitar las sorpresas que causaba la precedencia de los mismos en C++. No se proporciona el operador `sizeof` ni una sintaxis especial para moldeados pues ambos son implementables en el lenguaje.

El resultado de la división se define de igual forma que en C99, es decir con redondeo hacia cero y con el signo del residuo igual que el signo del dividendo.

No estamos seguros de cómo definir las operaciones de desplazamiento a nivel bit en los casos en los que algún operando es negativo por lo que sólo las definimos para enteros sin signo, aunque es posible sobrecargar dichos operadores y definir su semántica para otros casos. Algo similar ocurre para el operando derecho del operador `^`, que ahora se usa para exponenciación, en el caso de que ambos operandos sean enteros.

Por último, las expresiones de la forma `a < b < c` se evalúan de manera encadenada como es usual en contextos matemáticos por lo que la expresión anterior será equivalente a la expresión `a < b & b < c`. Es debido a esta característica que se decidió darle la misma precedencia a todos los operadores relacionales.

Conversiones implícitas y explícitas

A diferencia de C++ no se define ninguna conversión implícita que implique pérdida de precisión o modificaciones en la semántica:

```
int a = 3.14;          // mal, pérdida de precisión
int b = '@';           // mal, semántica diferente
```

Sin embargo las conversiones pueden realizarse fácilmente:

```
int a = adapt(3.14);   // ok, adaptar 3.14 a int
int b = adapt('@');    // ok, adaptar '@' a entero
```

La función `adapt` tiene por objetivo definir conversiones cuya semántica es intuitiva (por ejemplo, la conversión de `float` a `int` trunca la parte decimal) y puede implementarse en el lenguaje. Se verá cómo se logra esto al tratar el tema de constructores e inicialización uniforme.

Por defecto la operación de asignación no devuelve nada (aunque puede redefinirse) por lo que el siguiente código no compilará:

```
if (a = b) {            // error, void en condición
      //...
}
```

Ya que la función `adapt` está definida para apuntadores de tipos diferentes es posible implementar el resto de los moldeados fácilmente y de ser necesario, en términos de conversiones que traten a sus operandos como patrones de bits. El lenguaje no proporciona un mecanismo de información de tipos en tiempo de ejecución pues proporcionarlo incurriría necesariamente en una sobrecarga en espacio; es por esto que no existe el equivalente del moldeado `dynamic_cast` de C++.

Sentencias de control

Las sentencias de control definidas en el lenguaje son las siguientes:

Sentencia	Descripción

`if condición {`
`  ...`
`}`
`else if condición {`
`  ...`
`}`
`else {`
`  ...`
`}`
	Condicional simple

`while condición {`
`  ...`
`}`
	Ciclo con condicional verificada al inicio de cada iteración

`do {`
`  ...`
`} while condición;`
	Ciclo con condicional verificada al final de cada iteración

`for inicializacion;`
`  condición;`
`  actualización {`
`  ...`
`}`
	Ciclo con actualización

`case expresión {`
`expresión:`
`  ...`
	Condicional simple mediante elección de casos

`expresión:`
`  ...`
`?:`
`  ...`
`}`	

`break;`	Terminación de ciclo o de caso
`Continue;`	Terminación de iteración
`goto etiqueta;`	Salto incondicional
`return expresión;`	Retorno de función

Los paréntesis para las expresiones de control ya no son necesarios pero en cambio las llaves son obligatorias a pesar de que sólo exista una sentencia en el bloque de la sentencia de flujo; esto evita la ambigüedad del `else` colgante. Además, la falta de paréntesis en la condición de la sentencia `do` hace más evidente por qué que se requiere un `;` al final de la condición.

El comportamiento de la sentencia `case` es equivalente al de la sentencia `switch` de C++ con sentencias `break` al final de cada caso por lo que el siguiente fragmento de código imprimirá `1`:

```
case 1 {
 1: print(1);
 ?: print("desconocido");
}
```

C++ aprovechaba la ejecución en cascada de un `switch` para indicar la situación en la que valores diferentes compartían una misma acción. En el lenguaje presentado esto se expresa mediante una lista de valores separados por comas:

```
case n {
 1, 2, 3:
    print("1, 2 o 3");
}
```

La sentencia `break` sigue teniendo la misma semántica dentro de un `case` mientras que la sentencia `continue` adquiere el significado de salto al siguiente caso de la sentencia lo que permite expresar ejecución en cascada; el siguiente fragmento de código imprimirá `1 2`:

```
case 1 {
 1: print(1);
    continue;
 2: print(2);
    break;
}
```

Las variables declaradas en un case se consideran locales a su caso:

```
case n {
 1: int i = lee( );
    continue;
 2: print(i);          // error, i no declarado
}
```

A diferencia de C++ el lenguaje no permite bloques anónimos por lo que una llave abierta marca el inicio de una literal de tupla. Los bloques deberán recibir un nombre y éstos son los posibles destinos de la sentencia goto:

```
void f( )
{
    bloque {
        print("infinito");
    }

    goto bloque;
}
```

5.5 Referencias y funciones

Referencias

Una referencia sigue indicándose con el operador & aunque existe un cambio sintáctico similar al de los apuntadores pues ahora actúa como prefijo del tipo de la variable en lugar de ser prefijo del identificador.

El cambio más importante es que, al igual que const, una referencia no forma parte del tipo. Formalmente el operador & también es parte del enlace de un nombre y establece una semántica de alias sobre la localidad de memoria denotada:

```
int n;
&int x = n;
&const int y = n;

type(x) a;            // equivalente a int a;
type(y) b;            // equivalente a int b;

x = 0;                // ok
y = 0;                // error, el acceso es const
```

Por lo mismo, es ilegal declarar referencias como miembros de una estructura:

```
type prueba {
    &int n;           // error, uso inválido de &
}
```

Además, y a diferencia de C++, no es posible enlazar un valor-d con una referencia sin importar si es const:

```
&const int n = 5;       // error
```

Sin embargo poder declarar un parámetro que acepte ya sea un valor-d o un valor-i es importante: declararlo siempre con paso por valor generará copias aunque sólo se necesite acceso de sólo lectura; declarar siempre una referencia no permitirá usar un valor-d así sólo se necesite examinar su valor. El lenguaje lo resuelve mediante lo que llamamos comodín de enlace:

```
int acumula(?int[3] arr);
// devuelve la suma de los elementos del arreglo

acumula({ 1, 2, 3 });
// valor-d        ?int[3] ⟶ int[3]

int[3] a;
acumula(a);
// valor-i        ?int[3] ⟶ &int[3]

const int[3] b;
acumula(b);
// valor-i const  ?int[3] ⟶ &const int[3]
```

Funciones

Las funciones son muy similares a las de C++. Sintácticamente su declaración y uso es casi idéntico, se permite la sobrecarga de funciones y también existe el paso y el retorno por valor y por referencia.

A diferencia de C++ no existen las funciones que tomen un número indeterminado de parámetros en tiempo de ejecución. Este mecanismo es reemplazado por la generación de diferentes versiones de la función en base al número de argumentos usados en la invocación:

```
void f(int... n)        // cero o más ints
{
    print(n...);        // expansión de argumentos
}

f( );
/*
    se invoca la versión con cero argumentos

    void f( )
    {
        print( );
    }
```

```
*/

f(1, 2, 3);
/*
    se invoca la versión con tres argumentos

    void f(int _n1, int _n2, int _n3)
    {
        print(_n1, _n2, _n3);       // expansión
    }
*/
```

Esto además de ser seguro en tipos en tiempo de compilación, permite usar una convención de llamada en la que el código de limpieza de la pila se genere del lado del llamado, disminuyendo el tamaño de los ejecutables.

Usar un valor-d como argumento de una función que tome su parámetro mediante paso por valor no generará copias adicionales; se usa el mismo temporal. En términos de lenguaje ensamblador esto requiere implementar el paso por valor como paso por referencia. El paso por valor entonces es simulado creando un temporal del lado del llamante y luego pasando ese temporal por referencia:

```
void f(int);

int n = 0;
f(n);             // conceptualmente una copia de n
```

es equivalente a

```
void f(&int);

int n = 0;
__invocacion_f_por_valor {
    int temp = n; // crear la copia
    f(temp);            // paso por referencia de la copia
}
```

La palabra reservada `move` considera a un valor-i como un valor-d para efectos de la expresión en la que se encuentra:

```
void f(int);

int n;
f(n);
// n es un valor-i
// se crea una copia y ésta es pasada por referencia

f(move(n));
// n es considerado un valor-d
// no se crea la copia y se pasa a n por referencia
```

La palabra reservada `forward` es equivalente a `move` excepto si se aplica a un nombre con enlace de referencia:

```
int n;
f(forward(n));          // equivalente a f(move(n));

&int m = n;
f(forward(m));          // equivalente a f(n);
```

La nueva implementación interna del paso por valor, el comodín de enlace y las palabras reservadas `move` y `forward` permiten dar solución al problema del reenvío que, como se comentó en la subsección 3.4.10, intenta habilitar las optimizaciones de operaciones sobre valores-d en todos los casos:

```
void g(int);            // recibe valores-d
void g(&int);           // recibe valores-i
void g(&const int);     // recibe valores-i const

void f(?int n)          // recibe todo, reenvía a g
{
    g(forward(n));
}

f(aleatorio( ));
// valor-d, invoca g(int):
//    f(?int)           → f(int)
//    g(forward(n))     → f(move(n))

int n;
f(n);
// valor-i, invoca g(&int):
//    f(?int)           → f(&int)
//    g(forward(n))     → g(n)

const int m;
f(m);
// valor-i, invoca g(&const int):
//    f(?int)           → f(&const int)
//    g(forward(n))     → g(n)
```

No existen sobrecargas inherentemente prohibidas como en C++:

```
void f(int);            // valores-d
void f(const int);      // valores-d const

int n;
f(move(n));             // llama a f(int)

const int m;
f(move(m));             // llama a f(const int)
```

5.6 Constructores e inicialización uniforme

Como se mencionó brevemente, se le denomina constructor a una función que es invocada automáticamente al crear una variable y su objetivo es inicializar a la variable con algún valor.

A diferencia de C++ no se tiene la noción de función miembro por lo que los constructores son funciones ordinarias. Un constructor es una función declarada con el nombre `create` y que toma como primer parámetro una referencia a la variable a ser inicializada:

```
type prueba {
    int n;
}

void create(&prueba p)
{
    p.n = 10;
}

prueba p;
print(p.n);             // imprime 10
```

Los constructores pueden tomar parámetros adicionales, sin embargo la notación del operador de asignación = debe habilitarse manualmente:

```
void create(&prueba p, int n)
{
    p.n = n;
}

void operator=(&prueba p, int n)
{
    create(p, n);
}

prueba p = 5;         // ok, error si no existe operator=
```

La función `adapt` usada durante la explicación de las conversiones explícitas en 5.4 puede implementarse de la siguiente manera:

```
type adapta_float {
    float dato;
}

adapta_float adapt(float f)
{
    adapta_float a;
    a.dato = f;

    return a;
```

```
}

void create(&int n, adapta_float a)
{
    // trunca parte decimal de a.dato
}

void operator=(&int n, adapta_float a)
{
    create(n, a);
}

int n = 3.14;           // error, no existe operator=
int n = adapt(3.14);    // correcto
```

El mecanismo descrito arriba vuelve innecesaria la palabra reservada `explicit` de C++. Además, ya que las expresiones delimitadas por llaves tienen un tipo estructurado ordinario es posible implementar constructores que asemejen la sintaxis de inicialización de arreglos:

```
void create(&mio, tuple(int, int));
void operator=(&mio, tuple(int, int));

mio v = { 1, 2 };       // correcto
```

5.7 Espacios de nombres y resolución de funciones

Al igual que C++ atacamos el problema del desarrollo de componentes de software independientes mediante un mecanismo explícito y dirigido a evitar la colisión de nombres mediante prefijos; esto a diferencia de otros lenguajes que usan la noción de archivo como la base fundamental de dicha separación.

Como en C++, un espacio de nombres es un ámbito. Los identificadores declarados en un espacio de nombres se acceden de la misma forma en la que se acceden a los campos de una estructura:

```
space uno {
    void f( );
}

space dos {
    type f { };
}

uno.f( );
dos.f v;
```

Como se comentó en la subsección 3.4.7, la invocación de algunas funciones se vuelve inconveniente si éstas están declaradas dentro de un

espacio de nombres. Sin embargo como se discute en esa misma sección, el algoritmo de búsqueda dependiente de C++ que estaba dirigido a eliminar dichos inconvenientes tiene problemas graves.

Al estudiar este problema bajo la restricción de usar espacios de nombres como principal medio para lograr la separación de componentes llegamos a la conclusión de que se necesita un algoritmo de naturaleza similar a la búsqueda dependiente de C++ por lo que desarrollamos una variante de éste que elimina todos sus problemas conocidos. El algoritmo también se activa cuando no se usa calificación explícita sobre la función y es el siguiente:

- Para cada argumento usado en la invocación a función:
    - Se determina su tipo y el espacio de nombres en donde está declarado dicho tipo.
    - Se buscan en ese espacio todas las funciones con el mismo nombre que la función a invocar y que tengan un parámetro del mismo tipo y en la misma posición que el argumento usado.
    - Se descartan las funciones que no puedan ser invocadas.
    - La información de las funciones restantes se propaga desde su espacio origen hasta al espacio global.
- Se realiza la resolución del nombre desde el punto de la invocación:
    - Si se encuentra alguna función propagada:
        - Se realiza la resolución sobre dicho conjunto y se termina la búsqueda.
    - Si se encuentra alguna función declarada:
        - Se descartan funciones no candidatas.
        - Si existen funciones que no hayan sido descartadas:
            - Se realiza la resolución de sobrecarga sobre ellas y se termina la búsqueda.
    - Si se encuentra algún otro símbolo con dicho nombre:
        - Se usa esa declaración y se termina la búsqueda.
    - Se continúa la búsqueda en el ámbito inferior.

A continuación se presenta un ejemplo:

```
int f;

space biblio1 {
    type tipo {
        //...
    }

    void f(biblio1.tipo v);
}
```

```
space biblio2 {
    void f(biblio1.tipo v);
    void g(biblio1.tipo v)
    {
        f(v);           // llama a biblio2.f
    }
}

space biblio3 {
    void g(biblio1.tipo v)
    {
        f(v);           // llama a biblio1.f
    }
}
```

Como puede verse en el ejemplo anterior, el algoritmo elige la función más cercana al ámbito en el que se realiza la invocación pero considera las intersecciones entre espacios como puntos de referencia para determinar la relevancia de las funciones locales con aquellas que fueron propagadas.

En base a lo anterior se observa que el espacio global es especial al ser el espacio padre de todos los demás; todas las funciones propagadas llegarán al espacio global por lo que la búsqueda dependiente de nombres declarados en bibliotecas diferentes se resolverá en este espacio. Sin embargo el espacio global es tradicionalmente el espacio del usuario por lo que se debe evitar que al declarar un nombre global no relacionado con los nombres usados entre bibliotecas se pueda interferir con la comunicación entre éstas.

Este problema también está presente en la búsqueda dependiente de C++ sin embargo rara vez es mencionado debido a que se ha aceptado como buena práctica de codificación que la búsqueda dependiente sea deshabilitada mediante calificación explícita durante el desarrollo de bibliotecas. Nosotros por otra parte debemos solucionarlo y esto se ha hecho dándole prioridad a las funciones propagadas sobre el resto de las declaraciones encontradas en el punto de búsqueda. Ver figura 3:

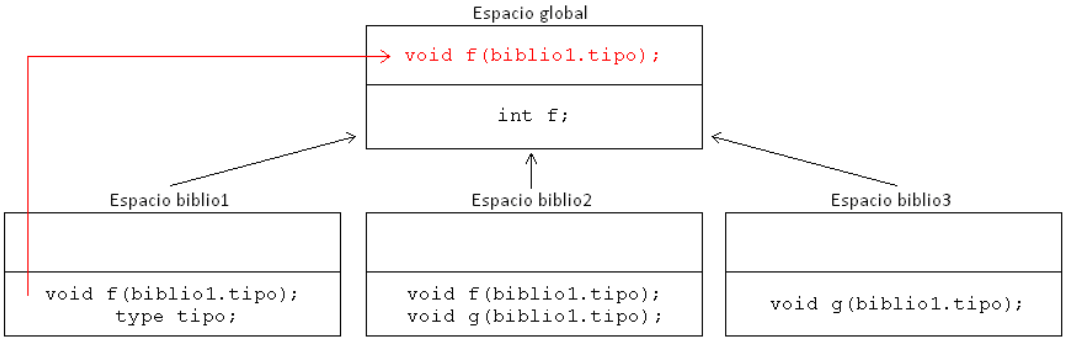


Figura 3. Búsqueda dependiente para la invocación `f(biblio1.tipo)`

La propagación de nombres no deseados sólo puede ser culpa del programador: una función sólo es propagada a los espacios inferiores y es común que los espacios anidados sean desarrollados por el mismo grupo de programadores o que el autor de dicho espacio anidado al menos conozca las declaraciones de la biblioteca que está extendiendo:

```
space mi_biblioteca {
    space version_2 {
        // usualmente codificado por el mismo autor
    }

    space detalle_implementacion {
        // usualmente codificado por el mismo autor
    }
}
```

5.8 Plantillas y expresiones constantes

Nosotros definimos una expresión constante como un valor o entidad conocida por el compilador. Para el lenguaje, cada una de estas entidades tiene su propio tipo de dato y éste es sintetizado mediante la palabra reservada `tag`:

```
5;              // su tipo es tag(5)
int;            // su tipo es tag(int)
operator+;      // su tipo es tag(operator+)
```

Se mantendrá la propiedad `tag` de los argumentos de una llamada a función siempre que esto sea posible. Esto permite definir los constructores de los tipos primitivos en términos de tags:

```
void create(&int8, tag(-128));
void create(&int8, tag(-127));
//...
void create(&int8, tag(127));

int8 n = 10;      // llama a create(&int8, tag(10))
int8 m = 1000;    // error, no existe candidata
```

Otro ejemplo es el siguiente:

```
void f(tag(int));
void f(tag(float));

f(int);           // llama a f(tag(int))
f(char);          // error, no existe candidata
```

Los tipos `tag` se clasifican en cinco grandes grupos:

- Tags que representan constantes: `tag(5)`, `tag('@')`, `tag(3.1416)`.
- Tags que representan tipos: `tag(int)`, `tag(char)`, `tag(bool[2])`.
- Tags que representan funciones: `tag(create)`, `tag(adapt)`.
- Tags que representan espacios: `tag(biblio)`.
- Tags que representan clases: `tag(entero)`, `tag(var)`.

Los identificadores `true`, `false` y `null` son variables `tag` predefinidas en el ámbito global. El tipo representante de un `tag` es el tipo comúnmente usado en tiempo de ejecución para manipular el valor del `tag`. Con la excepción de los tags que representan constantes, el tipo representante de un `tag` es sí mismo. Para los tags que representan constantes (y que nosotros denotamos tags dato) el tipo representante es el siguiente:

- Para tags de enteros, el tipo representante es `int`.
- Para tags de caracteres, el tipo representante es `char`.
- Para tags de flotantes, el tipo representante es `float`.
- Para tags de booleanos, el tipo representante es `bool`.
- Para tags de cadena, el tipo representante es `char[N]` con `N` siendo el tamaño de la cadena.
- Una literal de tupla tiene tipo `tag` si todos sus inicializadores son tags. Su tipo representante es la tupla formada por los tipos representantes de sus elementos.
- Para tags de apuntador, el tipo representante es `[T]` o `[const T]` como corresponda donde `T` es el tipo de la variable apuntada.

Una plantilla de función puede deducir el valor de un `tag` usado para la invocación. Para declarar una plantilla se usa la palabra reservada `template` seguido de la lista de parámetros de plantilla entre paréntesis (en lugar de usar paréntesis angulares como en C++):

```
template(type T)
void f(tag(T));   // deducción de tags tipo

template(int N)
void g(tag(N));   // deducción de tags enteros

template(space S)
void h(tag(S));   // deducción de tags espacio

f(int);           // ok, T = int
f(char);          // ok, T = char

g(5);             // ok, N = 5
g(lee( ));        // error, lee( ) no regresa un tag

h(biblio);        // ok, S = biblio
h(int);           // error, int no es un espacio
```

Puede observarse que a diferencia de C++ y otros lenguajes, los valores de los parámetros de plantilla siempre pueden inferirse mediante la lista de argumentos ordinaria. No es necesario pasar los argumentos constantes mediante una lista separada (y no proporcionamos una forma de hacerlo):

```
f<int>( );        // C++, C#, Java
f!(int)( );       // D
f[int]( );        // Scala
```

Además de reducir las variantes de la sintaxis de llamada a función, se pueden proporcionar algoritmos especializados que aprovechen de manera transparente que ciertos valores son conocidos en tiempo de compilación:

```
template(int N)
int[N] lee_n(tag(N))
{
    // lee N enteros y regresa un arreglo ordinario
}

lista(int) lee_n(int n)
{
    // lee n enteros pero necesita memoria dinámica
    // pues el valor de n no es conocido de antemano
}

lee_n(10);              // llama a lee_n(tag(10))
lee_n(aleatorio( ));    // llama a lee_n(int)
```

Las plantillas también permiten inferir el tipo de los argumentos como en C++:

```
template(type T)
void intercambia(&T, &T);

int a, b;
intercambia(a, b);      // correcto, T = int
```

En el caso de argumentos `tag`, el tipo inferido es el tipo representante del `tag`:

```
template(type T)
void f(T);

f(5);                   // correcto, T = int
```

A diferencia de C++ el valor de `T` se intenta unificar en casos como el siguiente:

```
template(type T)
T minimo(T, T);

int n;
float f;
```

```
minimo(n, f);          // ¿correcto? sí, T = float

// Sí existe la conversión implícita int ⟶ float
// No existe la conversión implícita float ⟶ int
```

Para determinar el tipo de la unificación se construye un grafo dirigido donde los nodos representan los tipos candidatos y una arista del tipo `T1` al tipo `T2` significa que la declaración `T2 a = b` con `T1` como el tipo de `b` compila. El tipo de la unificación es aquél al que se pueda llegar partiendo de cualquier nodo usando una sola arista y que sea el único con esta propiedad.

5.9 Organización física de programas

El proceso de instanciación de las plantillas de C++ tiene como consecuencia que la resolución de algunos símbolos referidos en una plantilla dependan del momento en que se hace la instanciación. En un caso extremo, la instanciación de una plantilla puede tener una semántica diferente después de procesar la última declaración presentada al compilador (hablando en términos de código fuente). Nosotros además hemos decidido que el orden de las declaraciones de funciones y tipos no sea relevante:

```
void f( )
{
    g( );      // correcto aunque g esté declarado abajo
}

void g( );
```

Si además deseamos evitar los inconvenientes presentados en 4.4 con respecto al uso de archivos múltiples en otros lenguajes, proporcionar un mecanismo de compilación separada se complica de manera importante. Es por esto que el lenguaje presentado en esta tesis no se preocupa por la compilación separada y asume que todo el código de la aplicación está presente. Sin embargo esto no es estrictamente necesario pues el análisis léxico-sintáctico de cada archivo puede realizarse de manera independiente y el resultado puede ser almacenado para su posterior uso; esto es equivalente al uso de cabeceras precompiladas provistas por muchas implementaciones de C y C++ y que reducen de manera importante el tiempo de compilación [83].

En el lenguaje presentado, para iniciar el proceso de compilación de un programa desarrollado sólo se necesita indicarle a la implementación la ubicación del archivo fuente raíz. Desde este archivo el usuario puede solicitar la inclusión de archivos adicionales mediante la palabra reservada `import`:

```
// archivo_raiz.txt

import "otro_archivo.txt";

void f( )
{
    g( );
}

// otro_archivo.txt

void g( );
```

La inclusión repetida de un archivo durante el proceso de compilación es ignorada. Esto permite que la implementación del lenguaje pueda construir un grafo ordenado con la información de cuáles archivos fueron incluídos y en qué orden. Posteriormente el código fuente se concantenará (conceptualmente) en el orden indicado por el grafo para continuar con el proceso de compilación. Las variables globales son inicializadas en el orden en el que las declaraciones son presentadas después de procesar las inclusiones:

```
// archivo1.txt

import "archivo2.txt";
int n = aleatorio( );

// archivo2.txt

import "archivo1.txt";
import "archivo3.txt";
int m = aleatorio( );

// archivo3.txt

int r = aleatorio( );
```

En el ejemplo anterior se inicializarán las variables globales en el orden `r`, `m`, `n` si el archivo inicial del proceso de compilación es `archivo1.txt`.

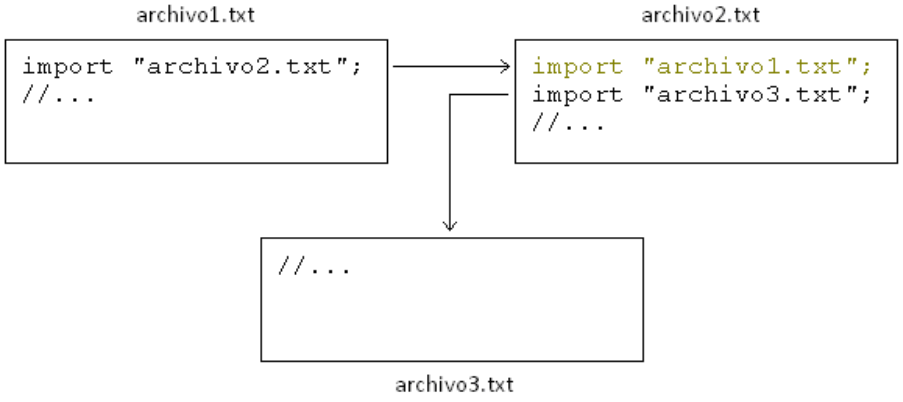


Figura 4. Orden de inclusión desde `archivo1.txt`

Realizar la compilación del programa completo permite tratar todos los problemas descritos en la subsección 3.4.9: se logra simplificar hasta el grado de volver trivial la verificación de la regla de la definición única, ya no existe la separación entre búsqueda de símbolos dependientes y no dependientes de los parámetros de plantilla además de que se garantiza que la instanciación repetida de una plantilla con los mismos parámetros tendrá siempre la misma semántica independientemente de los puntos de instanciación.

La palabra reservada `using` indica al compilador que si la búsqueda de un símbolo falla, se busque también en el espacio de nombres indicado:

```
using biblio;

space biblio {
    void f( );
}

void g( )
{
    f( );
    // no se encuentra f en la búsqueda ordinaria
    // se busca en biblio y se llama a biblio.f( )
}
```

Esto permite omitir las calificaciones explícitas de espacios de nombres en muchas situaciones y es útil para programadores principiantes. La sentencia `using` sólo afecta al archivo en el que se encuentra y no afecta la semántica del programa, sólo hace válidos algunos programas que no lo serían sin la presencia de dicha sentencia.

5.10 Clases y programación genérica

La característica de clases del lenguaje presentado en esta tesis es una característica distinta a lo que se le denomina clase en lenguajes como Java o C#. Son similares a los conceptos de C++0x aunque con sintaxis y semántica diferentes.

Una clase se declara con la palabra reservada `class` y consta de una serie de requerimientos en la forma de prototipos de función. Como primera declaración dentro de una clase se debe indicar el nombre simbólico con el que se hará referencia al tipo del cual se desea saber si cumple con los requerimientos de la clase:

```
class entero {
    using T;            // T es el tipo a comprobar
```

```
    T operator+(T, T);
    T operator-(T, T);
    T operator*(T, T);
    T operator/(T, T);
    T operator%(T, T);

    T operator+(T);
    T operator-(T);
}
```

Algunas clases pueden ser casos más refinados de otras; esto ocurre cuando una clase cumple con todas las restricciones de otra y además es posible que tenga restricciones adicionales. A la clase refinada se le denomina subclase y a las clases que ésta refina se les denomina superclases. Las superclases se indican después del nombre simbólico del tipo a verificar. El cumplimiento de las superclases se verificará en orden indicado antes de verificar las restricciones de la subclase:

```
class mamifero {
    //...
}

class cuadrupedo {
    //...
}

class gato {
    using T : mamifero, cuadrupedo;
}
```

Al declarar una variable se puede inferir el tipo de la variable declarada si en lugar del tipo se proporciona una clase y la variable tiene un inicializador:

```
class tiene_f {
    using T;

    void f(T);
}

type tipo_si { /*...*/ }
type tipo_no { /*...*/ }

void f(tipo_si)
{
    //...
}

tipo_si s1;
tiene_f s2 = s1;
// correcto, el tipo de s1 cumple tiene_f
// equivalente: tipo_si s2 = s1;

tipo_no n1;
```

```
tiene_f n2 = n1;
// error, el tipo de n1 no cumple tiene_f
```

Lo anterior también funciona al indicar el retorno y los parámetros de una función. Al igual que la inferencia, las clases usadas en la descripción de los parámetros son sustituidas por los tipos reales usados en la invocación y se genera la versión correspondiente de la función:

```
tiene_f g(tiene_f v)
{
    return v;
}

tipo_si s;
g(s);       // correcto, g(tiene_f) ⟶ g(tipo_si)

tipo_no n;
g(n);       // error, tipo_no no cumple tiene_f
```

La información sobre clases y subclases también es usada para guiar la resolución de sobrecarga de funciones y así elegir la función más específica:

```
class tiene_f_g {
    using T : tiene_f;

    void g(T);
}

type prueba {
    //...
}

// declaración de f y g para prueba

void h(tiene_f);
void h(tiene_fg);

prueba v;
h(v);
// candidatas:
//   h(tiene_f)
//   h(tiene_fg)
//
// llama a h(tiene_fg)
//   tiene_fg es subclase de tiene_f
```

A diferencia de los conceptos de C++0x, el uso de esta característica no restringe las operaciones que pueden realizarse sobre las variables:

```
class var {
    // sin restricciones
}
```

```
void suma(var a, var b)
{
    print(a + b);
}

int a, b;
suma(a, b);             // correcto
```

En la inferencia de parámetros de plantilla también se pueden restringir los valores de dichos parámetros mediante clases:

```
template(type T : con_asignacion)
void intercambia(&T a, &T b)
{
    T c = a;

    a = b;
    b = c;
}

template(type T : apuntador)
void prueba(tag(T) v)
{
    print("nombre de un tipo que es un apuntador");
}
```

La característica de clases en conjunto con el resto de las características descritas presentan un panorama que consideramos completo para la programación genérica. Retomando los puntos discutidos en 4.4 podemos verificar que las debilidades presentes en otros lenguajes no están presentes en el lenguaje descrito en esta tesis:

1. *Los algoritmos pueden sobrecargarse*: Versiones diferentes del mismo algoritmo pueden tener el mismo nombre lo que posibilita que la especialización de los algoritmos sea transparente para el usuario.

2. *Un algoritmo genérico es igual de eficiente que uno no genérico*: Al igual que C++ se genera una versión específica de la función en base a los valores de los parámetros de plantilla y a la sustitución de las clases con los tipos reales usados en la invocación.

3. *Es posible expresar tipos dependientes*: Poder enviar y recibir tipos como argumentos y valores de retorno permite describir relaciones entre tipos en términos de llamadas a función.

4. *Se permiten especializaciones parciales de algoritmos*: Ya que se permite la sobrecarga de funciones siempre y cuando éstas difieran de alguna manera en la descripción de sus parámetros se pueden describir tanto la

versión general como versiones más particulares del mismo algoritmo mediante el uso de plantillas y clases.

5. *La agrupación de tipos en torno a las estructuras algebraicas que modelan es posible y es implícita*: La verificación del cumplimiento de las restricciones de una clase por un tipo es automática. Si un tipo no cumple las restricciones de una clase sólo es necesario implementar las funciones faltantes.

6. *Existe un orden parcial en torno a la agrupación de tipos*: La información de clases y subclases permite crear jerarquías basadas en las propiedades de los tipos. La resolución de sobrecarga toma en cuenta dicha jerarquía.

7. *Existe una manera general de agregar funcionalidad a un tipo*: El lenguaje no tiene la noción de función miembro, sólo de funciones ordinarias. Cualquier usuario podrá extender la funcionalidad de un tipo y esto es suficiente pues las restricciones de una clase son expresadas exclusivamente en términos de prototipos de función.

6 La implementación del intérprete

En este capítulo se discute la implementación de un intérprete para el lenguaje descrito en el capítulo 5. Se exponen las metas de implementación y se describen las etapas en las que fue dividida ésta. Se discuten los problemas encontrados durante la implementación y cómo fueron resueltos. Se presentan algunas propiedades interesantes del intérprete desarrollado como ser capaz de ejecutar parcialmente programas mal formados además de que se hacen algunas comparaciones de rendimiento con un par de implementaciones de C y C++.

6.1 Metas de implementación

Como se comentó en 5.9 la compilación separada es un problema muy difícil de atacar en presencia de la programación genérica. Debido a esto el lenguaje presentado en el capítulo 5 decide ignorarla; a cambio de eso el lenguaje está diseñado para ser compilable en fases bien definidas, independientes y fáciles de implementar. La metas de implementación del intérprete están dirigidas a demostrar que lo anterior es cierto:

- *Debe ser práctico procesar todo el código de una aplicación grande sin usar compilación separada:* Independientemente de las posibles estrategias de preprocesamiento individual de los archivos fuente de un programa, el análisis semántico potencialmente necesita conocer todas las declaraciones del programa. La implementación debe tomar esto en cuenta y no ignorar el consumo de memoria durante la compilación.

- *Las diferentes etapas de compilación deben ser independientes*: Debido al preprocesador de C el análisis léxico en ese lenguaje debe ser secuencial. Su análisis sintáctico además necesita de la tabla de símbolos por lo que también es secuencial y se entremezcla con el análisis semántico. En el lenguaje presentado es posible realizar el análisis léxico-sintáctico de los diferentes archivos fuente paralelamente siempre y cuando el análisis semántico sea independiente.

- *Debe ser práctico ejecutar código de manera interpretada*: En un intérprete el tiempo total de ejecución de un programa incluye el tiempo requerido para traducir o compilar el código fuente; este proceso debe llevarse a cabo rápidamente, de otro modo no existiría ninguna ventaja con respecto a la compilación tradicional.

- *La implementación debe ser sencilla y razonablemente pequeña*: Los errores presentes en compiladores pueden ser difíciles de identificar y reproducir además de que no siempre existe solución provisional que los usuarios puedan o les sea conveniente usar. Un compilador sencillo y pequeño tendrá menos errores y será más fácil de modificar.

Las metas presentadas hacen poca distinción entre un compilador y un intérprete. Un intérprete implementado como un compilador justo a tiempo será muy similar a un compilador ordinario excepto en la etapa de generación de código pues este último usualmente tiene más tiempo disponible para optimizar el código generado.

6.2 Análisis léxico

Usualmente los tokens de un lenguaje de programación son descritos mediante las expresiones regulares que sirven para reconocerlos:

Literal entera decimal: `[0-9]([0-9]|_)*`

Lo anterior hace posible implementar el análisis léxico utilizando reconocimiento de patrones mediante expresiones regulares. Sin embargo esta implementación no es la más rápida pues involucra construir (generalmente en tiempo de ejecución) el autómata finito que las reconozca. Construir y optimizar el autómata en tiempo de compilación, por otro lado, sería similar en eficiencia a generar un analizador léxico parecido al siguiente:

```
// reconocer las palabras reservadas "tag" y "type"

si siguiente( ) = 't'
    si siguiente( ) = 'a'
        si siguiente( ) = 'g'
            si siguiente( ) no es una letra
                regresar "tag"
    sino si siguiente = 'y'
        si siguiente = 'p'
            si siguiente = 'e'
                si siguiente( ) no es una letra
                    regresar "type"
```

Aunque la sentencia `switch` de C++ puede ser útil, el compilador debe generar código para manejar el caso en el que ninguna opción del `switch` corresponde al valor proporcionado. Por lo anterior, la sentencia `goto` puede resultar en ejecutables más rápidos y compactos:

```
void* saltos[] = { &&etiq1, &&etiq2 };          // C++
```

```
goto saltos[aleatorio( ) % 2];

etiq1: /* hacer algo */;
etiq2: /* hacer algo */;
```

Este uso de la sentencia `goto` no es estandar ni en C ni en C++ pero es permitido en los compiladores gcc y g++ [84]. Una estrategia similar de implementación es usada por el generador léxico re2c y el código resultante es más eficiente que el código generado por otros analizadores léxicos [85]. En esta tesis se implementó un generador léxico similar a re2c aunque de uso específico para la implementación del lenguaje desarrollado en esta tesis.

A diferencia de muchos analizadores léxicos, el archivo fuente es leído completo en memoria. No consideramos que esto sea un problema ya que los archivos fuente editados por humanos son usualmente pequeños (menores a 10,000 líneas de código lo que ocupa menos de 1MB para líneas de 80 bytes). Esto permite que el análisis léxico sea más eficiente al poder usar un apuntador ordinario para leer la información del archivo ya en memoria.

El resultado del análisis léxico es un arreglo de tokens que será usado como entrada para el analizador sintáctico. Es posible entremezclar ambas fases sin muchos inconvenientes, sin embargo esto no se ha hecho así en la implementación realizada.

6.3 Análisis sintáctico

Como se explicó en el capítulo 5 el lenguaje presentado en esta tesis es LL(1) lo que permite usar analizadores sintácticos que se ejecuten en tiempo lineal con respecto al tamaño del archivo fuente. Por otra parte, un problema relativamente grave de las gramáticas LL(1) es que el nivel de factorización requerido es excesivo:

declaración =
declaración_variable | declaración_función ;

declaración_variable =
expresión nombre [inicializador];

declaración_funcion =
expresión nombre parámetros ;

// error, declaración =
// (expresión nombre inicializador) | (expresión nombre parámetros) ;

La gramática anterior no es LL(1) ya que ambas opciones de la regla *declaración* inician con *expresión nombre* al desarrollarse. Lo anterior puede factorizarse de la siguiente forma para transformar la gramática a su forma LL(1):

declaración =
declaración_prólogo ([inicializador] | parámetros) ;

declaración_prólogo =
expresión identificador ;

Esto puede complicarse para gramáticas más grandes. Además, muchos generadores sintácticos proporcionan una manera de inyectar acciones semánticas en torno a las reglas de la gramática por lo que requerir una factorización excesiva puede causar la aparición de reglas que no tienen una acción semántica bien definida y dificulta la creación de reglas que sí las tengan.

Ya que nosotros hemos implementado el analizador sintáctico manualmente la observación anterior no es de importancia. Sin embargo debido a dichos inconvenientes la gramática del lenguaje no tiene por qué estar expresada en su forma LL(1). Esto no implica que el lenguaje no sea LL(1).

El resultado del análisis sintáctico es un árbol abstracto de sintaxis: el análisis realizado procesa la información de precedencias y paréntesis y deja el árbol en un estado en el cual es suficiente una tabla de símbolos y visitar sus nodos en orden para emitir código.

Como se mencionó en la subsección 2.4.3 el análisis de la precedencia de operadores guiado mediante gramáticas usualmente requiere una llamada anidada por cada nivel de precedencia:

expr = expr_precedencia1 ;

expr_precedencia1 =
expr_precedencia2 ["|" expr_precedencia_2] ;

expr_precedencia2 =
expr_precedencia3 ["&" expr_precedencia_3] ;

expr_precedencia3 =
expr_precedencia4 [("<"|"<="|">"|">="|"=="|"!=") expr_precedencia4] ;

//…

expr_precedencia10 =
término [“^” término]

El análisis de la expresión `1^2` requerirá diez llamadas anidadas partiendo de la regla *expr*. El algoritmo *shunting yard* de Edsger Dijkstra también puede realizar el análisis de precedencias y lo hace transformando la expresión a su notación postfija [86], sin embargo consideramos que el algoritmo es complicado. Afortunadamente existe un tercer algoritmo llamado precedencias escalonadas [87], muy similar al algoritmo basado en gramáticas y que evita llamadas anidadas que no realizan ningún trabajo:

```
expr(precedencia p):
   leer el término t
   leer el operador r

   si prec(r) ≥ p
       regresa nodo_binario(t, r, expr(prec(r) + 1))
   sino
       regresa nodo_unario(t)
```

Este algoritmo verifica la precedencia del siguiente operador y llama directamente a la función que represente la regla *expr_precedencia* correspondiente evitando así las llamadas de precedencias intermedias que sólo redirigen a la precedencia siguiente.

6.4 Análisis semántico y generación de código

El análisis semántico está dividido en varias etapas debido a que hemos decidido que el orden de declaración de clases, espacios, funciones y tipos no afecte la validez o semántica de un programa. Además, por razones que se explicarán posteriormente la generación de código está entremezclada con el análisis semántico.

Las etapas de análisis semántico son:

- *Creación del árbol de declaraciones*: Se procesan las inclusiones de archivos lo que invoca recursivamente el análisis léxico, sintáctico y esta fase del análisis semántico para los archivos fuente adicionales. Una vez procesadas las inclusiones hechas por un archivo se extrae la información de las declaraciones encontradas en éste a nivel de espacio. Se crea y se extiende el árbol de espacios. El orden de inicialización de variables declaradas a nivel de espacio es determinado.

- *Generación de código de inicialización para variables globales*: Se determinan los tipos de las variables globales, se reserva memoria para ellas según requieran y se emite código de inicialización. Al llegar a esta fase ya se cuenta con la información de todas las declaraciones del programa lo que permite determinar la acción a realizar al invocar funciones en este punto. Si durante el análisis de las funciones llamadas se hace referencia a una variable global todavía no inicializada se genera un error de dependencia cíclica y la compilación termina.

- *Inicio de la compilación a partir del punto de entrada*: Se compila la función que represente el punto de entrada del programa. Esto invoca recursivamente el proceso de compilación para todas las funciones a las que se haga referencia desde dicha función inicial. Una consencuencia es que funciones no referenciadas no serán analizadas semánticamente.

- *Ejecución del código generado*: Se liberan todos los recursos usados durante la compilación con la excepción de la memoria asignada para variables globales y el código generado. Se invoca el código generado para la inicialización de las variables globales y se invoca el código de la función de entrada del programa.

Al declarar una variable de un tipo o estructurado o tupla se procesa la información de dicho tipo (si esto no se ha hecho anteriormente). Este proceso involucra calcular la memoria requerida por los tipos de sus miembros. Los tipos primitivos y apuntadores tienen un tamaño conocido de antemano:

```
type prueba1 {
   [prueba1] p;
   // válido aunque se desconozca el tamaño de prueba1
}

type prueba2 {
   prueba2 p;
   // ok pero se generará un error posteriormente
}

prueba1 v1;             // ok
prueba2 v2;             // error, dependencia cíclica
```

El tamaño de `prueba2` no puede calcularse al depender de sí mismo pero esto no se detecta hasta el punto en el que se declara una variable de dicho tipo. Esto se ha hecho así en parte porque determinar la dependencia cíclica de tipos inmediatamente después de procesar su declaración no siempre es fácil:

```
type prueba3 {
   prueba4 p;                // ¿correcto? no
}
```

```
type prueba4 {
   prueba3 p;                // dependencia cíclica
}
```

Además no se desea ralentizar el proceso de compilación en el caso de tipos para los que nunca se hace referencia dentro del programa; lo mismo se hace para clases y funciones. Esto quiere decir que la implementación podrá ejecutar algunos programas estrictamente inválidos:

```
void f( )
{
   (5 + 2) a = "\2fsfs";
   // error semántico: tipo inválido en declaración
   // error léxico: secuencia de escape desconocida
}

void main( )
{
   // punto de inicio del programa
   // no se llama a f, el programa podrá ejecutarse
}
```

En cuanto a la dependencias cíclicas en definiciones, éstas son muchas veces causa de contradicciones y paradojas matemáticas que evidencian algunos problemas no computables [88, 89]. En el lenguaje presentado en esta tesis existen lugares adicionales en los que se deben evitar las dependencias cíclicas:

```
var f(float f)
{
    return int;
}

var f(f(5) n)           // ¿qué función f es llamada?
{
    return f(5);        // ¿cuál es el tipo de retorno?
}


class auto_clase {
    using T;            // evitar refinar auto_clase

    auto_clase g(T);
    // tal vez alguna condición que int no cumple
}

int g(int);             // ¿int cumple auto_clase?
```

En el caso de `auto_clase`, si durante la verificación de la restricción `g` se decide por decreto que `int` cumple `auto_clase` entonces se deberán verificar las siguientes restricciones; si alguna de estas falla entonces nuestra suposición inicial era incorrecta. Por otro lado si se decide por decreto que `int` no cumple `g`

podría ocurrir que ésa era la única restricción que causaría que `int` no cumpliera `auto_clase`.

La dependencia cíclica de funciones se evita bloqueando las definiciones de las sobrecargas mientras se determina el tipo de retorno y el tipo de los parámetros de alguna de ellas. La dependencia cíclica de clases en la lista de superclases se realiza de una manera similar a la verificación de autorreferencia para tipos (usando variables bandera para identificar si se llega recursivamente a la misma situación); la verificación en los tipos de retorno de los prototipos usados como restricciones es trivial.

Uno de los objetivos del intérprete es demostrar que el proceso de compilación del lenguaje puede llevarse a cabo rápidamente. Debido a esto se decidió priorizar la generación de código correcto sobre la generación de código eficiente. Esto tiene algunas consecuencias:

- *Se intenta emitir código en una sola pasada*: Aunque las sentencias de flujo crean algunas dificultades al no poder predecir siempre las direcciones a usar en las instrucciones de salto, esto puede corregirse fácilmente emitiendo una instrucción marcador para posteriormente sólo sobreescribirla ya con la dirección correcta.

- *Se emite código nativo, no código ensamblador*: Emitir ensamblador agregará sobrecarga adicional al proceso de interpretación al tener que invocar un ensamblador, el cual generalmente será un proceso externo. Implementar esto eficientemente requiere de comunicación interproceso. Aunque es posible hacerlo, es indeseable para los fines de esta tesis.

- *Se evitará consumir tiempo en la optimización del código generado*: No es buena idea intentar demostrar que el lenguaje presentado es tan rápido como C usando una implementación interpretada. Además nos sería imposible desarrollar un optimizador que iguale las optimizaciones realizadas por compiladores comerciales de C y C++.

Prácticamente todas las optimizaciones realizadas por compiladores requieren una o más pasadas sobre una representación intermedia del código y a veces una pasada adicional sobre la representación en ensamblador.

En parte por esto, se investigó si era conveniente delegar la generación de código a alguna herramienta externa. El proyecto LLVM [90] consiste en un conjunto de bibliotecas destinadas al desarrollo de intérpretes y compiladores de alto rendimiento y ha sido usado para desarrollar una nueva implementación de Python denominada Unladen Swallow [91], capaz de

compilar justo a tiempo las rutinas seleccionadas. Sin embargo los autores de Unladen Swallow encontraron que LLVM consume una gran cantidad de tiempo y memoria durante la generación de código nativo aún sin solicitar la optimización del código [92] por lo que decidimos implementar la generación de código nosotros mismos, aunque fuera poco sofisticada.

Se decidió implementar la evaluación de expresiones constantes en tiempo de compilación pues es necesaria en muchos contextos:

```
int[2 + 3] arr;       // debe ser válido
```

Curiosamente esto puede implementarse usando el propio lenguaje:

```
tag(10) operator+(tag(0), tag(10));       // 0 + 10
tag(10) operator+(tag(1), tag(9));        // 1 + 9
tag(10) operator+(tag(2), tag(8));        // 2 + 8
//...
```

Desafortunadamente el número de funciones a definir es infinito no contable. Por esto la implementación crea las definiciones correspondientes al momento de su uso. Esto permite que la optimización sea transparente y no sea un caso especial ni en el lenguaje ni dentro del análisis semántico.

Adicionalmente se consideró implementar la asignación de registros y se encontraron algunos algoritmos que se ejecutan en tiempo lineal y que son de una sola pasada. Un algoritmo adecuado, el de exploración lineal [29], consiste en dar prioridad a las variables cuya declaración sea la más cercana:

```
// suponer la existencia de los registros A y B

int v0;           // se asigna al registro A
int v1;           // se asigna al registro B
int v2;           // se asigna al registro A, volcar v0
```

El algoritmo favorece así las variables las cuales considera se usan más frecuentemente. Esto es usualmente cierto en los ciclos anidados y aunque se requiere de un análisis del ámbito de las variables éste de todos modos es necesario para determinar los lugares en los que se debe emitir código para construir y destruir las variables. Sin embargo un problema aparece con el uso de apuntadores:

```
[int] p = @m;
int n = 1;              // se asigna a un registro

while n++ > 0 {         // ¿se usa el registro de n?
    if aleatorio( ) == 0 {
        p = @n;
```

```
    }

    if aleatorio( ) == 0 {
        [p] = 0;
    }
}
```

Sólo una variable en memoria posee dirección, no una variable en un registro por lo que solicitar la dirección de una variable almacenada en alguno de éstos dentro de una sentencia iterativa puede entorpecer la generación de código aún para instrucciones que sean léxicamente anteriores a la solicitud de la dirección de la variable. Aunque es posible realizar un análisis que intente maximizar el uso de los registros aún en estos casos, la generación de código no podrá ser de una sola pasada. Debido a esto se decidió no implementar la asignación de registros. No consideramos que esto sea un problema grave pues la primera implementación del motor de ECMAScript V8 presentada en 2008 tampoco la implementaba y a pesar de ello fue bien recibida al ser la implementación más rápida de ECMAScript conocida hasta ese momento [23].

Aunque para el lenguaje no existen restricciones de rango para un `tag` entero, la implementación desarrollada tiene impuesto un límite de 64 bits:

```
tag(20_000_000_000_000_000_000) grande;
// válido en el lenguaje, aún no implementado
```

Bajo algunos sistemas operativos se necesita hacer una llamada especial al sistema para obtener memoria con permisos de ejecución. Ésta constituye la única dependencia del intérprete con el sistema operativo que no está contemplada en la bibliotecas estándar de C y C++. La generación y ejecución del código por otro lado es trivial:

```
char* ejecutable = pide_memoria_ejecutable(2);

ejecutable[0] = 0x90;   // NOP, instrucción de x86
ejecutable[1] = 0xC3;   // RET, instrucción de x86

void(*llamar)( ) = (void(*)( ))ejecutable;
llamar( );              // llamar al ejecutable
```

Gracias a que la invocación de código generado en tiempo de ejecución es realmente algo muy sencillo de hacer usando C y C++ es posible reutilizar la pila de ejecución asignada al intérprete para la ejecución del programa del usuario; sólo es necesario emitir código que manipule dicha pila correctamente aunque los requerimientos de alineación de las variables deben tenerse muy presentes pues desafortunadamente algunas interfaces binarias muy usadas difieren en algunos aspectos. Esto puede crear problemas de interoperabilidad entre código ensamblador generado por diversos autores.

El número de operaciones primitivas definidas en C++ es considerable:

- Conversiones entre enteros: `int` y `short`, `int` y `long`, `int` y `long long`, `int` y `unsigned short`, etc.
- Conversiones entre flotantes: `float` y `double`, etc.
- Conversiones entre caracteres: `char` y `unsigned char`, `char` y `signed char`, etc.
- Conversiones entre tipos diferentes: `int` y `char`, `int` y `float`, etc.
- Operaciones entre tipos: `operator+`, `operator*`, `operator<`, etc.

Tanto C como C++ permiten incrustar código ensamblador mediante la palabra reservada `asm`. Esto es útil ocasionalmente pues permite usar instrucciones de la plataforma no conocidas por el compilador o interactuar directamente con un dispositivo de hardware.

```
void f( )                                          // C++
{
    int n;

    // gcc, salto de línea es separador de instrucción
    asm("PUSH EAX\n\tPOP EAX");

    // dmd, sintaxis no usa literales de cadena
    asm {
        MOV EAX, n
    }
}
```

La sintaxis usada para incorporar fragmentos de código ensamblador es dependiente de la implementación en C++. El lenguaje presentado en esta tesis también soporta esta característica aunque con una sintaxis diferente:

```
void f( )
{
    int n;

    asm {
        "PUSH EAX";
        "POP EAX";
        "MOV EAX, [@n]";
    }
}
```

Desafortunadamente C++ prohibe la sobrecarga de operadores entre tipos primitivos. Nosotros sí lo permitimos y esto abre la posibilidad de definir dichas operaciones directamente en el lenguaje. Sin embargo y como ya se mencionó, el intérprete desarrollado no tiene dependencias externas por lo que la implementación de esta característica está incompleta y las literales de cadena

usadas dentro de un bloque `asm` se inyectan directamente al flujo de código nativo usado internamente por el intérprete:

```
void f( )
{
    int n;

    asm {
        "\x50";              // PUSH EAX
        "\x58";              // POP EAX
        "\x8B\x45\xFC";      // MOV EAX, [EBP - 4]
    }
}
```

Aunque impráctico, fue suficiente para poder implementar algunas de las operaciones de los tipos primitivos sin necesidad de modificar el intérprete y así realizar pruebas de funcionamiento. Esto además permite aprovechar las capacidades del lenguaje para disminuir la cantidad de primitivas a definir manualmente:

```
bool operator==(int, int);    // en ensamblador
bool operator==(int8, int8);  // en ensamblador
// definiciones adicionales para tipos iguales

template(type T, type U)
var tipo_mas_grande(tag(T), tag(U))
{
    return (sizeof(T) > sizeof(U) ? T : U);
}

class entero {
    //...
}

template(type T : entero, type U : entero)
bool operator==(T a, U b)
    // comparación para enteros de tipos diferentes
{
    tipo_mas_grande(T, U) x = a;
    tipo_mas_grande(T, U) y = b;

    return x == y;
}
```

La ausencia de un optimizador efectivo, por otra parte, causa que el código generado sea sumamente ineficiente: comparar un `int8` con un `int16` requerirá realizar ocho llamadas a función: dos a `tipo_mas_grande`, cuatro a `sizeof`, una a `operator==(int8, int16)` y una a `operator==(int16, in16)`. Sin embargo tanto `sizeof` como `tipo_mas_grande` pueden evaluarse en tiempo de compilación (toman tags como parámetros y regresan tags) por lo que no realizan ninguna tarea efectiva en tiempo de ejecución; un optimizador

moderno podría eliminar esas llamadas a función además de optimizar el trabajo real realizado dentro de `operator==`.

6.5 Comparaciones con implementaciones de C y C++

Las dos implementaciones a comparar son:

- El compilador gcc desarrollado por el proyecto GNU desde 1987.
- El compilador Clang [93] desarrollado por el proyecto LLVM desde 2005.

Tanto LLVM como Clang son auspiciados por Apple y tienen por objetivo ser un reemplazo de gcc ya que argumentan que éste tiene un alto consumo de recursos, es demasiado lento y su código fuente es demasiado complicado por lo que sólo los programadores con una amplia experiencia modificándolo pueden realizar cambios importantes.

Clang consiste en un conjunto de bibliotecas de alto rendimiento dirigidas a realizar el análisis léxico, sintáctico y semántico de la familia de lenguajes basados en C entre los que se encuentran C++. Clang emite código intermedio para LLVM el cual se encarga de las optimizaciones y de emitir código nativo.

El hecho de que el lenguaje presentado en esta tesis tenga un análisis léxico y sintáctico mucho más simple que C y C++ además de que se haya puesto especial énfasis en el rendimiento del proceso de compilación incluído en el intérprete se ve reflejado en la pruebas realizadas:

Se generó un archivo fuente de 11.1MB equivalente a 1635000 líneas de código en el cual se declararon 408750 funciones diferentes. Se midió el rendimiento del análisis léxico entre Clang 2.0 y el intérprete con los siguientes resultados:

	Tiempo de ejecución	Memoria consumida
Clang 2.0	< 1 segundo	48MB
Intérprete	0.15 segundos	46MB

No existe una manera de solicitar únicamente el análisis léxico de gcc. Tampoco existe una manera de realizar exclusivamente el análisis sintáctico para Clang y gcc puesto que, como ya se mencionó, dicho análisis en C y C++ necesita del análisis semántico; es por esto que la opción `-fsyntax-only`

existente tanto en Clang como en gcc realiza el análisis sintáctico-semántico aunque omite la generación de código. Los resultados son los siguientes:

	Tiempo de ejecución	Memoria consumida
Clang 2.0	~3 segundos	130MB
gcc 4.5.1	~7 segundos	564MB
Intérprete	0.78 segundos	155MB

La generación de código presenta estrategias muy diferentes. Clang 2.0 emite código intermedio para LLVM mientras que GCC tiene su propio conjunto de instrucciones intermedio llamado lenguaje de transferencia entre registros [94]. Desafortunadamente para esta prueba el consumo de memoria en ambos compiladores aumenta considerablemente superando los 2GB de la máquina usada para la realización de ésta. En una prueba reportada en la lista de usuarios de Clang y usando un archivo fuente de 12MB de tamaño, la memoria requerida por Clang es de 8GB mientras que la de GCC es de 2.6GB [95].

A pesar de que el intérprete no usa una representación interna y emite código nativo directamente, la generación de código para un número grande de funciones también representa un problema en éste pues es común que se asignen múltiplos del tamaño de una página (usualmente 4KB) por cada petición de memoria ejecutable. Si bien es posible reutilizar el espacio sobrante de una página solicitada anteriormente, esto aún no se ha implementado. Afortunadamente no constituye un problema serio a corto plazo pues para proyectos pequeños el número de funciones difícilmente igualará el número de funciones del archivo fuente usado como prueba.

7 Conclusiones

Esta tesis presenta el diseño y la implementación de un lenguaje de programación que elimina los problemas léxicos, sintácticos y semánticos de C y C++ y provee un adecuado soporte para la programación genérica, la cual tiene por objetivo lograr la implementación abstracta y eficiente de algoritmos a diferencia de otros paradigmas de programación que consideran el problema de expresar algoritmos como resuelto y que por lo mismo están dirigidos al modelado y diseño de software.

En esta tesis se muestra cómo al intentar cumplir los postulados de la programación genérica se hacen evidentes las carencias de muchos lenguajes de programación en cuanto a su capacidad de expresar algoritmos eficientes en términos generales. También se explica cómo C++ intentó eliminar sin éxito sus debilidades en cuanto a su soporte para este paradigma. Se muestra cómo el lenguaje presentado resuelve todas las debilidades encontradas tanto en C++ como en el resto de los lenguajes analizados. Además de presentar el diseño de algunas características especialmente diseñadas para resolver algunas necesidades de la programación genérica se diseñó: un algoritmo novedoso de resolución de funciones en presencia de espacios de nombres, hasta ahora considerado un problema abierto; un mecanismo de inicialización uniforme para variables y un modelo general para la manipulación de expresiones constantes.

La implementación del lenguaje presentada en esta tesis es un intérprete justo a tiempo que realiza el proceso de compilación eficientemente lo cual es necesario para poder sobrellevar las dificultades causadas por la programación genérica en cuanto a la factibilidad de realizar compilación separada. Además, la implementación permite ejecutar algunos programas mal formados de manera similar a los intérpretes de lenguajes dinámicos. Lo más importante es que la implementación logra demostrar la factibilidad del lenguaje presentado.

8 Trabajo futuro

El trabajo futuro contemplado para este proyecto es el siguiente:

- Publicar los resultados de la investigación realizada durante esta tesis.
- Diseñar e implementar un mecanismo que permita expresar literales de tipos definidos por el usuario.
- Proporcionar un compilador tradicional y un optimizador para el lenguaje.
- Diseñar e implementar una biblioteca de utilidades para el lenguaje.
- Diseñar e implementar un mecanismo de resolución de funciones en tiempo de ejecución de variables con tipos desconocidos.

Con respecto a la publicación de resultados, ya se ha enviado y está en espera de ser aceptado un artículo dirigido a la Mexican International Conference on Computer Sciente (ENC 2011) que, en caso de ser aceptado, será presentado en marzo de 2011 y publicado por la IEEE.

Parte del segundo punto ya se ha hecho pero no se presenta en esta tesis pues no se ha llegado hasta la etapa de implementación. El tercer punto ya se ha comenzado mediante la modificación del generador del código del intérprete desarrollado para emitir código C en lugar de código máquina. De esta forma pueden aprovecharse los compiladores y optimizadores existentes para C.

Con respecto al cuarto punto, consideramos que la construcción de bibliotecas para el lenguaje puede apoyarse en la traducción de código escrito en el lenguaje presentado hacia C o viceversa. Emitir código C permite utilizar los optimizadores de otros compiladores pero complica el uso de código C compilado al no existir compatibilidad binaria con este último. Traducir de C al lenguaje desarrollado elimina el problema de la compatibilidad binaria pero no resuelve el problema de las optimizaciones. Una alternativa híbrida resuelve ambos problemas.

El último punto constituye un área de investigación importante pues el mecanismo de funciones virtuales de C++ tiene limitaciones. Sin embargo alcanzar en tiempo de ejecución la expresividad lograda por las plantillas y las clases requeriría realizar instanciación de plantillas y generar código interactivamente, lo que impactaría el rendimiento del programa. Este es un problema que todavía no ha sido atacado a profundidad principalmente por las carencias en el soporte de la programación genérica en los lenguajes actuales.

Anexo

A continuación se presenta una serie de tablas comparativas que presentan las diferencias más significativas entre C++ y el lenguaje desarrollado en esta tesis.

Características generales	*C++*	*Lenguaje desarrollado*
Dirigido a	Programación de sistemas y aplicaciones	Programación de sistemas y aplicaciones
Paradigmas soportados	Imperativo, procedural, orientado a objetos, programación genérica	Imperativo, procedural, programación genérica
Compatibilidad con	C	Ninguno
Número de palabras reservadas	62 (72 en C++0x)	22
Preprocesador	Sí	No
Análisis léxico independiente de fases subsecuentes y de otros archivos fuente	No	Sí
Análisis sintáctico independiente de fases subsecuentes y de otros archivos fuente	No	Sí
Grámatica LL mínima necesaria para realizar análisis sintáctico	LL(*)	LL(1)
Permite compilación separada	Sí	No (análisis semántico requiere todo el código)

Sistema de tipos	*C++*	*Lenguaje desarrollado*
Fuerza de tipado	Fuerte	Fuerte
Seguridad del tipado	Inseguro	Seguro
Especificación de tipos	Explícita con inferencia opcional	Explícita con inferencia opcional
Agrupación de tipos	Jerarquías de tipos (explícita)	Jerarquías de clases (implícita, propiedades en común)
Compatibilidad de tipos	Nominal	Nominal
Verificación de tipos	Estática	Estática

Otras características	*C++*	*Lenguaje desarrollado*
Arreglos	Sí	Sí
Apuntadores	Sí	Sí
Constantes	Sí	Sí
Tipos compuestos (estructuras)	Sí	Sí
Tuplas	No	Sí
Sobrecarga de funciones	Sí	Sí
Sobrecarga de operadores	Sí	Sí
Funciones con parámetros variables	Sí	Sí
Tipos como parámetros y valores de retorno	No	Sí
Cadenas funcionan en sentencia case	No	Sí
Herencia de tipos	Sí	No
Resolución de funciones en tiempo de ejecución	Sí (funciones virtuales)	No
Acceso a ensamblador	Sí	Sí
Espacios de nombres	Sí	Sí
Plantillas	Sí	Sí
Parámetros de plantilla restringidos	No	Sí (mediante clases)
Orden de inicialización definido para variables globales de múltiples archivos	No	Sí
Elevación de excepciones	Sí	No
Información de tipos en tiempo de ejecución	Sí	No
Recolección de basura	No	No

Referencias

[1] A.D. Birrell, "System Programming in a High Level Language", Tesis de doctorado, Universidad de Cambridge, 1977.

[2] B. Stroustrup, The Design and Evolution of C++, Reading, MA: Addison Wesley, 1994.

[3] J. Gosling y H. McGilton, The Java™ Language Environment - A White Paper, Mountain View, CA: Sun Microsystems, 1996.

[4] I.H. Kazi, B. Stanley, D.J. Lilja, A. Verma, y S. Davis, "Techniques for obtaining high performance in Java programs", ACM Computing Surveys, vol. 32, 2003.

[5] D.R. Musser, G.J. Derge, y A. Saini, "Foreword", STL Tutorial and Reference Guide, Boston, MA: Addison-Wesley, 2001.

[6] J.W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of Zürich ACM-GAMM Conference", International Conference on Information Processing, UNESCO, 1959, págs. 125-132.

[7] B. Ford, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl", ACM SIGPLAN international conference on Functional programming, ACM New York, 2002.

[8] ECMAScript Language Specification, ECMA International, 2009.

[9] "The Python Language Reference", The Python Language Reference, 2010. http://docs.python.org/reference/ [15 dic 2010].

[10] M.E. Lesk y E. Schmidt, "Lex - A Lexical Analyzer Generator". http://dinosaur.compilertools.net/lex/index.html [15 dic 2010].

[11] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I", Communications of the ACM, vol. 3, 1960.

[12] "ANTLR", ANTLR Parser Generator. http://www.antlr.org/ [15 dic 2010].

[13] "Bison - GNU parser generator". http://www.gnu.org/software/bison/ [15 dic 2010].

[14] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler".

[15] F.E. Allen, "A technological review of the FORTRAN I compiler", AFIPS '82 Proceedings, 1982.

[16] D.M. Ritchie, "The Development of the C Language", Cambridge, Mass.: 1993.

[17] D.A. Watt, B.A. Wichmann, y W. Findlay, Ada language and methodology, Reino Unido: Prentice Hall International, 1987.

[18] D.R. Musser y A. Stepanov, "Generic Programming", Roma, Italia: 1988.

[19] T.L. Veldhuizen, "C++ Templates are Turing Complete", 2003.

[20] M. Arnold, S.J. Fink, D. Grove, M. Hind, y P.F. Sweeney, "A Survey of Adaptive Optimization in Virtual Machines", Proceedings of the IEEE, 2005.

[21] M. Anton Ertl y D. Gregg, "The Structure and Performance of Efficient Interpreters", Journal of Instruction-Level Parallelism, vol. 5, 2003, págs. 1-25.

[22] J. Aycock, "A brief history of just-in-time", ACM Computing Surveys, vol. 35, 2003.

[23] M.S. Ager, "V8 Internals", 2009.

[24] Y. Shi, D. Gregg, A. Beatty, y A.M. Ertl, "Virtual Machine Showdown: Stack Versus Registers", Chicago, Illinois: 2005.

[25] G. Garen, "Announcing SquirrelFish", Surfin' Safari, 2008. http://webkit.org/blog/189/announcing-squirrelfish/ [15 dic 2010].

[26] M. Berndl, B. Vitale, M. Zaleski, y A.D. Brown, "Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine

Interpreters", Proceedings of the international symposium on Code generation and optimization, 2005.

[27] M. Stachowiak, "Introducing SquirrelFish Extreme", Surfin' Safari, 2008. http://webkit.org/blog/214/introducing-squirrelfish-extreme/ [15 dic 2010].

[28] J. Runeson y S. Nystr, Generalizing Chaitin's Algorithm: Graph-Coloring Register Allocation for Irregular Architectures, Department of Information Technology, Uppsala University, 2002.

[29] E. Johansson y K.F. Sagonas, "Linear Scan Register Allocation in a High-Performance Erlang Compiler", Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages, 2002.

[30] C.A. Lattner, "Macroscopic Data Structure Analysis and Optimization", Tesis de Tesis de doctorado, Universidad de Illinois, 2005.

[31] P. Pedriana, "EASTL -- Electronic Arts Standard Template Library", 2007. www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html [15 dic 2010].

[32] B. Stroustrup, "C++ Applications", C++ Applications, 2010. http://www2.research.att.com/~bs/applications.html [15 dic 2010].

[33] "Learn About Java Technology", Learn About Java Technology. http://www.java.com/en/about/ [15 dic 2010].

[34] C# Language Reference, Microsoft Corporation, 2007.

[35] American National Standard for Information Systems - Programming Language C, American National Standards Institute, 1989.

[36] "C - Approved standards", ISO/IEC 9899 - Programming languages - C, 2010. http://www.open-std.org/jtc1/sc22/wg14/www/standards.html [15 dic 2010].

[37] Rationale for International Standard - Programming Languages - C, ISO/IEC JTC1/SC22/WG14, 2003.

[38] B. Stroustrup, The C++ Programming Language, Addison Wesley, 1997.

[39] M.A. Ellis y B. Stroustrup, The Annotated C++ Reference Manual, Addison Wesley.

[40] "C++ - Standards", 2010. http://www.open-std.org/jtc1/sc22/wg21/docs /standards [15 dic 2010].

[41] J. Benito, "C - The C1X Charter", 2007.

[42] H. Sutter, "C++0x Timing Options for Kona Discussion", 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2417.pdf [15 dic 2010].

[43] "Minutes of WG21 Meeting No. 41, October 1-6, 2007", 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2452.html [15 dic 2010].

[44] H. Sutter, "C++0x Meeting Schedule", 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3082.pdf [15 dic 2010].

[45] IEEE Standard for Floating-Point Arithmetic, IEEE, 2008.

[46] "Programming languages - C", 2010. http://www.open-std.org/jtc1/sc22 /wg14/www/docs/n1516.pdf [15 dic 2010].

[47] "Working Draft, Standard for Programming Language C++", 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf [15 dic 2010].

[48] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumen 1, Intel Corporation, 2009.

[49] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumen 2A, Intel Corporation, 2009.

[50] M. Lumpe, "Re: LL(1) grammar for dangling else?", comp.compilers, 1995. http://compilers.iecc.com/comparch/article/95-06-031 [15 dic 2010].

[51] "C++ Standard Core Language Defect Reports", C++ Standard Core Language Defect Reports, 2001. www.open-std.org/jtc1/sc22/wg21/docs /cwg_defects.html#106 [15 dic 2010].

[52] A. Jönsson, "Calling conventions on the x86 platform", Calling conventions on the x86 platform, 2005. http://www.angelcode.com/dev /callconv/callconv.html [15 dic 2010].

[53] "C++ ABI Summary", C++ ABI Summary, 2001. http://www.codesourcery.com/public/cxx-abi/ [15 dic 2010].

[54] Technical Report on C++ Performance, ISO/IEC, 2006.

[55] C. de Dinechin, "C++ exception handling for IA-64", Proceedings of the 1st conference on Industrial Experiences with Systems Software, 2000.

[56] H. Sutter y T. Plum, "Why We Can't Afford Export", 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf [15 dic 2010].

[57] "Minutes of PL22.16 Meeting, March 08, 2010", 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3080.pdf [15 dic 2010].

[58] P. Dimov, H.E. Hinnant, y D. Abrahams, "The Forwarding Problem: Arguments", 2002. www.open-std.org/jtc1/sc22/wg21/docs/papers/2002 /n1385.htm [15 dic 2010].

[59] B. Stroustrup y G. Dos Reis, "General Constant Expressions for System Programming Languages", Sierre, Suiza: ACM, 2010.

[60] B. Stroustrup, "Uniform initialization design choices (Revision 2)", 2008. www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2532.pdf [15 dic 2010].

[61] A. Stepanov, "Notes on Programming", 2006.

[62] R. Garcia, J. Jarvi, A. Lumnsdaine, J.G. Siek, y J. Willcock, "A comparative study of language support for generic programming", Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, ACM SIGPLAN Notices, 2003.

[63] G. Dos Reis y J. Jarvi, "What is Generic Programming?", 2005.

[64] A. Church, "Annals of Mathematics", Annals of Mathematics, 1932, págs. 346-366.

[65] A. Stepanov y M.A. Marcus, Notes on the Foundations of Programming, 2005.

[66] "Clay Programming Language", 2010. http://tachyon.in/clay/ [15 dic 2010].

[67] "D Programming Language 2.0", Digital Mars, 2010. http://www.digitalmars.com/d/2.0/index.html [15 dic 2010].

[68] "The Go programming language", The Go programming language, 2010. http://golang.org/ [15 dic 2010].

[69] J. Bernardy, P. Jansson, M. Zalewski, S. Schupp, y A. Priesnitz, "A comparison of c++ concepts and haskell type classes", Proceedings of the ACM SIGPLAN workshop on Generic programming, ACM SIGPLAN, 2008.

[70] P. Wuille y T. Schrijvers, "Breaking the Complexity Barrier of Pure Functional Programs with Impure Data Structures", Inglaterra: 2008.

[71] S. Clamage, "2003 Five-Year Maintenance Review", 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1423.txt [15 dic 2010].

[72] "Minutes of ISO WG21 Meeting, April 6, 2003", 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1458.html [15 dic 2010].

[73] J.G. Siek, "The C++0x Concept Effort", 2010.

[74] J.G. Siek, "A Language for Generic Programming", Tesis de Tesis de doctorado, Universidad de Indiana, 2005.

[75] "Minutes of WG21 Meeting, September 15-20, 2008", 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2784.html [15 dic 2010].

[76] "Minutes of WG21 Meeting, July 13, 2009", 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2920.html [15 dic 2010].

[77] B. Stroustrup, “Concepts Requirements”, 2009.

[78] B. Stroustrup, “Simplifying the use of concepts”, 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf [15 dic 2010].

[79] “Minutes of WG21 Meeting, August 2, 2010”, 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3119.pdf [15 dic 2010].

[80] G. Brandl, “Things that will Not Change in Python 3000”, Python Programming Language – Official Website, 2009. www.python.org/dev /peps/pep-3099/ [15 dic 2010].

[81] The Unicode Standard, Version 6.0.0, Mountain View, CA: The Unicode Consortium, 2011.

[82] Java Language Specification, Sun Microsystems, 2005.

[83] N. Llopis, “The Care and Feeding of Pre-Compiled Headers”, Games from Within, 2005. http://gamesfromwithin.com/the-care-and-feeding-of-pre-compiled-headers [15 dic 2010].

[84] “6.3 Labels as Values”, Using the GNU Compiler Collection (GCC), 2010. http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html [15 dic 2010].

[85] P. Bumbulis y D.D. Cowan, “RE2C: a more versatile scanner generator”, ACM Letters on Programming Languages and Systems, vol. 2, pág. 1993.

[86] E.W. Dijkstra, “ALGOL-60 Translation”, ALGOL Bulletin, vol. 10, 1961.

[87] T. Norvell, “Parsing Expressions by Recursive Descent”, Parsing Expressions by Recursive Descent, 2001.

[88] S. Feferman, “Predicativity”, 2002.

[89] R.M. Smullyan, Diagonalization and Self-Reference, Oxford Science Publications, 1994.

[90] C.A. Lattner, “The LLVM Compiler Infrastructure”, The LLVM Compiler Infrastructure, 2010. http://llvm.org/ [15 dic 2010].

[91] “unladen-swallow - A faster implementation of Python”, 2010. www.code.google.com/p/unladen-swallow/ [15 dic 2010].

[92] “ProjectPlan - Plans for optimizing Python”, unladen-swallow - A faster implementation of Python, 2009. http://code.google.com/p/unladen-swallow/wiki/ProjectPlan [15 dic 2010].

[93] “clang: a C language family frontend for LLVM”, clang: a C language family frontend for LLVM. http://clang.llvm.org/ [15 dic 2010].

[94] “10 RTL Representation”, GNU Compiler Collection (GCC) Internals. http://gcc.gnu.org/onlinedocs/gccint/RTL.html [15 dic 2010].

[95] “Memory and time consumption for larger files”, Clang - An Open Source C/C++ Compiler Front End, 2010. http://www.comments.gmane.org/gmane.comp.compilers.clang.devel/8225 [15 dic 2010].

Glosario

- *Alineación en memoria*: requerimiento del procesador con respecto a las posiciones de memoria en las que un dato puede ser almacenado.
- *Análisis léxico*: separación de los símbolos reconocibles de la entrada.
- *Análisis LL*: análisis sintáctico descendente y de izquierda a derecha.
- *Análisis LR*: análisis sintáctico ascendente y de izquierda a derecha.
- *Análisis semántico*: determinación del significado de una cadena.
- *Análisis sintáctico*: validación estructural de los símbolos de una cadena.
- *Análisis sintáctico ascendente*: análisis que parte de la forma general de la entrada hacia una estructura sintáctica particular.
- *Análisis sintáctico descendente*: análisis que parte de la estructura sintáctica particular para inferir su forma general.
- *Apuntador*: número que representa una dirección de memoria.
- *Árbol abstracto de sintaxis*: árbol que captura la estructura simplificada de una expresión donde el orden de evaluación y agrupamiento están implícitos.
- *Arreglo*: colección homogénea de elementos accedidos mediante un índice.
- *Asociatividad de un operador*: regla que se aplica para desambiguar el orden de evaluación para operadores con la misma precedencia.
- *Autómata de pila*: autómata que puede realizar análisis sintáctico mediante transiciones simples basadas en los símbolos de la entrada y en una pila auxiliar de memoria.
- *Autómata finito*: autómata que puede reconocer lenguajes mediante transiciones simples basadas en los símbolos de la entrada.
- *Biblioteca*: conjunto de algoritmos, tipos y otras utilidades de software.
- *Búsqueda dependiente*: algoritmo de resolución de C++ que inyecta un conjunto de funciones no encontradas en una llamada a función ordinaria.

- *C*: lenguaje de programación de sistemas diseñado por Dennis Ritchie en 1971.
- *C++*: lenguaje de programación de sistemas diseñado por Bjarne Stroustrup en 1982.
- *Clase*: conjunto de funciones que debe poseer un tipo para ser considerado miembro de dicha clase.
- *Compilador*: programa que realiza la transformación del código de un lenguaje de alto nivel a ensamblador.
- *Concepto*: característica de C++0x que permite describir conjuntos de restricciones que los tipos deben cumplir.
- *Constante*: valor que no puede ser modificado durante la ejecución de un programa.
- *Constructor*: función invocada automáticamente para inicializar una variable.
- *Conversión de dato*: información interpretada usando un tipo distinto al original.
- *Espacio de nombres*: contenedor de identificadores que establece un prefijo para los mismos al ser referidos fuera de éste.
- *Estándar de un lenguaje*: especificación de un lenguaje reconocida por una organización nacional o internacional.
- *Estructura*: colección heterogénea de elementos accedidos mediante nombres diferentes.
- *Estructura algebraica*: conjunto de propiedades mediante el cual se puede decidir si otros conjuntos cumplen o no éstas.
- *Excepción*: mensaje de error elevado por alguna parte de un programa que puede ser capturado por otra.
- *Expresión constante*: expresión con un valor que puede ser conocido durante el proceso de compilación.
- *Función*: agrupamiento de instrucciones que realizan una acción relacionada y que puede ser utilizado por diferentes partes de un programa.

- *Función virtual*: función que es resuelta en tiempo de ejecución usando el tipo real del parámetro principal de la misma.
- *Generación de código*: etapa en la que se produce el código destino de un proceso de compilación.
- *Gramática libre de contexto*: descripción simbólica de un lenguaje reconocible mediante un autómata de pila.
- *Herencia*: mecanismo mediante el cual un tipo adquiere características de otro tipo relacionado jerárquicamente.
- *Instanciación*: proceso de generar una definición completa mediante una plantilla y un conjunto de valores para los parámetros de la misma.
- *Intérprete*: programa que determina el significado del código en un lenguaje y lo ejecuta sin producir un programa ejecutable utilizable posteriormente.
- *Lenguaje ensamblador*: lenguaje de bajo nivel con una traducción directa al lenguaje máquina del procesador.
- *Lenguaje de alto nivel*: lenguaje que ofrece abstracciones que no siempre tienen una traducción directa a lenguaje ensamblador.
- *Lenguaje de bajo nivel*: lenguaje que tiene una traducción directa o casi directa a lenguaje ensamblador.
- *Lenguaje regular*: lenguajes reconocidos por un autómata finito.
- *Optimización de código*: proceso de reemplazar código por otro que genera los mismos resultados pero de manera más eficiente.
- *Palabra del procesador*: agrupamiento de un número fijo de bits que es el agrupamiento natural para alguna arquitectura de procesadores dada.
- *Plantilla*: porción parametrizada de código que puede ser reutilizado para generar versiones distintas para valores distintos de sus parámetros.
- *Precedencia de operador*: regla que determina el orden de evaluación de las operaciones de una expresión.
- *Programación genérica*: disciplina que se encarga de la búsqueda y organización de componentes de software genéricos y eficientes.

- *Programación orientada a objetos*: paradigma que se centra en la descripción de tipos, su organización jerárquica y en la abstracción de los detalles de implementación.
- *Punto flotante*: formato en el cual la representación de un número puede variar en base lo que se encuentra antes y después del punto decimal de dicho número.
- *Referencia*: nombre o identificador alternativo para otra variable.
- *Sobrecarga y resolución de funciones*: elección de la función más adecuada sobre un conjunto de funciones que tienen el mismo nombre pero parámetros diferentes.
- *Soporte en tiempo de ejecución*: capacidad de examinar el estado del programa durante ejecución la cual es necesaria para implementar algunas características de un lenguaje o biblioteca.
- *Tag*: tipo que representa el valor de una expresión constante.
- *Tipo dependiente*: tipo que participa en la descripción de una estructura algebraica.
- *Tupla*: colección heterogénea de elementos accedidos mediante un índice.
- *Valor-d*: valor que sólo puede aparecer del lado derecho de una asignación.
- *Valor-i*: valor que representa una localidad que a la que se le puede ser asignar un valor.