CENTRO DE INVESTIGACION Y ESTUDIOS AVANZADOS DEL

INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRIA

SECCION DE COMPUTACION

" PRECOMPILADOR DE C++ A C "

T E S I S

QUE PRESENTA EL INGENIERO

LUIS RUBEN RUSILES ZAMORA

PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS EN
LA ESPECIALIDAD DE
INGENIERIA ELECTRICA
CON OPCION DE COMPUTACION

TRABAJO DIRIGIDO POR:

DR. JOSEF KOLAR SABOR

BECARIO DE COSNET

MEXICO, D.F. A 3 DE ABRIL DE 1989

dedicada

a mis padres:

Angel y Martha

AGRADECIMIENTOS.

la gratitude est la memoire du coer.

Quiero externar aqui ademas de mi agradecimiento, mi admiración por el Dr. JOSEF KOLAR.

Quiero agradecer al Dr. JAN JANECEK y al M. en C. OSCAR OLMEDO por aceptar ser sinodales en mi examen.

A la Sección de Computación, al Depto. de Ingenieria Electrica y al Centro de Investigación y Estudios Avanzados, les estoy profundamente agradecido, por haberme dado la oportunidad de ser parte integrante en ellos.

Al Consejo del Sistema Nacional de Educación Tecnológica (COSNET) le doy las gracias,por apoyarme económicamente para

PRECONFILADOR DE C++ A C

I N D I C E :

FASES:

APENDICES:

P R E F A C I O .

En el presente escrito se describen las fases seguidas en el desarrollo del *Precompilador de C++ a C*.

C++ es un lenguaje de proposito general para programación orientada a objetos. Es un superconjunto del lenguaje C y compatible con él. Maneja clases, sobrecarga de funciones y sobrecarga de operadores. Su diseño se originó alrededor de 1980 por Bjarne Stroustrup en AT&T Bell Labs, Murray Hill, New Jersey.

La programación orientada a objetos es recomendable en grandes proyectos y en la creación de "Circuitos Integrados de Software". Esta actitud cambia el enfoque de la programación estructurada: pues en lugar de preguntar: ¿Cuáles son mis entradas y cuáles mis salidas? y ¿Como se transforman unas en otras?; pregunta: ¿Cuáles son mis objetos? y ¿Qué quiero que esos objetos hagan?.

La meta inicial del *Precompilador de C++ a C*, fue la de ser un Sistema Programado que, dado un programa en C++ , produjera el el código objeto correspondiente. Que tuviera dos pasos internos: traducir de C++ a C y compilar C. Y que usara 2 de las herramientas del Sistema Operativo UNIX: YACC y LEX.

Sin embargo, debido a la gran cantidad de trabajo que esto representa, en esta tesis solo se incluyen: La descripción global

del *Precompilador de C++ a C* y el desarrollo de algunas de las rutinas más importantes.

Adicionalmente en el presente trabajo se sugiere, y ejemplifica en forma intrínsica, una metodología a seguir en proyectos de investigación. Esta metodología consta de 5 fases: ESTUDIO, DISEÑO, DESARROLLO, PRUEBA E IMPLEMENTACIÓN. En la fase de DISEÑO se hace incapié en el uso de estándares de programación para dar consistencia y mantenibilidad al Sistema en su etapa final.

I. FASE DE ESTUDIO.

I.1 INTRODUCCION.

En esta fase se delimitó el problema especificando los objetivos y alcances del proyecto. Se determinaron las necesidades del usuario que serían satisfechas, así como de las posibles maneras de hacerlo. Se eligió una de esas formas y se justificó su elección.

I.2 OBJETIVOS.

I.2.1 Proveer de una herramienta para compilar C++.

I.2.2 Bosquejar los requerimientos del usuario.

I.2.3 Hablar de las posibles soluciones, elegir una, y justificarla.

I.3 PRODUCTOS ENTREGADOS.

Una sección para cada uno de los objetivos anteriormente planteados.

I.3.1 OBJETIVOS Y ALCANCE DEL PROYECTO.

El objetivo principal del presente proyecto fue el de proveer una herramienta para compilar C++.

I.3.2 REQUERIMIENTOS DEL USUARIO.

El usuario necesita un Compilador de C++, con el fin de investigar las posibilidades que proporciona la programación orientada a objetos.

I.3.3 SOLUCIONES POTENCIALES, SOLUCION ELECTA Y JUSTIFICAION.

Algunas de las soluciones que pueden satisfacer los anteriores requerimientos son:

a) La adquisición de un compilador comercial de C++.

b) El desarrollo completo de un compilador de C++.

c) Alguna solución intermedia.

Al comprar un compilador comercial, no se dispone de su código fuente. Es prácticamente incosteable realizar cambios en sus estatutos. Además, un compilador comercial, no es transportable de una computadora a otra.

El desarrollo total de un Compilador que sea de buena calidad, no es algo sencillo. Requiere de un todo un equipo de gentes, lo cual es muy costoso.

La solución que se eligió está relacionada con el punto 3 anterior. Consiste en compilar el lenguaje C++, usando un código intermedio. Este código es el lenguaje C. De aquí surgió el *Precompilador de C++ a C*.

El *Precompilador de C++ a C* fue la herramienta que se eligió como producto final del proyecto. El *Precompilador de C++ a C* es un Sistema que, dado un programa en C++, produce su código objeto. Internamente tiene dos pasos: la traducción de C++ a C y la compilación de C.

Las causa por la que se eligió C como código intermedio, fue la compatibilidad que estos lenguajes tienen. A continuación se mencionan antecedentes de ellos.

I.3.1.1 LENGUAJE DE PROGRAMACION C.

C es un lenguaje de proposito general diseñado e implementado alrrededor de 1972 por Dennis Ritchie en los laboratorios Bell. Su crecimiento está asociado al Sistema Operativo UNIX donde fue desarrollado, ya que tanto este Sistema Operativo como la mayoría de los programas que corren en él están escritos en C.

El lenguaje C fue originalmente diseñado para programación de Sistemas, esto es, compiladores, sistemas operativos y editores de texto. Pero ha probado ser muy satisfactorio en otras aplicaciones, como bases de datos, sistemas telefónicos, programas de ingeniería, etc. Hoy, C es uno de los lenguajes mas usados y existen versiones para casi cualquier computadora.

C tiene su origen en BCPL, diseñado por Martin Richards alrrededor de 1967. BCPL solo tiene un tipo de datos, la palabra de máquina. En 1970, Ken Thompson diseño una versión modificada de BCPL para ser usada con el primer sistema UNIX en una PDP-7, este lenguaje fue llamado B y también tenía solo un tipo de datos. El lenguaje C fue originalmente un intento de trabajar con mas tipos de datos, agregando esta noción al lenguaje B.

C posee pocos operadores para manejar objetos complicados como

una unidad (arreglos, estructuras). No tiene operaciones de entrada y salida como parte del lenguaje. No tiene manejo de memoria, como la función NEW de Pascal, no tiene facilidades para programación concurrente, como el mecanismo rendez-vous de Ada. De cualquier manera el grado en el que son omitidas algunas instrucciones de C es una de sus características distintivas.

I.3.3.2 LENGUAJE DE PROGRAMACION C++.

C++ es un lenguaje de proposito general que, excepto por pequeños detalles, es un superconjunto de C. Su diseño se originó alrededor de 1980 por Bjarne Stroustrup y las principales influencias aparte de C son Simula67 y Algol68. Adiciona a las caracteristicas de C, el soportar tipos abstractos de datos.

C++ fue instalado inicialmente en 1983. Hoy, hay varios miles de instalaciones que lo usan. Ha sido usado en grandes proyectos universitarios de investigación y para desarrollo de software de gran escala en compañias como Apple, Apolo, AT&T y Sun. Ha sido aplicado en varias ramas de la programación, incluyendo la banca, CAD, construcción de compiladores, manejo de bases de datos, procesamiento de imágenes, graficación, sintesis de musica, redes, ambientes de programación, robótica, simulación, computación científica, conmutación y VLSI.

C++ preserva la fortaleza de C (flexibilidad, eficiencia, disponibilidad y portabilidad), y remedia algunos de sus mas obvios problemas. Por ejemplo:

CHEQUEA EL TIPO DE LOS ARGUMENTOS DE UNA FUNCION.

```
extern double sqrt(double);
/* declara la función raiz cuadrada */

double d1=sqrt(2);
/* bien, 2 es convertido a double */

double d2=sqrt("dos");
/* error de compilación, sqrt no acepta un string */
```

PROVEE TIPOS ABSTRACTOS DE DATOS.

Esta es una técnica de programación en la cual se definen tipos de propósito general y tipos de propósito especial como base para aplicaciones. Estos tipos definidos por el usuario son convenientes para programadores de aplicaciones ya que permiten referencia local y ocultamiento de datos.

```
class complex {
   double re,im;
public:
   complex (double r, double i)
   {
       re=r; im=i;
   }

   complex (double r); /* conv. real -> complex */
   {
```

```
re=r;
i=0;
```

```
    friend complex operator + (complex,complex);
    friend complex operator - (complex,complex);
    friend complex operator * (complex,complex);
    friend complex operator / (complex,complex);
    friend complex operator - (complex);  /* - unario */
};
```

La primera parte especifica la representacion de un complejo y es por omisión privada. Esta representación, dos numeros *double*, es accesible solo por funciones definidas dentro de la declaración de complex.

La segunda parte especifica como un usuario puede crear y manejar números complejos. Esta parte es pública y es la interfaz para el público en general. Consiste en 2 constructores, funciones con el mismo nombre que la clase), y la aritmética normal. Un constructor es una función que construye un valor de un tipo dado. El primer constructor construye un número complejo dadas sus dos coordenadas, y el segundo constructor lo hace dada solo la parte real.

Las funciones aritméticas son definidas como *friend*. Estas funciones son completamente ordinarias, excepto que tienen permitido el acceso a la representación de los números complejos. Esta representación sería de otra manera inaccesible. Estas funciones pueden ser definidas de la siguiente manera

```
complex operator + (complex a1, complex a2)
```

```
return complex(a1.re+a2.re, a1.im+a2.im);
```

y usadas así

```
main()
{
   complex a=2.3;
   complex b=(1/a,7);
   complex c=a+b+complex(1,4.5);
}
```

El ocultamiento de datos es la llave para la modularidad. Programación con clases mueve el énfasis del diseño de algoritmos al diseño de clases (tipos definidos por el usuario). Cualquier objeto en un programa es de alguna clase que define el conjunto de operaciones legales para ese objeto. Esto permite al usuario programar en un lenguaje con un conjunto de tipos o conceptos apropiados a la aplicación. Un ingeniero podría usar números complejos, matrices; mientras un diseñador de software para graficación preferiría tipos como línea, polígono, circulo, etc.

PERMITE PROGRAMACION ORIENTADA A OBJETOS.

C++ permite especificar clases organizadas jerárquicamente, esta es la característica principal para soportar programación orientada a objetos. La organización jerárquica es muy importante para tratar con topicos complejos en muchos campos de la ciencia, y ha demostrado ser también una buena manera de organizar

programas para una gran variedad de áreas de aplicación.

En C++, la habilidad para extender un programa adicionando nuevas variaciones de un concepto básico (esto es, nuevas clases derivadas dada una clase base) sin tocar el viejo código, es una de las principales ventajas. Cuando se usan técnicas tradicionales, estas adiciones requieren acceso al código fuente del sistema que se quiere extender, requiere el entendimiento de los detalles de la implementación y lleva el riesgo de introducir errores en el código ya probado.

ADA provee facilidades para tipos abstractos de datos, pero no tiene un mecanismo de herencia para soportar programación orientada a objetos, así C++ tiene mayor potencia expresiva en esta área.

Smalltalk soporta programación orientada a objetos, pero C++ se distingue de él por una variedad de factores:

- El énfasis en la estructura del programa.
- La flexibilidad de los mecanismos de encapsulado.
- La portabilidad.
- La eficiencia en la corrida.
- La habilidad para correr en un sistema pequeño.

PERMITE LA SOBRECARGA DE FUNCIONES Y OPERADORES.

Sobrecarga de funciones.

Se puede usar el mismo nombre para funciones que ejecutan una acción análoga a diferente tipo de argumentos. Por ejemplo, si se definen:

```
rota(circulo);
rota(poligono);
rota(shape);
```

entonces

```
rota(x);
```

escoge la instancia de la función de acuerdo a la clase a la que pertenezca el objeto x.

Sobrecarga de operadores.

Permite cambiar la semántica de la notación convencional. Esta notación usa operadores pequeños, y por tanto es compacta. Una expresión puede tener diferentes significados ... .

```
s = c+p;
```

puede significar:

a). Sumar c y p, y almacenar el resultado en c.

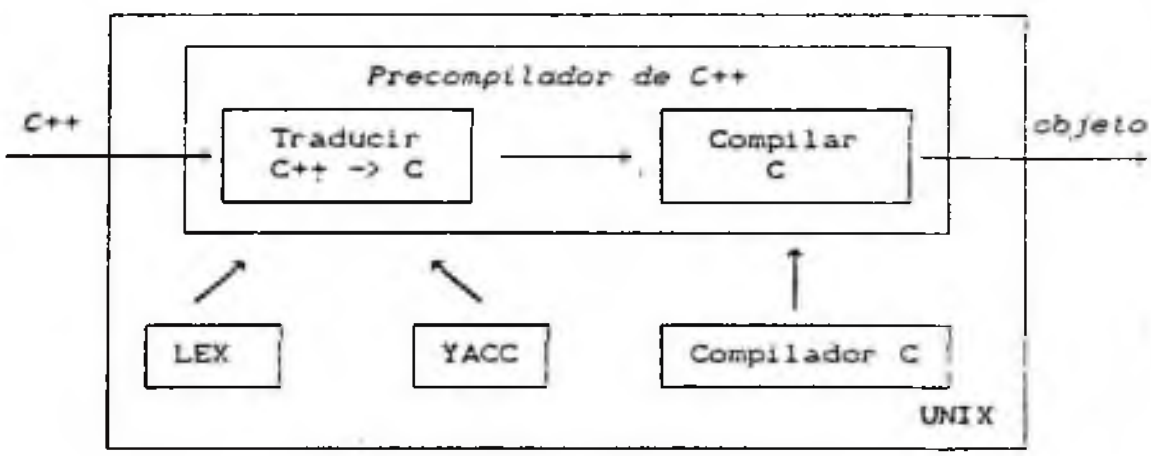
b). El conjunto s, es la unión del conjunto c con el conjunto p.

c). Formar la figura s, mediante la unión de las figuras c y p, dependiendo del tipo de los operandos.

I.3.3.3 PRECOMPILADOR DE C++ A C.

Esta herramienta tendría las características de C++ y por tanto de C, además sería una herrmienta *abierta*. Lo anterior significa que se dispondría de la información necesaria para efectuar los cambios sintácticos o semánticos de C++ que posteriormente se consideren pertinentes. Por ejemplo, se podría agregar la palabra reservada *private* en la definición de clases; o permitir que una clase se derive de varias (herencia multiple).

El *Precompilador de C++ a C* fue desarrollado usando 2 de las herramientas del Sistema Operativo UNIX: LEX y YACC. LEX es una utilería para generar analizadores léxicos. YACC permite generar *parsers*, es decir, analizadores sintácticos.



II. F A S E D E D I S E Ñ O .

II.1 INTRODUCCION.

Aquí se verificaron las decisiones tomadas en la Fase de Estudio. Se hizo el diseño detallado del Sistema. El Diseño fue dividido en 2 tipos: externo e interno. En el Diseño Externo se describen las facilidades con que el usuario contará; y en el Diseño Interno, la manera en la que estas facilidades serán implementadas.

II.2 OBJETIVOS.

Verificar las decisiones de la Fase de Estudio.

Complementar, detallar y documentar el Diseño.

II.3 PRODUCTOS ENTREGADOS.

II.3.1 DISEÑO EXTERNO.

II.3.1.1 BOSQUEJO DE LA ARQUITECTURA EXTERNA DEL SISTEMA.

II.3.1.2 FACILIDADES PARA EL USUARIO.

II.3.2 DISEÑO INTERNO:

II.3.2.1 BOSQUEJO DE LA ARQUITECTURA INTERNA DEL SISTEMA.

II.3.2.2 DESCRIPCION DE LOS PROCESOS.

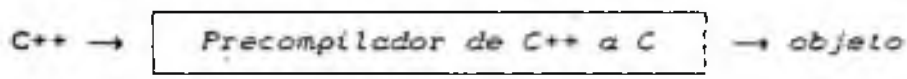
II.3.2.3 DESCRIPCION DE LAS ESTRUCTURAS DE DATOS Y FUNCIONES.

II.3.2.4 DESCRIPCION DE LOS ARCHIVOS Y FUNCIONES.

II.3.1 DISEÑO EXTERNO.

II.3.1.1 BOSQUEJO DE LA ARQUITECTURA EXTERNA DEL SISTEMA.

Externamente el Precompilador de C++ se vería como



el usuario teclearía su programa en C++, usando cualquier editor: de línea, *ed*; ó de bloque, *vi*. Este texto se compilaría llamando al *Precompilador de C++ a C*. La sintáxis de la llamada sería igual a la de la llamada al Compilador de C

```
$ ccc myprog myobj
```

los errores encontrados en el tiempo de compilación serían mostrados en la salida estandard de error (stderr).

Cuando la compilación fuera exitosa, en ausencia total de errores, se produciría el codigo objeto correspondiente. Este código se podría ejecutar como cualquier programa C compilado.

```
$ myobj
```

II.3.1.2 FACILIDADES PARA EL USUARIO.

Las instrucciones de que dispondría el usuario del *Precompilador de C++ a C* para escribir sus programas serían, básicamente, las del lenguaje C++ diseñado por Stroustrup, salvo algunas pequeñas diferencias. Existen diferencias notables entre C++ y C.

a) DIFERENCIAS DE C++ Y C.

Nuevas Palabras Reservadas.

Las palabras reservadas son aquellas que no pueden ser usadas como identificadores.

En C las palabras reservadas son:

auto	char	struct	if	break
static	float	union	else	continue
extern	double	enum	while	return
register	int		do	goto
typedef	short		for	case
	long		switch	default
sizeof	unsigned			

En C++ se incluyen como palabras reservadas las de C y las siguientes:

const	public	class	this
	friend		
new	inline		
delete	overload		
virtual			
operator			

Declaración de Función sin Argumentos.

La declaración de una función f

```
f();
```

tiene diferente significado en C que en C++. En C, f() puede tomar cualquier número y tipo de argumentos. En C++f() no tiene argumentos.

Identificador 'extern'.

En C, un identificador '*extern*' puede ser definido varias veces. En C++, se debe definir solo una vez.

Mismos nombres de Identificadores.

En C++ un identificador simple y una estructura.pueden tener el mismo nombre. En C++, esto no se permite.

b) CARACTERISTICAS DE C++

Comentarios tipo Algol.

Estos comentarios comienzan con //, y terminan al final de la línea.

Declaración y Uso de Funciones.

Sobrecarga de nombres de función.

Se puede dar el mismo nombre a funciones con diferentes parámetros. Al llamar una función, se escoge la instancia de acuerdo al número y/o tipo de parámetros.

Sustitución Inline.

Una función puede ser declarada *inline*. Al llamar a una función *inline*, se sustituye su código por la llamada.

Sobrecarga de operadores.

Los operadores (+, *, <<, &, etc.) pueden tener varios significados. El significado de un operador depende del número y tipo de los operandos. Los operadores paréntesis y corchetes también pueden ser sobrecargados.

Identificadores y Tipos.

Identificadores con valor constante.

Un identificador puede ser declarado constante (const). Una función puede tener parámetros que sean identificadores constantes. El valor de un identificador constante NO puede ser modificado.

Identificadores de tipo referencia.

Un identificador puede declararse de tipo referencia. Los identificadores de tipo referencia pueden ser usados en la llamada a funciones para el paso de argumentos por referencia. El tipo referencia hace posible tener varios nombres para una misma localidad de memoria.

*Identificadores de tipo void **.

Un identificador puede ser declarado de tipo *void **. Cualquier apuntador puede ser asignado al tipo *void ** sin el uso de 'cast' (conversión explicita de un tipo a otro).

Declaración con valor inicial y usando 'cast'.

Un identificador puede recibir un valor inicial al ser declarado. Si el valor inicial es de otro tipo, se puede hacer la conversión explícitamente a través de un 'cast'.

Uso de Clases.

Tipo Clase.

Un tipo puede ser declarado como *class* (clase). Una clase tiene 2 partes: privada y pública. Datos y funciones pueden ser declarados tanto en la parte privada como en la parte pública de una clase, y son llamados miembros. Los miembros de la parte privada son accesibles solo para las funciones declaradas dentro de la clase (miembros función).

Una clase es un tipo. Un objeto de clase, es un objeto de ese tipo.

Miembros Datos.

static

Un miembro dato puede ser declarado estático (*static*) dentro de una clase. Un miembro *static* es común a todos los elementos de una clase.

Miembros Funciones.

Las funciones miembros también permiten las características de

"Sobrecarga de Nombres", "Sustitucion en Linea" y "Sobrecarga de Operadores", mencionadas anteriormente.

Constructores.

Un constructor en una función que crea un objeto de una cierta clase. Son constructores los miembros funciones que tienen el nombre de la clase. Si se declara un objeto de una clase dada, en el tiempo de corrida se hacen 2 cosas: se le asigna la memoria necesaria al objeto, y se le aplica la función constructor de su clase (si esta función existe).

Una clase puede tener cero o más constructores. Los nombres de los constructores también pueden ser sobrecargados (ver 2.2.1.).

Destructores.

Un destructor es una función que destruye un objeto de una clase. Son destructores los miembros funciones que tienen el mismo nombre de la clase pero precedido por '~'. Si se declara un objeto de una clase en un cierto bloque, en el tiempo de corrida ocurre lo siguiente: al salir el flujo del programa del bloque, se le aplica al objeto el destructor de la clase (si este destructor existe).

Una clase tiene solamente un destructor.

Amigo ('friend').

Un miembro función puede ser declarada *friend* de una o varias clases. Una función *friend* de una cierta clase, tiene acceso a la parte privada de los objetos de esa clase.

Convertidores de tipo.

Un miembro función puede ser convertidor de tipo. Un convertidor de tipo es una función que convierte un objeto de clase a un tipo simple (int, float, etc.). Un convertidor de tipo se declara como un operador sobrecargado, donde el operador es precisamente el tipo simple.

Autoreferencia en funciones: 'this'.

Un objeto de una clase tiene memoria asignada. En esta memoria se encuentran los miembros datos particulares del objeto. Existe un apuntador a esa memoria identificado con la palabra reservada *this*. Este apuntador puede ser usado por cualquier función que

tenga acceso al objeto.

Tipos: 'class', 'struct', 'union' y 'enum'.

El nombre de una clase (*class*, *struct*, *union*) o el de una enumeración (*enum*) son tipos. Estos tipos pueden ser usados como cualquier *typedef*.

struct es un caso particular de clase. En la clase *struct* todos los miembros, tanto datos como funciones, son públicos.

union es un caso especial de clase. En la clase *union* solo hay miembros funciones que son constructores.

Una *union* puede ser anónima (sin nombre). Los miembros de una unión anónima no se califican cuando son usados.

Clases Derivadas.

Descripcion.

Se puede derivar una clase (llamada derivada) de otra (llamada base). Los miembros privados de la clase base NO son accesibles para la clase derivada. Los miembros públicos de la clase base son privados de la clase derivada, a menos que se especifique otra cosa.

Las clases derivadas tienen las características mencionadas para las clases (ver 2.3.5.), con las siguientes consideraciones:

Constructores

El orden de construcción para un objeto de un clase derivada es el siguiente: aplicar el constructor de la clase base y, enseguida, aplicar el constructor de la clase derivada. Este orden se aplica recursivamente a los objetos de clase derivada de clase derivada.

Destructores.

El orden de destrucción para un objeto de una clase derivada es

el siguiente: aplicar el destructor de la clase derivada y,enseguida, aplicar el destructor de la clase base. Es decir, un orden contrario al de construcción. Este orden de destrucción se aplica también recursivamente a los objetos de clase derivada de clase derivada.

Funciones Virtuales.

Un miembro función de una clase puede ser declarado virtual. Una función virtual es declarada en la clase base y, posteriormente definida en la clase derivada.

Expresiones.

new y delete.

Hay 2 operadores incluidos en $C++^{P}$ para el manejo de memoria dinámica: *new y delete*. *new* pide la memoria para un cierto tipo de dato, y *delete* la devuelve.

Acceso a Identificadores Globales.

Se puede tener el mismo identificador para una variable global y una local. Dentro del entorno de la variable local, el uso del identificador simple hace referencia a la variable local. Se debe añadir el prefijo '::' al identificador, para hacer referencia a la variable global.

II.3.2 BOSQUEJO DE LA ARQUITECTURA INTERNA DEL SISTEMA.

II.3.2.1 INTRODUCCION.

Algunas de las herramientas con las que disponemos para la descripción del proceso de datos, son las siguientes.

a) Diseño de Flujo de Datos. Orientado a Entradas y Salidas. ("HIPO", Hierarchical Input-Process-Output)
b) Diseño de Flujo de Datos. Orientado a Transacciones.
c) Diseño Orientado a Objetos.
d) Diseño Orientado a Estructuras de Datos.
e) Diseño de Sistemas en Tiempo Real.
f) Diseño Orientado por Gramática.

Es difícil los procesos de datos de un compilador de un lenguaje como C o C++. Esto es debido a dos cosas: la inexistencia de entes suficientemente independientes, por una parte; y la naturaleza recursiva del lenguaje por otra.

Por ejemplo, si consideramos "definición" y "estatuto" como dos entes, estos no son independientes, pues una "definición" de una función, tiene estatutos en su interior.

Por otra parte, en C++, una función puede tener una declaración que contenga una función que contenga una declaración, y así

recursivamente.

El Diseño Orientado por Gramática es el más adecuado para un compilador; pero es también el menos entendible en un nivel superior de abstracción.

En el *Precompilador de C++ a C* se usan las siguientes 3 herramientas:

a) Diseño por Flujo de Datos (nivel superior de abstracción).
b) Diseño Orientado a Estructuras de Datos (otros niveles)
c) Diseño Orientado por Gramática (otros niveles).

Es importante notar que en la creación de un Compilador puede ser usado el Diseño Orientado a Objetos, siendo estos objetos de tipo recursivo.

II.3.2.2 DIAGRAMA DE FLUJO DE DATOS.

En este Diagrama la naturaleza recursiva del lenguaje y la no independencia de entes, se manejan a través de la TABLA de SIMBOLOS (TBL_SYM), y del STACK SEMANTICO (STK_SEM).

II.3.2.3 DESCRIPCION DEL DIAGRAMA DE FLUJO DE DATOS.

PROCESOS del Precompilador de C++ a C :

a) Análisis Léxico.

b) Análisis Sintáctico y Semántico.

c) Emisión de Código.

d) Compilación de Código.

ESTRUCTURAS DE DATOS del Precompilador de C++ a C.

a) Tabla de Símbolos. (TBL_SYM)

b) Tabla de Etiquetas. (TBL_LBL)

c) Arbol de Destrucción. (TREE_DEST)

d) 'Stack' Semántico. (STK_SEM)

e) 'Stack' para Inicialización. (STK_INIT)

ARCHIVOS del Precompilador de C++ a C .

a) Archivo de Código Ligado. (FI_COD)

b) Archivo de Código en C. (FI_COD_C)

PROCESOS del Precompilador de C++ a C ..

a) Análisis Léxico.

Consta del Proceso: Analizar Léxico. A este proceso entran caracteres y salen "token"s (entidades sintácticas). Un caracter es '*', 'a','(', etc. Un token es 'while', 'int', '%', etc. Los "token"s pueden ser: palabras reservadas, identificadores, constantes, operadores o caracteres.

b) Análisis Sintáctico y Semántico.

Lo forman los siguientes procesos: Analizar Sintáxis, Chequear Declaración, Chequear Estatuto, Chequear Expresión.

Los "token"s del Análisis Léxico son agrupados en entidades sintácticas. Estas entidades pueden ser: declaraciones, estatutos o expresiones.

Las entidades sintácticas son chequeadas semánticamente para determinar errores tales como: definición doble del mismo identificador, error de sintáxis en un estatuto, expresion inválida (suma de 2 estructuras), etc.

c) Emisión de Código.

Una entidad sintáctica, al pasar por el análisis semántico, y después de él, produce de acuerdo a su calidad ...

Mensajes de Error.

Código C. (Que simula al de C++).

d) Compilación de Código.

El código producido por el *Precompilador de C++ a C*, se pasa a través del Compilador C, y se obtiene código objeto.

ESTRUCTURAS DE DATOS Del Precompilador de C++ a C .

Introduccion.

A continuación se describen las estructuras usadas en el *Precompilador de C++ a C*. Las funciones que accesan estas estructuras las podemos clasificar en:

Funciones Básicas.

Funciones Conceptuales.

Las *Funciones Básicas* realizan acciones directas, tales como pedir y dar valor a los campos de las estructuras. El hecho de escribir estas rutinas "triviales" trae consigo ciertas ventajas y desventajas.

Ventajas del uso de Funciones Básicas.

Permiten a las otras rutinas llamantes, abstraerse de la implementación real de la estructura.

Facilitan la programación de rutinas superiores.

Realizan chequeos sobre errores de programacion ("bugs").

Eliminan el uso tedioso de apuntadores.

Desventajas del uso de Funciones Básicas.

Tiempo. El código resultante es más lento.

Trabajo. Hay que invertir trabajo en ellas, pero se recupera en la programación de nivel superior.

Las *Funciones Conceptuales* agregan información. Verifican que la nueva información sea correcta por si misma, y que sea correcta con relación a la información que ha llegado previamente.

ESTRUCTURAS DE DATOS del Precompilador de C++ a C .

a) Tabla de Símbolos.

Introducción.

La tabla de Símbolos es la parte medular del *Precompilador de C++ a C*. En ella se registra la información necesaria para manejar clases, sobrecarga de funciones y sobrecarga de operadores.

En C++ un Símbolo define un ambiente de cierto alcance. Ahí los alcances léxico y semántico están separados.

El alcance léxico es el que usa la liga sintáctica (atributos) para determinar si un Símbolo "pertenece" a otro. Esta liga usa los campos de atributos (atributos, 'Attr'; atributos públicos, 'PubAttr'; atributos privados, 'PrivAttr'). En el ejemplo

```
/* nivel global */

struct r_A {
   int i;
} r1;

struct r_B {
   struct r_C {
      int i;
   } r2;
} r3;

main() {
{
   r1.i;
   r3.r2.i;
```

```
}
```

se usa la liga semántica para encontrar 'r1'. Una vez encontrado, se usa la liga sintáctica para encontrar 'i'. Similarmente sucede con 'r3.r2.i'.

El alcance semántico es el que usa la liga semántica (ancestro) para determinar si un Símbolo "se ve" desde el ambiente determinado por otro. Esta liga usa el campos de ancestro ('Parent'). En el siguiente ejemplo

```
/* global */
int i;

class c_A {
   int i;
public:
   int print();
};

c_A :: f()
{
   printf("%d", i);
}

main()
{
   c_A c1;
   c1.print();
}
```

se imprime el valor de la 'i' de c1. Lo anterior sucede porque,

cuando en f() se busca 'i', el ancestro de f() no es 'global', sino la clase c_A.

Existe una tercera forma de accesar un Símbolo. Es el permiso de acceso amigo ('friend'). Una función puede ser amiga ('friend') de una clase. Una función que es amiga de una clase, tene acceso a su parte privada. Por ejemplo:

```
/* global */

class c_g {
   int i,j;
public:
   friend g();
} c1;

f()
{
    g1.i_g = 3;    /* error, no acceso a parte privada */
    g1.print();    /* correcto */
}

g()
{
    g1.i_g = 5;    /* correcto ! */
    g1.print();/* correcto */
 }
```

sucede lo siguiente:

como f() tiene de ancestro a 'global'.

'g1' SI se encuentra.

'i_g' se busca en los atributos PUBLICOS de 'g1', y NO se encuentra.

como g() tiene de ancestro a 'global' y de amigo a 'c1'.

'g1' SI se encuentra.

'i_g' se busca en los atributos PUBLICOS de 'g1', y NO se halla.

'i_g' se busca en los atributos PRIVADOS de 'c1', debido a que g() es amigo de la clase, y SI se encuentra.

Descripción de la Tabla de Símbolos (Ver Módulo TBLSYMO.C)

Los Símbolos que maneja el *Precompilador de C++ a C* son:

global	(GLOBAL)
clase	(CLASS) *
función	(FUNCTION)
bloque de función	(HEADER)
objeto	(OBJECT)

* se incluyen: 'class', 'struct', 'union' y 'enum'

Cada Símbolo tiene asociada una estructura. En los campos de esta estructura se almacena la información relacionada con el Símbolo.

La información acerca de un símbolo puede ser

propia ó
liga con otro Símbolo.

En la información propia de un Símbolo, se almacenan sus datos tales como:

Nombre del Símbolo	(NAME)
Tipo	(TYPE)
Nivel Sintáctico	(LEVEL)
Declarador	(DCLTR)

Una liga de un Simbolo con otro puede ser

Referencia	(Ref)
Ancestro	(Parent, FlgPar)
Atributo	(Attr, PubAttr, PrivAttr)
Amigo	(Friend)
Gemelo	(Twin)
Hermano	(Next)

con los siguientes significados:

Referencia (Ref)

'a' es Referencia de 'b', si 'b' es TYPEDEF o CLASS, y 'a' es de tipo 'b'.

Ancestro (Parent, FlgPar)

El ancestro de 'a' se fija de la siguiente forma:

Si 'a' es GLOBAL, su ancestro es NULO.

Si 'a' es 'class' base, su ancestro es 'global'.

Si 'a' es 'class' derivada, su Ancestro es su 'class' base.

Si 'a' es función no miembro, su Ancestro es 'global'.

Si 'a' es función miembro de una 'class', su ancestro es esa 'class'.

Atributo (Attr, PubAttr, PrivAttr)

'a' es Atributo de 'b', si 'a' esta declarado en el interior de 'b'.

Amigo (Friend)

'a' es una función Amiga de la clase 'b', si 'a' tiene permiso privado sobre los miembros de 'b'.

Gemelo (Twin)

'a()' es función gemela de 'ap()', si tienen el mismo nombre pero diferente número o tipo de argumentos (sobrecarga de nombres de función).

Hermano (Next)

'a' es hermano de 'b', si ambos son atributos del mismo símbolo.

En el campo Declarador (DCLTR) de un Símbolo se almacenan operadores de declaración. Los operadores de declaración son:

arreglo	(ARR)
apuntador	(PTR)
función	(FUNC)
referencia	(REF)
apuntador constante	(CPTR)

referencia constante (CREF)

Los operadores arreglo (ARRD) y función(FUNC), necesitan guardar información adicional. El operador arreglo (ARRD), necesita guardar un valor; y el operador función (FUNC), una lista de parámetros. Esta información adicional se almacena en otra estructura de datos llamada TABLA DE DIMENSIONES (TBL_DIMD. La Tabla de Dimensiones es descrita más adelante.

b) Tabla de Etiquetas. (TBL_LBL)

Las etiquetas se manejan en una tabla separada de la tabla de Simbolos. El alcance de una etiqueta es toda la función donde está declarada.

En esta tabla también se tiene información del árbol de destrucción de objetos de clase [DEW87]. Este árbol de describe en la siguiente sección.

La estructura detallada de la tabla de etiquetas y las funciones que la manejan, NO se encuentran en este trabajo.

c) Arbol de Destrucción. (TREE_DEST)

Una clase puede tener varios constructores, pero solo un destructor. A un objeto de clase que es declarado en un bloque, se le asigna la memoria necesaria, y se le aplica enseguida el constructor. Se le aplica su destructor a un objeto de clase, cuando sale de su entorno.

En esta tabla se almacena la información necesaria para la destrucción de objetos de clase (con destructor). Esta destrucción se hace al salir del bloque, en 'break's , en 'continue's, en 'goto's y en 'return's.

d) 'Stack' Semántico. (STK_SEM)

A diferencia de otros lenguajes, tales como C, PASCAL o MODULA. En C++ el alcance sintáctico y el semántico estan separados. La correspondencia entre ellos es llevada por el Precompilador de C++ a C a través del 'Stack' Semántico.

En el tiempo de "Precompilación" se tiene un apuntador (CURRENT) al Símbolo que define el alcance semántico para un cierto ambiente sintáctico. Cuando se entra a un nuevo ambiente sintáctico, se estrae del 'Stack' Semántico, el alcance sintáctico previo.

e) 'Stack' para Inicialización. (STK_INIT)

Es usado para la inicialización de arreglos y estructuras. La descripción detallada de campos y funciones asociadas NO se encuentran incluidas en este trabajo.

ARCHIVOS del Precompilador de C++ a C

a) Archivo de Código Ligado. (FI_COD)

La traducción de C++ a C no puede ser hecha paralelamente. Es decir, no se puede leer código en C++ y escribir código en C simultaneamente. Las razones principales son las siguientes:

Existencia de Funciones Anidadas.
Declaraciones con inicialización de objetos.

A diferencia de C, en C++ una función puede estar contenida sintácticamente en otra (siempre y cuando haya una declaración intermedia de clase).

En C++ una clase puede tener en su interior una función. A su vez esta función puede contener una clase, y así recursivamente. La traducción a C de este tipo de construcciones, requiere extraer las funciones internas y redefinirlas a nivel global.

Por otra parte, para traducir una declaración de un objeto de clase con inicializador, es necesario traducir solo la declaración y guardar la inicialización para emitirla al inicio de la parte de estatutos.

Un ejemplo de traducción se muestra a continuación:

Código en C++.

```
/* global */

int g;

int f()
{
   class c {
      int i;
   PUBLIC:
      int g()
      {
          int i;
      }
   }

}
```

Código equivalente en C.

```
/* global */

int g;

int f()
{
   struct c {      /* 'class' se implementa con 'struct' */
      int i;
    }
}

int _1g()          /* se extra la función */
{
   int i;
}
```

La solución que se dio a este problema fue el usar un archivo

temporal de código. Este archivo guarda el código de manera ligada. El código final se obtiene reescribiendo el archivo ligado en el orden dado por las ligas.

Se tienen las siguientes ligas.

liga global de declaraciones.
liga global de inicializaciones.
liga local de declaraciones.
liga local de estatutos.

Al hablar de archivo ligado, nos estamos refiriendo a este archivo.

b) Archivo de Código en C. (FI_COD_C)

Este archivo resulta de la reescritura del Archivo Ligado.

III. F A S E D E D E S A R R O L L O . .

III.1 INTRODUCCION.

En esta fase se desarrollaron en C los productos arrojados por la FASE DE DISEÑO. Se escribieron las declaraciones de las Estructuras de Datos y de Archivos. Se codificaron y probaron las funciones que manejan esas Estructuras y Archivos.

III.2 OBJETIVOS.

Especificar, Codificar y Probar los Módulos definidos en el Diseño del Sistema.

III.3 PRODUCTOS ENTREGADOS.

III.3.1 ESPECIFICACION DETALLADA DE MODULOS Y FUNCIONES.

III.3.2 MODULOS Y FUNCIONES PROBADOS.

III.3.3 (MODULOS INTEGRADOS PROBADOS).

III.3.4 (PRUEBA CON DATOS. COMPROBACION DE RESULTADOS ESPERADOS).

( ) NO FUERON ALCANZADOS.

III.3.1 ESPECIFICACION DETALLADA DE MODULOS Y FUNCIONES.

PROCESOS del Precompilador de C++ a C :

a) Análisis Léxico (Ver Módulo CPP.L)

Este es el Módulo que se le alimenta a 'LEX'. 'LEX' es un programa que produce analizadores léxicos. 'LEX' recibe como entrada, básicamente, una descripción de 'token's y de acciones.

Los token's se describen usando expresiones regulares, y las acciones son estatutos simples o compuestos de C. La salida de 'LEX' es el analizador léxico LEXYY.C .

El Módulo CPP.L tiene 4 partes:

declaraciones globales
definiciones
reglas
subrutinas de usuario

En las *declaraciones globales* se encuentra la descripción de lo que importa y exporta el Módulo.

En las *definiciones* se asigna un nombre a una expresión

regular. Una definición tiene 2 partes:

nombre

expresión regular

Las *reglas* constan tambien de 2 partes:

expresión regular

acción (estatuto en C)

Las *subrutinas de usuario* que se tienen son:

screen()

yywrap()

screen()

Es una rutina que, dada una palabra, regresa

su token, si es palabra reservada.

token de IDENTIFIER, en otro caso.

yywrap()

Esta rutina es llamada por la función principal del analizador léxico (yylex()), al terminar la lectura de 'token's. Si yywrap() devuelve

0, yylex() continua leyendo.

1, yylex() termina de leer.

b) Análisis Sintáctico y Semántico (Ver Apendice E)

Este es el Módulo que se le alimenta a 'YACC'. 'YACC' es un programa que produce analizadores sintácticos. 'YACC' recibe básicamente como entrada una gramática y acciones semánticas asociadas; y produce como salida un analizador sintáctico y semantico llamado YTAB.C .

'YACC' detecta los conflictos de la gramática y los hace saber. Estos conflictos pueden ser 'shift-reduce' o 'reduce-reduce'. En el Módulo CPP.Y. se comentan al principio los conflictos detectados en la gramática de C++ y su causa.

El Módulo CPP.Y tiene escencialmente 3 partes:

Código para el Preprocesador de C.

Definiciones

Reglas.

Código para el Preprocesador de C.

En esta parte se encuentran la descricion de lo que importa y exporta el Módulo.

Definiciones

Dentro de esta parte se encuentran.

- La definición de la unión que usara el 'Stack' del analizador sintáctico producido por 'YACC'.

- La definición de 'token's, incluyendo palabras reservadas y operadores. Aquí también se especifica que miembro de la unión usan los 'token's que necesitan guardar información en el 'Stack'.

- La especificación del miembro de la unión que usan los símbolos no terminales, que necesitan guardar información en el 'Stack'.

Reglas.

En las *Reglas* se especifica:

- La regla sintáctica.
- La acción semántica asociada, en las reglas que lo necesitan.

La gramática empleada para el análisis sintáctico de C++ es una extensión de una gramática LALR(1) de C [HARB84]. En su sintáxis se maneja la prioridad de operadores.

Al agregar las estructuras sintácticas de C++ a esta gramática, se obtuvieron conflictos.

Los conflictos de tipo 'reduce-reduce' fueron completamente eliminados. En estos conflictos, YACC elige como símbolo para ser reducido, el no terminal de la primero regla en conflicto. Aún así, estos conflictos indican ambigüedad en el lenguaje.

Los conflictos de tipo 'shift-reduce' fueron analizados. YACC elige la acción 'shift' en ellos. Si la elección de YACC era adecuada, se permitía el conflicto. En otro caso, se eliminaba.

Un ejemplo de conflicto 'shift-reduce' surgió al agregar la construcción sintáctica de 'clase derivada' que permite C++. El mensaje de YACC fue el siguiente:

62 shift/reduce conflict (shift 130, red'n 50) en ':'

```
state 62
    class_specifier : aggr class_tag_dcltr _      (50)
    class_specifier : aggr class_tag_dcltr _ (a0003)
    class_specifier : aggr class_tag_dcltr _  class_base_name
                      (a0003)
```

y su causa se detalla a continuación.

Debido a la regla :

```
    member_data_declaration
        : opt_decl_specifier  member_dcltr  ';'
```

donde

```
    member_dcltr
        : ':' const_exp
```

el símbolo ':' esta en el 'follow' de opt_decl_specifier Es por esto que se quiere hacer la acción 'reduce' sobre este símbolo.

Debido a la regla

```
    opt_decl_specifier
        : class_specifier
```

donde:

```
    class_specifier
```

```
        : aggr class_tag_dcltr
        : aggr class_tag_dcltr class_base_name '{' a0003 '}'
y     class_base_name
        : ':' TYPEDEF_NAME
```

es igual el 'follow' de los siguientes símbolos no terminales:

```
opt_class_specifier
class_specifier
class_tag_dcltr
```

y por tanto, ':' esta en el 'first' y en el 'follow' de class_tag_dcltr, y se tiene un conflicto 'shift-reduce'.

La elección de YACC en este caso fue acertada. Pues, después de un nombre de clase, considerará ':' como miembro de la clase base, y NO como campo de bits.

Conflictos de este tipo no son fáciles de visualizar. En la Gramática (de C++) para el *Precompilador de C++ a C*, se lograron reducir a 9. Estos conflictos, se comentan en la parte superior del módulo, con una breve explicación de su causa.

En el *Precompilador de C++ a C*, solo se tienen las rutinas que

realizan parte del análisis semántico de las declaraciones. No se incluye el tratamiento de errores sintácticos, tampoco la emisión de código.

Una declaración de un identificador consta de 4 partes.

Almacenamiento	(STGE)	Ej. static
Tipo	(TYPE)	Ej. int
Operadores de Declaración	(DCLTR)	Ej. *, ()
Nombre	(NAME)	Ej. static int * a()

En el análisis de las declaraciones, se usaran atributos sintetizados y variables globales.

El analizador léxico cuando lee un identificador regresara una estructura que contiene:

el nombre del identificador.
el Símbolo previo con el mismo nombre, si existe.

En el Símbolo 'local', se guardara el Almacenamiento, el Tipo y los Operadores de Declaración de un Identificador. El nombre y la referencia, que fueron regresados por el analizador léxico, se transmitiran ascendentemente en atributos sintetizados.

c) Emisión de Código.

NO INCLUIDA EN EL TRABAJO.

d) Compilación de Código.

NO INCLUIDA EN EL TRABAJO.

ESTRUCTURAS DE DATOS del Precompilador de C++ a C :

a) Tabla de Simbolos. (Ver Módulo TBLSYMO.C)

En este Módulo se describen:

Implementacion del tipo Simbolo.

Tipos de los campos de los Simbolos (y sus valores nulos).

Campos de los Simbolos.

Simbolos.

Funciones Básicas.

Funciones Conceptuales.

Implementacion del tipo Simbolo.

Como fue mencionado en la ETAPA DE DISEÑO, un Símbolo puede estar ligado a otro. En general, se tienen listas en la Tabla de Símbolos. Por esta causa el tipo Símbolo se implementó con una estructura que contiene:

- Una bandera indicadora del tipo de Símbolo.
- Un apuntador (a la memoria dedicada al Símbolo).

y que se puede ver como un apuntador con tipo.

La memoria dedicada al Símbolo es alojada dinámicamente.

Tipos de los campos de los Símbolos (y sus valores nulos).

Con el fin de poder modificar la representación de cierto campo de un Símbolo, sin tener la necesidad de modificar las funciones asociadas, se definieron con MACROS los tipos de los campos.

A cada tipo de campo se le asocia un valor nulo. Este valor nulo es usado en el valor de regreso de una función (posiblemente con error) y al limpiar el campo.

Campos de los Símbolos.

Los Símbolos tienen información semejante. Por ejemplo un Símbolo función (FUNCTION), un Símbolo clase(CLASS) y un Símbolo objeto (OBJECT) tienen el campo Nombre (NAME). Debido a esto se creo una lista de todos los posibles campos que cualquier Símbolo puede tener.

Símbolos.

Al definir la estructura para un Símbolo, se eligen los campos que este Símbolo puede ocupar. Por ejemplo, la estructura del

Símbolo objeto (OBJECT), ocupa los campos:

objeto	OBJECT
Nombre	FLD_NAME
Almacenamiento	FLD_STGE
Tipo	FLD_TYPE
Declarador	FLD_DCLTR
Nivel	FLD_LEVEL
Referencia	FLD_REF
Hermano	FLD_NEXT
Valor	FLD_VAL.

Esta estructura es usada también en el 'cast' del apuntador del tipo Símbolo, cuando se pide el valor de un campo de dicho Símbolo.

Funciones Básicas de la Tabla de Símbolos (Ver Módulo TBLSYM1.C)

Las *Funciones Básicas de la Tabla de Símbolos* se dividen en 2 grupos:

Funciones que manejan campos.

Funciones que manejan Símbolos.

Funciones que manejan campos.

Entre estas funciones se encuentran:

- Pedir el valor de un campo (GetFld())
- Dar el valor a un campo (PutFld())
- Imprimir el valor de un campo (PrintFld())

Funciones que manejan Símbolos.

Entre estas funciones se encuentran:

- Pedir el tipo de un Símbolo (GetSym())
- Preguntar si un Símbolo es nulo (NullSym())
- Alojar un Símbolo (en memoria dinámica).

(AllocGlobal(), AllocClass(), AllocFunction(), etc.)

- Imprimir un Símbolo (PrintSym())
- Borrar un Símbolo (ClearSym())
- Remover un Símbolo (Rem())
- Liberar la memoria de un Símbolo (Free())

Funciones Conceptuales de la Tabla de Simbolos.

(Ver Módulo TBLSYM2.C)

Las *Funciones Conceptuales de la Tabla de Simbolos*, se dividen en 2 grupos:

Funciones para campos.

Funciones para Simbolos.

Funciones para campos (Ver Módulo TBLSYM2.C)

En esta Módulo se describen :

Banderas

AddType();

AddStge(); *ChkStge()*;

AddDcltr(); *AddDcltrFunc()*; *AddDcltrArr()*;

Banderas

Algunos campos manejan banderas. Estas banderas se implementan en mapas de bits. Para el manejo de banderas, se definieron las

MACROS siguientes:

Flag(d,f)	Levanta la bandera 'f' del descriptor 'd'.
FlagP(d,f)	Pregunta por la bandera 'f' de 'd'.
RemFlag(d,f)	Baja la bandera 'f' del descriptor 'd'.

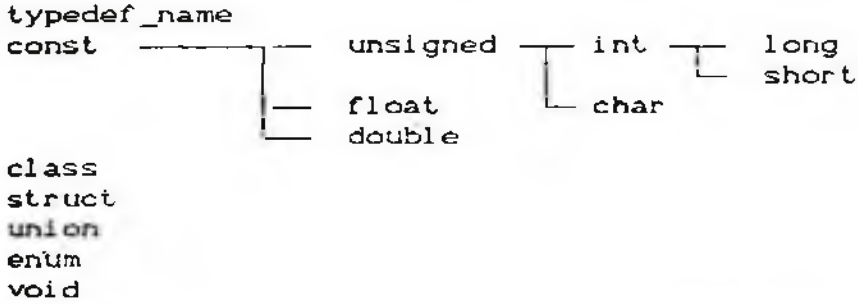
AddType()

Agrega una nueva bandera de Tipo, a un Símbolo.

Chequea que las banderas anteriores no sean excluyentes con la nueva. Si lo son, marca error, elimina las banderas previas excluyentes y levanta la nueva.

NOTA: Una bandera es excluyente consigo misma.

Los tipos permitidos son:



En la figura anterior se ve que cada bandera tiene un conjunto de banderas previas no excluyentes. Por ejemplo:

para 'char', son no excluyentes 'const' y 'unsigned'.
para 'int', lo son 'const','unsigned','long' y 'short'.

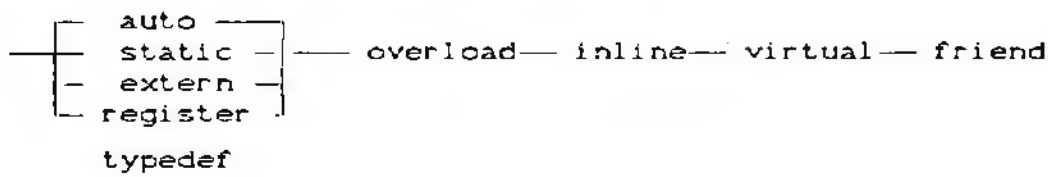
AddStge()

Agrega una nueva bandera de Almacenamiento, a un Símbolo.

Chequea que las banderas anteriores no sean excluyentes con la nueva. Si no lo son, marca error, elimina las banderas previas excluyentes y levanta la nueva.

NOTA: Una bandera es excluyente consigo misma.

Los Almacenamientos permitidos son:



En la figura anterior se ve que cada bandera tiene un conjunto de banderas previas no excluyentes no excluyentes. Por ejemplo, para 'auto', son no excluyentes: 'overload', 'inline', 'virtual' y 'friend'

ChkStge()

Chequea las banderas de Almacenamiento permitidas para un Símbolo, en cierto entorno. Esta rutina es llamada por las

Funciones para Simbolos.

Las banderas de Almacenamiento permitidas, dado un Simbolo y un entorno, se muestran en la siguiente tabla.

Simbolo	Ambiente			
	global	parametro de func.	bloque de func.	clase
Objeto	extern static typedef	register	auto extern register static typedef	static
decl. función	extern static (overload) (inline)		extern	(virtual) (friend)
def. función	static (inline)			(friend) [overload] [inline]

Con la siguiente convención:

bandera	permitida por C.
(bandera)	permitida por C++.
[bandera]	permitida por C++ (default).

AddDcltr(); *AddDcltrFunc();* *AddDcltrArr();*

Antes de describir estas rutinas, se dará una explicación de la

manera como se almacena un declarador.

Un declarador de un Símbolo tiene 2 partes

Descriptor

Apuntador a la Tabla de Dimensiones.

Descriptor.

En el *Descriptor* se guarda la secuencia de operadores del declarador. Estos operadores pueden ser

Apuntador	PTR
Arreglo	ARR
Función	FUNC
Apuntador Constante	CPTR
Referencia	REF

En la *TABLA DE DIMENSIONES* se guarda información extra del declarador. Esta tabla es descrita posteriormente.

Se usa un renglón de la *TABLA DE DIMENSIONES* para cada operador Arreglo (ARR) ó Función (FUNC). En ese renglón se almacena para:

Arreglo (ARR) dimensión (expresión) especificada.

Funcion (FUNC) Primer simbolo de la lista de argumentos.

Al llegar un nuevo declarador, se chequea contra el inmediato anterior. Si no hay compatibilidad entre los dos declaradores, se marca un error.

Cada operador de declaración tiene un conjunto de operadores previos permitidos. Esta relación se muestra en la siguiente tabla:

DCLTR	DCLTR's previos permitidos				
	PTR	ARR	FUNC	CPTR	REF
PTR	x	x	x		(x)
ARR	x	x		x	x
FUNC	x				
CPTR	x	x		x	(x)
REF	(x)		x	(x)	x

( ) banderas que se anulan mutuamente.

Por ejemplo, el análisis sintáctico de la declaración

```
/* global */
int  * (* f ) (float) [10]
```

arroja, debido a la prioridad, los siguientes operadores

PTR

FUNC(float)

ARR(10)

PTR

en ese orden. Esta declaración significa

"apuntador a una función (de argumento float) que regresa un arreglo (de 10 elementos) de apuntadores a enteros"

y se guarda en el declarador, mapa de bits, como

```
                    543     210   bits
PTR ←  ARR ← FUNC ←  PTR
       10    float
```

ya que cada operador ocupa 3 bits. (Ver Módulo TBLSYM2.C)

Por otra parte el *Apuntador a la TABLA DE DIMENSIONES* señala la localidad que contiene 'float'. Esta localidad esta ligada a otra que contiene '10'.

AddDeltr() (Ver TBLSYM2.C)

Adiciona un nuevo operador de declaración a un Símbolo. Cheque contra el operador inmediato anterior. Si hay incompatibilidad, marca error, y agrega el nuevo. En otro caso, solo agrega el nuevo operador.

AddDcltrFunc()

Tiene como entradas dos Símbolos. Adiciona un operador FUNC al primer Símbolo. Aloja un renglón para argumentos en la *TABLA DE DIMENSIONES*, y le asigna el segundo Símbolo. Este segundo Símbolo deberá ser el primer argumento del operador función.

AddDcltrArr()

Tiene como entradas un Símbolo y un Valor. Adiciona un operador ARR al Símbolo. Aloja un renglón para valor en la *TABLA DE DIMENSIONES* y guarda en ese renglón el Valor. Este Valor deberá ser la dimensión del arreglo.

TABLA DE DIMENSIONES (Ver Módulo TBLDIMO.C)

En este módulo se describen

Implementación del tipo "Dimensión".

Funciones Básicas.

Implementación del tipo "Dimensión".

El tipo "*Dimensión*" formado por

Bandera de tipo de *Valor*.

Valor.

Apuntador.

La *Bandera* indica si el *Valor* es

Símbolo (F_SYMD

Expresión (F_EXPD

El *Valor* debe ser por tanto

Símbolo (TBL_SYMD

Expresión (TREE_EXPD

El *Apuntador* sirve para ligar los renglones de la tabla. Este *Apuntador* puede usarse en la liga de lugares libres, ó bien, en la liga de referencias de operadores Arreglo (ARRD o Función (FUNCD de un Declarador (Ver Módulo TBLSYMD.CD.

Funciones Básicas.

Existen 2 grupos de *Funciones Básicas* para la *TABLA DE DIMENSIONES*:

Funciones para campos.

Funciones para renglones ("Dimensiones").

Funciones para campos.

Pide el *Valor* de un renglón	(GetValDim())
Da el *Valor* a un renglón	(PutValDim())
Imprime el *Valor* de un renglón	(PrintValDim())
Pide el *Apuntador* de un renglón	(GetNextDim())
Da el *Apuntados* a un renglón	(PutNextDim())
Imprime el *Apuntador* de un renglón	(PrintNextDim())

Funciones para renglones ("Dimensiones").

Aloja renglón para Símbolo	(AllocSymb())
Aloja renglón para Expresión	(AllocExp())

Imprimir renglón	(PrintDim())
Borrar renglón	(ClearDim())
Liberar renglón	(FreeDim())

Para mayor información ver TBLDIM0.C y TBLDIM1.C .

Funciones para Símbolos.

Las siguientes funciones aún NO han sido implementadas.

ChkClassCurrSpc()

AddClassCurrSpc()

AddInitializer()

AddCurrFuncDef()

AddCurrDcltr()

AddCurrFieldDcltr()

AddCurrEnumDcltr()

AddCurrFormDcltr()

AddMemberInit

b) Tabla de Etiquetas. (TBL_LBL)

Ver FASE DE DISEÑO. NO fue desarrollada.

c) Arbol de Destrucción. (TREE_DEST)

Ver FASE DE DISEÑO: NO fue desarrollada.

d) 'Stack' Semántico (Ver Módulo STKSEM0.C).

Esta estructura es un simple 'Stack' que almacena objetos de tipo Símbolo.

Las funciones que tiene son: (Ver Módulo STKSEM1.C)

Agregar un Símbolo al 'Stack' (PushStkSem())

Sacar un Símbolo del 'Stack' (PopStkSem())

e) 'Stack' para Inicialización. (STK_INIT)

Ver FASE DE DISEÑO. NO fue desarrollada.

ARCHIVOS del Precompilador de C++ a C

a) Archivo de Código Ligado. (FI_COD)

Hay 6 Módulos dedicados al manejo de este archivo:

Pedido y Liberado de Memoria Dinámica	(Móds. FIPTR0-1.C)
Tabla de Código	(Móds. TBLCOD0-1.C)
Lista de Código	(Móds. LSTCOD0-1.C)

Pedido y Liberado de Memoria Dinámica (Módulos FIPTR0-1.C)

En este Módulo se describen:

Tipo Apuntador a Archivo.	
Alojar Memoria en Archivo	(AllocFi())
Liberar Memoria de Archivo	(FreeFi())
Pedir Valor de la Liga	(PutLink())
Dar Valor a la liga	(GetLink())

Tipo Apuntador a Archivo.

Esta implementado con una estructura que contiene:

renglon ROW

columna COL

Para que esta representación tenga sentido, el archivo se maneja como un arreglo de caracteres, de ciertas dimensiones. Actualmente, n * 70. Donde n es el número de renglones. El valor máximo de n queda determinado por la memoria máxima en un archivo.

Alojar y Liberar Memoria de Archivo (AllocFi(), FreeFi())

El archivo se puede ver como un arreglo al que se le puede agregar el número de renglones que se necesite.

Inicialmente, el archivo es un arreglo sin renglones. Cualquier número de bytes que se pidan, provocará que se agregue un renglón. Si la memoria pedida es mayor que el tamaño del renglón, se marca un error En otro caso, se devuelve un apuntador al inicio del renglón. Si quedan suficientes bytes de memoria libre, se crea con ellos una celda de memoria libre, y en ese momento, el apuntador a la memoria libre apunta a esa celda.

Cuando se libera una celda de memoria, se liga al inicio de la lista de celdas de memoria libre.

Al pedir memoria, se recorre la liga de celdas libres. si existe alguna con suficiente memoria, se usa. Si no, se usa un nuevo renglón.

Pedir y Dar Valor a la liga (GetLink(), PutLink())

Estas funciones son usadas localmente para manejar la liga de celdas vacias. Sin embargo, también se exportan:

Piden y Dan valor a la liga de una celda de memoria en Archivo.

La liga de una celda se encuentra en los últimos bytes de información !. Por lo anterior, si se quieren usar estas funciones, se deberá pedir memoria adicional para la liga. El número de bytes que ocupa la liga también se exporta. y es NLINK.

Tabla de Código (Móds. TBLCOD0-1.C)

En este Módulo se describen

Implementación del Tipo TBL_COD.

Funciones para campos.

Funciones para renglones.

Implementación del Tipo TBL_COD.

Es una celda de memoria en archivo de tamaño variable y que cosnta de 2 partes:

Información (STRING)

Apuntador (NEXT)

Esta implementado con un apuntador a archivo (FI_PTR)

Funciones para campos.

Pide el valor de la Información (GetString())

Da el valor de la Información (PutString())

Pide el valor de la liga (GetNextTC()) *

Da valor a la liga (PutNextTC()) *

* con estas funciones, se pueden construir listas ligadas en archivo.

Funciones para renglones.

Alojar un renglón (AllocTC())

Liberar un renglón (FreeTC())

Preguntar si un renglón es nulo (NullTC())

Al alojar un renglón, solo se pide la memoria para la información, la celda contiene "internamente" la liga.

Lista de Código (Móds. LSTCODO-1.C)

En este Módulo, se describen:

Tipo Lista de Código (*LST_COD*)

Funciones (*InsStrBegLC(), InsStrEndLC()*)

Tipo Lista de Código (*LST_COD*)

Es una lista de celdas de memoria en archivo (TBL_COD), y esta implementada con un apuntador a una estructura que contiene

dos partes:

Apuntador al inicio de la lista (TBL_COD)

Apuntador al final de la lista (TBL_COD)

Funciones (*InsStrBegLC()*, *InsStrEndLC()*)

InsStrBegLC()

Con esta función hacemos crecer una lista de código por su parte final.

InsStrEndLC()

Con esta función podemos agregar un elemento al final de una lista.

b) Archivo de Código en C. (FI_COD_C)

Es el archivo secuencial, compilable en C, que se crea al finalizar el Análisis Sintáctico y Semántico.

III.3.2 MODULOS Y FUNCIONES PROBADOS.

Em el Apendice C, se incluye una lista de los Modulos mencionados. así como la parte de presentación de cada uno de ellos.

El Archivo de Descripción del Módulo C"include"

El Archivo de Implementación de las Rutinas Básicas

III.3.3 (MODULOS INTEGRADOS PROBADOS).

NO ALCANZADO.

III.3.4 (PRUEBA CON DATOS. COMPROBACION DE RESULTADOS ESPERADOS)

NO ALCANZADO.

V. F A S E D E P R U E B A.

I.1 INTRODUCCION.

En esta fase se debió probar el *Precompilador de C++* en su etapa final. Había que verificar que fueran satisfechos todos los requerimientos del usuario que se plantearon en la ETAPA DE DISEÑO. Además, también en esta fase, se deberían haber mejorado las características operacionales del *Precompilador de C++*, determinando las rutinas más críticas y optimizandolas, o reemplazandolas por otras mejores.

I.2 OBJETIVOS.

I.2.1 Verificar que los requerimientos del usuario sean satisfechos.

I.2.2 Lograr características operacionales óptimas.

I.2.3 Finalizar la prueba completa del Sistema.

I.3 PRODUCTOS ENTREGADOS.

El *Precompilador de C++* no fue terminado, por tanto, no pudo probarse completamente.

V I . F A S E D E I M P L E N T A C I O N .

VI.1 INTRODUCCION.

En esta fase se debía hacer pasar al *Precompilador de C++ a C* por cierto control de calidad. Este control sería el encargado de verificar que se cumpliera con todo lo prometido. Ademas revisaria que la programación y documentación cumpliera con los estándares para la Programación de Sistemas, de tal forma que fuera fácil su posterior mantenimiento.

VI.2 OBJETIVOS.

VI.2.1 Complementar la implementación de todos los procedimientos y funciones.

VI.2.2 Verificar que se cumpla con los criterios de aceptación y servicio de proceso.

Realmente lamento no haber podido entregar estos productos.

CONCLUSIONES

El *Precompilador de C++ a C* NO fue completamente terminado. Fue un intento de proveer una herramienta que facilitara el diseño y la programación. Sin embargo, lograr esto en un corto tiempo requiere, ó de un equipo de personas, ó de una gran experiencia en la construcción de Compiladores. Y no se tenían ninguna de esas 2 condiciones. Sin embargo, se desea anhelosamente lograr otro propósito de más alcance, sembrar la inquietud de construir algo que nos permita crear herramientas bien hechas. Ese "algo" es una Metodología de Desarrollo de Proyectos.

APENDICE A: METODOLOGIA DE DESARROLLO.

A continuación se describen brevemente los pasos que se deben seguir en cada una de las fases de la Metodología propuesta en este trabajo.

I. FASE DE ESTUDIO.

I.1 INTRODUCCION.

I.2 OBJETIVOS.

OBJETIVOS DEL PROYECTO.

REQUERIMIENTOS DEL USUARIO.

SOLUCIONES POTENCIALES.

SOLUCIONES POTENCIALES.

COSTOS Y BENEFICIOS

SOLUCION ESPECIFICA RECOMENDADA.

PLAN DEL PROYECTO.

I.3 PRODUCTOS A ENTREGAR.

DOCUMENTO DE OBJETIVOS DEL PROYECTO.

DOCUMENTO DE REQUERIMIENTOS DEL USUARIO.

DISEÑO CONCEPTUAL DE LA SOLUCION RECOMENDADA.

II. FASE DE DISEÑO.

II.1 INTRODUCCION.

II.2 OBJETIVOS.

VERIFICAR DECISIONES.

COMPLEMENTAR Y DOCUMENTAR EL DISEÑO.

II.3 PRODUCTOS A ENTREGAR.

II.3.1 DISEÑO EXTERNO.

VISTA DE LA ARQUITECTURA DEL SISTEMA.

FACILIDADES PARA EL USUARIO.

DESCRIPCION DE DATOS:

Diccionario de Datos.

Diseño lógico de las Bases de Datos.

II.3.2 DISEÑO INTERNO.

VISTA DEL PROCESO DE DATOS.

(Diagrama que muestra principales funciones, interfaces y dependencias).

DISENO DE LAS ESTRUCTURAS DE DATOS Y ARCHIVOS.

DESCRIPCION DE LAS TRANSACCIONES DEL SISTEMA.

(Organización de Módulos y Programas, Diagrama de Estructura, Estandares de Programación y Organización).

PROCEDIMIENTOS ESPECIALES.

(Excepción de los estándares).

III. FASE DE DESARROLLO.

III.1 INTRODUCCION.

III.2 OBJETIVOS.

ESPECIFICAR, CODIFICAR, PROBAR E INTEGRAR TODOS LOS MODULOS DEFINIDOS EN EL DISEÑO DEL SISTEMA.

PREPARAR LA PRUEBA Y LA DOCUMENTACION

III.3 PRODUCTOS A ENTREGAR.

ESPECIFICACION DETALLADA DE PROGRAMAS Y MODULOS.

PROGRAMAS Y MODULOS PROBADOS.

PROGRAMAS INTEGRADOS PROBADOS.

PRUEBA CON DATOS. COMPROBACION DE RESULTADOS ESPERADOS.

BOSQUEJO DE LA DOCUMENTACION DEL MANEJO DEL SISTEMA.

BOSQUEJO DE LA DOCUMENTACION DE LOS SERV. DE PROCESAMIENTO.

BOSQUEJO DE LA DOCUMENTACION PARA EL USUARIO.

IV. FASE DE PRUEBA.

IV.1 INTRODUCCION.

IV.2 OBJETIVOS.

QUE LOS REQUERIMIENTOS DEL USUARIO SEAN SATISFECHOS.

CARACTERISTICAS OPERACIONALES OPTIMIZADAS.

PRUEBA DEL SISTEMA COMPLETO FINALIZADA.

IV.3 PRODUCTOS A ENTREGAR.

PRUEBA COMPLETA DEL SISTEMA.

V. FASE DE IMPLEMENTACION.

V.1 INTRODUCCION.

V.2 OBJETIVOS.

COMPLEMENTAR LA IMPLEMENTACION DE TODOS LOS PROCEDIMIENTOS Y FUNCIONES DEL SISTEMA.

ACEPTAR QUE EL SISTEMA CUMPLA CON LOS CRITERIOS DE ACEPTACION Y SERVICIO DE PROCESAMIENTO.

V.3 PRODUCTOS ENTREGADOS.

SISTEMA FUNCIONANDO.

DOCUMENTACION PARA EL USUARIO.

DOCUMENTACION DEL MANEJO DEL SISTEMA.

DOCUMENTACION DE LOS SERVICIOS DE PROCESO.

CODIGO DE LOS MODULOS.

VI. CONCLUSIONES.

APENDICE B. ESTANDARES DE PROGRAMACION.

A continuación se describen los siguientes puntos:

Propósito de los Estándares.

Módulo de Código.

Estándares para declaraciones.

Estándares para código.

Propósito de los Estándares.

Dar consistencia y manteribilidad a un Sistema.

Módulo de Código.

Es un archivo compilable en C. Los rutines que constituyen un módulo, realizan funciones relacionadas lógicamente, ó forman un conjunto para realizar una función específica.

Un módulo de código esta formado de 2 partes:

Módulo de Descripción.

Módulo de Implementación.

En el *Módulo de Descripción*, se incluyen las declaraciones de los tipos, variables y funciones que exporta el Módulo.

En el *Módulo de Implementación*, se construyen estas funciones, con la ayuda de ciertos tipos, variables y funciones importadas. Aquí se pueden tener también tipos, variables y funciones locales.

Ambos *Módulos*, tienen en su parte superior un encabezado que incluye:

Nombre del Proyecto.

Nombre del Módulo y Fecha.

Descripción del Módulo.

Programador.

Las rutinas del *Módulo de Implementación*, también tienen un

Una declaración por renglón, comentando enseguida el uso de la variable.

Los nombres de las identificadores están formados por 2 partes:

prefijo.

sufijo.

El *prefijo* depende del tipo de uso del identificador. El sufijo es un nemónico para diferenciar un identificador de otro.

Los tipos de prefijos son:

prefijo	*significado*
n	longitud
i,j,k	indice
f	bandera
c	caracter
s	'string'
r	estructura
p	apuntador
a	arreglo
x,y,z	otras

Las letras MAYUSCULAS se usan para 'DEFINE's, 'MACROS' y 'TYPEDEF's.

Las variables locales se agrupan lógicamente de acuerdo a su tarea.

Estandares para Código.

El *Código* se encuentra dividido en bloques. Cada bloque tiene un comentario en su parte superior que describe su acción. Se usa una identación de 3 espacios en estatutos compuestos y siguiendo el formato "tradicional" de C. Este formato, en algunos de los estatutos, se muestra a continuación:

```
if (   ) {
...
}
else {
...
}
```

```
while (...) {
...
}
```

```
do {
...
} while (...);
```

```
for (... ; ... ; ... ) {
...
}
```

```
switch (...) {
   case ...  :
   ...
   default   :
}
```

y para la continuación de línea:

```
if (   ...   ;   ...   ;
/**/     ... ) {
}
```

Los comentarios tienen el alcance dado por la identación.

En las condiciones no esperadas se mensajes de errores de programación ('bug's).

```
/*****************************************************************/
/* PROYECTO    : Precompilador C++ -> C                          */
/*****************************************************************/
/*                                                               */
/* MODULO      : TBLSYM0.C                          Mar 22/89    */
/*                                                               */
/* DESCRIPCION : 'Include' para                                  */
/*               Entrada y Salida de TABLA DE SIMBOLOS           */
/*               (v. TBLSYM1.C)                                  */
/*                                                               */
/* PROGRAMADOR : Ruben Rusiles                                   */
/*                                                               */
/*****************************************************************/


/*****************************************************************/
/* IMPORTA                                                       */
/*****************************************************************/
#include "message0.c"
#include "treexp0.c"

/*****************************************************************/
/* EXPORTA                                                       */
/*****************************************************************/

/*****************************************************************/
/* tipos de campos                                               */
/*****************************************************************/
typedef             /* TBL_SYM */
   struct {
      int    i_type;    /* tipo: T_GLOBAL - T_CLASS */
      char *    ptr;    /* apuntador                */
   } TBL_SYM;
#define   NAME        char *
```

```
#define   STGE        unsigned int
#define   TYPE        unsigned int
#define   DCLTR       unsigned long int
#define   LEVEL       int
#define   TOKEN       int
#define   PARENT      TBL_SYM
#define   FLGPAR      int
#define   TWIN        TBL_SYM
#define   ATTR        TBL_SYM
#define   HEAD        TBL_SYM
#define   TREE_DEST   int
#define   TBL_LBL     int

/*****************************************************************/
/* valores nulos                                                 */
/*****************************************************************/
extern  TBL_SYM          NULL_TBL_SYM; /* definido en TBLSYM1.C */
#define NULL_NAME        (NAME)       "NULL_NAME"
#define NULL_STGE        (STGE)       0
#define NULL_TYPE        (TYPE)       0
#define NULL_DCLTR       (DCLTR)      0
#define NULL_LEVEL       (LEVEL)      0
#define NULL_TOKEN       (TOKEN)      0
#define NULL_PARENT      NULL_TBL_SYM
#define NULL_FLGPAR      (FLGPAR)     0
#define NULL_TWIN        NULL_TBL_SYM
#define NULL_ATTR        NULL_TBL_SYM
#define NULL_HEAD        NULL_TBL_SYM
#define NULL_TREE_DEST   (TREE_DEST)  0
#define NULL_TBL_LBL     (TBL_LBL)    0


/*****************************************************************/
/* nombres de campos                                             */
```

```
/*****************************************************************/
#define FLD_NAME        NAME        s_name;
                                    /* Mi nombre                  */
#define FLD_STGE        STGE        d_stge;
                                    /* Mi descriptor              */
                                    /*  de almacenamiento         */
#define FLD_TYPE        TYPE        d_type;
                                    /* Mi descriptor de tipo      */
#define FLD_DCLTR       DCLTR       d_dcltr;
                                    /* Mi descriptor              */
                                    /*  de declaradores           */
#define FLD_LEVEL       LEVEL       i_lvl;
                                    /* Mi nivel sintactico        */
#define FLD_OPER        TOKEN       i_oper;
                                    /* Operador sobrecargado      */
#define FLD_CONV        TOKEN       i_conv;
                                    /* El tipo al que realizo     */
                                    /*  una conv. implicita       */
#define FLD_STMT        TOKEN       i_stmt;
                                    /* Instruccion que tengo      */
                                    /*  asociada                  */
#define FLD_REF         TBL_SYM     p_ref;
                                    /* Mi referencia              */
                                    /*  una 'CLASS', 'TYPEDEF'    */
#define FLD_PARENT      PARENT      p_par;
                                    /* Mi ancestro (liga          */
                                    /*  semantica)                */
#define FLD_FLGPAR      FLGPAR      f_par;
                                    /* Mi tipo de ancestro        */
                                    /*  (privado o publico)       */
#define FLD_NEXT        ATTR        p_next;
                                    /* Atributo siguiente de      */
                                    /*  mi entorno                */
#define FLD_TWIN        TWIN        p_twin;
```

```
                                        /* Mi otra instancia con  */
                                        /*  diferentes argumentos */
#define FLD_MEMB_INIT ATTR      p_minit;
                                        /* Mis miembros que se    */
                                        /*  inicializan           */
#define FLD_ATTR      ATTR      p_attr;
                                        /* Mi primer atributo     */
#define FLD_PUB_ATTR  ATTR      p_pub;
                                        /* Mi primer atributo     */
                                        /*  publico               */
#define FLD_PRIV_ATTR ATTR      p_priv;
                                        /* Mi primer atributo     */
                                        /*  privado               */
#define FLD_FLG_MEMB  FLGPAR    f_memb;
                                        /* Mi bandera de atrib.   */
                                        /*  que estan declarando  */
#define FLD_CURR_HEAD HEAD      p_chead;
                                        /* Mi bloque actual       */
#define FLD_ENC_HEAD  HEAD      p_ehead;
                                        /* Bloque que me encierra */
#define FLD_DTOR      ATTR      p_destor;
                                        /* Destructor de los      */
                                        /*  objetos de mi clase   */
#define FLD_TREE_DEST TREE_DEST p_tdest;
                                        /* Arbol de destruccion   */
                                        /*  de objetos de mi clase*/
#define FLD_LBL       TBL_LBL   p_lbl;
                                        /* Mi primera etiqueta    */
#define FLD_VAL       TREE_EXP  p_val;
                                        /* Valor o Inicializador  */


/*************************************************************/
/* estructuras de los simbolos                               */
```

```
/**********************************************************************
typedef                 /* GLOBAL */
   struct {
      FLD_LEVEL
      FLD_PARENT
      FLD_ATTR
      FLD_TREE_DEST
   } GLOBAL;

typedef                 /* CLASS */
   struct {
      FLD_NAME
      FLD_LEVEL
      FLD_PARENT
      FLD_FLGPAR
      FLD_NEXT
      FLD_PUB_ATTR
      FLD_PRIV_ATTR
      FLD_FLG_MEMB
      FLD_DTOR
   } CLASS;

typedef                 /* FUNCTION */
   struct {
      FLD_NAME
      FLD_STGE
      FLD_TYPE
      FLD_DCLTR
      FLD_LEVEL
      FLD_OPER
      FLD_CONV
      FLD_REF
      FLD_PARENT
      FLD_NEXT
```

```
        FLD_TWIN
        FLD_MEMB_INIT
        FLD_ATTR
        FLD_CURR_HEAD
        FLD_TREE_DEST
        FLD_LBL
    } FUNCTION;

typedef                 /* HEADER */
    struct {
        FLD_LEVEL
        FLD_STMT
        FLD_ATTR
        FLD_ENC_HEAD
        FLD_TREE_DEST
    } HEADER;

typedef                 /* OBJECT */
    struct {
        FLD_NAME
        FLD_STGE
        FLD_TYPE
        FLD_DCLTR
        FLD_LEVEL
        FLD_REF
        FLD_NEXT
        FLD_VAL
    } OBJECT;

/*****************************************************************/
/* manejo de banderas                                            */
/*****************************************************************/
#define FlagP(d,f)    ( (d&f) ? 1 : 0 )
#define Flag(d,f)     ( (d) |= (f) )
```

```
#define RemFlag(d,f) ( (d) &= ~(f) )

/*****************************************************************/
/* banderas de tipo de simbolo                                   */
/*****************************************************************/
#define   F_NONE       0
#define   F_GLOBAL     1        /* tipos de simbolos */
#define   F_CLASST     2        /* T diferencia de DCLTR */
#define   F_FUNCTION   3
#define   F_HEADER     4
#define   F_OBJECT     5

/*****************************************************************/
/* banderas de clase de almacenamiento ('stge')                  */
/*****************************************************************/

/*          bandera      definicion       2 bytes = 16 banderas   */

#define   F_AUTO          (STGE) 0x0001
#define   F_STATIC        (STGE) 0x0002
#define   F_EXTERN        (STGE) 0x0004
#define   F_REGISTER      (STGE) 0x0008
#define   F_TYPEDEF       (STGE) 0x0010
#define   F_OVERLOAD      (STGE) 0x0020
#define   F_INLINE        (STGE) 0x0040
#define   F_VIRTUAL       (STGE) 0x0080
#define   F_FRIEND        (STGE) 0x0100
#define   F_ANYSTGE       (STGE) 0xffff

/*          bandera              banderas NO excluyentes

   line FY_AUTO     ( F_OVERLOAD|F_INLINE|F_VIRTUAL|F_FRIEND )
```

```
#define FY_STATIC    ( F_OVERLOAD|F_INLINE|F_VIRTUAL|F_FRIEND )
#define FY_EXTERN    ( F_OVERLOAD|F_INLINE|F_VIRTUAL|F_FRIEND )
#define FY_REGISTER  ( ~F_ANYSTGE )
#define FY_TYPEDEF   ( ~F_ANYSTGE )
#define FY_OVERLOAD  ( F_ANYSTGE & ~(F_OVERLOAD|F_TYPEDEF
                                     |F_REGISTER))
#define FY_INLINE    ( F_ANYSTGE & ~(F_INLINE  |F_TYPEDEF
                                     |F_REGISTER))
#define FY_VIRTUAL   ( F_ANYSTGE & ~(F_VIRTUAL |F_TYPEDEF
                                     |F_REGISTER))
#define FY_FRIEND    ( F_ANYSTGE & ~(F_FRIEND  |F_TYPEDEF
                                     |F_REGISTER))

/*****************************************************************/
/* banderas de tipo ('type')                                     */
/*****************************************************************/

/*          bandera          definicion          2 bytes = 16 banderas*/

#define   F_TYPEDEF_NAME  (TYPE) 0x0001
#define   F_CHAR          (TYPE) 0x0002
#define   F_FLOAT         (TYPE) 0x0004
#define   F_DOUBLE        (TYPE) 0x0008
#define   F_INT           (TYPE) 0x0010
#define   F_SHORT         (TYPE) 0x0020
#define   F_LONG          (TYPE) 0x0040
#define   F_UNSIGNED      (TYPE) 0x0080
#define   F_VOID          (TYPE) 0x0100
#define   F_CLASS         (TYPE) 0x0200
#define   F_STRUCT        (TYPE) 0x0400
#define   F_UNION         (TYPE) 0x0800
#define   F_ENUM          (TYPE) 0x1000
```

```
#define   F_CONST          (TYPE) 0x2000
#define   F_ANYTYPE        (TYPE) 0xffff

/*        bandera                banderas NO excluyentes          */

#define   FY_TYPEDEF_NAME  ( ~F_ANYTYPE )
#define   FY_CHAR          ( F_CONST | F_UNSIGNED )
#define   FY_FLOAT         ( F_CONST )
#define   FY_DOUBLE        ( F_CONST )
#define   FY_INT           ( F_CONST|F_UNSIGNED|F_LONG|F_SHORT )
#define   FY_SHORT         ( F_CONST|F_UNSIGNED|F_INT )
#define   FY_LONG          ( F_CONST|F_UNSIGNED|F_INT )
#define   FY_UNSIGNED      ( F_CONST|F_INT     |F_LONG|F_SHORT )
#define   FY_VOID          ( ~F_ANYTYPE )
#define   FY_CLASS         ( ~F_ANYTYPE )
#define   FY_STRUCT        ( ~F_ANYTYPE )
#define   FY_UNION         ( ~F_ANYTYPE )
#define   FY_ENUM          ( ~F_ANYTYPE )
#define   FY_CONST         (F_UNSIGNED|F_FLOAT|F_DOUBLE|F_INT|
                            F_CHAR|F_SHORT|F_LONG);

/*****************************************************************/
/* banderas de declarador ('dcltr')                              */
/*****************************************************************/
/* 4 bytes.3 bits/band => 9 band + F_NODCL  */

/*        bandera          definicion          */

#define   N_FLG_DCLTR      3                    /* bits que ocupa */
#define   F_NODCL          (DCLTR) 0x0000
#define   F_ARR            (DCLTR) 0x0001
#define   F_PTR            (DCLTR) 0x0002
#define   F_FUNC           (DCLTR) 0x0003
```

```
#define   F_REF          (DCLTR) 0x0004
#define   F_CPTR         (DCLTR) 0x0005
#define   F_CREF         (DCLTR) 0x0006
#define   F_ANYDCL       (DCLTR) 0x0007


/*****************************************************************/
/* banderas de ancestro ('parent')                               */
/*****************************************************************/
#define   F_PRIV         1
#define   F_PUB          2


/*****************************************************************/
/* funciones para CAMPOS                                         */
/*****************************************************************/

/* TblSym */

extern int        GetSym();
extern int        NullSym();

/* PUT's */

extern NAME       PutName();
extern STGE       PutStge();
extern TYPE       PutType();
extern DCLTR      PutDcltr();
extern LEVEL      PutLevel();
extern TOKEN      PutOper();
extern TOKEN      PutConv();
extern TOKEN      PutStmt();
extern TBL_SYM    PutRef();
extern PARENT     PutParent();
```

```
extern FLGPAR      PutFlgPar();
extern ATTR        PutNext();
extern TWIN        PutTwin();
extern ATTR        PutMembInit();
extern ATTR        PutAttr();
extern ATTR        PutPubAttr();
extern ATTR        PutPrivAttr();
extern FLGPAR      PutFlagMemb();
extern HEAD        PutCurrHead();
extern HEAD        PutEncHead();
extern ATTR        PutDtor();
extern TREE_DEST   PutTreeDest();
extern TBL_LBL     PutLbl();
extern TREE_EXP    PutVal();

/* GET's */

extern NAME        GetName();
extern STGE        GetStge();
extern TYPE        GetType();
extern DCLTR       GetDcltr();
extern LEVEL       GetLevel();
extern TOKEN       GetOper();
extern TOKEN       GetConv();
extern TOKEN       GetStmt();
extern TBL_SYM     GetRef();
extern PARENT      GetParent();
extern FLGPAR      GetFlgPar();
extern ATTR        GetNext();
extern TWIN        GetTwin();
extern ATTR        GetMembInit();
extern ATTR        GetAttr();
extern ATTR        GetPubAttr();
extern ATTR        GetPrivAttr();
```

```
extern FLGPAR      GetFlgMemb();
extern HEAD        GetCurrHead();
extern HEAD        GetEncHead();
extern ATTR        GetDtor();
extern TREE_DEST   GetTreeDest();
extern TBL_LBL     GetLbl();
extern TREE_EXP    GetVal();

/* PRINT's */

extern NAME        PrintName();
extern STGE        PrintStge();
extern TYPE        PrintType();
extern DCLTR       PrintDcltr();
extern LEVEL       PrintLevel();
extern TOKEN       PrintOper();
extern TOKEN       PrintConv();
extern TOKEN       PrintStmt();
extern TBL_SYM     PrintRef();
extern PARENT      PrintParent();
extern FLGPAR      PrintFlgPar();
extern ATTR        PrintNext();
extern TWIN        PrintTwin();
extern ATTR        PrintMembInit();
extern ATTR        PrintAttr();
extern ATTR        PrintPubAttr();
extern ATTR        PrintPrivAttr();
extern FLGPAR      PrintFlgMemb();
extern HEAD        PrintCurrHead();
extern HEAD        PrintEncHead();
extern ATTR        PrintDtor();
extern TREE_DEST   PrintTreeDest();
extern TBL_LBL     PrintLbl();
extern TREE_EXP    PrintVal();
```

```
/*****************************************************************
/* funciones para SIMBOLOS                                       *
/*****************************************************************
/* ALLOC's */

extern TBL_SYM    AllocGlobal();
extern TBL_SYM    AllocClass();
extern TBL_SYM    AllocFunction();
extern TBL_SYM    AllocHeader();
extern TBL_SYM    AllocObject();

/* PRINT */
extern TBL_SYM    Print();

/* CLEAR */
extern TBL_SYM    Clear();

/* REMOVE's */
extern TBL_SYM    Remove();

/* FREE */
extern TBL_SYM    Free();
```

```
%{
/*****************************************************************/
/*                                                               */
/* MODULO       : CPP.L          Version: 1.0          Feb 18/89 */
/*                                                               */
/* CAMBIOS      :                                                */
/*                                                               */
/* DESCRIPCION : Analisis Lexico de C++ (entrada para LEX.EXE)   */
/*            LEXYY.C es el analizador lexico (salida de LEX.EXE)*/
/*                                                               */
/* IMPORTA      : ***                                            */
/*                                                               */
/* EXPORTA      : yywrap()                                       */
/*                yynerrs                                        */
/*                                                               */
/* LOCALES      : screen()                                       */
/*                main()                                         */
/*                                                               */
/* NOTAS        : Version de prueba.                             */
/*                NO HACE BIEN LA BUSQUEDA BINARIA               */
/*                                                               */
/* PROGRAMADOR : Ruben Rusiles                                   */
/*                                                               */
/*****************************************************************/


/*****************************************************************/
/* IMPORTA                                                       */
/*****************************************************************/
#include "ytab.h"     /* tabla de tokens creada por YACC.EXE */
#define END(v) (v-1+sizeof v / sizeof v[0])
#define token(x) x
```

```
/*****************************************************************/
/* EXPORTA                                                       */
/*****************************************************************/
int yynerrs=0;


/*****************************************************************/
/* LOCALES                                                       */
/*****************************************************************/
static int screen();
main();
static struct rwtable {        /* tabla de palabras reservadas */
   char * rw_name;                /* representacion */
   int rw_yylex;                  /* valor de yylex() */
}
rwtable[] = {                  /* ordenadadas */
   "auto"     ,   AUTO       ,
   "break"    ,   BREAK      ,
   "case"     ,   CASE       ,
   "char"     ,   CHAR       ,
   "class"    ,   CLASS      ,
   "const"    ,   CONST      ,
   "continue" ,   CONTINUE   ,
   "default"  ,   DEFAULT    ,
   "delete"   ,   DELETE     ,
   "do"       ,   DO         ,
   "double"   ,   DOUBLE     ,
   "else"     ,   ELSE       ,
   "enum"     ,   ENUM       ,
   "extern"   ,   EXTERN     ,
   "float"    ,   FLOAT      ,
   "for"      ,   FOR        ,
   "friend"   ,   FRIEND     ,
   "goto"     ,   GOTO       ,
```

```
    "if"       ,   IF          ,
    "inline"   ,   INLINE      ,
    "int"      ,   INT         ,
    "long"     ,   LONG        ,
    "new"      ,   NEW         ,
    "operator" ,   OPERATOR    ,
    "overload" ,   OVERLOAD    ,
    "public"   ,   PUBLIC      ,
    "register" ,   REGISTER    ,
    "return"   ,   RETURN      ,
    "short"    ,   SHORT       ,
    "sizeof"   ,   SIZEOF      ,
    "static"   ,   STATIC      ,
    "struct"   ,   STRUCT      ,
    "switch"   ,   SWITCH      ,
    "this"     ,   THIS        ,
    "typedef"  ,   TYPEDEF     ,
    "union"    ,   UNION       ,
    "unsigned" ,   UNSIGNED    ,
    "virtual"  ,   VIRTUAL     ,
    "void"     ,   VOID        ,
    "while"    ,   WHILE       ,
};

/* definicion de tokens */

%%

let                     [a-zA-Z]
spc                     [ ]

dig                     [0-9]
nonzero_dig             [1-9]
```

```
oct_dig          [0-7]
hex_dig          [0-9a-fA-F]
long_mark        [l|L]
hex_mark         ("0"[xX])
dig_seq          ({dig}+)
dot_digs         ({dig_seq}"."{dig_seq}?|"."{dig_seq})
exp              ([eE][+-]?{dig_seq})
dec_const        ({nonzero_dig}{dig}*{long_mark}?)
oct_const        ("0"{oct_dig}*{long_mark}?)
hex_const        ({hex_mark}{hex_dig}+{long_mark}?)
float_const      ({dig_seq}{exp}|{dot_digs}{exp}?)

char_esc_code    [ntbrfv\]
num_esc_code     ({oct_dig}{oct_dig}{oct_dig}?)?)
esc_code         ({char_esc_code}|{num_esc_code})
esc_char         ("\"{esc_code})

print_char       ({let}|{dig})
char             ({print_char}|{esc_char})
char_const       (""{char}"")
string_const     (""{char}*"")

first_char       ({let}|"_")
follow_char      ({let}|"_"|{dig})
ident            ({first_char}{follow_char}*)

other
```

```
%%

"->"            return token(MG);
"++"            return token(PP);
"--"            return token(MM);
"<<"            return token(LL);
">>"            return token(GG);
"<="            return token(LE);
">="            return token(GE);
"=="            return token(EE);
"!="            return token(NE);
"&&"            return token(AA);
"||"            return token(OO);
"+="            return token(PE);
"-="            return token(ME);
"*="            return token(TE);
"/="            return token(DE);
"%="            return token(MDE);
">>="           return token(GGE);
"<<="           return token(LLE);
"&="            return token(AE);
"^="            return token(XE);
"|="            return token(OE);

{dec_const}     return token(DEC_INT);
```

```
{hex_const}             return token(HEX_INT);
{oct_const}             return token(OCT_INT);
{float_const}           return token(FLOATLIT);
{char_const}            return token(CHARLIT);
{string_const}          return token(STRING);

{ident}                 return screen();

{spc}+                  ;

{other}                 return token(yytext[0]);

%%

/*****************************************************************/
/*                                                               */
/* RUTINA     : screen()                                         */
/*                                                               */
/* FUNCION    : Dado un identificador, devuelve                  */
/*                token correspondiente, si es palabra reservada */
/*                 token de IDENTIFIER, en otro caso             */
/*              mediante una busqueda binaria.                   */
/*                                                               */
/*****************************************************************/
```

```
static int screen()
{
   int t;

   struct rwtable      /* apunts. a la tabla de palabras resev. */
      *plow = rwtable,          /* inferior */
      *phigh = END(rwtable),    /* superior */
      *pmid;                    /* medio */

   while (plow<=phigh) {
      pmid=plow+(phigh-plow)/2;
      if ( (t=strcmp(pmid->rw_name,yytext)) == 0 )
         return pmid->rw_yylex;
      else if (t<0)
         plow = pmid+1;
      else
         phigh = pmid-1;
    }
    return token(IDENTIFIER);
}

/*****************************************************************/
/*                                                               */
/* RUTINA     : yywrap()                                         */
/*                                                               */
/* FUNCION    : Llamada por yylex() al terminar la lectura.      */
```

```
/*              si devuelve                                          */
/*                  1 - yylex() termina.                             */
/*                  0 - yylex() continua leyendo                     */
/*                                                                   */
/*******************************************************************/
int yywrap()
{
   return(1);
}
```

```
/*****************************************************************/
/*                                                               */
/* MODULO      : CPP.Y        Version: 1.0          Feb 19/89    */
/*                                                               */
/* DESCPRICION: Analisis Sintactico de C++                       */
/*                                                               */
/* NOTAS       : 9 Conflictos shift_reduce                       */
/*                                                               */
/*               1. member_data_declaration                      */
/*                  : opt_decl_specifier / : const_exp (bit_field)*/
/*                  class_specifier  (shift)                     */
/*                    : aggr class_tag_dcltr / : TYPEDEF_NAME    */
/*                                                               */
/*               2. p2_dcltr                                     */
/*                     : OPERATOR operator / = init_exp          */
/*                  assgn_exp                                    */
/*                     : OPERATOR operator / assgn_exp           */
/*                                                               */
/*               3. local_data_declaration                       */
/*                     : type_specifier / dcltr ;                */
/*                  cpp_exp (shift)                              */
/*                     : simple_type_name / ( list_exp )         */
/*               4. simple_type_name                             */
/*                     : TYPEDEF_NAME /                          */
/*                  id_name (shift)                              */
/*                     : TYPEDEF_NAME / CC IDENTIFIER            */
/*                  id_name (shift)                              */
/*                     : TYPEDEF_NAME / CC operator_function_name*/
/*            5,6. p2_dcltr                                      */
/*                     : OPERATOR operator / ( list_exp )        */
/*                     : OPERATOR operator / ()                  */
/*                     : OPERATOR operator / ( list_exp )        */
/*               7. exp                                          */
/*                     : NEW TYPEDEF_NAME /                      */
/*                  exp (shift)                                  */
/*                     : NEW TYPEDEF_NAME / ( list_exp )         */
/*               8. d_name                                       */
/*                     : IDENTIFIER /                            */
/*                  member_dcltr (shift)                         */
/*                     : IDENTIFIER / : const_exp                */
/*               9. if_stmt                                      */
/*                     : IF ( list_exp ) stmt /                  */
/*                  ifelse_stmt                                  */
/*                     : IF ( list_exp ) stmt / ELSE stmt        */
/*                                                               */
/*               En 2,3,4,5 y 6, se elige la expresion en lugar d*/
/*               la declaracion, dentro de una funcion. .        */
/*                                                               */
/* PROGRAMADOR: Ruben Rusiles                                    */
/*                                                               */
/*****************************************************************/
```

```
%token AUTO
%token BREAK
%token CASE
%token CHAR
%token CLASS
%token CONST
%token CONTINUE
%token DEFAULT
%token DELETE
%token DO
%token DOUBLE
%token ELSE
%token ENUM
%token EXTERN
%token FLOAT
%token FOR
%token FRIEND
%token GOTO
%token IF
%token INLINE
%token INT
%token LONG
%token NEW
%token OPERATOR
%token OVERLOAD
%token PUBLIC
%token REGISTER
%token RETURN
%token SHORT
%token SIZEOF
%token STATIC
%token STRUCT
%token SWITCH
%token THIS
%token TYPEDEF
%token UNION
%token UNSIGNED
%token VIRTUAL
%token VOID
%token WHILE

%token IDENTIFIER
%token TYPEDEF_NAME
%token DEC_INT
%token HEX_INT
%token OCT_INT
%token FLOATLIT
%token STRING
%token CHARLIT

%token
%token  }
```

```
%token ','
%token '('
%token ':'
%token '='
%token ')'
%token '['
%token '*'
%token ']'
%token '.'
%token '&'
%token '-'
%token '!'
%token '~'
%token '/'
%token '%'
%token '+'
%token '<'
%token '>'
%token '^'
%token '|'
%token '?'

%token MG /* -> */
%token PP /* ++ */
%token MM /* -- */
%token LL /* << */
%token GG /* >> */
%token LE /* <= */
%token GE /* >= */
%token EE /* == */
%token NE /* != */
%token AA /* && */
%token OO /* || */
%token PE /* += */
%token ME /* -= */
%token TE /* *= */
%token DE /* /= */
%token MDE /* %= */
%token GGE /* >>= */
%token LLE /* <<= */
%token AE /* &= */
%token XE /* ^= */
%token OE /* |= */

%token CC /* :: */


%%

goal
```

```
    : program
    ;
a0001
    :
    | a0001    top_level_declaration
    ;
program
    : a0001
    ;
top_level_declaration
    : data_declaration
    | function_declaration
    ;
data_declaration
    : opt_decl_specifier    ';'
    | opt_decl_specifier    list_dcltr    ';'
    ;
function_declaration
    : opt_decl_specifier    dcltr    opt_base_init    compound_stmt
    ;
opt_base_init
    :
    | ':'    base_init
    ;
base_init
    : member_init
    | base_init    ','    member_init
    ;
member_init
    : '('    ')'
    | '('    list_exp    ')'
    | IDENTIFIER    '('    ')'
    | IDENTIFIER    '('    list_exp    ')'
    ;
typename_declaration
    : decl_specifier    abs_dcltr
    ;
opt_decl_specifier
    :
    | decl_specifier
    ;
decl_specifier
    : a0005
    ;
a0005
    : tc_specifier
    | a0005    tc_specifier
    ;
tc_specifier
    : std_class
    | type_specifier
    | fct_specifier
```

```
    ;
std_class
    : AUTO
    | STATIC
    | EXTERN
    | REGISTER
    | TYPEDEF
    ;
type_specifier
    : simple_type_name
    | class_specifier
    | enum_specifier
    | CONST
    ;
simple_type_name
    : TYPEDEF_NAME
    | std_type
    ;
std_type
    : CHAR
    | FLOAT
    | DOUBLE
    | INT
    | SHORT
    | LONG
    | UNSIGNED
    | VOID
    ;
fct_specifier
    : OVERLOAD
    | INLINE
    | VIRTUAL
    | FRIEND
    ;
/*****************************************************************/
/* class_specifier                                               */
/*****************************************************************/
class_specifier
    : aggr   class_tag_dcltr
    | aggr                                          '{'   a0003   '}'
    | aggr   class_tag_dcltr                        '{'   a0003   '}'
    | aggr   class_tag_dcltr   class_base_name      '{'   a0003   '}'
    ;
aggr
    : CLASS
    | STRUCT
    | UNION
    ;
class_tag_dcltr
    : IDENTIFIER
    ;
class_base_name
```

```
    : ':'   TYPEDEF_NAME
    | ':'   PUBLIC   TYPEDEF_NAME
a0003
    :
    | a0003   member_declaration
    ;
member_declaration
    : member_data_declaration
    | function_declaration
    | function_declaration   ';'
    | PUBLIC   ':'
    ;
member_data_declaration
    : opt_decl_specifier   member_dcltr   ';'   /* init_dcltr */
    ;
member_dcltr
    : dcltr
    | ':'   const_exp
    | IDENTIFIER   ':'   const_exp
    ;
/**********************************************************************/
/* enum_specifier                                                     */
/**********************************************************************/
enum_specifier
    : enum_ctype   enum_name
    | enum_ctype   a0007
    | enum_ctype   en_tag_dcltr   a0007
    ;
enum_ctype
    : ENUM
    ;
enum_name
    : IDENTIFIER
    ;
en_tag_dcltr
    : IDENTIFIER
    ;
a0007
    : '{'   a0006   '}'
    ;
a0006
    : enum_dcltr
    | a0006   ','   enum_dcltr
    ;
enum_dcltr
    : name_dcltr
    | name_dcltr   '='   const_exp
    ;
name_dcltr
    : IDENTIFIER
    ;
/**********************************************************************/
```

```
/* dcltr                                                              */
/*********************************************************************/
dname
    : simple_dname
    | /* TYPEDEF_NAME */ CC simple_dname
    ;
simple_dname
    : IDENTIFIER
    | '~'   TYPEDEF_NAME
    | operator_function_name
    | conversion_function_name
    ;
operator_function_name
    : OPERATOR   operator
    ;
operator
    : '&'
    | unary_op   /* add_op entra aqui */
    | mult_op
    | shift_op
    | rel_op
    | equ_op
    | '^'
    | '|'
    | AA
    | OO
    | assgn_op
    | '('   ')'
    | '['   ']'
    | NEW | DELETE
    ;
conversion_function_name
    : OPERATOR   TYPEDEF_NAME
    | OPERATOR   std_type
    ;
p1_dcltr
    : dname
    | '('   dcltr   ')'
    ;
p2_dcltr
    : p1_dcltr
    | p2_dcltr   '('   formals_declaration   ')'
    | p2_dcltr   '['   ']'
    | p2_dcltr   '['   list_exp   ']'
    | p2_dcltr   '('   list_exp   ')'          /* initializer */
    | p2_dcltr   '='   init_exp                /* initializer */
    ;
a0004
    : formal_declaration
    | a0004   ','   formal_declaration
    ;
formals_declaration
```

```
    :
    | a0004
    ;
formal_declaration
    : decl_specifier    dcltr
    | decl_specifier    abs_dcltr
    ;
p3_dcltr
    : p2_dcltr
    | '*'    p3_dcltr
    | '*'    CONST    p3_dcltr
    | '&'    p3_dcltr
    | '&'    CONST    p3_dcltr
    ;
dcltr
    : p3_dcltr
    ;
a0010
    : dcltr
    | a0010    ','    dcltr
    ;
list_dcltr
    : a0010
    ;
/*****************************************************************/
/* abs_dcltr                                                     */
/*****************************************************************/
p1_abs_dcltr
    : '('    p3_abs_dcltr    ')'
    ;
p2_abs_dcltr
    : p1_abs_dcltr
    | p2_abs_dcltr    '('    ')'
    | '('    ')'
    | p2_abs_dcltr    '['    ']'
    | p2_abs_dcltr    '['    list_exp    ']'
    | e_abs_dcltr    '['    ']'
    | e_abs_dcltr    '['    list_exp    ']'
    ;
p3_abs_dcltr
    : p2_abs_dcltr
    | '*'    p3_abs_dcltr
    | '*'    e_abs_dcltr
    ;
e_abs_dcltr
    :
    ;
abs_dcltr
    : p3_abs_dcltr
    | e_abs_dcltr
    ;
/*****************************************************************/
```

```
/* stmt                                                              */
/*********************************************************************/
a0013
    :
    | a0013   decl_or_stmt
    ;
compound_stmt
    : '{'   a0013   '}'
    ;
decl_or_stmt
    : stmt
    ;
local_data_declaration
    : decl_specifier   ';'
    | decl_specifier   list_dcltr   ';'
    ;
basic_stmt
    : local_data_declaration
    | e_stmt
    | compound_stmt
    | do_stmt
    | break_stmt
    | continue_stmt
    | return_stmt
    | goto_stmt
    | null_stmt
    ;
stmt
    : basic_stmt
    | while_stmt
    | for_stmt
    | ifelse_stmt
    | if_stmt
    | switch_stmt
    | label   stmt
    ;
ifelse_stmt
    : IF   '('   list_exp   ')'   stmt   ELSE   stmt
    ;
if_stmt
    : IF   '('   list_exp   ')'   stmt
    ;
e_stmt
    : list_exp   ';'
    ;
while_stmt
    : WHILE   '('   list_exp   ')'   stmt
    ;
do_stmt
    : DO   stmt   WHILE   '('   list_exp   ')'   ';'
    ;
a0014
```

```
    : ';'   ')'   stmt
    | ';'   list_exp   ')'   stmt
    ;
a0015
    : ';'   a0014
    | ';'   list_exp   a0014
    ;
for_stmt
    : FOR   '('   a0015
    | FOR   '('   list_exp   a0015
    ;
switch_stmt
    : SWITCH   '('   list_exp   ')'   stmt
    ;
break_stmt
    : BREAK ';'
    ;
continue_stmt
    : CONTINUE ';'
    ;
return_stmt
    : RETURN ';'
    | RETURN list_exp ';'
    ;
goto_stmt
    : GOTO label_name   ';'
    ;
label_name
    : IDENTIFIER
    ;
null_stmt
    : ';'
    ;
label
    : name_label
    | case_label
    | default_label
    ;
name_label
    : IDENTIFIER ':'
    ;
case_label
    : CASE exp ':'
    ;
default_label
    : DEFAULT ':'
    ;
/*****************************************************************/
/* exp                                                           */
/*****************************************************************/
literal
     DEC_INT
```

```
    | OCT_INT
    | HEX_INT
    | FLOATLIT
    | CHARLIT
    | STRING
    ;
id_name
    : IDENTIFIER
    | operator_function_name
    | TYPEDEF_NAME   CC   IDENTIFIER
    | TYPEDEF_NAME   CC   operator_function_name
    ;
field_name
    : IDENTIFIER
    ;
paren_exp
    : '('   list_exp   ')'
    ;
primary_p1_exp
    : id_name
    | literal
    | paren_exp
    | THIS
    | CC   IDENTIFIER
    ;
primary_p2_exp
    : primary_p1_exp
    | primary_p2_exp   '['   list_exp   ']'
    | primary_p2_exp   '('   ')'
    | primary_p2_exp   '('   list_exp   ')'
    | primary_p2_exp   '.'   field_name
    | primary_p2_exp   MG   field_name
    ;
primary_exp
    : primary_p2_exp
    ;
postfix_exp
    : primary_exp
    | postfix_exp   postfix_op
    ;
postfix_op
    : PP
    | MM
    ;
prefix_exp
    : postfix_exp
    | SIZEOF   prefix_exp
    | prefix_op   cast_exp
    | '*'   cast_exp
    | '&'   cast_exp
    | unary_op   cast_exp
    ;
```

```
prefix_op
    : PP
    | MM
    ;
unary_op
    : '+'
    | '-'
    | '!'
    | '~'
    ;
cast_exp
    : prefix_exp
    | '('   typename_declaration   ')'   cast_exp
    ;
cpp_exp
    : cast_exp
    | SIZEOF   '('   typename_declaration   ')'
    | simple_type_name   '('   list_exp   ')'  /* cast funcional */
    ;
unary_oper_exp
    : cpp_exp
    ;
mult_oper_exp
    : unary_oper_exp
    | mult_oper_exp   mult_op   unary_oper_exp
    ;
mult_op
    : '*'
    | '/'
    | '%'
    ;
add_oper_exp
    : mult_oper_exp
    | add_oper_exp   add_op   mult_oper_exp
    ;
add_op
    : '+'
    | '-'
    ;
shift_oper_exp
    : add_oper_exp
    | shift_oper_exp   shift_op   add_oper_exp
    ;
shift_op
    : LL
    | GG
    ;
rel_oper_exp
    : shift_oper_exp
    | rel_oper_exp   rel_op   shift_oper_exp
    ;
rel_op
```

```
    : '<'
    | LE
    | GE
    | '>'
    ;
equ_oper_exp
    : rel_oper_exp
    | equ_oper_exp    equ_op    rel_oper_exp
    ;
equ_op
    : EE
    | NE
    ;
bitand_oper_exp
    : equ_oper_exp
    | bitand_oper_exp    '&'    equ_oper_exp
    ;
bitxor_oper_exp
    : bitand_oper_exp
    | bitxor_oper_exp    '^'    bitand_oper_exp
    ;
bitor_oper_exp
    : bitxor_oper_exp
    | bitor_oper_exp    '|'    bitxor_oper_exp
    ;
and_oper_exp
    : bitor_oper_exp
    | and_oper_exp    AA    bitor_oper_exp
    ;
or_oper_exp
    : and_oper_exp
    | or_oper_exp    OO    and_oper_exp
    ;
cond_exp
    : or_oper_exp
    | or_oper_exp    '?'    list_exp    ':'    cond_exp
    ;
assign_exp
    : cond_exp
    | cond_exp    assgn_op    exp
    ;
assgn_op
    : '='
    | PE
    | ME
    | TE
    | DE
    | MDE
    | GGE
    | LLE
    | AE
    | XE
```

```
    : OE
    ;
exp
    : assign_exp
    | NEW    '('    typename_declaration    ')'
    | NEW    TYPEDEF_NAME
    | NEW    TYPEDEF_NAME    '('    list_exp    ')'    /* TYPEDEF_NAME */
                                                        /* dev. void* */
    | DELETE    prefix_exp
    | DELETE    '['    const_exp    ']'    prefix_exp    /* dev. void* */
    ;
list_exp
    : exp
    | list_exp    ','    exp
    ;
const_exp
    : exp
    ;
init_exp
    : exp
    | '{'    a0009    '}'
    | '{'    a0009    ','    '}'    '
    ;
a0009
    : init_exp
    | a0009    ','    init_exp
    ;
```

APENDICE B. BIBLIOGRAFIA.

[YOUR86] Yourdon, E.

"What ever happened to Structured Analysis"

Datamation, pp 133-138, june 1986.

[KAY85] Kaylan, D.

"Modular Programming in C: An Approach and an example"

SigPlan Notices, Vol 20, No 3, pp 9-15, Marzo 1985.

[BERG88] Bergin, Joseph. Greenfield, Stuart

"What does Modula-2 need to fully support Object Oriented Programming"

SigPlan Notices, Vol 23, No 3, pp 77-82, Marzo 1988

[TORO87] Toro C, Victor Manuel

"Diseño, Especificación y Prototificación de Sistemas Interactivos"

Publicación Centro de Documentación CIFI, Oct. 1987.

[KER78] Kernighan, B. W. Ritchie, D. M.

"The C Programming Language"

Prentice Hall Inc., Englewood Cliffs, 1978.

[KER84] Kernighan, B. W. Pike, Rob

"The UNIX Programming Environment"
Prentice Hall Inc., Englewood Cliffs, 1984.

[HARB84] Harbison, Samuel P. Steele, Guy L.
"C a reference manual"
Prentice Hall Inc., Englewood Cliffs, 1984

[SHRE87] Shreiner, Axel T. Friedman, George H.
"Introduccion to Compiler Construction con UNIX"
Prentice Hall Inc., Englewood Cliffs, 1987.

[STRO] Stroustroup, Bjarne.
"The C++ Programming Language"
Addison Wesley Publishing Co.

[RITCH] Ritchie, D.M.
"A tour through the UNIX C Compiler."
Bell Labs, Murray Hills, New Jersey.

[JOHN] Johnson, S.C.
"A tour through the Portable C Compiler"
Bell Labs, Murray Hills, New Jersey.

[DEW87] Dewhurst, Stephen C.
"Flexible Symbol Table for compiling C++.

Software, Practice & Experience, Vol 17, pp 503-513, Aug 87

[IBM81] IBM Canada

"Information Systems: "Project Management Guide"

IBM Canada, August 1981.

[TREM81] Tremblay, Jean-Paul Sorenson, Paul G.

"The theory and Practice of Compiler Writing"

Mc Graw Hill Inc...1981.

El Jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional, aprobó esta tesis el día 3 de abril de 1989.

Dr. JOSEF KOLAR SAVOR

Dr. JAN JANECEK HYAN

M. en C. JOSE OSCAR OLMEDO AGUIRRE.