

UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR



INGENIERÍA EN INFORMÁTICA

CURSO 4º

PROCESADORES DE LENGUAJE II

Compilador C-Lite a MIPS R3000

Moisés Martínez Muñoz

100033605

Ignacio Álvarez Santiago

100044392

Índice de Contenidos

1 INTRODUCCIÓN	8
1.1 INTRODUCCIÓN AL DOCUMENTO	8
1.2 OBJETIVOS DEL DOCUMENTO	8
1.3 ALCANCE.....	9
1.4 BIBLIOGRAFÍA.....	9
2 DESCRIPCIÓN DEL COMPILADOR	10
2.1 DESCRIPCIÓN DE LOS MÓDULOS DEL COMPILADOR.....	10
3 DESCRIPCIÓN DEL PROCESADOR	13
3.1 INTRODUCCIÓN.....	13
3.2 CARACTERÍSTICAS DEL PROCESADOR.....	14
2.3 ARQUITECTURA DEL PROCESADOR	15
3.4 SEGMENTACIÓN.....	18
2.5 INSTRUCCIONES	21
3.6 INSTRUCCIONES ARITMÉTICAS O LÓGICAS	22
2.7 SERVICIOS OFRECIDOS POR EL SISTEMA	28
3.8 DIRECCIONAMIENTO.....	29
3.9 DIRECTIVAS.....	32
3.10 TIPOS DE DATOS	32
4 ANÁLISIS LÉXICO.....	34
4.1 DESCRIPCIÓN	34
4.2 LISTA DE TOKENS.....	34
4.3 DECISIONES DE DISEÑO.....	35
4.3.1 Decisiones generales	35
4.3.2 Comunicación con analizador sintáctico	36
4.3.3 Control de errores	37
4.4 PROBLEMAS ENCONTRADOS	37
5 ANÁLISIS SINTÁCTICO	42
5.1 DESCRIPCIÓN	42
5.2 DECISIONES DE DISEÑO.....	42
5.2.1 Pila de comunicaciones	42
5.2.2 Proceso de comunicación entre producciones	42
5.2.3 Definición de variables	46
5.2.3 Control de errores	47

5.3 PROBLEMAS ENCONTRADOS	47
6 ANÁLISIS SEMÁNTICO.....	50
6.1 DESCRIPCIÓN	50
6.2 DECISIONES DE DISEÑO.....	51
6.2.1 TABLA DE SÍMBOLOS	51
6.3 PROBLEMAS ENCONTRADOS	53
6.4 CONTROL DE ERRORES.....	54
7 GENERACIÓN DE CÓDIGO INTERMEDIO.....	57
7.1 GENERACIÓN DE EXPRESIONES	57
7.1.1 Transformación a forma polaca inversa	58
7.1.2 Algoritmo RPN para generación de expresiones postfijas	58
7.1.3 Ventajas de la forma polaca inversa	60
7.1.4 Creación de árboles binarios	60
7.1.5 Evaluación de árboles binarios de expresiones	62
7.1.6 Tratamiento de árboles de expresión.....	65
7.2 OPTIMIZACIÓN DEL USO DE REGISTROS	70
7.2.1 Algoritmo LRU	72
7.3 TABLA DE REGISTROS	73
7.4 TRATAMIENTO DE SENTENCIAS ANIDADAS	75
7.4.1 Algoritmo de profundidad.....	77
7.5 TRATAMIENTO DE LA PILA	79
7.6 TRATAMIENTO DE FUNCIONES	80
7.7 TRATAMIENTO DE LOS ARRAYS	82
8 GENERACIÓN DE CÓDIGO ENSAMBLADOR	85
8.1 DECISIONES DE DISEÑO.....	85
8.1.1 Tratamiento de registros.....	85
8.1.2 Cabecera del programa.....	86
8.1.3 Generación de código de variables	86
8.1.4 Generación de instrucciones de	87
8.1.5 Generación de instrucciones de almacenamiento	88
8.1.6 Generación de instrucciones de operación	88
8.1.7 Generación instrucciones para arrays.....	89
8.1.8 Generación de instrucciones de salto.....	89
8.1.9 Generación de etiquetas de salto.....	90
8.1.10 Generación de instrucciones de funciones	91
8.1.11 Entrada/Salida	92

9 OTROS ELEMENTOS	93
9.1 DESCRIPCIÓN DEL SISTEMA DE EJECUCIÓN	93
9.2 HERRAMIENTAS UTILIZADAS	94
9.2.1 <i>Ubuntu</i>	94
9.2.2 <i>GCC</i>	94
9.2.4 <i>Bison</i>	95
9.2.5 <i>Flex</i>	95
9.2.6 <i>Gedit</i>	96
9.2.7 <i>Office 2007</i>	96
9.2.8 <i>MARS</i>	97
10 PRUEBA REALIZADAS	97
10.1 PRUEBAS DE FUNCIONAMIENTO	97
<i>Prueba 1: Año Bisiesto</i>	98
<i>Prueba 2: Calculadora</i>	98
<i>Prueba 3: Calculador de Matrices</i>	99
<i>Prueba 4: Factorial Iterativo</i>	100
<i>Prueba 5:Factorial Recursivo</i>	100
<i>Prueba 6: Media Aritmética</i>	102
<i>Prueba 7: Raíz cuadrada</i>	102
<i>Prueba 8: Torres de Hanoi</i>	103
10.2 PRUEBA DE ERROR	104
11 PROGRAMACIÓN EN C-LITE	105
12 CONTENIDO DEL DVD	108
13 CONCLUSIONES	110
14 ANEXOS	113
14.1 CÓDIGO FUENTE ANALIZADOR LÉXICO.....	113
14.2 CÓDIGO FUENTE ANALIZADOR SINTÁCTICO	117

Índice de Tablas

TABLA 1 - DESCRIPCIÓN DE LOS MÓDULOS	12
TABLA 2 - REGISTROS DEL MIPS R3000	18
TABLA 3 - ELEMENTOS DE LAS INSTRUCCIONES	22
TABLA 4 - INSTRUCCIONES ARITMÉTICAS Y LÓGICAS	23
TABLA 5 - INSTRUCCIONES MANIPULACIÓN DE CONSTANTES.....	23
TABLA 6 - INSTRUCCIONES DE COMPARACIÓN	24
TABLA 7 - INSTRUCCIONES DE BIFURCACIÓN Y SALTO	25
TABLA 8 - INSTRUCCIONES DE CARGA	26
TABLA 9 - INSTRUCCIONES DE EXCEPCIÓN Y TRAP	26
TABLA 10 - INSTRUCCIONES DE ALMACENAMIENTO	27
TABLA 11 - INSTRUCCIONES DE MOVIMIENTO DE DATOS.....	27
TABLA 12 - INSTRUCCIONES DEL COPROCESADOR DE COMA FLOTANTE	28
TABLA 13 - SERVICIOS DEL SISTEMA	29
TABLA 14 - TIPOS DE DATOS	33
TABLA 15 - LISTA DE TOKENS ANALIZADOR LÉXICO.....	35
TABLA 16 - TABLA DE ERRORES SEMÁNTICOS	56
TABLA 17 - CORRESPONDENCIA DE TIPOS ENTRE R3000 Y MIPS.....	87
TABLA 18 - INSTRUCCIONES DE CARGA SOPORTADAS POR EL COMPILADOR R3000.....	87
TABLA 19 - INSTRUCCIONES DE ALMACENAMIENTO SOPORTADAS POR EL COMPILADOR R3000	88
TABLA 20 - INSTRUCCIONES DE OPERACIÓN SOPORTADAS POR EL COMPILADOR R3000	89
TABLA 21 - INSTRUCCIONES DE BIFURCACIÓN SOPORTADAS POR EL COMPILADOR R3000	90
TABLA 22 - DESCRIPCIÓN DE LA PRUEBA 1.....	98
TABLA 23 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 1	98
TABLA 24 - DESCRIPCIÓN DE LA PRUEBA 2.....	98
TABLA 25 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 2	99
TABLA 26 - DESCRIPCIÓN DE LA PRUEBA 3.....	99
TABLA 27 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 3	99
TABLA 28 - DESCRIPCIÓN DE LA PRUEBA 4.....	100
TABLA 29 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 4	100
TABLA 30 - DESCRIPCIÓN DE LA PRUEBA 5.....	101
TABLA 31 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 5	101
TABLA 32 - DESCRIPCIÓN DE LA PRUEBA 6.....	102
TABLA 33 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 6	102
TABLA 34 - DESCRIPCIÓN DE LA PRUEBA 7.....	102
TABLA 35 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 7	103
TABLA 36 - DESCRIPCIÓN DE LA PRUEBA 8.....	103

TABLA 37 - EXPERIMENTOS REALIZADOS SOBRE LA PRUEBA 8	104
TABLA 38 - EXPERIMENTO REALIZADOS CON ERRORES SEMÁNTICOS.....	104

Índice de Ilustraciones

ILUSTRACIÓN 1- DIAGRAMA DE UTILIZACIÓN DE MODULOS	10
ILUSTRACIÓN 2 - ANVERSO Y REVERSO DE UN MIPS R3010	14
ILUSTRACIÓN 3 - REPRESENTACIÓN DE LOS DATOS BIG-ENDIAN.....	15
ILUSTRACIÓN 4 - ESTRUCTURA BÁSICA DEL MIPS R3000.....	15
ILUSTRACIÓN 5 - MAPA DE MEMORIA DEL MIPS R3000	16
ILUSTRACIÓN 6 - CAUCE SEGMENTADO DE INSTRUCCIONES	19
ILUSTRACIÓN 7 - HARDWARE SEGMENTADO	20
ILUSTRACIÓN 8 - INSTRUCCIÓN TIPO R	21
ILUSTRACIÓN 9 - INSTRUCCIÓN TIPO I	21
ILUSTRACIÓN 10 - INSTRUCCIONES TIPO J.....	22
ILUSTRACIÓN 11 - DIRECCIONAMIENTO INMEDIATO	29
ILUSTRACIÓN 12 - DIRECCIONAMIENTO A REGISTRO	30
ILUSTRACIÓN 13 - DIRECCIONAMIENTO BASE MÁS DESPLAZAMIENTO.....	30
ILUSTRACIÓN 14 - DIRECCIONAMIENTO RELATIVO A PC	31
ILUSTRACIÓN 15 - DIRECCIONAMIENTO PSEUDODIRECTO.....	31
ILUSTRACIÓN 16- REPRESENTACIÓN DE LOS DATOS	33
ILUSTRACIÓN 17 - EXPRESIONES REGULARES NÚMEROS.....	38
ILUSTRACIÓN 18 - CONDICIÓN DE PARADA COMENTARIO //.....	39
ILUSTRACIÓN 19 - CONDICIÓN DE PARADA COMENTARIO /*	39
ILUSTRACIÓN 20 - EXPRESIONES REGULARES PARA TRATAMIENTO DE TEXTOS Y CARACTERES.....	40
ILUSTRACIÓN 21 - ESTRUCTURAS DE COMUNICACIÓN ENTRE PROCESOS.....	43
ILUSTRACIÓN 22 - REGLAS SOBRE LAS QUE SE HAN APLICADO TIPOS DE DATOS	46
ILUSTRACIÓN 23 - FUNCIÓN DE ERROR DEL ANALIZADOR SINTÁCTICO	47
ILUSTRACIÓN 24 - APLICACIÓN DE PREFERENCIAS EN SENTENCIA_IF.....	48
ILUSTRACIÓN 25 - COMUNICACIÓN DE VALORES ENTRE PRODUCCIONES	49
ILUSTRACIÓN 26 - ESTRUCTURA DE LA TABLA DE SÍMBOLOS	52
ILUSTRACIÓN 27 - PROCESO DE GENERACIÓN DE UN ÁRBOL BINARIO DE OPERACIONES	62
ILUSTRACIÓN 28 - PROCESO DE TRATAMIENTO DEL ÁRBOL DE EXPRESIONES	64
ILUSTRACIÓN 29 - ÁRBOL DE EXPRESIONES PREVIO AL PROCESO DE ETIQUETADO	68
ILUSTRACIÓN 30 - ÁRBOL DE EXPRESIONES ETIQUETADO	68
ILUSTRACIÓN 31 - ESTRUCTURA DE LA TABLA DE REGISTROS.....	74
ILUSTRACIÓN 32 - ESTRUCTURA DE UN REGISTRO DE LA TABLA DE REGISTROS	75
ILUSTRACIÓN 33 - SISTEMA DE EJECUCIÓN DEL COMPILADOR	93

1 Introducción

1.1 Introducción al documento

Es este documento se incluye toda la información referente al proceso de desarrollo de un compilador de lenguaje C LITE a lenguaje ensamblador MIPS R3000. El documento se encuentra dividido en 14 puntos, que son los siguientes:

- Introducción
- Descripción del compilador
- Descripción del procesador
- Análisis léxico
- Análisis sintáctico
- Análisis semántico
- Generación de código intermedio
- Generación de código objeto
- Otros elementos
- Pruebas realizadas
- Programación en C-Lite
- Contenido del DVD
- Conclusiones
- Anexos

1.2 Objetivos del documento

El objetivo principal de este documento es la descripción detallado del proceso de construcción del compilador, indicándose para cada una de las etapas del proceso de compilación, como se ha realizado, que errores han sido encontrados y de que forma han sido resueltos.

Otro objetivo de este documento es indicar el conjunto de pruebas a las que ha sido sometida la aplicación a largo de su desarrollo y al final de este, de forma que se demuestre que cumple todas las funcionalidades descritas en el documento de

descripción del proyecto y que realiza de forma correcta todos los procesos para los que ha sido desarrollado.

Además en este documento también se indican aquellas herramientas que han sido utilizadas a lo largo del desarrollo del proyecto, indicándose su utilidad de forma que puedan servir a personas que vayan a emprender desarrollos similares al descrito en este documento.

1.3 Alcance

Este documento va dirigido principalmente a los profesores de la asignatura de Procesadores de Lenguajes II, a modo que puedan comprobar el proceso que se ha seguido a lo largo del desarrollo del compilador y observar como se han resuelto los problemas encontrados en dicho desarrollo y que herramientas se han utilizado.

1.4 Bibliografía

- [1] Bison 1.27. Charles Donnelly y Richard Stallman.
- [2] Flex 2.5. Vern Paxon.
- [3] Yacc and Lex. John R. Levine, Tony Mason & Doug Brown.
- [4] Compiladores Principios, técnicas y herramientas. Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman.
- [5] Construcción de Compiladores Principios y práctica. Kenneth C. Louden.
- [6] Estructura de datos Algoritmos, abstracción y objetos. Luis Joyanes Aguilar e Ignacio Zahonero Martínez.
- [7] Sistemas Operativos Una visión aplicada. Jesús Carretero Pérez, Félix García Carballería, Pedro de Miguel Anasagasti y Fernando Pérez ostota.
- [8] MIPS32® Architecture For Programmers Volume I.
- [9] See MIPS run. Dominic Sweetman
- [10] MIPS Assembly Language Programmer's Guide.
- [11] MIPS Assembly Language Programming. Robert L. Britton.

2 Descripción del compilador

2.1 Descripción de los módulos del compilador

A continuación se presenta una descripción de los distintos módulos que han sido desarrollados para el compilador R3000 y que fase del proceso de compilación ha intervenido cada uno de ellos.

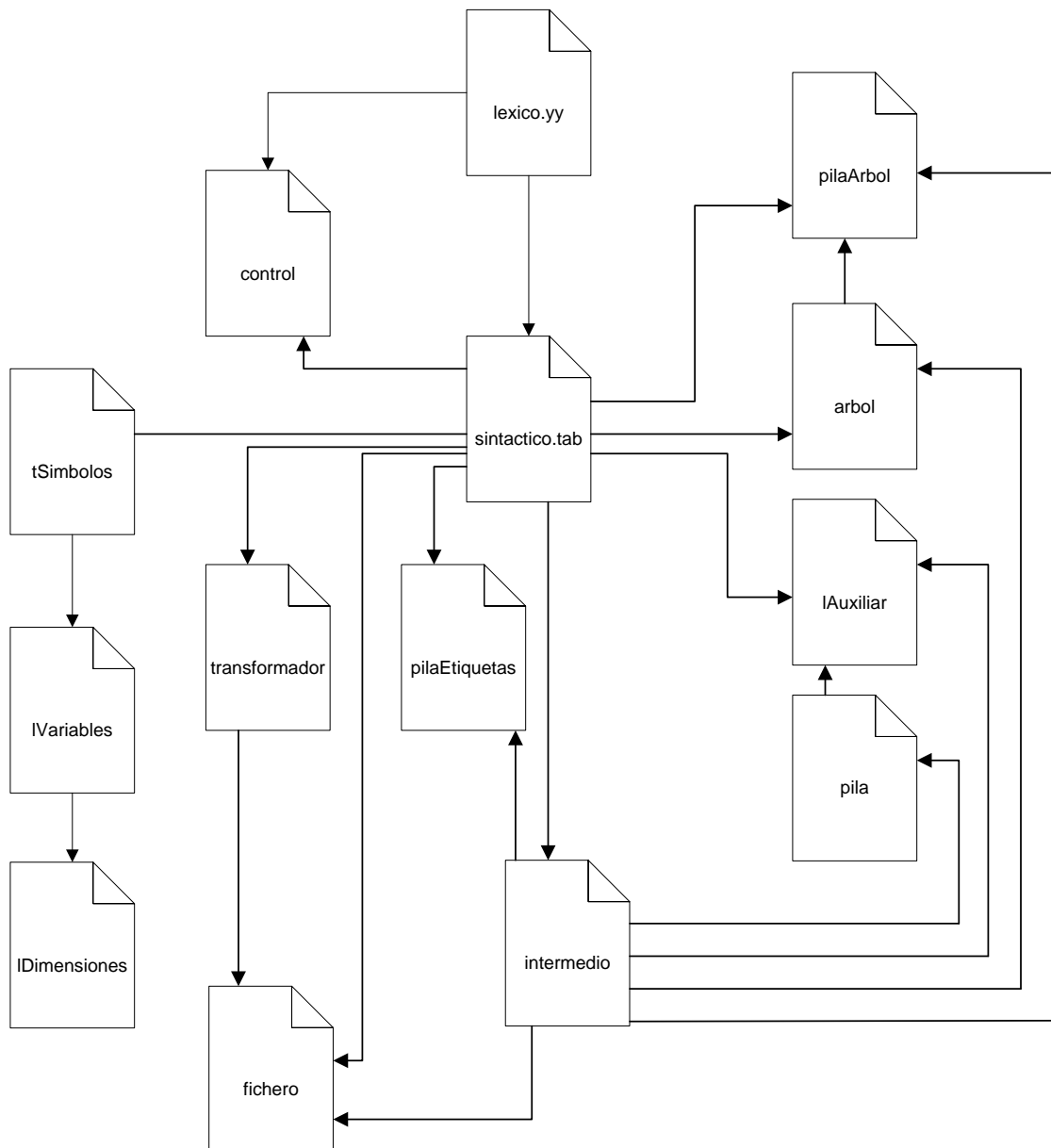


Ilustración 1- Diagrama de utilización de módulos

A continuación se presenta una tabla en la que se indica una breve descripción de la utilidad de cada uno de los módulos que forman el compilador.

Módulos	Descripción
lexico.yy	En este módulo se almacena el código generado por el analizador léxico lex y que ejecuta el analizado léxico construido.
sintactico.tab	Este es principal módulo del sistema ya que en el se encuentra el método main del compilador, es generado por el analizador sintáctico bison , además en el se encuentran todas la llamadas a los métodos de los proceso de análisis semántico, generación de código intermedio y generación de código ensamblador.
control	En ese módulo se almacena la funciones auxiliares utilizadas a lo largo del compilador así como los distintos proceso de control del sistema mediante una estructura formado por un conjunto de flags.
tSimbolos	En este módulo está construido el primer elemento de la tabla de símbolos, en el cual se realiza el almacenamiento y tratamiento de las funciones y métodos definidos en el archivo de entrada del compilador.
IVariables	En este módulo está construida la lista de variables que posee cada símbolo, cada elemento de la estructura tSimbolos está formado por una lista de variables, considerándose como variables las constantes, variables y parámetros.
IDimensiones	En este módulo está construida la lista de dimensiones de una variable de tipo array, las variable de la lista de variables de un método o función poseerá una lista de dimensiones solo en el caso de que sea de tipo Array.
arbol	En este módulo está construido un árbol binario, el cual es utilizado en la generación de los códigos intermedios de las sentencias condicionales y operacionales.
pilaArbol	Es este módulo está construida un pila que almacena nodos del árbol binario, es utilizada en el proceso de creación del árbol binario a partir de una expresión en forma polaca inversa.
IAuxiliar	En este módulo está implementado una lista auxiliar para almacenamiento de tokens, es utilizada en mucho momento desde

Módulos	Descripción
	lista de almacenamiento de dimensiones, hasta la lista el almacenamiento de las expresiones en forma normal o en forma polaca inversa.
pila	Es este módulo está construida un pila que almacena nodos de lista auxiliar, fue construida para el proceso de transformación de una expresión a forma polaca inversa.
pilaEtiqueta	Es este módulo está construida un pila que almacena las etiquetas para los procesos de los bucles y las sentencias condicionales.
transformador	En este módulo está construida la estructura de optimización de registros, así como los distintos métodos de generación del código ensamblador.
intermedio	Es este módulo está construida las distintas funciones que realizan el tratamiento de expresión(asignaciones, expresiones condicionales, bucles, etc) y la generación del código intermedio que es utilizada para la posterior generación del código ensamblador.
fichero	Es este módulo está construida un conjunto de funciones para el tratamiento de fichero, en este caso para la manipulación del fichero de salida con el código ensamblador generado por el compilador R3000.

Tabla 1 - Descripción de los módulos

3 Descripción del procesador

3.1 Introducción

El procesador elegido como objetivo para la compilación es el MIPS R3000. Este es un procesador de 32 bits (por tanto su tamaño de palabra es de 4 bytes) basado en el concepto de computadora RISC, lo que supone que dispone de un número muy reducido de instrucciones muy simples que se ejecutan de forma muy sencilla.

Este modelo de procesador apareció en 1988 y funciona con números de tipo entero, pero la versión real para la que se compilará el lenguaje C será para la versión R3010, una versión modificada del procesador que dispone de un coprocesador para el tratamiento de números en coma flotante (basados en el estándar del IEEE).

Los principales motivos por los que se ha escogido este procesador es por su simplicidad de manejo, por disponer de un número de instrucciones lo suficientemente importante como para no ser engorrosa su programación, por disponer de la posibilidad de manejar muchos tipos de datos, no sólo palabras de 4 bytes y además por permitir el uso de números reales (coma flotante).

El MIPS R300 es un procesador usado generalmente como big-endian lo que supone que los bytes se almacenan pro orden desde el menos significativo hasta el más significativo, de una forma similar a la representación numérica pro parte los seres humanos.

Aunque el procesador tiene ya 20 años y su juego de instrucciones (MIPS I) es antiguo el procesador no se diferencia mucho de otros más modernos por tratarse de un procesador de 32 bits y permitir el cauce segmentado en las ejecuciones de instrucciones. De hecho otras versiones más modernas de MIPS usadas en la actualidad sólo han introducido algunas instrucciones para propósitos concretos como son de control de registros, de cómputo 3D o de manejo de hilos.

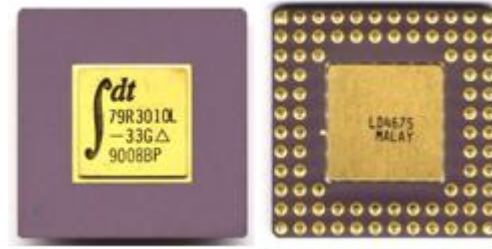


Ilustración 2 - Anverso y reverso de un MIPS R3010

El uso de los procesadores MIPS es muy extendido, aunque nunca se orientó a PCs o grandes equipos. En general están destinados a ser usados en sistemas empujados de todo tipo, sistemas en los cuales la inmensa mayoría de los procesadores son MIPS.

Han sido usados también en videoconsolas y es especialmente destacable su uso en los routers CISCO.

3.2 Características del procesador

El MIPS R3000 es un procesador RISC de 32 bits basado en una arquitectura Von Neuman. El procesador dispone de un reloj de 33 MHz y tiene de dos caches de 32 KB, una para instrucciones y otra para datos.

El procesador dispone de un banco de registros, con un total de 32, siendo a su vez de 32 bits cada uno, tiene además un UCP (núcleo y decodificador de instrucciones del procesador) llamado coprocesador 0. El MIPS R3000 dispone de una ALU para la ejecución de operaciones con enteros. Aunque dispone de la posibilidad de tener varios coprocesadores más, lo habitual es que disponga de un coprocesador con una ALU para el tratamiento de números reales y un banco de 32 registros en formato de coma flotante.

El procesador además se compone de un PC (contador de programa) y una memoria direccionable de hasta 4 GB.

El procesador MIPS R3000 podía ser configurado para trabajar como Big-endian o Little-endian, aunque la forma habitual de trabajar (y la forma en la que trabajan los simuladores) es Big-endian. A continuación se muestra la codificación de una palabra en formato Big-endian:

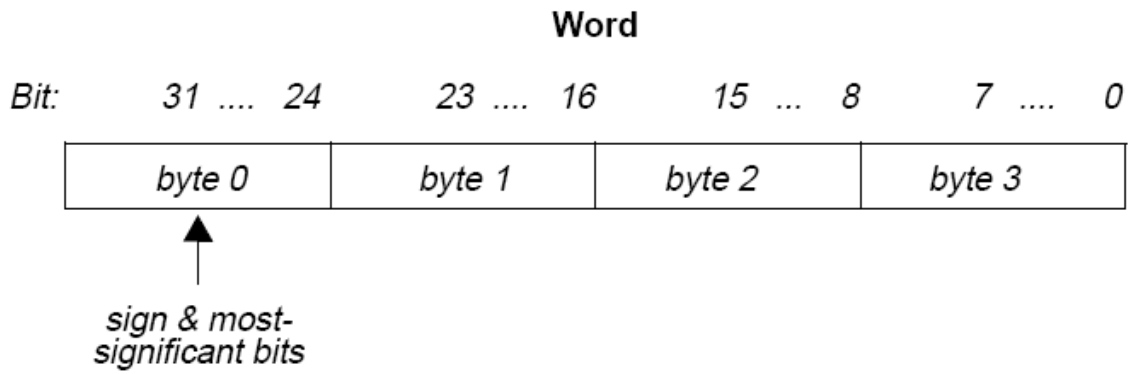


Ilustración 3 - Representación de los datos Big-endian

2.3 Arquitectura del procesador

El siguiente diagrama muestra la arquitectura básica del procesador MIPS orientada a la estructura básica de Von Neuman.

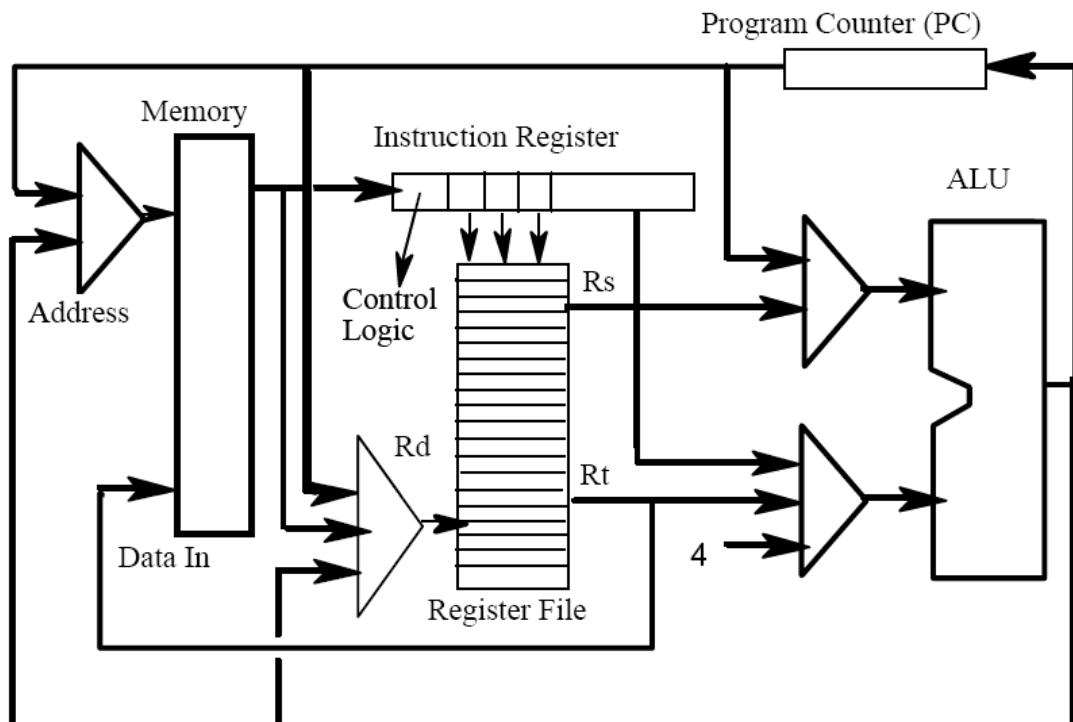


Ilustración 4 - Estructura básica del MIPS R3000

En él se aprecia que las instrucciones son leídas tal cual marca el PC (contador del programa), que una vez son leídas de memoria se almacenan en un registro de instrucciones donde son decodificadas. Posteriormente a su decodificación se usan los registros necesarios (del banco de registros) y se opera con ellos en la ALU. Los

cálculos pueden ser usados para ir nuevamente a registros, a posiciones de memoria o usarse para actualizar el contador del programa (en el caso de los saltos).

La memoria, como ya se ha dicho, tiene un tamaño máximo de 4 GB y está dividida en bloques accesibles de 4 bytes cada uno (32 bits). En la memoria se almacena el segmento de datos del programa, que son todas las variables definidas en él. Se almacena también el segmento de pila, que es una región de tamaño variable donde el programa almacenará sus datos parciales internos y finalmente contendrá el segmento de código del programa, que será aquel que contenga las instrucciones en sí del programa. La organización de la memoria por parte del procesador (en un máximo de 4GB) es la siguiente:

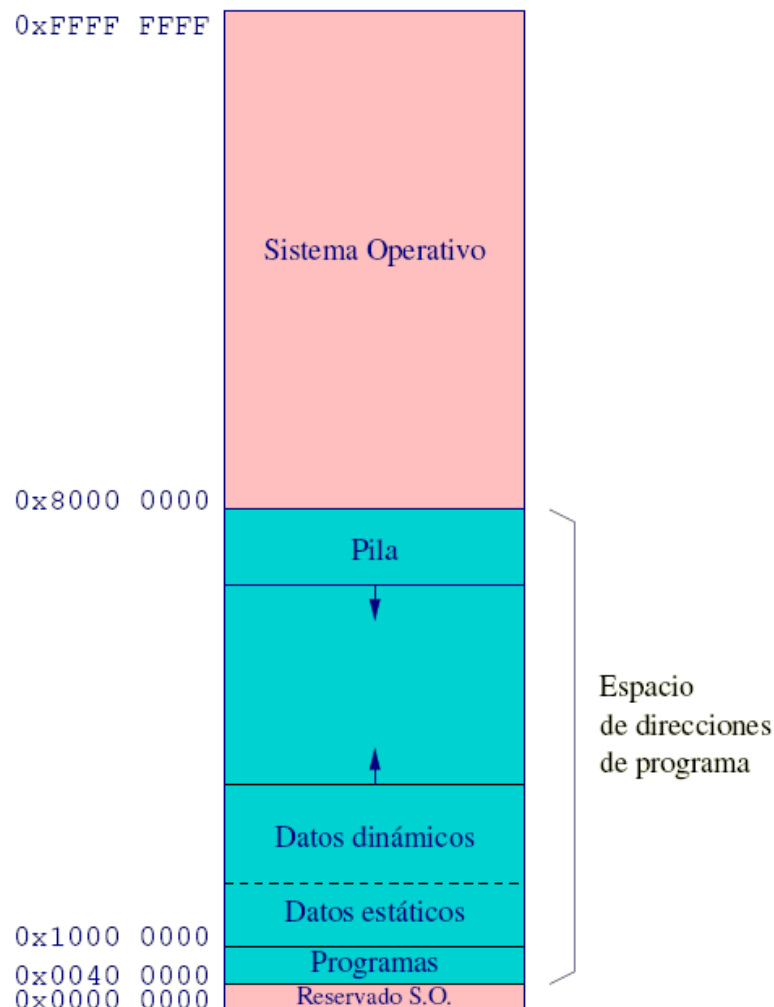


Ilustración 5 - Mapa de memoria del MIPS R3000

El banco de registros se compone de 32 registros generales de 32 bits que se definen a continuación:

Número	Nombre	Uso
\$0	\$zero	Constante 0 del ensamblador
\$1	\$at	Uso propio para el traductor de pseudoinstrucciones
\$2	\$v0	Valor de retorno de una subrutina
\$3	\$v1	
\$4	\$a0	Argumentos de una subrutina
\$5	\$a1	
\$6	\$a2	
\$7	\$a3	
\$8	\$t0	Registros temporales que no se conservan entre llamadas
\$9	\$t1	
\$10	\$t2	
\$11	\$t3	
\$12	\$t4	
\$13	\$t5	
\$14	\$t6	
\$15	\$t7	Registros temporales que se conservan entre llamadas
\$16	\$s0	
\$17	\$s1	
\$18	\$s2	
\$19	\$s3	
\$20	\$s4	
\$21	\$s5	
\$22	\$s6	
\$23	\$s7	
\$24	\$t8	Registros temporales que no se conservan entre llamadas
\$25	\$t9	
\$26	\$k0	Registros para el uso del sistema operativo (kernel)

Número	Nombre	Uso
\$27	\$k1	
\$28	\$gp	Puntero a zona de variables estáticas
\$29	\$sp	Puntero a la pila
\$30	\$fp	Puntero de marco
\$31	\$ra	Dirección de retorno de una subrutina

Tabla 2 - Registros del MIPS R3000

Además de esta estructura expresada del procesador principal de MIPS R3000, éste también disponía de un coprocesador para operaciones con número reales. El coprocesador disponía de un juego de instrucciones para la carga, almacenamiento y operaciones con este tipo de números y un banco de registros de un total de 32 (del \$f0 al \$f31) de 32 bits cada uno para la representación de números en coma flotante de precisión simple. Además estos registros podían usarse dos a dos conjuntamente obteniendo 16 registros de 64 bits (comenzando en las posiciones pares) que podían usarse para operaciones de coma flotante de doble precisión.

3.4 Segmentación

Como ya se mencionó, el MIPS R3000 es un procesador superescalar (o segmentado). Esto significa que ejecuta más de una instrucción por ciclo de reloj. Esto supone que la velocidad de ejecución de los programas se acelera muchísimo, ya que mientras se ejecuta una parte de una instrucción en una sección hardware del procesador, se puede ejecutar en otra parte del procesador otra sección de una instrucción diferente.

Las instrucciones del MIPS R3000 se componen de cinco segmentos (esto no significa que duren 5 ciclos), por lo que en teoría, se podrían ejecutar cinco instrucciones simultáneamente. Cada una de esas cinco etapas tiene una finalidad diferente que se definirá a continuación:

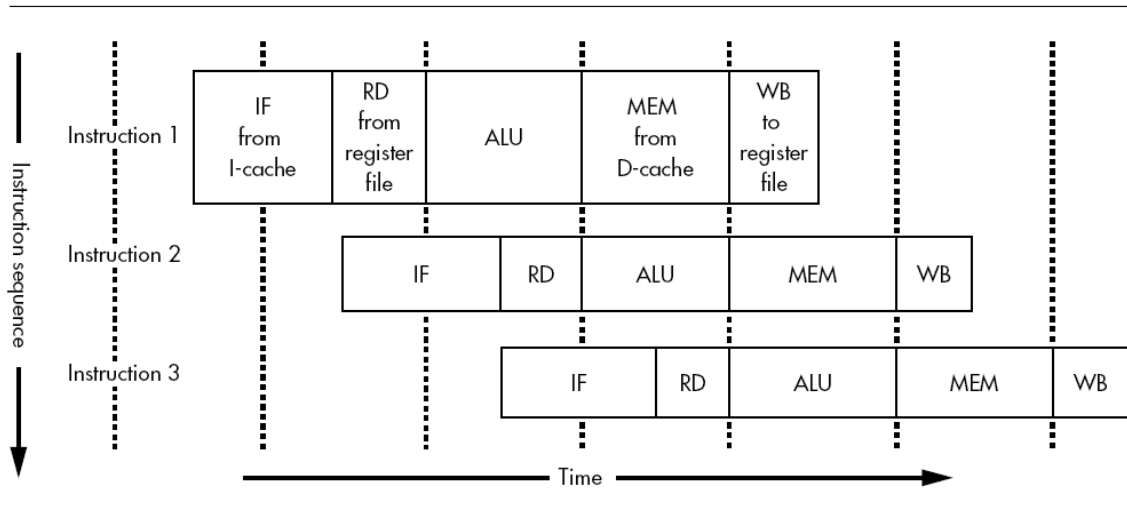


Ilustración 6 - Cauce segmentado de instrucciones

IF: Representa el Fetch de la instrucción, que podría considerarse como la búsqueda y obtención de la instrucción, datos, etc.

RD: Representa el decoding de la instrucción (señales que se deben activar por la UCP, etc.)

ALU: Representa el tiempo de ejecución de la instrucción en la unidad aritmético-lógica, es el procesamiento en sí de la instrucción.

MEM: Representa la escritura de datos en la cache de la memoria (en la sección de datos). Esta fase sólo se realiza en las cargas y almacenamientos de variables.

WB: Representa el Write Back, que es la escritura de los resultados en un registro.

Como se aprecia en la figura, varias instrucciones se ejecutan simultáneamente pero siempre que no se solapen dos fases del mismo tipo (ya que cada fase ocupa una sección del hardware).

Hay una serie de reglas básicas que rigen cuando dos o más instrucciones pueden ejecutarse en paralelo. Dos instrucciones se pueden ejecutar en paralelo siempre y cuando no existan dependencias de datos entre ellas. Estas dependencias son básicamente tres:

RAW (lectura después de escritura): No puedo comenzar una segunda instrucción si algún operando de ésta es el resultado de la primera.

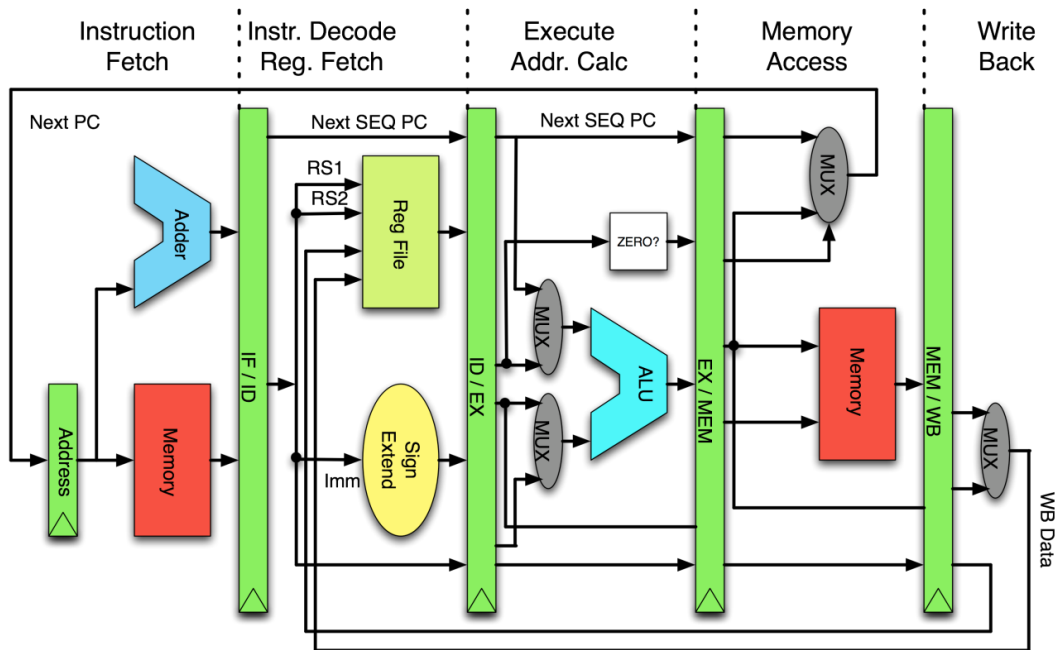


Ilustración 7 - Hardware segmentado

WAR (escritura después de lectura): No puedo comenzar una segunda instrucción si el resultado de ésta es un operando de la primera en ejecución. Esta dependencia se da debido a que no todas las instrucciones tardan el mismo tiempo en ejecutarse (una suma es mucho más rápida que una división).

WAW (escritura después de escritura): Una segunda instrucción no puede comenzar si el resultado se almacenará en la misma posición que la que está en ejecución. Al igual que antes, debido al tiempo de las instrucciones podría darse que el último valor establecido fuera el de la primera instrucción, siendo esto erróneo.

Donde se aprecia que región del hardware está destinada a cada una de las 5 etapas del proceso de segmentación. Los latch verdes que aparecen entre las diferentes etapas son elementos hardware que almacenan los datos de la instrucción actual, ya que en las demás regiones hardware hay otras instrucciones y se necesitan recordar datos de la instrucción actual. Sin ellos sería imposible el cauce segmentado de instrucciones. La presencia de dos cuadros rojos referenciando la memoria, no significa que se trate de dos memorias, sino que en la etapa IF se referencia la búsqueda de la instrucción y en MEM se referencia la escritura en memoria de datos (memoria o cache, ya que aquí no se representan las jerarquías de memoria).

Es importante destacar que el procesador MIPS R3000 permite el cauce segmentado con cinco etapas, pero no permite técnicas más avanzadas como la planificación dinámica de instrucciones (o lo que es lo mismo, ejecución fuera de orden), esto está reservado para versiones más modernas y potentes de MIPS. La ejecución fuera de orden consiste en que se pueden ejecutar instrucciones con un orden distinto al orden de memoria (orden lógico del programa) siempre y cuando no afecten al resultado final de la ejecución.

2.5 Instrucciones

Existen en MIPS R3000 y en todo el juego de instrucciones MIPS I tres tipos básicos de instrucciones (las R, las I y las J) más las denominadas pseudoinstrucciones, que son aquellas que no tienen codificación binaria de por sí y han de ser convertidas en una serie de instrucciones de los tres tipos antes mencionados para poder ser ejecutadas.

Instrucciones de tipo R: En este tipo de instrucciones sólo se usan registros. Su formato es el siguiente

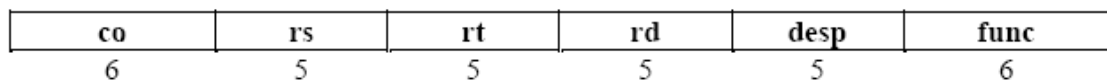


Ilustración 8 - Instrucción tipo R

Instrucciones de tipo I: En este tipo de instrucciones se usan dos registros y un literal inmediato (un entero). Su formato es el siguiente:



Ilustración 9 - Instrucción tipo I

Instrucciones de tipo J: Estas son instrucciones que carecen de registro, sólo tienen un código de instrucción y una dirección de memoria. Son las instrucciones de salto incondicional. Su formato es el siguiente:

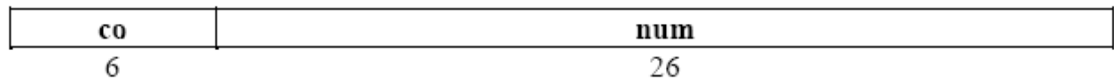


Ilustración 10 - Instrucciones tipo J

Las referencias a los elementos de las instrucciones mencionadas son las siguientes:

Abreviatura	Descripción
co	Código de operación
rs	Primera referencia a un registro fuente
rt	Segunda referencia a un registro fuente
rd	Referencia a un registro destino
desp	Valor del desplazamiento
func	Variantes del código de operación

Tabla 3 - Elementos de las instrucciones

3.6 Instrucciones aritméticas o lógicas

A continuación se describe el juego de instrucciones del MIPS R3000 (el juego se denomina MIPS I).

Instrucción	Descripción
abs Rdest, Rsrc	Valor absoluto
add Rdest, Rsrc1, Src2	Suma con desbordamiento
addu Rdest, Rsrc1, Src2	Suma sin desbordamiento
and Rdest, Rsrc1, Src2	Operación lógica AND
div Rsrc1, Rsrc2	Divide con desbordamiento. Deja el cociente en el registro <i>lo</i> y el resto en el registro <i>hi</i>
divu Rsrc1, Rsrc2	Divide sin desbordamiento. Deja el cociente en el registro <i>lo</i> y el resto en el registro <i>hi</i>
div Rdest, Rsrc1, Src2	Divide con desbordamiento
div Rdest, Rsrc1, Src2	Divide sin desbordamiento
mul Rdest, Rsrc1, Src2	Multiplica sin desbordamiento
mulo Rdest, Rsrc1, Src2	Multiplica con desbordamiento

mulou Rdest, Rsrc1, Src2	Multiplicación con signo y con desbordamiento
mult Rsrc1, Rsrc2	Multiplica, la parte baja del resultado se deja en el registro <i>lo</i> y la parte alta en el registro <i>hi</i>
mult Rsrc1, Rsrc2	Multiplica con signo, la parte baja del resultado se deja en el registro <i>lo</i> y la parte alta en el registro <i>hi</i>
neg Rdest, Rsrc	Niega el valor (detecta desbordamiento)
negu Rdest, Rsrc	Niega el valor (sin desbordamiento)
nor Rdest, Rsrc1, Src2	Operación Lógica NOR
not Rdest, Rsrc	Operación Lógica NOT
or Rdest, Rsrc1, Src2	Operación Lógica OR
rem Rdest, Rsrc1, Src2	Resto (Módulo), pone el resto de dividir Rsrc1 por Src2 en el registro Rdest.
rol Rdest, Rsrc1, Src2	Rotar a la izquierda
ror Rdest, Rsrc1, Src2	Rotar a la derecha
sll Rdest, Rsrc1, Src2	Desplazamiento lógico de bits a la izquierda
srl Rdest, Rsrc1, Src2	Desplazamiento lógico de bits a la derecha
sra Rdest, Rsrc1, Rsrc2	Desplazamiento aritmético de bits a la derecha
sub Rdest, Rsrc1, Src2	Resta (con desbordamiento)
subu Rdest, Rsrc1, Src2	Resta (sin desbordamiento)
xor Rdest, Rsrc1, Src2	Operación Lógica XOR

Tabla 4 - Instrucciones aritméticas y lógicas

Instrucción	Descripción
li Rdest, inmediato	Cargar valor inmediato
lui Rdest, inmediato	Cargar los 16 bits de la parte baja del valor inmediato en la parte alta del registro. Los bits de la parte baja se pone a 0.

Tabla 5 - Instrucciones manipulación de constantes

Instrucción	Descripción
seq Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 y Src2 son iguales, en otro caso pone 0.
sge Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es mayor o igual a Src2, y 0 en otro caso (para números con signo).
sgeu Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es mayor o igual a Src2, y 0 en otro caso (para números sin signo).
sgt Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es mayor que Src2, y 0 en otro caso (para números con signo).
sgtu Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es mayor que Src2, y 0 en otro caso (para números sin signo).
sle Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es menor o igual a Src2, en otro caso pone 0 (para números con signo).
sleu Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es menor o igual a Src2, en otro caso pone 0 (para números sin signo).
slt Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es menor a Src2, en otro caso pone 0 (para números con signo).
sltu Rdest, Rsrc1, Src2	Pone Rdest a 1 si Rsrc1 es menor a Src2, en otro caso pone 0 (para números sin signo).
sne Rdest, Rsrc1, Src2	Pone Rdest to 1 si el registro Rsrc1 no es igual a Src2 y 0 en otro caso.

Tabla 6 - Instrucciones de comparación

Instrucción	Descripción
b etiqueta	Bif. incondicional a la instrucción que está en etiqueta.
beq Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 es igual a Src2.
beqz Rsrc, etiqueta	Bif. condicional si el registro Rsrc es igual a 0.
bge Rsrc1, Src2, etiqueta	Bif. condicional si el registro Rsrc1 es mayor o igual a Src2 (con signo).
bgeu Rsrc1, Src2, etiq	Bif.. condicional si el registro Rsrc1 es mayor o igual a Src2 (sin signo).
bgez Rsrc, etiqueta	Bif. condicional si el registro Rsrc es mayor o igual a 0.
bgezal Rsrc, etiqueta	Bif. condicional si el registro Rsrc es mayor o igual a 0. Guarda la dirección actual en el registro \$ra (\$31).

Instrucción	Descripción
bgt Rsrc1, Src2, etiqueta	Bif. condicional si el registro Rsrc1 es mayor que Src2 (con signo).
bgtu Rsrc1, Src2, etiqueta	Bif. condicional si el registro Rsrc1 es mayor que Src2 (sin signo).
bgtz Rsrc, etiqueta	Bif. condicional si Rsrc es mayor que 0.
ble Rsrc1, Src2, etiqueta	Bif. Condicional si Rsrc1 es menor o igual a Src2 (con signo).
bleu Rsrc1, Src2, etiqueta	Bif. Condicional si Rsrc1 es menor o igual a Src2 (sin signo).
blez Rsrc, etiqueta	Bif. Condicional si Rsrc es menor o igual a 0.
bltzal Rsrc, etiqueta	Bif. Condicional si Rsrc es menor que 0. Guarda la dirección actual en el registro \$ra (\$31).
blt Rsrc1, Src2, etiqueta	Bif. Condicional si Rsrc1 es menor que Src2 (con signo).
bltu Rsrc1, Src2, etiqueta	Bif. Condicional si Rsrc1 es menor que Src2 (sin signo).
bltz Rsrc, etiqueta	Bif. Condicional si Rsrc es menor que 0.
bne Rsrc1, Src2, etiqueta	Bif. Condicional si Rsrc1 no es igual a Src2.
bnez Rsrc, etiqueta	Bif. Condicional si Rsrc no es igual a 0.
j etiqueta	Salto incondicional.
jal etiqueta	Salto incondicional, almacena la dirección actual en \$ra (\$31).
jalr Rsrc	Salto incondicional, almacena la dirección actual en \$ra (\$31).
jr Rsrc	Salto incondicional.

Tabla 7 - Instrucciones de bifurcación y salto

Instrucción	Descripción
la Rdest, dirección	Carga dirección en Rdest (el valor de dirección, no el contenido)
lb Rdest, dirección	Carga el byte de la dirección especificada y extiende el signo
lbu Rdest, dirección	Carga el byte de la dirección especificada, no extiende el signo
ld Rdest, dirección	Carga Rdest y Rdest + 1 con el valor del double (64 bits) que se encuentra a partir de la dirección especificada.
lh Rdest, dirección	Carga 16 bits de la dirección especificada, se extiende el signo
lhu Rdest, dirección	Carga 16 bits de la dirección especificada, no se extiende signo

Instrucción	Descripción
lw Rdest, dirección	Carga una palabra de la dirección especificada.
lwcx Rdest, dirección	Carga una palabra en el registro Rdest del coprocesador z.
lwl Rdest, dirección	Carga bytes en la palabra por la izquierda de la palabra desalineada.
lwr Rdest, dirección	Carga bytes en la palabra por la derecha de la palabra desalineada.
ulh Rdest, dirección	Carga media palabra (16 bits) de palabras desalineadas.
ulhu Rdest, dirección	Carga media palabra (16 bits) de palabras desalineadas (extiende el signo).
ulw Rdest, dirección	Carga una palabra (32 bits) de direcciones de memoria desalineadas.

Tabla 8 - Instrucciones de carga

Instrucción	Descripción
rfe	Retorno de una excepción, restaura el registro de estado.
syscall	Llamada al sistema, el registro v0 contiene el número de la llamada al sistema.
break n	Provoca una excepción de valor n.
nop	No operation, no hace nada.

Tabla 9 - Instrucciones de excepción y trap

Instrucción	Descripción
sb Rsrc, dirección	Almacena el byte más bajo de Rsrc en la dirección indicada.
sd Rsrc, dirección	Almacena un double (64 bits) en la dirección indicada, el valor de 64 bits es proviene de Rsrc y Rsrc + 1.
sh Rsrc, dirección	Almacena la media palabra (16 bits) baja de un registro en la dirección de memoria indicada.
sw Rsrc, dirección	Almacena la Rsrc en la dirección indicada.
swcx Rsrc, dirección	Almacena el valor del registro Rsrc del coprocesador z.
swl Rsrc, dirección	Almacena los bytes del registro empezando por la derecha en la dirección de memoria indicada.
swr Rsrc, dirección	Almacena los bytes del registro empezando por la izquierda en la

Instrucción	Descripción
	dirección de memoria indicada.
ush Rsrc, dirección	Almacena la parte baja del registro en una dirección desalineada.
usw Rsrc, dirección	Almacena la palabra del registro en una dirección desalineada.

Tabla 10 - Instrucciones de almacenamiento

Instrucción	Descripción
move Rdest, Rsrc	Mueve el contenido del registro Rsrc al registro Rdest.
mfhi Rdest	Mueve el contenido del registro HI al registro Rdest.
mflo Rdest	Mueve el contenido del registro LO al registro Rdest.
mthi Rsrc	Mueve el contenido del registro Rsrc al registro HI.
mtlo Rsrc	Mueve el contenido del registro Rsrc al registro LO.
mfcz Rdest, CPsrc	Mueve el contenido del registro CPsrc del coprocesador z al registro de la CPU Rdest.
mfc1.d Rdest, FRsrc1	Mueve el contenido de los registros FRrc1 y FRsrc1+1 a los registros de la CPU Rdest y Rdest + 1.
mtcz Rsrc, CPdest	Mueve los contenidos del registro Rsrc de la CPU al registro Cpdest del coprocesador z.

Tabla 11 - Instrucciones de movimiento de datos

Instrucción	Descripción
abs.s fd, fs	Valor absoluto
add.s fd, fs, ft	Suma de los registros fs y ft
c.eq.s fs, ft	Compara fs y ft, si son iguales pone el flag de condición de coma flotante true. Se deben utilizar bc1t o bclf para comprobar el valor del flag.
c.le.s fs, ft	Si fs es menor o igual que ft pone el flag de condición de coma flotante true. Se deben utilizar bc1t o bclf para comprobar el valor del flag.
c.lt.s fs, ft	Si fs es menor que ft pone el flag de condición de coma flotante true. Se deben utilizar bc1t o bclf para comprobar el valor del flag.
div.s fd, fs, ft	Divide fs entre ft y deja el resultado en fd
l.s fdest, direc	Lee de la dirección de memoria indicada un real y lo almacena en fd.
mov.s fd, fs	Mueve el contenido del registro fs al registro fd.

mul.s fd, fs, ft	Multiplica los registros fs y ft y deja su resultado en fd.
neg.s fd, fs	Niega el valor del real contenido en fs y lo almacena en fd
s.s fd, direc	Escribe en la dirección de memoria indicada el real contenido en el registro fd
sub.s fd, fs, ft	Resta el contenido de dos registros y almacena el resultado en fd.
bc1t dir_relativa	Pasa a ejecutar el código de la dirección indicada si el flag de coma flotante es 1 (true).
bc1f dir_relativa	Pasa a ejecutar el código de la dirección indicada si el flag de coma flotante es 0 (false).

Tabla 12 - Instrucciones del coprocesador de coma flotante

2.7 Servicios ofrecidos por el sistema

Ahora se describirán los servicios del sistema que se usaran en los simuladores del procesador. En total se describen 16 servicios, de los cuales los 10 primeros son usables en el simulador SPIM y en el MARS. Los últimos 6 servicios en principio son solo compatibles con el MARS.

Servicio	Código de llamada	Argumentos	Resultado
print_int	1	\$a0 = entero	
print_float	2	\$f12 = real (32 bits)	
print_double	3	\$f12 = real (64 bits)	
print_string	4	\$a0 = cadena	
read_int	5		Entero (en \$v0)
read_float	6		Real 32 bits (en \$f0)
read_double	7		Real 64 bits (en \$f0)
read_string	8	\$a0=buffer, \$a1 = longitud	
sbrk	9	\$a0 = cantidad	Dirección (en \$v0)
exit	10		
print_char	11	\$a0 = caracter	
read_char	12		\$a0 = carácter leído
file_open	13	\$a0 = ruta completa,	\$a0 = descriptor de fichero

		\$a1= flags (0x0 escritura, 0x1 lectura/escritura, 0x100 crear, 0x200 truncar, 0x8 continuar, 0x4000 texto, 0x8000 binario), \$a2 = permisos (0x100 lectura, 0x80 escritura)	
file_read	14	\$a0 = descriptor de fichero, \$a1 = dirección del buffer, \$a2 = bytes a leer	\$a0 = datos leídos (-1 error, 0 eof)
file_write	15	\$a0 = descriptor de fichero, \$a1 = dirección del buffer, \$a2 = bytes a escribir	\$a0 = datos escritos (-1 error, 0 eof)
file_close	16	\$a0 = descriptor del fichero	

Tabla 13 - Servicios del sistema

3.8 Direccionamiento

Hay cinco tipos de direccionamiento en las instrucciones del MIPS R3000 (y en general en todos los juegos de instrucciones de los MIPS). Las formas de direccionar son:

Inmediato: El direccionamiento inmediato hace referencia al uso de constantes numéricas dentro del programa. Se utiliza en instrucciones de tipo I como por ejemplo:

```
addi $s2, $s2, 32
```

Y su formato es el siguiente:



Ilustración 11 - Direccionamiento inmediato

Registro: Este direccionamiento hace referencia a cuando el dato a utilizar está almacenado en un registro. Se utiliza en instrucciones de tipo R e I como por ejemplo:

```
add $s1, $s1, $s2
```

Y su formato es el siguiente:

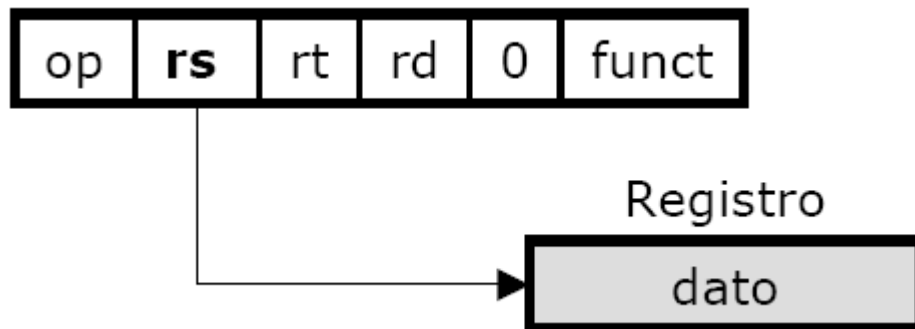


Ilustración 12 - Direccionamiento a registro

Base más desplazamiento: Este direccionamiento se usa para acceder a posiciones contiguas en una dirección almacenada en un registro más un escalar sumado. Se utiliza en instrucciones de tipo I como por ejemplo:

```
sw $t, 12($s0)
```

Y su formato es el siguiente:

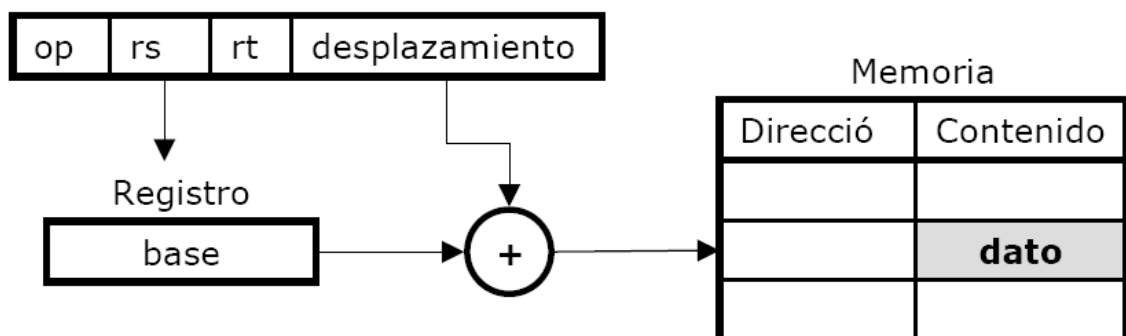


Ilustración 13 - Direccionamiento base más desplazamiento

Relativo a PC: Es el direccionamiento típico de las instrucciones de salto condicional. Referencia una dirección de 16 bits (una etiqueta) a la cual se saltará en caso de que se cumpla o no una condición. Se utiliza en instrucciones de tipo J como por ejemplo:

```
bne $t1, $t2, etiqueta
```

Y su formato es el siguiente:

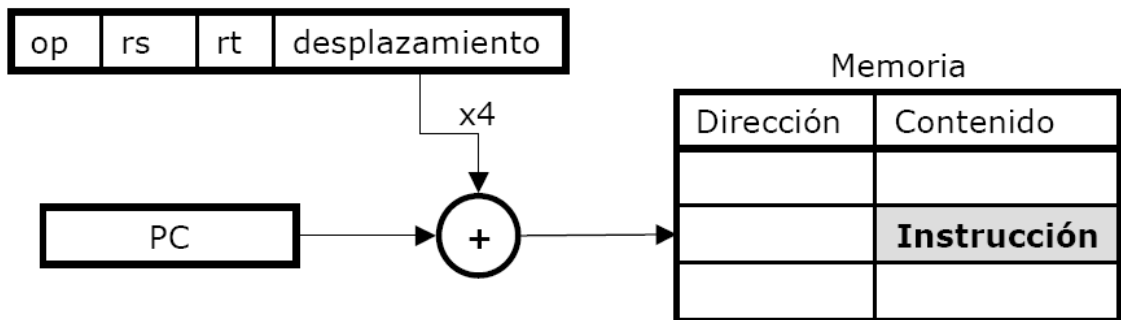


Ilustración 14 - Direccionamiento relativo a PC

Pseudodirecto: Es el direccionamiento de las instrucciones de salto incondicional, tales como la llamada a subrutinas. Se salta siempre a la etiqueta que representa una dirección de 26 bits. Se utiliza en instrucciones de tipo J como por ejemplo:

```
j subrutina
```

Y su formato es el siguiente:

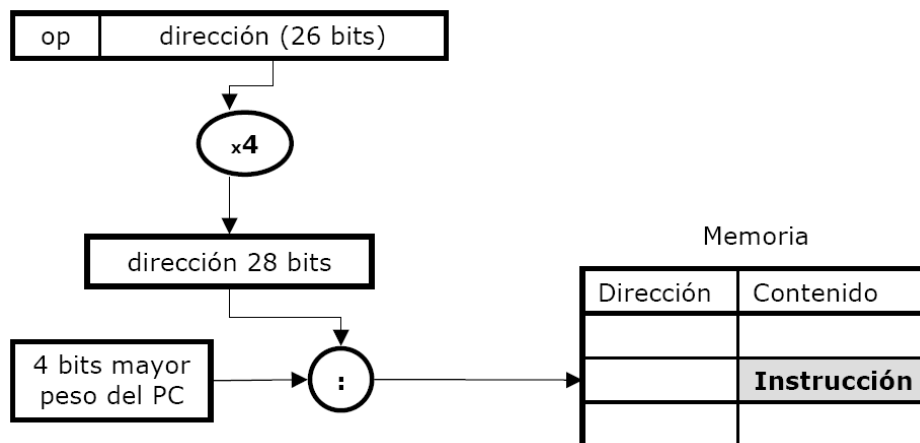


Ilustración 15 - Direccionamiento pseudodirecto

3.9 Directivas

Hay una serie de directivas en los programas del ensamblador del MIPS R3000 que deben ser respetadas para que el programa se cargue adecuadamente en memoria y sea entendible posteriormente por el procesador. Las principales directivas presentes en el MIPS R3000 (a excepción de los tipos de datos) son las siguientes:

.align n: Representa la forma de alinear los datos en memoria. Se alinearan de modo 2^n .

.data: Representa el segmento de datos del programa. Es la región de la RAM donde se localizan las variables.

.rdata: Representa los datos de “read” eso quiere decir que es la región de los datos almacenados en memoria pero que serán constantes a lo largo de la ejecución del programa.

.globl: Define el elemento como único. Se suele usar para el main del programa.

.space n: Reserva n bytes sin asignar dentro de la memoria RAM. Puede usarse para arrays no inicializados.

.text: declara el segmento de código del programa. El procesador entenderá que ese segmento son instrucciones a ejecutar. A no ser que se especifique lo contrario el segmento de .text comienza en la dirección de memoria 0x00400000.

3.10 Tipos de datos

Los tipos de datos básicos manejados por el segmento de datos en el procesador son los siguientes:

Nombre	Tipo	Tamaño
.ascii	Cadena sin terminal nulo	Definido por el usuario
.asciiz	Cadena con terminal nulo	Definido por el usuario
.byte	Entero	1 byte
.half	Entero	2 bytes
.word	Entero	4 bytes

.dword	Entero	8 bytes
.float	Real	4 bytes
.double	Real	8 bytes

Tabla 14 - Tipos de datos

Los tipos de datos antes mencionados son representados nivel de byte (y bits) en big-endian dentro de memoria en el procesador MIPS R3000 del siguiente modo:

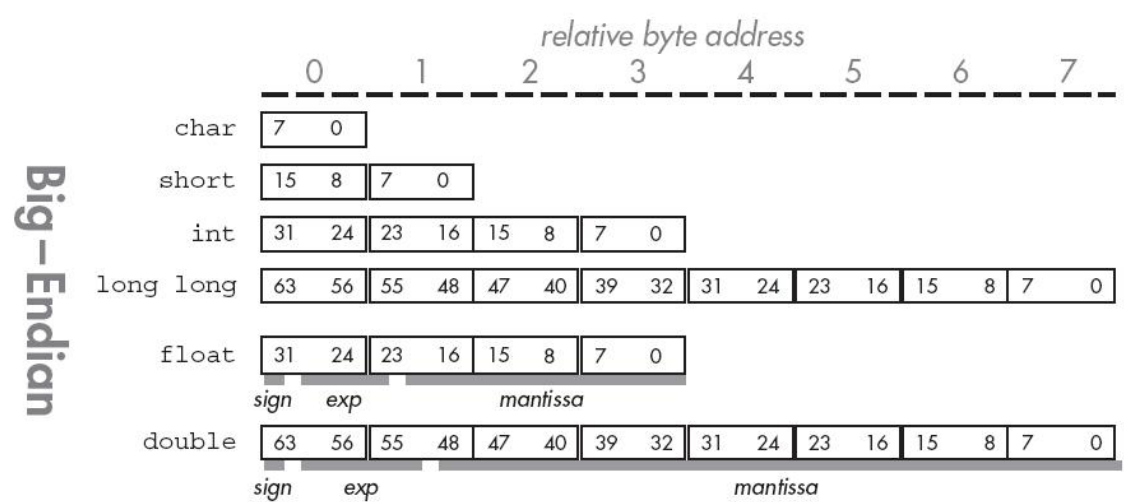


Ilustración 16- Representación de los datos

4 Análisis Léxico

4.1 Descripción

El análisis léxico es la primera de las etapas que posee un compilador, en nuestro caso para su desarrollo se ha utilizado la herramienta flex [], que nos permite la creación de analizadores léxicos de forma sencilla solo definiendo los tokens a utilizar mediante la utilización de expresiones regulares o condiciones de parada.

4.2 Lista de tokens

A continuación se presenta los distintos tokens definidos en el analizador léxico y que representan el conjunto de no terminales de la gramática del lenguaje C-Lite admitido por el compilador.

Token	Descripción
float	Tipo de dato, reales en coma flotante.
int	Tipo de dato, entero de 4 bytes.
short	Tipo de dato, entero corto de 2 bytes.
char	Tipo de dato, carácter ascii 127.
void	Tipo de dato void.
if	Token de expresiones condicionales.
else	Token de expresiones condicionales.
for	Token de bucle FOR.
while	Token de bucle WHILE.
return	Token de retorno de función.
#define	Token de definición de constantes.
printf	Token de escritura de cadenas de texto.
scanf	Token de lectura de variables.
+	Token de operación binaria suma.
++	Token de operación unaria suma + 1.
-	Token de operación binaria resta.
--	Token de operación unaria resta - 1.
*	Token de operación binaria multiplicación.
/	Token de operación binaria división.
%	Token de operación binaria resto de la división entera.
+=	Token de asignación y suma binaria.
-=	Token de asignación y resta binaria.
*=	Token de asignación y multiplicación binaria.
/=	Token de asignación y división binaria.
==	Token de comparación, igualdad.

Token	Descripción
<=	Token de comparación, menor o igual.
>=	Token de comparación, mayor o igual.
<>	Token de comparación, distinto.
<	Token de comparación, menor.
>	Token de comparación, mayor.
=	Token de asignación simple.
&	Token lógico, AND lógico.
;	Token sintáctico punto y coma.
	Token lógico, OR lógico.
,	Token sintáctico coma.
(Token sintáctico de apertura de paréntesis.
)	Token sintáctico de cierre de paréntesis.
[Token sintáctico de apertura de corchete.
]	Token sintáctico de cierre de corchete.
{	Token sintáctico de apertura de llave.
}	Token sintáctico de cierre de llave.
ID	Cadena de caracteres alfanuméricos donde el primer elemento debe ser siempre un carácter alfabético.
Texto	Cadena de caracteres alfanuméricos que permite cualquier tipo de carácter.
Caracter	Carácter ascci.
print	Token de escritura de variables.
printfn	Token de escritura de variables con fin de línea.
"	Token sintáctico de comillas dobles.

Tabla 15 - Lista de tokens analizador léxico

4.3 Decisiones de diseño

4.3.1 Decisiones generales

A continuación se presentan el conjunto de decisiones de diseño que fueron tomadas para la creación del analizador léxico.

- Las palabras reservadas del lenguaje de entrada solo podrán ser introducidas en letras minúsculas.
- Los tipos de datos permitidos por el lenguaje de entrada serán los siguientes, el valor comprendido entre -2 y 3 representa el peso del tipo el cual es utilizado para el proceso de comprobación de tipos del análisis semántico (ver [apartado 6](#)) y para el proceso de generación de la instrucciones del lenguajes ensamblador (ver [apartado 8](#)):

- -2: void
- -1: texto (String)
- 0: char
- 1: short
- 2: int
- 3: float
- Existirán cinco tipos de básico en el lenguaje de entrada que serán void, char, short, int y float, pudiendose crear arrays de n elementos y n dimensiones de cualquiera de esos tipos.
- Se permitirán la inserción de comentarios de dos tipo aquellos que comiencen por doble barra “//” o aquellos que comiencen con “/*” y finalicen con “*/”, el analizador ha sido diseñado para omitir estos elementos en el caso de que aparezcan.
- Se permiten la inserción de comentarios dentro de una sentencia de escritura **print** y **println** y este será omitido por el analizador léxico.
- Se omitirán aquellos caracteres especiales como el tabulador.
- Cada vez que se encuentre un retorno de línea, es decir un \n el sistema realizará el incremento de una variable llevando así el computo de la línea actual para indicar los posibles errores léxico, sintácticos y semánticos.

4.3.2 Comunicación con analizador sintáctico

El proceso de comunicación con el analizador sintáctico, se realizó mediante el uso de la estructura **yyval**, esta estructura permite la comunicación entre Flex y Bison en ella se almacena información referente a los valores de los id, los números, las cadenas de caracteres así como ciertos códigos que nos permiten saber cierta información referente a los tipos de datos que han sido leído. Esta estructura se define de forma más exhaustiva en el [apartado 5.2.1](#) del analizador sintáctico.

Además se ha creado un modulo especial de control que nos permite transmitir información más detallada entre los distintos proceso del compilador, está en la unidad de control que está formado con un conjunto de flags que no informan en todo momento de información referente a los tokens leídos del fichero de entrada. A continuación se describen los flags más importantes del modulo de control.

- Linea: Indica la línea en la cual se encuentra el proceso de análisis, de forma que en el caso de que se produzca un error nos indique la línea.
- Log: Indica que el usuario ha solicitado al sistema que se produzca la escritura de las trazas de ejecución.
- leidoIf: Indica que se ha producido la lectura de un IF, sirve para aplicar ciertos tratamientos especiales en ciertas zonas de la gramática para las sentencias IF.
- leidoWhile: Indica que se ha producido la lectura de un WHILE, sirve para aplicar ciertos tratamientos especiales en ciertas zonas de la gramática para las sentencias IF.
- leidoFor: Indica que se ha producido la lectura de un FOR, sirve para aplicar ciertos tratamientos especiales en ciertas zonas de la gramática para las sentencias IF.
- Profundidad: Indica el nivel de profundidad del código analizado, es decir que nos indica la profundidad en las sentencias anidadas por ejemplo una sentencia If dentro de una sentencia While, su utilidad reside en la eliminación, la optimización y el control de correcto uso de los registros del procesador. Para más información consultar el [apartado 7.4.1](#) de este documento.

4.3.3 Control de errores

El proceso de control de errores en el analizador léxico no existe ya que el analizador sintáctico lee información y la asocia con una de las reglas que tiene definidas, los errores solo se muestra a partir del proceso de análisis sintáctico donde el token enviado por el analizador léxico no se corresponde con ninguna de las producciones de la gramática, por lo tanto en el analizador léxico no se produce control de errores.

4.4 Problemas encontrados

A continuación se presentan aquellos problemas que surgieron a la hora de realizar el analizador léxico y las soluciones que se desarrollaron para que el analizador fuera capaz de realizar de recoger todos los tokens definidos en el [apartado 4.2](#).

Números negativos

Inicialmente se decidió definir los números negativos como reglas en el yacc donde un número podía tener o no un signo de menos delante, pero debido a la estructura de la gramática había situaciones en las cuales la gramática en vez de definir el número como negativo, generaba una operación de tipo resta sobre el identificador anterior, de forma que decidimos crear una expresión regular para cada uno de los tipos de números que permitíamos que en nuestro caso eran reales y enteros, de forma que pudieran tener o no símbolo negativo delante, a continuación se presentan las dos expresiones regulares.

digito	[0-9]
-?{digito}+	//Número entero
-?{digito}+"."{digito}+	//Número real

Ilustración 17 - Expresiones regulares números

Como se puede observar en la ilustración 17 se utilizó el **carácter ?** que indica que el carácter inmediatamente anterior puede aparecer 0 o 1 vez en el token.

Comentarios

El principal de los problemas que nos surgió en el analizador léxico fue la omisión de comentarios de modo que estos fueran detectados siempre y no fueran enviadas al analizador sintáctico como tokens, ya que esto produciría un error debido a que generaba tokens que no equiparaban con ninguna de las producciones de la gramática., pudieron diferenciarse dos problemas principales derivados de los comentarios.

- Comentarios en entrada “print o println”

El principal problema que surgió fue la necesidad de poder insertar un comentario dentro de una instrucción de escritura a la salida estándar de modo que esta no fuera enviada al analizador semántico como parte de la cadena, tras varios procesos de pruebas comprobamos que las expresiones regulares

que permitieran esto eran demasiado complejas por lo que optamos por utilizar **condiciones de parada**.

- Computo de líneas

Otro de los problemas que encontramos en los comentarios, fue en la situación en la que apareciera un comentario que estuviera dividido en varias líneas, lo que producía que no se realizaba el incremento de la variable de control del número de líneas del sistema, lo que producía que a la hora de mostrar errores léxicos, sintácticos o semánticos no se indica de forma correcta la línea en la cual se encontraba el error, la única solución que definimos para poder solucionar este error fue el uso de las **condiciones de parada**.

Las condiciones de parada nos permitieron solventar todos los errores concernientes al tratamiento de comentarios, a continuación se indica la solución aplicada para cada uno de los tipos de comentarios.

"//"	BEGIN(comentarioLinea);
<comentarioLinea>[^\\n]*	
<comentarioLinea>\\n	{incrementarLinea();BEGIN(INITIAL);}

Ilustración 18 - Condición de parada comentario //

Como se puede observar en la ilustración 18 la condición de parada genera un bucle que se ejecuta cuando se encuentra una ocurrencia de "//" y no finaliza hasta que encuentra un fin de línea, incrementándose entonces la variable de número de línea y volviéndose al estado normal para que el analizador pueda seguir recogiendo tokens de forma normal.

"/*"	BEGIN(comentario);
<comentario>[^*\\n]*	
<comentario>"*" + [^*/\\n]*	
<comentario>\\n	{incrementarLinea();}
<comentario>"*" + "/"	BEGIN(INITIAL);

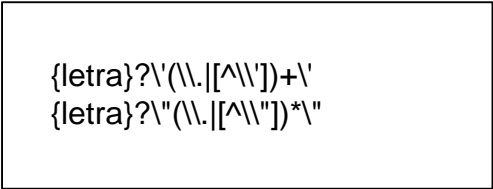
Ilustración 19 - Condición de parada comentario /*

Como se puede observar en la ilustración 24 la condición de parada genera un bucle que se ejecuta cuando se encuentra una ocurrencia de “/*” y no finaliza hasta que se encuentra un ocurrencia de “*/”, también se pueden observar tres expresiones regulares, siguiendo el orden en el aparecen:

- Expresión 1: Acepta cualquier carácter que no sea un * o un fin de línea “\n”.
- Expresión 2: Acepta líneas de * y no tiene en cuenta ni los fin de línea “\n” ni el fin de comentario.
- Expresión 3: Acepto caracteres de tipo fin de línea, de modo que se pueda realizar el incremento de la variable que indica el número de línea actual.

Cadenas de texto

Otro problema que surgió con las cadenas de texto fue el poder realizar la diferenciación entre un carácter y una cadena de texto, ya que a nivel sintáctico y semántico nos era muy útil realizar esta diferenciación a nivel léxico, por lo tanto se tuvieron que crear dos expresiones regulares muy similares para realizar esta diferenciación, de forma que una solo seleccionara un carácter y la otra seleccionara un cadena de un tamaño indeterminado.



```
{letra}?\'(\\.|[^\'])*\'
{letra}?\"(\\.|[^\"])*\"
```

Ilustración 20 - Expresiones regulares para tratamiento de textos y caracteres

En la ilustración 20 se presentan dos expresiones regulares:

- La primera expresión se refiere al tratamiento de caracteres, de forma que cada vez que se encuentre un carácter entre dos comillas simples será considerado por el compilador como un carácter, enviándose al analizador sintáctico el token CHARACTER.
- La segunda expresión se refiere al tratamiento de cadenas de texto, esta expresión captura todo el texto que se encuentre entre dos comillas dobles y lo

envía al analizador sintáctico como una cadena de texto mediante el token TEXTO.

Identificadores ensamblador

Uno de los problemas que surgieron al comenzar el proceso de generación de código ensamblador es que había ciertos identificadores que no se podían utilizar debido a que son palabras reservadas del lenguaje ensamblador, por lo que se decidió incluir una regla en el analizador léxico de forma que todos los identificadores (nombre de variables, nombre de constantes, nombre de funciones) se les añada una cláusula al comienzo “ID_” de forma que no se produzca errores de compilación en el lenguaje de ensamblador y se pueda utilizar cualquier identificador en el lenguaje C-Lite.

5 Análisis Sintáctico

5.1 Descripción

El análisis sintáctico es la segunda de las etapas que posee un compilador, en nuestro caso su desarrollo se ha realizado utilizando la herramienta Bison [], que nos permite la creación de analizadores semánticos definiendo los tokens y las reglas (producciones) que forman la gramática en formato BNG. Debido al haber utilizado esta herramienta el analizador semántico será la parte más importante de nuestra aplicación ya que sobre sus reglas se realizarán todos los procesos de análisis semántico, generación de código intermedio y generación de código objeto, todos ellos se realizarán en tiempo real debido a que el proceso de compilado se realiza en una sola pasada.

5.2 Decisiones de diseño

5.2.1 Pila de comunicaciones

El generador de analizadores sintácticos yacc posee un sistema de comunicación entre producciones de forma similar a como funciona una pila de datos, mediante la cual es capaz de mandar información desde una producción a la producción que produjo el salto o recorrido a la otra, entendiéndose como salto o recorrido como el proceso de equiparación de los tokens en la gramática a través de las reglas.

Mediante esta pila de comunicación es posible transferir valores a zonas determinadas a modo de poder realizar operaciones complejas como la inserción de variables definidas de forma encadenada (int a,b,c;).

5.2.2 Proceso de comunicación entre producciones

A continuación se presenta de forma detallada el proceso de comunicación entre producciones que se ha realizado en el compilador R3000 mediante la utilización de la pila de llamadas que provee yacc.

Debido al proceso de análisis que realiza yacc y a que se producen envíos de información desde el analizador léxico al analizador sintáctico que debe ser recogida para más utilizarse de las demás fases del proceso de análisis se han tenido que desarrollar una serie de estructuras de almacenamiento que posteriormente han sido

asociadas a determinadas producciones para poder recoger la información enviada por el analizador léxico y/o generada por el analizador sintáctico de forma que el sistema fuera capaz de realizar las siguientes acciones:

- Inserción de variables en la tabla de símbolos indicándose el tipo, el identificador, el ámbito y en el caso de que fuera una constante el valor.
- Comprobación de tipos en las sentencias matemáticas y condicionales.
- Transferir valores de control para optimizar ciertas acciones de los analizadores sintáctico y semántico y del proceso generación de código intermedio.

A continuación se presenta la ilustración 21 donde se incluyen todas las estructuras de control utilizadas en los proceso de comunicación entre producciones.

```
%union
{
    char cadena [4096];
    char cadenaCorta [32];
    int entero;
    float real;

    struct {
        int entero;
        float real;
        char nombre [4096];
        int tipo;
        char dimensiones[32];
    } general;

    struct {
        int entero;
        float real;
        int tipo;
    } numeros;
}
```

Ilustración 21 - Estructuras de comunicación entre procesos

A continuación se realiza una descripción detallada de cada uno de los elementos que forman la estructura unión.

- cadena: Este elemento almacena una cadena de datos de hasta 4096 caracteres, se utiliza para enviar desde el analizador léxico al analizador semántico las cadenas de caracteres, los identificadores y los caracteres, además también es utilizada en las producciones de la gramática para el proceso de comunicación de información.
- cadenaCorta: Este elemento almacena una cadena de datos de hasta 32 caracteres, es utilizada en el proceso de comunicación entre producciones en aquellas situaciones donde no es necesario la utilización de una cadena de gran tamaño.
- entero: Este elemento almacena un número entero, es utilizado para enviar los números enteros desde el analizador léxico, además también se utiliza para comunicar información entre varias reglas de la gramática.
- real: Este elemento almacena un número real, es utilizado para enviar los números enteros desde el analizador léxico, además también se utiliza para comunicar información entre varias reglas de la gramática.
- General: Esta es la estructura más importante que se ha utilizado el proceso de comunicación debido a que su principal uso consistía en obtener la información referentes a las distintos identificadores que aparecen a lo largo de los archivos a compilar.
 - Entero: variable utilizada para el almacenamiento de los números enteros utilizados en asignaciones, sentencias condiciones y sentencias matemáticas.
 - Real: variable utilizada para el almacenamiento de los números reales utilizados en asignaciones, sentencias condiciones y sentencias matemáticas.
 - Nombre: variable utilizada para el almacenamiento del nombre de identificador.
 - Tipo: variable utilizada para el almacenamiento del entero que define el tipo de dato del identificador, pudiendo tomar los siguientes valores.
 - -2: void
 - -1: texto (String)

- 0: char
- 1: short
- 2: int
- 3: float
- Dimensiones: variable utilizada para almacenar una cadena con las dimensiones del identificador en el caso de que sea un array, el almacenamiento de las dimensiones se realizaría de la siguiente forma.

D1, D2, D3, , DN

Donde cada dimensiones está separada de la siguiente por una coma a modo de diferenciarlas, este procedo de almacenamiento se realizar en todas las situaciones en las cuales aparece un array, independientemente de que sea en la definición del arrays, en una asignación o en su utilización en una sentencia matemática o de comparación.

- Numeros: Esta es una estructura que fue definida para la gestión de número en el proceso de análisis sintáctico y semántico.
 - Entero: variable utilizada para el almacenamiento de los números enteros utilizados en asignaciones, sentencias condiciones y sentencias matemáticas.
 - Real: variable utilizada para el almacenamiento de los números reales utilizados en asignaciones, sentencias condiciones y sentencias matemáticas.
 - Tipo: variable utilizada para el almacenamiento del entero que define el tipo de dato del identificador, pudiendo tomar los siguientes valores.
 - 2: Entero
 - 3: Real

Como se puede apreciar la estructura general engloba los elemento de la estructura números, pero decidió realizarse esta diferenciación debido a que no se realizara un consumo excesivo de la memoria que poseía la pila de yacc la cual para

nosotros era desconocida, por lo tanto modo de optimización del proceso de comunicación entre proceso se desarrollaron estas dos estructuras, basándonos en los mismos criterios se definieron los otros elementos de la unión a modo de disminuir la carga en los procesos de comunicación.

A continuación se presentan aquellas reglas de la gramática que han sido tipadas para permitir la comunicación de información entre ellas.

%type <cadenaCorta>	dimension
%type <entero>	tipo_dato
%type <entero>	elemento_base
%type <entero>	base_expresion_logica
%type <entero>	elemento_expresion_matematica
%type <entero>	elemento_expresion_logica
%type <entero>	expresion_matematica
%type <entero>	expresion_logica
%type <entero>	sentencia_especial
%type <entero>	sentencia_llamada
%type <entero>	elemento_argumento
%type <general>	identificador_base
%type <general>	valor_dimension
%type <general>	dimension_operacion
%type <general>	elemento_base_parametro
%type <general>	elemento_constante
%type <general>	identificador_operacion
%type <numeros>	numero
%type <general>	nombre_funcion
%type <general>	elemento_variable

Ilustración 22 - Reglas sobre las que se han aplicado tipos de datos

5.2.3 Definición de variables

Inicialmente la gramática permitió la definición de variables en cualquier parte del código pero debido a los diversos errores que indicaba yacc al generar la gramática se decidieron eliminar estas producciones lo cual supuso varias mejoras:

- Disminución de la complejidad de la gramática eliminando la mayor parte de los conflictos que se producían.
- Disminución de la complejidad del proceso de tratamiento y comprobación de variables.

5.2.3 Control de errores

El control en el analizador sintáctico se realiza un control de los tokens sobre las reglas de las gramáticas, en caso de que se lea un token que no corresponde con el token que posee la regla que se está tratando se realiza la finalización del proceso indicándose al usuario el token que se ha leído y que ha producido el error y la línea en la que se ha producido, para ellos se ha definido la función de error siguiente que se encuentra en el archivo de definición del analizador léxico.

```
int yyerror(char *s)
{
    printf("Se ha producido un error sintactico en la linea %d en el token\n", valorFlag(0), yytext);
    return -1
}
```

Ilustración 23 - Función de error del analizador sintáctico

5.3 Problemas encontrados

Los errores que han aparecidos en el analizador sintáctico han sido principalmente debidos a la estructura de la gramática ya que para obtener la información de los tokens y realizar ciertas acciones a nivel del analizador semántico o a nivel de la fase de código intermedio han sido necesario realizar modificaciones en la gramática. A continuación se definen a nivel general

- Recursividad izquierda

Inicialmente la gramática que fue definida estaba construida en su totalidad en FNG (Forma Normal de Greibach) y sin recursividad izquierda, lo cual produjo que al realizar la generación surgieran una gran cantidad de conflictos de tipo Reducción/Desplazamiento y Reducción/Reducción de forma que esto suponía que en algunas situaciones la gramática no se recorriera de forma correcta generando errores sintácticos inexistente, por lo que decidimos modificar todas las producciones de la gramática que poseían recursividad a la derecha y que producción algún tipo de conflicto por recursividad a la izquierda de forma que la mayor parte de los conflictos que habían surgido desaparecieron.

- Reglas vacías

Al igual que en el caso anterior la gramática inicialmente no poseía ninguna producción vacía, pero debido a los cambios que se realizaron para solventar los problemas derivados por la recursividad derecha, en algunas de las producciones en las que se aplicó recursividad izquierda hubo que añadir una regla vacía para que la recursividad finalizara y no produjera ningún conflicto.

- Redenominaciones

En algunos casos de la gramática nos ha sido necesario el uso de reglas de redenominación. Una regla de redenominación consiste en llamar a un no terminal que retornará directamente otro elemento. En nuestro caso ese elemento será normalmente un terminal. Esto ha sido necesario ya que no se permite la inclusión de código semántico en medio de una sentencia de la gramática. Por tanto, si hacemos una redenominación del elemento, podremos asignarle el código semántico al reducirse dicha regla. Esto es muy útil ya que en muchos casos es conveniente o directamente necesaria la inclusión de dicho código en medio de una regla gramatical.

- Conflictos

A pesar de las modificaciones realizadas en la gramática ha existido un conflicto Reducción/Desplazamiento que no se ha podido eliminar mediante la modificación de las reglas en las que se producía, este ha sido en la regla de definición de las condiciones, por lo que decidimos aplicar preferencia sobre la producción de la sentencia condicional sin else a la hora de que el analizador eligiera o no una regla.

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

sentencia_if
    : IF PARENTESISIZ expresion_logica fin_expresion cuerpo_sentencia %prec
      LOWER_THAN_ELSE;
    | IF PARENTESISIZ expresion_logica fin_expresion cuerpo_sentencia ELSE
      cuerpo_sentencia
    ;
```

Ilustración 24 - Aplicación de preferencias en sentencia_if

Como se pueden observar en la ilustración 24 se ha debido de definir un control en el proceso de análisis de las producciones de las sentencias condicionales el cual se ha realizado mediante la realización de las siguientes acciones.

- Inclusión del terminal de control LOWER_THAN_ELSE para poder aplicar el proceso de preferencia a la hora de realizar la equiparación de los tokens en las producciones.
- Desasociación de los terminales sobre los cuales se va a realizar el proceso de preferencia mediante la instrucción de yacc %nonassoc.
- Definición de la regla con mayor preferencia mediante la instrucción de yacc %prec, esta instrucción debe encontrarse siempre inmediatamente antes de uno de los token indicados mediante la instrucción %nonassoc.
- Comunicación de valores

Debido a como ha sido construida la gramática había situaciones en las cuales era necesario obtener valores de producciones no accesibles ya que esta información se encontraba en la producción anterior en la pila de llamadas es decir:

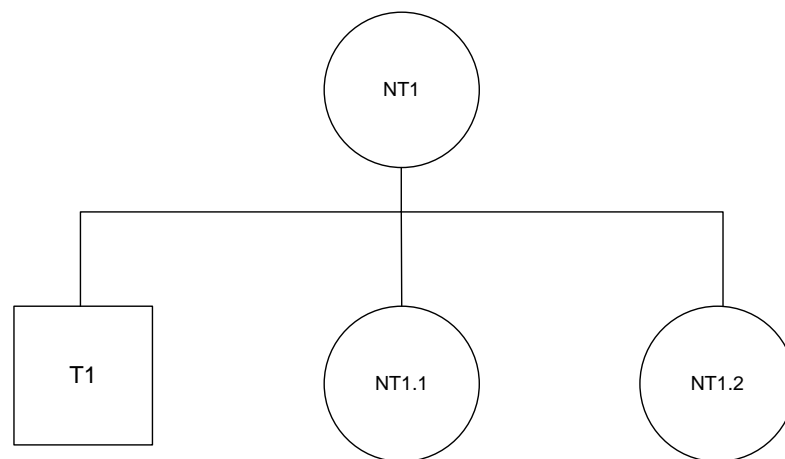


Ilustración 25 - Comunicación de valores entre producciones

El problema consistía en que desde el no terminal NT1.2 se quería acceder a la información del nodo terminal T1 mediante la pila del yacc, esto inicialmente parece imposible, pero yacc nos permite realizarlo mediante las siguientes reglas

\$<entero>-1

Mediante esta regla se puede acceder al primer elemento del no terminal NT1 desde el no terminar NT1.2.

6 Análisis Semántico

6.1 Descripción

El análisis semántico es la tercera de las etapas que posee un compilador, en nuestro caso este proceso se realiza de forma paralela al análisis sintáctico debido a que el proceso de compilado ha sido desarrollado para realizarse en una sola pasada, los procesos del analizador semántico son los siguientes:

- Control de tipos
Se realiza la comprobación de tipos en los procesos de asignación, de utilización en las sentencias condiciones y aritméticas y en llamadas a funciones.
- Control de acceso a variables
Se realiza la comprobación de que toda variable ha sido definida previamente.
- Control de ámbito de variables
Se realiza la comprobación de que una variable local de una función o método no puede llamarse igual que una variable global o una constante.
- Control de llamadas a funciones
Se realiza la comprobación de que en las llamadas a las funciones el número de parámetros que se pasan son el número establecido en la definición.
- Control de estructuración del acceso a los arrays
Se realiza la comprobación de que el acceso a los arrays es correcto, comprobando que el número de direcciones en la definición, es similar al número de dimensiones utilizadas en la utilización de la variable.

6.2 Decisiones de diseño

6.2.1 Tabla de símbolos

Para mantener la información y realizar un correcto proceso de análisis semántico, se ha creado una tabla de símbolos que almacena información de las funciones de la aplicación así como de sus variables y sus constantes. La estructura de la tabla de símbolos se compone de dos elementos básicos:

- Los símbolos que se corresponde con las variables que representan a las funciones, donde la primera función es siempre el global y se crea antes de que comience el proceso de análisis, se utiliza para el almacenamiento de las variables globales y de las constantes.
 - Nombre: Almacena el nombre de la función, existe un nodo especial denominado globals sobre el que se almacena la información referente a las variables globales y a las constantes.
 - Tipo: Almacena el tipo de la función que puede ser desde un tipo de dato de los permitidos por en el lenguaje de entrada, la lista de tipos se encuentra en el [apartado 4.3.1](#) del analizador léxico.
 - Variables: Almacena una lista de variables con las variables de la función.
 - Tamano: Almacena el tamaño de la función para calcular la cantidad de memoria que tiene que ser reservada en la pila, además del tamaño de las variables y del retorno, se añaden 4 bytes más para el almacenamiento del valor del registro \$ra.
 - tamUltimo: Almacena el tamaño de la última variable insertada en la tabla de símbolos para la función.
- Las variables, cada función posee un conjunto de variables entre las cuales pueden haber argumentos y variables locales.
 - Nombre: Almacena el nombre de la variable.
 - Tipo: Almacena el tipo de la variable, que puede ser desde un tipo de dato de los permitidos por en el lenguaje de entrada, la lista de tipos se encuentra en el [apartado 4.3.1](#) del analizador léxico.

- **Ámbito:** Almacena el ámbito de la variable donde los valores posibles son los siguiente:
 - 0: Constante
 - 1: Local
 - 2: Global
 - 3: Parametro
- **Tamano:** Almacena el tamaño del array en el caso de que la variable sea un array, sino almacena 0.
- **Valor:** Almacena el valor para las variable donde ámbito es 0 (Constante),
- **Dimensiones:** Almacena un puntero a la lista de dimensiones en el caso de que la variable sea de tipo Array, sino su valor es NULL.
- **Offset:** Almacena el valor en pila de la variable, su valor se expresa en número de bytes.

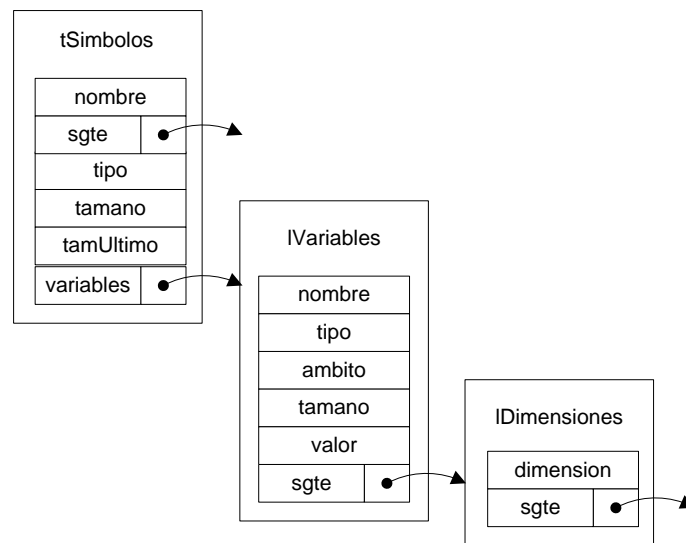


Ilustración 26 - Estructura de la tabla de símbolos

- Las dimensiones, en el caso de que una variable sea de tipo array se deberá crear un lista de dimensiones para almacenar la longitud de cada uno de las dimensiones, la elección de utilizar una lista es debido a dos motivos.
 - Nos permite almacenar las dimensiones de los arrays de forma similar independientemente de su número de dimensiones.

- Nos permite recorrer la lista de dimensiones de forma sencilla para aplicar el algoritmo de cálculo de dirección de memoria descrito en el [apartado 7.7](#) de este documento.

La tabla de símbolos no solo almacena información referente a las variables de las funciones que forma el programa sino que también almacenan información referente a las variables globales esto se realiza en un nodo especial que se crea al comienzo del programa y que se denomina global en el se insertan todas las variables globales y constantes del programa.

6.3 Problemas encontrados

Comprobación de tipos

El principal problema que nos surgió en el sistema de análisis semántico fue el proceso de comprobación de tipos ya que para cada variable debía de comprobarse dos cosas:

- En las asignaciones si los valores asignados se correspondían con el tipo con el que había sido definida la variable.
- En las sentencias en la que era utilizada si su tipo se correspondía con el tipo de los demás operadores con los que estaba interactuando

Para poder solventar de forma sencilla este proceso se le dio un valor específico a cada tipo de dato con un orden específico de forma que al comparar ese tipo supiéramos si los valores podían ser utilizados de forma correcta por lo tanto se creo un jerarquía con los distintos tipos de datos primitivos que podían interactuar entre si.

- Flotantes 3
- Enteros 2
- Enteros cortos 1
- Chars 0

Mediante esta correspondencia de valor un variable de un determinado tipo solo podía interactuar con variables cuyo tipo fuera menor o igual al suyo, de forma que un entero solo podía operar con entero, enteros cortos y chars.

Gestión de variables especiales

Uno de los problemas que nos surgió a la hora de realizar pruebas complejas sobre el análisis semántico y tras la construcción de la tabla de símbolos, fue que no se almacenaban en ningún lugar aquellas variables que no pertenecían a una determinada función, y lo mismo ocurría con las constantes las cuales solo eran globales. Para solucionar este problema se creo un nodo especial llamado global el cual almacena la información de constantes y variables globales en la tabla de símbolos.

6.4 Control de errores

En esta apartado se presenta una tabla con todos los errores semánticos que el compilador es capaz de detectar para cada uno de ellos también se indica el valor que devolverá en caso que de este compilador fuera llamado desde otra aplicación.

Código de Error	Mensaje
-2	
-3	La constante ya ha sido definida
-4	Variable global ya definida
-5	Falta el retorno de la función
-6	Variable global ya definida
-7	Función ya definida
-8	La constante no es de tipo entero. (Arrays)
-9	La variable no es de tipo entero. (Arrays)
-10	La contante o variable no está definida. (Arrays)
-11	Solo son validas posiciones enteras. (Arrays)
-12	Solo son validas posiciones enteras. (Arrays)
-13	Variable ya definida.
-14	Variable ya definida.
-15	Variable no definida.
-16	Utilizacion de array sin seleccionar posición. (Arrays)
-17	Variable no definida.
-18	Variable numero de dimensiones incorrecto. (Arrays)

Código de Error	Mensaje
-19	Selección de posición en una variable de tipo distinto Array
-20	Operación no permitida en bucle for.
-21	Operación no permitida en bucle for.
-22	Operación no permitida en bucle for.
-23	Error de tipos.
-24	Error de tipos.
-25	Variable global ya definida. (Funciones)
-26	Variable local ya definida. (Funciones)
-27	Perdida de precisión numérica.
-28	Asignación de char o void incorrecta,
-29	Variable no definida. (FOR)
-30	Utilización de array sin seleccionar posición. (FOR)
-31	Asignación de char y void incorrecta (FOR)
-32	Perdida de precisión numérica. (FOR)
-33	Variable no definida (FOR)
-34	Utilización de array sin seleccionar posición. (FOR)
-35	Asignación de char y void incorrecta (FOR)
-37	Asignación de char y void incorrecta (FOR)
-38	Variable no definida (ENTRADA/SALIDA)
-39	Variable no definida (ENTRADA/SALIDA)
-40	La constante no puede ser escrito (ENTRADA/SALIDA)
-41	Función Inexistente (Llamada a función)
-42	Faltan argumentos (Llamada a función)
-43	Número excesivo de argumentos (Llamada a función)
-44	Tipos incompatibles (Llamada a función)
-45	Perdida de precisión en la llamada (Llamada a función)
-46	Función Inexistente (Llamada a función)
-47	Número excesivo de argumentos (Llamada a función)
-48	Tipos incompatibles (Llamada a función)
-49	Perdida de precisión en la llamada (Llamada a función)
-50	Función Inexistente (Llamada a función)

Código de Error	Mensaje
-51	No se permiten variable globales como argumento de llamadas a función
-52	La función es de tipo Void (Sentencia return)
-53	Tipos incompatibles (Sentencia return)
-54	Perdida de precisión en la asignación (Sentencia return)
-55	No se pueden retornar variable globales en una función (Sentencia return)
-56	La función debe retornar un valor (Funciones)

Tabla 16 - Tabla de errores semánticos

Como se puede observar en la tabla presentada en ese apartado hay errores semánticos que poseen cada retorno, esto no es algo elegido al azar sino realizado adrede ya que dependiendo del valor de retorno se indica que el error se ha producido en un determinado lugar concreto de la gramática.

7 Generación de código intermedio

El tratamiento que hacemos del código intermedio en el compilador, no consiste en generar código propiamente dicho (de tres o cuatro direcciones), sino que consiste en plantear una serie de estructuras de datos con significado semántico que permiten hacer todas las transformaciones necesarias para pasar de código de alto nivel, a estados en los que poder generar de forma fácil e inmediata código objeto (en este caso ensamblador del MIPS R3000).

Estas estructuras generalmente son listas enlazadas, pilas y árboles binarios (tratados como árboles de expresión para ser más concretos). De hecho, un árbol binario de expresiones es totalmente equiparable en cuanto a contenido y semántica a código intermedio escrito como tal.

7.1 Generación de expresiones

Las expresiones tal cual son leídas por la gramática de Yacc, no son apropiadas para la generación de código por varios motivos. En primer lugar se hacen reducciones parciales de miembros de modo que se tiene una idea global de la expresión a la que el compilador se enfrenta. En segundo lugar, por no saber realmente el orden real de operaciones, debido no sólo a la precedencia de operadores, sino también a la presencia de paréntesis, no se puede hacer un uso adecuado de registros. Esto supone derrocharlos y abusar de instrucciones de carga/almacenamiento en pila, que sólo ralentizan la ejecución de los programas (en una computadora real, no en un simulador como es el caso).

Por tanto, en vez de tratar la expresión conforme es reducida por Yacc, lo que el compilador hará es ir añadiendo los tokens reconocidos una lista, de modo que al terminar de reconocerse la expresión (a modo de ejemplo y por simplificar, hablaremos de asignaciones) tendrá una lista con los elementos que componían dicha expresión. En esta lista cada nodo representará o bien paréntesis (izquierdo o derecho), operandos (matemáticos, de comparación o lógicos) u operadores (variables o literales).

Esta lista será la estructura de datos básica que nos permita realizar los procesos y transformaciones pertinentes para poder tratar la expresión de forma fácil y eficiente.

7.1.1 Transformación a forma polaca inversa

El objetivo de esta transformación, es poder obtener a partir de la lista de expresión original un formato que permita generar árboles de expresión binarios, los cuales serán usados como elementos base de generación de código objeto.

La forma postfija o polaca inversa nos permite representar esos árboles binarios debido a que en un árbol binario de expresión, el nodo padre representa el operador a aplicar sobre los dos hijos (operandos). En la forma postfija, aparecen los operandos que participan en una operación (dos a dos) seguidos por el operador que actúa sobre ellos, a modo de ejemplo la expresión “a + 2” (en notación infija o convencional) sería representada como “a 2 +” en notación postfija. De modo que como puede observarse, hay una correspondencia directa entre el árbol binario y la notación postfija basada en operaciones binarias recursivas.

Una característica muy interesante de este formato, es que la precedencia de las operaciones está intrínsecamente asociada a cómo se añaden los operadores en la forma polaca inversa haciendo que nunca deba plantearse el problema de qué debe ejecutarse primero. Otra característica no menos importante, es que en este tipo de notación no hay paréntesis, al igual que sucede con los operadores, la secuencia de qué operador actúa sobre qué subexpresión está es inherente a la forma de construir la expresión polaca inversa, por lo que estos símbolos no son necesarios.

7.1.2 Algoritmo RPN para generación de expresiones postfijas

El algoritmo que nos permite pasar de una lista en notación infija a postfija o polaca inversa, es el denominado RPN (Reverse Polish notation). El algoritmo para su funcionamiento requiere de una estructura de pila auxiliar. El funcionamiento del algoritmo es el que se describe a continuación:

```

Mientras lista_e != NULL =>

    Leer elemento lista_e

    Si elemento es operando => añadir elemento a lista_rpn

    Si elemento es ( => apilar (

    Si elemento es ) =>

        Mientras pila no vacía y cima de pila != ( =>

            Desapilar elemento y añadir a lista_rpn

        Si elemento es ( => Desapilar elemento

    Si elemento es operador =>

        Mientras pila no vacía y cima de pila sea
        operador de menor precedencia que el leído

            Desapilar elemento y añadir a lista_rpn

        Apilar operador

Mientras pila no vacía =>

    Desapilar elemento y añadir a lista_rpn
    
```

Este pseudocódigo planteado permite, partiendo de una lista de expresión y usando una pila auxiliar, generar un lista_rpn que contendrá la expresión en notación polaca inversa. Para entender mejor ciertas partes, se dirá que operando es cualquier literal numérico o variable del código, que los operadores son suma, resta, división, multiplicación y módulo, que la precedencia de operadores hace referencia a qué se opera primero (primero los *,/ y % y luego + y -).

A modo de ejemplo se va a plantear una expresión relativamente compleja y su equivalente en forma postfija. Sólo habría que seguir el pseudocódigo anterior para entender el proceso que genera ese formato de expresión.

$$(A + B - C + D * A / ((D + H) * (D - H))) + B$$

Esta expresión convencional, difícil de seguir de por sí por la ausencia de paréntesis en ciertas de sus partes, se convertiría en la siguiente expresión de forma polaca inversa:

$$A B + C - D A * D H + D H - * / + B +$$

7.1.3 Ventajas de la forma polaca inversa

Aparentemente no se ha ganado mucho, pero siguiendo una simple pauta se resolverá la expresión sin problemas. La idea es si leo un operando lo apilo, si leo un operador desapilo dos operandos, los opero y apilo el resultado. Con ese simple procedimiento se podría evaluar fácilmente la expresión.

Puesto que la forma polaca inversa con ayuda de una pila de evaluación, se comporta bien como estructura de código intermedio, por qué usar árboles binarios que representan lo mismo. Aparentemente es una pérdida de tiempo sin sentido, pero no lo es y por dos motivos de mucho peso: el primero es que si se piensa con perspectiva uno se dará cuenta que los árboles funcionarán mucho mejor y más fácilmente para los casos condicionales. En segundo lugar y la más importante de las razones, es que si se trata la expresión en un árbol puedo subir recursivamente hacia arriba los resultados parciales de los subárboles de modo que se hace una utilización óptima de los registros.

7.1.4 Creación de árboles binarios

Como ya se ha mencionado en el apartado anterior, la forma polaca inversa por sí misma es útil, pero no óptima para hacer un tratamiento de expresiones. Por lo que para la generación de ensamblador y por tanto ser usado a modo de código intermedio se usarán árboles binarios de expresión.

Como ya se ha dicho, un árbol binario de expresión es una forma estructurada de declarar código semejante al de tres o cuatro direcciones, donde los operandos son los elementos de la operación, los operadores la operación a realizar en sí y el nodo

padre (que es el operador binario) será usado como elemento temporal para almacenar el resultado de la operación.

El proceso para generar un árbol binario de expresión a partir de una lista en forma polaca inversa es el que se describe a continuación:

```
Mientras lista_rpn no vacía
    Extraemos elemento de lista_rpn
    Si elemento es operando
        Apilamos elemento
    Si elemento es operador
        Creamos árbol con raíz elemento
        Desapilamos y añadimos como hijo derecho
        Desapilamos y añadimos como hijo izquierdo
        Apilamos el nuevo árbol
```

De este modo se obtiene un árbol de expresión binario que podrá ser recorrido y usado para generar código y además hacerlo de forma optimizada.

A modo de ejemplo se adjunta una imagen que representa el proceso de generación de un árbol binario de expresión a partir de la siguiente lista en forma polaca inversa: A B + C * D +

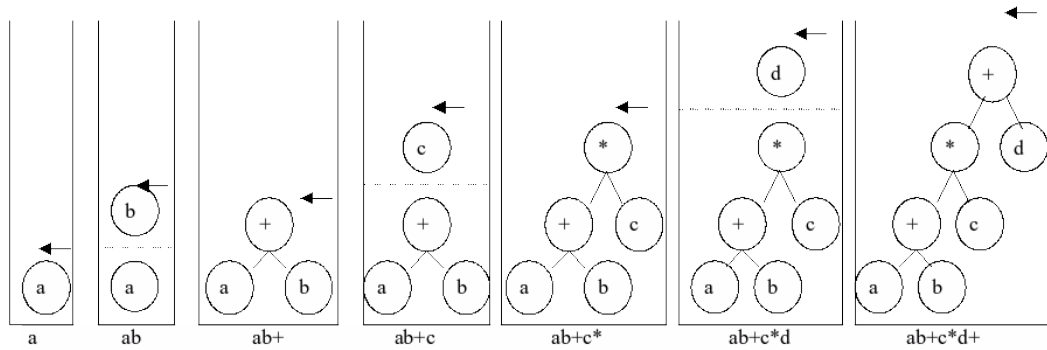


Ilustración 27 - Proceso de generación de un árbol binario de operaciones

Como se observa en la imagen desde el primer caso de la izquierda (primera inserción de un elemento), hasta el último caso de la derecha (árbol completo) se observa el proceso de generación completo para cada nuevo nodo leído.

Además se observa la equivalencia del árbol con la forma polaca inversa, ya que si recorremos el árbol recursivamente en postorden (nodo raíz, hijo izquierdo, hijo derecho) obtenemos el mismo resultado que recorrer de izquierda a derecha la expresión postfija original.

Se puede observar también, como de forma fácil y sencilla se puede ir subiendo recursivamente hacia arriba trabajando las diferentes expresiones. Por ejemplo, podría verse enfocada a la generación de código del siguiente modo tal y como se describe en el apartado siguiente.

7.1.5 Evaluación de árboles binarios de expresiones

La evaluación de estos árboles está enfocada a la generación de código y podría verse como el último paso de código intermedio antes de la generación final de código objeto.

Para evaluar un árbol binario de expresión, hay que recorrerlo recursivamente en postorden del siguiente modo: en primer lugar llamamos recursivamente al hijo izquierdo del árbol, luego al hijo derecho y finalmente realizamos operaciones, o lo que es lo mismo, llamamos al nodo raíz. Con lo que obtenemos un acceso en postfija al árbol.

El tratamiento que se hace durante el recorrido del árbol sería el siguiente:

Leo el contenido del nodo

Si es un operando =>

Genero instrucciones para cargar elemento

Si es un operador =>

Opero con hijo izquierdo miOperador hijo derecho

Borro hijo izquierdo

Borro hijo derecho

Modifico mi contenido con el resultado

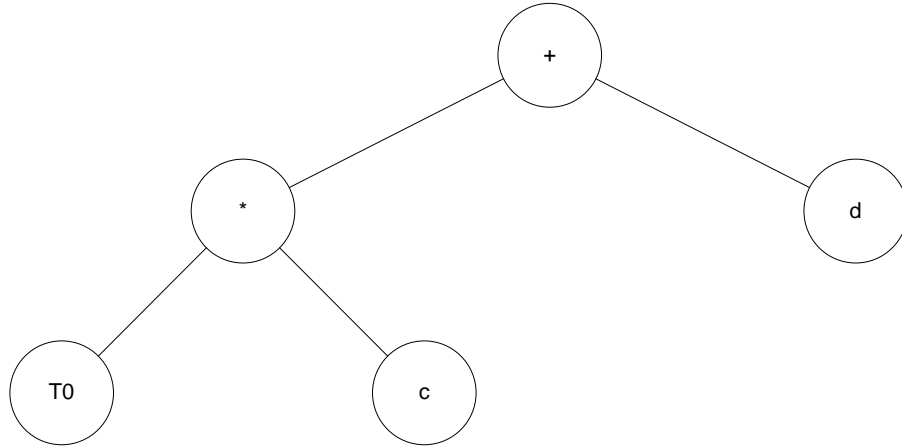
Con este proceso se evalúa el árbol o como está orientado el código, se generaría código objeto del MIPS R3000.

En el ejemplo anteriormente descrito (el de generación de un árbol desde lista postfija) el proceso sería el siguiente:

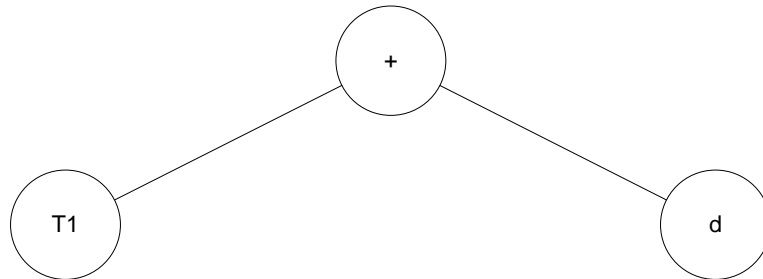
1. Cargo "a" en un registro si no lo está ya.
2. Cargo "b" en un registro si no lo está ya.
3. Opero con "+" "a" y "b".
4. Elimino "a".
5. Elimino "b".
6. Modifico "+" por "temp_0".
7. Cargo "c" en un registro si no lo está ya.
8. Opero con "*" "temp_0" "c"
9. Elimino "temp_0"
10. Elimino "c"
11. Modifico "*" por "temp_1"
12. Cargo "d" en un registro si no lo está ya.
13. Opero con "+" "temp_1" y "d".
14. Elimino "temp_1".
15. Elimino "d".

16. Modifico “+” por “temp_2”.

De forma gráfica resumida, el árbol al terminar el paso 6 estaría del siguiente modo:



Al terminar el paso 11, este sería el aspecto del árbol:



Y finalmente al finalizar la evaluación (T2 contendría la evaluación):

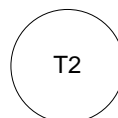


Ilustración 28 - Proceso de tratamiento del árbol de expresiones

Teniendo finalmente un árbol con un único nodo que contiene temp_2 que es el auxiliar cuyo valor es la resolución de la expresión.

Al realizar las operaciones de forma binaria de un operador con sus dos hijos (operandos), se puede hacer de forma fácil una reserva de espacio para el tipo de dato destino. Esto quiere decir, que si los operandos son de tipo entero, el resultado será entero. Si los dos operandos son de tipo flotante, el resultado será flotante, pero en el caso de tener uno de tipo entero y otro de tipo flotante, el resultado de la operación será flotante, lo que supone que habrá que transformar el entero a flotante y una vez hecho operar con los dos operandos.

Si sustituimos los procesos de operar, por generar instrucciones del tipo concreto y llevando un control de los registros que contienen los temporales, se puede perfectamente generar código objeto a partir del árbol de expresión. Además como se verá más adelante, este proceso me permite hacer una optimización bastante buena de los registros. Es más, la optimización es tan buena que por grande que sea la expresión a evaluar, con un número muy reducido de registros y sin hacer uso nunca de la pila, podemos evaluar perfectamente la expresión.

7.1.6 Tratamiento de árboles de expresión

Los árboles condicionales son, a nivel general, un tipo prácticamente idéntico a los árboles de expresión, de hecho la única diferencia significativa es que en vez de operadores matemáticos, aparecen operadores condicionales y lógicos.

El proceso de generación de árboles condicionales es exactamente el mismo que los árboles de expresión, en primer lugar se genera una lista de elementos, luego se transforma a forma polaca inversa y finalmente se genera con dicha lista el árbol.

Estos árboles de expresión condicionales son usados para resolver todo tipo de saltos condicionales, esto significa que serán usados del mismo modo en sentencias condicionales (del tipo if, if-else) o en bucles (for o while).

Las diferencias más importantes entre estos árboles y los de expresión radican en primer lugar en que en los árboles condicionales puede haber operadores unarios. Concretamente el operador negación "!". El tratamiento de este operador no puede realizarse del mismo modo que el resto en el árbol, ya que el árbol sólo funciona de forma correcta con operadores binarios (requieren de dos elementos para operarlos).

Esto supone que hay que tratar de forma especial este operador. Concretamente lo que se hace es a la hora de generar un subárbol, si el elemento leído de la forma postfija es un operador de negación, se invierten todos los símbolos lógicos del subárbol de la pila. Los operadores “|” pasarán a “&” y viceversa. En el caso de los operadores condicionales, se hace nuevamente una sustitución por el opuesto, de modo que al “<” le corresponderá el “>=”, al “==” el “!=” y al “>” el “<=”. Con eso se consigue invertir los casos en los que el árbol era cierto, que no es otra cosa que negar el árbol de expresión.

Otra diferencia sustancial en el tratamiento de los árboles condicionales, es que no se va recorriendo el árbol hacia arriba pasando el resultado al operando padre. Estos árboles sólo sirven para generar código condicional, no requieren saber el tamaño de la expresión resultante o que el resultado de una operación sea tenido en cuenta por nodos superiores. En lugar de eso, sólo se operan los subárboles cuyos nodos son hojas (no tienen hijos) que se corresponden con los operadores expresados en la sentencia condicional. Para tener en cuenta todos los símbolos lógicos que tienen por encima y generar código correctamente evaluable en estos árboles, se necesitan una serie de elementos asociados a los nodos además del propio contenido. Estos elementos son las etiquetas asociadas a una expresión y las etiquetas de salto en caso de cumplirse y en caso de no cumplirse el subárbol.

Por tanto, el algoritmo de tratar estos árboles tiene dos pasadas que se describirán a continuación:

En la primera pasada se le aplicarán las etiquetas al árbol, las de los tres tipos y además se realizará un proceso denominado de “aplanamiento de etiquetas”. Este proceso se desarrolla del siguiente modo:

Se recorre el árbol en inorden (nodo actual, hijo derecho e hijo izquierdo) y se le aplica como valor de etiqueta el ordinal que toque en la secuencia unívoca de etiquetas en el texto. Sólo se le aplica etiqueta al nodo si no es una hoja, salvo el caso de operador unario en hoja. Un operador es unario en hoja, si evidentemente es una hoja cuyo contenido no es un operador y su padre es un operador lógico.

Una vez se ha aplicado la etiqueta correspondiente al nodo (en caso de necesitarla), se hace el proceso de marcar las etiquetas de éxito y fracaso del árbol. En este caso se le aplican a todos los nodos que tengan etiqueta. Para agilizar el proceso lo que se hace es lo siguiente: si soy el nodo raíz, mi etiqueta de éxito es el éxito absoluto de la expresión y mi etiqueta de fallo el fallo absoluto. En los demás casos se aplica la siguiente regla: Si mi padre es "&" y yo soy su hijo izquierdo, mi etiqueta de éxito es un puntero al valor de la etiqueta del hijo derecho de mi padre y mi etiqueta de fallo es un puntero a la etiqueta de fallo de mi padre. En el caso de que mi padre sea un "|" y yo sea hijo izquierdo, mi etiqueta de éxito es un puntero a la de éxito de mi padre y la etiqueta de fallo es un puntero a la etiqueta del hijo derecho de mi padre. En caso de ser hijo derecho, mi etiqueta de éxito, es un puntero al éxito de mi padre e igualmente la de error es un puntero al error de mi padre, tanto si mi padre es un "&" o un "|".

Tras finalizar la recursividad y en el proceso que ésta realiza de vuelta atrás, se realiza el aplanamiento. Esto consiste en que el "padre" hereda el identificador de etiqueta del hijo izquierdo en caso de que éste la tenga, en otro caso conserva la suya. Esto es importante porque este proceso me aplana el árbol de modo que cuando tengo que saltar en una rama, no sé a qué rama debo ir, ahora en su lugar, se directamente la condición del subárbol que debo tratar. Cosa que es mucho más interesante de cara a la generación de código.

Para expresar todo esto de forma algo más clara a continuación se adjuntan una serie de diagramas de árbol que representan como se va transformando dicho árbol durante el transcurso del algoritmo, para la siguiente expresión lógica:

$$((a < b) \ \& \ (((a > b) \ \& \ c) \ | \ (s == a)))$$

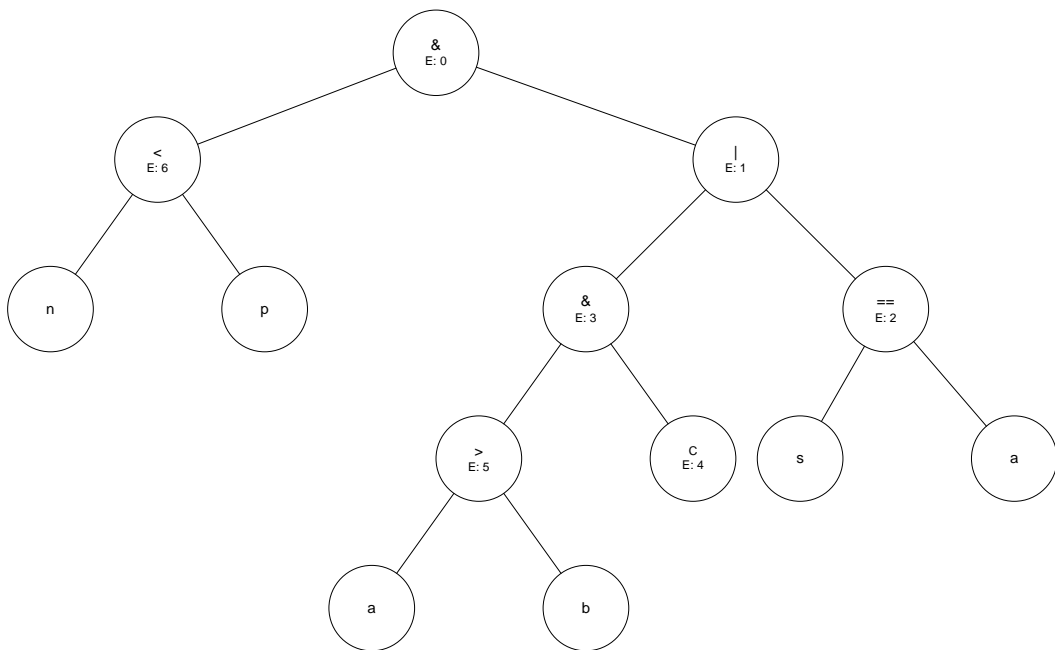


Ilustración 29 - Árbol de expresiones previo al proceso de etiquetado

En este primer árbol se observa la disposición espacial de la expresión una vez puesta en forma de árbol y además se han añadido las etiquetas primeras para dicho árbol suponiendo que el proceso global de nombrado de etiquetas comenzara en 0.

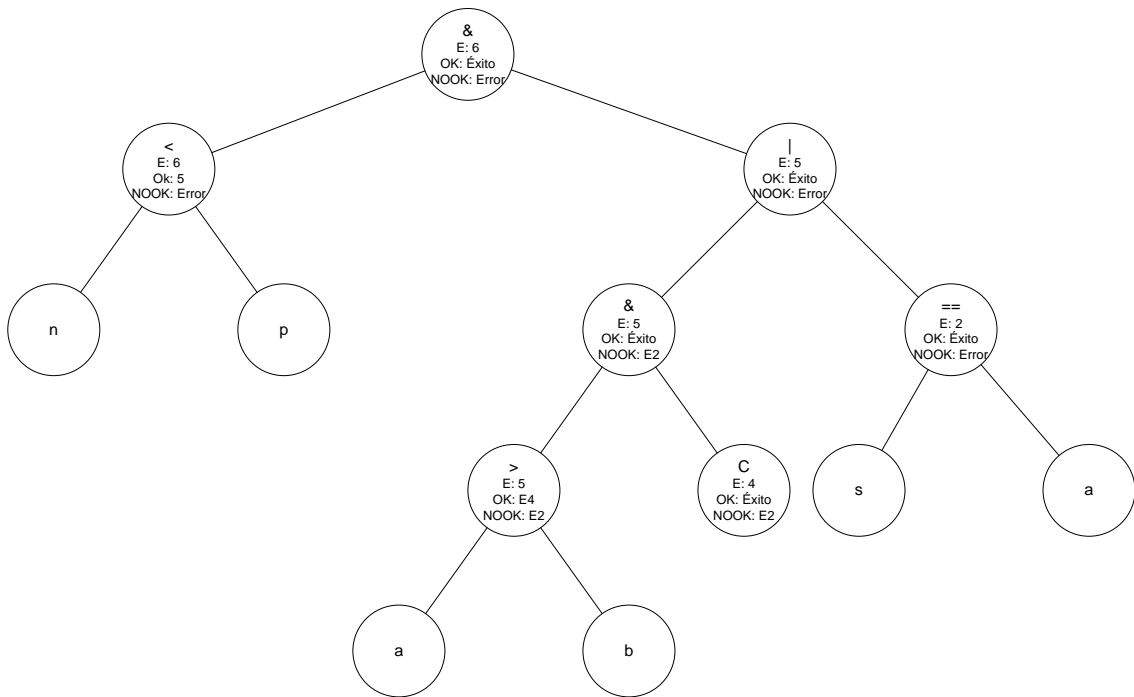


Ilustración 30 - Árbol de expresiones etiquetado

En este árbol se observa cómo se han aplanado la etiquetas (el padre herede su hijo izquierdo si éste tiene etiqueta). Además se ha añadido el valor del puntero de las etiquetas OK y NOOK, o lo que es lo mismo éxito y error del subárbol.

La forma esencial de uso del árbol sería la siguiente: si yo evalúo el subárbol de más a la izquierda ($n < p$) y se cumple, la siguiente expresión a evaluar es aquella que contiene la etiqueta 5 y puesto que solo se tratan las expresiones, se sabe que la etiqueta 5 corresponderá a la condición ($a > p$). En caso de no cumplirse es un error absoluto y salto al fin del if. Esto se ve claramente ya que al ser mi padre un & y haber fallado su parte izquierda la condición general ya no puede ser cierta.

El algoritmo para el tratamiento del árbol es en sí algo diferente de lo mencionado. Consistiría en los siguientes pasos:

- Inserto la etiqueta de la operación que hay que realizar.
- Cargo los elementos necesarios para la operación lógica.
- Genero la expresión lógica inversa a la que marca la operación.
- Genero como etiqueta de salto, la etiqueta NOOK del operador de la operación
- Luego repito el proceso para la siguiente expresión a evaluar.

Hay dos salvedades a este proceso que son:

- Si la operación a evaluar es un operando solo, en ese caso genero la operación inversa a $!= 0$.
- Si soy hijo izquierdo, mi padre es un | y soy un operador, entonces es una ramificación para evitar evaluar la parte derecha de un | si la izquierda es verdadera.

De modo que con este proceso la lógica de los operadores lógicos está implícita en el orden de los saltos y el marcado de etiquetas y además con el tipo de lógica inversa elegida se genera un código sin apenas ramificaciones incondicionales, quedando, por tanto, un código mucho más eficiente.

El pseudocódigo de tratamiento del árbol anterior tratado con estos algoritmos quedaría:

1. E6: si $n \geq p$ salta a Error
2. E5: si $a \leq b$ salta a E2
3. E4: si $c == 0$ salta a E2
4. Salto incondicional a Éxito
5. E2: si $s \neq a$ salta a Error
6. Éxito: ...
7. Error: ...

7.2 Optimización del uso de registros

Debido a que MIPS tiene 32 registros enteros, de los cuales por convenio sólo se usaran 18 (los $\$tx$ y los $\$sx$) y 32 registros flotantes y teniendo en cuenta que las operaciones sobre registros son las más eficientes que hay (en comparación con usar la pila o las cargas / almacenamientos), debemos optimizar el uso de los registros para así obtener un código lo más eficiente posible.

En el caso de los árboles binarios de condición, al no generarse resultados parciales no es necesaria una optimización, pero en el caso de las operaciones matemáticas o en el caso de las cargas de variables sí que es conveniente tratarlo de forma eficaz.

Para entender todos los procedimientos que se van a describir es necesario entender que el compilador tiene una tabla de registros (similar a la tabla de variables) en la que se describe qué datos tiene almacenado cada registros, si el registro está o no ocupado y el tipo de dato que ocupa el registro, además de una marca de tiempo cuyo significado se describirá posteriormente.

En el caso de las operaciones matemáticas, cuyos valores se almacenan en un registro temporal con el resultado y se suben hacia arriba en el árbol, el convenio de utilización es:

- Se marca el registro libre como ocupado y se rellenan sus campos con la información del resultado temporal.
- Una vez calculado un resultado temporal, se liberan todos los registros de los hijos que contuviesen resultados temporales (ya que no se usarán más por no realizar optimización de código generado).

- Las variables se conservan siempre como asignadas a registros para evitar así tener que hacer nuevas cargas repetidas de una variable (ya que por acceder a RAM sería una operación muy lenta). Esto se realiza también en condiciones.
- En el caso de accesos a arrays los registros auxiliares usados para calcular offsets y datos de contenido también serán liberados inmediatamente para que puedan ser reutilizados. Esto se realiza también en condiciones.

Hay que tener en cuenta que todos los resultados, registros y en general toda la información que se obtenga de una expresión lógica (como el caso de los ifs o la cabecera de los bucles) no puede ser almacenada una vez reducido su árbol y deben ser eliminados dichos registros como si de auxiliares o temporales se tratara.

Por ejemplo:

```
If ((v0 > 0) & (v1 < 0) & (v2 == v1)) | (v3 == v4))
```

En este ejemplo, al generar el árbol lógico de la expresión, habrá que generar el código necesario para cargar las variables y los datos necesarios para que se dé la ejecución que se dé, el código sea correcto y completo. Eso supone que hay que generar código para todos los posibles caminos que siga una ejecución del programa.

Al generar el árbol, el sistema tendría en su tabla (no siempre pero podría darse) registros ocupados con los valores de las variables v3 o v4, por lo que si dentro del if uso esas variables, el sistema recurriría a esos registros para optimizar (y ahorrarse accesos a memoria o a pila). Pero que sucedería si la expresión izquierda (la que está antes del OR), fuera cierta. Pues sencillamente que nunca se ejecutaría el código de la parte derecha del if por no ser necesario y los hipotéticos registros que contendrían v3 y v4 realmente no tendrían andas por tratarse de zonas de código no ejecutadas en la ejecución del programa y los usos de dichas variables en el cuerpo del if serían necesariamente erróneos.

Por tanto, todo registro, número o dato usado en el árbol de un if, debe ser desechado automáticamente como posiblemente erróneo una vez reducido un árbol lógico.

7.2.1 Algoritmo LRU

Este algoritmo servirá para decidir que debo desalojar de un registro (tanto entero como real) en caso de tener que almacenar algo y no tener ninguno disponible. El algoritmo se basa en el algoritmo usado en paginación de memorias y caches para saber que pagina debo desalojar de la RAM en caso de tener que cargar una nueva y no tener espacio. LRU significa (Least Recently Used) menos recientemente usado y consistirá en desalojar aquel dato más viejo del registro.

Antes se mencionó que todo registro dispone de un entero que es una marca temporal. Esa marca temporal es obtenida con la función `gettimeofday` que me retorna el tiempo transcurrido en segundos desde una fecha referencia hasta hoy (retorna más cosas pero sólo se usará el valor en segundos).

Como ya se mencionó, en caso de tener que almacenar algo en un registro y no tener espacio, se buscará el registro que lleve más tiempo sin ser usado (marca temporal más antigua) y se sobrescribirá con el dato que necesitamos.

El funcionamiento es el siguiente:

Cada vez que leemos un registro (su contenido) ya sea para almacenar en él un resultado, o simplemente leamos del registro su valor, actualizamos su marca temporal (haciéndose reciente).

Cuando no tenemos espacio para almacenar un dato, buscamos en los registros aquel que tenga la marca temporal más antigua y lo sobrescribimos con la nueva información.

Este algoritmo en teoría nos ayuda a manejar mejor los registros, de modo que se minimicen las cargas de variables basándonos en la máxima de que “si lleva mucho tiempo sin usarse, no tiene porque usarse inmediatamente”.

Como todos los algoritmos de sustitución no nos garantiza optimizad, pero en general se obtiene buenos resultados con su uso.

7.3 Tabla de registros

La tabla de registros, como ya se ha dicho durante esta memoria, es una tabla cuyo significado es similar al de la tabla de símbolos, pero encaminada al uso, optimización y manejo de los registros para una correcta generación del código ensamblador.

La tabla de registros es una array, con posiciones lógicas que representan intervalos donde se definen cada uno de los tipos de registros presentes en el procesador MIPS R3000. Por tanto, el array se fragmenta de forma lógica en secciones para enteros, sección para flotantes, para registros espaciales (como lo de Entrada/Salida), etc.

Cada elemento de cada uno de esos arrays, referencia a un registro concreto del procesador MIPS R3000, de modo que, por ejemplo, el primer elemento del array de los enteros usados para operaciones se corresponde con el registro \$t0 del procesador.

Cada uno de esos elementos está definido por una serie de atributos que contienen la información necesaria para determinar en todo momento su estado, contenido, etc. Y cualquier otro dato interesante para poder generar un código ensamblador optimizado. Dichos atributos son:

- ID: Es un identificador que determina el nombre de lo referenciado por dicho registro.
- Referencia: Es una marca temporal (con tiempo Unix) que representa la última vez que fue referenciado dicho registro, por tanto siempre que un registro es referenciado este campo temporal es actualizado (con la fecha actual en milisegundos).
- Estado: Representa el estado actual del registro, de modo que un 1 representa que el registro está actualmente en uso y 0 que está libre. Esto es necesario para no sobrescribir accidentalmente contenido de registros.
- Tipo: Tipo representa si el dato contenido en el registro concreto es una dirección de memoria (como podría ser la dirección base de un array) o si por

el contrario contiene el contenido directo de una variable. 0 representa variable y 1 dirección de memoria.

- Auxiliar: Representa si el dato almacenado en un registro es un dato final, como por ejemplo una variable o si por el contrario es un dato temporal obtenido en la reducción de un árbol matemático o condicional. Esto es útil para poder eliminar todos los datos temporales usados en una reducción de un árbol, ya que esos registros no serán útiles durante el resto de la generación del código.
- Profundidad: Representa la profundidad en la que fue usada la variable referenciada por el registro. Esto es imprescindible para el correcto funcionamiento del algoritmo de profundidad explicado más adelante.

La tabla de registros podría verse del siguiente modo:

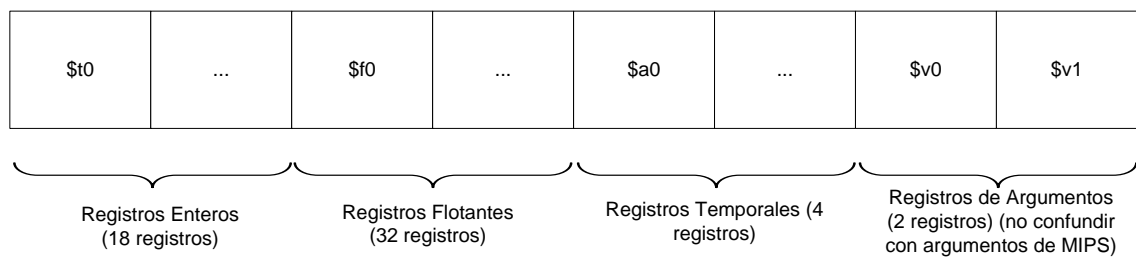


Ilustración 31 - Estructura de la tabla de registros

A modo de ejemplo, si tuviésemos almacenada en \$t0 la variable var1, habiendo sido usada en el main, dentro de un for no anidado (profundidad 1) su representación podría ser:

\$t0	
ID	var1
REFERENCIA	¿?
ESTADO	1
TIPO	0
AUXILIAR	0
PROFUNDIDAD	2

Ilustración 32 - Estructura de un registro de la tabla de registros

Este nodo representa que \$t0, contiene la variable var1, la fecha cuando fue usada dicha variable por última vez, que su estado es “en uso” que es una “variable”, que no es ningún elemento “auxiliar” y finalmente que fue usada a una profundidad 2 (la profundidad se inicia a 1, por lo que si se usó en un for no anidado su profundidad sería 2).

7.4 Tratamiento de sentencias anidadas

En la generación de código y por tanto en éste análisis intermedio que se hace de las expresiones, hay que tener sumo cuidado con qué bloque de código está contenido y en dónde. No es lo mismo ejecutar dos ifs consecutivos, que ejecutar un if si se cumplió el primero.

Es necesario para el correcto funcionamiento del código generado un uso correcto de sentencias que den la semántica de código embebido (contenido en) otro código para que respete la semántica original del código C lite escrito.

En principio, excluyendo en caso de funciones que no entra en estos tratamientos por tener su propio tratamiento independiente, un código general (aquel que se compone de todo tipo de sentencias) puede estar contenido dentro de if o un bucle (excluyendo el caso de no estar contenido). La idea para generar el anidamiento de código, es saber dónde se deben colocar las etiquetas que hacen referencia a dónde comienza y dónde termina el bloque de código embebido.

El compilador usará una pila de sentencias de finalización para ello. El funcionamiento de la pila y las etiquetas de fin para embeber código es el siguiente:

- Se genera la cabecera del primer bloque (if o bucle) y se apila su sentencia de fin.
- Se genera el código de forma convencional de lo que tenga dentro, ya sean nuevos ifs, bucles u otro tipo de sentencias.
- En caso de ser ifs o bucles, se hace lo mismo, se genera el código de forma normal y se apila su etiqueta de fin de bloque.
- En la gramática yacc, se asignan reglas que permitan reconocer los instantes donde se ha reconocido el fin de un bloque de código y es ahí donde se desopilará la etiqueta de fin de bloque y se añadirá al código.

El funcionamiento se basa en que al apilarse las etiquetas, la primera en salir corresponde a la última insertada y la última en salir la primera insertada. Con esto se crea el orden correcto de etiquetas que nos permiten embeber código dentro de ifs o bucles.

A modo de ejemplo si yo partiera de un código similar al siguiente:

```
While
```

```
    Cuerpo while
```

```
    If
```

```
        Cuerpo if
```

```

    Fin if

Fin while
    
```

El proceso generaría la cabecera lógica del while, luego apilaría la sentencia de fin de while y finalmente generaría sus instrucciones habituales. En ese punto aparece un if, repitiéndose el proceso, se genera la cabecera, se apila la etiqueta de fin y se generan sus instrucciones. Lo primero que encuentra el sistema es el fin del if, por tanto desopila la primera etiqueta de fin, que es la del if, luego aparece el fin del while y desapila su etiqueta de fin.

Con esto se ha conseguido que el if este anidado al código del while.

Con todo lo descrito anteriormente se consigue un anidamiento sintáctico del código, de modo que la ejecución por parte del procesador de ese ensamblador sería anidada, pero falta una parte esencial, el aseguramiento de la coherencia de los registros en los anidamientos. Cuando se genera el código (o el análisis intermedio que se hace en esta sección), genera cargas de elementos dentro de una sección de código diferente del flujo principal (ya sea anidado o sentencia de salto) queda reflejado en la tabla de registros, ya que si no fuera así, para cada uso de una variable en ese bloque de código habría que hacer su correspondiente load, siendo enormemente ineficiente. Pero de no hacerse nada, el código generado para una carga dentro de un if, podría afectarme negativamente al resto del código.

Para tratar el problema se usa lo que se ha denominado algoritmo profundidad, que se describirá en el siguiente apartado.

7.4.1 Algoritmo de profundidad

En un principio, el sistema de registros está pensado para usarlo masivamente, de modo que se utilicen con temporales y variables a fin de minimizar las cargas de memoria. Para ello, en la tabla de registros se marcan los registros como ocupados y además se hace referencia al dato que tienen. El problema es, como ya se ha dicho,

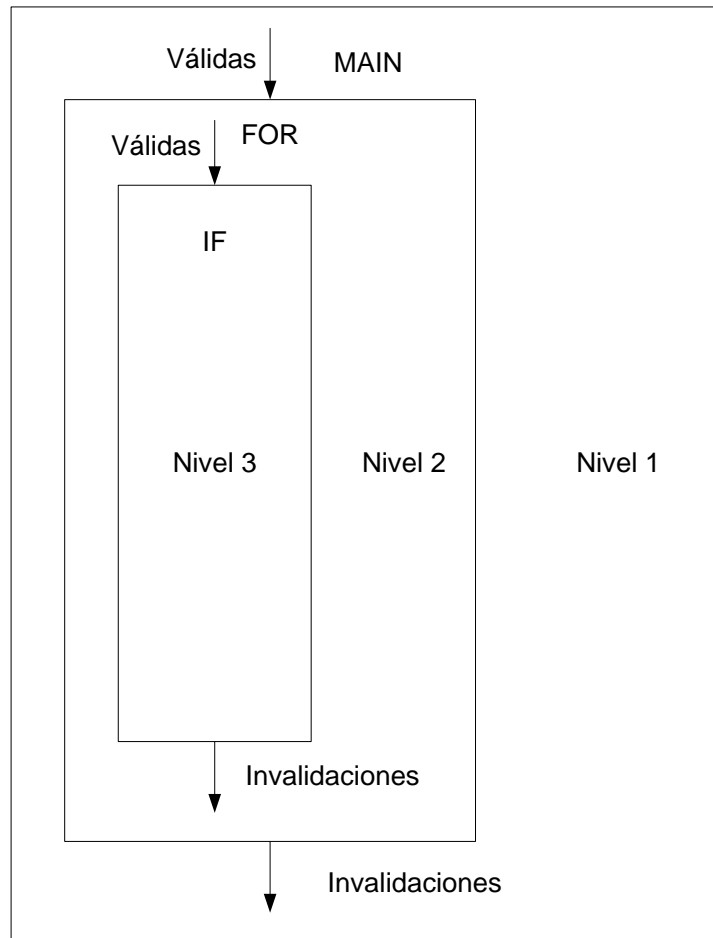
que si yo cargo una variable en un registro dentro de un if por ejemplo, pero en la ejecución del código no paso por ahí, eso queda reflejado en la tabla de registros, de modo que el siguiente uso de la variable se haría desde el registro si carga previa y sería erróneo.

El algoritmo de profundidad lo que hace básicamente es asignar la profundidad (inicialmente a 1 para el código del main) a la que el registro fue modificado, de modo que si profundidad es 1 se sabe que es tratado en el main y si es mayor entonces estará en una sentencia condicional o anidado a un nivel más profundo. Con este sistema podemos garantizar la validez de un registro siempre y cuando fuera generado en un nivel igual o menor al nuestro. Así se consigue que se acepte el código anterior, ya que se ha tenido que pasar por el necesariamente y además se vitan cargas masivas dentro de mi mismo nivel.

Finalmente para que el algoritmo funcione adecuadamente, es necesario invalidar todos los registros modificados en un nivel mayor al nuestro al regresar a nuestro nivel, ya que no podemos garantizar que ese bloque de código haya sido ejecutado por el procesador.

De este modo se consiguen hacer las cargas habituales de registros, se garantiza la consistencia e integridad de los datos almacenados y además permite anidar código hasta la profundidad que se desee.

El siguiente diagrama muestra de forma gráfica lo expresado sobre las forma de trabajar del algoritmo de profundidad:



7.5 Tratamiento de la pila

El tratamiento de la pila es la simulación que se realiza dentro de las estructuras de datos para poder manejar los datos de la pila del ensamblador fácil y ordenadamente.

En un principio, el compilador sólo hará uso de la pila para las funciones. En la pila cargará sus parámetros y sus variables locales y una vez finalizada la llamada se vaciará su parte correspondiente de la pila.

El problema se plantea cuando queremos hacer load y store ya que no se harán sobre RAM sino sobre la pila, a no ser que se trate de variables globales en cuyo caso sí que se harán sobre memoria. El problema de hacer un load o store sobre la pila, es que tengo que conocer perfectamente el offset a aplicar sobre dicha operación para

escribir o leer el resultado del segmento correcto. Para ello usamos los siguientes atajos en la tabla de símbolos:

- Toda función almacena un entero incrementado con cada inserción de variable que representa el tamaño total de las variables y argumentos de la función.
- Toda función recibe el tamaño temporal del padre a la hora de ser insertada en la tabla de símbolos más su tamaño en memoria.
- Partiendo de que una función en ejecución ocupa la última parte de la pila, que la pila crece y decrece con las llamadas y los retornos podemos garantizar que esos valores añadidos pueden ser usados como offsets para las cargas y stores de esas variables en pila.

7.6 Tratamiento de funciones

Como ya se mencionó anteriormente, las funciones tiene como principal característica que sus variables locales no se almacenan en RAM, sino que se añaden a la pila y son leídas y escritas en ella, de modo que cuando finalice la función, se libere su segmento de pila y no se ocupe ese tamaño de forma inútil.

El primer paso en la llamada a una función es por tanto reservar tamaño en pila para que pueda desarrollar la ejecución de código sin problemas. Esta reserva la realiza siempre la función invocante y para ello debe conocer el tamaño total de la función. El tamaño se calcula del siguiente modo:

$$\text{Tamaño_función} = \text{Tamaño_valor_retorno} + \text{Tamaño_argumentos_pasados} + \text{Tamaño_variables_locales} + 4$$

De modo que el valor del retorno será 0 en caso de void, 1 en caso de char, 2 en caso de short y 4 en int y float. El tamaño de los argumentos es la suma de los tamaños de todos los argumentos pasados (con los valores antes descritos). El tamaño de las variables locales sigue la misma dinámica expresada. El 4 se reserva para almacenar un

entero de 4 Bytes almacenado en el registro especial \$ra, que sirve para almacenar el puntero a la zona de memoria de la función que nos invocó (esto es útil en el caso de la recursividad o de llamadas encadenadas de funciones).

A modo de ejemplo el tamaño de:

```
int suma (int s1, int s2)
{
    int aux;

    aux = s1 + s2;

    return aux;
}
```

Sería $4 + (4 + 4) + (4) + 4 = 20$ bytes, por lo que en primer lugar la función invocante debería reservar 20 bytes en pila para la ejecución de la función.

Por convenio, se considera siempre que la primera parte de la pila está reservada para el retorno de la función, de modo que cualquier invocante sabe que el retorno lo debe recoger de esa posición, que será siempre la misma independientemente del tamaño de la función en sí.

Las siguientes posiciones de la pila están reservadas para los argumentos de la función, luego irían las variables locales y finalmente los 4 bytes de dirección de memoria de la invocación.

Una función deberá siempre acudir al desplazamiento relativo de un argumento o variable local sobre la pila para poder cargar o almacenar datos. Puesto que las reservas de pila las hace la función invocante (se reserva pila restando el puntero especial de pila), supone que no es necesario saber por dónde se ejecutará el

código, ya que pase por donde pase, se harán las reservas oportunas y los offsets relativos a la función serán válidos.

La reserva de memoria y el almacenamiento de \$ra en pila hacen posible dos escenarios:

- Llamadas encadenadas de funciones (que main llame a la función suma, ésta a obtener datos y ésta a su vez a mostrar, por ejemplo).
- Recursividad (funciones que se autollaman).

Cuando una función comienza, lo primero que hace es salvar el valor de \$ra en la última posición hábil de su pila (tamaño de función - 4). En cada return que tenga la función lo que hará será justamente restaurar en \$ra el valor almacenado en esa posición de pila. En caso de no hacerse se perdería la dirección de salto incondicional y terminaría la ejecución del programa.

Para la función suma antes descrita los offsets y valores serían los siguientes:

- En la posición pila estaría el futuro retorno de la función (escrito en el return)
- En pila + 4, estaría s1.
- En pila + 8, estaría s2
- En pila + 12, estaría aux.
- En pila + 16 estaría \$ra. Cuyo valor sería establecido al iniciar la función suma (primera instrucción) y recuperado de la pila en el return.

7.7 Tratamiento de los arrays

Los arrays son una colección ordenada de elementos que comparten tipo y por tanto tamaño. Tienen la propiedad (en nuestro sistema, no necesariamente en un entorno real) de ser un espacio de memoria consecutivo.

En general, las variables globales, constantes y variables del main se definen en el segmento de datos del programa, mientras que las variables locales a las funciones se tratan el segmento de pila. El caso de los arrays es una excepción, ya que por su especial complejidad se optó por una simplificación y pro convenio, todos los arrays de

un programa (sean locales o globales) son definidos en RAM (en el segmento de datos).

Los arrays son definidos como un espacio de memoria reservado desde una posición inicial (la primera libre para los datos) y que se propagan n bytes hasta completar su tamaño. El tamaño de un array viene definido por el producto de todas sus dimensiones (en caso de un vector su dimensión es su tamaño), multiplicado por el tamaño del tipo de dato que contiene, ocupando los char 1 bytes, los short 2 bytes y finalmente int y float 4 bytes. A modo de ejemplo una matriz de 3x4 de enteros ocuparía $3 \times 4 \times 4 = 48$ bytes.

Para acceder a las posiciones (elementos dimensionales) de un array (expresadas entre [] en el lenguaje de alto nivel), tenemos que calcular el desplazamiento de esas posiciones y sumárselo a la dirección origen en datos de dicho array. Eso nos daría la dirección de memoria que ocuparía el dato concreto al que queremos acceder. Mencionar en este punto que pese a que de forma lógica podemos tener arrays multidimensionales, la representación en memoria siempre es vectorial y el mapeo de multidimensionalidad del lenguaje de alto nivel, al acceso al vector físico debe ser controlado por el sistema. Dicho desplazamiento se puede calcular con la siguiente expresión:

$$\left(\sum_{i=0}^n \left(\text{acceso}_{i \times} \prod_{j=i}^{n-1} \text{dimension}_j \right) \right) \times \text{Tamaño del tipo de dato}$$

De modo que si tuviéramos un array de enteros de 4x4 y quisiéramos acceder a la posición [1][3], el cálculo para obtener el desplazamiento sería el siguiente:

$$((1 \times 4) + 3) \times 4 = 28 \text{ bytes.}$$

Por tanto, para acceder a la posición especificado bastaría con cargar de la memoria la dirección ocupada por la dirección base del array más esos 28 bytes de desplazamiento.

Supongamos que el anterior array de 4x4 es la siguiente matriz:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

Y que la dirección base de dicha matriz es la dirección 1000 del sistema. La representación de elementos y direcciones sería:

Datos:	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Direcciones:	1000	1004	1008	1012	1016	1020	1024	1028	1032	1036	1040	1044	1048	1052	1056	1060

De modo que la posición [1][3] es un 2 en dicha matriz (el último), que si comprobamos la representación en memoria vemos que el desplazamiento 28 mas la dirección base es 1028 y se corresponde con el último 2 de la memoria.

8 Generación de código ensamblador

El código objeto es la generación del código final del compilador. Este código en esta práctica será el ensamblador del MIPS R3000.

El ensamblador es un lenguaje próximo al código máquina, pero que sigue siendo entendible por humanos. De hecho las instrucciones de ensamblador tienen correspondencia con códigos binarios (máquina) que si serían interpretados por el hardware.

Para poder generar el código objeto, se han seguido las pautas que se marcaron en la descripción del procesador MIPS R3000, tanto arquitectónicas como funcionales.

Se ha procurado limitar otras partes del proceso de desarrollo tales como la gramática de modo que sólo permitan aquellas sentencias que pueden ser tratadas por el generador de código objeto.

Se ha procurado en la medida de lo posible abstraer todo el proceso de generación de código en funciones contenidas en la unidad transformador.c.

8.1 Decisiones de diseño

En general, se ha optado por plantear funciones que reciben como parámetros los registros implicados en la instrucción a generar y una serie de códigos de control (obtenidos del proceso y dinámica del código intermedio) de modo que permiten seleccionar que instrucción generar para esos registros en cuestión.

8.1.1 Tratamiento de registros

A la hora de generar ensamblador, es necesario tener un cierto tratamiento sobre los registros, ya que no todos son utilizables en todo momento. Para el tratamiento general del uso de registros leer el apartado de optimización de código intermedio.

A nivel de generación de código nos interesa poder obtener registros libres (acorde a todo lo mencionado) para almacenar en él resultados parciales. También nos interesa obtener si una variable está contenida en algún registro concreto y en cuál.

Por tanto, las operaciones de buscar registros libres y obtener registro con un contenido concreto serán la base de la generación de código a la hora de discernir que

registros debo pasar a las funciones de generación para así obtener instrucciones ensamblador coherentes.

8.1.2 Cabecera del programa

Es un mero formalismo con el cual se genera código ensamblador para que pueda iniciarse un programa y sea entendido adecuadamente por el simulador del R3000.

En este proceso se genera el inicio del programa, que es el segmento de texto, dejando el programa listo para iniciar el proceso de generación de instrucciones.

En MARS como ya se dijo, no se hace una llamada automática al main como en Spim, de modo que una parte importante de esta cabecera es incluir dicha llamada. Esto es importante porque sin ella el código comenzaría con la primera instrucción encontrada y en caso de no ser el main, el programa no se ejecutaría adecuadamente.

8.1.3 Generación de código de variables

El código generado de variables se corresponde con lo expresado en las constantes, variables globales a todo el programa y variables presentes en el main (no se pueden desalojar hasta el fin del programa). Las variables de las funciones como ya se dijo se encuentran en pila y no tienen tratamiento a la hora de escribir código con ellas.

Como convenio se ha decidido que toda variable comience por “_” (guión bajo) para no confundirlas con sentencias de ensamblador en ningún caso (ya que estas no empiezan nunca por “_”).

Las variables se generan añadiendo al nombre de la variable su tipo. En caso de MARS no es necesario añadirlas valor inicial, así que no se hará.

Los tipos definidos de variables son:

Tipo C	Tipo MIPS R3000
Char	.byte
Short	.half
Int	.word
Float	.float

Tipo C	Tipo MIPS R3000
Array	.space

Tabla 17 - Correspondencia de tipos entre R3000 y MIPS

8.1.4 Generación de instrucciones de

Las instrucciones de carga en MIPS se componen básicamente del tipo de carga concreta a realizar, el registro sobre el que se quiere cargar y una dirección de memoria (generalmente una etiqueta de una variable), que representa la dirección de donde se pretende cargar. Las cargas se realizan por tanto de memoria RAM, pudiendo ser de una variable del segmento de datos o de la pila.

El proceso de selección de registro destino se realiza tal cual se expresó en apartados anteriores. Los códigos necesarios para generar la instrucción salen de la lógica del código intermedio. Generalmente viene dado por el tipo de dato a cargar, aunque en el caso de los arrays se añade la semántica de las direcciones de memoria.

Los tipos de cargas contemplados son:

Instrucción	Código de control	Significado
lb	0	Caso de cargar un char
lh	1	Caso de cargar un short
lw	2	Caso de cargar un in
l.s	3	Caso de cargar un float
li	4	Caso de cargar una cosntante
la	5	Caso de cargar una dirección. Arrys

Tabla 18 - Instrucciones de carga soportadas por el compilador R3000

8.1.5 Generación de instrucciones de almacenamiento

El procedimiento de los almacenamientos es idéntico en todos los aspectos al de las cargas. En cargas se hace un load para tener presente un valor con el que operar. Los stores se realizan en el caso de las asignaciones, para poder llevar a la RAM o a la pila un valor concreto que perdure más allá de los registros.

Los stores tienen la siguiente tabla de instrucciones (muy similar a los load):

Instrucción	Código de control	Significado
sb	0	Caso de almacenar un char
sh	1	Caso de almacenar un short
sw	2	Caso de almacenar un in
s.s	3	Caso de almacenar un float

Tabla 19 - Instrucciones de almacenamiento soportadas por el compilador R3000

En el caso de los stores, no tiene sentido almacenar un dirección de memoria o un literal, esos elementos deben ser calculados en registros y luego almacenados pero no de forma directa.

8.1.6 Generación de instrucciones de operación

Las instrucciones de operación son aquellas que se realizan acorde a una operación solicitada en el código origen. Estas instrucciones vienen a ser por norma general operaciones matemáticas. Por tanto la lógica habitual de la generación de código para expresiones de este estilo será en primer lugar obtener un registro destino para el resultado de la operación (que será normalmente un auxiliar), luego se obtendrán los registros con los operados, de modo que siga la lógica marcada por el código intermedio.

El tipo de instrucción concreta que se realizará con los dos operados y el registro de valor de retorno. La operación viene marcada por el operando del árbol de expresiones. Por tanto, se generarán códigos de operación en función del operando leído.

La tabla con los datos sobre operaciones es la siguiente:

Instrucción	Código	Descripción
add	1	Suma entera
Sub	2	Resta entera
Mult	3	Multiplicación entera
Div	4	División / módulo
Add.s	1	Suma flotante
Sub.s	2	Resta flotante
Mul.s	3	Multiplicación flotante
div.s	4	División flotante

Tabla 20 - Instrucciones de operación soportadas por el compilador R3000

En el caso de la división y multiplicación enteras, hay que hacer uso de los registros especiales de ALU HI y LO.

Hay que tener en cuenta que para operar enteros con flotantes, hay que pasar primero los enteros a registros flotante, luego se hará una conversión de dato con `cvt.s.w` posteriormente se operará con un resultado flotante y de ahí en adelante el auxiliar que contiene el resultado de la operación será considerado flotante.

8.1.7 Generación instrucciones para arrays

En el caso de los arrays, en primer lugar debemos cargar de memoria su dirección. Una vez tenemos su dirección se calcula el offset para el elemento en concreto, esto se hará tal y como se describió en la generación de código intermedio. Ese dato concreto ya puede ser cargado desde la dirección más offset a un registro concreto. Estas operaciones de cálculo de offsets, cargas y almacenamiento, se realizan con las operaciones anteriormente definidas.

8.1.8 Generación de instrucciones de salto

Las instrucciones de salto se aplicaran tanto para los bucles como para las ramificaciones condicionales. En el caso de los bucles se hará uso de la lógica expuesta en código intermedio para condiciones, añadiendo la etiqueta del bucle que me permite saltar sobre mí mismo. En las ramificaciones condicionales (los ifs) la lógica de

la generación de las instrucciones de salto, es como ya se explico, invirtiendo el signo de la operación de comparación y saltando sobre la etiqueta de error del nodo actual.

Hay que tener en cuenta, al igual que en expresiones matemáticas, que se pueden hacer comparaciones entre enteros y flotantes, por lo que será necesario hacer los movimientos de registros y las conversiones permitentes.

En el caso de los flotantes, no hay sentencias para todos los tipos de condicionales. El proceso consiste en comparar los registros flotantes y almacenar en un registro especial f un 0 si no se cumple la condición y un 1 de cumplirse. Por lo demás el planteamiento es idéntico.

La tabla de las bifurcaciones es la siguiente:

Instrucción	Código	Significado
beq	11	==
ble	12	<=
bge	13	>=
blt	14	<
bgt	15	>
bne	16	!=
beq	17	==
c.eq.s	11-16	== flotante
c.le.s	12-15	<= flotante
c.lt.s	13-14	> flotante
bc1f	13-15-16	Salto flotante en caso NOOK
bc1t	11-12-14	Salto flotante en caso OK

Tabla 21 - Instrucciones de bifurcación soportadas por el compilador R3000

8.1.9 Generación de etiquetas de salto

Hay tres tipos de etiquetas de salto, por un lado están las etiquetas de identificación de la operación (if para los if, for para los for y while para los while). Todas ellas tienen un contador iniciado en 0 y que se incrementa en el momento de la generación del árbol lógico (ya que ifs y bucles en su cabecera lo que tienen es un árbol lógico como los ya descritos en código intermedio). Estas etiquetas sólo sirven

para referenciar en el código el punto donde comienza el if o el bucle. Toda etiqueta de operación tiene su equivalente de fin operación, que nuevamente sólo tiene un significado de guía, para entender dónde comienzan y terminan las operaciones.

Las etiquetas de operación se representan como `__ifN`, `__forN`, `__whileN`, donde N es el contador de identificadores. Las de finalización como `__finifN`, `__finforN`, `__finwhileN`.

Las etiquetas intermedias de salto, necesarias para seguir la lógica compleja de las condiciones (usadas para evitar recorrer expresiones lógicas completas cuando o es necesario), son globales a todas las condiciones y se referencian como: `__eN`.

Las etiquetas de error y éxito son comunes en nombre a los tres tipos definidos (if, for y while), por tanto el contador de generación de las mismas. Representan los éxitos globales o errores para una expresión lógica completa. Estas etiquetas se denominan como `__exiton` y `__errorN`.

8.1.10 Generación de instrucciones de funciones

Las funciones tienen la peculiaridad de que sus variables locales y argumentos, se encuentran en la pila en vez de en memoria. Por tanto, necesitan cargas especiales y almacenamiento especiales sobre la pila (puntero a `$sp`). Como ya se definió en código intermedio, se usarán offset en la tabla de símbolos para saber sus direcciones relativas a la hora de generar ese tipo de instrucciones.

Es necesario, por tanto, hacer previamente un movimiento del puntero de pila por parte de la función llamante. A continuación la función llamante carga los argumentos en la pila para que los pueda leer la función.

Hay un convenio entre función llamante y función llamada para el caso de los return. La función llamada siempre retorna el valor de retorno en su primera posición relativa de pila.

Una vez se termina la llamada a la función, la función llamante recoge el retorno y restaura el puntero de la pila.

8.1.11 Entrada/Salida

Para el caso de la salida, se optó por una dinámica similar a la de Java, que consiste en poner en una llamada a la función todos los elementos que se quieren imprimir, siendo el propio compilador el que determine el tipo de salida en función del tipo de dato que representa.

El caso de la entrada es, sin embargo, más similar a C, de modo que se pasa como parámetro de la función de entrada aquella variable que se desea leer. Nuevamente será el compilador el que se encargue de los trámites del tipo de entrada.

Los tipos, tanto para entrada como para salida, se resuelven comprobando en la tabla de símbolos el tipo concreto del dato a imprimir por pantalla o a leer de teclado.

El tratamiento de las operaciones de entrada es sencillo, se carga en un registro especial (\$v0) una constante (definido en la descripción del procesador MIPS). En función de esa constante (calculada a partir del dato que se quiere mostrar o leer) el sistema hará una operación u otra.

Para la impresión de datos, el contenido a imprimir debe estar almacenado en \$a0 si es entero o en \$f12 en caso de ser flotante.

En el caso de la lectura, será en \$v0 donde se almacene el valor leído por teclado en caso de ser un entero o \$f0 en caso de flotante.

Todas las llamadas al sistema se realizan por medio de la llamada syscall.

9 Otros elementos

9.1 Descripción del sistema de ejecución

El sistema de ejecución de la compilador r3000, es el conjunto de opciones que el compilador permite a la hora de ser ejecuta por línea de comando, a continuación se indica la sintaxis para poder ejecutar el compilar y la descripción detalla de cada uno de sus elementos.

```
r3000 [-t -help] fichero_entrada.c [-r fichero_salida.s]
```

Ilustración 33 - Sistema de ejecución del compilador

- -t
Indica al compilador que deben mostrarse la trazas de ejecución del proceso por la salida estándar.
- -help
Muestra la ayuda del compilar, indicándose las opciones y para que sirve cada una de ellas.
- fichero_entrada.txt
Indica el fichero de entrada donde se encuentra el código c que debe ser compilador, es el único parámetro obligatorio.
- -r
Indica que el resultado de la compilación debe ser almacenado en un archivo indicado por el usuario, que deberá ser indicado a continuación, en caso de que no se seleccione esta opción el fichero de salida tendrá el mismo nombre que el de entrada pero con extensión s.
- fichero_salida.s
Indica que nombre del fichero de salida de la aplicación, este debe estar formado por un nombre de al menos un caracteres y su extensión tiene que ser s.

9.2 Herramientas utilizadas

A continuación se presentan las herramientas utilizadas para el desarrollo de la aplicación r3000.

9.2.1 Ubuntu

Ubuntu es la distribución Linux que se ha usado para realizar el grueso del desarrollo del sistema. Se eligió una plataforma Linux por las ventajas que ofrecía, como eran:

- Últimas versiones de los programas necesarios para el desarrollo de la práctica
- Sistema libre y gratuito
- Entorno amigable y potente para el desarrollo de aplicaciones tales como compiladores, IDEs, editores perfectamente integrados en el sistema.

La elección de Ubuntu frente a otras distribuciones Linux se debió a que ofrece la robustez de un sistema basado en Debian, pero sin renunciar a las últimas versiones de Kernel y aplicaciones (aporta inseguridad pero muchas características). Además el sistema Ubuntu es en general un sistema fácil y visualmente atractivo.

La parte final del desarrollo y la preparación final del proyecto ha sido realizada con Ubuntu 9.04 versión de 32 bits. El comienzo del proyecto fue realizado con Ubuntu 8.10.

9.2.2 GCC

GCC (siglas de Colección de Compiladores GNU) es un compilador del lenguaje C (entre otros muchos) que nos permite transformar código de alto nivel, en el caso actual de C, a código máquina de procesador sobre el que está corriendo el sistema (i386 en general).

Se eligió este compilador por varios motivos:

- Es software libre
- Está disponible de forma nativa (o por repositorios) en todas las distribuciones Linux

- Se adapta a la perfección al estándar ANSI C
- Es eficiente y versátil (hasta hace poco el único capaz de compilar el núcleo de Linux)

El código fuente de la presenta práctica está escrito en el lenguaje C y es susceptible de ser compilado por GCC o por cualquier otro compilador, pero en el código se hacen llamadas y referencias explícitas a llamadas al sistema de Linux (estándar POSIX) por lo que es necesario compilar dicho código en un sistema que reconozca POSIX (como Linux por ejemplo) para que sea compilado de forma adecuada.

Se ha usado para la compilación final del proyecto la versión 4.3.

9.2.4 Bison

Bison es un programa del proyecto GNU que sirve como generador de analizadores sintácticos (sistema capaz de reconocer palabras de una gramática).

El sistema trabaja como generador de analizadores de tipo LALR genéricos a partir de una gramática en BNF. Está perfectamente integrado con C hasta el punto que permite introducir código C dentro de su sintaxis y la traducción de la gramática (su LALR) está disponible en código C (que podrá ser compilado a código máquina posteriormente).

Bison a su vez trabaja en estrecha colaboración con FLEX, que es el encargado de dar soporte para el reconocimiento de los tokens.

Bison es el programa actual utilizado compatible con YACC, que ya está en desuso.

La versión de Bison usada para la compilación del compilador fue la 2.4.

9.2.5 Flex

Flex es la alternativa libre del proyecto GNU al analizador LEX. FLEX es un analizador léxico, lo que supone que trabaja a nivel tokens o lo que es lo mismo partes que componen lo que se denomina “palabra” de una gramática.

Trabaja en estrecha colaboración con Bison, de modo que Flex lee el fichero de entrada byte a byte y le pasa a Bison un identificador diciéndole el tipo de token que ha encontrado en base a expresiones regulares y literales.

Esos tokens, son los que conforman las reglas BNF escritas en Bison con las cuales se podrá recorrer la gramática.

La versión final del Flex usada para la compilación final del desarrollo fue la 2.5.35.

9.2.6 Gedit

Gedit no s más que un editor muy simple del sistema Linux basado en escritorios Gnome. Gedit ha sido usado para escribir el código para la escritura del código de la práctica y ha sido elegido por varios motivos:

- Es muy sencillo y consume muy pocos recursos
- Tiene una perfecta integración con el escritorio Gnome
- Admite resaltado de sintaxis
- Admite balanceo de paréntesis, corchetes y llaves
- Tiene la opción de sangrado automático
- Muestra las líneas
- Y dispone de todas las opciones usuales de edición de un editor sencillo

9.2.7 Office 2007

Office 2007 es la última versión del paquete ofimático de Microsoft que ha sido usado para escribir esta memoria. En concreto se han usado los programas Word (como editor de textos) y Visio (como editor de diagramas).

Se eligió Office 2007 por ser el mejor paquete ofimático disponible, ya que permite obtener los mejores resultados y máximos niveles de personalización.

Esta versión requirió usar máquinas con Microsoft Windows instalados. En general no tiene importancia determinar más sobre Windows ya que no tiene nada que ver con el desarrollo del compilador, sólo fue el sistema operativo donde corría el Office 2007 en el que se documentó el compilador.

9.2.8 MARS

Es el emulador del ensamblador del MIPS R3000 escogido para ejecutar el código de ensamblador generado. Se ha elegido por ser más visible, más accesible y con más posibilidades que el tradicional SPIM. Además está implementado en JAVA y no requiere instalación (pero si una JVM superior a la 1.4). En general en Linux el simulador responde satisfactoriamente en todos los casos (en Windows no se ha comprobado). El simulador tiene algunas pequeñas diferencias en cuanto a semántica con Spim:

- No admite algunas de sus pseudoinstrucciones (como li.s, que se usa para la carga inmediata de valores flotantes).
- Las variables del segmento de datos no necesitan ser inicializadas (aunque por defecto si se inicializarán a 0).
- No se introduce automáticamente un jal main, esto debe hacerse manual si el programa no comienza por el main (se hace por defecto en todos los programas).

La versión de MARS usada para la comprobación de los códigos ensamblador generados ha sido la 1.6.

10 Prueba realizadas

Para la comprobación del correcto funcionamiento de la práctica se diseñó por un lado una serie de pruebas que abarcasen, en general, todos los aspectos tratados del compilador y por otro, que tuvieran en cuenta todos los posibles errores controlados del compilador (como errores sintácticos o semánticos). La descripción de las pruebas así como los valores y resultados con los que fueron probadas se exponen a continuación.

10.1 Pruebas de funcionamiento

Aquí se van a describir las pruebas que abarcan la funcionalidad en sí del compilador. Para cada prueba se hará una descripción genérica de qué contiene el código C en cuestión y una tabla que recogerá las entradas suministradas y los datos

obtenidos, que certifican el adecuado funcionamiento del código ensamblador generado.

Prueba 1: Año Bisiesto

Nombre del fichero	anno_bisiesto.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Pequeños cálculos • Condiciones • Salida

Tabla 22 - Descripción de la prueba 1

Este programa básicamente consiste en la lectura de un número por teclado y determinar por medio de una condición relativamente importante (tiene AND y OR y balanceo de paréntesis) si el número (año) suministrado es o no bisiesto mostrando un mensaje con el resultado.

Parámetros suministrados	Salida Esperada	Salida Obtenida
1900	No bisiesto	No bisiesto
1999	No bisiesto	No bisiesto
2000	Bisiesto	Bisiesto
2001	No bisiesto	Bisiesto

Tabla 23 - Experimentos realizados sobre la prueba 1

Prueba 2: Calculadora

Nombre del fichero	calculadora.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Cálculos • Salida • Condiciones sencillas • Números flotantes

Tabla 24 - Descripción de la prueba 2

Este programa es una calculadora sencilla que encapsula en funciones las principales operaciones flotantes como son la suma, la división, la multiplicación y el módulo, aunque éste último sólo para enteros.

Parámetros suministrados	Salida Esperada	Salida Obtenida
--------------------------	-----------------	-----------------

1, 1.5, 2.5	4.0	4.0
2, 1.25, 2.5	3.125	3.125
3, 4.0, 1.25	3.2	3.2
4, 100, 33	1	1

Tabla 25 - Experimentos realizados sobre la prueba 2

Prueba 3: Calculador de Matrices

Nombre del fichero	calculadora.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Cálculos complejos • Salida • Condiciones sencillas • Vectores bidimensionales • Bucles • Profundidad (anidamiento de bucles)

Tabla 26 - Descripción de la prueba 3

Este programa es una calculadora para operar con matrices de 2x2 de números enteros. Permite el uso de la suma de matrices, la multiplicación de una matriz por un escalar y finalmente el determinante de la matriz, siendo ésta última la operación más significativa ya que supone realizar una operación matemática compleja en una sola instrucción C.

Este programa además tiene bucles anidados para mostrar las matrices resultantes y para la lectura, siendo éste el más importante ya que además de 2 fors, anida un if-else en su interior, dentro del cual se realiza lectura de número directamente sobre la posición de la matriz.

Parámetros suministrados	Salida Esperada	Salida Obtenida
1, 2, 1, 1, 2	-3	-3
2, 1, 1, 2, 2, 3, 1, 2, 4	4, 2, 4, 6	4, 2, 4, 6
3, 1, 2, 3, 4, 5	5, 10, 15, 20	5, 10, 15, 20
4	No operación	Opción seleccionada no válida

Tabla 27 - Experimentos realizados sobre la prueba 3

Prueba 4: Factorial Iterativo

Nombre del fichero	factorial_iterativo.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Cálculos • Salida • Condiciones sencillas • Bucles

Tabla 28 - Descripción de la prueba 4

Es un programa sencillo enfocado al uso de bucles dentro de una función para cálculos numéricos que de otro modo serían muy complejos. Más concretamente es el cálculo del factorial de un número (menor que 14 para evitar la pérdida de precisión por desbordamiento).

Su inclusión era más una comparativa que una necesidad de prueba (aunque siempre es adecuado probar cuanto más mejor). La comparativa era observar las diferencias en ensamblador entre un código iterativo como este factorial y su versión recursiva.

Parámetros suministrados	Salida Esperada	Salida Obtenida
0	1	1
1	1	1
5	120	120
14	87178291200	El factorial de números mayores a 13 desborda 32 bits.

Tabla 29 - Experimentos realizados sobre la prueba 4

Prueba 5: Factorial Recursivo

Nombre del fichero	factorial_recursivo.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Cálculos • Salida • Condiciones sencillas • Recursividad/Llamadas encadenadas

Tabla 30 - Descripción de la prueba 5

Es la versión recursiva del programa anterior, teniendo exactamente el mismo formato, salvo que la función que calcula el factorial en vez de ser iterativa, es recursiva

Parámetros suministrados	Salida Esperada	Salida Obtenida
0	1	1
4	24	24
6	720	720
14	87178291200	El factorial de números mayores a 13 desborda 32 bits.

Tabla 31 - Experimentos realizados sobre la prueba 5

Prueba 6: Media Aritmética

Nombre del fichero	media_aritmetica.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Cálculos • Salida • Condiciones sencillas • Bucles

Tabla 32 - Descripción de la prueba 6

Es un programa que lee elemento por teclado y los almacena en sus correspondientes posiciones por medio de un for. Finalmente calcula la media aritmética del contenido de dicho vector y lo muestra por pantalla. El vector es por definición de 5 posiciones.

Parámetros suministrados	Salida Esperada	Salida Obtenida
1, 2, 3, 4, 5	3	3
2, 4, 6, 8, 10	6	6
1, 3, 1, 3, 1	1.8	1.8
5, 5, 5, 5, 5	5	5

Tabla 33 - Experimentos realizados sobre la prueba 6

Prueba 7: Raíz cuadrada

Nombre del fichero	raiz_cuadrada.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Cálculos • Salida • Condiciones sencillas • Bucles • Números flotantes

Tabla 34 - Descripción de la prueba 7

Este es un programa similar en ejecución al factorial, en el que se realizan una serie de cálculos iterativos para obtener un resultado matemático complejo. En este caso es el cálculo de la raíz cuadrada de un número flotante. El cálculo se realiza por

medio de varias iteraciones de distinto bucles aproximando la raíz cuadrado de un número dado.

Su principal interés era investigar cómo trabajar funciones de librería como sqrt y ver que muchas veces una sola llamada de una API, puede suponer cientos o miles de instrucciones a ejecutar.

Parámetros suministrados	Salida Esperada	Salida Obtenida
2.0	1.4142135	1.4142135
3.0	1.7320508	1.7320509
4.0	2.0	2.0
5.0	2.236067	2.236068

Tabla 35 - Experimentos realizados sobre la prueba 7

Prueba 8: Torres de Hanoi

Nombre del fichero	torres_hanoi.c
Funcionalidad a probar	<ul style="list-style-type: none"> • Entrada • Invocación de funciones • Salida • Condiciones sencillas • Recursividad/Llamadas encadenadas • Varias llamadas Recursivas por función

Tabla 36 - Descripción de la prueba 8

Este es el otro ejemplo recursivo planteado en la batería. En el fundamentalmente se muestra la solución recursiva de las torres de Hanoi para un número N de discos introducidos por teclado. Tiene la peculiaridad de encadenar varias llamadas consecutivas, por lo que era un reto el control minucioso de la pila para que un ejemplo como éste funcionase de forma correcta.

Parámetros suministrados	Salida Esperada	Salida Obtenida
1	A-C	A-C
2	A-B, A-C, B-C	A-B, A-C, B-C

Parámetros suministrados	Salida Esperada	Salida Obtenida
3	A-C, A-B, C-B, A-C, B-A, B-C, A-C	A-C, A-B, C-B, A-C, B-A, B-C, A-C
4	A-B, A-C, B-C, A-B, C-A, C-B, A-B, A-C, B-C, B-A, C-A, B-C, A-B, A-C, B-C	A-B, A-C, B-C, A-B, C-A, C-B, A-B, A-C, B-C, B-A, C-A, B-C, A-B, A-C, B-C

Tabla 37 - Experimentos realizados sobre la prueba 8

10.2 Prueba de Error

Estas pruebas tienen la finalidad de determinar que el compilador detecta de forma adecuada los errores léxico-sintácticos planteados en el programa a compilar, así como los errores semánticos (estos son los especialmente importantes, ya que los otros los detecta el propio Bison).

Para ello la siguiente tabla expresa el nombre del ejemplo con errores y qué tipo de error pretende mostrar para ver si el compilador es capaz de detectarlo de forma correcta.

FICHERO	TIPO ERROR	SALIDA
esem0.c	Semántico	Línea 8: Párida de precisión numérica.
esem1.c	Semántico	Línea 7: Variable e4 no definida.
esem2.c	Semántico	Línea 5: Función inexistente.
esem3.c	Semántico	Línea 12: Número excesivo de argumentos.
esem4.c	Semántico	Línea 5: Variable array número de dimensiones incorrecto
esem5.c	Semántico	Línea 5: Falta el retorno de la función suma.
esin0.c	Sintáctico	Error en línea 5 en token e1
esin1.c	Sintáctico	Línea 7: Función Inexistente
esin2.c	Sintáctico	Error en línea 3 en token e3
esin3.c	Sintáctico	Error en línea 10 en token
esin4.c	Sintáctico	Error en línea 5 en token 1.5

Tabla 38 - Experimento realizados con errores semánticos

11 Programación en C-LITE

C lite tiene numerosas diferencias con C estándar, pero también muchas similitudes. Muchas de las diferencias son o bien gramática no terminada (como serían punteros y otras cosas no tratadas) y otras serían simplificaciones como el caso de las funciones de entrada y salida.

En este apartado se pretende orientar un poco sobre cómo crear programa en C lite que sean admitidos por el compilador R3000.

Las principales características a tener en cuenta son:

C Lite no permite pasar expresiones matemáticas dentro de un if, deben ser almacenadas previamente en una variable, ya que los ifs de C lite admiten sólo operadores lógicos (AND y OR), comparadores matemáticos (<, >, !=, ==, >=, <=), constantes numéricas (enteras o flotantes) e identificadores de variables o posiciones de arrays.

C Lite no permite pasar expresiones matemáticas como argumentos a funciones, al igual que en el caso de los ifs, estos valores deben ser previamente almacenados en una variable y debe ser esa variable la que se pase como argumento de la función.

Al igual que en los dos apartados anteriores, C Lite no permite sentencias return de funciones con expresiones o constantes, C Lite sólo permite el retorno de variables.

- C Lite no permite el uso de estructuras.
- C Lite no permite el uso de punteros.
- C Lite no permite el uso de uniones.
- C Lite no permite el uso de typedef.
- C lite no permite ficheros disgregados ni variables extern.
- C Lite no permite el tipo long
- C Lite no permite el tipo double
- C Lite no permite arrays como parámetro de funciones.
- C Lite permite sentencias return vacías en funciones void.
- C Lite permite la declaración de arrays multidimensionales (sin límite de dimensiones).

- C Lite sólo permite la declaración de las variables o bien al comienzo tras las constantes o bien al inicio del main o al inicio de las funciones. Por tanto no se permite definición de variables en medio del código, todas han de ir al comienzo del bloque.
- C Lite obliga a que primero se definan las funciones locales y luego el main, sólo en ese orden.
- C Lite permite la definición de constantes al inicio del programa mediante la sentencia `#define id valor`.
- C Lite permite los operadores unarios `++` y `--`.
- C Lite permite las forma resumidas de igualación matemáticas tales como `+=`, `-=`, `*=`, `/=`.
- C Lite permite en los bucles for cualquier expresión matemática de incremento sin igualación, como por ejemplo `i+1`, `i*5`, `i-2`, `i++`, `i--`.
- C Lite permite tantos paréntesis como se quiera en una operación matemática y tantos elementos como se deseen, no importa el tamaño.
- C Lite permite tantos paréntesis como se quiera en una operación lógica condicional y tantos elementos como se deseen, no importa el número de condiciones.
- C Lite permite el uso de datos contenidos en arrays en operaciones matemáticas.
- C Lite obliga a que el inicio del programa sea el método `main ()`.
- C Lite permite las llamadas encadenadas de funciones, como que, por ejemplo, la función `a` llame a la `b` y esta a su vez a la `c`.
- C Lite permite la recursividad.
- C Lite permite la entrada de un parámetro por invocación por medio del uso de `scanf(id)`.
- C Lite permite la salida de elementos múltiples simultáneos, siempre que estos sean identificadores de variables o constantes del siguiente modo `print (id id1 id2...)` o bien `println (id id1 id2...)`. `Print` imprime en la misma línea, `println` imprime al final el salto de línea.

- C Lite permite la definición de constantes textuales (como el `char *` de C o el `String` de Java) del siguiente modo `#define id "texto"`.

12 Contenido del DVD

En este apartado se realiza una descripción de los elementos incluidos en el DVD adjunto con este documento.

El DVD contiene en la carpeta Ubuntu una máquina virtual en VMWare de la versión 9.04 de Ubuntu en su variante desktop de 32 bits. Esta máquina virtual contiene en el escritorio todos los directorios que se describen a continuación, para poder compilar, ejecutar y probar los ejemplos. Esta máquina tiene instalados los paquetes Ubuntu/Debian:

- build-essential, necesario para compilar programas C.
- flex, necesario para ejecutar el analizador léxico.
- bison, necesario para ejecutar el generador de analizadores sintácticos
- java jre 1.6, necesaria para poder ejecutar el MARS.
- El usuario y el password para acceder a la máquina es:
- Usuario: grupo12
- Pass: grupo12pl2

En la carpeta VMPlayer, contiene versiones del programa VMPlayer necesario para ejecutar máquinas virtuales de VMWare en sus versiones Windows y Linux.

En la carpeta Ejemplos, hay dos subdirectorios, el primero, llamado Ejemplo C Lite, contiene todos los ejemplos de ejecuciones correctas tratados en la memoria. La carpeta Ejemplo erróneos, contiene todos los ejemplos con errores que también han sido tratados en la presente memoria.

La carpeta Código fuente, contiene todos los ficheros necesarios para compilar el programa (todos los ficheros fuente y el Makefile).

La carpeta MARS, contiene el simulador MARS 1.6 para comprobar el código ensamblador generado por el compilador. Esta versión es universal y puede ser ejecutada en cualquier Sistema Operativo con una Máquina Virtual de Java 1.4 o superior.

En el directorio raíz del DVD se encuentran una copia en PDF de la presente memoria y un fichero denominado autorex.txt con los autores de la práctica.

13 Conclusiones

La primera conclusión que se puede sacar tras la realización de la práctica, es la gran complejidad que entraña realizar un compilador (aunque sólo sea hasta generación de ensamblador y simplificando algunas de sus fases).

En esta práctica se ha procurado hacer especial hincapié en la generación del ensamblador, realizando un especial esfuerzo en que el código generado, en la medida de lo posible, fuera lo más óptimo. Esto supuso la consulta de bastante bibliografía, tanto de libros de compiladores, como de libros de la arquitectura del procesador MIPS o libros de programación de ensamblador directamente.

Se consideró la generación y optimización de código la parte más importante de la práctica, ya que consideramos que si la finalidad de un compilador es obtener códigos ejecutables por una máquina, se espera que dichos códigos sean eficientes o de lo contrario la herramienta sería completamente inútil. Este especial esfuerzo en esta faceta de la compilación imposibilitó profundizar a la misma profundidad en todos los niveles, de modo que otros apartados como la recuperación de errores fue tratada de forma más simple (se detecta el primer error y se para el proceso de compilación).

Podría haberse invertido más tiempo y esfuerzo en tareas propiamente sintácticas, pero se consideró que para esta práctica era más interesante, llamativo e incluso docente centrarse más en la parte “arquitectónica del compilador” que no tanto en la interfaz hombre-compilador, aunque este apartado sea muy importante, sobre todo para agilizar los procesos de compilación evitando lo más posible la repetición de errores por parte del programador.

No obstante, aunque no se haya centrado el desarrollo en esa faceta del compilador, se han hechos grandes esfuerzos en el estudio, comprensión y aprendizaje de las herramientas de análisis léxico y sintácticos (Flex y Bison), atendiendo a cosas como la interacción y comunicación entre ambas, procurar que el análisis léxico evitase dedicar tiempo a todo aquello que no fuera código (caso de los comentarios), procurar construir una gramática que fuera ágil para Bison (Yacc), estudiar cómo sacarle el máximo partido al sistema pro medio de las facilidades que ofrecían ambas herramientas, etc.

Por todo ello, se puede decir que el desarrollo de la presente práctica ha sido determinante para aprender a manejar herramientas de generación de compiladores, que pueden ser utilizadas para muchos fines y no sólo para la compilación (traducción, interpretación de cadenas, automatización de procesos, etc.).

La realización de la práctica también nos ha permitido profundizar mucho en el conocimiento de la arquitectura de los procesadores, más concretamente en el MIPS R3000, aprendiendo su funcionamiento, estructura, usos, etc. De modo que podemos decir que para poder programar un lenguaje de bajo nivel de la forma más correcta posible, es imprescindible el conocimiento de la arquitectura del procesador, para así poder sacarle el máximo partido a los programas.

Todo lo mencionado con la arquitectura está estrechamente ligado con el lenguaje (o su abstracción humana para ser más correcto) de la máquina, el ensamblador (realmente es el binario, ya que el ensamblador es una interpretación humana entendible). El conocimiento de la estructura de un procesador y el correcto manejo de su ensamblador permite sacarle el máximo partido a las máquinas, no sólo para el caso de la realización de un compilador, sino para cualquier ámbito de la programación en general. Por esto, se realizó en especial esfuerzo en procurar generar un código optimizado, para ver cómo a bajo nivel, pequeñas decisiones marcan al diferencia entre un buen programa eficiente o uno malo.

Para la implementación de la práctica se optó por el lenguaje C (haciendo uso de rutinas Posix por usarse Linux como sistema operativo) debido a la potencia y calidad del mismo. En una práctica como la presente, de pequeña envergadura en comparación con sistemas reales, no se llegaría a apreciar la diferencia, pero sería incomparable la eficiencia y potencia de la compilación con un ambiente C (por ejemplo compilado con GCC) que no un compilador que trabajase en un ambiente 100% Java, que requiere de bytecode e interpretación por la máquina virtual. Por tanto se consideró que el uso de C como lenguaje para desarrollar el compilador le dotaba de mucho más realismo al sistema en su conjunto.

Hay que decir, que una cosa es la realidad de las ventajas de C para el desarrollo de este tipo de entornos (aunque como se ha dicho en la presente práctica no habría

diferencias reales entre C y otros lenguaje como Java) y otra bien diferente los problemas que ocasiona su desarrollo. C no está orientado a objetos y carece de polimorfismo, por tanto, intentar emular esas actuaciones, por medio de funciones con punteros a void y en general punteros a funciones, es una técnica complacida y difícil de trabajar y depurar.

Para concluir, consideramos que esta práctica nos ha sido muy útil en primer lugar para el conocimiento de los procesos, fases y actuaciones que hay que llevar a cabo para construir un compilador. Pero además nos ha servido para profundizar en conocimientos de arquitectura de procesadores, programación en ensamblador y adquisición de conocimientos, en general, aplicables a muchos ámbitos de la informática más allá de la generación de compiladores en sí.

14 Anexos

14.1 Código fuente analizador léxico

```
%{
#include <stdio.h>
#include "sintactico.tab.h"
#include "control.h"
%}

%x comentario
%x comentarioLinea

digito                [0-9]
letra                 [a-zA-Z]
alpha                 [a-zA-Z0-9_]

%%

-?{digito}+          {yylval.entero = atol(yytext); return(ENTERO);}
-?{digito}+".{digito}+ {yylval.real  = atof(yytext); return(REAL);}

"float"              {yylval.entero = 3; return(FLOAT);}
"int"                {yylval.entero = 2; return(INT);}
"short"              {yylval.entero = 1; return(SHORT);}
"char"               {yylval.entero = 0; return(CHAR);}
"void"               {yylval.entero = -2; return(VOID);}
"if"                  {activarFlag(8);activarFlag(11); return(IF);}
"else"               return(ELSE);
"for"                 {activarFlag(10);activarFlag(11); return(FOR);}
"while"               {activarFlag(9);activarFlag(11);
return(WHILE);}
"main"               return(MAIN);
"return"             return(RETURN);
```

"#define"	return(DEFINE);
"print"	return(ESCRIBIR);
"println"	return(ESCRIBIRLN);
"scanf"	return(LEER);
"+"	return(MAS);
"++"	return(MASUNARIO);
"--"	return(MENOSUNARIO);
"_"	return(MENOS);
"*"	return(MULTIPLICACION);
"/"	return(DIVISION);
"%"	return(MODULO);
"+="	return(MASIGUAL);
"-="	return(MENOSIGUAL);
"*="	return(MULIGUAL);
"/="	return(DIVIGUAL);
"=="	return(COMPARACION);
"<="	return(MEI);
">="	return(MAI);
"!="	return(DISTINTO);
"<"	return(MENOR);
">"	return(MAYOR);
"="	return(ASIGNACION);
"&"	return(AND);
","	return(PUNTOYCOMA);
" "	return(OR);
","	return(COMA);
"("	return(PARENTESISIZ);
")"	return(PARENTESISDE);
"["	return(CORCHETEIZ);
"]"	return(CORCHETEDE);
":"	return(DOSPUNTOS);
"{"	return(INICIO);

```

"}"                return(FIN);
"\\""             return(COMILLAS);
""                 return(COMILLASIMPLE);

{letra}?\'(\\.|[^\'])*\'    {strcpy(yyval.cadena,yytext);return(CARACTER);}
{letra}?\"(\\.|[^\"])*\"    {strcpy(yyval.cadena,yytext);return(TEXTO);}
{letra}{alpha}*        {

    sprintf(yyval.general.nombre,"ID_%s",yytext);

    return(ID);

}

[ \t\v\f]

[ \n]              {incrementarLinea();}

"//"              BEGIN(comentarioLinea);
<comentarioLinea>[^\n]*      /* se come cualquier cosa que no sea un
'\n' */

<comentarioLinea>\n          {incrementarLinea();BEGIN(INITIAL);}

"/*"              BEGIN(comentario);
<comentario>[^\n]*          /* se come cualquier cosa que no sea un '*' */
<comentario>"*" + [^\n]*     /* se come '*'s que no continuen con '/' */
<comentario>\n              {incrementarLinea();}
<comentario>"*" + "/"       BEGIN(INITIAL);

.

%%

int yywrap (void)
{
    return 1;
}

```

```
}
```

```
int yyerror(char *s)
```

```
{
```

```
    printf("Se ha producido un error sintactivo en la linea %d en el token %s\n",  
valorFlag(0), yytext);
```

```
    return -1;
```

```
}
```

14.2 Código fuente analizador sintáctico

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include "control.h"  
#include "tSimbolos.h"  
#include "arbol.h"  
//#include "pila.h"  
#include "pilaArbol.h"  
#include "pilaEtiquetas.h"  
#include "IAuxiliar.h"  
#include "transformador.h"  
#include "intermedio.h"  
#include "fichero.h"  
  
#define YYDEBUG 1  
#define BUFEMS 4096  
#define DIMENSIONES 4  
  
tSimbolos TS = NULL;           //Variable de almacenamiento de la tabla de  
simbolos.  
IAuxiliar IAux = NULL;  
IAuxiliar IEx = NULL;  
PilaE pEti    = NULL;  
PilaE pFor    = NULL;  
arbol aAux    = NULL;  
Pila pEx      = NULL;  
  
int retornoFuncion      = 0;  
int numArgumento        = 1;  
int numeroArgumentos;    //Variable auxiliar  
int numParametros;       //Variable auxiliar
```

```
int auxiliarArray;          //Variable auxiliar de almacenamiento de valor de
dimensiones.
```

```
int auxiliarTipo;          //Variable auxiliar sde almacenamiento de los
tipos de funciones y variables, reutilización de valores.
```

```
int auxiliarOffSet;
```

```
char buffer[256];          //Utilización para el almacenamiento de nombres
o generación de cadenas con sprintf.
```

```
char funcionActual[256];    //Variable auxiliar para la comprobación de la función que
se trata actualmente.
```

```
char funcionLlamada[256];
```

```
char bufferCorto[32];      //Utilización para el almacenamiento de nombres cortos
o generción de cadenas con sprintf cortas.
```

```
char registro[5];          //Utilización para el almancenamiento de los
nombre de los registros a utilizar.
```

```
int etiqueta = 0;
```

```
int identificador = 0;
```

```
%}
```

```
%union
```

```
{
```

```
    char cadena [4096];
```

```
    char cadenaCorta [32];
```

```
    int entero;
```

```
    float real;
```

```
    struct {
```

```
        int entero;
```

```
        float real;
```

```
        char nombre [4096];
```

```
        int tipo; //-1 Texto(STRING) -2 void 0:char 1:entero corto 2:entero
```

```
3:float
```

```
        char dimensiones[32];
```

```

    } general;

    struct {
        int entero;
        float real;
        int tipo; //1:entero 3:float
    } numeros;

    int dimArrays[4];
}

```

%start programa

```

%token <cadena> TEXTO
%token <cadena> CHARACTER
%token <general> ID
%token <entero> ENTERO
%token <real> REAL
%token <entero> INT
%token <entero> CHAR
%token <entero> FLOAT
%token <entero> LONG
%token <entero> SHORT
%token <entero> VOID
%token IF
%token ELSE
%token FOR
%token WHILE
%token MAIN
%token RETURN
%token DEFINE
%token ESCRIBIR

```

%token ESCRIBIRLN
 %token LEER
 %token MAS
 %token MASUNARIO
 %token MENOS
 %token MENOSUNARIO
 %token MULTIPLICACION
 %token DIVISION
 %token MODULO
 %token MASIGUAL
 %token MENOSIGUAL
 %token MULIGUAL
 %token DIVIGUAL
 %token COMPARACION
 %token MEI
 %token MAI
 %token DISTINTO
 %token MENOR
 %token MAYOR
 %token ASIGNACION
 %token AND
 %token PUNTOYCOMA
 %token OR
 %token COMA
 %token PARENTESISIZ
 %token PARENTESISDE
 %token CORCHETEIZ
 %token CORCHETEDE
 %token DOSPUNTOS
 %token INICIO
 %token FIN
 %token NOT

%token COMILLAS

%token COMILLASIMPLE

%type <cadenaCorta> dimension

%type <entero> tipo_dato

%type <entero> elemento_base

%type <entero> base_expresion_logica

%type <entero> elemento_expresion_matematica

%type <entero> elemento_expresion_logica

%type <entero> expresion_matematica

%type <entero> expresion_logica

%type <entero> sentencia_especial

%type <entero> sentencia_llamada

%type <entero> elemento_argumento

%type <general> identificador_base

%type <general> valor_dimension

%type <general> dimension_operacion

%type <general> elemento_base_parametro

%type <general> elemento_constante

%type <general> identificador_operacion

%type <numeros> numero

%type <general> nombre_funcion

%type <general> elemento_variable

%nonassoc LOWER_THAN_ELSE

%nonassoc ELSE

%%

programa

: constantes elementos programa_principal

;

/*****

ZONA DE DECLARACION DE CONSTANTES

*****/

constantes

: constante constantes

| /*LAMBDA*/

;

constante

: DEFINE ID elemento_constante

;

elemento_constante

: numero

| TEXTO

;

/*****

INICIO ZONA DE DECLARACION DE FUNCIONES

Y VARIABLES GLOBALES

*****/

elementos

: elementos elemento_vf

| /*LAMBDA*/

;

elemento_vf

: tipo_dato identificador_base estructura_variables PUNTOYCOMA

```
| cabecera_funcion estructura_funcion
;
```

estructura_variables

```
: estructura_variables COMA identificador_base
| /*LAMBDA*/
;
```

cabecera_funcion

```
: tipo_dato ID
;
```

estructura_funcion

```
: parametros cuerpo
;
```

tipo_dato

```
: INT
| CHAR
| SHORT
| FLOAT
| VOID
;
```

dimension

```
: CORCHETEIZ valor_dimension CORCHETEDE dimension
| CORCHETEIZ valor_dimension CORCHETEDE
;
```

valor_dimension

```
: ENTERO
| ID //Se utiliza una constante para definir la dimensión
```

;

dimension_operacion

: CORCHETEIZ valor_dimension CORCHETEDE dimension_operacion
| CORCHETEIZ valor_dimension CORCHETEDE
;

parametros

: PARENTESISIZ lista_parametros PARENTESISDE
| PARENTESISIZ PARENTESISDE
;

lista_parametros

: elemento_parametro
| elemento_parametro lista_parametros
;

elemento_parametro

: tipo_dato elemento_base_parametro COMA
| tipo_dato elemento_base_parametro
;

elemento_base_parametro

: ID
| ID CORCHETEIZ CORCHETEDE
;

cabecera_principal

: tipo_dato MAIN PARENTESISIZ PARENTESISDE
;

programa_principal

```

: cabecera_principal cuerpo
;

```

```

/*****
FIN ZONA DE DECLARACION DE FUNCIONES
*****/

```

```

identificador_base
: ID
| ID dimension
;

```

```

identificador_operacion
: ID
| ID dimension_operacion
;

```

```

numero
: ENTERO
| REAL
;

```

```

operador_asignacion
: ASIGNACION
| MASIGUAL
| MENOSIGUAL
| MULIGUAL
| DIVIGUAL
;

```

```

operador_matematico
: MAS

```

```

    | MENOS
    | MULTIPLICACION
    | DIVISION
    | MODULO
;

```

```

operador_comparacion
: COMPARACION
    | MEI
    | MAI
    | DISTINTO
    | MAYOR
    | MENOR
;

```

```

operador_logico
: AND
    | OR
;

```

```

operador_unario
: MASUNARIO
    | MENOSUNARIO
;

```

```

/*****
DEFINICION EXPRESIONES MATEMATICA
*****/

```

```

parentesis_izq_expresion
: PARENTESISIZ
;

```

parentesis_der_expresion

: PARENTESISDE

;

negacion

: NOT

;

elemento_base

: identificador_operacion

| numero

| TEXTO

;

expresion_matematica

: elemento_expresion_matematica operador_matematico
expresion_matematica

| elemento_expresion_matematica

;

elemento_expresion_matematica

: elemento_base

| parentesis_izq_expresion expresion_matematica parentesis_der_expresion

;

operador

: operador_logico

| operador_comparacion

;

expresion_logica

```

: elemento_expresion_logica operador expresion_logica
| elemento_expresion_logica
;

```

```

elemento_expresion_logica
: base_expresion_logica
| parentesis_izq_expresion expresion_logica parentesis_der_expresion
;

```

```

base_expresion_logica
: negacion elemento_base
| elemento_base
;

```

```

fin_expresion
: PARENTESISDE
;

```

```

/*****
DEFINICION EXTRUCTURAS
*****/

```

```

cuerpo
: INICIO variables sentencias FIN
;

```

```

variables
: elemento_variable variables
| /*LAMBDA*/
;

```


elemento_variable

```
: tipo_dato identificador_base identificador_variable PUNTOYCOMA
;
```

identificador_variable

```
: COMA identificador_base identificador_variable
| /*LAMBDA*/
;
```

sentencias

```
: sentencia sentencias
| sentencia
;
```

sentencia

```
: sentencia_asignacion
| sentencia_if
| sentencia_while
| sentencia_for
| sentencia_retorno
| sentencia_entrada_salida
| sentencia_llamada PUNTOYCOMA
| sentencia_especial PUNTOYCOMA
;
```

cuerpo_sentencia

```
: INICIO sentencias FIN
| sentencia
;
```

/*****

SENTENCIA TIPO: ESPECIAL

*****/

sentencia_especial

: identificador_operacion operador_unario

;

/*****

SENTENCIA TIPO: ASIGNACION

*****/

sentencia_asignacion

: identificador_operacion operador_asignacion expresion_matematica
PUNTOYCOMA

| identificador_operacion operador_asignacion sentencia_llamada
PUNTOYCOMA

;

/*****

SENTENCIA TIPO: CONDICION IF

*****/

elemento_else

: ELSE

;

sentencia_if

: IF PARENTESISIZ expresion_logica fin_expresion cuerpo_sentencia %prec
LOWER_THAN_ELSE

| IF PARENTESISIZ expresion_logica fin_expresion cuerpo_sentencia
elemento_else cuerpo_sentencia

;

/*****

SENTENCIA TIPO: BUCLE FOR

*****/

cabecera_for_asignacion

: ID operador_asignacion expresion_matematica PUNTOYCOMA
;

cabecera_for_comparacion

: ID operador_comparacion expresion_logica PUNTOYCOMA
;

cabecera_for_modificacion

: expresion_matematica
| sentencia_especial
;

sentencia_for

: FOR PARENTESISIZ cabecera_for_asignacion cabecera_for_comparacion
cabecera_for_modificacion PARENTESISDE cuerpo_sentencia
;

/*****

SENTENCIA TIPO: BUCLE WHILE

*****/

cabecera_while

: WHILE PARENTESISIZ expresion_logica fin_expresion
;

sentencia_while

: cabecera_while cuerpo_sentencia

```

;

/*****

SENTENCIA TIPO: ENTRADA/SALIDA
*****/

sentencia_entrada_salida
    : sentencia_escritura PUNTOYCOMA
    | sentencia_lectura PUNTOYCOMA
    ;

sentencia_escritura
    : ESCRIBIR PARENTESISIZ elementos_entrada PARENTESISDE
    | ESCRIBIRLN PARENTESISIZ elementos_entrada PARENTESISDE
    ;

elementos_entrada
    : valor_entrada elementos_entrada
    | valor_entrada
    ;

valor_entrada
    : identificador_operacion
    ;

sentencia_lectura
    : LEER PARENTESISIZ identificador_operacion PARENTESISDE
    ;

*****/

```

SENTENCIA TIPO: LLAMADA

*****/

nombre_funcion

: ID

;

sentencia_llamada

: nombre_funcion PARENTESISIZ lista_argumentos PARENTESISDE

;

lista_argumentos

: lista_argumentos elemento_argumentos

| /*lambda*/

;

elemento_argumentos

: COMA elemento_argumento

| elemento_argumento

;

elemento_argumento

: numero

| identificador_operacion

;

/*****

SENTENCIA TIPO: RETORNO

*****/

sentencia_retorno

: RETURN identificador_operacion PUNTOYCOMA

| RETURN PUNTOYCOMA

;